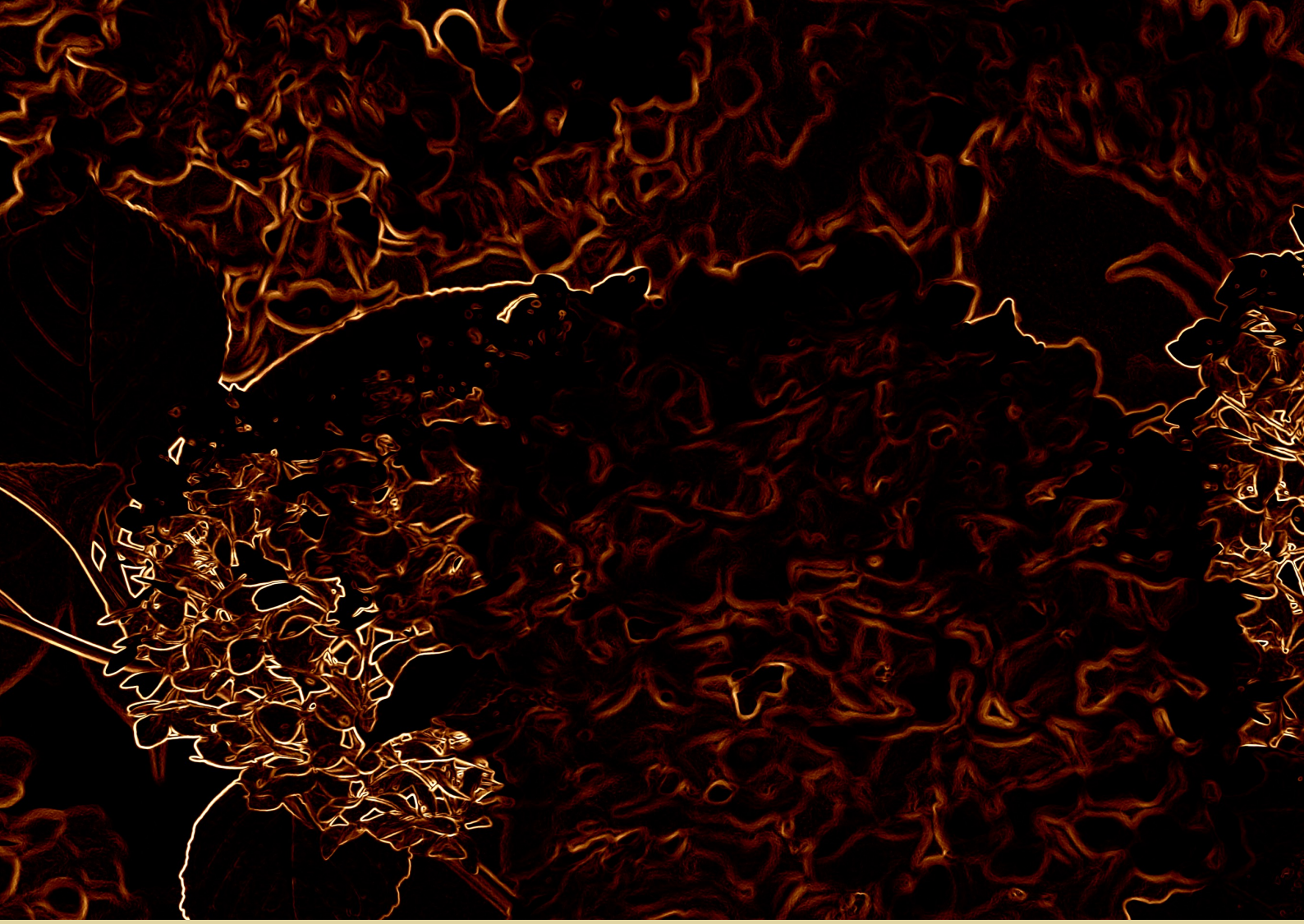

Leonor Isabel de Albuquerque Melo

Self Adaptation in Ant Colony Optimisation

Doctoral thesis submitted to the Doctoral Program in Information Science and Technology,
supervised by Full Professor Ernesto Jorge Fernandes Costa and Assistant Professor Francisco José Baptista Pereira, and presented to the Department of Informatics Engineering
of the Faculty of Sciences and Technology of the University of Coimbra.

August 2018

· U C ·

Universidade de Coimbra

Self Adaptation in Ant Colony Optimisation

A thesis submitted to the University of Coimbra
in partial fulfilment of the requirements for the
Doctoral Program in Information Science and Technology

by

Leonor Isabel de Albuquerque Melo
leonor@dei.uc.pt

Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

Coimbra, August 2018

This dissertation was prepared under the supervision of

Ernesto Jorge Fernandes Costa
Full Professor
of the Department of Informatics Engineering
of the Faculty of Sciences and Technology
of the University of Coimbra

and
Francisco José Baptista Pereira
Assistant Professor
of the Department of Informatics and Systems
of the Polytechnic Institute of Coimbra

To Sofia

Agradecimentos

Sou realmente sortuda, chegado este momento existem tantas pessoas a quem quero agradecer!

Primeiro que tudo, aos meus orientadores, Doutor Ernesto Costa e Doutor Francisco Pereira. Foi uma honra e um prazer desenvolver este trabalho sob a vossa orientação. Obrigada pelo encorajamento e permanente disponibilidade para ajudar.

Aos meus colegas do ISEC, obrigada pelo profissionalismo e inúmeras gargalhadas na sala do café.

Aos ECOS, obrigada pelas discussões interessantes e patuscadas animadas. Um agradecimento especial ao Nuno Lourenço, que sempre encontra um tempinho para ajudar.

À minha família, irmãos, irmã, cunhado e cunhadas: obrigada pelas vossas bizarrias, bom humor, honestidade e generosidade. Aos meus sobrinho e sobrinhas: cada um de vós é perfeito, tal e qual como é, adoro-vos!

Aos meus sogros, obrigada pelo vosso carinho, paciência e compreensão. À minha mãe, a sua independência e otimismo são uma fonte de inspiração: obrigada!

Aos meus compadres, comadres e amigos: obrigada pela vossa amizade, por toda a vossa ajuda, e pelas incontáveis jantaradas, passeatas e bons momentos.

Ao meu moço favorito: obrigada por muito, muito mais do que cabe aqui. Já foram 21, venham mais 210!

À pirralha voadora que faz os meus dias mais brilhantes e o meu coração mais grato. Obrigada por seres quem és. Gosto muito de ti meu amor!

Leonor

Coimbra, Agosto 2018

Resumo

A otimização baseada em colónias de formigas (ACO) é uma metaheurística global vagamente inspirada no comportamento social das formigas. Múltiplas variantes foram propostas ao longo das últimas duas décadas e, durante esse período, têm vindo a ser aplicadas com sucesso na resolução de problemas difíceis de otimização combinatória. Contudo, e apesar da sua relevância na área da otimização, os algoritmos de colónias de formigas possuem alguns inconvenientes conhecidos. Uma limitação importante é a sua sensibilidade à parametrização, de tal modo que, para uma dada situação, diferentes configurações podem obter resultados significativamente diferentes. Além disso, as componentes sôfregas do algoritmo podem facilmente levar à perda de diversidade e à convergência prematura.

Esta dissertação propõe dois novos algoritmos auto-adaptativos baseados em colónias de formigas. Ambas as propostas se baseiam na coexistência de grupos heterogéneos de formigas para a resolução de um mesmo problema de otimização, mas em que cada grupo possui a sua própria estratégia de pesquisa. Além disso a estratégia de pesquisa não é fixa e os algoritmos podem adaptar, de forma autónoma, o seu comportamento às diversas fases da resolução do problema de otimização. A auto-adaptação em tempo real (on-line) tem duas vantagens cruciais: liberta o utilizador da necessidade de definir cuidadosamente as configurações para cada situação de otimização específica, e concede ao algoritmo a capacidade de ajustar o seu comportamento de acordo com a estrutura do espaço de pesquisa.

A primeira contribuição é o MC-Ant, um algoritmo de formigas com várias colónias. Cada colónia consiste num grupo de formigas com as suas próprias configurações e conhecimento adquirido. As diferentes colónias existem em simultâneo e resolvem de forma independente o mesmo problema. Periodicamente são partilhadas soluções de boa qualidade e estratégias de pesquisa. Os resultados obtidos com o NPP mostram que o MC-Ant tem um desempenho superior a abordagens de colónia única, reforçando a relevância da migração para evitar a convergência prematura e permitir uma auto-adaptação eficaz.

O Multi-caste ACS é a segunda contribuição deste trabalho. É uma abordagem ACO multi-estratégica e auto-adaptativa alternativa, concebida de forma a evitar alguns problemas de eficiência apresentados pelo MC-Ant. As formigas são divididas em castas, e cada casta tem o seu próprio q_0, um parâmetro crucial na definição da estratégia de pesquisa. As formigas podem migrar entre castas de acordo com algumas regras simples. Deste modo, permite-se que o algoritmo ajuste a sua estratégia de pesquisa de forma autónoma e alcance um equilíbrio adequado entre a utilização do conhecimento adquirido e a exploração de novas áreas do espaço de pesquisa.

O Multi-caste ACS foi aplicado ao TSP simétrico, tanto à versão estática como à versão dinâmica, periódica e não cíclica do problema. Os resultados confirmam a vantagem da abordagem heterogénea. As variantes clássicas dos ACO são eficazes num subconjunto de cenários de otimização, mas falham completamente em outros. Pelo contrário, abordagens com várias castas são extremamente robustas e são capazes de manter um

desempenho equilibrado em todos os cenários de otimização considerados. Essa robustez é particularmente evidente em ambientes dinâmicos.

Palavras Chave

Auto-adaptação, Multi-população, Otimização baseada em colónia de formigas, Problemas dinâmicos, Robustez

Abstract

ACO is a global metaheuristic loosely inspired by the behaviour of social ants. Several variants were proposed over the past two decades and, throughout this period, they have been successfully applied to solve difficult combinatorial optimisation problems. Notwithstanding its relevance in optimisation, ant colony algorithms have several well-known drawbacks. One important limitation is that they tend to be particularly sensitive to parameterisation and different settings may obtain significant different results on the same situation. Also, they have strong greedy components that can easily lead to the loss of diversity and to premature convergence.

This dissertation proposes two novel self-adaptive ant algorithms. Both of them rely on the coexistence of heterogeneous groups of ants within a single optimisation framework, each set with its own search strategy. Moreover, the search strategy is not fixed and, instead, the algorithms can autonomously adapt their behaviour to the different stages of the optimisation problem being solved. On-line self-adaptation has two crucial advantages: it frees the practitioner from having to carefully define settings for each specific optimisation situation and it grants the algorithm the ability to adjust its behaviour in accordance to the structure of the search landscape.

The first contribution is MC-Ant, a multi-colony ant algorithm. Each colony is defined as a group of ants with its own search settings and acquired knowledge. Different colonies coexist in the algorithm while independently solving a problem. Periodically, good quality solutions migrate and effective search strategies are shared. Results obtained with the NPP show that MC-Ant outperforms single colony approaches, reinforcing the relevance of migration to avoid premature convergence and to allow an effective parameter self-adaptation.

Multi-caste ACS is the second contribution of the work. It is an alternative self-adaptive, multi-strategy ACO approach, designed in such a way as to avoid a few efficiency issues exhibited by MC-Ant. In this framework, ants are divided in castes, and each caste has its own q_0 value, a critical parameter to define the search strategy. Ants can migrate between castes according to some simple rules and this allows the algorithm to autonomously adjust its search strategy achieving a suitable balance between exploitation and exploration. Multi-caste ACS was applied to the symmetric TSP, both to the static and to the periodic and non-cyclic dynamic variants. Results confirm the advantage of the heterogeneous approach. Standard ACO variants excel in a subset of the optimisation scenarios but fail completely on others. On the contrary, Multi-caste approaches are extremely robust and are able to keep a balanced performance across all optimisation scenarios considered. This robustness is particularly evident in the dynamic environments.

Keywords

Ant Colony Optimisation, Dynamic Problems, Multi-population, Robustness, Self-adaptation

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

1LS-FI First Improvement 1-swap Local Search. 31

ACE Ant Colony Extended. 15

ACO Ant Colony Optimization. 3–5, 7, 12, 13, 15, 19–22, 29, 47, 51–54, 75, 84

ACS Ant Colony System. 6, 8–10, 15, 22, 25, 47–49, 53, 55, 56, 64–66, 69, 73, 76–78, 81, 84

AS Ant System. 6–10

AS-SCS Ant Colony Optimization for the Shortest Common Supersequence problem. 15

BMSN Bidirectional Manhattan Street Network. xiii, 26–28, 32

DLB Don't Look Bits. 31

DTSP Dynamic Travelling Salesperson Problem. 84, 85

EAS Elitist Ant System. 8

MACS-VRPTW Multiple Ant Colony System for Vehicle Routing Problem with Time Windows. 15

MC-Ant Multi-Colony Ant Colony System. 21–26, 30, 31, 45–47

MMAS Max-Min Ant System. 6, 10–12, 15, 22

Multi-caste ACS Multi-Caste Ant Colony System. 46–49, 51–53, 55, 64, 67, 69, 73, 76, 78, 81, 84, 97, 103

NPP Node Placement Problem. 24, 27–29

P-ACO Population based ACO algorithm. 19, 20

RAS Rank-Based Ant System. 8

TSP Travelling Salesperson Problem. 15, 52, 53, 64, 76, 84, 97, 103

WDM Wavelength Division Multiplexing. 26

List of Symbols

α ACO parameter. Relative importance of the learned information (trail), $\in \mathbb{R}^+$, usually 1 for ACS. 7, 9, 24, 25, 33, 45, 48, 54

β ACO parameter. Relative importance of the heuristic information, $\in \mathbb{R}^+$, usually between 2 and 5 for ACS. 7, 9, 24, 25, 33, 45, 48, 54

c_{ij} Solution component. Assignment of the value v_i^j to the variable x_i. 4, 7, 10

δ MMAS parameter. Amount of smoothing to be done if $\delta = 1$ the trail is re-initilized, if $\delta = 0$ no smoothing is done,$\in [0, 1]$. 12

$\Delta\tau_{ij}$ Amount of pheromone deposited on component c_{ij} during the pheromone trail update. 5, 7, 10, 11

η_{ij} Heuristic value associated with component c_{ij}. 4, 5, 7, 35, 45, 48, 53

γ MC-Ant parameter. Disturbance range in case of migration. 0.05 by default. 24, 25, 33

ξ ACS parameter. Local pheromone decay rate, $\in]0, 1]$, usually 0.1. 9, 54

λ MMAS parameter. Used to determine the branching factor, $\in \mathbb{R}^+$. 11, 12

L_{best} Quality of the solutions used to update the pheromone trail in ACS. 10, 25

m ACO parameter. Number of ants per colony, $\in \mathbb{N}^+$, usually 10 for ACS. 7, 8, 24, 54

nn ACS parameter. Candidate list length. 9, 54

p_{best} MMAS parameter. Probability of constructing a solution composed only of the components with the higher pheromone, once the algorithm has converged, $\in]0, 1[$. 11

q_0 ACS parameter. Relative importance of exploitation (high q_0) versus biased exploration(low q_0), $\in]0, 1[$, usually 0.9. xiv, 9, 24, 25, 31, 33, 48–50, 52, 54–56, 60, 64, 67–69, 73, 76, 81, 84

ρ ACO parameter. Global pheromone decay rate, $\in]0, 1[$, usually 0.1 for ACS. 7, 10, 11, 24, 25, 33, 54, 84

τ_{ij} Pheromone intensity value associated with component c_{ij}. 4, 5, 7, 9–12, 48, 53

τ_0 ACO parameter. Initial trail intensity, $\in \mathbb{R}^+$, small value for AS and ACS, large value for MMAS. 7–10, 24, 33, 54

τ_{max} MMAS variable, maximum trail intensity, $\in \mathbb{R}^+$, problem dependent. 10–12

τ_{min} MMAS variable, minimum trail intensity, $\in \mathbb{R}^+$, problem dependent. 10, 11

1 Introduction

Nature has many examples of biological systems composed by populations of simple agents that exhibit a collective intelligent behaviour. Individual agents interact locally and with the environment. Typically they follow a set of simple rules, do not require central coordination, and are robust to noisy and dynamic situations or to the malfunction of some individuals [Floreano and Mattiussi, 2008]. These basic local interactions may allow the emergence of a self-organised collective behaviour, able to address and solve complex situations. Examples of this emergent collective behaviour are the instinctive swarming (collective motion of a large number of insects), schooling (a group of fishes swimming in the same direction in a coordinated manner), flocking (behaviour exhibited by a group of birds while foraging or flying) or herding (a group of mammals gathering and acting as a group without centralised direction while, e.g., fleeing a predator).

Swarm Intelligence comprises a set of computational methods that are inspired by the collective behaviour of societies of natural agents. In these algorithms, a group of artificial individuals cooperate, in a decentralised manner, to solve a given problem. They are able to address complex situations in different fields such as optimisation, robotics or data analysis.

Ant Colony Optimization (ACO) is one remarkable example of a swarm intelligence metaheuristic. Metaheuristics are approximation global optimisation algorithms. They are particularly effective in hard optimisation scenarios for which an analytical solution does not exist or cannot be obtained in a feasible time.

Also, given their stochastic behaviour and problem-independence, they can tackle incomplete, noisy, and dynamic problems.

The ACO field comprises a set of bio-inspired optimisation methods loosely inspired by the foraging behaviour of natural ant colonies. Marco Dorigo proposed the original variant [Dorigo et al., 1996], consisting of an iterative algorithm controlling a set of artificial agents (the ants) that cooperatively explore the search space seeking for good quality solutions for a given problem. Each artificial ant builds solutions guided by problem-specific heuristic information and dynamic feedback provided by the other ants of the colony. Feedback takes the form of stochastic pheromone information signalling promising solution components, *i.e.*, components that tend to appear in good quality solutions. This mechanism models indirect communication between ants and allows for the emergence of a cooperative problem-solving behaviour. Today there are many ACO variants, with differences in the problem-construction strategy or the pheromone update methods. ACO methods have been successfully applied to a range of problems, including routing, scheduling, subset selection, among many others. Furthermore, they have been applied, not only to deterministic, discrete, single objective, static problems but also to multi objective, dynamic, continuous, or stochastic problems. An overview of some of the most notable applications of ACO can by consulted in [Stützle et al., 2011].

1.1 Motivation

In many complex optimisation scenarios, metaheuristcs offer an efficiency/effectiveness tradeoff better than that of traditional, or exact, methods. However, the advantage of problem independence is also one of their main weaknesses. Standard metaheuristics frameworks are black-box approaches that take few assumptions about the problem being solved and, as such, they are unable to exploit the characteristics of the optimisation problem at hand. An experienced heuristic designer, with a deep understanding of the key properties of a particular problem, would almost always be able to develop a problem-specific heuristic that is superior to a black-box approach, regardless of the metaheuristic framework used [Sörensen et al., 2018].

Focusing on ACO algorithms, the adjustment of the main components and a careful parameter setting may have a large impact in performance, allowing their application to new situations and/or enhancing effectiveness on problems usually addressed. Specifically, the optimal settings depend on the problem being tackled, on the specific problem instance being solved and even on the particular moment in the search process. However, performing the right modifications is far from trivial and it requires a deep understanding of both the algorithm's behaviour and the properties of the problem to solve. Also, a straightforward trial and error strategy is usually too time consuming to be considered in difficult optimisation situations, where efficiency is vital.

Self-adaptive ACO that autonomously tune their parameters are a promising alternative to manual tuning. Granting an algorithm the ability to change its parameters during execution, both simplifies the initial parameter-setting step and allows the optimisation method to autonomously and dynamically self-adapt its behaviour to the different stages of the search process, e.g., by adjusting the exploration / exploitation tradeoff according to the structure of the landscape.

Another common hindrance in ACO is the danger of premature convergence to a low quality local optimum. If the time available is small, the ability to exploit the knowledge already acquired is important to reach good solutions fast, but, if the strategy is too greedy, it may lead the algorithm to converge to a local optimum and prevent further improvements.

Dynamic optimisation problems pose additional challenges to ACO. When a change occurs, quite possibly the optimum also changes. As such, when dealing with a dynamic problem, the aim of the algorithm becomes not just finding the optimum, but to track it through the changing environment [Branke, 2002]. Intuitively, ACO seems well suited to deal with dynamic problems, as natural ants are clearly very apt to deal with change. However, reality shows that once the ACO algorithm converges, it looses the ability to quickly adapt to a new environment. It is crucial to maintain a certain level of diversity to prevent convergence and to encourage the exploration of the search space. A possible tatic to maintain diversity in a population-based algorithm is to rely on a heterogeneous distributed multi-population approach, able to simultaneously maintain different search strategies. This framework might promote an enhanced exploration of the search space

by taking advantage of the specific strengths of each sub-population in different stages of the optimisation.

1.2 Objective and Contributions

The main goal of this work is to design and study two novel self-adaptive multi-population ACO frameworks. Results presented in this dissertation reveal that heterogeneous self-adaptive ACO architectures are robust search procedures, with enhanced effectiveness. The increased robustness reflects on a simplified task of defining the initial settings and, particularly, on the ability to self-adapt the search behaviour throughout the iterative optimisation process. The outcomes of our work also reveal that the self-adaptation skills are particularly relevant for dynamic optimisation situations, as they provide a simple and effective way to deal with periodic and non-cyclic changing environments.

1.2.1 Contributions

This dissertation proposes two novel contributions to the area of self-adaptive ACO. Both contributions are based on the existence of heterogeneous groups of ants, each with its own search strategy. Information between groups is periodically exchanged, leading to the emergence of a robust cooperative search strategy.

MC-Ant

The Multi-Colony Ant Colony System (MC-Ant) is an ACS-based framework and it comprises a set of different colonies with independent search behaviour. Each colony builds and updates its own individual pheromone trail (a model of the knowledge that the colony acquired about the problem being solved) and has its own set of parameters. Cooperation emerges by a periodic migration of ants and settings. This way successful colonies influence the search behaviour of other groups of ants.

The robustness and effectiveness of MC-Ant is assessed by applying it to the Node Placement Problem (NPP), a difficult combinatorial optimisation problem. A detailed analysis is presented to gain insight about the advantages provided by

the multi-colony framework, when compared to a standard single-colony approach. The migration flows and the evolution of the parameter settings throughout the optimisation are critically examined. The main conclusions of this work were presented in [Melo, 2009] and [Melo et al., 2009].

Multi-caste ACS

The Multi-Caste Ant Colony System (Multi-caste ACS) considers a single colony composed by different groups of ants, known as castes. All ants share and update a common knowledge pool. However, each group pursues a different exploration strategy by maintaining its own q_0 value, a parameter that strongly influences Ant Colony System (ACS) search behaviour. By granting the algorithm with alternative strategies, it might select the most promising caste at any given search stage. Also, the relative size of the castes changes during the optimisation, thus reinforcing specific strategies when needed and autonomously modifying the global behaviour of the algorithm.

Multi-caste ACS is applied to the Travelling Salesperson Problem (TSP) and to the Dynamic Travelling Salesperson Problem (DTSP). Results reveal that the approach is particularly effective and robust in large DTSP instances. We present a detailed analysis correlating the effectiveness of the framework with the magnitude and frequency of change, two important factors when considering dynamic optimisation problems.

The introduction of Multi-caste ACS and its application to the TSP problem was presented in the paper [Melo et al., 2011]. The application of Multi-caste ACS to the DTSP and the detailed analysis of the results was presented in [Melo et al., 2013b], [Melo et al., 2013a], and [Melo et al., 2014].

1.3 Structure of the dissertation

The remainder of this dissertation is structured as follows.

In Chapter 2, the most relevant variants of the ACO metaheuristic are presented. We discuss the most common approaches for defining parameter settings and provide an overview on multi-population ACO approaches. Lastly, we present

a taxonomy of dynamic optimisation problems and discuss several algorithmic features related to dynamic environments.

Chapter 3 comprises the presentation of the MC-Ant approach. The framework is applied to the NPP and a detailed analysis of the most important results is performed.

In Chapter 4, the Multi-caste ACS framework is presented and applied to the TSP. A thorough analysis is described to gain insight of the main strengths and weaknesses of the approach.

The extension of Multi-caste ACS to dynamic environments is presented in Chapter 5. A brief overview of the most relevant existing approaches is performed, followed by a detailed analysis of the main results achieved with the DTSP.

Finally, Chapter 6 draws the main conclusions of the thesis and presents some possible directions regarding future work.

2

State of the Art

One way to find the global optimum to a combinatorial optimisation problem is to generate and test each possible solution, but the computational cost of such algorithm makes it impractical in nearly all relevant situations. For this reason, when dealing with larger or harder combinatorial optimisation problems, a common approach is to rely on approximate methods, such as ACO.

2.1 ACO

ACO is inspired in social insects such as ants. In many species an ant walking to or from a food source leaves a volatile chemical substance in the ground called pheromone. An ant moves essentially at random, but, if there is pheromone on the ground, the ant can detect it and tends to follow it with a high probability, ultimately reinforcing it. If not reinforced by ants, the pheromone trail evaporates over time loosing intensity.

The "double bridge experiment" showed that the pheromones played an autocatalytic role in the ants foraging behaviour. Given two bridges with the same length linking the nest to the food source, initially the ants randomly choose the bridge to cross. But if the pheromone intensity of one of the bridges becomes a little larger than that of the other, the bridge with more pheromone attracts more ants, which in turn deposit more pheromone thus increasing the difference between the bridges [Deneubourg et al., 1990]

The "double bridge experiment variant", where one of the bridges is longer than the other, showed that after some time the ants prefer the shortest path [Goss et al., 1989]. This can be explained as the shortest bridge takes less time to be transversed and, thus, each of its sections were reinforced more often.

ACO mimics this indirect communication behaviour [Dorigo et al., 2006]. In an ACO algorithm, artificial ants iteratively construct a solution for a given optimisation problem guided by both an artificial pheromone trail and some heuristic information. The pheromone trail encodes the information gathered by the ants in previous iterations of the algorithm. While the heuristic information is static, the pheromone trail is updated according to the quality of the solutions found in previous iterations [Dorigo et al., 2006]. As the search progresses, the trail tends to reflect the best solutions found so far.

2.1.1 General overview

An artificial ant in ACO can be seen as a constructive procedure. Thus ACO is suited for combinatorial optimisation problems for which a constructive heuristic exists [Dorigo and Stützle, 2010]. A combinatorial optimisation problem, $P = (S, \Omega, f)$, consists of three elements:

1. a finite search space S comprising all candidate solutions s. Each solution is composed by a set of discrete variables x_i, where x_i can take values $v_i^j \in \{v_i^1, \cdots, v_i^{D_i}\}$;

2. a set $\Omega(t)$ of constraints over those variables;

3. an objective function f, which assigns a cost value $f(s,t)$ to each candidate solution $s \in S$.

The parameter t indicates that either or both the objective function and the constraints can be time dependent, as in the case of dynamic problems. A feasible solution $s \in S$ is a complete assignment of values to variables that satisfy every constraint in Ω. The goal of the algorithm is to find the global minimum[1], *i.e.*, a solution $s^* \in S$, such that $f(s^*) \leq f(s), \forall s \in S$.

[1] Any maximisation problem can be trivially converted in a minimisation problem: maximising a function g is equivalent of minimising the function $f = -g$

Let us call component to each possible assignment of a value to a variable. Each variable x_i generates $|D_i|$ components. The set of all possible solution components is called C and component c_{ij} denotes $x_i = v_i^j$. Both an intensity of the trail (pheromone) value, τ_{ij}, and an heuristic value, η_{ij}, are assigned to each component, c_{ij}. These values reflect the desirability of adding the component to the solution.

A graph $G = (C, L)$ where the set L establishes connections between the components of C, is built. Two values are associated with each component: the artificial trail, obtained from the pheromone values, τ_{ij}, and the heuristic information based on the heuristic values, η_{ij}. The trail acts as a memory about the search process and, ideally, will help to guide the ants to promising regions on the solution landscape. The heuristic information represents knowledge about the problem and is known before the search procedure starts: e.g., an estimation of the cost of adding a particular component to the solution. Heuristics can be particularly useful in the early stages of the search, when the pheromone trail has not yet enough experience accumulated.

In the construction procedure, each ant begins with an empty solution and probabilistically adds components one by one until a complete solution is obtained. At each decision point, only components that do not violate the feasibility constraints are considered. Once the solution is completed and evaluated, the trail is updated by depositing pheromone on the connections that the ant used to build the solution. The amount of pheromone placed, $\Delta\tau_{ij}$, may be proportional to the quality of the solution.

The ACO metaheuristic

A metaheuristic is a general-purpose, high-level strategy, that guides the way simpler heuristics are applied. It can be easily customised to solve different kinds of problems. ACO is a metaheuristic for difficult combinatorial optimisation problems [Dorigo and Stützle, 2010].

The general algorithm consists of three phases (see Algorithm 2.1). After the initialisation and until some termination criterium is met, the following steps are repeated: the ants construct some solutions, the solution(s) are improved by local search (this step is optional) and, at last, the pheromone trail is updated. These

```
Data: problem instance, parameters
Result: best-so-far solution
begin
    load the instance
    set parameters
    initialise the pheromone trail
    while termination condition not met do
        construct ant solutions
        apply local search (optional)
        update pheromone trail
    end
end
```

Algorithm 2.1: Ant Colony Optimisation metaheuristic

three steps can be detailed as follows:

Construct ant solutions: Starting from an empty solution, each ant adds components as it transverses the trail. At each decision point, only the components that do not violate the feasibility constraints are considered. The choice is probabilistic and biased by the pheromone value of the components and the heuristic information. The specific formula used to select a component varies according to the variant.

Apply local search: The solutions found are improved by applying some local search algorithm. The local search can be applied to only a subset of the solutions.

Update pheromone trail: The goal of the pheromone updating is to increase the value associated with the components found in good solutions and to decrease the pheromone value of the other components. First, all the pheromone values are slightly decreased through "evaporation", to avoid stagnation; then the components used in the solutions built by the ants are reinforced. The increment is proportional to the quality of the solutions found, biasing future choices made by the ants.

2.1.2 ACO variants

The first ant algorithm, Ant System (AS), was proposed by [Dorigo et al., 1991] as a multi-agent, positive feedback stochastic search strategy for optimisation problems. It proved to be a viable way of solving hard combinatorial problems, but its performance on large instances was not as good as other, more fine-tuned, algorithms. Since then, many other variants were developed to try and overcome this limitation, notably ACS and Max-Min Ant System (MMAS).

Ant System (AS)

AS [Dorigo et al., 1991, Colorni et al., 1991, Dorigo et al., 1996] was the first ACO algorithm and served as the base for all the others. The number of ants, m, is proportional to the problem size. Its main features are described in the next paragraphs.

Initialise the pheromone trail When the algorithm starts, the pheromone level of each component is set to an arbitrary small value, τ_0.

Construct ant solutions Each ant starts with an empty solution and, at each decision point, it adds a single component. Let s^p be a partial solution and $N(s^p)$ be the set of components that may be added to the solution without violating any of the feasibility constraints. The probability p_{ij} of the component $c_{ij} \in N(s^p)$ to be added to s^p is given by Equation 2.1

$$p_{ij} = \begin{cases} \frac{{\tau_{ij}}^{\alpha} \times {\eta_{ij}}^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{il}^{\alpha} \times \eta_{il}^{\beta}} & \text{if } c_{ij} \in N(s^p) \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where τ_{ij} is the pheromone intensity and η_{ij} is the heuristic value for the presence of c_{ij} in the solution. The parameters α and β are used to control the relative influence of the trail and the heuristic, respectively. A larger α emphasises the influence of the knowledge gathered in the trail, whereas a larger β bias the decision towards the components with a more favourable heuristic value.

Update pheromone trail Let τ_{ij} denote the pheromone level of the component c_{ij}. At the end of each iteration, Equation 2.2 is used to update the pheromone value of each component.

$$\tau_{ij} \leftarrow (1-\rho) \times \tau_{ij} + \sum_{k=1}^{m} {\Delta\tau_{ij}}^{k} \tag{2.2}$$

where ρ represents the evaporation rate, m is the number of ants in the colony and ${\Delta\tau_{ij}}^{k}$ is the amount of pheromone laid on component c_{ij} by ant k. ρ should be $0 < \rho \leqslant 1$ and ${\Delta\tau_{ij}}^{k}$ is calculated using Equation 2.3.

$${\Delta\tau_{ij}}^{k} = \begin{cases} \frac{Q}{L_k} & \text{if ant } k \text{ used component } c_{ij} \text{ in its solution} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where Q is a constant, possibly 1, and L_k is the quality of the solution found by ant k. This way, first all components have their pheromone value decreased by a small amount. Afterwards, only the components used to build the solutions are reinforced. Components that were used often and were present in good solutions have a stronger reinforcement, and are more likely to be choosen in future solutions. Components not present in the solutions have the trail "erased" and are less likely to be chosen in the future.

The steps *Construct ant solutions* and *Update pheromone trail* are repeated for a given number of iterations, until the termination criterion is met.

Early improvements to AS Two early improvements on the update pheromone trail procedure used by AS were proposed: Elitist Ant System (EAS) and Rank-Based Ant System (RAS).

In EAS [Dorigo et al., 1991, Dorigo et al., 1996], there is an additional reinforcement of the trail made by the best-so-far ant. In addition to the usual update pheromone trail procedure, at each iteration, the components belonging to the best solution found since the start of the algorithm, are further reinforced. Results presented in the aforementioned study revealed that this alteration increased the quality of the solutions found and reduced the time needed to find them.

In RAS [Bullnheimer et al., 1997] the ants are sorted according to the quality of the solution they constructed. During the update pheromone procedure only the

best-ranked ants get to update the trail and the amount of pheromone deposited by an ant is inversely proportional to its rank. Additionally, like in EAS, the best-so-far ant also updates the trail.

Ant Colony System (ACS)

ACS is a variant of AS that aims to improve its effectiveness when applied to the larger instances of the TSP [Dorigo and Gambardella, 1997]. It differs mainly in three aspects: the pheromone update occurs only on the components used by the best-so-far solution, an additional pheromone decay is used during the solution construction phase, and the solution component selection mechanism is more aggressive. The number of ants, usually 10, is also smaller than the value used in AS, that recommended using as many ants as the size of the problem (number of variables).

Initialise the pheromone trail In the beginning, each segment of the trail is set to τ_0. τ_0 can be a predefined parameter, although it is frequently calculated using Equation 2.4

$$\tau_0 = \frac{1}{n \times L_h} \tag{2.4}$$

where n represents the size of the problem and L_h is the quality of the solution found using a simple heuristic procedure.

Construct ant solutions ACS has a distinctive component selection mechanism, different from the original AS. At each decision point, ACS chooses component c_{ij} using the pseudorandom proportional rule indicated in Equation 2.5:

$$c_{ij} = \begin{cases} argmax_{c_{ij} \in N(s^p)} \left\{ \tau_{ij}^{\alpha} \times \eta_{ij}^{\beta} \right\} & \text{if } q < q_0 \\ \text{probabilistic selection according to equation 2.1} & \text{otherwise} \end{cases} \tag{2.5}$$

where $q_0 \in [0, 1]$ is a predetermined parameter, q is a uniformly distributed variable over $[0, 1]$ and $argmax_x f(x)$ represents the value of x for which the value

of $f(.)$ is maximised. As the setting q_0 increases, the algorithm steadily becomes greedier. In ACS α is frequently set to 1.

To counterbalance the greediness of the component selection, ACS promotes exploration by introducing a pheromone update mechanism during the construction step. At each decision point the ants update the trail, slightly decreasing the pheromone level of the component they just chose, according to Equation 2.6

$$\tau_{ij} \leftarrow (1 - \xi) \times \tau_{ij} + \xi \times \tau_0 \tag{2.6}$$

where $\xi \in (0, 1]$ is the pheromone decay coefficient and τ_0 is the trail initial pheromone value. This way, it is less likely that two ants will choose the exact same components in a given iteration.

ACS also proposed the use of a candidate list, when applied to large instances. The candidate list comprises, for each decision point, a fixed number, nn, of preferred components. When constructing a solution, an ant chooses the next component among those of the candidate list, if possible. When all the members of the candidate list for that decision point have already been used, one of the remaining components is chosen. The candidate list is also known as the nearest neighbour list and is usually computed in the *load the problem* step.

Update pheromone trail The global pheromone update, at the end of each iteration, considers only one ant, either the iteration best or the best-so-far. The formula is presented in Equation 2.7,

$$\tau_{ij} = \begin{cases} (1 - \rho) \times \tau_{ij} + \rho \times {\Delta\tau_{ij}}^{best} & \text{if } c_{ij} \text{ belongs to the solution} \\ \tau_{ij} & \text{otherwise} \end{cases} \tag{2.7}$$

where ${\Delta\tau_{ij}}^{best}$ is calculated using Equation 2.8. L_{best} can be either the quality of the best solution found in the current iteration, the quality of the best solution ever or a combination of both according to the designer decision.

$${\Delta\tau_{ij}}^{best} = \begin{cases} \frac{1}{L_{best}} & \text{if } c_{ij} \text{ belongs to the solution} \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

ACS has proved to be an improvement over AS, while maintaining most of the simplicity of the original algorithm.

Max-Min Ant System (MMAS)

MMAS [Stutzle and Hoos, 1997] was proposed as an improved version of AS. Like in ACS only the best ant is allowed to reinforce the trail. Yet, as this strategy increases the chance of early stagnation, explicit limits to the maximum, τ_{max}, and minimum, τ_{min}, values for the trail strength are introduced (hence the name, Max-Min). Furthermore, a "trail smoothing" mechanism is used in case of search stagnation.

Initialise the pheromone trail τ_0 is set to a large value, to increase exploration [Stützle and Hoos, 2000].

Construct ant solutions It adopts the strategy from AS.

Update pheromone trail After all the ants have built a solution, the pheromone values are updated by applying Equation 2.9

$$\tau_{ij} \leftarrow (1-\rho) \times \tau_{ij} + {\Delta\tau_{ij}}^{best} \tag{2.9}$$

where ${\Delta\tau_{ij}}^{best}$ is calculated using Equation 2.8. Like in ACS, L_{best} can refer to the quality of either the iteration-best solution, or the best-ever solution, or a combination of both. To ensure that none of the components becomes too extreme, trail limits are imposed and the pheromone values are adjusted according to Equation 2.10

$$\tau_{ij} = \begin{cases} \tau_{max} & \text{if } \tau_{ij} > \tau_{max} \\ \tau_{min} & \text{if } \tau_{ij} < \tau_{min} \\ \tau_{ij} & \text{otherwise} \end{cases} \tag{2.10}$$

This guarantees that, at all times and for all components, $\tau_{min} \leq \tau_{ij} \leq \tau_{max}$. Every time a new best-ever solution is found, τ_{max} is recalculated using Equation 2.11

$$\tau_{max} = \frac{1}{\rho \times L_{best}}. \tag{2.11}$$

Since τ_{min} is function of τ_{max} it has to be recalculated as well using Equation 2.12.

$$\tau_{min} = \frac{\tau_{max} \times (1 - \sqrt[n]{p_{best}})}{(avg - 1) \times \sqrt[n]{p_{best}}}. \tag{2.12}$$

where avg is the the average number of choices that an ant has at each decision point and p_{best} is a parameter that should reflect the desired probability of building a solution where all the components are the ones with higher pheromone intensity, once the algorithm has converged - if the user sets p_{best} to 0 or very close to 0, MMAS sets $\tau_{min} = \tau_{max}$, and only heuristic information will be used; if p_{best} is set to 1, $\tau_{min} = 0$. MMAS has converged if, for each decision point, the pheromone value of one of the components is τ_{max} and for all other components is τ_{min}.

Pheromone trail smoothing is an additional mechanism used when the algorithm is very close to converge or a number of iterations without a new best is reached. The convergence is detected by the average λ branching factor, where λ is a parameter of the algorithm. We can define the λ branching factor of a node i as follows [Gambardella and Dorigo, 1995]: let τ_i^{max} and τ_i^{min} be respectively the largest and the smallest pheromone values on all edges exiting from i. Given a parameter λ ($0 \leqslant \lambda \leqslant 1$), the λ branching factor of i is given by the number of edges exiting from i that have a pheromone value higher than $\lambda \cdot (\tau_i^{max} - \tau_i^{min}) + \tau_i^{min}$. If the pheromone trail smoothing is triggered, the trail values will be altered according to Equation 2.13

$$\tau_{ij} \leftarrow \tau_{ij} + \delta \times (\tau_{max} - \tau_{ij}). \tag{2.13}$$

where $\delta \in [0, 1]$ is the amount of smoothing that should be done. Since the value τ_{ij} after the smoothing is proportional to its original value, if $\delta < 1$, the information gathered in the trail becomes weaker but it is not completely lost. If $\delta = 1$, the trail is restarted.

MMAS improved the results of AS, although it implies an increasing the complexity of the original algorithm by adding parameters and non trivial computa-

tions, such as the average λ branching factor.

2.2 ACO parameter settings

Combinatorial optimisation problems can be hard to solve given the large number of possible solutions. When we use metaheuristics, the objective shifts to obtaining best possible solution in a reasonable amount of time. The flexibility of metaheuristics results, in part, from the possibility to tune its parameters according to the specific problem being solved and the computational time available. The number of parameters and the combination of values they can take can be very large, and the result of the optimisation is highly influenced by these settings [Ridge, 2007]. However, tuning an optimisation algorithm is far from trivial. It is mostly done empirically and it requires expert knowledge, both pertaining the method and the problem being solved.

Like in other metaheuristics, ACO settings strongly influence the algorithm behaviour [Stützle et al., 2010]. To understand the relation between the parameters setting, problem characteristics, and the performance of ACO, several approaches have been considered. Offline configuration strategies try to find the appropriate settings before the algorithm is deployed. They can be classified as [Ridge, 2007]:

Analytical It recommends parameter settings based on mathematical proof

Empirical It is done by either trial-and-error or systematic testing

The analytical techniques for ACO are fairly recent (for instance [Dorigo and Blum, 2005], [Pellegrini et al., 2006]), and are not yet capable of fully formalising the behaviour and performance of existing metaheuristics, resorting to large simplifications of the algorithms. These over simplifications can render the conclusions inapplicable for practical use [Ridge, 2007].

Empirical testing requires that a range of possible parameters is provided (advisable values can be found, e.g., in [Dorigo and Stützle, 2004]), but testing all the values may be unfeasible (e.g., when ACO are used on embedded systems). Even if possible, it may lead to lengthly computations on fast computers [Laptik, 2011].

In most ACO approaches the settings remain constant during the optimisation, but varying the parameters values, according to some pre-set rule or depending

on the search progress, can improve the performance of the algorithms [Stützle et al., 2010]. Online tuning consists of changing the parameter setting while the algorithm is solving a problem [Stützle et al., 2010]. It can be useful when the instances from a given problem are heterogeneous and the parameters that are adequate for the average instance may produce very poor results for a specific situation. Online tuning can also be useful when the explorative and exploitative phases of the algorithm require different settings. Another situation is when the algorithm is used in a context not covered by the offline tuning [Stützle et al., 2010]. Yet, it is expected that the enhancement in effectiveness might be sometimes compromised by the increase in the complexity of the original algorithm, as they require extra parameters to be set [Laptik, 2011].

Several online parameter control procedures have been proposed in the literature to adapt the parameters of an ACO algorithm while solving a problem instance. A taxonomy of those techniques is [Stützle et al., 2010]:

Pre-scheduled Parameter values change according to some pre-established rule depending on computational time or iterations. This is possibly the simplest self-adapting strategy, but it requires additional parameters to control the schedule.

Adaptive The changes to the parameters are function of some statistics on the algorithm behaviour. The function must be predefined.

Self-adaptation The ACO itself optimises its own parameters while solving the problem.

Search-based adaptation Another heuristic is used to find the best ACO parameters.

An extensive review of online tuning strategies can be found in [Stützle et al., 2010]. In this work several strategies found in literature are described and classified, but all of them deal with static optimisation problems.

2.3 Multi-population ACO approaches

Parallel computing techniques have been used to increase the efficiency of population based algorithms, specially when dealing with larger problem instances, so ACO is a clear candidate for parallelisation [Stützle, 1998]. The parallelisation level can be fine (ant-based parallel approaches, like [Talbi et al., 1999]) or coarse grained (multi-population or parallel colony approaches). The communication cost implied by the parallelisation at the ant level seems to recommend the coarse grained approaches [Bullnheimer et al., 1998].

The idea of using more than one group of ants was present from the early stages of ACO, as parallel strategies for ACO were studied ([Bullnheimer et al., 1998], [Stützle, 1998]). More than simple independent runs of the algorithm, the various groups also share the information throughout the run. A multi-population ACO is similar to the notion of island-model for genetic algorithms: it has several groups of ants working in an independent fashion that regularly exchange information. The idea is to give the ants an opportunity to evolve different pheromone trails and thus track multiple optima in the search space. Communication can take place, e.g., by exchanging ants with good solutions, and using those ants to modify the pheromone trail of the other populations.

One advantage of a multi-population approach is to enable the distribution of the computational effort by different processors. Moreover, even if sharing the same processor, it is possible to simultaneously explore several regions of the search space, thus allowing for the population to evolve in more or less isolated niches. This will likely delay convergence, enhancing the probability of obtaining better final solutions [Branke, 2002]. This last aspect is particularly relevant for our work.

[Middendorf et al., 2002] presents a study using several homogeneous colonies to analyse the influence of information exchange strategies on the degree of similarity of the pheromone trails. For the TSP, with exchange of information every 50 iterations, circular exchange of locally best solution was the best option. The study also compares a single run for T iterations with k independent runs and a multi-colony ant algorithm with k colonies for T/k iterations and concludes that, if there is enough time k, independent run and k colonies are better. Also, communication is important to achieve good quality results. In [Chu and Zomaya,

2006], a multi-colony with circular exchange was tested across a small number of processors. Conclusions reveal that the multiple colony implementation offered a clear improvement over multiple runs of the single colony.

Parallel techniques can differ according to several criteria like granularity, homogeneity/heterogeneity of the ants, degree of communication, or topology of communication. A multi-population ACO can be homogeneous, where all the ants behave the same, or heterogeneous, where the groups of ants differ in their behaviour. In heterogeneous approaches, the groups can differ in one or more of the following topics:

Objective function When tackling a multi-objective problem, each colony can be dedicated to the optimisation of a different objective function. One remarkable example is [Gambardella et al., 1999] dealing with the Multiple Ant Colony System for Vehicle Routing Problem with Time Windows (MACS-VRPTW). Here, two colonies are used, the first one to minimise the number of vehicles, while the second minimises the travelled distance.

Solution construction procedure [Michel and Middendorf, 1999] deals with the application of Ant Colony Optimization for the Shortest Common Supersequence problem (AS-SCS). All colonies are applied to the same problem, but using different construction methods. Two types of colonies, forward and backward, are used. The difference is that the backward colonies do not work on the original sequence, but instead on its reverse. In [Ghimire et al., 2014], half the colonies run ACS and the other half use MMAS. Periodically all the trails are updated reinforcing the components that are common to the two randomly selected colonies, and decreasing the pheromone amount in the ones that are not.

Information used With a strong biological inspiration, Ant Colony Extended (ACE) [Escario et al., 2012] is applied to autonomous surface vessels path planning problems and it divides the ants in two sets: patrollers and foragers. Foragers only use the pheromone trail for guidance, while patrollers can use either the trail or the heuristic information. All ants update the trail. The population dynamics are responsible for regulating the proportion of foragers and patrollers, as well as the total number of active ants.

This might encourage either exploration or exploitation, according to the patrollers results. ACE also uses a partial state space representation, instead of the conventional complete graph. In [Escario et al., 2015], ACE is applied to the TSP.

Multi-colony approaches have been applied to a vast range of problems, both multi-objective (see [Angus and Woodward, 2009] or [García-Martínez et al., 2007] for an overview) and single-objective. Some proposals run on a single processor, whereas others were designed for parallel computing environments, where each of the p colonies runs in one of the p processors ([Janson et al., 2005, Middendorf et al., 2002, Ellabib et al., 2007]). A synopsis of parallel approaches can be found in [Stützle, 1998, Randall and Lewis, 2002, Middendorf et al., 2002, Janson et al., 2005, Ellabib et al., 2007, Pedemonte et al., 2011].

2.4 Dynamic Problems

Many real world problems are dynamic and subject to stochastic change: machines break down, roads close or new ones are added, the quality of material used in manufacture varies over time. To deal with uncertainty one needs powerful heuristics [Branke, 2002].

2.4.1 Dynamic problems features

When a change occurs in an optimisation problem (optimisation goal, constraints, problem instance), the optimum likely changes as well. If that is so, a new best solution, or an adaptation of the previous one is needed. [Branke, 2002]. One way to deal with the change is to find the new optimum by solving the problem from scratch. This implies that the system has to be able to detect the change, so it can start to solve the new problem. It also means that, even if the new optimum should not be very different form the old one, none of the information previously acquired is used.

Another possibility is to continuously adapt the solution to the changes in the environment, reusing the information gathered in the past. Clearly, ants in nature are able to adapt in a continuos fashion, using the information kept in the trail.

Our intuition is that artificial ants are also capable of this. An ideal algorithm to deal with dynamic problems should [Branke, 2002]:

- be able to adapt the solution in an efficient and continuos manner
- integrate the cost of change, so a good trade-off between the solution quality and the change cost can be determined
- generate solutions robust enough so that a good solution quality can be maintained even if the environment changes slightly

During the search, the algorithm gathers information about the search space. If we retain that information, it may be used to our advantage when looking for the next optimum after change has occurred, as long as the change is not so severe as to make the search landscape, before and after change, completely different. If an environment changes so drastically that the new optimum has no resemblance with the previous one, the adaptation is of no use, and the information gathered before has no value. The problem needs to be solved as if a completely new instance has arrived.

Some aspects that need to be considered when devising an algorithm to deal with dynamic environments include [Branke, 2002]:

- Visibility of change: is the change explicit or does the algorithm need to detect the change?
- Is it necessary to change the representation?
- Aspect of change: does the change imply a new optimisation function, a new problem instance, or some constraints?
- Algorithm influence on the environment: will the solution of a previous instance influence or restrict the next solution?

Detecting changes in the environment

An additional difficulty presented by dynamic problems is that the change can be hard to detect and go unnoticed for some time. Sometimes, changes are not

explicitly known by the system. In those cases, the degradation of the performance is sometimes used as an indirect indicator of change. Another way is to keep some individuals constant and see if the fitness value of any of them changes from one iteration to another. Yet another way is to keep a model of the environment and test if the response predicted by the model is the same as the one obtained by the environment. If using this approach, when a change in the environment is detected, the model is updated [Branke, 2002].

Measuring performance

Since there is not a single optimum to reach but a sequence of optima, usually a curve depicting the mean best fitness over time is used. Frequently the comparisons are just visual [Branke, 2002]. For a more accurate comparison, numeric measures are necessary. Let e_t be the t-th evaluation and let T be the total number of evaluations.

- Online performance: average of all evaluations over the entire run, as described by Equation 2.14.

$$P_{online} = \frac{1}{T} \times \sum_{t=1}^{T} e_t. \tag{2.14}$$

  This measure assumes that each solution will be used in real time, as it is produced.

- Offline performance: average of the best-so-far solution found at each time step. Since the environment changes the best-so-far means the best-since-last-change, and is described by Equation 2.15, where $e_t^* = max\{e_\tau, e_{\tau+1}, ..., e_t\}$ with e_τ being the first evaluation since last change. This measure requires that we know at which time step the change occurred.

$$P_{offline} = \frac{1}{T} \times \sum_{t=1}^{T} e_t^* \tag{2.15}$$

  Offline performance is the best measure for our study, as the usual output of an ACO algorithm is the best-so-far solution.

- Average peak: average of the solutions found by the algorithm on the first iteration after each change. The measure is described by Equation 2.16, where C is the set of iterations immediately after a change, and e_i is the solution found at iteration i.

$$Q = \frac{1}{T} \times \sum_{i \in C} e_i \tag{2.16}$$

  This measure gives an idea of the capacity of the algorithm to keep a good performance in the event of change.

2.4.2 Taxonomy of dynamic problems

Not all dynamic problems are the same, and different dynamics probably require different approaches. The following features can be used to characterise a dynamic problem [Branke, 2002]:

- Frequency of change: how often does the problem changes? This helps to establish how much time does the algorithm have to reach the new optimum. In experiments with ACO, the number of iterations between changes has been frequently used as a measure.

- Severity of change: how far is the new best solution from the previous one? Can the new best solution be found just by applying local search to the former solution?

- Predictability of change: Is there a pattern? Is it possible to predict the time, severity or the direction of change, given the changes so far?

Even though the above features may be hard to measure in a way that allows for an accurate comparison between different problems, it is still possible to vary them in a single problem to ascertain their influence in a given algorithm.

2.4.3 ACO approaches for dynamic problems

There are several general methods employed to deal with dynamic problems, including: restart / re-initialisation, increasing diversity, implicit or explicit memory,

modifying greediness level or multi-population approaches.

Some of these strategies can be found in ACO. Generic ACO algorithms retain the acquired information in the trail. The "smoothing" of the pheromone trail values can be used to forget previous information. If the change in the environment is small so that some of the information is still relevant, an higher evaporation rate may be enough. If the change is more severe a re-initialisation of the trail may be better suited. The smoothing technique can be found in several approaches like [Guntsch and Middendorf, 2001, Guntsch et al., 2001, Eyckelhof and Snoek, 2002, Angus and Hendtlass, 2002, Angus and Hendtlass, 2005, Mavrovouniotis and Yang, 2013a, Mavrovouniotis and Yang, 2014b], but it depends on the frequency of change being known beforehand or on the ability of the algorithm to detect (or be informed) that a change has occurred.

Another popular strategy, Population based ACO algorithm (P-ACO) [Guntsch and Middendorf, 2002b], is to keep in memory a population of iteration-best ants. Only those ants are allowed to update the trail. At each iteration, a new ant is added to the memory. Once the memory capacity is exhausted, at each iteration, one ant is removed from the memory and is used to negatively update trail. This way the previous information is deleted as there is no evaporation in P-ACO. The ants kept in memory may need to be repaired in order to be used in the new environment. Several works are based on P-ACO, namely [Guntsch and Middendorf, 2002b, Sammoud et al., 2009, Eaton and Yang, 2014, Wang et al., 2016]. In a similar fashion to P-ACO, immigrant schemes have also been used to increase diversity and avoid stagnation. Immigrants are new solutions used to replace part of the current population. Immigrants can be random or the result of the adaptation of the best ants found in previous iterations. Immigrant schemes are used in [Mavrovouniotis and Yang, 2010, Mavrovouniotis and Yang, 2011c, Mavrovouniotis and Yang, 2011b, Mavrovouniotis and Yang, 2011a, Mavrovouniotis and Yang, 2014c, Mavrovouniotis and Yang, 2014a].

Many applications of ACO to dynamic routing problems share a common strategy [Caro et al., 2008]. Each ant is a simple mobile agent, usually an intelligent control packet, responsible for finding a suitable path between the source node and the assigned destination node. The ants explore the network, collecting information and learning the network dynamics, and continually update the node

routing policies using pheromone values to express them. The emission of ants can be periodical, like [Caro and Dorigo, 1998, Schoonderwoerd et al., 1997], done in an on-demands basis if the bandwidth is scarce, [Guinand and Pigné, 2010], or both [Caro et al., 2004, Gunes et al., 2002]. One crucial aspect of this strategy is balancing the rate of ants generating, and thus the network monitoring, with the resulting network overload.

Another possibility is changing the trail update rule. For example, in [Liu, 2005] the local update rule is replaced by a Q-learning based strategy. Also, it uses a rank-based proportional rule to improve exploration efficiency on larger problems by making nodes with similar pheromone values more distinct.

An overview of some ACO algorithms inspired by the ant behaviour used to deal with dynamic problems can be found in [Leguizamón and Alba, 2013].

3

MC-Ant: a multi-colony ant algorithm

ACO is a successful metaheuristic. Yet, like most metaheuristics, setting the parameters can be cumbersome. Its behaviour is highly dependant on the parameters tuning, but the optimal setting depends, not only on the specific problem instance being solved, but also it will likely change during the run according to the search stage.

Another drawback of ACO is premature convergence, since the search tends to quickly get trapped on local optima. Once the algorithm starts converging, all ants follow similar trails, constructing and reinforcing identical solutions. This makes exploration almost impossible and the algorithm becomes incapable of discovering new solutions. There is a need to find a good balance between exploitation and exploration, so that the algorithm is able to produce good solutions quickly, but does not loose the ability to look for other promising areas in the search space.

MC-Ant is an heterogeneous, self-adapting, multi-population ACO intended to overcome these limitations. Multi-population approaches have a natural ability to keep the diversity. Also, in MC-Ant diversity is further encouraged, as the ants are heterogeneous regarding the settings they use, and these parameters change and adapt over time. This way, the algorithm comprises several coexisting search strategies at the same time, increasing the likelihood of covering a larger area of the search space and hopefully retaining the best qualities of each of the configurations, thus enhancing the robustness of the approach. The heterogeneity and self-adaptation also make the choice of the initial parameters less crucial, as the

algorithm is able to both test several configurations and autonomously adjust its settings to the specific situations faced during the optimisation.

MC-Ant is based on ACS. Both ACS and MMAS have been widely used and are considered to be some of the best performing ACO. ACS is considered to be the more aggressive of the two and able to find good solutions in shorter computation times [Dorigo and Stützle, 2004]. By combining ACS and a heterogeneous multi-colony approach we try to keep the ability to find good solutions fast, while retaining diversity and thus avoiding premature convergence.

3.1 MC-Ant architecture

MC-Ant ([Melo, 2009, Melo et al., 2009]) is a multi-population ACO architecture. We can define a colony as a group of ants that use and update a common trail while solving a problem. When we have more than one set of ants, each using its own trail, so that trails of different colonies are not necessarily equal, we have a multi-colony approach [Janson et al., 2005]. Multi-colony is different from multiple runs of the algorithm, as the colonies periodically communicate by exchanging solutions.

Besides owning its own trail, in MC-Ant each colony also has its own settings and is able to self tune them. Still, all the colonies share the same solution construction procedure, heuristic function and evaluation function, so it does not require additional work for the user. The combination of using several sets of parameters and keeping the trails independent allows for the search process to encompass different behaviours and gives it the ability to simultaneously focus on different search areas.

3.1.1 General Description

The main MC-Ant algorithm is presented in Algorithm 3.1.

MC-Ant has general parameters (number of iterations to run, number of colonies, number of ants per colony (m), disturbance range (γ), and τ_0) as well as colony-specific parameters (α, β, ρ, q_0). All of them are set at the *set parameters* phase. The values can be configured by the user or be set by default. Most general parameters have a default value (8 colonies, $m = 8$, $\gamma = 0.05$, 2500 iterations), while,

```
Data: problem instance, parameters
Result: best-so-far solution
begin
    load the instance
    foreach colony do
        set the parameters
        initialise the pheromone trail
    end
    while termination condition not met do
        foreach colony do
            construct ant solutions
            apply local search (optional)
        end
        migrate best solution and disturb parameters
        foreach colony do update pheromone trail
    end
end
```

Algorithm 3.1: MC-Ant algorithm

for colony-specific parameters, the values are chosen at random from a pre-defined reasonable range [Dorigo and Stützle, 2004]:]0, 10[for α and β, and]0, 1[for ρ and q_0. τ_0 can either be set by the user or have its value computed according to Equation 2.4. Initially all colonies share the same parameters.

During the *initialise the pheromone trail* procedure, each segment of all the trails is initialised with τ_0. The *termination condition* is met if either the optimal solution is found or a predefined number of iterations is reached.

In the *construct ant solutions*, each of the ants constructs a solution. The specific procedure to construct a solution will depend on the problem being solved. In our research we used the NPP and the construction procedure is described in section 3.3. One or more of the best solutions found by each colony in the current iteration are improved in the *apply local search* procedure.

In the *migrate best solution*, the rank of each colony is determined according to the quality of the best solution hd, ever found by that colony. Let hd_{best} (respectively, hd_{worst}) represent the quality of the best (respectively, worst) best-so-far solutions found by the colonies. Let x be a random variable uniformly distributed

over [0, 1]: should $x < \frac{hd_{worst} - hd_{best}}{hd_{best}}$, migration happens[1]. In that case, the solution from the best colony migrates to the worst, becoming the iteration best solution for that group of ants. Also, the worst colony parameters become identical to the best colony parameters, affected by a small disturbance. The amplitude of the disturbance is itself a parameter, γ, and has a default value of 0.05. An uniformly distributed value over $[-\gamma, \gamma]$, called d, is calculated for each parameter. Let p be the parameter to be disturbed, and let p_b be the value of p in the best colony. When exchanging information, the value p_w of this parameter in the worst colony is given by Equation 3.1.

$$p_w = \begin{cases} p_b + d & \text{if } p \in \{\rho, q_0\} \text{ and } (p_b + d) \in]0, 1[\\ p_b + 10 \times d & \text{if } p \in \{\alpha, \beta\} \text{ and } (p_b + 10 \times d) \in]0, 10[\\ p_w & \text{otherwise} \end{cases} \tag{3.1}$$

Each of the received parameters is disturbed by adding a random value between $(-10 \times \gamma)$ and $(10 \times \gamma)$, for α and β, or between $-\gamma$ and γ for ρ and q_0. The parameter values that, after the disturbance, stay within an acceptable range ( ]0, 10[for α and β, and]0, 1[for ρ and q_0), are adopted by the receiving colony.

In the *update pheromone trails* step, each colony uses its own best-so-far solution to update the pheromone trail according to Equation 2.7. L_{best} corresponds to the quality of that colony best-so-far solution. In colonies that received information, the update will be made using the new best solution and the new parameter values.

Contrary to conventional ACS, there is no step-by-step on-line pheromone update as that is a computationally costly step. We expect to preserve diversity by using different trails and parameter configurations. MC-Ant has the following features (using the nomenclature proposed in [Janson et al., 2005] where possible):

- The neighbourhood topology is not fixed. At each communication step, only two colonies are selected as potential neighbours: the colony with the best best-so-far solution and the colony with the worst best-so-far solution.

- The communication time is quality dependent. The colonies that produced the best and worst best-so-far solutions are selected to be potential neigh-

[1] Assuming a minimisation problem, where a lower *hd* corresponds to a better solution.

bours. Should the difference between the quality of their best-so-far solutions be large enough, communication happens.

- The information exchanged is the best-so-far solution discovered by one of the colonies and the parameters used in the colony that found it.

- The migrated solution becomes the new elitist solution in the receiving colony, where it will be used to update the pheromone matrix. The received parameters suffer a small disturbance and will replace the old ones.

- It is heterogeneous within an iteration approach, as each colony uses different parameters. Since each colony updates its own trail, even if just one colony finds a better solution in the present iteration, there is no risk of the improvement being lost, as it will be used to update that colony trail.

- The colonies are heterogeneous between iterations since, if migration occurs, the colony that receives the solution also suffers an alteration in its parameters.

The size of the neighbourhood and frequency of communication are kept small to preserve the diversity. If all the colonies use the same solutions to update the trails, the matrices will become identical. One of the purposes of the communication is to help in case of a given colony gets "trapped" in a local optima. Updating the stagnant trail with an, hopefully, unrelated solution will make the dominant path less pronounced and thus increase the chance of escaping convergence. We share only a single solution so its influence on the pheromone trail does not become overly strong. Changing the parameters is also intended to stimulate the converged colony to pursue a different search direction.

3.2 The Node Placement Problem

We tested the MC-Ant approach on the problem of finding the optimal node assignment in a multihop Wavelength Division Multiplexing (WDM) lightwave network with a virtual BMSN topology.

3.2.1 Problem description

WDM allows for a high level of concurrency in optical networks [Brackett, 1990]. Using different wavelengths of laser lights, multiple signals can be transmitted in parallel over a single strand of fiber, thus making high-speed and large capacity communication possible. Other advantage of using WDM is the ability of creating a virtual network topology different from the underlying physical one. The assignment of wavelengths implicitly determines the virtual topology of the network, and is a strong factor in the efficiency of the network.

The parallel channels of WDM networks can be used to construct single-hop networks where source and destination nodes communicate directly, or multihop networks, where the packets can be routed through intermediate nodes [Mukherjee, 1992a, Mukherjee, 1992b]. Usually single-hop networks are impractical in large-scale networks due to amount of coordination needed between the prospective communicating nodes [Banerjee and Sarkar, 2001], as well as the cost, since nodes must be able to tune different channels quickly. Multihop networks are comparatively static, so tuning times have little impact, but it is unlikely that there is a direct path between every pair of nodes, so a packet will have to hop through zero or more intermediate nodes, until it reaches the destination. The multihop virtual topology should be made close to optimal according to some measure, e.g., that the average hop distance between nodes should be small, or that no packet should have to make more that a pre-determined number of hops. Also the routing mechanism should be simple to minimise the processing time [Mukherjee, 1992b].

Multihop networks can be regular, where every node is connected to the same number of nodes, or irregular. Irregular networks are usually designed to address the optimality problem directly, but the lack of node connectivity structure can make the routing mechanism complex. Regular networks have simplified routing mechanisms, but the connection structure also imposes constraints to the optimality problem and to the number of nodes used. A few multihop topologies have been proposed. BMSN [Maxemchuk, 1985, Maxemchuk, 1987] is one of the regular multihop topologies that provides higher performance when compared with other regular topologies [Marsan et al., 1992, Freire and da Silva, 2001, Baransel et al., 1995]. The BMSN is a toroidal mesh with 2 dimensions, where each node is di-

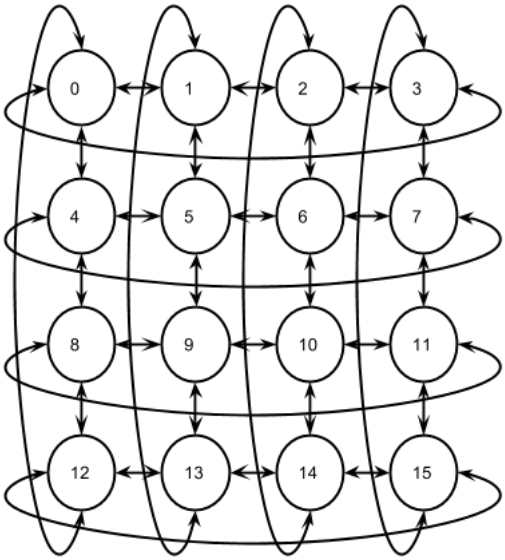

Figure 3.1: A 4 by 4 BMSN

rectly connected to 4 other nodes and is exemplified in Figure 3.1. In a BMSN, each node has a direct link to four other nodes, e.g., in Figure 3.1, node 5 has a direct link to nodes 1, 4, 6 and 9. Communication between some pairs of nodes require several hops. In Figure 3.1 we can verify that node 5 needs to use intermediate node 6 to communicate with node 7, so a packet travelling from node 5 to node 7 will have to make, at least, 2 hops. In fact, in multihop networks, a significant fraction of system capacity is lost due to packet forwarding [Ganz and Koren, 1991].

The logical topology optimisation problem in regular topologies can be considered a NPP [Kato et al., 1999], and is also referred to as Node Assignment Problem or Optimal Node Assignment Problem [Siu and Chang, 2002]. We chose to address the NPP where the objective is to minimize the weighted average hop distance between nodes, *i.e.*, the logical topology should be made such that the average number of hops a packet must make is as small as possible.

Let us consider a BMSN of x by y nodes, with $x \times y = n$. The network can be represented as a graph $G = (V, E)$, where V is the set of node slots and E is the set of bidirectional edges. Each of the n nodes $(0, 1, ..., n-1)$ of the network can be assigned to one of the n slots of the graph without duplication. Let us also

assume that the amount of traffic among each pair of nodes i, j can be given by a traffic matrix T. Let t_{ij} denote the amount of traffic from node i to node j, and let $t_{ij} \in \Re_0^+$. Two nodes i, j can communicate directly if they are in adjacent slots in the graph; otherwise, they require intermediate nodes to communicate, and each packet sent must travel more than 1 hop to reach the destination. Let us assume that there is a function $h(i, j)$ that returns the hop distance of the shortest path between the slots where two nodes, i and j, are located. The objective of NPP is to find a placement σ that minimises the average weighted hop distance between the nodes, *i.e.*, to minimise the function f indicated in Equation 3.2.

$$f(\sigma) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} t_{ij} \times h(i, j) \tag{3.2}$$

3.2.2 Existing approaches for the NPP

Several methods were proposed to solve the NPP in a virtual BMSN. The best until recently, [Kato and Oie, 2000], reports the results obtained with several metaheuristics, such as a greedy method, local search, tabu search, genetic algorithm, simulated annealing, threshold acceptance and multistart local search. It concluded that tabu search following a greedy method yielded the best results. Simulated annealing was used by [Komolafe and Harle, 2003] to study the impact of the traffic patterns in the efficacy of the NPP. [Yonezu et al., 2007] proposes a two-stage hierarchical algorithm similar to [Kato and Oie, 2000], but reports better performance specially for large instances. An iterated local search algorithm based on variable depth search, IKLS, is proposed by [Katayama et al., 2007]. IKLS also outperforms [Kato and Oie, 2000] in terms of time and 1-swap moves. An yet better performing metaheuristic, IGKLS, is proposed in [Toyama et al., 2008]. It is an iterated greedy algorithm with a construction and a destruction phases, and a variable deep search employed after the construction phase.

3.3 MC-Ant for the NPP

To solve the NPP using MC-Ant we need to devise a construction mechanism and to find a way to represent both the heuristic information and the pheromone trail.

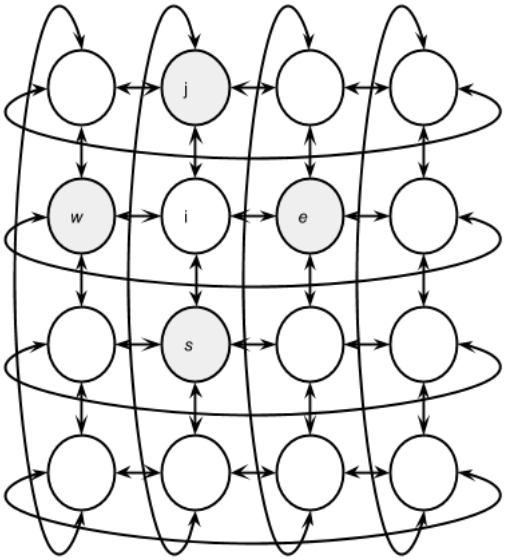

Figure 3.2: Slots σ_j, σ_e, σ_s, and σ_w are directly connected with slot σ_i

3.3.1 Heuristic and pheromone trail information

The heuristic information is obtained from the traffic matrix, T. In concrete, for each pair of nodes i, j, the heuristic value, η_{ij}, is equal to $t_{ij} + t_{ji}$.

Initial experiments were made using a simple trail where τ_{ij} would represent the desirability of placing i and j as direct connections. In an effort to provide a little more guidance, a trail representation adding also the relative orientation of the directly connected nodes was developed. This second representation proved to be more successful and was kept. As such, the trail is used to assign a value to each triple (i, d, j), where i and j are nodes and $d \in \{north, east, south, west\}$. Let σ_i represent the slot where i is placed. τ_{idj} denotes the value associated with placing j in the directly connected slot to the d of σ_i. For example, if d is north, τ_{idj} stands for the value of placing j in the directly connected slot to the north of σ_i. Figure 3.2 illustrates an example, where j is placed (in the directly connected slot) to the north of σ_i.

Since half the information is redundant, (for eg. $\tau_{iNorthj}$ is equal to $\tau_{jSouthi}$), internally, we only keep information for directions north and east.

3.3.2 Solution construction procedure

A solution is constructed adding nodes to the free slots, one by one, until all nodes and slots are exhausted. At each step we much choose both which node to add and in which slot to place it. The first node, i, is randomly selected and positioned at a random slot, σ_i. Then the heuristic information is used to probabilistically choose which unplaced nodes (if any) should be the neighbours of i. Afterwards, for each of the chosen desirable neighbours, the trail information is used to select the slot (from the free slots that are immediately to the north, east, south or west from σ_i) where it should be placed. After all the selected neighbours are placed, the node i is said to be connected, and the process is repeated with each of the newly placed nodes. If all the placed nodes are connected but there are still some unplaced nodes, one random unplaced node is selected and placed at a random free slot, in order to continue the construction of the solution. This process is repeated until all the nodes are placed.

For a given placed node i, with empty slots immediately to the north, east, south or west, let C be the set of all available nodes j for which $\eta_{ij} > 0$. If C is empty, no neighbour is selected, and i is considered connected. Otherwise we use a pseudo-proportional rule to select the next neighbour, j, to be placed. The rule can be consulted in Equation 3.3, where q is a uniformly distributed variable over $[0, 1]$, $q_0 \in [0, 1]$ is a MC-Ant parameter and $argmax_x f(x)$ represents the value of x for which the value of $f(.)$ is maximised.

$$j = \begin{cases} argmax_{j \in C} \left\{ \eta_{ij}^{\beta} \right\} & \text{if } q < q_0 \\ \text{probabilistic selection according to Equation 3.4} & \text{otherwise} \end{cases} \tag{3.3}$$

Equation 3.4 calculates the probability, p_{ij}, of a node $j \in C$ being selected as the next neighbour of i.

$$p_{ij} = \frac{\eta_{ij}^{\beta}}{\sum_{l \in C} \eta_{il}^{\beta}} \tag{3.4}$$

The formula used to determine the slot where to place the selected node j is also a pseudo-proportional rule. Let D be the set of directions in which the slots surrounding i are free. The direction to be used, e, is given by Equation 3.5, where

q is once again a uniformly distributed variable over $[0, 1]$.

$$e = \begin{cases} argmax_{d \in D} \left\{ \tau_{idj}^{\alpha} \right\} & \text{if } q < q_0 \\ \text{probabilistic selection according to Equation 3.6} & \text{otherwise} \end{cases} \tag{3.5}$$

p_{idj} is the probability of choosing direction d and is calculated using Equation (3.6)

$$p_{idj} = \frac{\tau_{idj}^{\alpha}}{\sum_{d \in D} \tau_{idl}^{\alpha}} \tag{3.6}$$

The placed but unconnected nodes are stored in a FIFO queue.

3.3.3 Local search procedure

The local search procedure is a greedy algorithm with a first improvement 1-swap neighbourhood and a Don't Look Bits (DLB) speed-up mechanism[2] [Bentley, 1992] and is similar to the First Improvement 1-swap Local Search (1LS-FI) algorithm described in [Katayama et al., 2007]).

3.4 Experiments

Several experiments were performed to compare the MC-Ant behaviour as the number of colonies varies. The benchmark instances used were the ones provided by [Katayama et al., 2007] and also used by [Toyama et al., 2008]. The instances were constructed in a similar way to that described in [Yonezu et al., 2007,Komolafe and Harle, 2003, Kato and Oie, 2000].

The benchmark set consists of 80 instances of 4 problem sizes ($n = 4 \times 4, n = 8 \times 8, n = 16 \times 16, n = 32 \times 32$) with 20 traffic matrices for each given size. The instances have 16, 64, 256, 1024 nodes respectively. The traffic matrices were generated according to the following general principles: there are two types of

[2] a bit (DLB) is associated with each node and initially set to 0. If for a given node no improvement can be found its DLB is set to 1, and in the next iteration that node is not considered as a possible starting node for a swap. If an arc incident to that node is later changed by a swap then the DLB is set to 0 again.

traffic flow, heavy and light, denoted by t_H and t_L, respectively. Links with heavy traffic were attributed the value 1 in the traffic matrix, and light flow links were given the value 0. Starting from a random σ, $a \times 4 \times n$ links are randomly selected to have heavy traffic, with $0 < a < 1$ (recall that a BMSN of n nodes has a total of $4 \times n$ links). The number of outgoing heavy traffic links per node is limited to a maximum, L_{max}, with $(1 \leq L_{max} \leq 4)$. All other links are assigned light traffic flow. In the specific benchmark set used, the values for a and L_{max} were 0.3 and 3 respectively. Given the way the problem instances were created, we have access to the optimal solution quality, L, (Equation 3.2) for each instance size, *i.e.*, we know that the optimal hop-distance is the largest integer less or equal than $a \times 4 \times n$.

In the experiments, the initial values for parameters α, β, ρ and q_0 were those recommended in [Dorigo and Stützle, 2004]: $\alpha = 1$, $\beta = 2$, $\rho = 0.1$, $q_0 = 0.9$. Parameter γ was set to 0.05 and $\tau_0 = 1/L$, where L is the lower bound for the optimal solution quality. The number of iterations was set to 2500. In the experiments we tested several MC-Ant configurations, comprising a varying number of colonies and amount of ants per colony. Specific values depend on the size of the problem instance. The full list of configurations used are listed on Table 3.1.

To maintain a fair comparison between configurations, for a given instance size, the total number of ants is always the same. Also the total number of solutions subjected to local search remained constant for a given instance size. In configurations with few colonies more solutions per colony were selected for improvement, when compared to configurations with a larger number of colonies (see Table 3.1).

To perform the experiments reported here we selected all the instances in the $n = 4 \times 4$ data set, the first 10 instances in the $n = 8 \times 8$ and $n = 16 \times 16$ sets and the first instance in the $n = 32 \times 32$ data set. Thirty runs were performed with each setting.

3.5 Results

Table 3.2 displays the general results achieved by all configurations for each problem instance size. Column "Avg hd" displays the hop-distance (equation 3.2) of the best-ever solution at the end of the optimisation, averaged over the 30 runs. When several instances of the same size were tested, results are also averaged over

Instance size	configuration name	colonies	ants per colony	local search per colony	local search total
n=4x4	1x16	1	16	4	4
	2x8	2	8	2	
	4x4	4	4	1	
n=8x8	1x64	1	64	8	8
	2x32	2	32	4	
	4x16	4	16	2	
	8x8	8	8	1	
n=16x16	1x256	1	256	16	16
	2x128	2	128	8	
	4x64	4	64	4	
	8x32	8	32	2	
	16x16	16	16	1	
n=32x32	1x1024	1	1024	32	32
	2x512	2	512	16	
	4x256	4	256	8	
	8x128	8	128	4	
	16x64	16	64	2	
	32x32	32	32	1	

Table 3.1: MC-Ant configurations used in the experiments

Instance Size	optimum	configuration	Avg hd	Min hd	Max hd
n=4x4	19	1x16	19.0	19.0	19.1
		2x8	19.0	19.0	19.0
		4x4	19.0	19.0	19.0
n=8x8	76	1x64	76.9	76.1	79.0
		2x32	76.7	76.1	78.7
		4x16	76.5	76.1	77.9
		8x8	**76.4**	76.1	**77.1**
n=16x16	307	1x256	340	321	361
		2x128	338	318	357
		4x64	335	317	357
		8x32	333	**316**	350
		16x16	**332**	317	**349**
n=32x32	1228	1x1024	**1790**	1710	1843
		2x512	1791	1709	1849
		4x256	1791	1732	1850
		8x128	1791	**1664**	**1829**
		16x064	1794	1735	1854
		32x032	1795	1729	1847

Table 3.2: Best solutions found by each MC-Ant configuration. Results are averages of the several instances of the same size.

these instances ($n = 4 \times 4$, $n = 8 \times 8$ and $n = 16 \times 16$). In the analysis we will refer to this value as the mean best fitness. Column "Min hd" (respectively, "Max hd") displays the lower (respectively higher) hd discovered in a specific run. As before, results are averaged over instances of the same size.

It can be observed that for smaller sized instances ($n = 4 \times 4$ and $n = 8 \times 8$) the mean best fitness is identical, regardless of the number of colonies used. Still, in configurations with more colonies, the maximum (worse) hop-distance is smaller than in configurations with less colonies, so more colonies seem to protect against the premature convergence. Following the same trend, for instances of size $n = 16 \times 16$, the results improve with the number of colonies, both in terms of the average result, and in avoiding final poor solutions. For the instance of size $n = 32 \times 32$, the average results are similar. The configuration with 8 colonies and 128 ants per colony exhibits both a good average performance and it discovers the lowest overall hop-distance. Still, results for $n = 32 \times 32$ are less conclusive since only one instance was used. Also the number of iterations was kept constant regardless the size of the instance, so in larger instances a smaller proportion of the search space was explored.

Figure 3.4 depicts the results achieved by each configuration on the 10 $n = 16 \times 16$ instances, averaged for all repetitions. There are slight variations in the results from instance to instance. For example, all configurations were more successful on instance 02 than on instance 10. Also, on instance 06 it is evident a relative order for the outcome of the configurations for each category, (minimum, average, maximum), while, on instance 09, the results almost overlap. There is, nevertheless, a general trend. Figure 3.3 presents the results averaged over all $n = 16 \times 16$ instances. Here we can clearly confirm the advantage provided by the configurations with more colonies.

The same general pattern can be discerned for the $n = 8 \times 8$ instances. Figure 3.5 displays the results averaged for all instances, whereas Figure 3.6 details the results for the individual instances.

Instances of size $n = 4 \times 4$ were so easily solved that there are no visible differences between configurations. With $n = 32 \times 32$ only one instance was tested and results are not conclusive.

Looking at the instances within a given dimension, the easier were those with

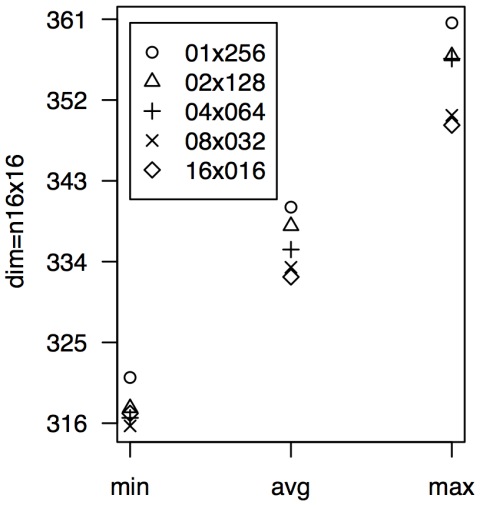

Figure 3.3: Average results of the MC-Ant configurations for the $n = 16 \times 16$ instances

a higher percentage of nodes i, j with high traffic both from i to j and from j to i. This result is expected, since a high traffic link in both ways will produce a more pronounced η_{ij} (and η_{ji}), thus making the choice more obvious. Also, since the number of links with high traffic is constant for a given dimension, and some of the links are consumed in this double connection, the average number of nodes with which a given node has a high traffic link is reduced, making the choices less critical.

We complement the results with the evolution of the mean best fitness of each configuration, averaged over instances of the same size. Results can be consulted in Figures 3.7, 3.8, and 3.9. The vertical axis displays the hop-distance of the mean best fitness and the horizontal axis contains the number of iterations.

Results from Figure 3.7 reveal that, for the $n = 8 \times 8$ instance set, most of the improvement happens up to the first 500 iterations. In the initial iterations, configurations with a lower number of colonies seem to have a slight advantage. However, from about 45 iterations forward and until the end of the run, the performance improves with the number of colonies.

As for the $n = 16 \times 16$ instance set, Figure 3.8 reveals that after 2500 iterations the solutions are still being improved. This is particularly visible for the configu-

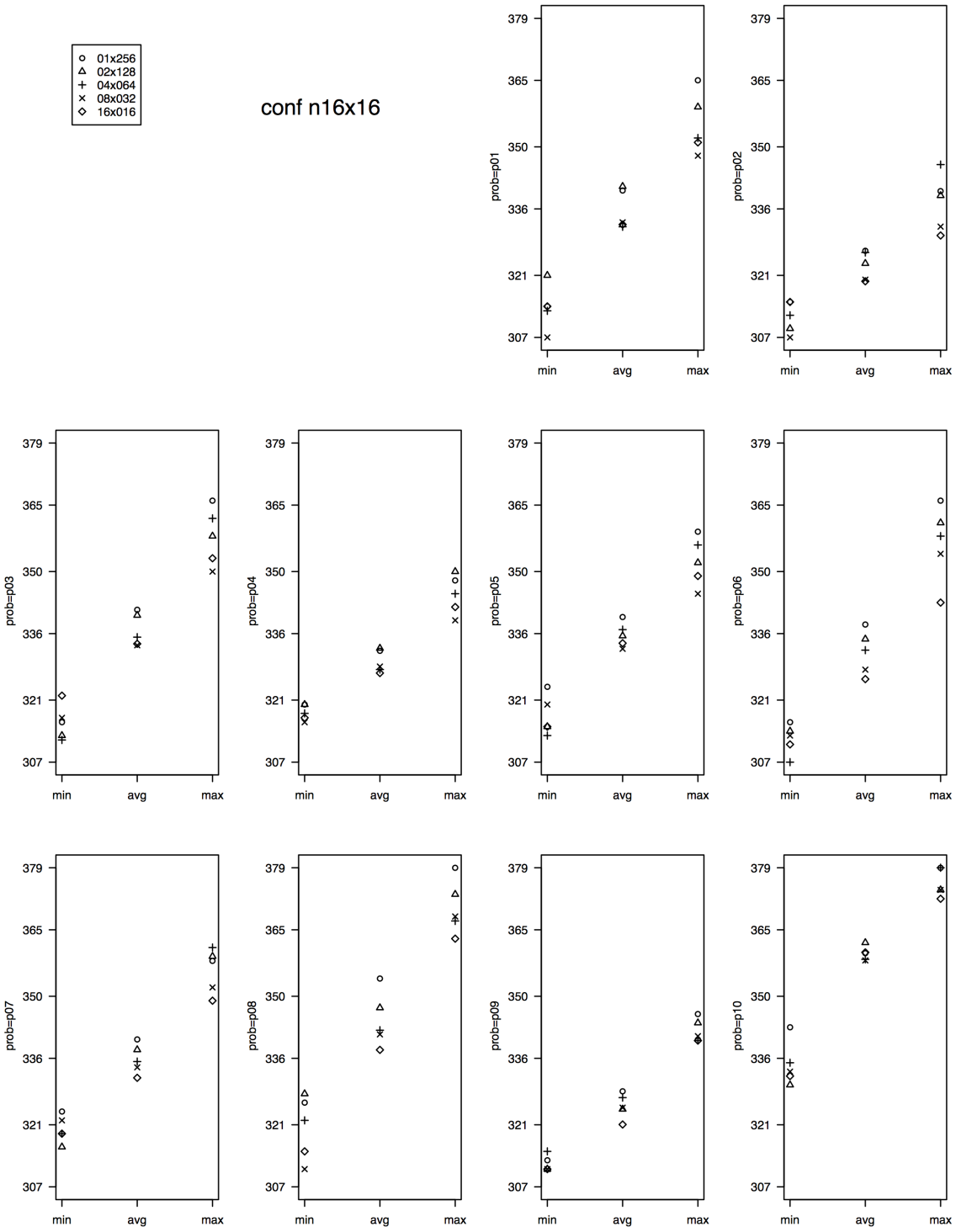

Figure 3.4: Average results of the MC-Ant configurations for each of the $n = 16\times16$ instances

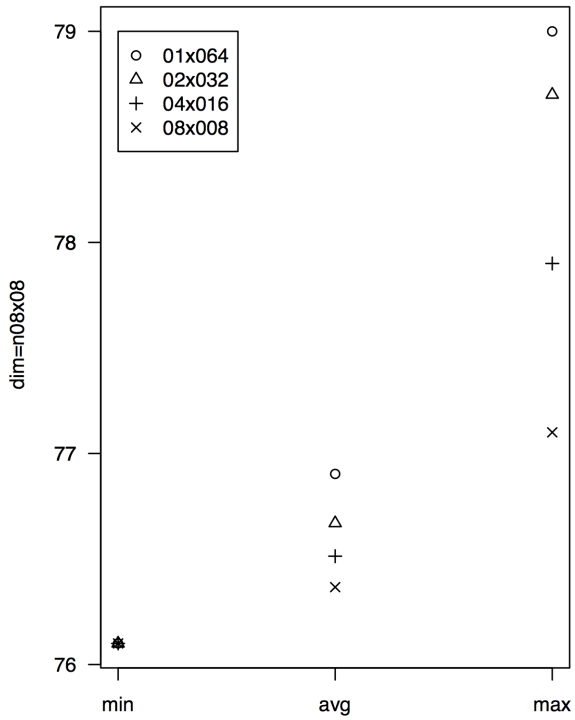

Figure 3.5: Average results of the MC-Ant configurations for the $n = 8\times8$ instances

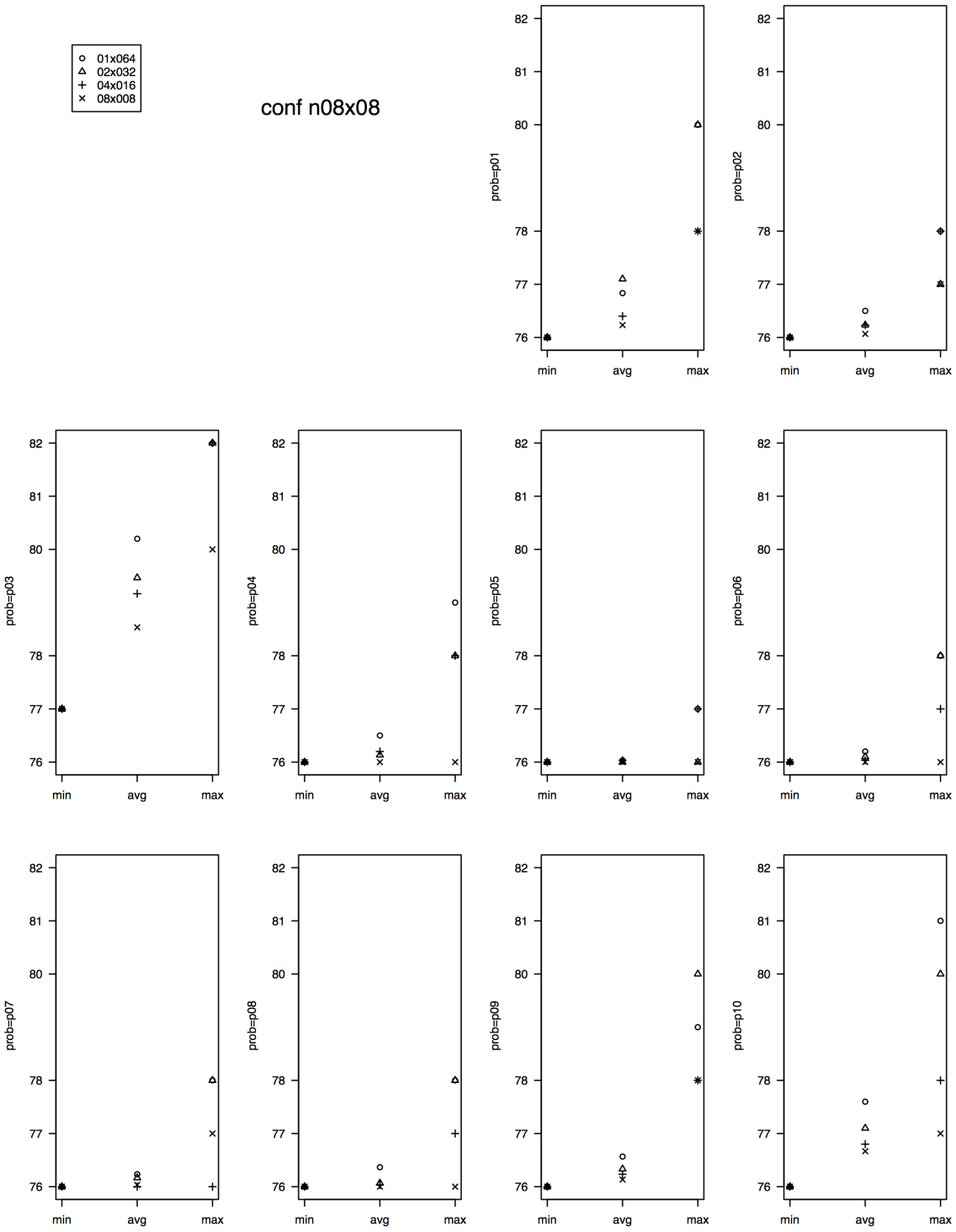

Figure 3.6: Average results of the MC-Ant configurations for each of the $n = 8 \times 8$ instances

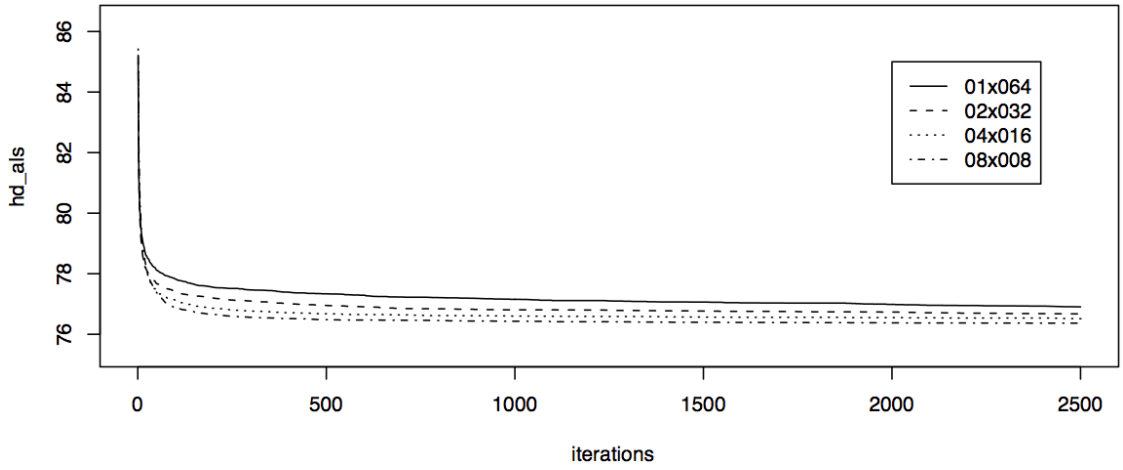

Figure 3.7: Evolution of the mean best fitness of the MC-Ant configurations for the $n = 8 \times 8$ set. Results are averaged over the several instances

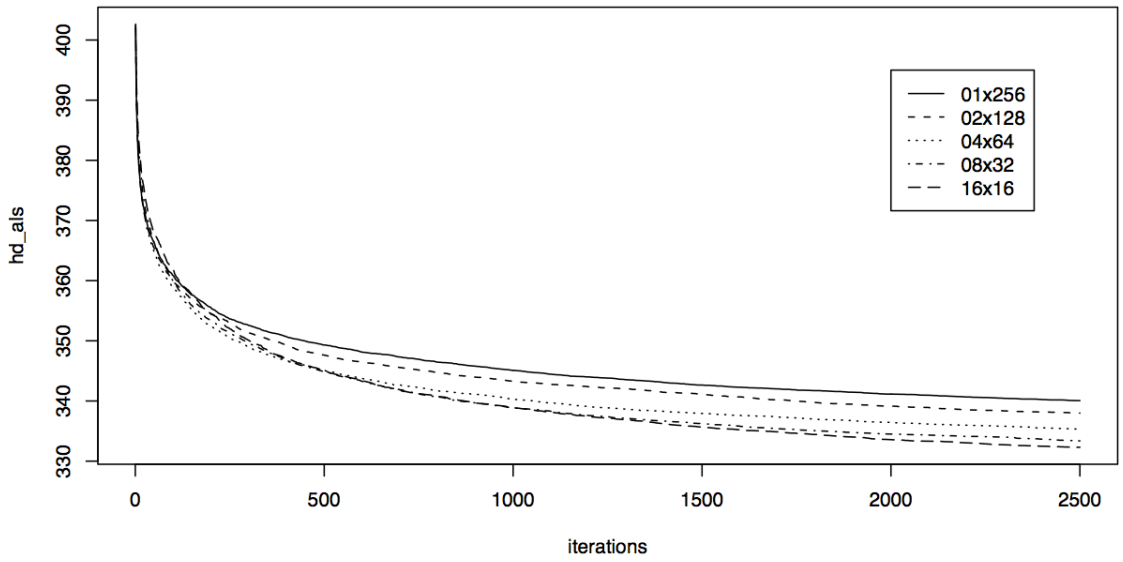

Figure 3.8: Evolution of the mean best fitness of the MC-Ant configurations for the $n = 16 \times 16$ set. Results are averaged over the several instances

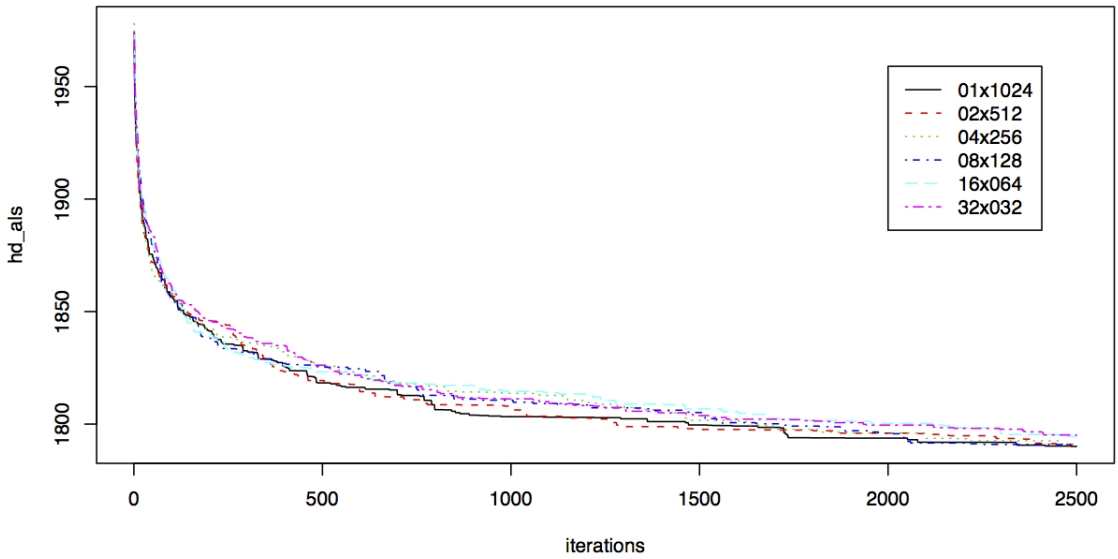

Figure 3.9: Evolution of the mean best fitness of the MC-Ant configurations for the $n = 32 \times 32$ instance

rations with more colonies, as can be observed by the more pronounced slope of the curves. Again, in the initial iterations configurations with less colonies achieve better results, while, in the latter stages, configurations with more colonies produce better solutions: up to iteration 11 configuration 01x256 is the best; from iteration 11 to 25 the best configuration is 02x128; from 26 to 399 the best configuration is 04x64; from iteration 27 to about 640 configuration 08x32 is the best; from iteration 640 to about 1040 the best configurations are 08x32 and 16x16, having almost identical results; from iteration 1040 forward configuration 16x16 is the best one.

In Figure 3.9 the pattern is less distinct but the configurations with up to 4 colonies are the ones that consistently achieve the best results, specially 01x1024 and 02x512, and, from 2000 iterations forward also 04x256. In line with the behaviour observed in smaller sized instances, we speculate that 2500 iterations may correspond, in instances of this size, to the initial stages of the search.

We also studied the migration rate. As expected, it tends to be more intense in the beginning of the run and then it slowly becomes less frequent, although it does not stop. Still, for instances where the optimal solution was harder to find, the frequency of migration remained high for longer, when compared with

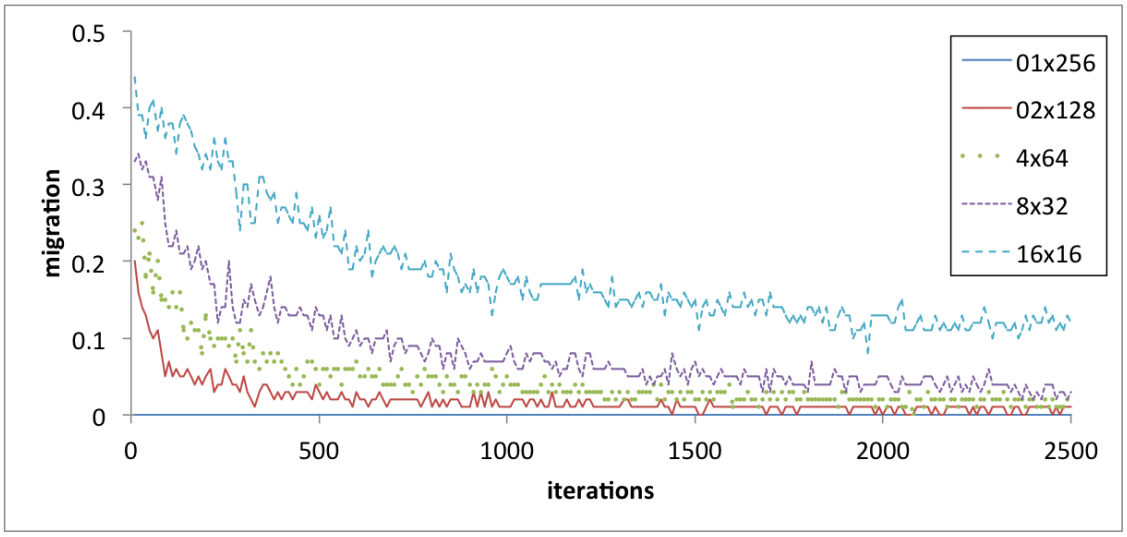

Figure 3.10: Evolution of the average migration rate of the MC-Ant configurations for the $n = 16 \times 16$ set

instances that were more easily solved. This is expected since, on easier instances, more colonies are expected to achieve good results and thus the difference between the quality of the solutions is small. Also, as expected, configurations with more colonies have a higher frequency of migration In figure 3.10 we can observe the average migration rate of the several configurations for the $n = 16 \times 16$ instance set.

In addition to the migration of solutions, the proposed architecture allows for the self-adaptation of parameters. For each run of a given instance, we recorded the value of the parameters when the best-ever solution, of that run, was found. This allowed us to determine a range ($r_{general}$) and an average value ($a_{general}$) for the parameters used when finding the best-ever solution of a given run. We then selected the subset of runs where the solution with the highest quality, for that instance, was found. We calculated the range (r_{best}) and the average value (a_{best}) of each parameter considering only those runs, i.e., the range and average value of the settings that generated the closer to optimal solutions.

An example is depicted in Figure 3.11 showing the values obtained by the configuration using 8 colonies of 32 ants each, on instance 01 of the 16×16 set. This is an example of a situation were the best solution was found by several

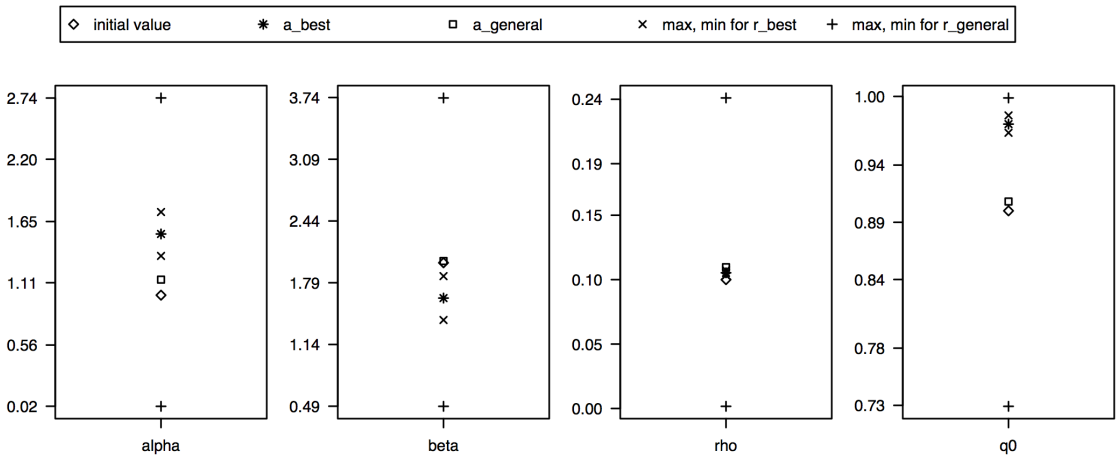

Figure 3.11: Parameters' ranges obtained by 08×032 configurations for instance 1 of the $n = 16 \times 16$ set

colonies in multiple runs. We can see that, for all parameters, r_{best} is narrower than $r_{general}$. Also a_{best} does not coincide neither with $a_{general}$ nor with the parameters initial value. This result suggests that the parameters have influence in the quality of the solution found and allowing for the parameters to adjust may improve the algorithm performance, particularly in situations where the optimal settings are not known in advance.

One important point to investigate is whether the migration is so intense that it leads to all the colonies converging to same path. In order to ascertain this we measured the evolution of the average distance between the trails of each pair of colonies. In Figures 3.12, 3.13 and 3.14 we present the average trail difference for the different instance sizes. Initially all colonies have the same trail and, as expected, in the early stages of the optimisation they become distinct. Then, results displayed in the charts reveal that the trails are able to remain separated until the end of the execution. In the smaller instances, after the initial rise in the distances, there is a slight decrease and then the curves remain stable as can be seen in Figure 3.12. We believe that the decrease is due to a more intense migration as soon as some colonies find high quality solutions. After some iterations all the colonies are able to find a very good solution and as such there is little alteration in the paths.

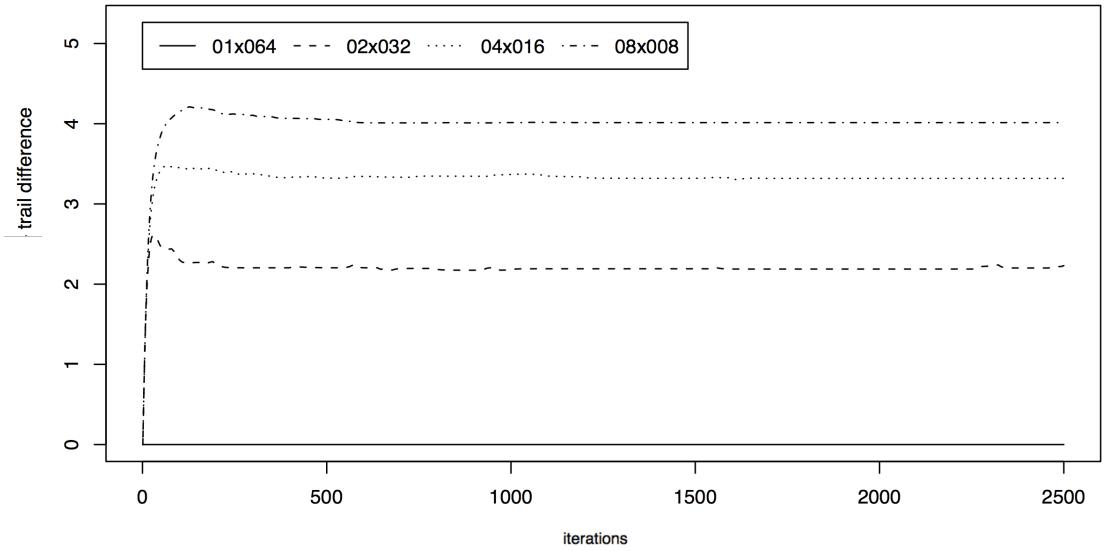

Figure 3.12: Average trail difference for the $n = 08 \times 08$ set

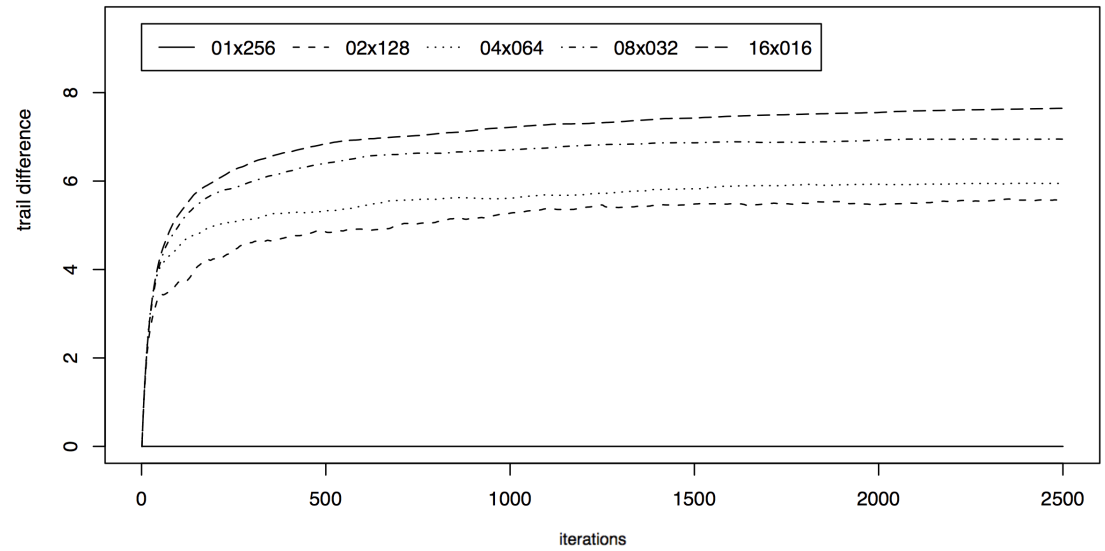

Figure 3.13: Average trail difference for the $n = 16 \times 16$ set

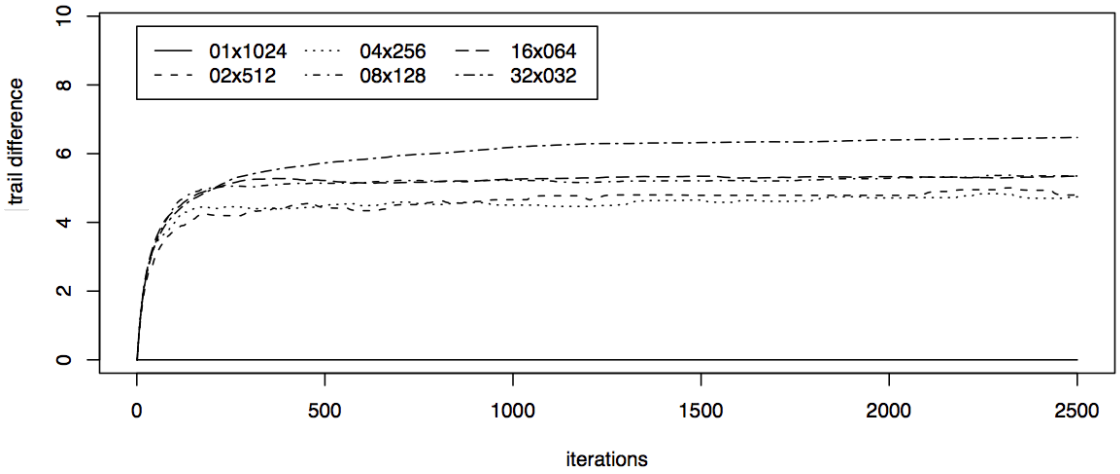

Figure 3.14: Average trail difference for the $n = 32 \times 32$ instance

3.5.1 Conclusion

The results show that the multi-colony configurations consistently outperform the single colony. For almost every instance, the mean best-fitness decreases as the number of colonies increases. The ideal number of colonies seems to depend on the number of iterations available. Also, the multi-colony configurations were able to avoid premature convergence, and by that reason, continue to improve at a higher rate in later stages of the search. This effect is more noticeable in configurations with more colonies. Multi-colony configurations have the additional advantage of self-adapting the parameters and, in doing so, avoiding the need to know the optimal setting beforehand. The migration flow behaved as expected, being stronger in the beginning, and in the configurations with more colonies, and gradually decreasing. Still the migration was gentle enough to allow for the trails to remain separated and thus avoid the convergence of the colonies to the same trail.

MC-Ant also has some less desirable traits. Having several trails means having several pheromone matrices. As the problem size increases, this clearly compromises efficiency. Also, allowing for β to differ from one population to another, indirectly implies using one heuristic matrix per population since, for efficiency reasons, what is really stored in the matrixes is ${\eta_{ij}}^{\beta}$ (refer to Equations 3.3 and 3.4, 3.5 and 3.6). Any modification of α and β immediately implies recomputing

the pheromone and heuristic matrices.

MC-Ant was devised to run on a single processor, but a possible way to deal with the computational cost would be to adapt the algorithm so it could be distributed across several processors. Another possibility, and the one we have chosen in this work, is to adapt the algorithm so it does not need several pheromone trails, and to carefully choose the parameters that can differ from population to population and self-adapt. The result of the adaptation is Multi-caste ACS, explained in Chapter 4.

4

Multi-caste ACS: the static case

Multi-caste ACS ([Melo et al., 2011,Melo et al., 2013b,Melo et al., 2014,Melo et al., 2013a]) can be considered as a more efficient version of MC-Ant. Multi-caste ACS is an heterogeneous, self-adapting, multi-population approach, like MC-Ant, but done in such a way as to require less computational effort. Additionally, its modus operandi is similar to the original ACS. From a user perspective, the application of Multi-caste ACS to a given problem is identical to the application of the original ACS.

One limitation of classical ACO is that it requires a set of parameters, often set by trial and error, that remain constant throughout the optimisation process. The idea behind Multi-caste ACS is to modify the base ACS, by adding self-adaptation capabilities to the parameters, so that the settings may change over time, according to the search stage. Also, the initial choice of parameters becomes less crucial, as the algorithm is able to autonomously adapt its behaviour to the search stage.

Another drawback of ACO is the inability to find new solutions once the algorithm converges. Multi-caste ACS is designed to keep a moderate diversity level, as a strategy to prevent premature convergence.

4.1 Multi-caste ACS architecture

4.1.1 Motivation

One of the less desirable features of MC-Ant is the amount of memory required, justified by each colony having its own trail. When an ant constructs a solution, at each decision point, it consults the trail to look for the solution component to add. Ideally, the trail should have a pheromone value for each component. However, this leads to an explosion in the memory space requirements, as hard problems tend to have a large amount of solution components, which are then multiplied by the number of colonies. A less demanding approach is to have all ants using the same trail. We compensate some of the diversity lost by having just one trail, by re-introducing the local pheromone update, (see Equation 2.6), present in the original ACS, and by allowing different q_0 values to co-exist.

4.1.2 The choice of q_0

When designing Multi-caste ACS the parameters allowed to self-adapt were revised, with the objective of keeping the computational effort as low as possible. Both α and β are computationally demanding, since, if we allowed one α or β for each sub-population, either we would have to re-compute every ${\tau_{ij}}^{\alpha}$ and ${\eta_{ij}}^{\beta}$ for each ant at each decision point (see Section 2.5), or, if relying in the usual optimisation technique of using a matrix that stores directly ${\tau_{ij}}^{\alpha}$ and ${\eta_{ij}}^{\beta}$, we would need one such matrix for each sub-population. For the sake of simplicity, in Multi-caste ACS only one parameter self-adapts, although the same idea could have been applied to some other, or a combination of others, parameters. q_0 is a natural candidate as it is considered one of the most influential parameters in ACS [Ridge, 2007, Gaertner, 2004, Dorigo and Stützle, 2004]. By varying q_0 we can have ants with clearly different search strategies, either more exploitative of the prevailing trails or more prone to explore new regions of the search space. Retaining the ability to explore is also very important to prevent premature convergence.

Instead of directly changing the q_0 value, Multi-caste ACS changes the proportion of ants using a given q_0 over time, according to the search results. The idea behind this approach is to favour either exploration or exploitation, according to

the search needs.

4.1.3 The architecture

In the Multi-caste ACS framework, the ants are divided in sub-populations, named castes. The term caste is inspired by the behaviour of biological ants. In many species, colonies are composed of several castes: queen, soldiers, scouters and drones. To the best of our knowledge, the term was first used for artificial ants in [Botee and Bonabeau, 1998]. Ants belonging to different castes have different q_0 values. When constructing the solution, each ant uses the q_0 value characteristic of its caste. The total number of ants remains constant, but the amount of ants in a given caste can change over time, depending on the migration strategy. The main Multi-caste ACS algorithm is presented in Algorithm 4.1.

```
Data: problem instance, parameters
Result: best-so-far solution
begin
    load the instance
    set general parameters
    initialise the pheromone trail
    foreach caste do set caste parameters
    while termination condition not met do
        construct ant solutions
        apply local search (optional)
        adjust castes' size
        update pheromone trail
    end
end
```

Algorithm 4.1: Multi-caste ACS algorithm

Like in ACS, during the *load the instance* step, the problem instance is loaded and used to build the heuristic information. Also, should it exist, the nearest neighbour list is computed (this list was proposed for ACS [Dorigo and Gambardella, 1997], and is used during the construct ant solution step).

Most of the parameters of Multi-caste ACS are common to all the colony and, during the *set the parameters* procedure, are set as usual in ACS. Additionally, for

each caste, we run *set caste parameters* procedure where the q_0 to be used by that caste and the initial amount of ants are set.

At the end of each iteration, in the *adjust castes' size* step, one of the ants may be selected to change the caste to which it belongs. The choice is probabilistic and it depends on the migration strategy and quality of the solutions produced.

Migration strategies

Two castes' size adjustment mechanisms were tested.

Const No adjustment is made. This is the simplest form, as the dimension of the castes remains constant throughout the optimisation. With this strategy we aim to determine if the simple co-existence of different q_0 values can be an advantage.

Jump Two ants are selected at random. If the ants belong to different castes, the quality of the solutions they built on the most recent iteration is compared; if one of the solutions is worse than the other, and the ant that created the worse solution comes from a caste with more than half of its original size, this ant is transferred to the caste of the other contender. The castes are able to increase or decrease in size, but no caste can become smaller than half its original size. This limit is imposed to prevent the extinction of a given caste, and to foster migration. Should one of the castes become much larger than all the others, it would be less likely that 2 random ants should belong to different castes and movement would eventually cease.

The two variants were devised to understand, not only if the simultaneous use of different castes would have an impact on the algorithm behaviour, but also if allowing for the castes size to adapt over time would be beneficial. In concrete, we aim to investigate how the migration policy would influence the behaviour of the algorithm when applied to problems of varying characteristics. Since we limit the minimum size of the castes, no caste risks disappearing. Also, indirectly, as the size of the colony remains constant, there is a limit to the maximum size of the castes, so no group can become overly dominant, and the diversity is not lost.

We can describe Multi-caste ACS as a multi-population ACO that adopts the following design options (using the nomenclature proposed in [Janson et al., 2005] where possible) [1]:

- There is no direct information exchange in Multi-caste ACS, but rather the physical migration of the ants from one caste to another. The topology of this migration depends on the variant.
- At each iteration, at most one ant will migrate to another caste. The winning caste is reinforced with one additional ant and, in the following iterations, the migrant will adopt the parameters of the group it was transferred to.
- The migration frequency is quality dependent. Migration only happens if the qualities of the compared solutions are different. It also depends on the size of the caste selected to loose one ant.
- The population in Multi-caste ACS is heterogeneous within an iteration, as each caste uses different settings. The trail is updated with the best solution, regardless of the caste to which it belongs.
- The population in Multi-caste ACS can also be considered heterogeneous between iterations, since the dimension of the castes can change over time, and, thus, so do the number of ants using a specific q_0 value.

4.2 Application of the Multi-caste ACS to the TSP

The Travelling Salesperson Problem (TSP) is a famous NP-hard combinatorial optimisation problem. It can be described as the problem faced by a salesperson who must visit a given set of cities exactly once and return home. Additionally, the salesperson wants to minimise the cost function, where the cost may represent, for instance, the tour length, or the time to complete the circuit. The TSP can be symmetric where for each pair of cities, i, j the cost of going from one city to the

[1] Two additional migration strategies will be presented in the next chapter. These design options are valid for all variants.

other is the same regardless of the direction, or asymmetric. Possible variations of the problem include: respecting a timeframe, multiple salespersons, multiple starting/ending points, additional constraints to the tours or salespersons.

4.2.1 Motivation

The TSP was the first problem addressed by ACO algorithms, both because it is a hard optimisation situation and it can be modelled in a suitable way for the exploration performed by artificial ants. Given a set of cities and all pairwise distances between them, the goal is to discover the shortest tour that visits every city exactly once.

We chose the TSP to conduct our experiments with Multi-caste ACS, not only because it is a problem with several practical applications, but also because it is frequently used as a testbed for new algorithms, where a good performance on the TSP is considered an indication of their usefulness [Dorigo and Stützle, 2004].

4.2.2 The Symmetric TSP

In the symmetric version of the problem, a specific TSP instance is represented by a fully connected undirected graph $G = (V, E)$, where V is the set of nodes representing the cities, and E is the set of edges representing the roads that connect each pair of cities. Each edge from E has a distance, e_{ij}, associated. The objective is to find the Hamiltonian cycle of minimal total cost of the graph.

Let a path, π, be represented as permutation of the nodes indices, $\{1, 2, ..., n\}$, and let $f(\pi)$, given by Equation 4.1, represent the length of path π,

$$f(\pi) = \left(\sum_{i=1}^{n-1} e_{\pi(i)\pi(i+1)} \right) + e_{\pi(n)\pi(1)} \tag{4.1}$$

An optimal solution to the TSP is a path, π, such that $f(\pi)$ is minimal.

In Euclidean TSP instances, we are usually provided with the 2-dimensional coordinates of each city, and e_{ij} is determined as the straight line distance between i and j. As such, e_{ij} can be calculated as indicated in equation 4.2, being (x_i, y_i) and (x_j, y_j) the coordinates of cities i and j, respectively.

$$e_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}. \tag{4.2}$$

4.2.3 Application of ACO to the TSP

To apply an ACO algorithm to a problem, one must first define the solution components. A connected graph is then created by associating each component with a vertex and creating edges to link the vertices. The TSP representation previously defined immediately, establishes the graph where the ants will operate.

The pheromone value, τ_{ij}, represents the desirability of visiting city j immediately after city i. The higher the value, the more attractive that edge is for the ants. Associated with each edge there is also an heuristic value, η_{ij}, that represents the attractiveness of that edge from a greedy point of view. In this problem, the desirability of visiting city j directly after city i is inversely proportional to the distance between the cities, i.e., $\eta_{ij} = \frac{1}{e_{ij}}$.

Ants start building a solution in a random vertex and iteratively add components by following a specific edge. At each decision point, an ant makes a probabilistic choice of the next city to visit, considering only cities that were not visited before. The choice is biased by the pheromone level and heuristic knowledge of each possible edge.

4.3 Experiments

We used the publicly available ACOTSP software [Stützle, 2002], both to obtain the standard ACS results reported here, and as the base for our own implementation of Multi-caste ACS.

We run our experiments on several symmetric euclidean TSP instances. We selected seven instances from the TSPLIB 95 [Reinhelt, 1995], for which the optimum is known.We chose instances of varying size because, even though the structure of the TSP instance is vital to reveal how hard it is for optimisation, the size of the search space also impacts the behaviour of the algorithm [Fischer et al., 2005, Ridge and Kudenko, 2008]. The size, name and optimum of each of the TSP instances used can be found on Table 4.1.

Size	Name	Optimum
99	rat99	1211
198	d198	15780
417	fl417	11861
783	rat783	8806
1577	fl1577	22249
3038	pcb3038	137694
5934	rl5934	556045

Table 4.1: TSP instances size and optimum

Parameter	Symbol	Value
trail relative importance	α	1
heuristic information relative importance	β	2
number of ants	m	10
candidate list length	nn	20
global pheromone decay	ρ	0.1
initial pheromone value	τ_0	$1/(n \cdot L_{nn})$
local pheromone decay	ξ	0.1

Table 4.2: Parameter values used

Thirty runs were performed for each combination of problem instance and algorithmic configuration. In order to show the performance of the several configurations over time, the results were averaged. Unless otherwise noted, the settings recommended in [Dorigo and Stützle, 2004] and stated on Table 4.2 were used. On Table 4.2, n, is the size of the instance, i.e. the number of cities, and L_{nn} is the length of the tour found using the nearest neighbour heuristic [Stützle, 2002,Dorigo and Stützle, 2004].

To gain a better insight into the influence of the proposed framework, experiments were first conducted without local search. However, and as ACO is often combined with local optimisation, for completeness the experiments were later repeated using local search. In this case, local search was applied to all ants. We selected the 3-opt algorithm [Lin, 1965] since, considering the improvement strategies available in the software used ([Stützle, 2002]), it is the one that usually obtains the best performance [Johnson and McGeoch, 1997] on the symmetric TSP. When using 3-opt, 2 tours are considered neighbours if they differ, at most,

castes	ants per caste	q_0	configuration
1	10	0.75	c75
		0.90	c90
		0.95	c95
		0.99	c99

Table 4.3: ACS Configurations used in the TSP experiments

in 3 edges.

While conducting our study we divided the configurations in 3 groups, according to the number of castes. In Table 4.3 we have the regular ACS configurations, which correspond to having a single caste of 10 ants each. The ACS configurations differ only on the selected q_0 value. The dual-caste configurations have two castes, with an initial composition of 5 ants each, and can be consulted in Table 4.4. One of the castes has a q_0 value that ranges from 0.50 to 0.95, and the other caste has a different q_0 value that can go from 0.90 to 0.99. Quad-caste configurations are depicted in Table 4.5. They comprise four castes, three of those with q_0 values of 0.90, 0.95 and 0.99. The other caste has a q_0 value of 0.50 or 0.75. Three settings are possible for the initial number of ants per caste, 2, 3, or 5, so the colony may have a total of 8, 12 or 20 ants.

4.4 Results

We start by ascertaining the impact of the q_0 parameter on the performance of the conventional ACS. To do so, we recorded the performance, for the configurations in Table 4.3, that correspond to conventional ACS with different values for q_0, namely, $q_0 = \{0.75, 0.9, 0.95, 0.99\}$. Afterwards we investigate the impact of both Multi-caste ACS update strategies and the influence of the different q_0 values on the dual-caste configurations. We used the settings indicated in Table 4.4. We also study the impact of both update strategies, initial size of the colonies and q_0 values in the quad-caste configurations (see Table 4.5). In all quad-caste configurations studied in this work, three of the castes have q_0 values of 0.90, 0.95 and 0.99. Then, only the variation on the q_0 value of the remaining caste was worth studying.

castes	ants per caste	update strategy	lower q_0	higher q_0	configuration
2	5	const	0.50	0.90	c50_90
				0.95	c50_95
				0.99	c50_99
			0.75	0.90	c75_90
				0.95	c75_95
				0.99	c75_99
			0.90	0.95	c90_95
				0.99	c90_99
			0.95	0.99	c95_99
		jump	0.50	0.90	j50_90
				0.95	j50_95
				0.99	j50_99
			0.75	0.90	j75_90
				0.95	j75_95
				0.99	j75_99
			0.90	0.95	j90_95
				0.99	j90_99
			0.95	0.99	j95_99

Table 4.4: Dual-caste configurations used in the TSP experiments

castes	common q_0	update strategy	lower q_0	ants per caste	configuration
4	0.90, 0.95, 0.99	const	0.50	2	c50quads08
				3	c50quads12
				5	c50quads20
			0.75	2	c75quads08
				3	c75quads12
				5	c75quads20
		jump	0.50	2	j50quads08
				3	j50quads12
				5	j50quads20
			0.75	2	j75quads08
				3	j75quads12
				5	j75quads20

Table 4.5: Quad-caste configurations used in the TSP experiments

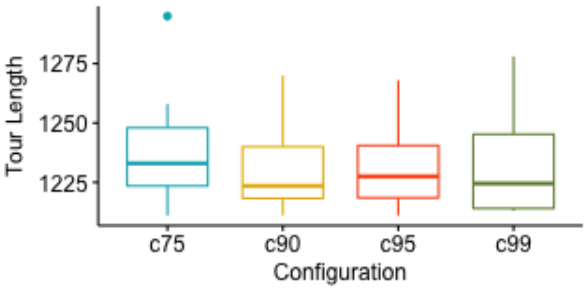

Figure 4.1: ACS configurations for the rat99 instance, without local search

The analysis is mostly empirical, based on comparative search performance box-plots and charts displaying the evolution of quality over time. We complement and support the empirical analysis with statistical tests that relate the outcomes achieved by the two migration strategies and that compare the effectiveness of selected Multi-caste ACS configurations with standard ACS variants.

4.4.1 Regular ACS: the q_0 influence

One important question when using ACS is to determine the relationship between the q_0 value and the outcome of the algorithm, specifically the tour length. We selected the range of values usually found in the literature for q_0: in most cases they belong to the 0.90-0.99 range. Since this is a minimisation problem, lower values for the tour length mean better results. The thick central bar of the box-plots marks the median.

ACS solution quality: without local search

The box-plots depicted in this section refer to the tour length found by the algorithm without using local search. Regardless of the problem instance, each repetition was allowed to run for 10000 iterations.

The results for the rat99 instance can be consulted in Figure 4.1. There is no clear relation between the q_0 value and the performance. c75 seems to be slightly worst, in a small degree, from the rest. This result is expected, given the small size of the instance and the number of iterations allowed.

On the d198 instance (Figure 4.2), the results seem to follow a curve that

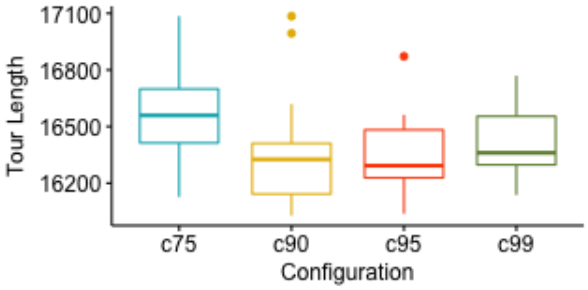

Figure 4.2: ACS configurations for the d198 instance, without local search

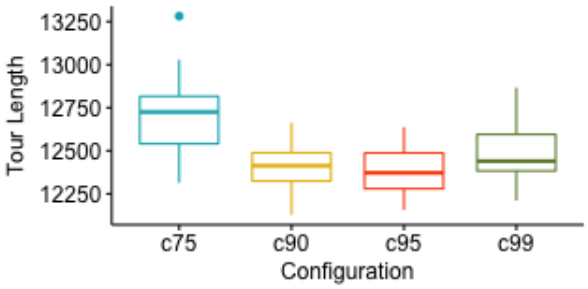

Figure 4.3: ACS configurations for the fl417 instance, without local search

has its minimum around 0.90. Given the relative small size of the instance, and since the number of iterations is sufficiently large, the q_0 of 0.90 provides a good compromise between exploitation and exploration. It is worth noting that a q_0 of 0.90 and 0.95 have a few outliers, suggesting that these configurations could benefit from a little more exploitation.

Figure 4.3 shows the results for instance fl417. Again, the relationship between the tour length and q_0 seems to follow a parabola with the minimum around 0.90 or 0.95. A lower q_0 is clearly detrimental.

When solving instance rat783 (Figure 4.4), configuration c75 is clearly worse than the other ones. Among the c90, c99 and c99 configurations, the relationship between the q_0 and the tour length follows a slight curve with the minimum around c95.

Figure 4.5 shows the results for instance fl1577. The c75 performance is once again clearly worse than all the others. The relationship between the tour length and q_0 seems to follow a curve with the minimum somewhere between 0.90 and 0.95. A more extreme value of 0.99 is less beneficial than the previous values.

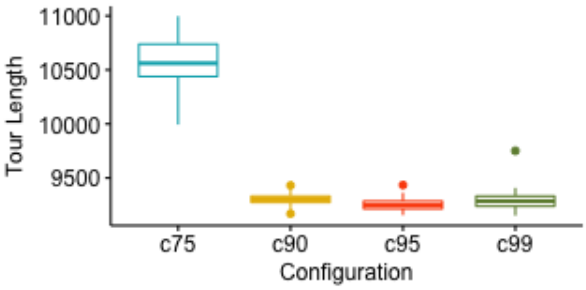

Figure 4.4: ACS configurations for the rat783 instance, without local search

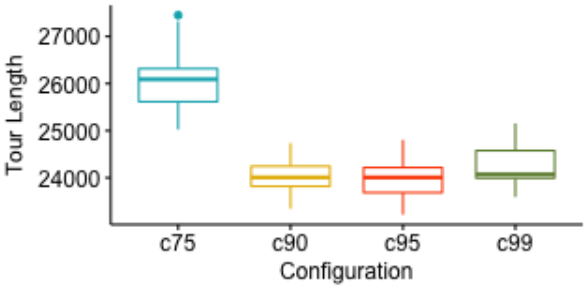

Figure 4.5: ACS configurations for the fl1577 instance, without local search

The performance of the ACS configurations when solving the larger instances pcb3038 and rl5934 is quite similar, as can be observed in Figures 4.6 and 4.7. For both instances, the tour length decreases as the q_0 increases in a slight, but evident, curve.

In the ACS configurations without local search, it was observed that the relationship between the q_0 and the tour length resembles a parabola. For the smaller instances, the configurations that reached smaller average tour length tended to

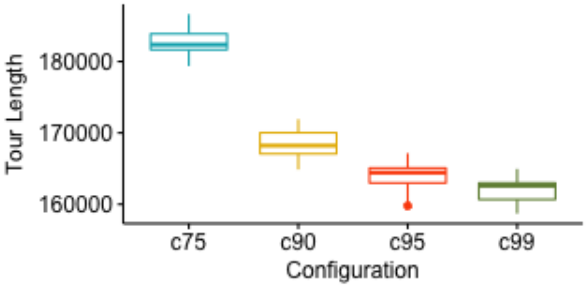

Figure 4.6: ACS configurations for the pcb3038 instance, without local search

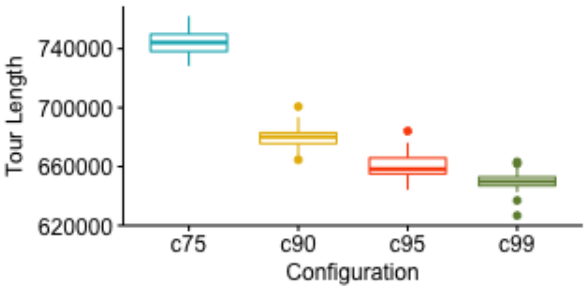

Figure 4.7: ACS configurations for the rl5934 instance, without local search

be the ones with a moderate q_0, whilst the advantage of a higher q_0 increased with the size of the instance.

We recall that the number of iterations granted to the algorithm was the same regardless the the instance. This can help to explain the results, as a more exploratory behaviour (corresponding to a lower q_0) is particularly valuable when there is enough time to sample the search space. A higher q_0 is advantageous in situations when the number of iterations is not enough to fully explore the search space. This corresponds to the larger instances of this study, when there are comparatively less iterations granted to the algorithm. In this case, the best strategy is to quickly exploit the promising trails early discovered, as they can lead to reasonably good solutions fast.

ACS solution quality: with local search

In this section we look at the tour length of the solutions found on the various instances, while using local search. Regardless of the instance, each repetition was allowed to run for 50 iterations.

While solving instance rat99 using local search, all configurations, with the exception of c99, are able reach the optimum in every run. c99 is unable to find the optimum 14 times out of 30 tries. The results can be consulted in Figure 4.8.

The results achieved while solving instances d198 and fl417 can be consulted in Figures 4.9 and 4.10, respectively. In neither of the instances there is a clear relationship between the q_0 value and the tour length. In instance d198, c90 appears to be the most successful configuration, despite several existing outliers. In instance fl417, all configurations have a similar performance.

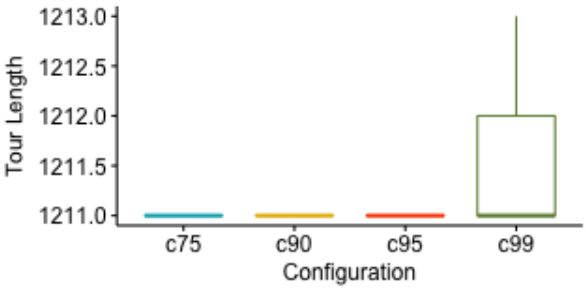

Figure 4.8: ACS configurations for the rat99 instance, with local search

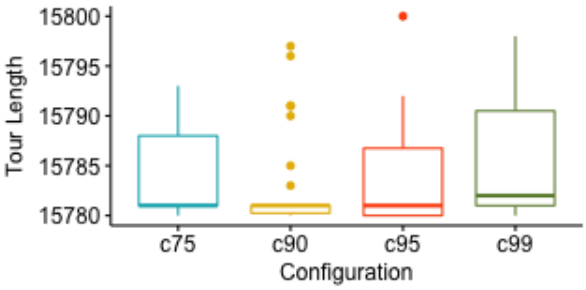

Figure 4.9: ACS configurations for the d198 instance, with local search

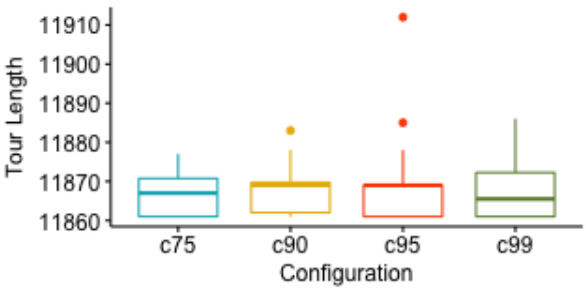

Figure 4.10: ACS configurations for the fl417 instance, with local search

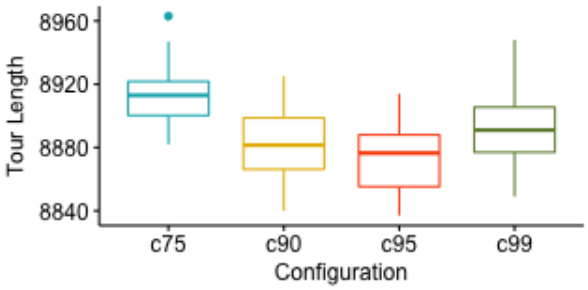

Figure 4.11: ACS configurations for the rat783 instance, with local search

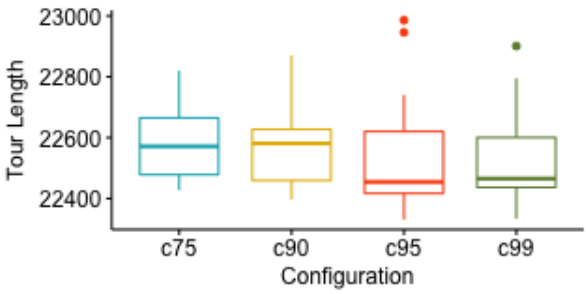

Figure 4.12: ACS configurations for the fl1577 instance, with local search

The relationship between the tour length and the q_0 when local search is combined with ACS to solve the rat783 instance follows a parabola, that reaches the minimum around the q_0 value of 0.95. The results can be observed in Figure 4.11. As can be seen in Figure 4.12, for instance fl1577 the relationship between the tour length and the q_0 value is once again not marked.

The results for instance pcb3038 can be observed in Figure 4.13. It is quite clear that the performance increases directly with the q_0 value, and the improvement is most apparent for larger values of q_0. Observing the results for instance rl5934 with local search in Figure 4.14, again we can see an improvement in performance as the q_0 increases, but while the difference between c75 to c95 is clear, configurations c95 and c99 achieve an almost identical performance.

When applying local search the overall results improve as expected, but the q_0 influence is less noticeable. This is expected, as much of the optimisation effort is taken by a specialised local algorithm for this problem. Still, a variation in the optimal q_0 setting was observed, favouring lower q_0 values in smaller instances and higher q_0 values in larger ones. Once more, we allowed the same number of

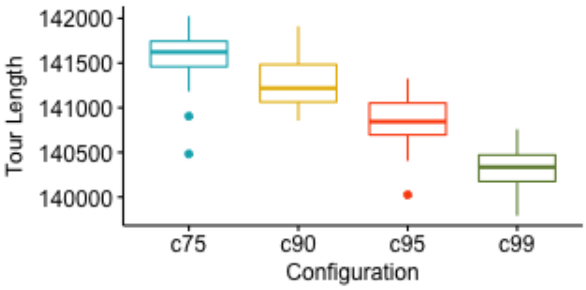

Figure 4.13: ACS configurations for the pcb3038 instance, with local search

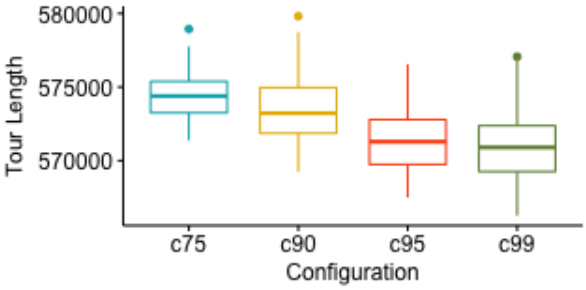

Figure 4.14: ACS configurations for the rl5934 instance, with local search

iterations regardless the size of the instance.

4.4.2 Regular ACS: Solution quality over time

To better understand the q_0 value influence on the various search stages we are also interested in the performance over time, measured as the evolution of the best-ever solution over iterations. Values were taken for the different values of $q_0 = \{0.75, 0.9, 0.95, 0.99\}$.

ACS solution quality over time: without local search

The outcome of ACS without local search can be observed in Figures 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, and 4.21. On the horizontal axis we have the number of evaluations (since there are 10 ants per colony, 1 iteration equals 10 evaluations). The vertical axis displays the relative error measured as the distance to the optimum. Given the total number of iterations and the fact that, particularly in the smaller instances, variation was most visible in the first iterations, we depict

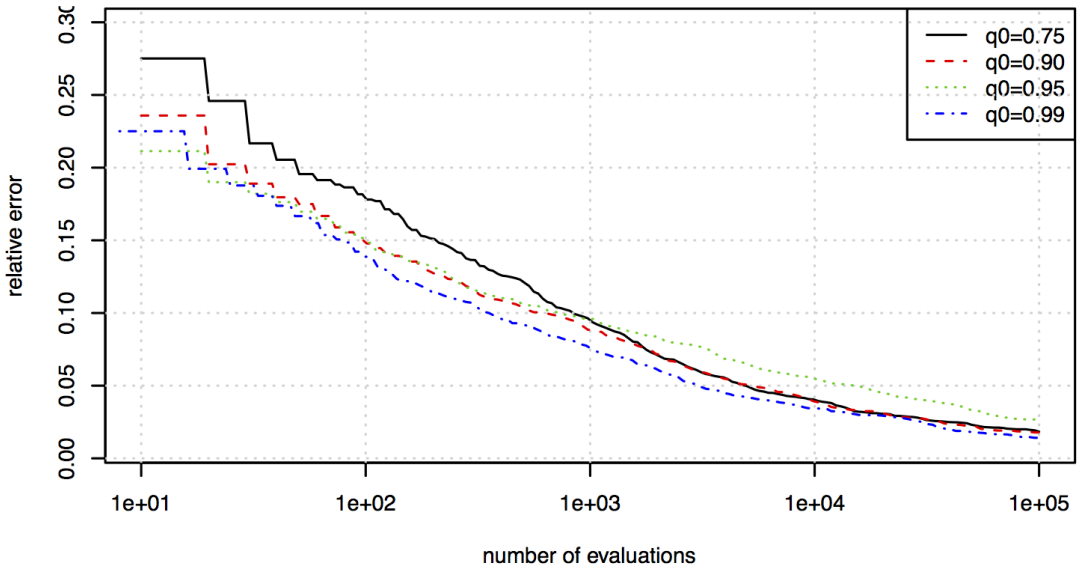

Figure 4.15: Solution quality over time for the rat 99 instance, without local search

the evaluations in a logarithmic scale. Also, the range of the relative error varies according to the instance.

In the small instances, the best performing q_0 changes over time. For example, as can be seen in Figures 4.15, 4.16 and 4.17, a $q_0 = 0.75$ is clearly outperformed in the first iterations, but it becomes more competitive in the later ones; on the contrary, a $q_0 = 0.95$ is very good in the initial phase of the search, but comparatively worse later.

In the larger instances, the relative performance does not change as much, but the differences are more marked. Still, the general conclusion is that the best parameter values depend on the specific stage of the search and, in most cases, a $q_0 = 0.95$ is preferable in the first iterations, while a $q_0 = 0.99$ is better later on.

ACS solution quality over time: with local search

For the tests performed with local search, the evolution of the quality over time is depicted in Figures 4.22, 4.23, 4.24, 4.25, 4.26, 4.27, and 4.28.

When using local search, the ideal q_0 depends both on the stage of the search and the problem instance. For example, in instance fl417 (Figure 4.24), low q_0 values (0.75 or 0.9) are ideal in the early stages of the optimisation, but, these

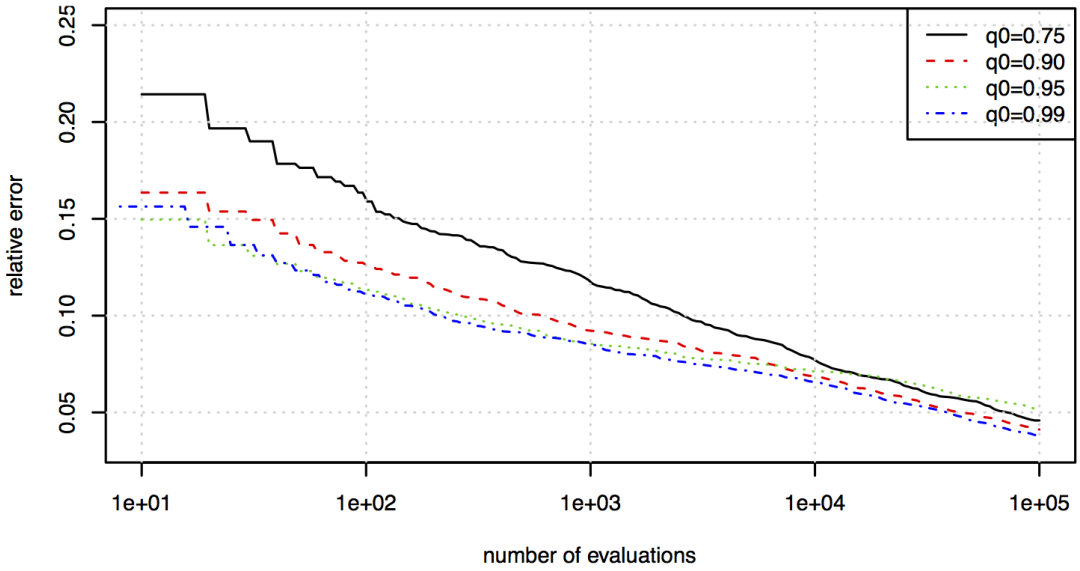

Figure 4.16: Solution quality over time for the d198 instance, without local search

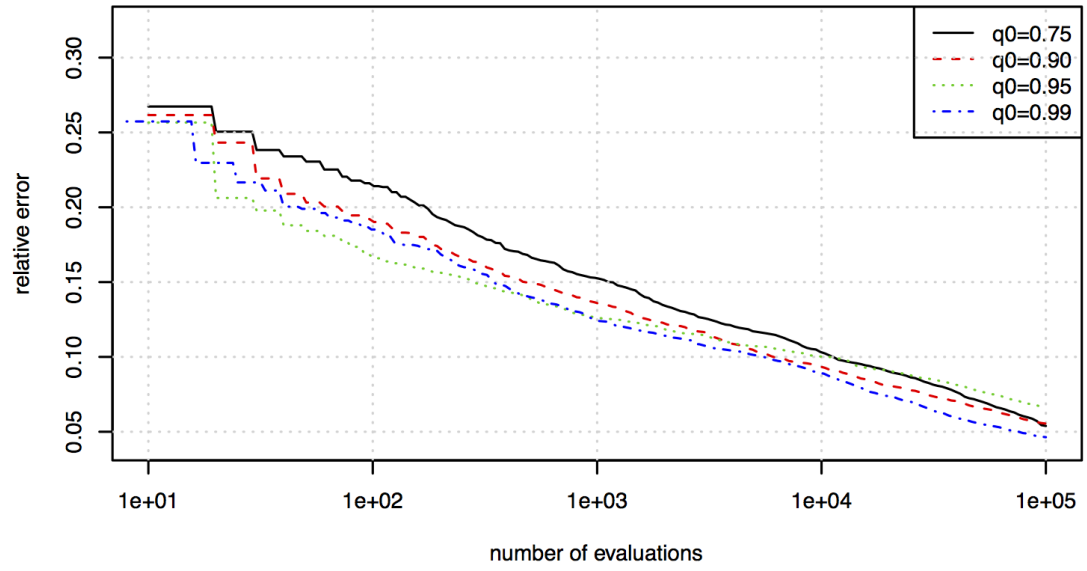

Figure 4.17: Solution quality over time for the fl417 instance, without local search

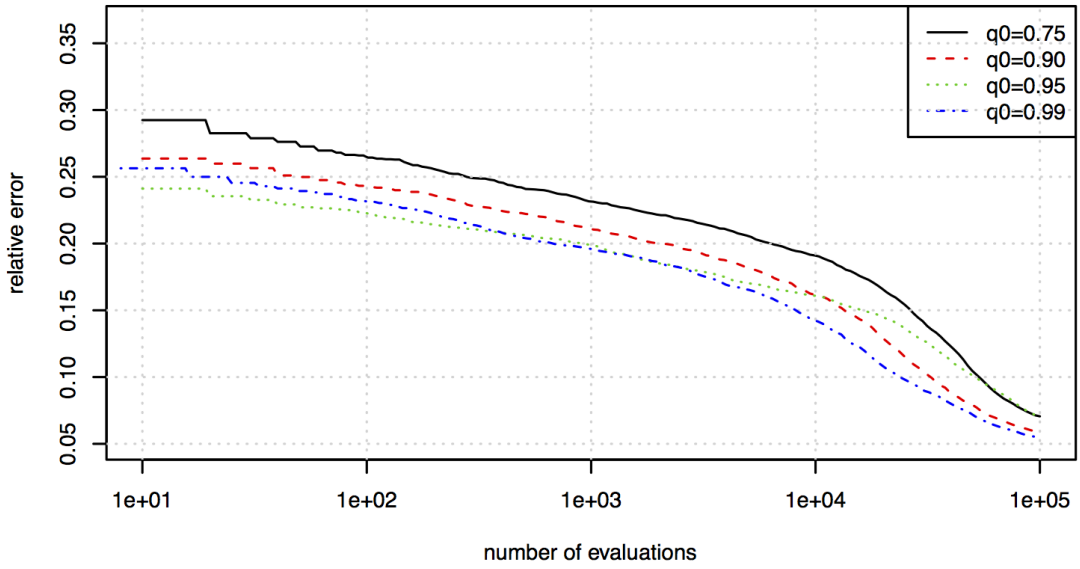

Figure 4.18: Solution quality over time for the rat783 instance, without local search

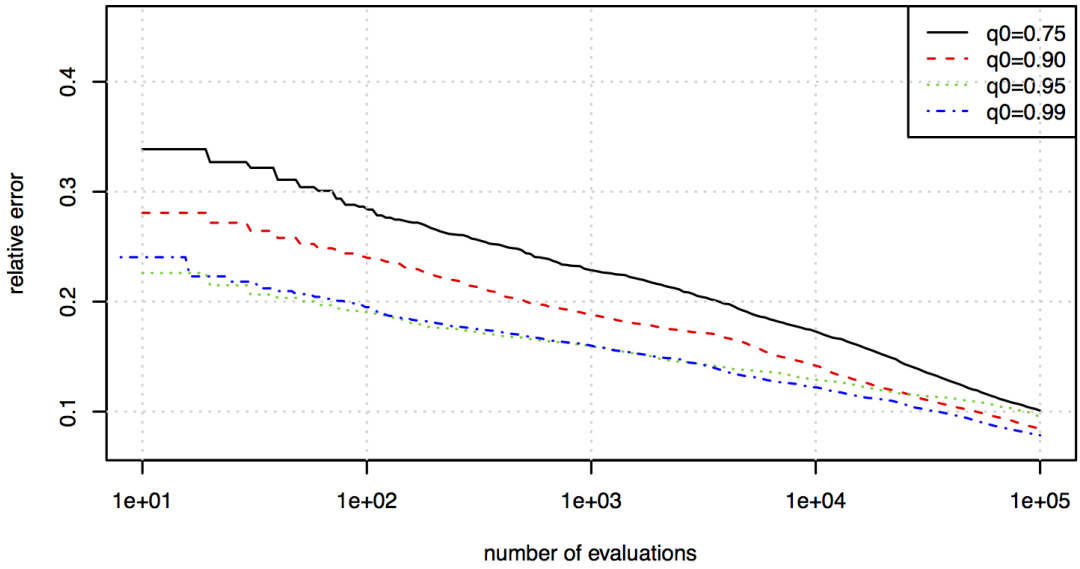

Figure 4.19: Solution quality over time for the fl1577 instance, without local search

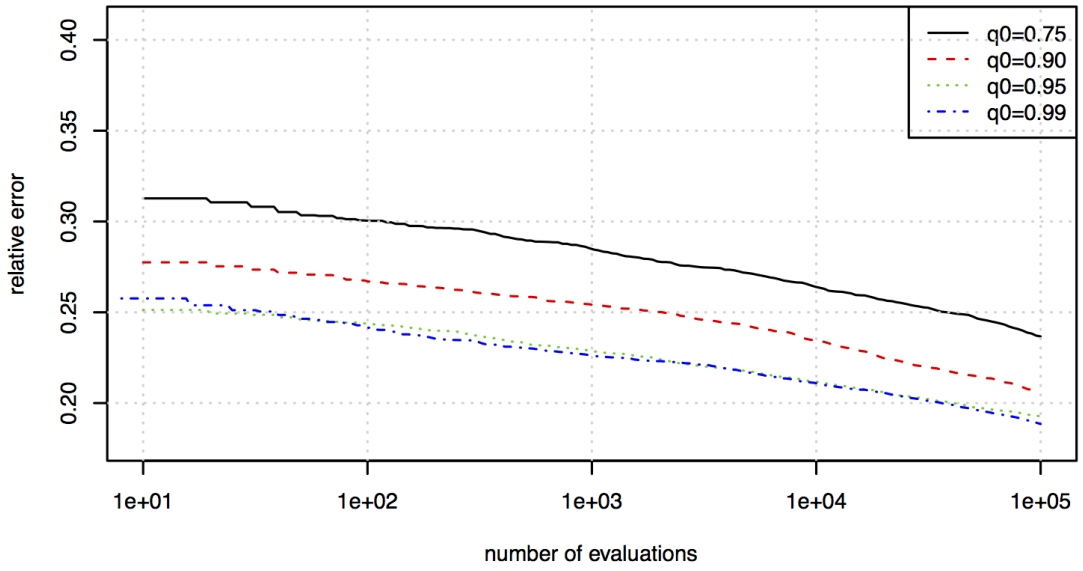

Figure 4.20: Solution quality over time for the pcb3038 instance, without local search

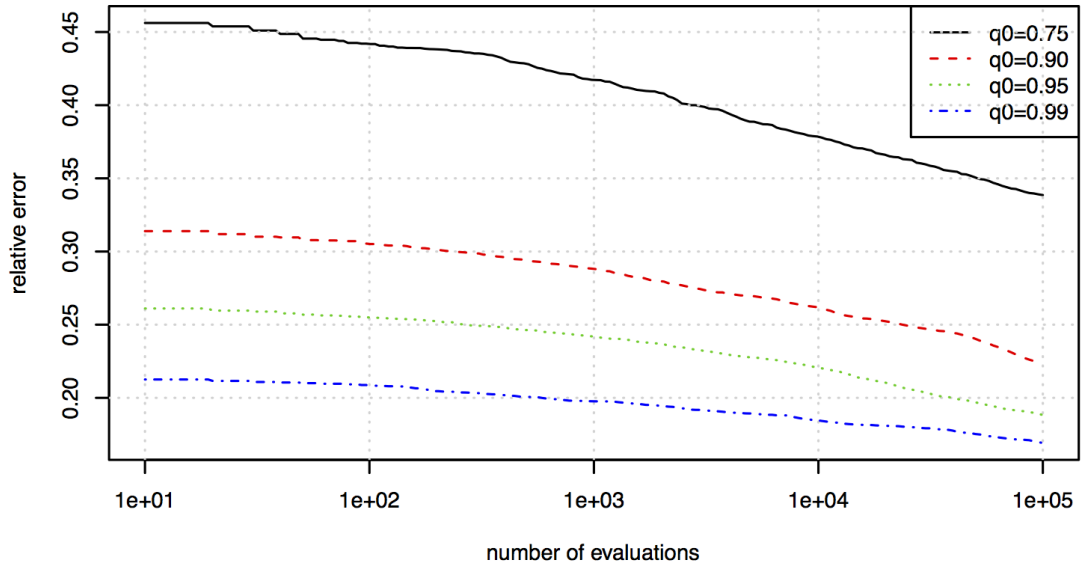

Figure 4.21: Solution quality over time for the rl5934 instance, without local search

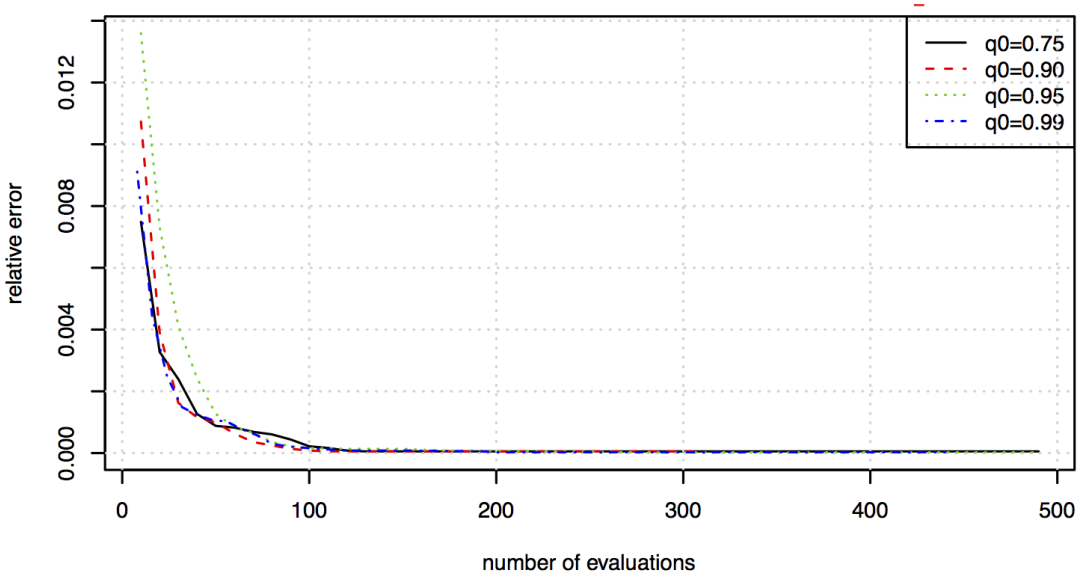

Figure 4.22: Solution quality over time for the rat 99 instance, with local search

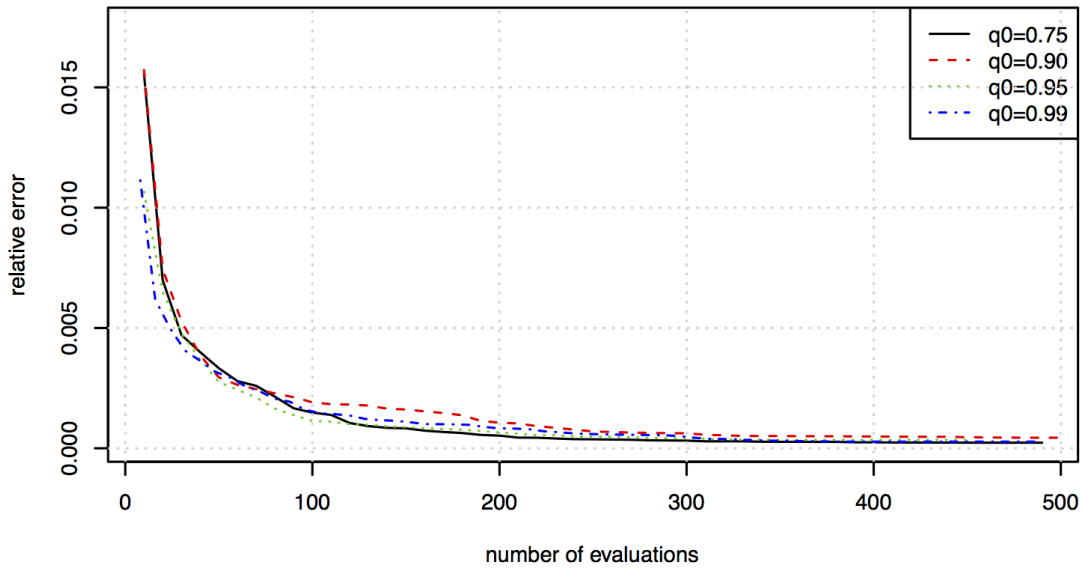

Figure 4.23: Solution quality over time for the d198 instance, with local search

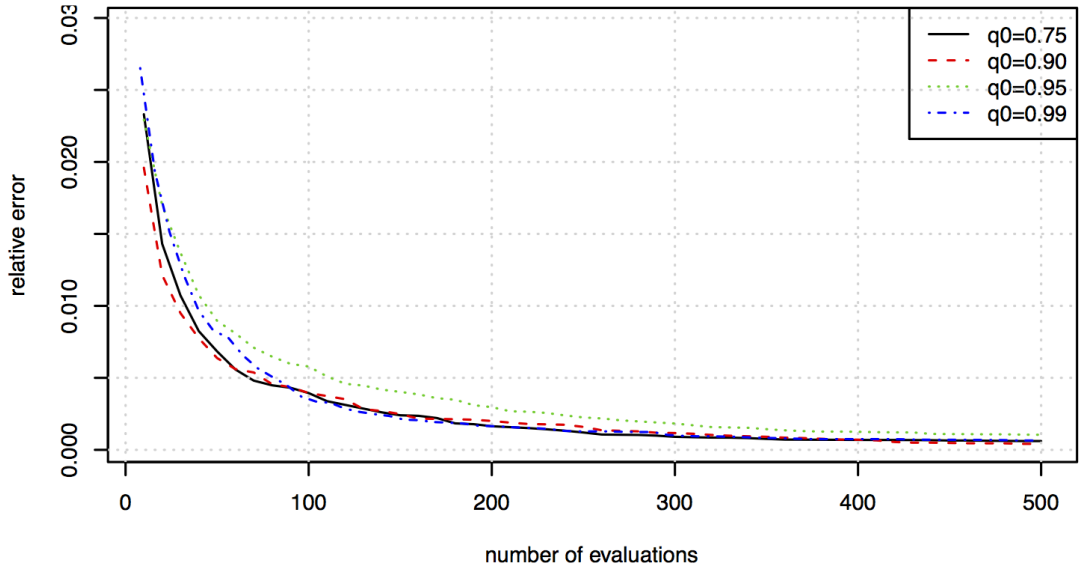

Figure 4.24: Solution quality over time for the fl417 instance, with local search

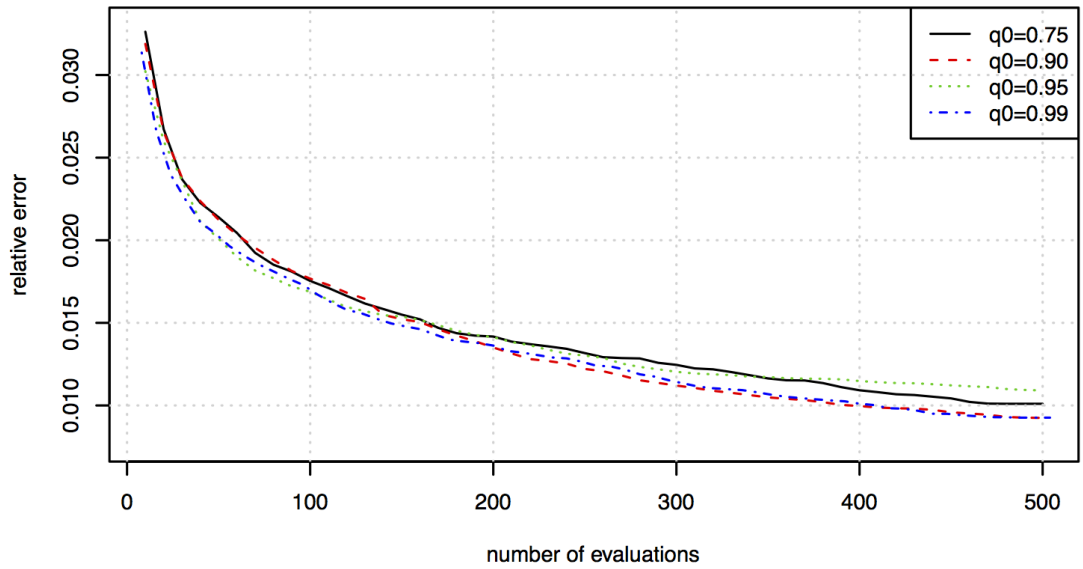

Figure 4.25: Solution quality over time for the rat783 instance, with local search

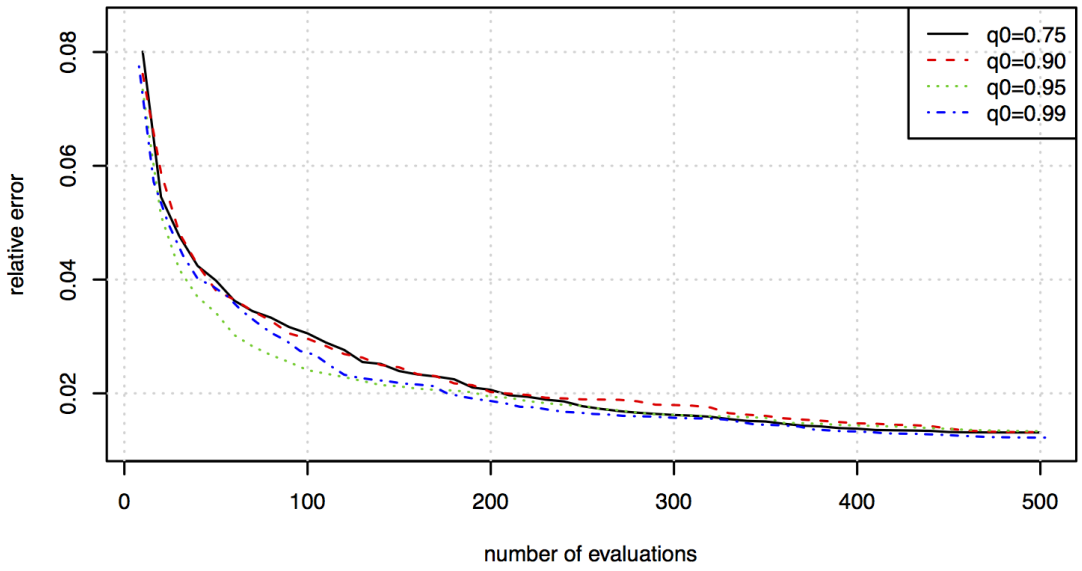

Figure 4.26: Solution quality over time for the fl1577 instance, with local search

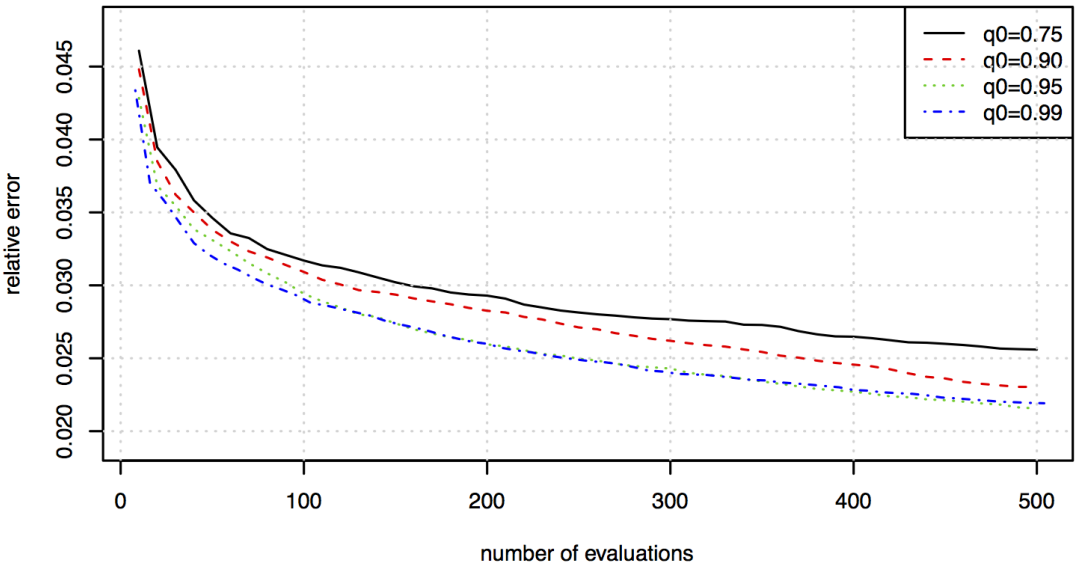

Figure 4.27: Solution quality over time for the pcb3038 instance, with local search

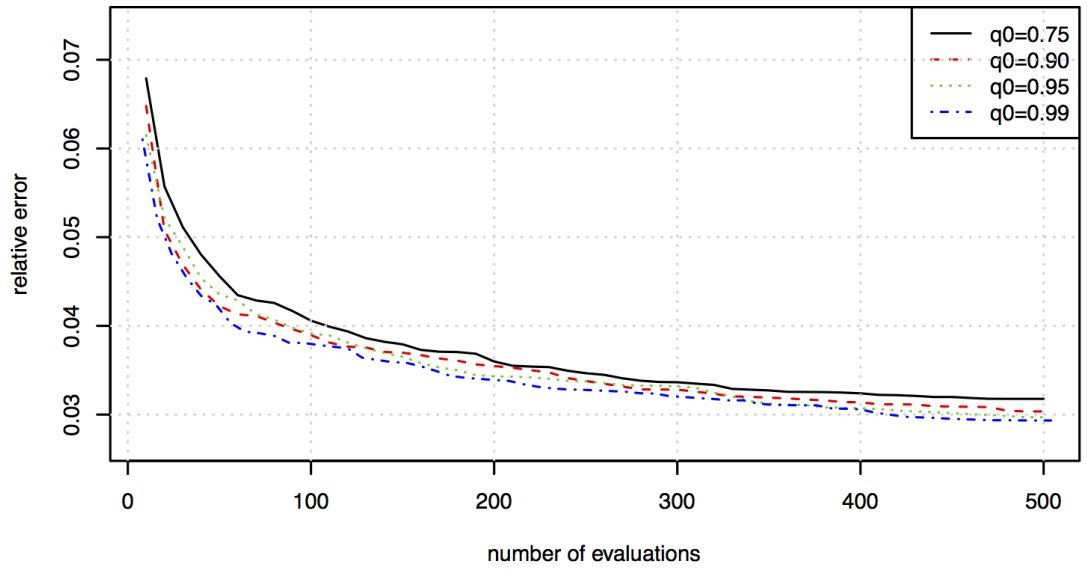

Figure 4.28: Solution quality over time for the rl5934 instance, with local search

same q_0 values used in the first stages of instances fl1577 or pcb3038 (Figures 4.26 and 4.27 respectively) obtain a comparatively poor result. In a larger instance, rl5934 (Figure 4.28), a higher q_0 tends to yield better results, particularly in the later stages of the search.

4.4.3 Dual-caste configurations: solution quality

In configurations with two castes, three factors can potentially influence the outcome of the algorithm: the q_0 of each of the castes and the update strategy (consult Table 4.4 for reference). The q_0 values of each of the castes will be referred to as the "lowerQ0" and "higherQ0". To simplify the reading of the horizontal axis in the box-plots, we present the q_0 values multiplied by 100, so a lowerQ0 of 50, means that $q_0 = 0.50$. Once again we look separately at the results achieved using local search from the ones that do not use it.

Dual-caste solution quality: without local search

The box-plots depicted in this section refer to the length of the tour found by the algorithm without using local search. Regardless of the instance, each repetition

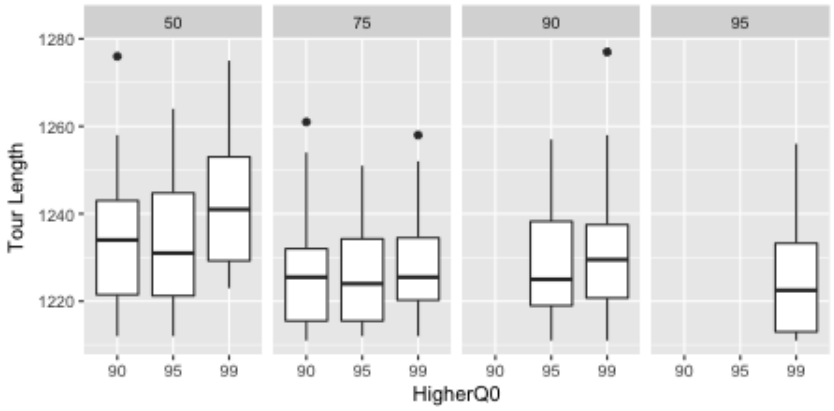

Figure 4.29: Instance rat99, without local search, tour length, by lowerQ0, const update strategy

was allowed to run for 10000 iterations.

When comparing the average tour length obtained by the const update strategy against the jump update strategy, we considered the samples related, since they both use the same q_0 values (e.g., the first 30 points of each strategy use 0.50 and 0.90, the next 30 points use 0.5 and 0.95, and so on). As such, and since the results discussed in this section present significant deviations from normality, we selected the Wilcoxon signed rank test with continuity correction, with a significance level of 0.05, as provided by the 'stats' package of the [R, 2011] software. For all comparisons reported, the null hypothesis states that the tour lengths obtained using either of the update strategies have the same median.

rat99 instance In Figure 4.29, each pane refers to a fixed lowerQ0 value, and the box-plots in that given pane correspond to the different higherQ0 values. For example, in the leftmost pane we can observe the results of the configurations c50_90, c50_95, and c50_99. Figure 4.29 depicts the results achieved by the const update strategy, on the rat99 instance. As a rule, the tour length tends to decrease as the lowerQ0 increases and, for a given lowerQ0, the results tend to improve with smaller values of higherQ0.

The results achieved using the jump update strategy can be consulted in Figure 4.30. The jump update strategy makes the algorithm less sensitive to the q_0 values used and, as such, the pattern between the tour length and this parameter is less

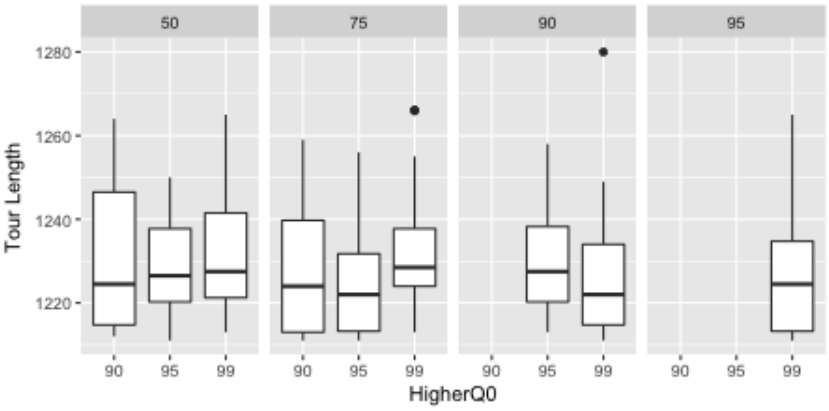

Figure 4.30: Instance rat99, without local search, tour length, by lowerQ0, jump update strategy

clear. This different sensitivity to the q_0 values according to the update strategy can also be observed in Figure 4.31, where we have a scatter plot of the results achieved by the various configurations, using const (in green) and jump (in violet) update strategies. Axis x and y are the lowerQ0 and higherQ0 values respectively, while axis z is the tour length. We then super-impose the fit of a regression plane (of general formula $tourlength = a + b_1 * lowerQ0 + b_2 * higherQ0$) to the solutions produced by each of the update strategies. We can observe that the fitting achieved by the jump results is almost parallel to the xy plane, and upon investigation, the regression was non-significant. The fitting of the results achieved using the const update strategy changes with the values of lowerQ0 and higherQ0, and tour lengths tend to decrease when using a larger lowerQ0 and smaller higherQ0 (the regression was significant and adequate to the sample, but with low predictive power).

The application of the Wilcoxon signed rank test (significance level = 0.05) revealed that the difference between the results obtained using one or the other update strategies was considered statistically non-significant ($p = 0.15$).

d198 instance In Figure 4.32 we can see the results for the instance d198, using the const update strategy. The results tend to improve with larger lowerQ0 values, but extreme values of higherQ0 should be avoided - this is specially visible when the lowest q_0 is 0.50 or 0.75.

The results achieved using the jump update strategy, for the same instance,

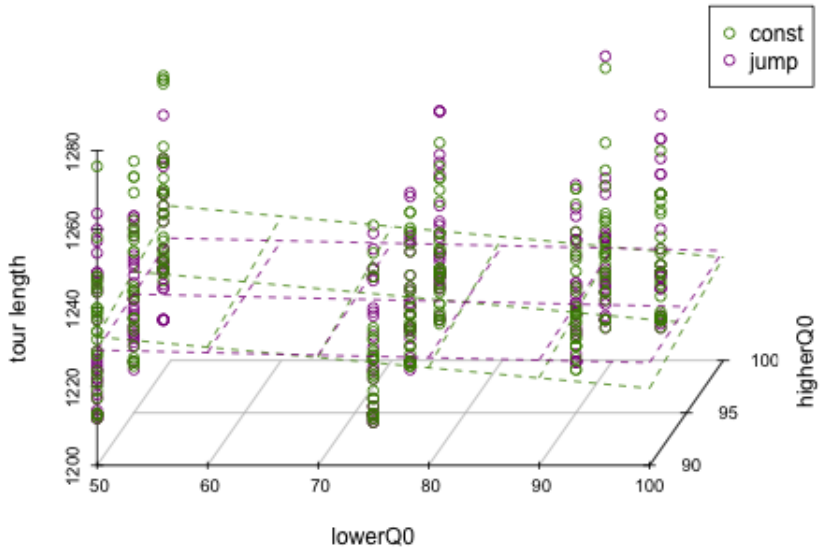

Figure 4.31: Instance rat99, without local search, dual caste configurations

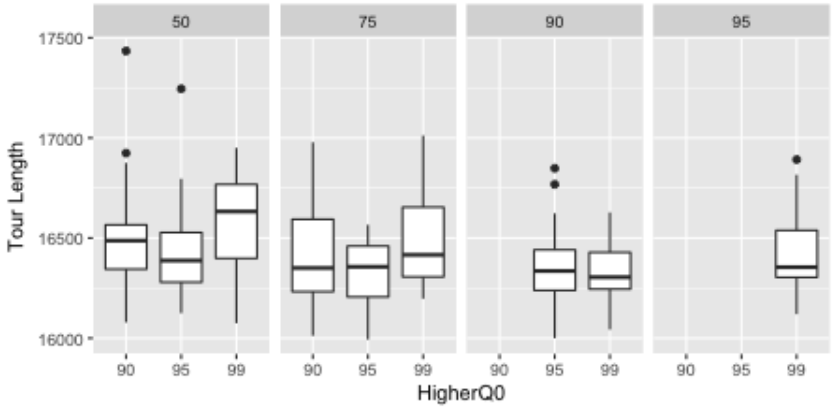

Figure 4.32: Instance d198, tour length, by lowerQ0, const update strategy, without local search

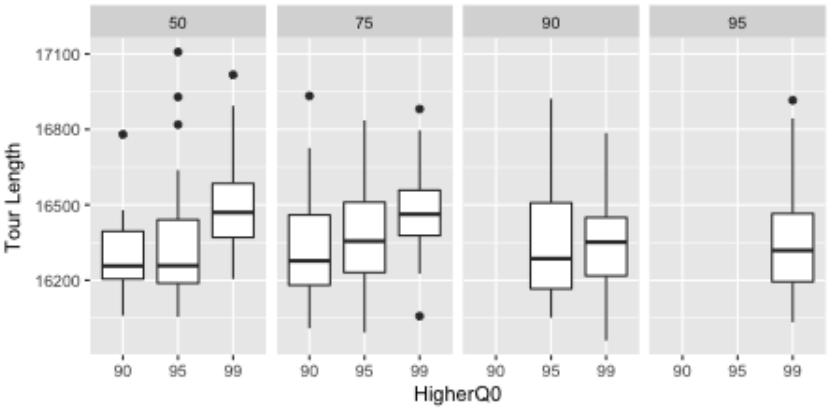

Figure 4.33: Instance d198, tour length, by lowerQ0, jump update strategy, without local search

can be observed in Figure 4.33. Here, the higherQ0 is the most determinant factor, and lower values of higherQ0 are preferred. As for the lowerQ0, the influence is not as clear, but higher values seem to be preferable.

When comparing the average tour length obtained by the two update strategies we found that the difference was significant ($p = 0.003041$) and, as such, we reject the null hypothesis. The jump update strategy obtains better, *i.e.*, shorter, average tour length than the const update strategy, so the ability to change the relative amount of exploratory/exploitative ants is an advantage.

fl417 instance The results achieved by the const update strategy can be observed in Figure 4.34. Unlike the previous figures, here the box-plots are grouped by the higherQ0 value and they reveal that, for a given higherQ0 value, the results improve with larger values of lowerQ0. When we consider a specific lowerQ0, the best results are usually achieved with smaller or intermediate values of higherQ0.

The results achieved by the jump update strategy are depicted in Figure 4.35. Once again the box-plots are grouped by the higherQ0 value, and the best configurations are those with a large lowerQ0 and intermediate higherQ0. Compared with the const strategy, the influence of higherQ0 is greater, but the influence of the lowerQ0 is less marked.

When comparing the average tour length obtained by the two update strategies we found that the difference was significant ($p = 0.02032$). The jump update

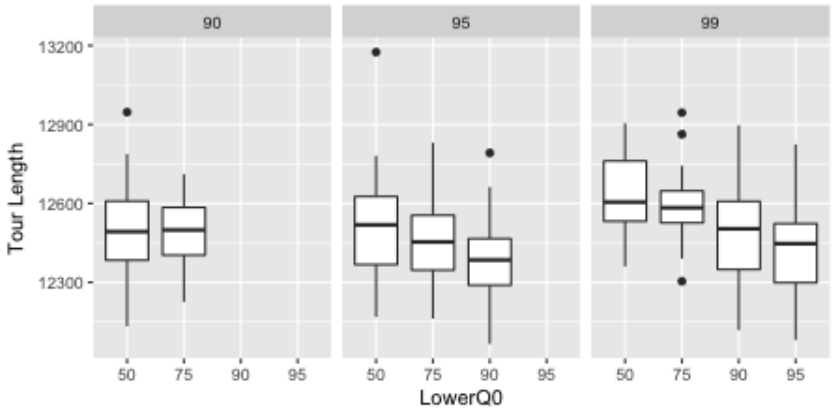

Figure 4.34: Instance fl417, tour length, by higherQ0, const update strategy, without local search

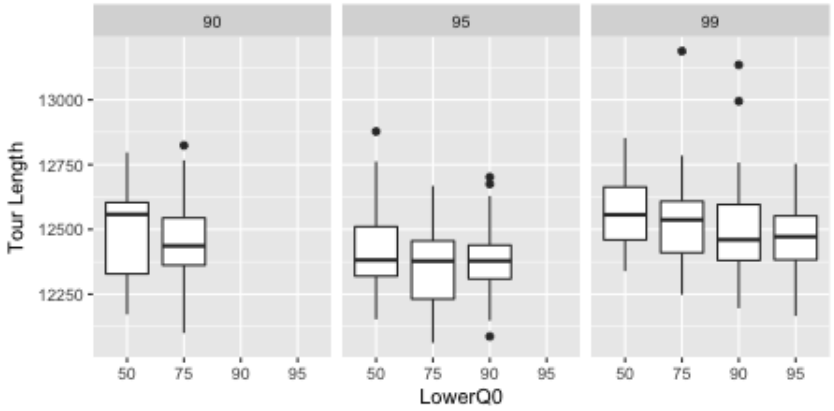

Figure 4.35: Instance fl417, tour length, by higherQ0, jump update strategy, without local search

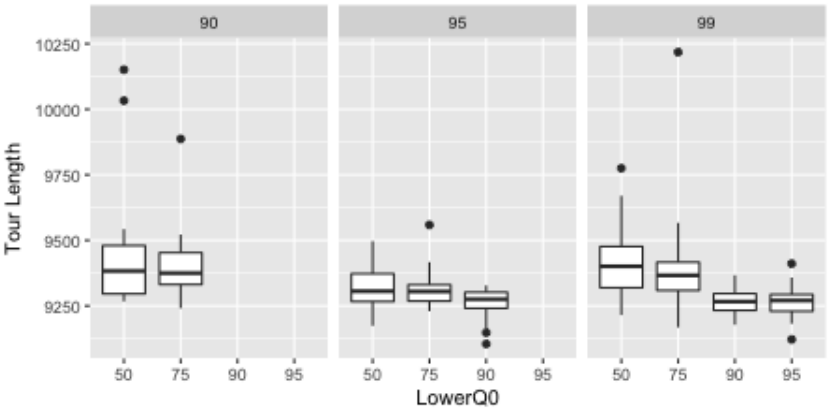

Figure 4.36: Instance rat783, tour length, by higherQ0, const update strategy, without local search

strategy obtains again better, *i.e.*, shorter, average tour length than the const update strategy.

rat783 instance The tour length results using the const update strategy can be observed in Figure 4.36. They reveal that, for a given higherQ0, the results tend to improve with increasing lowerQ0 values. Also, for a given lowerQ0, intermediate higherQ0 values are preferable. Lower values of lowerQ0 also tend to have a larger variance, making the quality of the solution harder to predict.

When using the jump update strategy, the influence of specific q_0 values is less visible, as can be observed in Figure 4.37. Here, the impact of lowerQ0 is not evident, although intermediate values of higherQ0 seem to provide an advantage.

Once again, the difference between the results obtained with the two update strategies is significant ($p = 2.8 \times 10^{-8}$). The jump strategy is better, reinforcing the advantage of adapting the exploration/exploitation tradeoff.

fl1577 instance Figure 4.38 presents the results obtained by the const update strategy on instance fl1577. We observe that, as a rule, higher values of lowerQ0 ($q_0 = 0.95$) are preferable, as are intermediate values of higherQ0. When using the jump update strategy (Figure 4.39), the lowerQ0 looses some of its influence. The most desirable value for lowerQ0 becomes less extreme, $q_0 = 0.90$, coupled with intermediate value for higherQ0. In this instance there are no significant

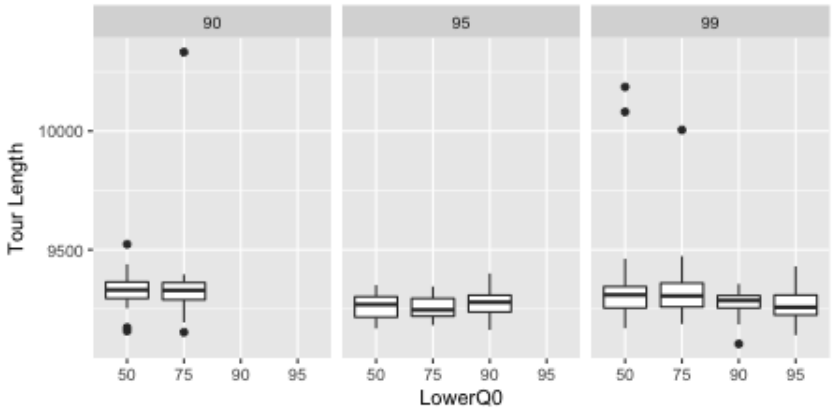

Figure 4.37: Instance rat783, tour length, by higherQ0, jump update strategy, without local search

differences between the results achieved by the two update strategies ($p = 0.166$).

pcb3038 instance The results obtained by the const update strategy can be consulted in Figure 4.40. The higherQ0 parameter has a large influence on the outcome, and low values should clearly be avoided. As for the lowerQ0, it also influences the performance although at a smaller degree, and larger values are also preferred. This is an expected behaviour, as previous experiments with ACS have shown that instance pcb3038 is the one that more clearly requires a large q_0 value. These results can be conferred in Figure 4.20.

As we can observe in Figure 4.41, the outcomes achieved by the jump update strategy reveal that the influence of the lowerQ0 is not as important. Also, the impact of higherQ0 is less marked. This behaviour is consistent with the one observed on smaller instances. The difference between the results obtained with the two update strategies is significant ($p = 2.0 \times 10^{-7}$). The jump strategy is, once again, able to produce better results.

rl5934 instance The const update strategy performance is depicted in Figure 4.42. In agreement with results presented for instance pcb3038, the most influent factor is the higherQ0, that should be high. LowerQ0 also has some, less determinant, influence and should also be high.

When using the jump update strategy (Figure 4.43), the lowerQ0 parameter

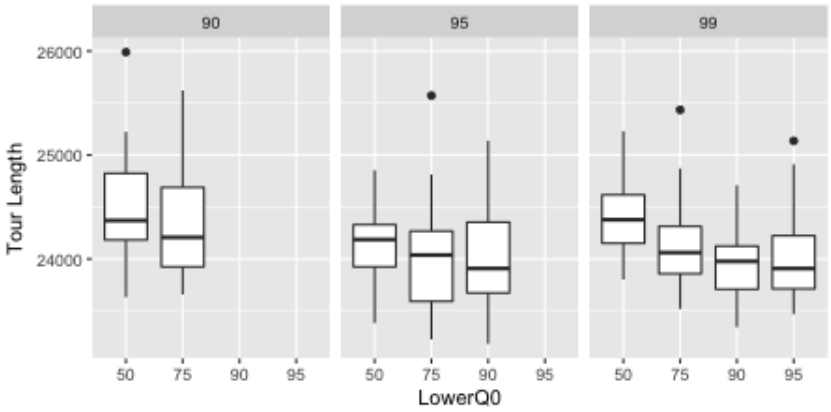

Figure 4.38: Instance fl1577, tour length, by higherQ0, const update strategy, without local search

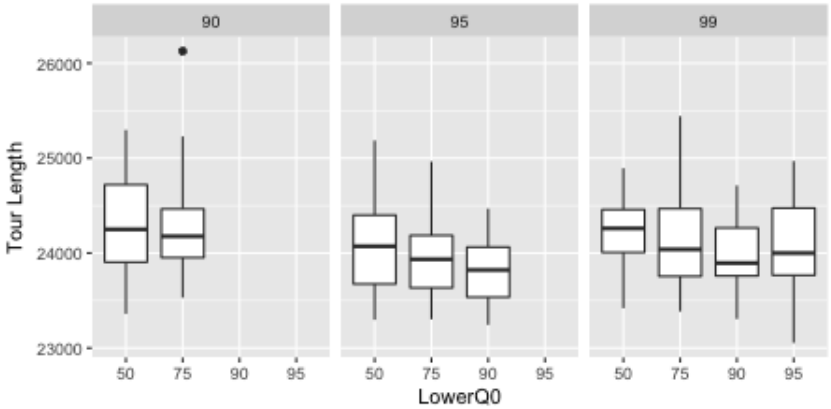

Figure 4.39: Instance fl1577, tour length, by higherQ0, jump update strategy, without local search

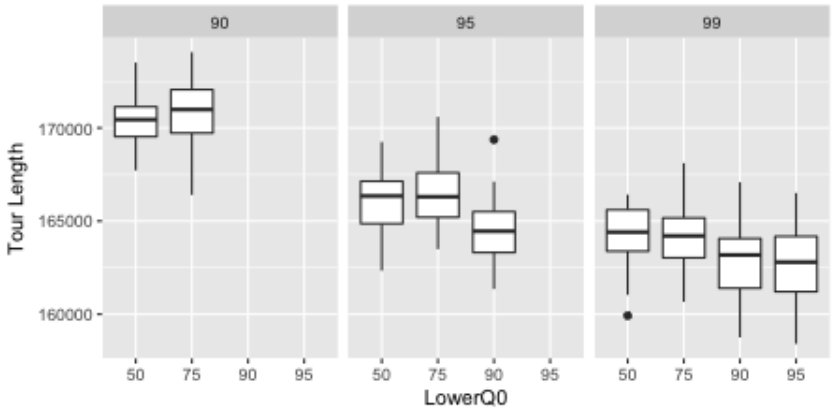

Figure 4.40: Instance pcb3038, tour length, by higherQ0, const update strategy, without local search

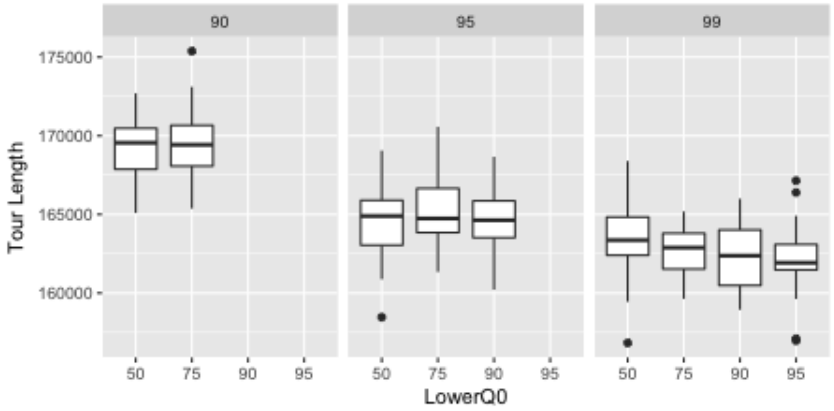

Figure 4.41: Instance pcb3038, tour length, by higherQ0, jump update strategy, without local search

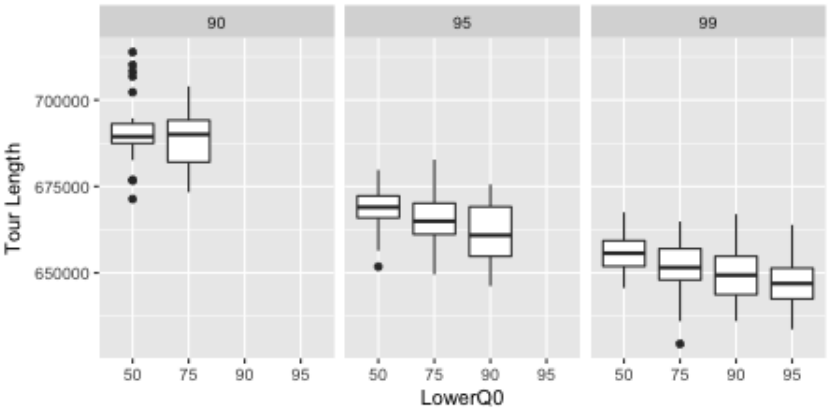

Figure 4.42: Instance rl5934, tour length, by higherQ0, const update strategy, without local search

looses influence and results are solely influenced by the higherQ0 values. This is likely because the algorithm was able to increase the size of the higherQ0 caste and then relied mostly on the results achieved by those ants, thus making the exact q_0 value of the lowerQ0 caste less important. The irrelevance of the lowerQ0 when using the jump update strategy can also be observed in the scatter plot of the tour lengths achieved by the various configurations depicted in Figure 4.44. The difference between the results obtained with the two update strategies is significant ($p = 0.01$) and the jump strategy presents a lower mean tour length.

Discussion: Dual caste configurations without local search

Table 4.6 provides a summary of some of the main features of the dual caste configurations performance. The symbol "nci" stands for "no clear influence". The caste that most influences the behaviour of the algorithm changes according to the instance, as does the specific q_0 value for that caste. In smaller instances, the q_0 value of the higherQ0 caste should be moderate or low, and the q_0 value of lowerQ0 caste should avoid very low values. As the size of the instance increases, higher values for both lowerQ0 and higherQ0 are preferable. The q_0 value of the lowerQ0 caste is more relevant when using the const update strategy than when using the jump one. As a rule, the jump strategy leads to results as good as or better than the const strategy.

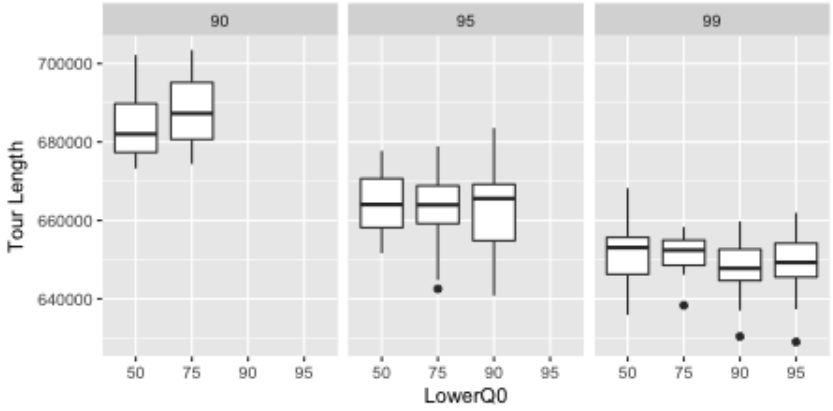

Figure 4.43: Instance rl5934, tour length, by higherQ0, jump update strategy, without local search

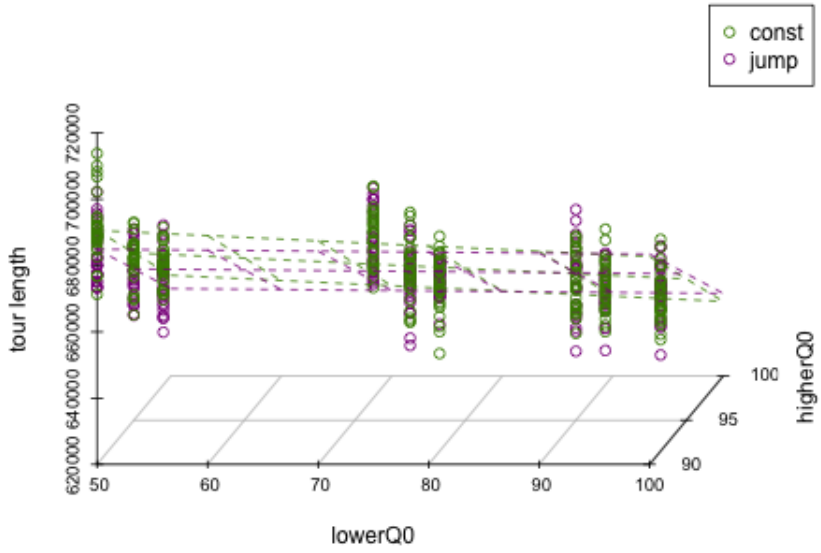

Figure 4.44: Instance rl5934, without local search, dual caste configurations

instance	update strategy	most influential	lowerQ0 should be	higherQ0 should be
rat99	const	mostly lowerQ0	not very low	moderate or low
	jump	nci	nci	nci
d198	const	mostly lowerQ0	not very low	moderate or low
	jump	both	nci	moderate or low
fl417	const	mostly lowerQ0	high	moderate
	jump	both	high	moderate
rat783	const	mostly lowerQ0	high	moderate or high
	jump	higherQ0	nci	moderate
fl1577	const	mostly higherQ0	high	moderate
	jump	mostly higherQ0	high or moderate	moderate or high
pcb3038	const	mostly higherQ0	high	high
	jump	higherQ0	nci	high
rl5934	const	mostly higherQ0	high	high
	jump	higherQ0	nci	high

Table 4.6: Summary of influential factors for the dual caste without local search

The advantage of q_0 around 0.9 and 0.95 in smaller instances, and larger $q_0 = 0.99$ in the larger instances can be observed in Table 4.6, to a certain degree, particularly in the Const variant. For example, when one of the castes is $q_0 = 0.50$ or $q_0 = 0.75$ and the other caste is $q_0 = 0.90$ or $q_0 = 0.99$, (c50_90, c50_99, c75_90 and c75_99), the results range from poor to indifferent. Is clear that when the higher q_0 is 0.90, (c50_90 and c75_90) the results are particularly poor in the larger instances, while the other configurations (c50_99 and c75_99) have more difficulties with the smaller instances.

The desirability of a large q_0 in the large instances, and smaller q_0 in the smaller instances is also seen on jump configurations (j50_90, j55_99, j75_90 and j75_99), but to a lesser degree. This may be explained by the fact that it is likely that a $q_0 = 0.50$ or $q_0 = 0.75$ is too extreme. Since half of the ants on the const configurations must use that q_0, some of the exploitative potential is wasted. On the contrary, the jump configurations may adjust the caste size, so the sub-optimality of the q_0 values employed is less visible.

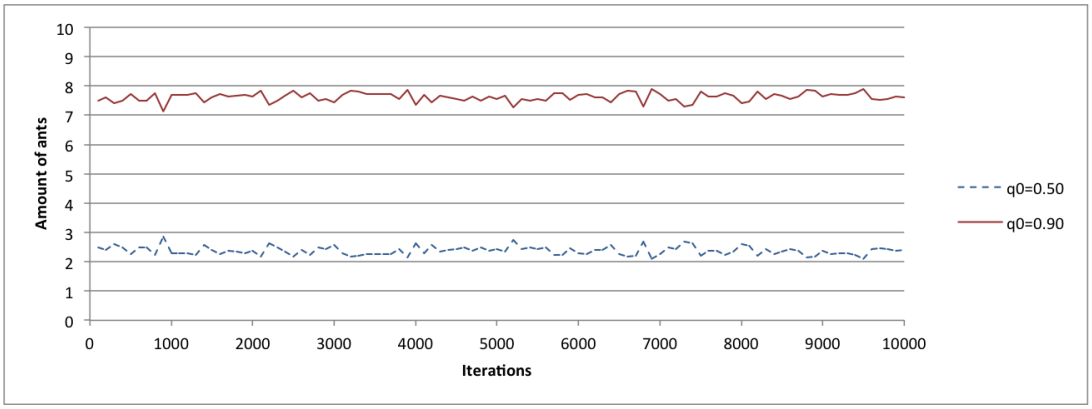

Figure 4.45: Average castes size over time, for the rl5934 instance, without local search, using j50_90 configuration

Castes' size The analysis of the castes' size indicates that the ants do migrate, thereby adapting to the search needs. For example, in Figure 4.45 we can observe the average number of ants in each caste of configuration j50_90 when solving instance rl5934. The values were taken each 10 iterations and averaged for the 30 repetitions. It can be seen that the number of ants in the $q_0 = 0.90$ caste remains clearly higher than those in the $q_0 = 0.50$ caste. This may explain why j50_90 is able to avoid the very poor results of c50_90.

Average q_0 of the caste that found the best-so-far solution over time Figure 4.46 registers the q_0 of the caste that found the best-so-far solution over time for configuration c50_90 on problem instance rl5934, averaged over the 30 repetitions. It can be observed that, although there are 5 ants in each caste, the ants that found the best solutions were usually those from the $q_0 = 0.90$ caste. For configuration j50_90, and this same instance, the caste with the lower q_0 found the best-so-far-solution even less frequently, which is expected given the instance.

In fact, as a rule, the configurations of the jump variant had a lower average tour length than the configurations with similar castes but of the const variant, excepting c95_99. This exception may be explained by looking at Figure 4.47, that depicts the average q_0 of the caste that found the best-so-far solution over time, for instance rl5934 and configurations c95_99 and j95_99. We can see that, contrary to what happened in Figure 4.46, in this case both castes are contributing

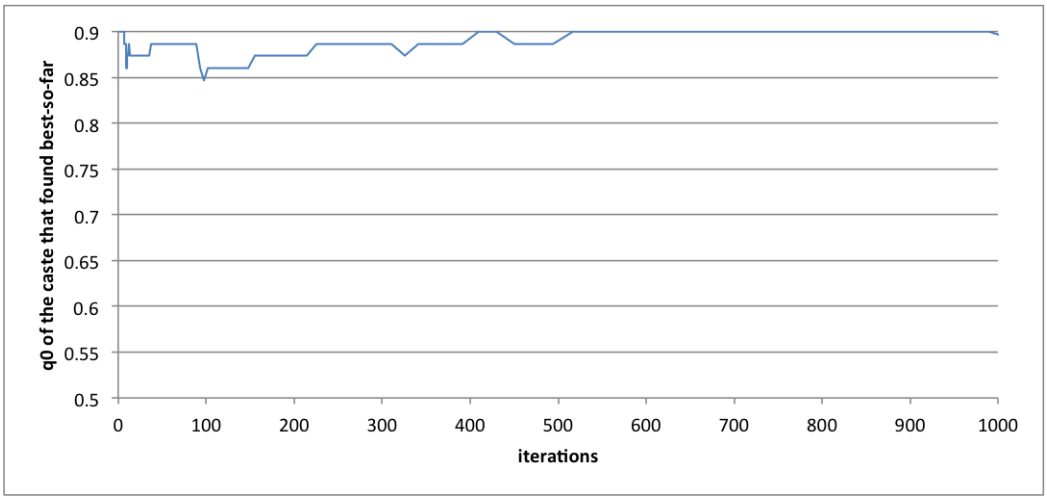

Figure 4.46: Average q_0 of the caste that found the best-so-far solution over time, for the rl5934 instance, without local search, using c50_90 configuration

in almost equal footing to the finding of good solutions, so the need for a drastic change in the caste size is less acute.

Dual-caste solution quality: with local search

The box-plots depicted in this section refer to the tour length found by the algorithm while using local search. Regardless of the instance, each repetition was allowed to run for 50 iterations. When comparing the performance obtained by the different update strategies we relied on the Wilcoxon signed rank test with continuity correction (significance level=0.05), as described in the previous section.

rat99 instance The performance achieved by the const update strategy is depicted in Figure 4.48. As it can be seen, regardless of the configuration, most of the repetitions reached the optimum. This is to be expected as the problem is small and easily solved by the hybrid algorithm. The results achieved using the jump update strategy can be consulted in Figure 4.49. In this case, the higherQ0 seems to have some influence, as configurations with lower values for this parameter are able to avoid suboptimal results. As this instance is usually best solved using lower q_0 values - as we can see in Figure 4.8, it is likely that the jump mechanism, by enlarging the lowerQ0 caste size is able to counteract the negative influence

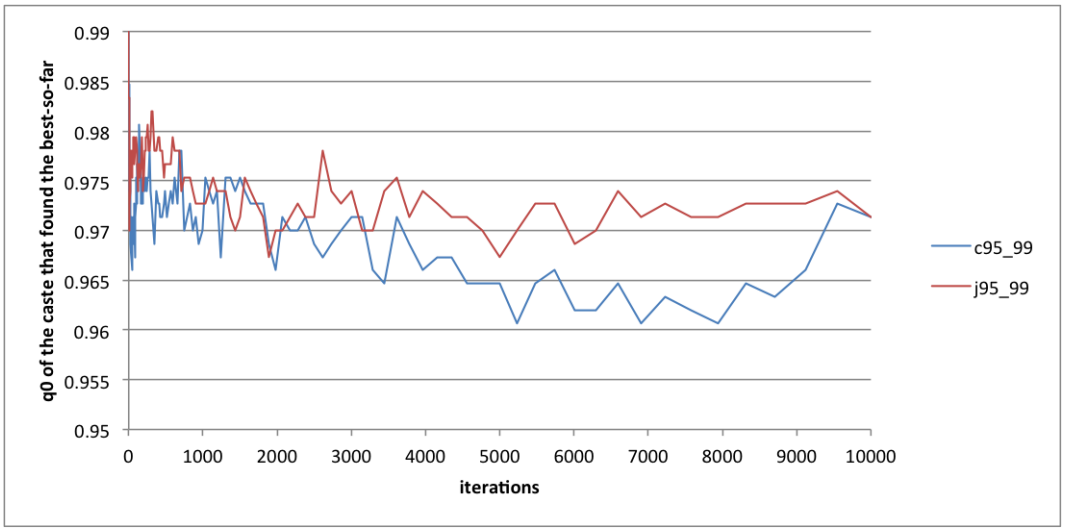

Figure 4.47: Average q_0 of the caste that found the best-so-far solution over time, for the rl5934 instance, without local search, using c95_99 and j95_99 configurations

of higher q_0 values, albeit only to a certain extent. In this instance there are no significant differences between the results achieved by the two update strategies ($p = 0.54$).

d198 instance The results achieved by the const update strategy are depicted in Figure 4.50, whilst those obtained by the jump update strategy are presented in Figure 4.51. A brief perusal of the charts reveals no clear influence produced by either the higherQ0 or the lowerQ0. This is explained by the fact that the problem instance is small and can be partially solved by the local search algorithm. Also, it was previously reported that this particular instance is not as sensitive to the q_0 value (see Figure 4.9), as other instances. When comparing the overall performance of the const update strategy versus the jump update strategy, the difference between the means was found non significant, with $p = 0.33$.

fl417 instance Figure 4.52 depicts the results achieved by the const update strategy. The influence of the q_0 values is faint, yet higher values of higherQ0 contribute to the existence of outliers, meaning that sometimes the algorithm may be prone to premature convergence. As for the lowerQ0, moderate values

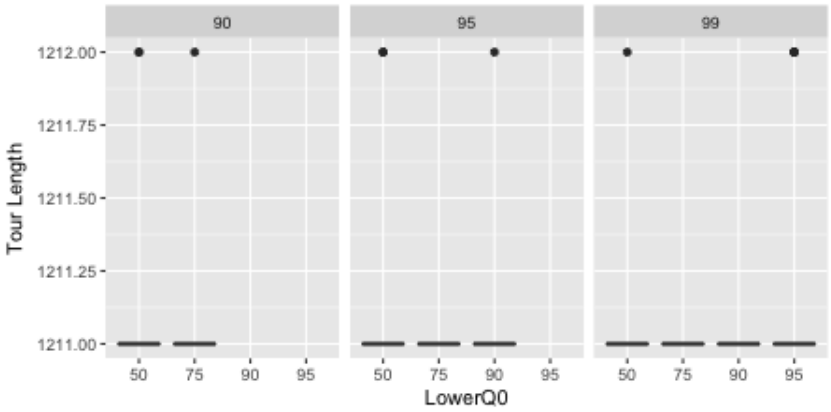

Figure 4.48: Instance rat99, tour length, by higherQ0, const update strategy, with local search

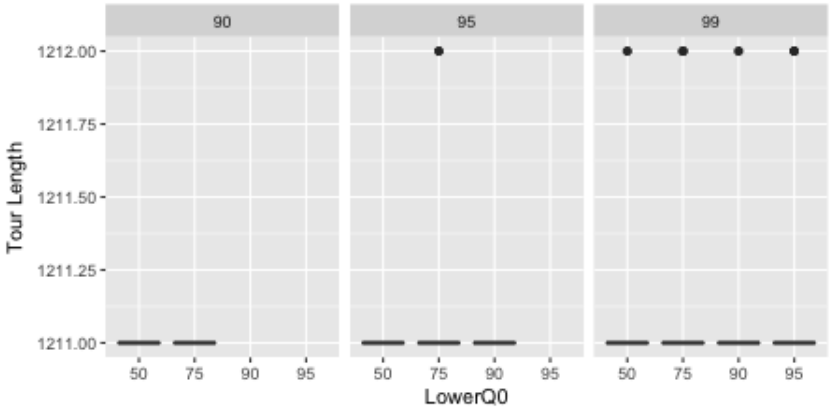

Figure 4.49: Instance rat99, tour length, by higherQ0, jump update strategy, with local search

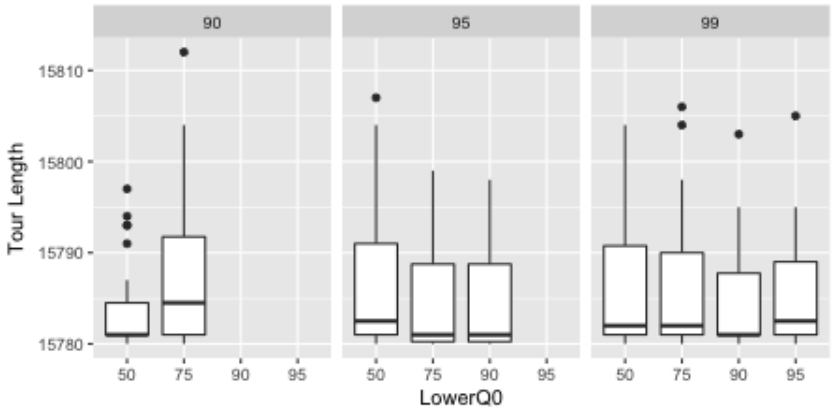

Figure 4.50: Instance d198, tour length, by higherQ0, const update strategy, with local search

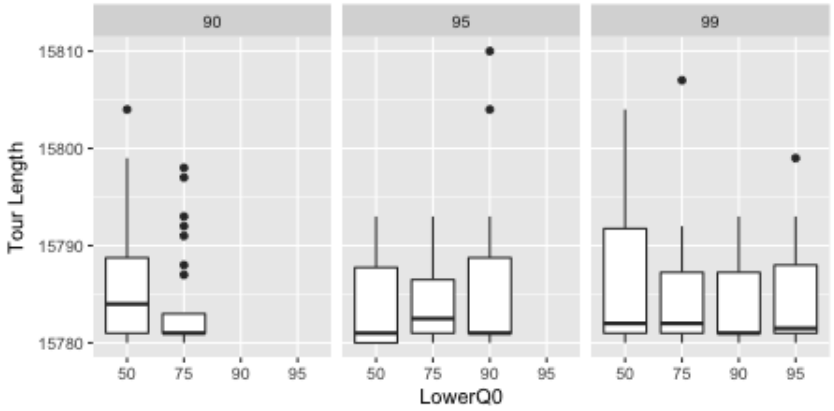

Figure 4.51: Instance d198, tour length, by higherQ0, jump update strategy, with local search

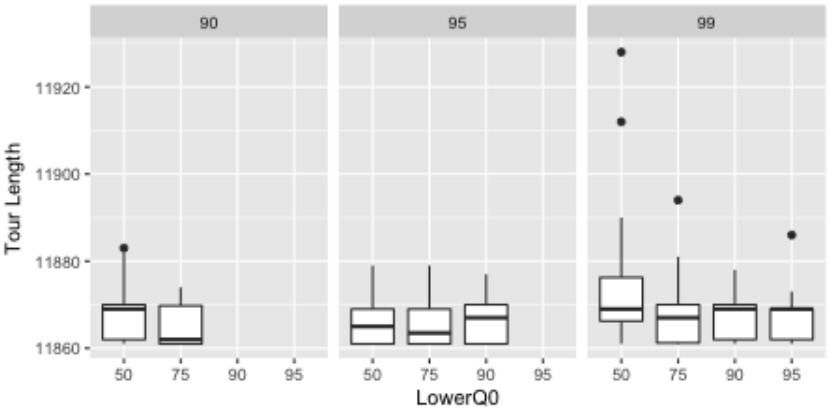

Figure 4.52: Instance fl417, tour length, by higherQ0, const update strategy, with local search

consistently achieved the better results. The results achieved using the jump update strategy are depicted in Figure 4.53. Here the influence of lowerQ0 becomes almost null, and even the higherQ0 influence is smaller and only relevant when combined with low values of lowerQ0. Again, the difference between the results obtained by the two configurations is not significant with a $p = 0.30$.

rat783 instance The results achieved by the const update strategy can be consulted in Figure 4.54. One can observe that they tend to improve with increasing lowerQ0 values. The higherQ0 influence is more subtle, but intermediate values are, as a rule, preferable. When using the jump strategy, the influence of the lowerQ0 becomes less distinct as can be seen in Figure 4.55. In this scenario, the influence of the lowerQ0 is clear only when combined with a high higherQ0 and, in that case, the lowerQ0 should be moderate to high. As for the higherQ0, moderate values are still the best option. For this instance, the jump update strategy obtained significant better results than the const strategy ($p = 0.010$).

fl1577 instance The application of the const strategy does not reveal a clear pattern of influence by either the lowerQ0 or the higherQ0 parameters. As it can be confirmed in Figure 4.56, the boxes are quite similar, although lower values of lowerQ0 tend to achieve lower (better) mean tour lengths when the higherQ0 is also lower. Similar results are achieved when employing the jump update strategy,

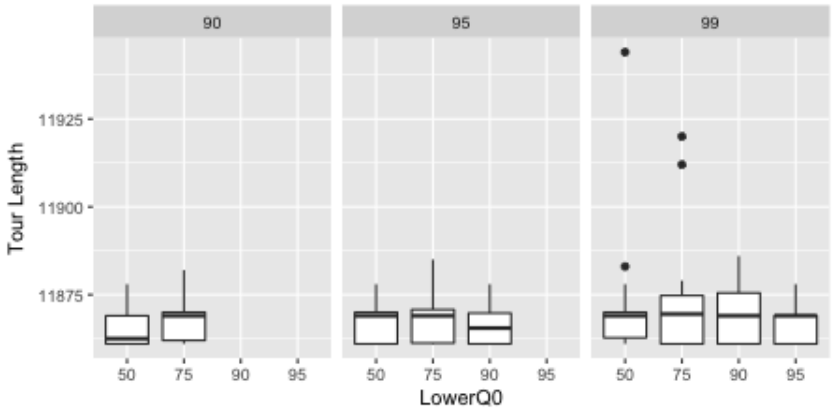

Figure 4.53: Instance fl417, tour length, by higherQ0, jump update strategy, with local search

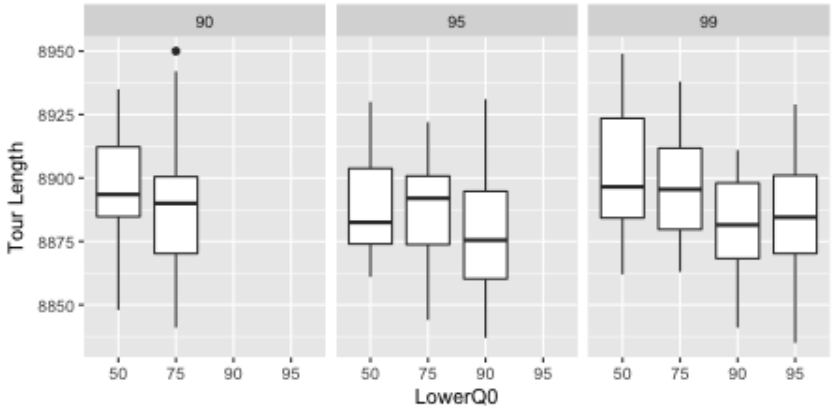

Figure 4.54: Instance rat783, tour length, by higherQ0, const update strategy, with local search

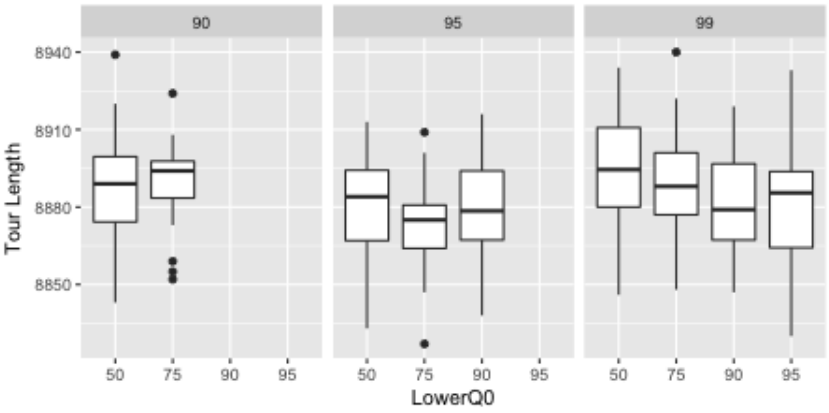

Figure 4.55: Instance rat783, tour length, by higherQ0, jump update strategy, with local search

depicted in Figure 4.57 where still no pattern arises. In this case, the difference between the results obtained by the two configurations is not significant with a $p = 0.88$.

pcb3038 instance Figure 4.58 depicts the performance of the const strategy. In this case, we can visually perceive the influence of the higherQ0, as the group of box-plots in each panel stand at a different level, while the impact of the lowerQ0 is not clear. Higher values of higherQ0 are preferable. The results achieved while using the jump update strategy can be found in Figure 4.59. Again the higherQ0 has a large influence on the quality of the results, while the relevance of the lowerQ0 is only visible for intermediate values of higherQ0. The preference for larger higherQ0 values is expected, as that was also the case while using standard ACS (Figure 4.13).

If we visually compare the results achieved by the configurations with a larger higherQ0, we can see that the ability of dynamically adjusting the size of the castes proves to be an advantage, as the average tour length tends to be smaller (better) when using the jump update strategy. In this case, moving the ants to the clearly preferable caste allows the algorithm to perform better. The statistical analysis of the results supports this claim, as the jump update strategy obtained significant better results than the const strategy ($p = 8 \times 10^{-7}$).

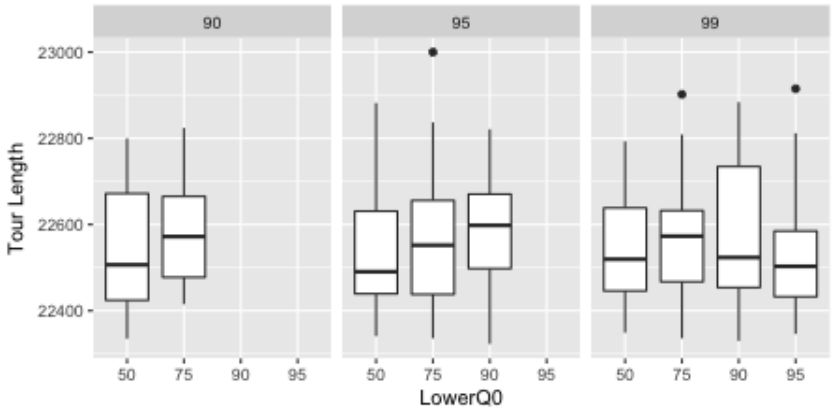

Figure 4.56: Instance fl1577, tour length, by higherQ0, const update strategy, with local search

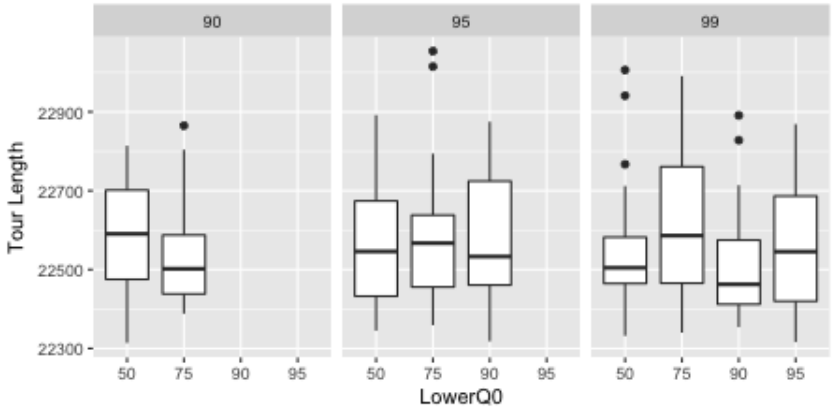

Figure 4.57: Instance fl1577, tour length, by higherQ0, jump update strategy, with local search

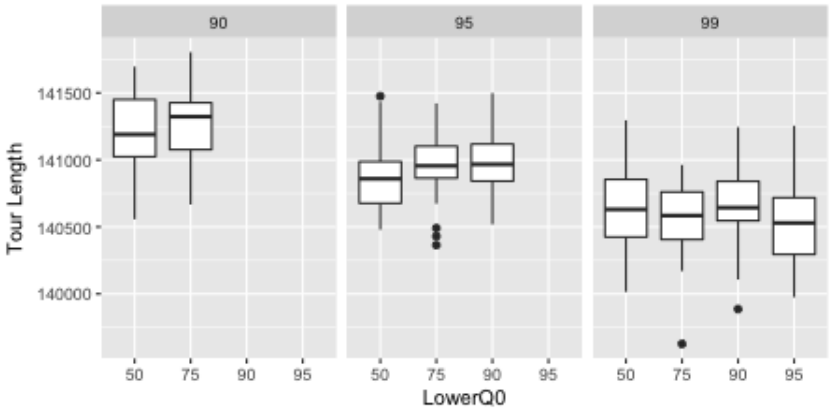

Figure 4.58: Instance pcb3038, tour length, by higherQ0, const update strategy, with local search

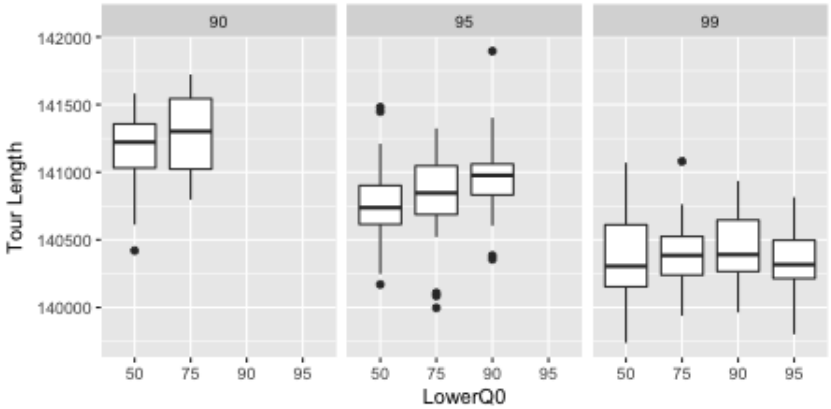

Figure 4.59: Instance pcb3038, tour length, by higherQ0, jump update strategy, with local search

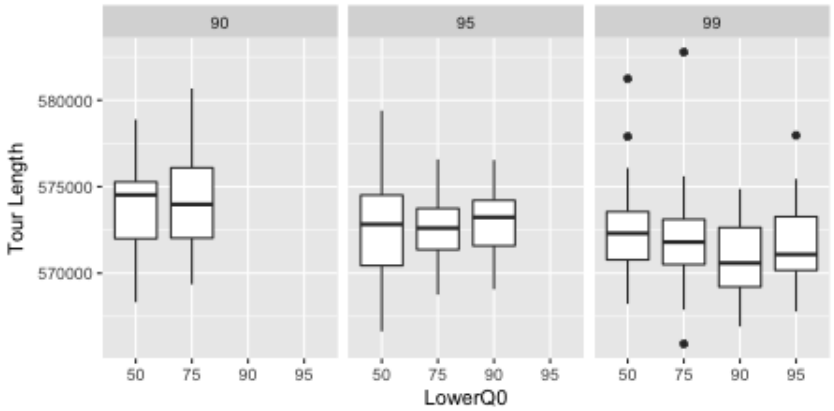

Figure 4.60: Instance rl5934, tour length, by higherQ0, const update strategy, with local search

rl5934 instance The performance of the const update strategy on the largest instance can be observed Figure 4.60. In this case we can perceive that the main influence is the higherQ0 value. Larger values of higherQ0 value are preferable and there is no clear pattern for the lowerQ0. Results obtained with the jump update strategy appear in Figure 4.61. Just like in the pcb3038 instance, the lowerQ0 looses influence and only the higherQ0 impacts performance (higher values are preferable for this parameter). In this case, the difference between the results obtained by the two configurations is not significant with a $p = 0.93$.

Discussion: Dual caste configurations with local search

On Table 4.7 we present a summary of some of the main features of the dual caste configurations performance, when using local search. The symbol "nci" stands for "no clear influence". The influence of specific q_0 values and of the update strategy is harder to discern when using local search. This is expected, as much of the optimisation is taken over by the specialised local search algorithm, so the impact of the multi-caste approach is more subtle. When using the jump update strategy, the parameter lowerQ0 tends to loose influence, particularly in the larger instances.

In any case, a detailed analysis reveals that the choice of q_0 still impacts the results. The most robust configuration is j95_99, a combination of the q_0 from the two best ACS configurations. Regardless of the instance, j95_99 is similar to,

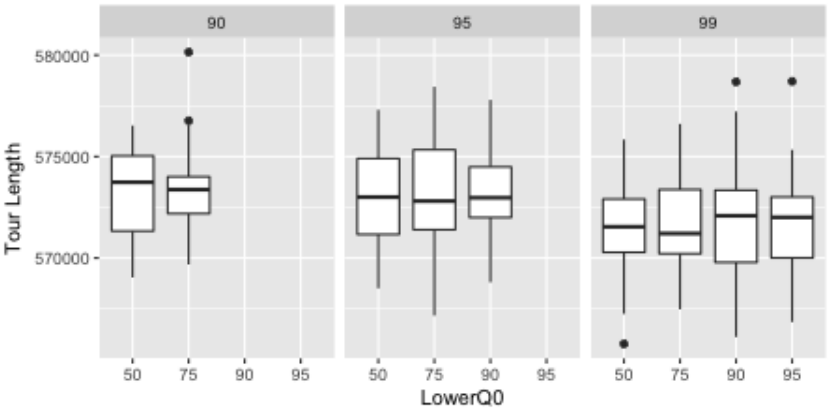

Figure 4.61: Instance rl5934, tour length, by higherQ0, jump update strategy, with local search

instance	update strategy	most influential	lowerQ0 should be	higherQ0 should be
rat99	const	nci	nci	nci
	jump	higherQ0	nci	low or moderate
d198	const	nci	nci	nci
	jump	nci	nci	nci
fl417	const	both	moderate	moderate or low
	jump	mostly higherQ0	nci	moderate or low
rat783	const	mostly lowerQ0	moderate to high	moderate
	jump	higherQ0	nci	moderate
fl1577	const	nci	nci	nci
	jump	nci	nci	nci
pcb3038	const	higherQ0	nci	high
	jump	higherQ0	low	high
rl5934	const	higherQ0	nci	high
	jump	higherQ0	nci	high

Table 4.7: Summary of influential factors for bi-caste with local search

or better than, any of the ACS configurations. All configurations with two castes of the Jump variant that had one caste with $q_0 = 0.99$ achieved good results. As a general rule, one of the castes should have a $q_0 = 0.99$, and the other should be $q_0 \geqslant 0.90$ if using the const migration strategy or, $q_0 \geqslant 0.75$, if using the jump variant. The jump migration strategy seems to be preferable, particularly for configurations with castes of the form 50_90, 50_99, 75_90 and 75_99.

4.4.4 Quad-caste configurations: solution quality

In configurations with four castes, three factors can potentially influence the outcome of the algorithm (consult Table 4.5 for reference):

- the specific q_0 of the castes with lower q_0 value - the other three castes have q_0 values of 0.90, 0.95 and 0.99;
- the update strategy - const or jump;
- the colony size (total number of ants) - 8, 12 or 20;

In the following sections, the q_0 value of the caste with lower q_0 will be referred to as the "lowerQ0". To simplify the reading of the horizontal axis in the boxplots, we present the q_0 values multiplied by 100, so a lowerQ0 of 50 means that $q_0 = 0.50$. We refer to the colony size as nAnts. When the algorithm initiates, each caste has the same amount of ants, so 2, 3, or, 5, depending on the nAnts. Results without and with local search are presented separately.

Quad-caste solution quality: without local search

In this section, regardless of the instance, each repetition was allowed to run for approximately 100000 evaluations - each ant/solution was evaluated once per iteration - so $100000/nAnts$ iterations: 12500 iterations for colonies with 8 ants, 8333 iterations for colonies with 12 ants and 5000 iterations for colonies with 20 ants.

Unless explicitly stated, results presented in this section have significant deviations from normality. In accordance, when comparing the performance obtained by the different update strategies, we relied on the Wilcoxon signed rank test with continuity correction (significance level=0.05), as described in the previous sections.

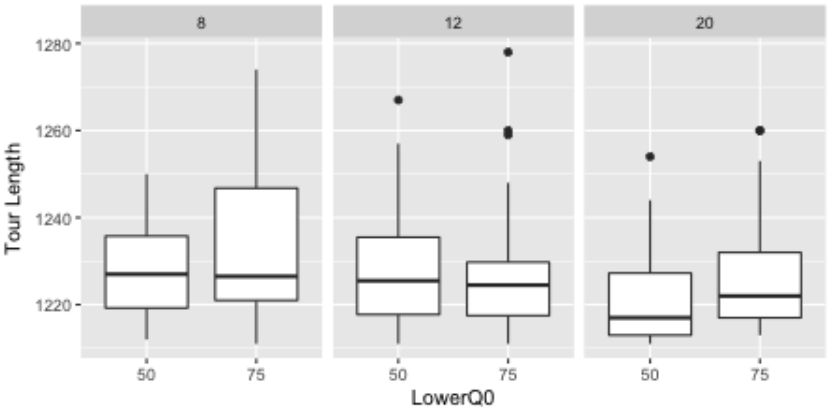

Figure 4.62: Instance rat99, tour length, by nAnts, const update strategy, without local search

rat99 instance As it can be seen in Figure 4.62, the most influential parameter when using the const update strategy is the number of ants. Larger colonies achieve better results and the lowerQ0 is only relevant in these situations. When using the jump update strategy (Figure 4.63), both the colony size and the lowerQ0 value loose influence. No significant difference was found between the results obtained by the two update strategies ($p = 0.46$).

d198 instance Figure 4.64 displays the results obtained by the const update strategy. There is not a clear pattern of influence, but smaller colonies appear to be the more promising, particularly if combined with a higher lowerQ0. When using the jump update strategy (see Figure 4.65) the behaviour is similar, although the lowerQ0 value looses influence in the larger colonies. No significant difference was found between the results obtained by the two update strategies ($p = 0.69$).

fl417 instance The result on fl417 also lack a clear pattern of influence, although const update strategy tends to obtain better results with a smaller colony size (see Figure 4.66). The influence of the lowerQ0 is mostly felt when the colony is large. When using the jump update strategy, the performance of the configurations become more homogeneous, as can be confirmed in Figure 4.67. No significant difference was found between the results obtained by the two update strategies ($p = 0.78$).

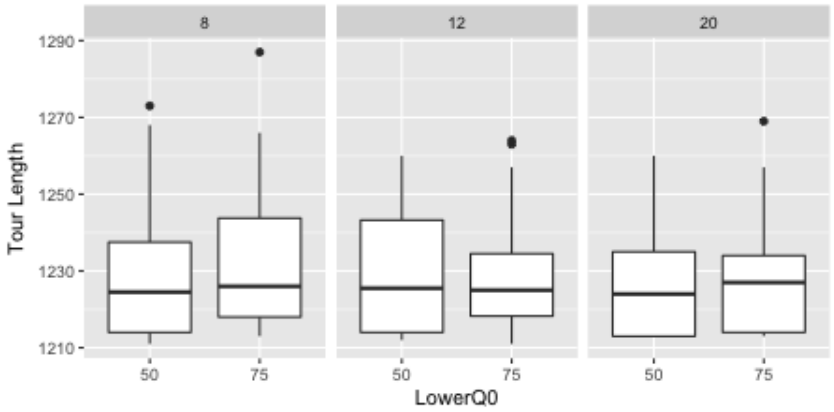

Figure 4.63: Instance rat99, tour length, by nAnts, jump update strategy, without local search

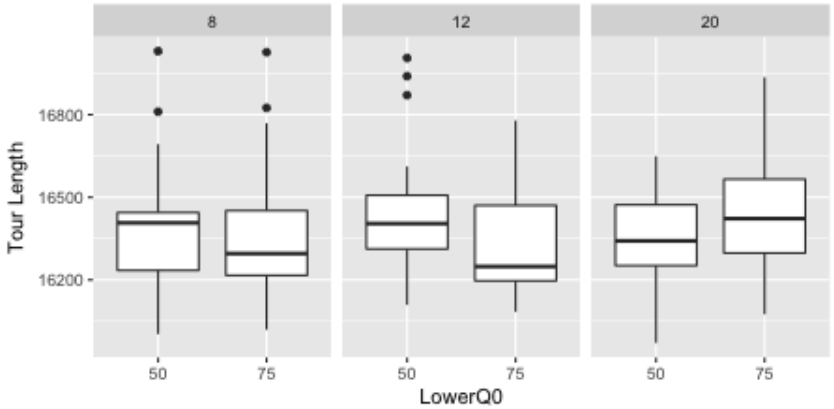

Figure 4.64: Instance d198, tour length, by nAnts, const update strategy, without local search

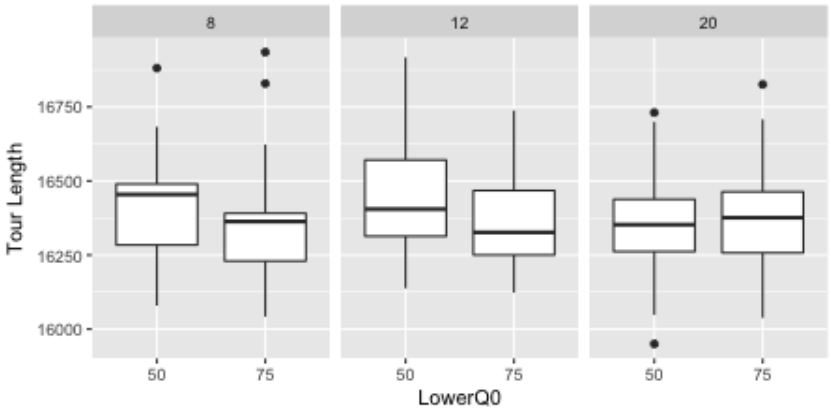

Figure 4.65: Instance d198, tour length, by nAnts, jump update strategy, without local search

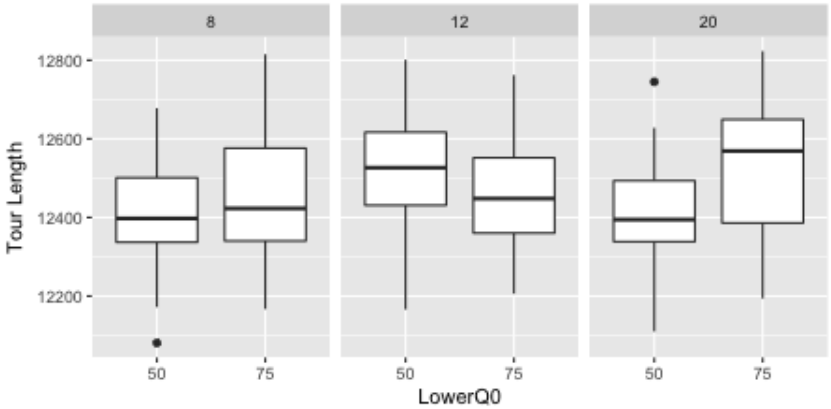

Figure 4.66: Instance fl417, tour length, by nAnts, const update strategy, without local search

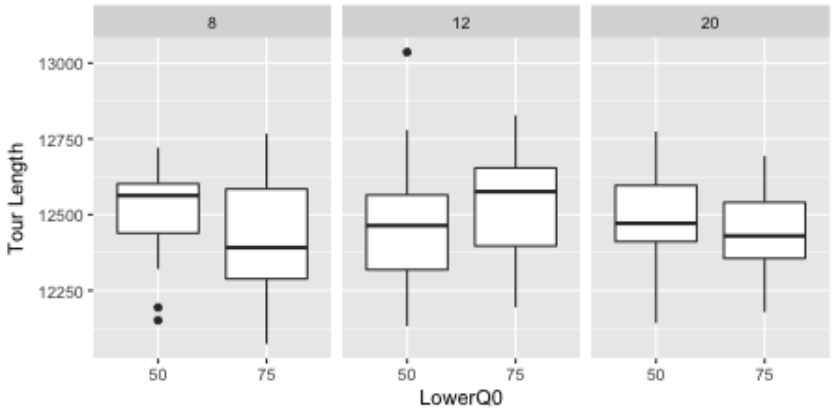

Figure 4.67: Instance fl417, tour length, by nAnts, jump update strategy, without local search

rat783 instance Figure 4.68 displays the performance of the const update strategy. The main influence is the number of ants. In smaller colonies (8 to 12 ants) the lowerQ0 has no influence, but a higher lowerQ0 is preferable for larger colonies. The behaviour when using the jump update strategy is depicted in Figure 4.69. The influence pattern is all but lost, although medium sized colonies (12 ants), specially when paired with an larger highQ0, present fewer and smaller outliers. Overall, the jump update strategy leads to an improved performance ($p = 2.8 \times 10^{-3}$).

fl1577 instance Figure 4.70 reveals that, when using the const strategy, there is a deterioration in the performance as the colony size increases, but the lowerQ0 does not have a clear influence. The behaviour of the jump update strategy can be observed in Figure 4.71. In this case the colony size is not relevant, as all configurations tend to have a similar performance. LowerQ0 seems to have some influence over the smaller colonies.

In figure 4.72 we can observe the scatter plot of the solutions obtained for instance fl1173. The results achieved by the const update strategy are depicted in green, while those in magenta belong to the jump update strategy. We also added two regression planes - green for const, magenta for jump - to make the trends more visible. We can observe that the plane from the jump configurations is parallel to both the x-axis and the z-axis, meaning that neither the nAnts nor the lowerQ0

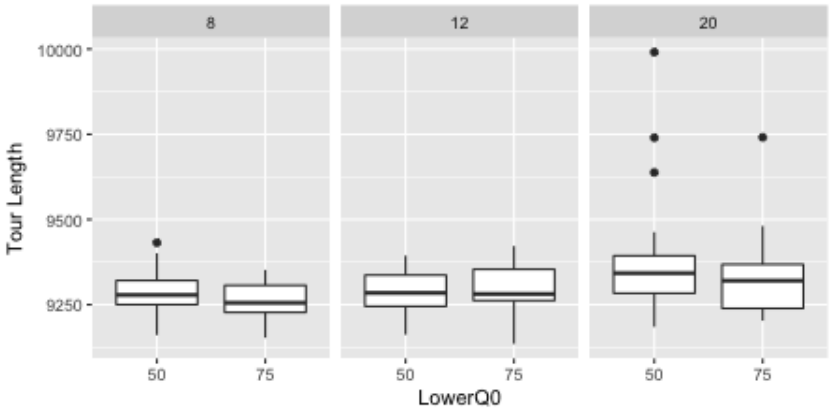

Figure 4.68: Instance rat783, tour length, by nAnts, const update strategy, without local search

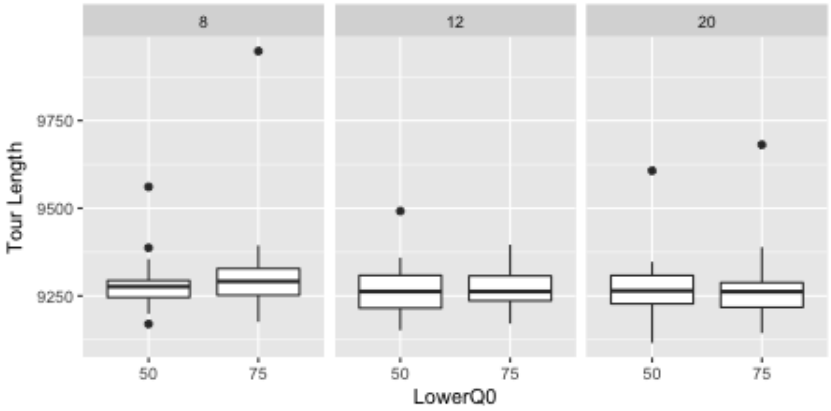

Figure 4.69: Instance rat783, tour length, by nAnts, jump update strategy, without local search

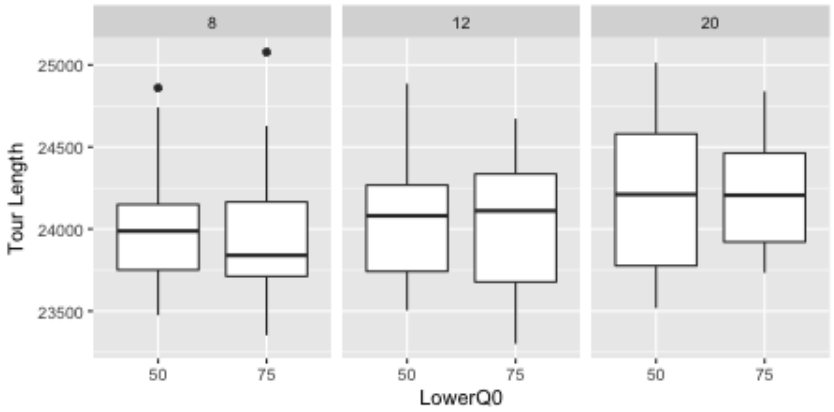

Figure 4.70: Instance fl1577, tour length, by nAnts, const update strategy, without local search

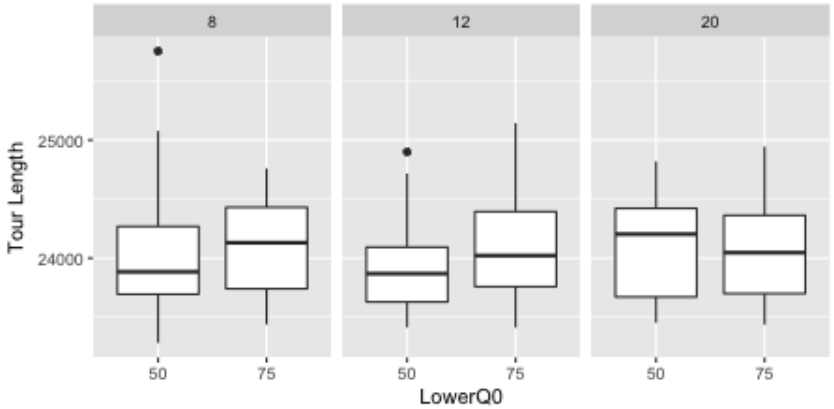

Figure 4.71: Instance fl1577, tour length, by nAnts, jump update strategy, without local search

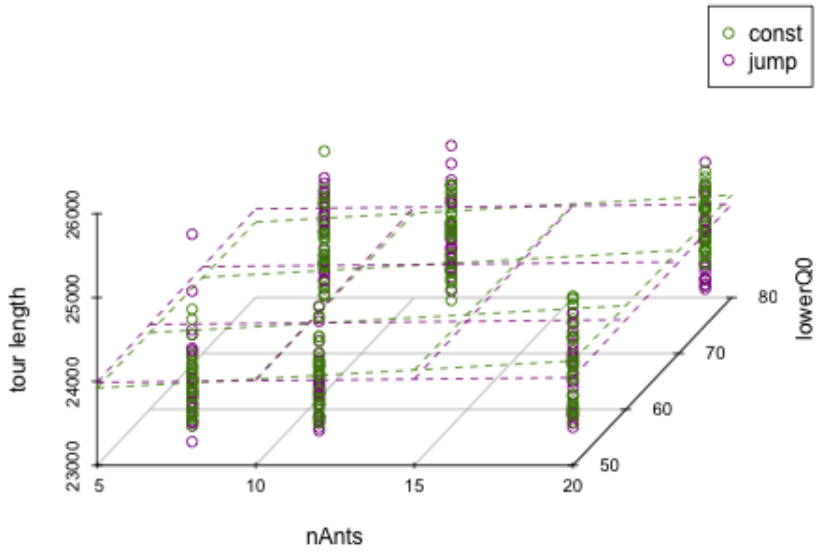

Figure 4.72: Instance fl1577, tour length scatter plot and regression planes, quad castes, without local search

have a large influence on the algorithm behaviour. By contrast, the green plane, belonging to the const update strategy, is not parallel to the x-axis and the tour length grows with the number of ants. In smaller colonies, the lowerQ0 also has a small influence, which can be confirmed by the fact that the plane is not perfectly parallel to the z-axis on that side. In this instance, no significant difference was found between the results obtained by the two update strategies ($p = 0.23$).

pcb3038 instance Figure 4.73 reveals that the main influence when using the const update strategy is the number of ants, while the impact of lowerQ0 is not visible. When using the jump update strategy, the performance of the algorithm becomes almost perfectly uniform, regardless of the caste size or lowerQ0, as can be observed in Figure 4.74. Like in other instances, the ability of the castes to adapt their size compensates for the suboptimal parameters selection, regarding, not only the specific lowerQ0 value, but also the frequency of trail update.

Since the tour length values using both the const or the jump update strategy follow a normal distribution, we used the Levene's Test for Homogeneity of Variance provided in the 'car' package from the [R, 2011] software. The test was found non-significant, with $p = 0.195$, so we do not reject the hypotheses that the

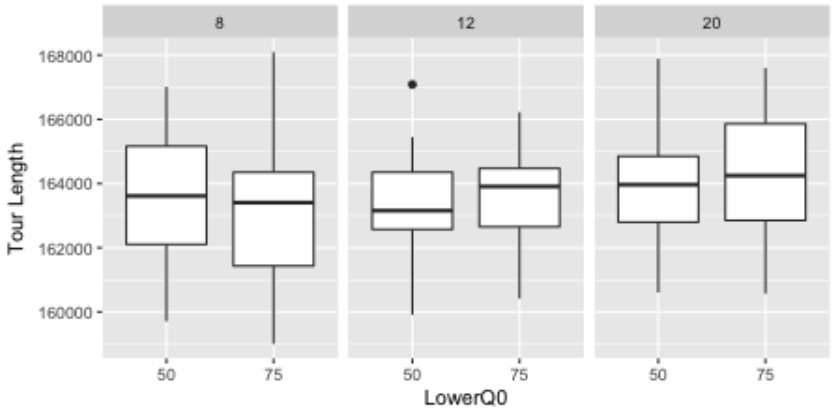

Figure 4.73: Instance pcb3038, tour length, by nAnts, const update strategy, without local search

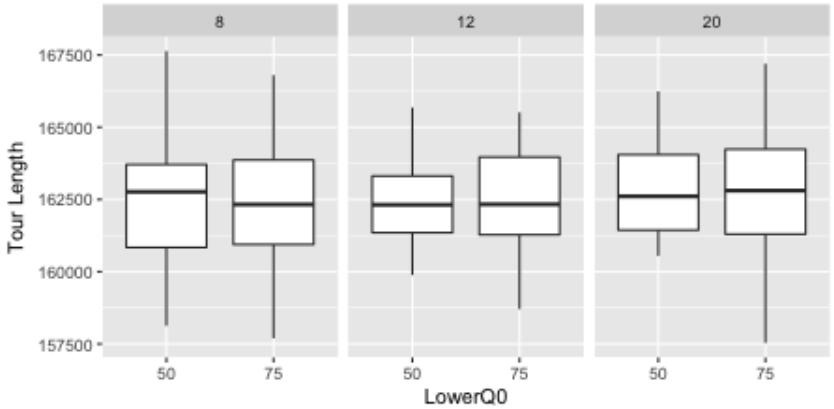

Figure 4.74: Instance pcb3038, tour length, by nAnts, jump update strategy, without local search

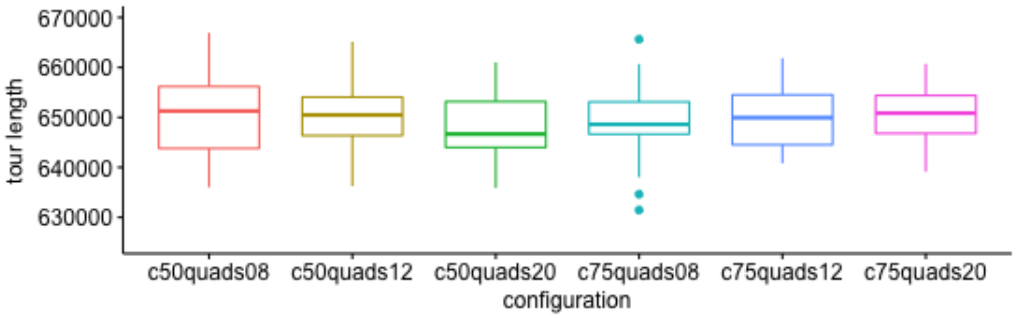

Figure 4.75: Instance rl5934, tour length, by configuration, const update strategy, without local search

samples came from populations with identical variations and the assumptions for the parametric comparison of the means were met. We used the paired two sample t-test, provided by the 'stats' package from the [R, 2011] software to compare the means. The difference between the means was found significant ($p = 6.0 \times 10^{-8}$), so we reject the hypothesis that the sets came from populations with identical means. The average tour length found using the const update strategy was 163672.3, and using the jump update strategy was 162573.0.

rl5934 instance When solving this instance, while the behaviour of the different configurations varied to some degree, there is not a clear pattern of influence by either the colony size or the lowerQ0 value. Figures 4.75 and 4.76 display the results obtained by the const and jump strategies, respectively. The indifference to the parameters, notably the fact that even for const configurations, a smaller colony size is not necessarily desirable, can be explained by the size of the search space. Having a larger pool of solutions to choose from, may compensate for the less frequent trail update.

To compare the average tour length produced by the configurations using the const update strategy against those produced by the jump one, and since both sets follow the normality and equal variance conditions, we used the paired two sample t-test. The difference between the means was found not significant, with $p = 0.46$.

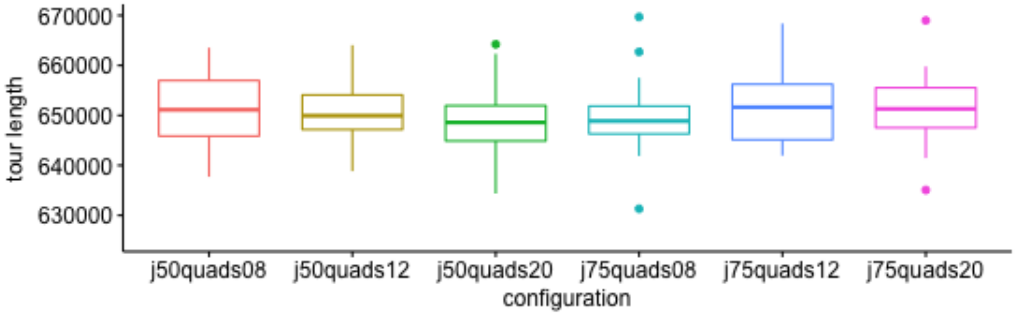

Figure 4.76: Instance rl5934, tour length, by configuration, jump update strategy, without local search

Discussion: Quad caste configurations without local search

Table 4.8 presents a summary of some of the main features of the quad caste configurations performance, when not using local search. It is more difficult to discern patterns of influence when we have four castes. This is not unexpected, since the lowerQ0 only affects one of the four castes, so its impact is reduced. The caste size has a larger influence, as it determines the frequency of the trail update and, in fact, the colony size is usually the most influential factor. The adaptability of the jump migration strategy can be observed on the behaviour of configurations with four castes: on configurations using the jump variant, the total number of ants seems to be less important and frequently all j50quads or all j75quads configurations exhibit the same behaviour.

The preference of const configurations for lower colony sizes, while solving medium to large instances, may be explained by the fact that smaller colonies have a more frequent trail update. In very large instances, the enlarged pool of solutions provided by big colonies may be an advantage. The importance of having a greedy approach in these instances was already established by previous results, namely ACS and dual-caste configurations that, as a general rule, favoured higher q_0 values when solving the same situations.

The jump update strategy frequently lessens the impact of the configuration settings. This is expected, as the ability to adjust the castes size allows it to compensate for the sub-optimal q_0 value of the fourth caste. Also, the ability to increase the number of ants in the more greedy castes when needed, justifies why

instance	update strategy	most influential	lowerQ0 should be	nAnts should be
rat99	const	nAnts	low	20
	jump	nci	nci	nci
d198	const	nci	nci	nci
	jump	nci	low	20
fl417	const	nci	low	8
	jump	nci	nci	nci
rat783	const	nAnts	high	8
	jump	nci	high	12
fl1577	const	nAnts	nci	8
	jump	nci	nci	nci
pcb3038	const	nAnts	nci	8 or 12
	jump	nci	nci	nci
rl5934	const	nci	nci	nci
	jump	nci	nci	nci

Table 4.8: Summary of influential factors for quad-caste without local search

the colony size becomes less important when using the jump update strategy, even on instances that require either a frequent trail update or a more greedy search.

Quad-caste solution quality: with local search

For completeness we repeated the experiments while using local search. Regardless of the instance, each repetition was allowed 1000 evaluations - each ant/solution was evaluated twice per iteration (once after the solution construction and another after local search) - so $1000/2 \times nAnts$ iterations: 62 iterations for colonies with 8 ants, 42 iterations for colonies with 12 ants and 25 iterations for colonies with 20 ants.

The local search algorithm used is effective in solving the simpler instances and, in accordance, the performance of the different configurations is identical. In concrete, for instances rat99 and fl417, the colony size, lowerQ0 and update strategy have little to no influence on the outcome and, as such, we choose to omit the results pertaining to those instances.

Unless explicitly stated, results presented in this section have significant deviations from normality. In accordance, when comparing the performance obtained

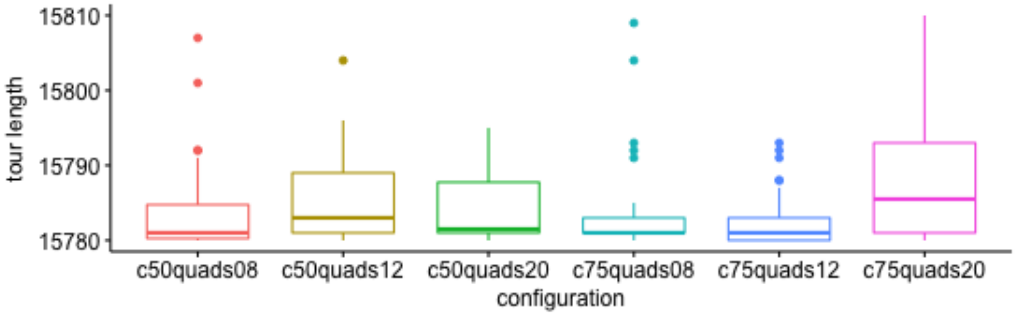

Figure 4.77: Instance d198, tour length, by configuration, const update strategy, with local search

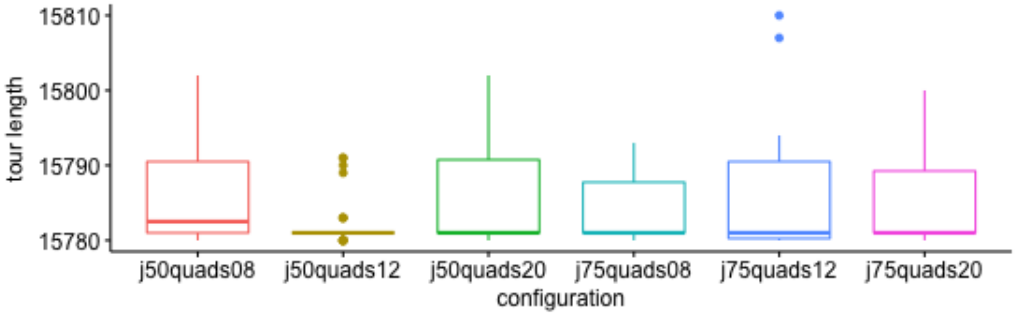

Figure 4.78: Instance d198, tour length, by configuration, jump update strategy, with local search

by the different update strategies, we relied on the Wilcoxon signed rank test with continuity correction (significance level=0.05), as described in the previous sections.

d198 instance The results obtained by the const strategy are presented in Figure 4.77. The most successful configurations are those that have only 8 ants or that have 12 ants, but a higher lowerQ0. Configuration c75quads20 stands out as being poorer than the others. When solving the same instance with the jump update strategy the behaviour becomes more uniform, although j50quads12 is particularly effective in that instance, as can be observed in Figure 4.78. In this instance, no significant difference was found between the results obtained by the two update strategies ($p = 0.43$).

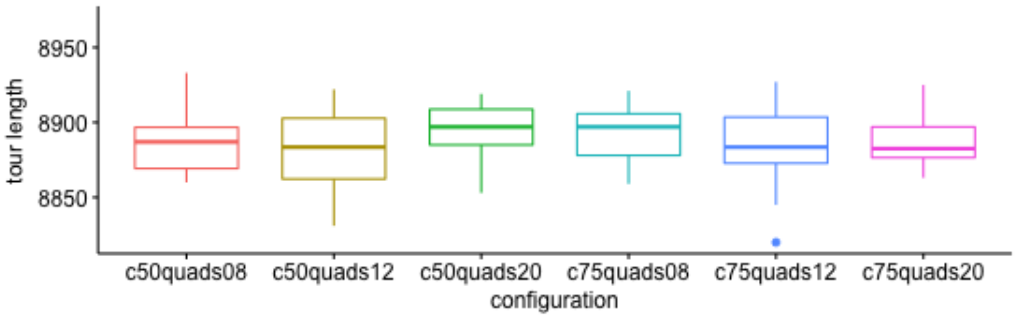

Figure 4.79: Instance rat783, tour length, by configuration, const update strategy, with local search

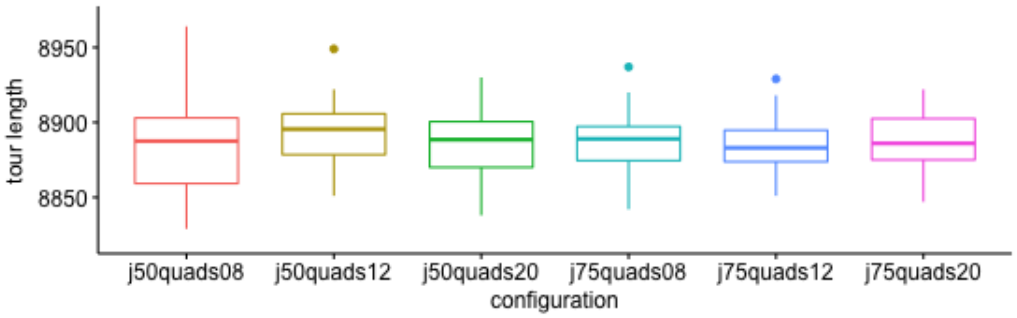

Figure 4.80: Instance rat783, tour length, by configuration, jump update strategy, with local search

rat783 instance The const update strategy exhibits a faint pattern, as can be observed in Figure 4.79. The combinations of a large colony with a low lowerQ0 or of a small colony with a high lowerQ0 should be avoided. As usual, the jump strategy smooths the performances, as can be confirmed in Figure 4.80. In this instance, no significant difference was found between the results obtained by the two update strategies ($p = 0.34$).

fl1577 instance Figure 4.81 reveals that the colony size impacts the results achieved by the const update strategy. When the lowerQ0 is low, small colonies are desirable, but, for a higher lowerQ0, the influence is more subtle and the best choice is a medium sized colony. When using the jump update strategy, as shown in Figure 4.82, the colony size looses influence and most configurations have a similar performance. One notable exception is j75quads20, that is not as successful as the

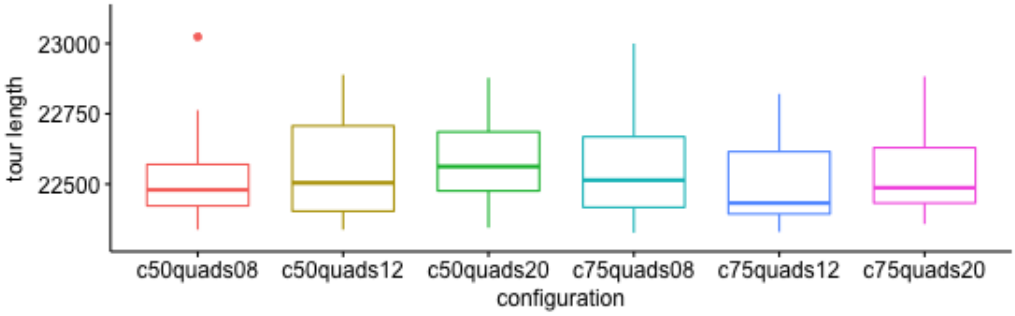

Figure 4.81: Instance fl1577, tour length, by configuration, const update strategy, with local search

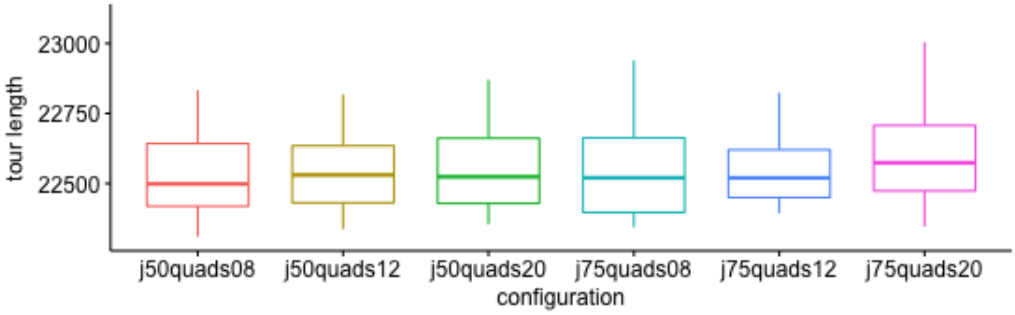

Figure 4.82: Instance fl1577, tour length, by configuration, jump update strategy, with local search

other configurations. Once again, no significant difference was found between the results obtained by the two update strategies ($p = 0.23$).

pcb3038 instance Figure 4.83 depicts the performance of the const update strategy. As a rule, smaller colonies are more successful and, once again, a large colony with a high lowerQ0 is the less successful combination. Since a larger colony means a larger pool of solutions to choose from at each trail update, but fewer trails updates, it is possible that a higher lowerQ0 prevents some useful exploration. For the jump strategy, the colonies with 12 ants and a low lowerQ0 are the most successful, as can be observed in Figure 4.84. Large colonies remain the less effective option.

Since the tour lengths produced by both update strategies follow a normal distribution and have similar variances, we applied the paired two sample t-test.

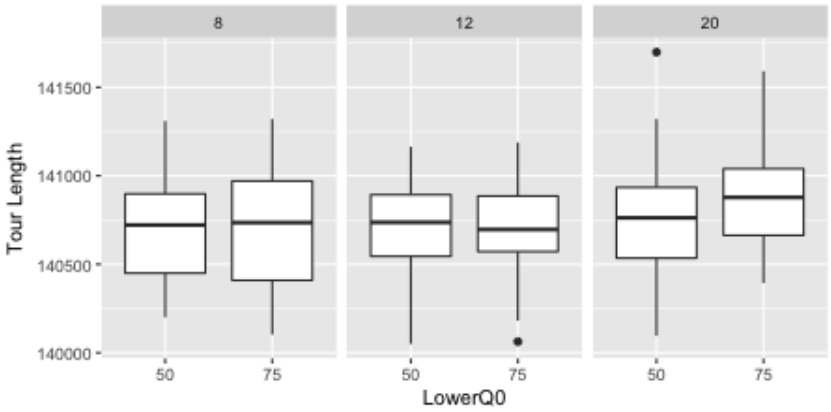

Figure 4.83: Instance pcb3038, tour length, by nAnts, const update strategy, with local search

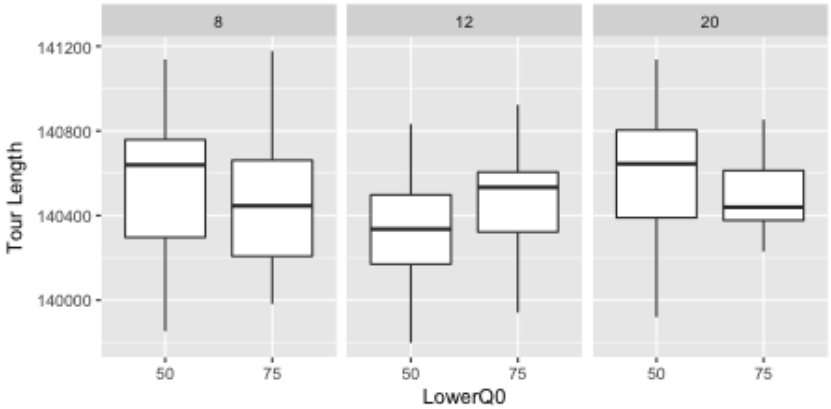

Figure 4.84: Instance pcb3038, tour length, by nAnts, jump update strategy, with local search

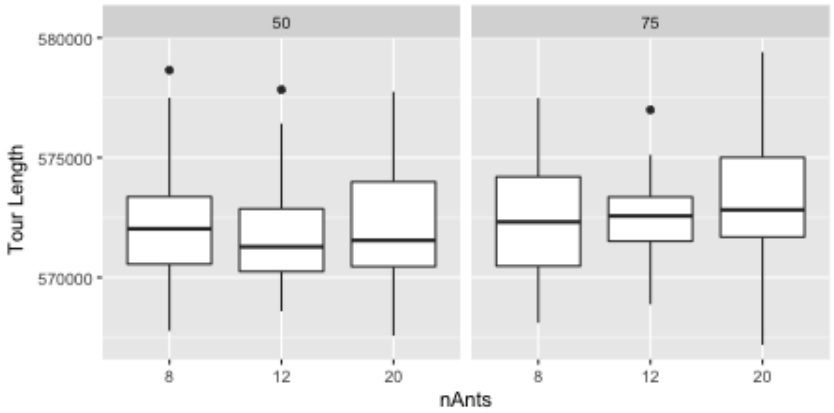

Figure 4.85: Instance rl5934, tour length, by lowerQ0, const update strategy, with local search

The difference was found to be significant with $p = 6.0 \times 10^{-14}$, and so we reject the null hypothesis, and conclude that the jump configurations have an enhanced performance when compared to the const configurations.

rl5934 instance The lowerQ0 is the most important parameter while using the const update strategy. The results can be consulted in Figure 4.85 and we can observe that the influence is mostly visible in the larger colonies. The results obtained by the jump update strategy (Figure 4.86) reveal a reduced influence of the lowerQ0. Colonies of 12 ants are the ones with the best outcome, when compared with other configurations of similar lowerQ0. No significant difference was found between the results obtained by the two update strategies ($p = 0.46$).

Discussion: Qual caste configurations with local search

When using local search, the already subtle differences previously discovered further reduce (consult Table 4.9). As a rule, the colony size tends to have at least a small impact on the performance, particularly on the const configurations. Medium to small colonies are preferable and very large colonies are detrimental.

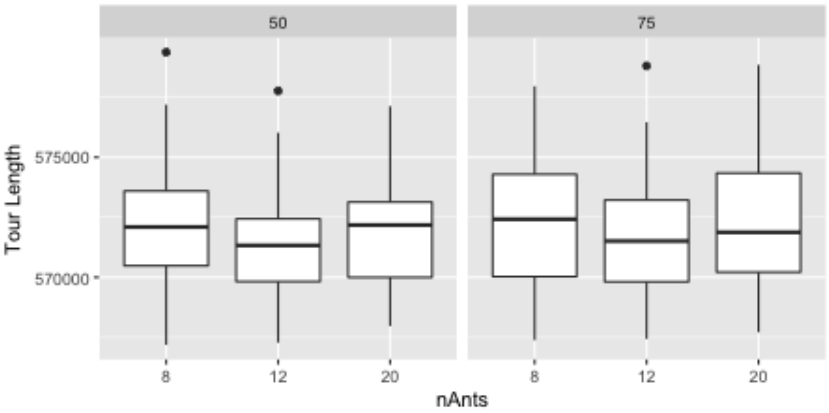

Figure 4.86: Instance rl5934, tour length, by lowerQ0, jump update strategy, with local search

instance	update strategy	most influential	lowerQ0 should be	nAnts should be
rat99	const	nci	nci	nci
	jump	nci	nci	nci
d198	const	nAnts	nci	8
	jump	nci	nci	nci
fl417	const	nci	nci	nci
	jump	nci	nci	nci
rat783	const	nci	nci	nci
	jump	nci	nci	nci
fl1577	const	nAnts	nci	12
	jump	nci	nci	nci
pcb3038	const	nAnts	nci	8 or 12
	jump	nAnts	nci	12
rl5934	const	lowerQ0	low	nci
	jump	nci	nci	nci

Table 4.9: Summary of influential factors for quad-caste with local search

4.4.5 Performance comparison: Const versus Jump

The framework proposed in this chapter aims to gain insight about the possible advantages of allowing for the simultaneous existence of different q_0 values, and the possible further improvement of letting the number of ants using a specific q_0 change over time. In the Tables 4.10 and 4.11 the symbol $c > j$ represents that the average tour length obtained the const update strategy was larger (worse) than the one achieved by the jump update strategy, whereas $c = j$ means that the difference between the means was not statistically significant. The symbol "wo ls" (respectively, "ls") identifies experiments without local search (respectively, with local search).

Const versus Jump dual-caste configurations

Table 4.10 shows that, as a rule, the performance achieved using the jump update strategy is better than or, at least, as good as the performance of the const update strategy. This is most noticeable when not employing local search.

avg length	rat99	d198	fl417	rat783	fl1577	pcb3038	rl5934
wo ls	c = j	c > j	c > j	c > j	c = j	c > j	c > j
ls	c = j	c = j	c = j	c > j	c = j	c > j	c = j

Table 4.10: Performance according to the update strategy for bi-caste configurations

Const versus Jump quad-caste configurations

The pattern previously identified in configurations with two colonies is less clear but still visible when four castes are used. As a rule, the jump update strategy produces results as good as or better than the const update strategy, as can be observed in Table 4.11.

avg length	rat99	d198	fl417	rat783	fl1577	pcb3038	rl5934
wo ls	c = j	c = j	c = j	c > j	c = j	c > j	c = j
ls	c = j	c = j	c = j	c = j	c = j	c > j	c = j

Table 4.11: Performance according to the update strategy for quad-caste configurations

4.4.6 Performance comparison: ACS versus multi-caste ACS

We compared the quality of the solutions obtained by selected multi-caste and conventional ACS configurations. The main goal is to ascertain if the multi-caste configuration chosen is able to obtain, on average, results as good as the ACS ones.

We consider the two most successful ACS configurations, c95 and c99 and a multi-caste configuration j95_99. We choose c95 and c99, since for 11 of the 14 scenarios tested (7 instances with and without local search), either one those configurations had the smaller average tour length among the ACS. Only in instances rat99 without local search and d198, configuration c90 had a slightly lower average tour length than c95 or c99. The multi-caste configuration was selected since the jump strategy, as a rule, tends to exhibit enhanced robustness. Also this configuration contains both q_0 values of the two standard ACS selected. This way we can verify if multi-caste frameworks can profit from having different coexisting q_0 values along the optimisation.

The samples - sets with the 30 tour lengths produced by each configuration for each instance - are independent and numeric so, to decide which test should be used to compare the means, we followed the recommended criteria [Zar, 2009]:

- Select ANOVA if the samples have identical or similar variances and are normal to moderately non-normal;
- Select Kruskal-Wallis if the samples have identical variations and similar distributions, but are very different from normally distributed.

The samples were tested for normality both visually and using the Anderson-Darling normality test (ad.test) provided by the 'nortest' package of the [R, 2011]

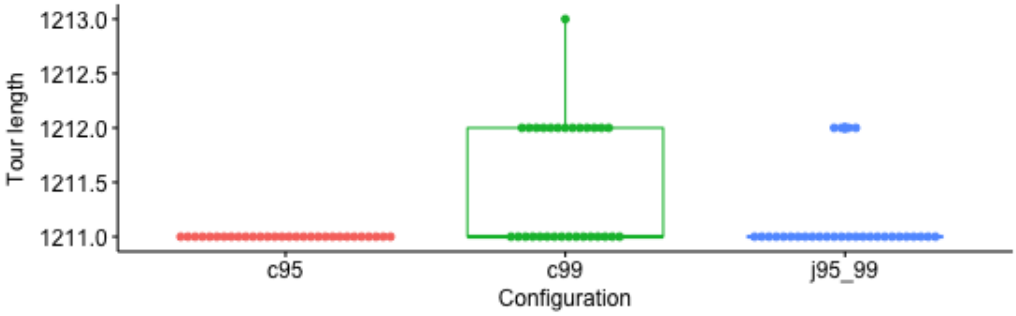

Figure 4.87: Instance rat99, with local search, tour length

software, with a significance level of 0.05. Regarding the results obtained when not using local search, only in instance rat99 one of the configurations presents deviations from normality. In the results achieved with local search, only the results from the instances rat783, pcb3038 and rl5934, do not present deviations from normality in any of the configurations. Within each group of samples, the shapes of the distributions were similar except for the results produced for rat99 with local search. A box and dot plot of the results achieved for the rat99 with local search can be consulted in Figure 4.87.

The groups of samples were tested for homogeneity of variance using the Levene test (leveneTest) provided by the 'car' package of the [R, 2011], with a significance level of 0.05. We selected the Levene test since it is less sensitive to departures from normality [Zar, 2009] and some of our samples are not normal. Only the solutions generated while solving the rat99 instance using local search were found to have significant difference between the variances.

To compute the Kruskal-Wallis statistic we used the kruskal.test function from the 'stats' package, and for ANOVA we used the aov function from the 'stat' package, both from the [R, 2011] software with a significance level of 0.05. We are interested in comparing each of the configurations with the best ACS configuration for that instance. To do so, when the Kruskal-Wallis test found a significant difference between the means, we use the multiple comparison test ('two-tailed' comparison, treatments versus control) after Kruskal-Wallis test, the "control" being the ACS configuration with smaller average tour length for that instance. We used the kruskalmc function from the 'pgirmess' package of the [R, 2011] software

	rat99	d198	fl417	rat783	fl1577	pcb3038	rl5934
c95							
c99							
j95_99							
test	KW	ANOVA	ANOVA	ANOVA	ANOVA	ANOVA	ANOVA

Figure 4.88: Mean tour length comparison without local search

with a level of significance 0.05. In the case where we used the ANOVA, and there was a significant difference between the means we used a post-hoc Dunnett test to determine if the means obtained by the various configurations were significantly different from the best ACS configuration for that instance. The computations were done using the glht function form the 'multicomp' package of [R, 2011] software.

The conclusions are depicted in Figures 4.88 and 4.89. The first line indicates the problem instance, the last line the test used to compare the means (KW for Kruskal-Wallis) and the first column contains the name of the configuration that line pertains to. Cells in green identify situations where the configuration in that line is not significantly different from the best ACS configuration. Conversely, cells in red highlight results significantly different from the best.

When not using local search, (Figure 4.88) we can observe that c99 was twice inferior to the best ACS (fl417 and fl1577), and the same happens for c95 (instances pcb3038 and rl5934). On the contrary, the dual-caste j95_99 is never inferior to the best ACS configuration.

When using local search, the difference in the shape of the distribution of the samples for instance rat99 are clearly different. However, it is clear from Figure 4.87 that c99 performance is inferior to those of the other two configurations. Results for the other instances are presented in Figure 4.89. In these instances, c99 is worse than the best ACS once (rat783) and so is c95 (pcb3038). Once again, j95_99 levels up with the best ACS. In conclusion, for every instance tested, dual-caste j95_99 was able to perform just as well as the best of the two selected ACS configurations. This is a remarkable example of the enhanced robustness of the multi-caste framework.

	rat99	d198	fl417	rat783	fl1577	pcb3038	rl5934
c95							
c99							
j95_99							
test		KW	KW	ANOVA	ANOVA	ANOVA	ANOVA

Figure 4.89: Mean tour length comparison with local search

4.4.7 Relative error

To better understand the relationship between the configurations, we also calculated the relative error averaged over the 30 repetitions. The relative error at any given iteration is computed with equation 4.3:

$$E_i = \frac{e_i - s^*}{s^*} \tag{4.3}$$

Where e_i represents the best-so-far solution at iteration i, and s^* represents the optimum.

To simplify the analysis, in this section we consider only 8 of the more successful configurations: three ACS configurations (c90, c95, c99), three dual-caste configurations (c95_99, j75_99, j95_99) and three quad-caste (c75quads12, j50quads08, j75quads12).

Relative error without local search

Relative error box-plots To build the box-plots we considered the best-so-far solution by the end of the run. In this section we will present a brief overview of the main conclusions. The complete box-plots with the selected configurations can be consulted in Appendix A.

The results for the smaller instance reveal that Multi-caste ACS tends to have smaller interquartile ranges than ACS. We attribute it to the capacity of avoiding premature convergence given by the added diversity. Also the median value of the Multi-caste ACS configurations does not exceeds those of ACS. On median size instances (d198, fl417 and rat783), Multi-caste ACS is usually not as good as the best ACS configuration. However, multi-caste configurations are always better than the worst standard ACS. In these instances, ACS configurations have similar

performance and the relative error is kept small.

As expected, since the number of iterations remains constant, the relative error increases with the size of the instance. For larger instances, like fl1577, pcb3038 or rl5934, Multi-caste ACS configurations are still able to keep close to the better ACS configuration, even though the best ACS configuration is not always the same.

Relative error over time The average relative error evolution over time for instances d198, rat783 and pcb3038, are depicted in Figures 4.90, 4.91 and Figure 4.92. The number of evaluations (horizontal axis) is depicted in logarithmic scale and the range of the relative error (vertical axis) varies according to the problem instance.

The ACS configurations that reached the lowest relative error were the ones with $q_0 = 0.90$ for d198, $q_0 = 0.95$ for rat783 and $q_0 = 0.99$ for pcb3038. Although the optimal q_0 differs, the robustness of the selected Multi-caste ACS can be observed.

The outcomes reinforce the general conclusion that the optimal value of q_0 changes with the problem instance. Even staying within the range [0.9, 0.99], the differences in performance can be clear. The coexistence of several q_0 values in a single algorithm allow Multi-caste ACS to exhibit a more robust behaviour. In any case, it is worth noting that the absolute performance of this framework depends on several factors: the combination of the q_0 values employed; the migration strategy and, as a rule, the ability to change the relative size of the castes is an advantage; the number of castes/ants per caste, as this impacts the tradeoff between number of iterations and trail updates.

Relative error with local search

Relative error box-plots The box-plots comparing the relative error of the selected configurations can be consulted in Appendix B,

As expected, the relative error on the smaller instances is very small and all the selected configurations have a similar behaviour. The only configuration that stands out is c99 on the rat99 instance. On medium sized instance (rat783 and fl1577), all configurations tends to have a similar average error and inter-quartile range. For large instances (pcb3038 and rl5934), standard c90 is clearly the worst

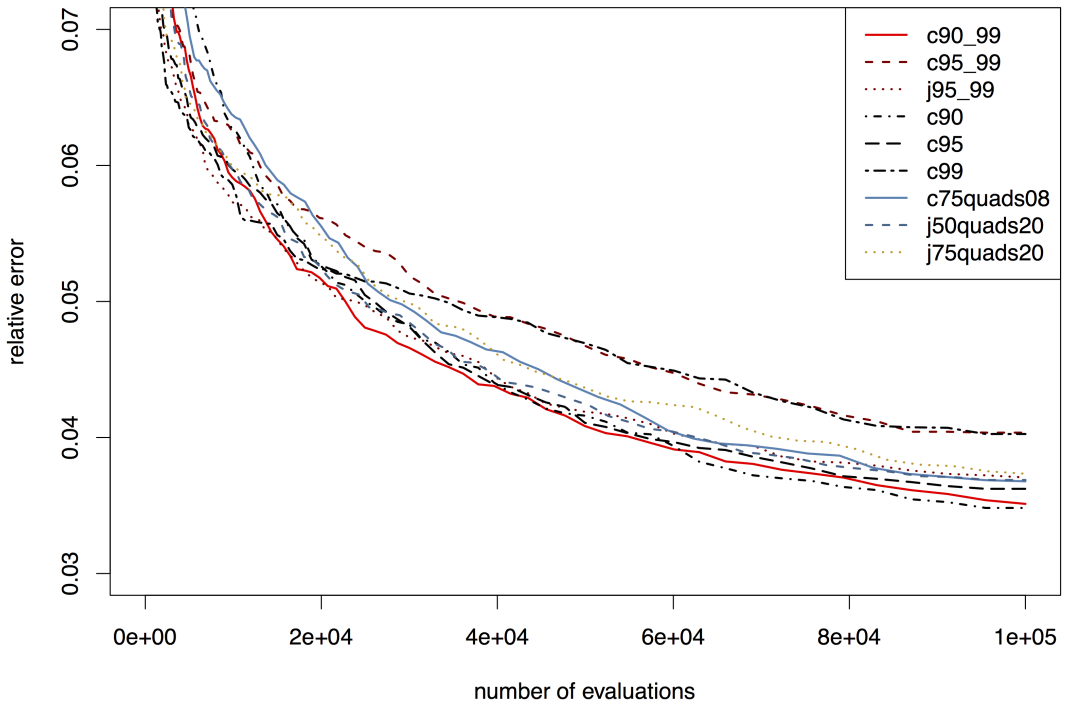

Figure 4.90: TSP d198, selected configurations, relative error over time, no local search

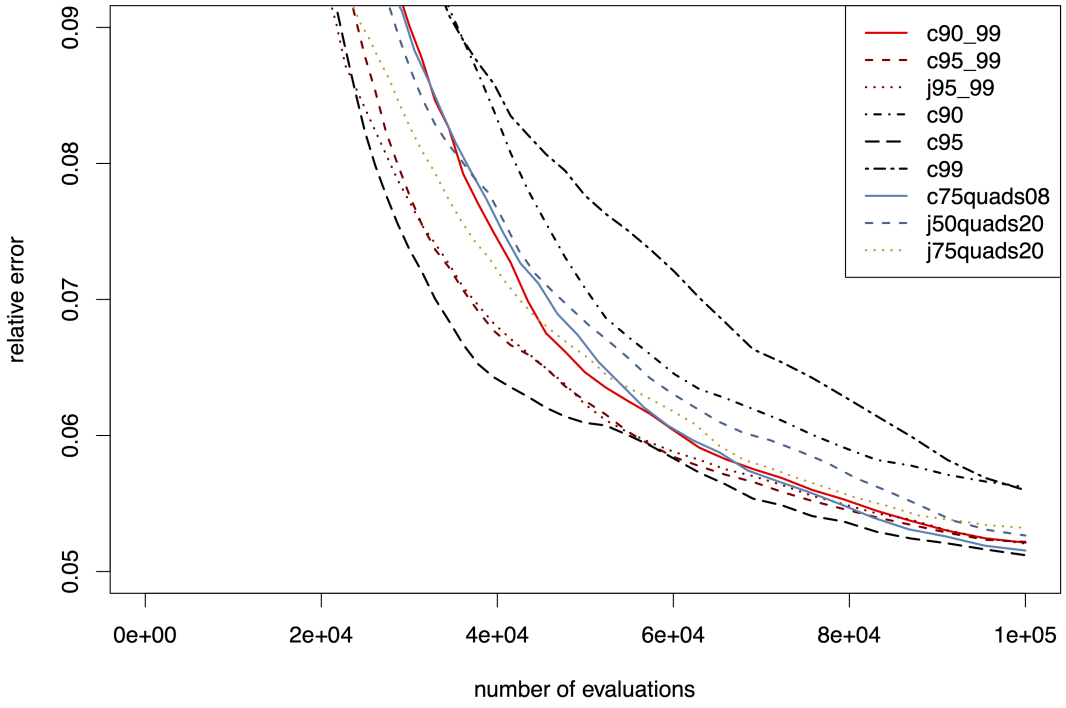

Figure 4.91: TSP rat783, selected configurations, relative error over time, no local search

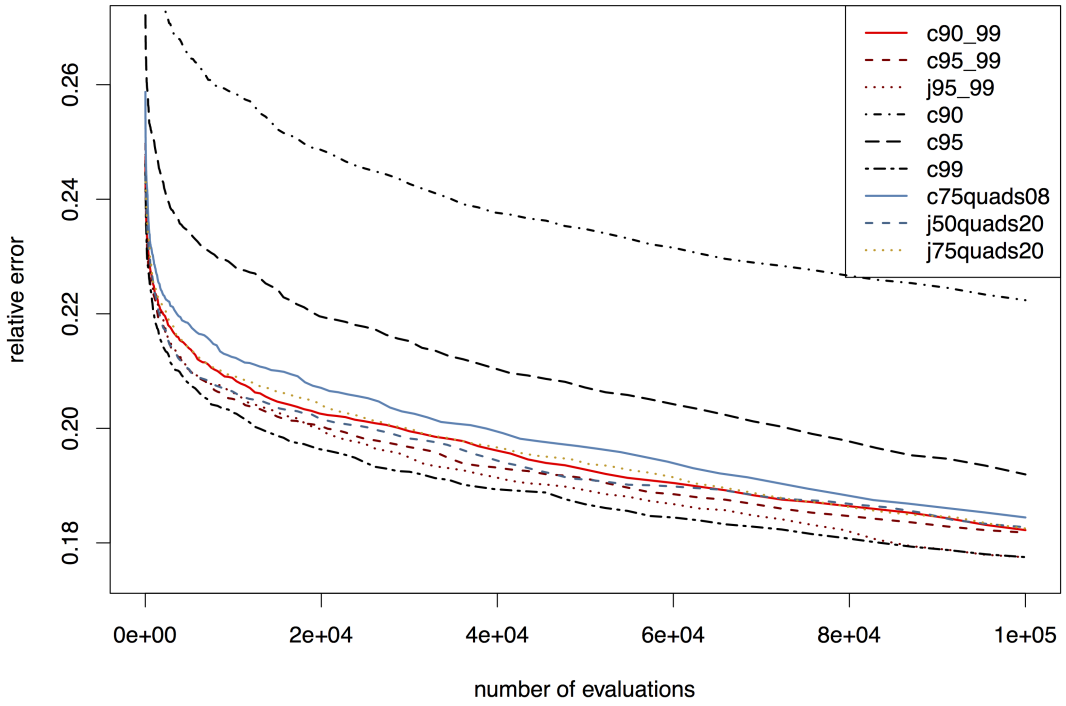

Figure 4.92: TSP pcb3038, selected configurations, relative error over time, no local search

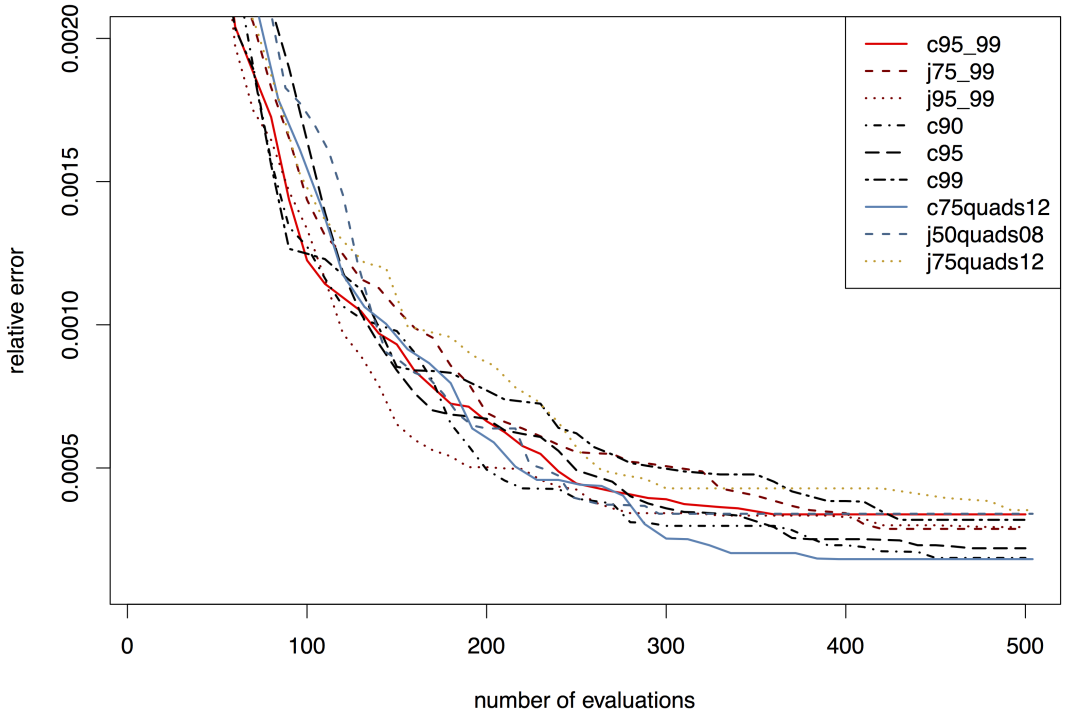

Figure 4.93: TSP d198, selected configurations, relative error over time, with local search

configuration, whereas c99 is the best one. The selected Multi-caste ACS configurations tend to exhibit a behaviour delimited by the two best performing ACS variants and clearly better than c90.

Relative error over time On Figures 4.93, 4.94, and 4.95 we present the evolution of the average relative error over time for instances, d198, rat783 and pcb3038. As when not using local search, the very best ACS configuration depends on the problem instance, being c90 for d198, c95 for rat783 and c99 for pcb3038. Still, even if the optimal q_0 differs, the selected Multi-caste ACS are robust and remain close to the best performing configurations.

As a general conclusion, when using local search, the average error is smaller,

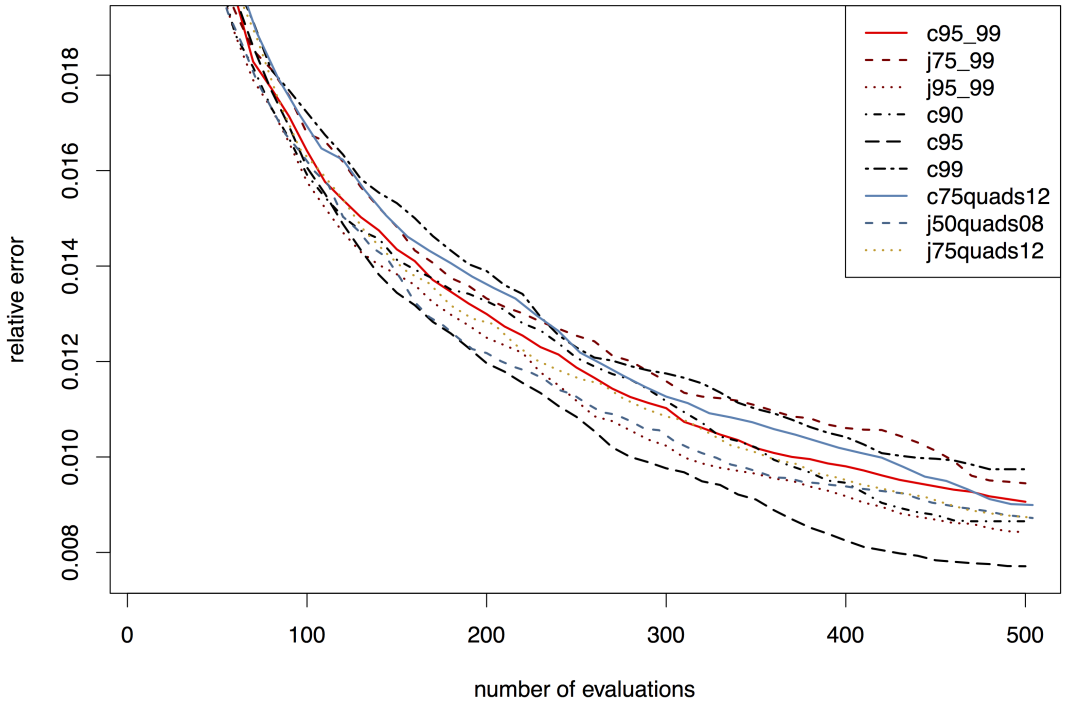

Figure 4.94: TSP rat783, selected configurations, relative error over time, with local search

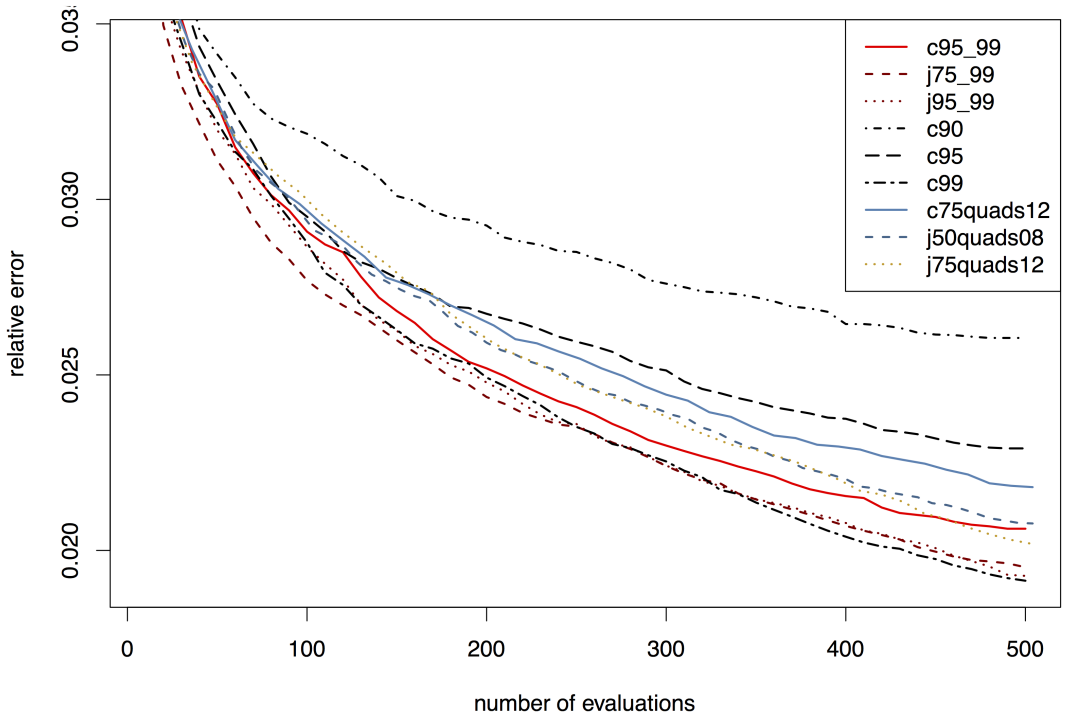

Figure 4.95: TSP pcb3038, selected configurations, relative error over time, with local search

but the q_0 parameter still has impact. The most successful q_0 values are in the range [0.95, 0.99]. By combining these values in a Multi-caste ACS approach, the algorithm becomes more robust.

4.5 Conclusion

A multi-caste approach, with several coexisting q_0 values, was presented in this chapter. The relevance of specific factors, such as the number of castes, the values of the q_0 in each caste, and update strategy, was studied using several symmetric instances of the TSP.

When not using local search, the influence of the q_0 value on the solution quality achieved by the algorithm is more evident. Results suggest that the jump migration strategy, by allowing ants to move from one caste to another, further enhances the robustness of the algorithm and allows for an autonomous adaptation to the different search stages.

Interestingly, should local search be employed or not, the more robust configurations tended to be those with high q_0 values, namely c95_99 and j95_99. This is an expected result, since the most successful ACS configurations also tended to be either c95 or c99. Clearly the multi-caste approached combined the abilities of the individual variants, allowing for the colony to benefit from the qualities of both configurations.

As for configurations with four castes, the const strategy usually performs better if using a total of 8 or 12 ants. When using the jump strategy, the total number of ants becomes less important, possibly because the ability to migrate the ants allows the algorithm to self-adjust its behaviour.

We also presented a statistical analysis that investigated if the multi-caste approach is a more robust alternative to the conventional ACS. The results show that j95_99 indeed provides enhanced robustness, when compared to either c95 or c99.

5

Multi-caste ACS: the dynamic case

We have seen in the previous chapter that the optimal ACO settings may change with the problem instance and even over time, according to the search stage. When facing a dynamic problem, that can be viewed as a succession of problem instances over time, this difficulty is even more noticeable. Another drawback of ACO, the inability to continue looking for new solutions once the algorithm converges, is also particularly undesirable in dynamic environments. In order to autonomously adapt to change, the algorithm must explore the new environment after each change, and thus the ability to maintain its explorative faculties are crucial.

By varying q_0, we can have coexisting ants with different search strategies, either more exploitative of the present trail or more exploratory of new regions. This is specially useful when tackling a dynamic situation since, after each change, even if the problem representation is kept the same, the optima will likely become different. The knowledge gathered before may be more or less useful, according to the intensity of the change. Retaining, and even autonomously increasing, the ability to explore after a change, without the need to restart the algorithm and without loosing all the acquired knowledge, is a desirable trait, both to react to change and to prevent premature convergence. The restart of the algorithm is only a good option when the change is so extreme that the knowledge previously acquired becomes misleading.

5.1 Modifications to the Multi-caste ACS architecture

The Multi-caste ACS algorithm, revised for dynamic environments, is presented in Algorithm 5.1.

```
Data: sequence of problem instances, parameters
Result: sequence of best-so-far solutions
begin
    load the first instance
    set general parameters
    initialise the pheromone trail
    foreach caste do set caste's parameters
    while termination condition not met do
        if change occurred then
            make change
        end
        construct ant solutions
        apply local search (optional)
        adjust castes' size
        update pheromone trail
    end
end
```

Algorithm 5.1: Multi-caste ACS algorithm, dynamic version

When applied to dynamic problems, every time a change occurs, the *make change* procedure is run. The change is represented as a new instance of the problem being solved so, during this step, the new instance is loaded and the candidate list is recomputed. Also the best-so-far solution quality must be re-evaluated according to the new problem instance. This is required since a given solution in a new environment has usually a different quality than it had in the previous one. Keeping the outdated quality value could prevent the algorithm from updating the best-so-far ant correctly, and thus a sub-optimal solution would be used to reinforce the trail.

5.1.1 Migration strategies

The update strategy, namely the ability of changing the caste size over time, was a crucial success factor of the original Multi-caste ACS. In dynamic environments, the ability to adapt the size of the caste is likely even more relevant. Accordingly, two new migration mechanisms are considered.

SuperJump Selects a random ant from each caste, and, from this subset, determines the solutions with best and worst quality obtained in the most recent iteration. If these qualities are different and the caste of the ant with the poorer solution has, at least, size 2, then migration can occur. This way, the ant that achieved the poorer solution is transferred to the caste of the ant that obtained the best solution. This approach is the one that encourages migration the most since, even if a caste is reduced to just one ant, it will still be selected for comparison. Also, because the ants are chosen at random, the size of the castes becomes less relevant.

GreedyJump From each caste, selects the ant that produced the solution with the best quality in the current iteration. From this subset of ants, it determines the solutions with best and worst quality. If these qualities are different and the caste of the ant with the poorer solution has, at least, size 2, then migration occurs just like in the SuperJump strategy. The motivation for the greedy approach is to make the mechanism more sensitive to very good results from any given caste.

5.2 Application of the Multi-caste ACS to the dynamic TSP

Dynamic problems need robust algorithms that are able to keep diversity. An approach like Multi-caste ACS is well equipped to deal with dynamic environments, as it comprises several coexisting search strategies and self-adaptive mechanisms to adjust the exploitation/exploration tradeoff. To test our proposal we selected a dynamic version of the symmetric TSP.

5.2.1 Motivation

The two most important factors to consider in dynamic problems are the magnitude and frequency of change. When using ACO to solve a dynamic problem, should the magnitude of change be extreme, the restart of the algorithm is the best option as the knowledge acquired in the previous environment becomes invalid in the new one. But when less severe changes occur, the new best solution after a change is likely related to the old one and, thus, reusing information can be beneficial [Guntsch et al., 2001]. Still, the amount of information inherited from the previous environment should be enough to guide the search procedure, but not so much as to disable the algorithm from exploring other regions. The q_0 parameter determines how strongly the information is exploited. By having the possibility of dynamically adjust this value, Multi-caste ACS is able to avoid starting the search too near the old solutions and thus escape local optima, while preserving the exploitation ability once a new promising direction is found. The frequency of change determines, indirectly, the amount of information gathered, and limits the time available to optimise a given environment.

5.2.2 The dynamic TSP

DTSP was first described in [Psaraftis, 1988]. Two main types of dynamism can be added to the TSP: adding and removing cities to the problem instance and changing the cost between pairs of cities. In the literature we can find work done on each type of dynamism and even on the combination of both:

- Adding and removing cities [Guntsch and Middendorf, 2001, Guntsch et al., 2001, Angus and Hendtlass, 2002, Guntsch and Middendorf, 2002b, Angus and Hendtlass, 2005, Sammoud et al., 2009, Mavrovouniotis and Yang, 2010, Mavrovouniotis and Yang, 2011a, Mavrovouniotis et al., 2017a];

- Changing the cost between pairs of cities [Eyckelhof and Snoek, 2002, Liu, 2005, Mavrovouniotis and Yang, 2011c, Mavrovouniotis and Yang, 2011b, Simões and Costa, 2011, Mavrovouniotis and Yang, 2013b, Simões and Costa, 2013, Mavrovouniotis and Yang, 2014c, Mavrovouniotis and Yang, 2014b,

Mavrovouniotis et al., 2014, Mavrovouniotis et al., 2015, Wang et al., 2016, Strąk et al., 2017, Mavrovouniotis et al., 2017b];

- A combination of both ([Zhou et al., 2003, Li et al., 2006, Li, 2011]).

We will address DTSP where the cost between pairs of cities changes over time. In the literature this modification is also known as inserting traffic jams or adding a traffic factor. Moreover we will be using the periodic, non-cyclic version of it.

There is no widely accepted benchmark for DTSP, as most works create their own test sets. When evaluating our algorithm we created an array of scenarios of varying magnitude and frequency of change that aim to ascertain the impact of each of these factors. The problem instances were built in a similar manner as proposed in [Mavrovouniotis and Yang, 2011b]. For each pair of cities, i and j, e_{ij} represents the weight associated with the edge (i, j) and is calculated as $e_{ij} = d_{ij} \times f_{ij}$, where d_{ij} is the original distance between i and j, and f_{ij} is the traffic factor between those cities. Every F evaluations, a random number uniformly distributed over $R \in [F_L, F_U]$ is generated where R represents the traffic or delay at that moment, and F_L and F_U are the lower and upper bounds for the traffic. With probability M the link (i, j) can change its traffic factor, f_{ij}, to $1 + R$ or, otherwise, reset it to 1 (meaning no delay). The magnitude of change is represented by M, and F controls the number of iterations between changes, i.e., its frequency. By changing M and F we may create different dynamic scenarios and study how our algorithm responds to them.

Some important definitions about the DTSP with traffic jams, considering the taxonomy proposed in [Branke, 2002], are:

- Visibility of change: like other ACO approaches to the DTSP, Multi-caste ACS assumes that the change is made known to the algorithm and explicitly reacts to it (loading the new problem instance, re-computing the candidate list and re-evaluating the best-so-far solution). If the problem and candidate list could be altered without the algorithm knowledge, then a simple strategy would be to re-evaluate the best-so-far solution at each iteration.

- Necessity to change the representation: for the DTSP, a change in the traffic implies a change in the heuristic information, but the representation can be

maintained.

- Aspect of change: a change in the DTSP may be reduced to a new problem instance.
- Algorithm influence on the environment: a solution is built as a whole, and no change can occur while it is being created. As such, a solution built before a change, imposes no restriction on a solution built after the change.

5.2.3 Existing approaches

The existing ACO approaches to the DTSP, can be briefly described as:

- In [Guntsch and Middendorf, 2001], three pheromone modification strategies are compared: restart every pheromone value by the same degree, equalise all the edges incident to a new city j according to the heuristic value ($\eta - strategy$), or according to the trail information ($\tau - strategy$).
- [Guntsch et al., 2001] combines the strategies presented in [Guntsch and Middendorf, 2001] and adds a new Elitist Ant, which is computed form the best-so-far solution. The cities that were removed are deleted from the solution and the cities added are inserted in the place where they cause the minimum increase in length.
- To promote exploration after the change, [Eyckelhof and Snoek, 2002] does not allow the pheromone value of a trail segment go bellow a given value τ_0. It also shakes the trail if a segment pheromone value becomes much higher than the other segments leaving the same city. Shaking changes the pheromone on several segments, while preserving the relative order.
- To speed the adaptation, [Angus and Hendtlass, 2002] smooth the trail after each change. For each city, the original pheromone levels are divided by the highest pheromone concentration considering all of the edges incident to that city.
- [Guntsch and Middendorf, 2002a] presents P-ACO a variant without evaporation and with a population/memory of limited size. Every iteration, the

best-so-far ant enters the population and reinforces the trail, whilst the older ant partially erases the trail and leaves the colony.

- [Guntsch and Middendorf, 2002b] adapts P-ACO adding a KeepEllitist procedure that consists in applying the ElitististAnt from [Guntsch et al., 2001] to repair solutions from the population/memory after cities are inserted and deleted. The trail is reinitialised with respect to this new population.

- A rank based ACO, with a rank-based nonlinear selective pressure function and a modified Q-learning method, is presented in [Liu, 2005]. The first component alters the construct ant solutions rule to emphasise the difference between the amount of pheromone in the segments when the trail is faint. The trail update is reinforced and combined with standard Q-learning.

- [Sammoud et al., 2009] use a strategy similar to P-ACO, with a population of solutions that serve as memory of the best-so-far found solutions. Contrary to P-ACO, this set is not used for updating the trail, but only to rebuild the trail after a change. When a change occurs, the population is repaired applying the KeepEllitst from [Guntsch and Middendorf, 2002b]. The new trail is tested for relevance and adjusted according to the quality of solutions it produced. After recalibration, the algorithm resumes as usual until the next change.

- [Mavrovouniotis and Yang, 2010] is also an adaptation of P-ACO, but with a short-term memory: the ants are replaced each iteration, so there is no need for the repair mechanism, and a percentage of the worse performing ants are replaced by immigrants. Three kinds of immigrants are introduced: RIACO (randomly generated ants), EIACO (elitist ants are mutated versions of the best ants from previous generation), HIACO (hybrid immigrants: half are random and the other half are elitist).

- The approach proposed in [Mavrovouniotis and Yang, 2011c] is similar to the one in [Mavrovouniotis and Yang, 2010], but suited for cyclic changes. It uses a new king of immigrant: MIACO that relies on a long term memory to preserve the best ant form previous environments. Ants in the long term

memory are reevaluated each time a change occurs. Every iteration, immigrants are created based on mutations of the best long term memory ant. At the end of the iteration, the best so-far ant replaces an ant in the long term memory using a proximity metric.

- EIIACO algorithm presented in [Mavrovouniotis and Yang, 2011b] is another extension of [Mavrovouniotis and Yang, 2010], but this time the immigrants are generated using environmental information (the ants from the previous iteration). All the ants stored in the population/memory are used as a base to probabilistically construct the immigrants.

- [Mavrovouniotis and Yang, 2011a] propose MACO, a memetic ACO, that hybridises P-ACO and the KeepEllitist procedure, with a local search method (inver-over operator). The Inver-over operator generates an offspring by inverting a segment of the parent tour (either selecting a random segment or using information from solution). The best between the parent and the offspring remain in the population. Random immigrants (RIACO from [Mavrovouniotis and Yang, 2010]) are used when the population/memory diversity falls below a given threshold.

- In [Mavrovouniotis and Yang, 2013b], a comparative study of RIACO, EIACO [Mavrovouniotis and Yang, 2010] and MIACO [Mavrovouniotis and Yang, 2011c] is presented.

- [Mavrovouniotis and Yang, 2014a] propose two hybrid immigrants schemes: non-interactive and interactive. The immigrants are a mix of RIACO and EIACO [Mavrovouniotis and Yang, 2010] immigrants. In the non-interactive scheme, each generation half of the immigrants are EIACO and the other half RIACO. In the interactive approach, EIACO immigrants are used by default, but, if the algorithm stagnates, RIACO immigrants are selected.

- [Mavrovouniotis and Yang, 2014b] propose a self-adaptive evaporation mechanism to cope with change: low evaporation corresponds to slow adaptation, while high evaporation means faster adaptation. Each ant chooses its own

ρ. Two trails exist, corresponding to the different ρ values. The ants are responsible, not only to build the solution, but also to choose the ρ value.

- A multi-colony approach is presented in [Mavrovouniotis et al., 2014]. Each colony has its own trail and evaporation rate. Every time a new global best-so-far ant appears, it is used to do an extra reinforcement on all the trails.

- [Mavrovouniotis et al., 2015] propose *MM*AS_US, an ACO memetic algorithm: the pheromone update policy is based on the MMAS and local search is based on the unstringing and stringing (US) operator. The best-so-far solution is provided to the US local search procedure before the trail update. US operates by removing (unstringing) two cities and inserting (stringing) them into such positions that improve the overall tour cost.

- NSACO proposed in [Wang et al., 2016] also hybridises P-ACO with the KeepEllitist procedure, with three local search procedures (swap, insertion and 2-opt), to optimise the solutions found by ants.

- [Mavrovouniotis et al., 2017a] extend *MM*AS_US to contemplate also the asymmetric DTSP. The new US operator is applied only when a new best ant is discovered, as no improvement is possible after the application of the local search operator (local search is applied until no further optimisation is possible).

- [Mavrovouniotis et al., 2017b] adapt the trail information after each change, with a heuristic based pheromone strategy. If after a change, the heuristic information pertaining a segment decreased (the cities became closer), then the pheromone level is increased; inversely, if the heuristic information increased, then the pheromone value is decreased.

Several dynamic scenarios have been explored by ACO, when testing with DTSP. They can vary along the following dimensions:

Frequency of change from short to long intervals between changes, from 10 iterations [Melo et al., 2013b, Melo et al., 2013a, Melo et al., 2014], to 750

iterations([Guntsch et al., 2001, Guntsch and Middendorf, 2002b]); either a single, or a couple, of changes ([Guntsch and Middendorf, 2001, Eyckelhof and Snoek, 2002, Liu, 2005]), or somewhere in between;

Severity of change from 0.5%([Guntsch et al., 2001, Guntsch and Middendorf, 2002b]) to 75%([Mavrovouniotis and Yang, 2010, Mavrovouniotis and Yang, 2011a]) of the cities added/removed; from 1%([Eyckelhof and Snoek, 2002, Liu, 2005]) to 75%([Mavrovouniotis and Yang, 2011c, Mavrovouniotis and Yang, 2011b, Melo et al., 2013b, Melo et al., 2014]) of the edges affected by traffic jams;

Cycle length and accuracy in some scenarios it is known that the system will return to a previous situation and this is explored by [Mavrovouniotis and Yang, 2011b, Mavrovouniotis and Yang, 2013b];

Visibility of change all algorithms, with the exception of [Liu, 2005], need to know explicitly when a change occurred. This triggers some special actions, from simple tasks like reevaluating the best-so-far ant to be consistent with new environment, to trail adjustments or extra local search procedures applied to one or more ants.

Also, although there is sometimes a pattern as to the time at which the change will occur (predictability of change), no algorithm explores it. The dimension of the instances used to test the algorithms varies from 14 cities [Angus and Hendtlass, 2002] to those with 1173 [Melo et al., 2014]. The different properties of the instances where these approaches were applied makes it impossible to directly compare them.

5.3 Experiments

The number of cities is one of the factors that helps to define the hardness of a TSP instance. With the goal to test our approach under scenarios of varying difficulty, we chose 7 symmetric TSP instances of varying size from the TSPLIB 95 [Reinhelt, 1995]: kroA100, kroA200, att532, rat575, rat783, pr1002 and pcb1173. These

10 ants	8 ants	12 ants
10	12 or 13	8 or 9
20	25	16 or 17
100	125	83 or 84
200	250	166 or 167

Table 5.1: Number of iterations between changes

examples have been frequently used in the DTSP literature, with the exception of pr1002 and pcb1173 that were considered to expand the study to large instances.

For each instance, we consider 20 dynamic scenarios (4 values of $F\times$ 5 values of M); $F = \{10, 20, 100, 200\}$, where $F = 10$ defines a rapid changing environment and $F = 200$ represents a slow changing environment; $M = \{10, 25, 50, 75, 90\}$, with $M = 10$ and $M = 90$ establishing a small and large degree of change, respectively. These ranges were chosen to provide an ample set of scenarios with different characteristics. The traffic factor ranged from $F_L = 0$ to $F_U = 5$, as these are the values usually found in the literature. For each pair, (instance, M value), 900 variants of the instance were created (30 variants per run $\times$ 30 runs). Each independent run was allowed to run for $30 \times F \times 10$ evaluations.

To simplify the analysis, in the results we refer to the interval between changes as being 10, 20, 100 or 200 iterations, but for the configurations with 8 or 12 ants/evaluations per iteration, there are small adjustments to this number. The general rule is that a change occurs if the total number of evaluations is at least $i \times F \times 10$ ($i = 1, ..., 29$). The precise values can be consulted in Table 5.1.

Unless otherwise stated, the settings recommended in [Dorigo and Stützle, 2004] and presented on Table 4.2 were used. All experiments reported in this chapter include local search. For each experiment, *i.e.*, for each triplet (configuration, magnitude of change, frequency of change), the results presented are the average of the $T = 30$ independent runs.

In our experiments, we used the configurations shown in Table 5.2. For the Multi-caste ACS configurations, the x in the name stands for the initial letter of the migration strategy: **c** (const), **j** (jump), **sj** (SuperJump), **gj** (GreedyJump). We kept the configurations that performed better in the experiments with the TSP,

algorithm	castes	ants per caste	configuration	q_0
ACS	1	10	c90	0.90
			c95	0.95
			c99	0.99
ACS with restart	1	10	c99rs	0.99
Multi-caste ACS	2	5	x01_99	0.01, 0.99
			x05_99	0.05, 0.99
			x10_99	0.10, 0.99
			x25_99	0.25, 0.99
			x50_99	0.50, 0.99
			x75_99	0.75, 0.99
			x90_99	0.90, 0.99
			x95_99	0.95, 0.99
	4	2	x50quads08	0.50, 0.90, 0.95, 0.99
		3	x50quads12	
		2	x75quads08	0.75, 0.90, 0.95, 0.99
		3	x75quads12	

Table 5.2: Configurations used in the DTSP experiments. The x in the configuration name stands for the initial letter of the migration strategy.

namely the dual castes, being one of them $q_0 = 0.99$. We also consider several four castes configurations, but just with a total of 8 or 12 ants. For completeness we added a few more dual configurations, where one of the castes has a very low q_0. We also consider a restart variant of ACS using the recommended value of $q_0 = 0.99$ [Dorigo and Stützle, 2004]. When using the ACS with restart, every time a change occurs, the new τ_0 is computed according to the new problem instance, the trail is reset to this value, and the best-so-far-solution is voided.

5.4 Results

Previous results confirmed that the optimality of a configuration depends both on the instance being solved and the search stage. The multi-caste approach proved to be particularly robust, so we performed several experiments to ascertain if, and which, multi-caste configurations are better suited to deal with dynamic problems, and how they compare with conventional ACS.

5.4.1 Offline performance

Given the multitude of configurations and scenarios tested, we started by measuring the average offline performance for each triplet (problem instance, scenario, configuration), according to Equation 2.15. As we aim to gain insight about the comparative merits and difficulties of each configuration, to lessen the natural variation of the average offline performance due to the problem instance and scenario, we instead used, for each triplet, the offline performance normalised error, $P_{offlineError}$, computed as indicated in Equation 5.1:

$$P_{offlineError} = \frac{P_{offline} - P^*_{offline}}{P^*_{offline}} \tag{5.1}$$

In Equation 5.1, $P_{offline}$ represents the average offline performance, calculated according to Equation 2.15, and $P^*_{offline}$ refers to the best of all average offline performances for that instance and scenario. With these values we built tables, one for each configuration c, and problem instance, such as the one exemplified in Figure 5.1. Let $v_{f,m}$ be the value present in the cell at line f and column m: $v_{f,m}$ represents the offline performance normalised error of configuration c in a scenario where change occurs every f iterations and affects $m/100$ of the links. The higher the value $v_{f,m}$, the worse that configuration performed when compared to the others, in that same scenario.

Given the large number of results (52 configurations, over 7 instances, with 20 scenarios per instance), and to simplify the interpretation, the shading is proportional to the relative value of the cell (when compared to all other values for that instance). The greener the cell, the better the average off-line performance, when compared with others, the redder the cell, the worse the performance. The analysis is mostly empirical and it aims at discovering patterns in comparative performance trends regarding the standard ACS variants and the selected Multi-caste ACS configurations. We complement and support the empirical analysis with a statistical comparison of the optimisation results obtained by selected configurations.

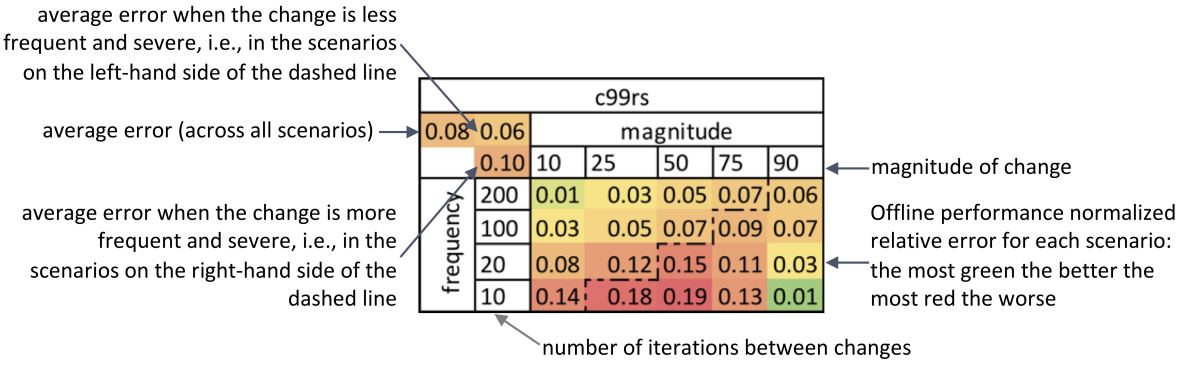

Figure 5.1: Example of how a performance table should be read

Conventional ACS

In Figure 5.2 we can see relative error of the average offline performance for both conventional ACS with several q_0 values, and ACS with restart. It is visible that low q_0 values are only competitive on smaller instances and, on those, excel in slow changing environments, or environments where the change is relatively small. c99 has comparatively better results as the instance size increases (att532, rat575), and becomes the preferred configuration for those instances in slower and less drastic changing environments. As the size of the instances continues to increase (pr1002 and pcb1173), the performance of c99 markedly decreases in scenarios where the change is more drastic, becoming a less desirable option.

As a rule, c99rs performance is extreme, being either the very best or the very worst. With few exceptions, c99rs is extremely poor in scenarios where the magnitude of change is small, regardless of the instance, and it is frequently very good when the magnitude of change is extreme. As the size of the instances increases, so tends to do the number of environments where c99rs reaches a good performance.

c01_99, c05_99, c10_99 and c25_99

Figure 5.3 depicts the results achieved by the Const dual-caste variant, one with $q_0 = 0.99$ and the other with a very low q_0 ($q_0 \leq 0.25$). The results for this group of configurations show a common pattern. Unlike what happened in the ACS

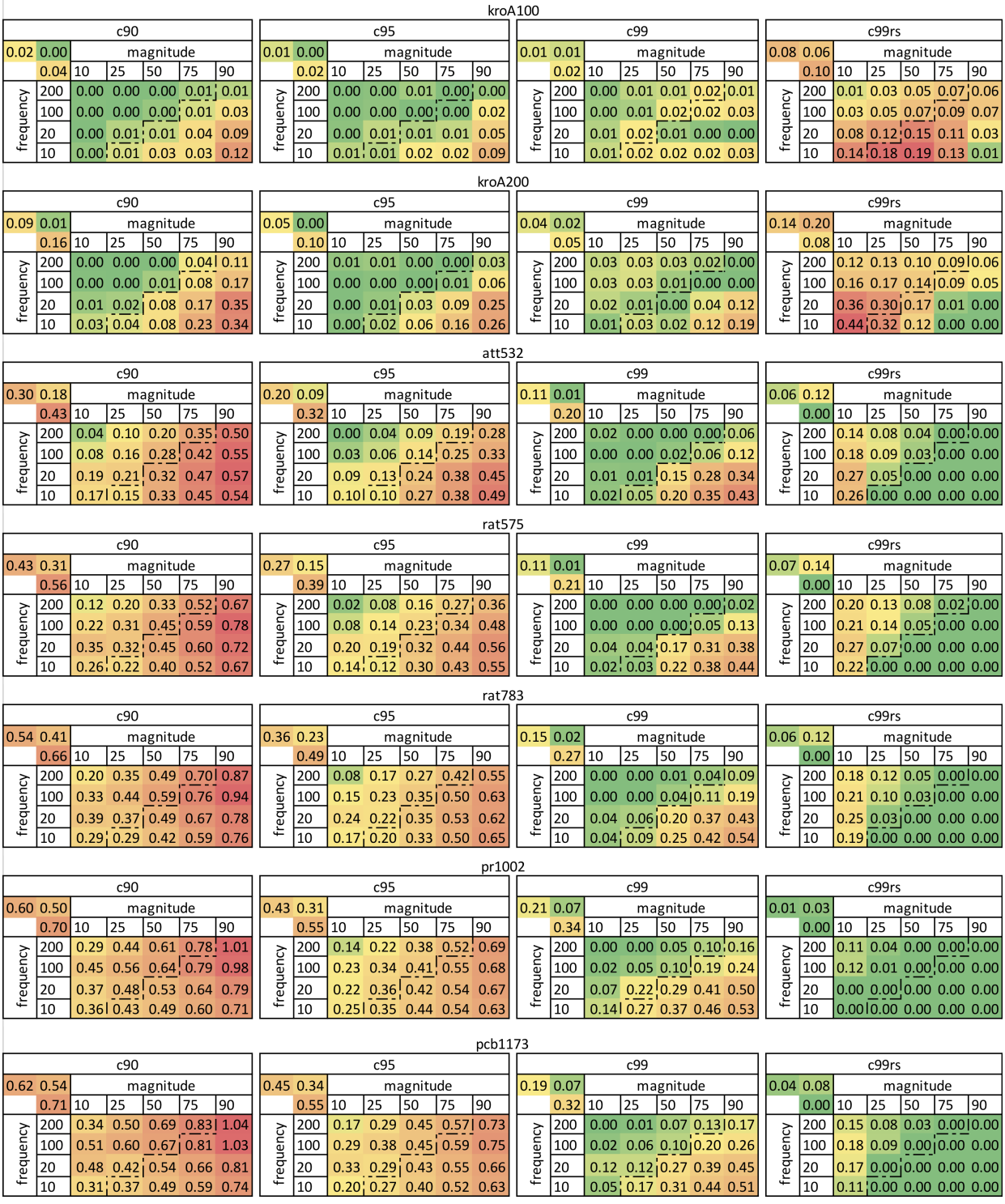

Figure 5.2: Offline performance on DTSP: ACS configurations

configurations, the variation of the comparative performance is less influenced by the change of scenario or even the problem instance. It is seldom very good or extremely bad, remaining mostly in the orange area.

In the smallest instances (first row, kroA100, kroA200) they perform better in slow changing environments where the magnitude the change is small to moderate. This is not unexpected, as those are the scenarios where the lower q_0 ACS configurations also performed better. Still, having half the ants with a $q_0 \leq 0.25$ can leave too few ants available for the proper exploitation of both the trail and heuristic information, particularly when the interval between changes is small. The scenarios were the magnitude of change is extreme are the ones where these configurations have more difficulty in achieving comparatively competitive results.

In larger instances (rat783 and larger), these configurations are competitive in fast changing environments (change every 20 iterations), provided that the magnitude of change is bellow 50. When having a short time between changes, only a small portion of the search space may be covered, so the ability to explore new parts of the trail and then focus on promising areas is more important. The advantage of having a caste with a very low q_0 when solving bigger instances can likewise be observed in other multi-caste configurations.

c50_99, c75_99, c90_99 and c95_99

The comparative results of the Const dual-caste variant, one with $q_0 = 0.99$ and the other with a moderate-to-high q_0 ($q_0 \geq 0.50$) can be consulted in Figure 5.4. It can be seen that they perform comparatively better in the smaller instances. The performance decreases as the problem size increases and as the change becomes faster and more pronounced. Together with c95, configurations c90_99 and c95_99, are very competitive for the smallest instances, kroA100 and kroA200. The performance of c50_99 and c75_99 in the smaller instances is more dependent on the magnitude and speed of change, favouring more stable scenarios, where they can achieve good results. Still the results are, overall, poor. Configuration c95_99 can maintain a good comparative performance for all instances, as long as the change is less frequent and less intense.

For the larger instances, c50_99 exhibits a behaviour similar to the Const con-

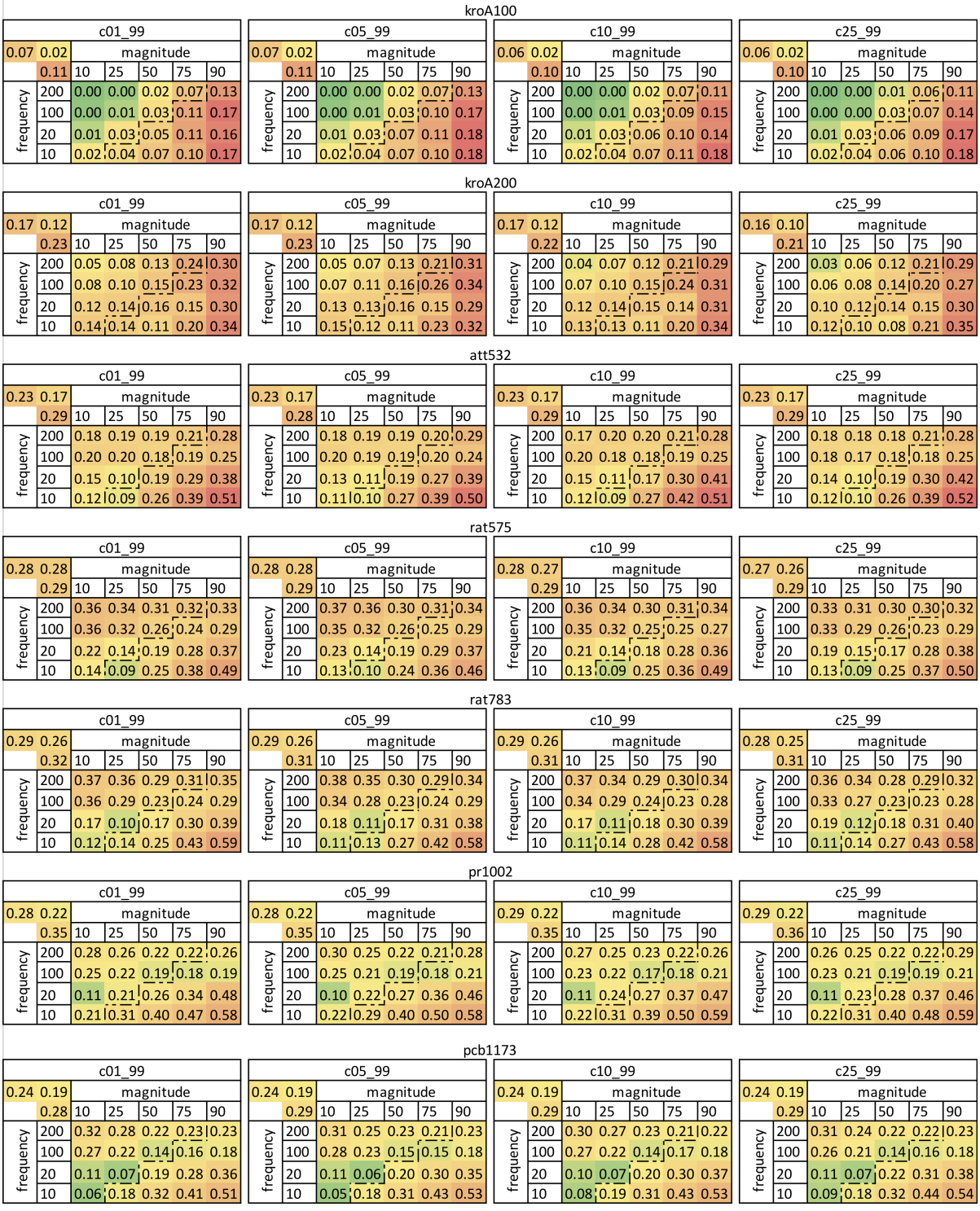

Figure 5.3: Offline performance on DTSP: Const dual-caste, one low and one high q_0

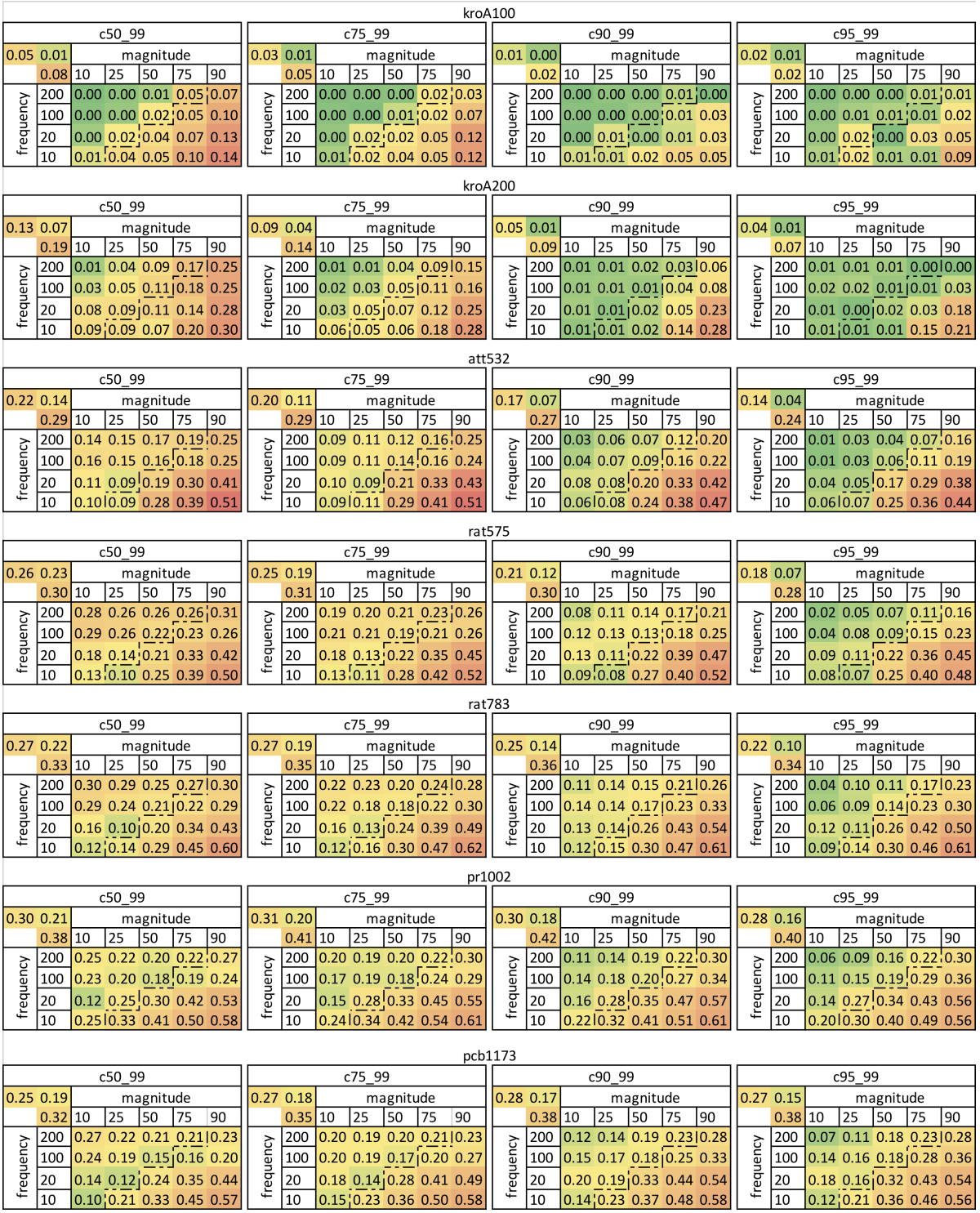

Figure 5.4: Offline performance on DTSP: Const dual-caste, q_0 moderate-to-high q_0

figurations with a smaller q_0. It is competitive in fast changing scenarios (change every 20 iterations), while the other configurations favour slower modifications. Again, having a smaller q_0 can be an advantage when escaping local optima, specially if finding a good, but sub-optimal, solution fast is not essential.

c50quads08, c75quads08, c50quads12, c75quads12

The results obtained using the Const variant with four castes are depicted in Figure 5.5. We tested 4 configurations: three of the castes had q_0 of 0.95, 0.90 and 0.99, and the fourth caste could either be 0.50 or 0.75 The total number of ants could be 8 or 12, equally divided by the 4 castes.

Regardless of the specific configuration, the variant with four castes performs better in the smaller instances, being competitive in most slow changing (100 and 200 iterations between changes) scenarios of kroA100. For kroA200, it achieves good results in the slow changing environments where the magnitude of change does not exceed 50%. Overall, these configurations prefer smaller instances and less intense change with large intervals between changes. An explanation for the difficulties in other scenarios could be the size of the caste with an high q_0. As we can confirm by observing the conventional ACS results (Figure 5.2), those ants play a very important role in the larger instances. Having only 2 or 3 ants with $q_0 = 0.99$ is probably not enough in many situations. Still, by having a set of high and low q_0 values available (e.g., the c50quads), they are usually able to exhibit lower average error than c90 and c95 in the larger instances.

As a rule, c50quads perform equal or worse than c75quads on the smaller instances (kroA100, kroA200). Conversely, it is similar or better on the larger instances, highlighting the relevance of lower q_0 when addressing these situations. Also, cquads08 (2 left columns of Figure 5.5) have consistently lower average error than cquads12, suggesting that when the q_0 values are suboptimal, updating the trail more frequently is preferable to having a larger pool of solutions to choose from.

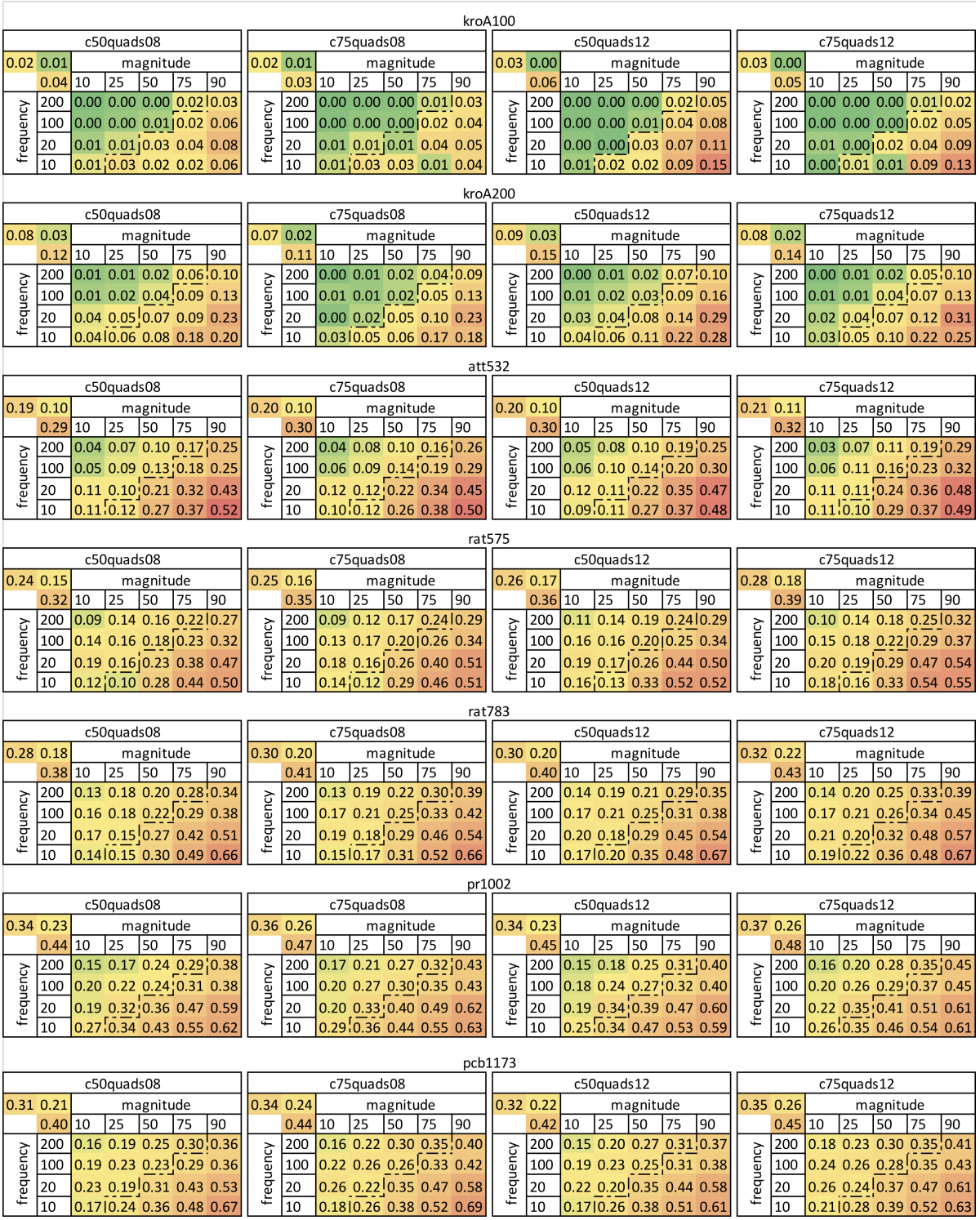

Figure 5.5: Offline performance on DTSP: Const quad-caste

j01_99, j05_99, j10_99 and j25_99

The comparative performance of the Jump strategy using 2 castes, one with a q_0 from 0.01 to 0.25, and the other with a q_0 of 0.99 can be observed in Figure 5.6. Contrary to the Const variant using the same q_0 values, the performance of the Jump configurations increases with the instance size, having the best outcomes on the largest instances. The performance improvement is more evident, as the instances grow in size and in situations with a high degree of dynamism (large and/or frequent changes). This is an expected result, as insisting on the current trail when a considerable change just occurred (the typical behaviour of ACS with a high q_0 value), is not a good strategy. The problem is even more serious, if the time available to reach a new solution is limited.

The ability of adjusting the castes size is clearly an advantage. Comparing the present results with the ones achieved by the Const variant of similar q_0 values (Figure 5.3) we can observe that the Jump variant consistently achieves better, or in the least comparable, results for almost every scenario of every problem instance. As the difference between the q_0 values is large, and hence the behaviour of the different ants is quite marked, adjusting the size of the castes allows for the greedy exploitation of the trail but, in case of change, to quickly switch to exploration mode, and then return to an exploitive behaviour as new trails are established. The j01_99 and j05_99 configurations are able to find remarkable good results in the fast changing environments of the larger instances.

j50_99, j75_99, j90_99 and j95_99

On Figure 5.7 we can see the comparative performance of the Jump dual-caste variant, one with $q_0 = 0.99$ and the other with a moderate-to-high q_0 ($q_0 \geq 0.50$). Interestingly j50_99 shares, to a certain point, the behaviour of the low q_0 dual caste Jump configurations, performing better in larger instances. The performance of j75_99 is hardly influenced by the size of the instance, while j90_99 and j95_99 performance decreases as the instance size increases. In general, the performance of the configurations j75_99, 90_99 and j95_99, is less extreme than that of the other Jump configurations, and clearly smoother than c99 and c99rs. In any case, Jump configurations with both high q_0 castes, are not as good as combining an

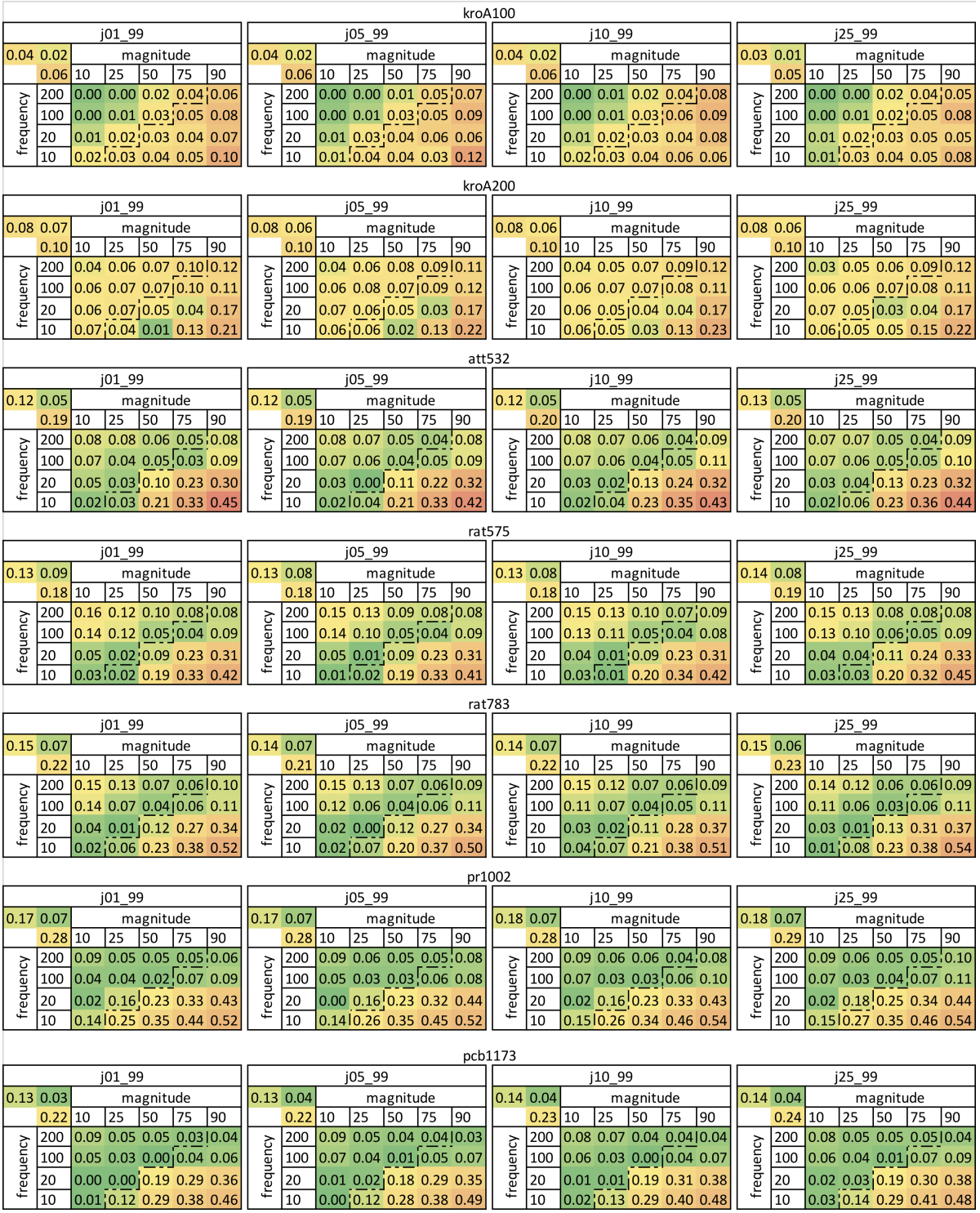

Figure 5.6: Offline performance on DTSP: Jump dual-caste, one low and one high q_0

high and a low q_0 caste, particularly on larger instances.

The Jump variant is usually better than the Const version of the same configurations (Figure 5.4), but, as expected, the difference is less noticeable in fast changing environments where both castes have similar q_0 values (j95_99 and j90_99). In this case, both castes have the high q_0 required for this scenario, but neither can provide the very low q_0 to make it ideal. The Const version occasionally surpasses the Jump strategy in the smaller instances and in slow changing environments, where the intensity of change is small to moderate. This can be explained as the Jump variant tends to favour the greedier castes and those are the exact instances and scenarios where a low q_0 (c90) is better suited. As such, enforcing half of the ants to belong to the lower q_0 caste can be beneficial.

j50quads08, j75quads08, j50quads12, j75quads12

The results obtained the Jump variant with four castes are depicted in Figure 5.8. These configurations have a good performance on the smaller instance (kroA100), and, in all but the very large or extreme change scenarios of kroA200. On the other instances the performance is worse, but it can, for the vast majority of instances and scenarios, avoid the very poor results.

Configuration j50quads12 is the most robust. As a rule, the configurations with 12 ants tend to perform as good or better than the ones with 8. This is visible in the larger instances (rat575 and larger) and when the interval between changes is large or very large. The quads12 configurations have more ants that can shift, thus adjusting the castes size to a finer degree, and these are the scenarios where greedier ants work best. The j50quads also tends to perform as good or better than the j75quads, specially on the larger instances, confirming the advantage of having a caste with a very low q_0 in these scenarios. The Jump quads are clearly superior to the Const quads, except for the 2 smaller instances, as can be discerned by comparing with Figure 5.5.

sj01_99, sj05_99, sj10_99 and sj25_99

Figure 5.9 displays the comparative performance of SuperJump configurations with two castes, one with a very low q_0 and other with $q_0 = 0.99$. One of the most robust

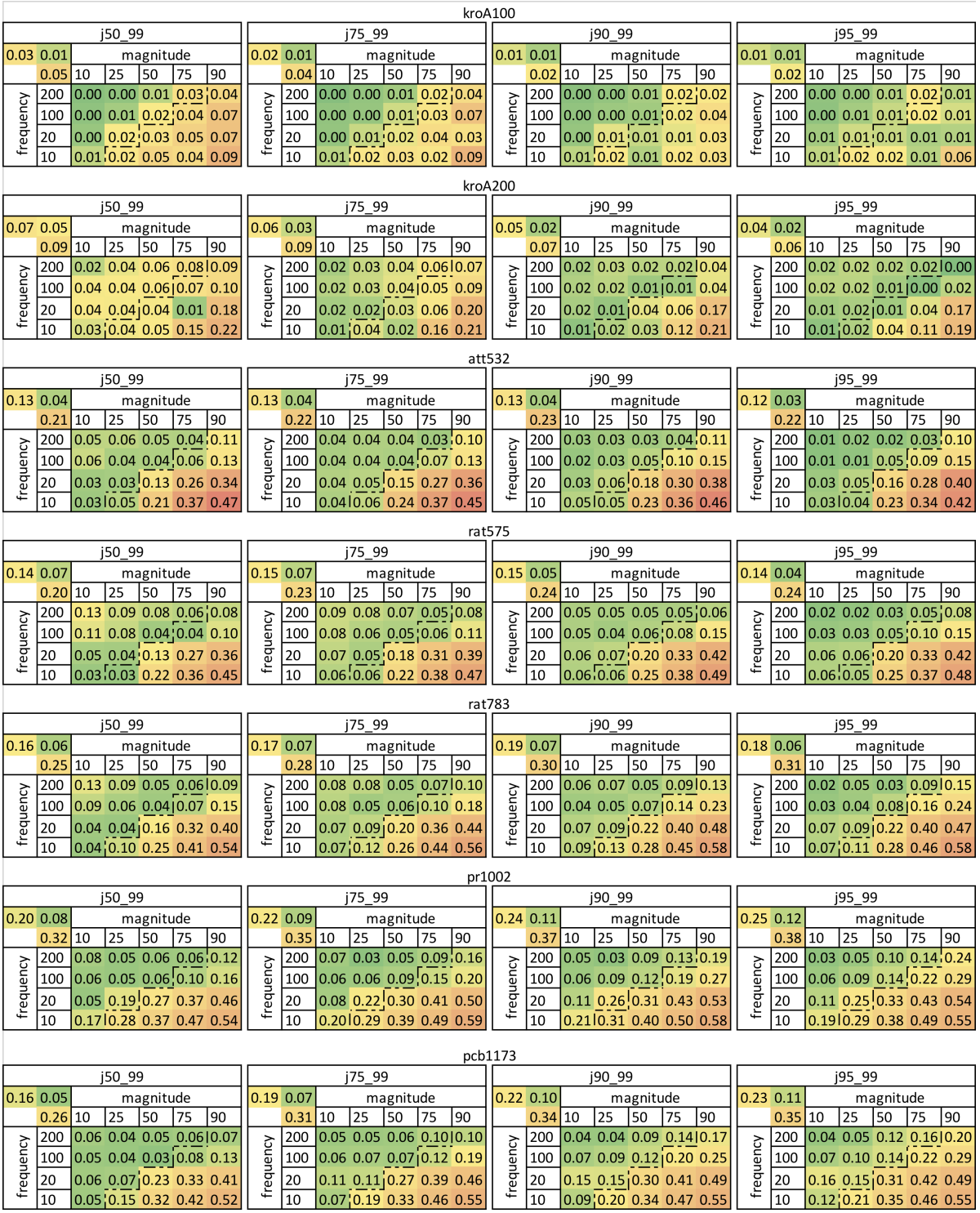

Figure 5.7: Offline performance on DTSP: Jump dual-caste, q_0 moderate-to-high q_0

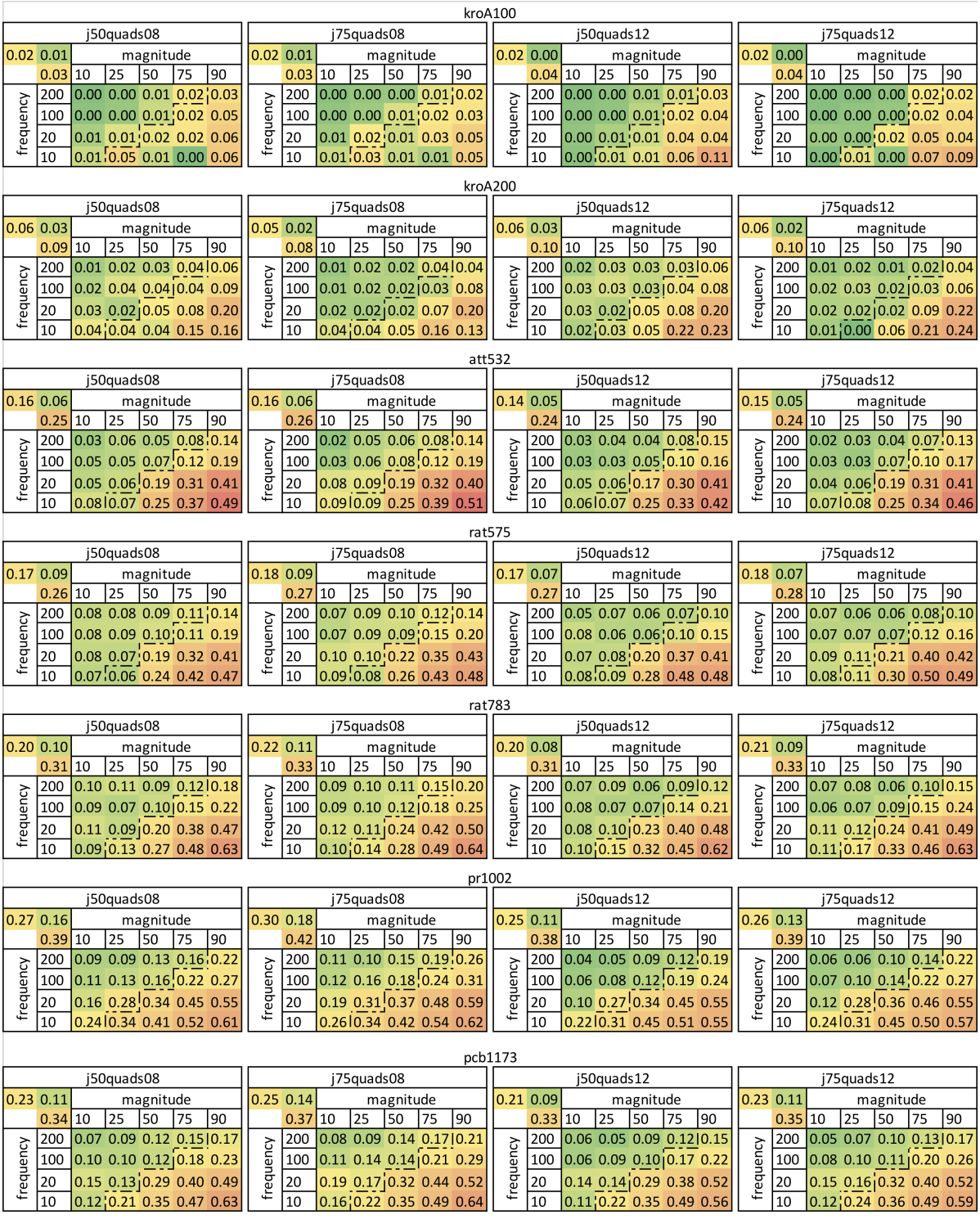

Figure 5.8: Offline performance on DTSP: Jump quad-caste

configurations, sj01_99, belong to this group. Comparing instance by instance, the results tend to be more uniform than the ones achieved by ACS, Const multi-caste or Jump multi-caste. The results are very good on the larger instances but, on the slight and slow changing scenarios of the smaller ones (kroA100 and kroA200), both c90 and c95, and the quads configurations are superior. For the smaller instances, these configurations are more competitive in environments where the change is faster or more dramatic, but, in larger instances, they achieve good results in almost every scenario.

The performance of the four configurations is similar, but the small and slow changing environments are more favourable for higher q_0, while very drastic and rapid changing environments are better suited for the ones with a very low q_0. Considering all the 140 scenarios, and regardless of the very best configuration being c99 or c99rs, SuperJump strategies always achieve lower average offline errors than the worse of the two, except for a few of the most extreme scenarios where the error is comparable. In the most extreme scenarios (change every 10 or 20 iterations with a magnitude of 90) of instances kroA100 and kroA200, the average error of this group is larger than both c99 and c99rs. Configuration sj01_99 is remarkably robust, possibly offering a safer option than c99 or c99rs. Most of the scenarios are fitted to one of the two standard ACS configurations, but adverse to the other, and sj01_99 usually has a smaller average error than the worse of the two configurations.

As a rule, SuperJump outperforms the Jump variant in the slower changing scenarios. This is expected, since a higher q_0 is desirable. When compared with the Jump variant, the SuperJump castes of higher q_0 tend to be larger, as this migration strategy allows a minimum caste size of 1 (and thus, when we only have 2 castes, and 10 ants total, the maximum caste size allowed is 9). In the larger instances, in fast and drastic changing environments, the Jump variant is better than the SuperJump, likely due to the pronounced benefit of having very low q_0 ants in these scenarios and the fact that the Jump variant tends to keep larger low q_0 castes. This is mostly evident when comparing sj01_99 with j01_99. It is worth noting that, even if Jump variant performs better in these scenarios, both strategies are very competitive.

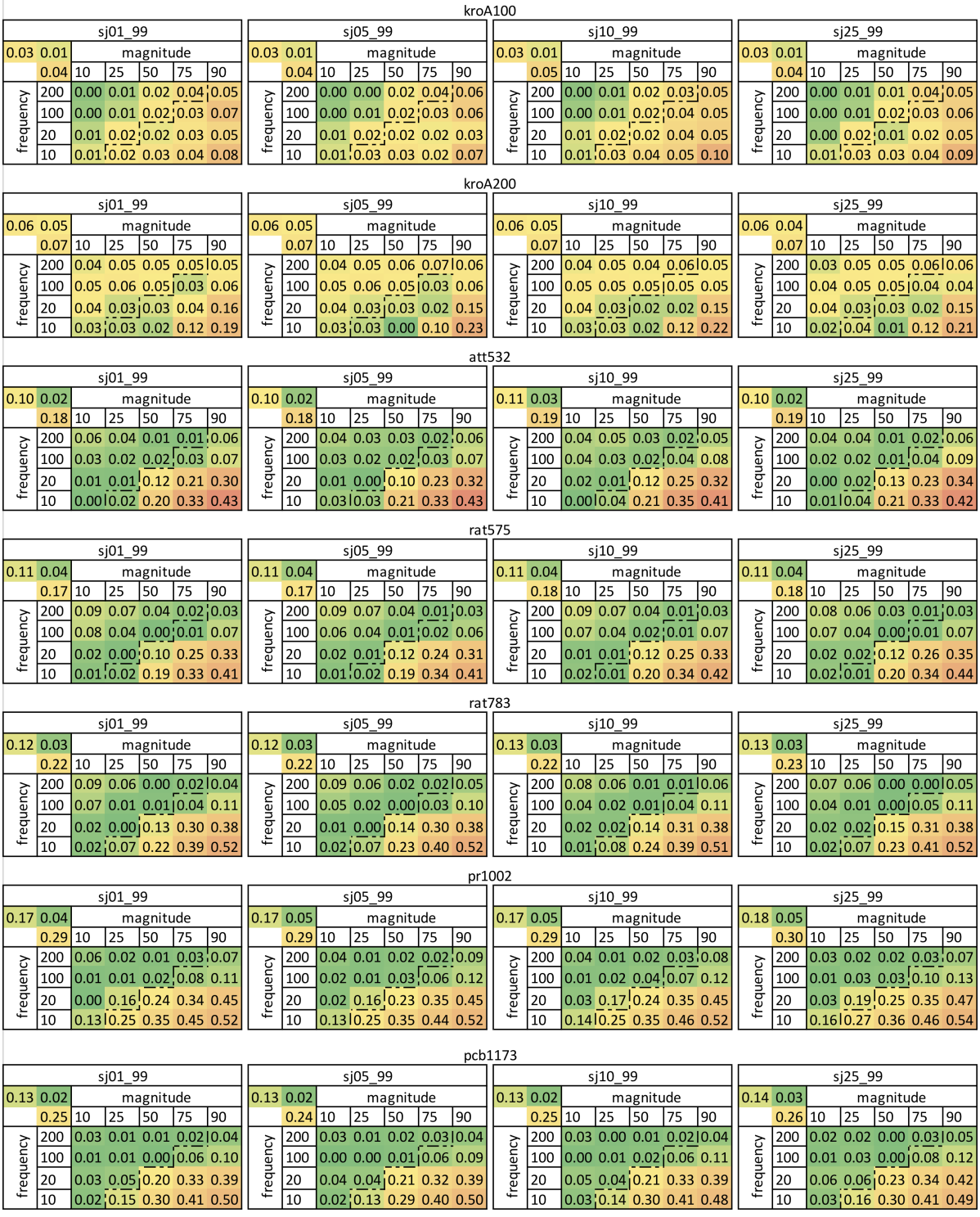

Figure 5.9: Offline performance on DTSP: SuperJump dual-caste, one low and one high q_0

sj50_99, sj75_99, sj90_99 and sj95_99

The trend observed in the last section continues when considering the comparative performance of the dual-caste SuperJump variant of moderate to high q_0, as can be observed in Figure 5.10. These SuperJump configurations are not as successful as the ones with a smaller q_0, yet are very good in avoiding very poor results. Within this range of configurations, the ones with lower q_0 perform better as the size of the instances increases, while the higher q_0 are better suited for smaller instances. As expected, larger q_0 configurations also have more difficulty when the change is frequent or large. When comparing with c99 and c99rs, the number of scenarios where this group of SuperJump configurations is surpassed by one or both ACS variants, increases with the q_0 of the configuration and the dimension of the instance.

As expected the difference between Jump and SuperJump is less noticeable, as the q_0 values of the castes are more similar. The pressure to increase the size of the greedier caste is felt more keenly when the difference in q_0 is large. In situations where the lower q_0 is higher, Jump and SuperJump configurations have more similar castes sizes (see Figure 5.26) and, as a consequence, so is the quality of the results achieved.

sj50quads08, sj75quads08, sj50quads12, sj75quads12

The results obtained by the SuperJump variant with four castes are depicted in Figure 5.11. SuperJump quads, particularly sj50quads12, are the best four caste configurations. Still, the comparative results are modest. As a rule, sjquads12 are similar or better than sjquads08, namely in the slower changing environments of instances rat575 and larger. Yet, on a few more rapid or extreme change scenarios of smaller instances, (kroA100, kroA200, rat575 and rat783), sjquads8 present a smaller average error. As expected, sj50quads are slightly better than sj75quads on the larger instances. On the smaller instances the relation is not as clear, since sj50quads12 is almost identical to sj75quads12, but sj50quads08 is a little worse than sj75quads08.

SuperJump quads are similar or have a slightly smaller average error than Jump quads, for all instances and almost every environment. This is expected,

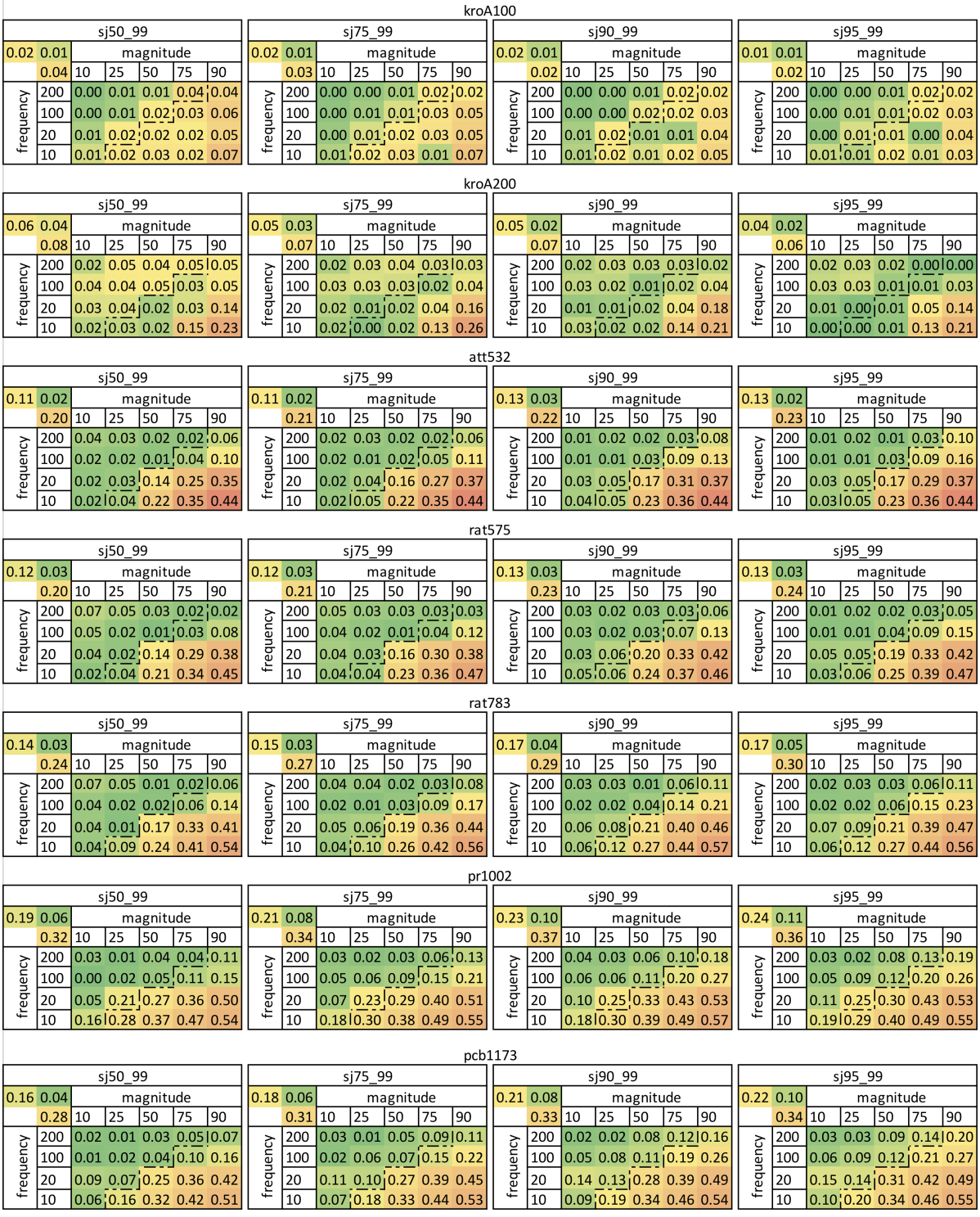

Figure 5.10: Offline performance on DTSP: SuperJump dual-caste, q_0 moderate-to-high q_0

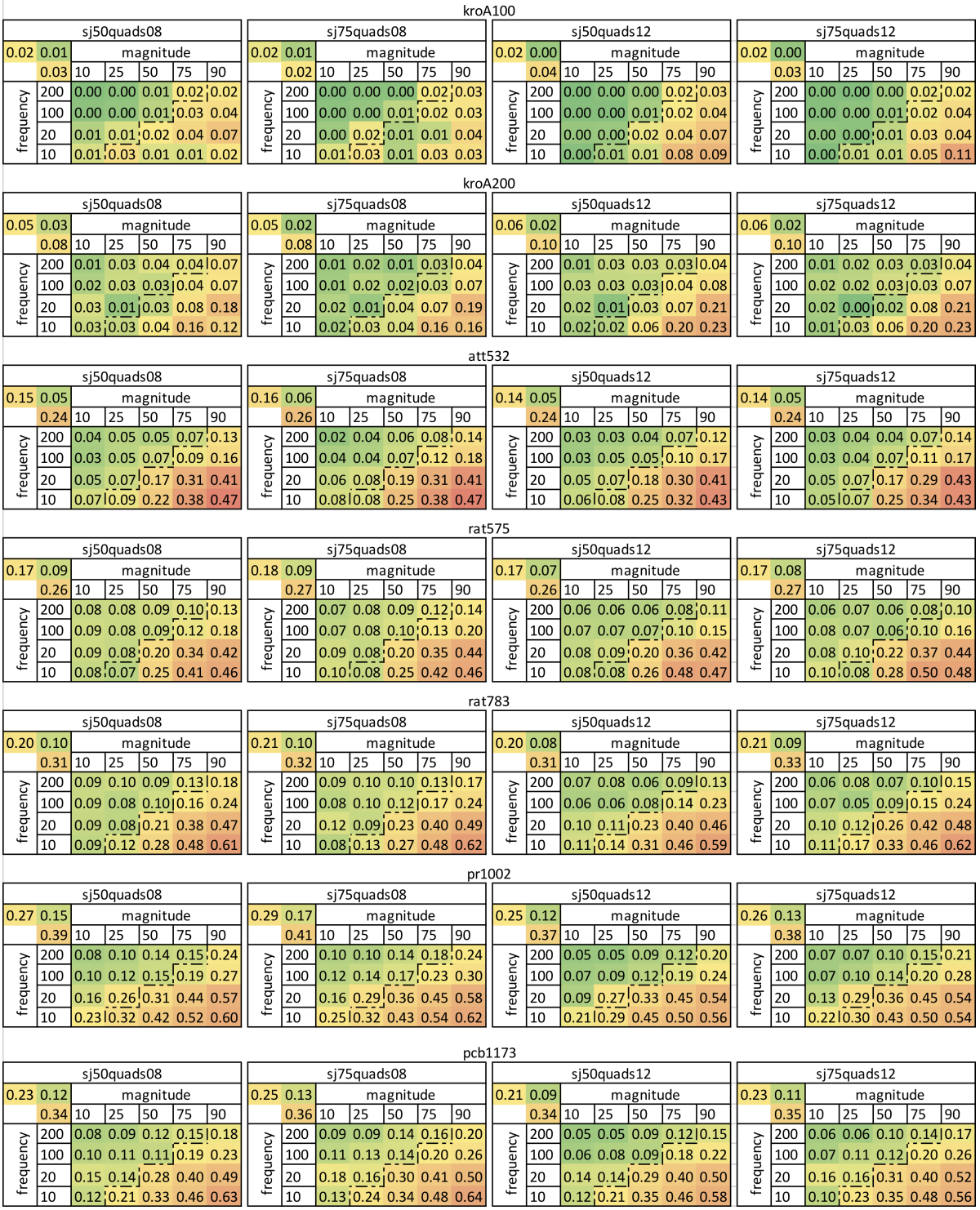

Figure 5.11: Offline performance on DTSP: SuperJump quad-caste

as the total number of ants that can be moved from one caste to another is reduced when compared to the dual caste configurations. Still we can see a small improvement by using the SuperJump variant, particularly in colonies with just 8 ants, likely because SuperJump encourages movement the most, making the adjustments faster.

gj01_99, gj05_99, gj10_99 and gj25_99

The comparative results of the configurations with one low and one high q_0 caste using the GreedyJump migration strategy are depicted in the Figure 5.12. These configurations are quite robust, exhibiting good results in the larger instances, and avoiding very poor results in the smaller ones. They achieve good comparative results in almost all instances and scenarios, except for the smaller instances, in the scenarios of small magnitude of change and large interval between changes. These are scenarios which are better suited for c90, c95 and the quads configurations.

The results achieved by GreedyJump are similar to the ones obtained by SuperJump with similar castes: a little superior in a few scenarios of the smaller instances, a little inferior, in a couple scenarios of the larger instances. The number of scenarios were GreedyJump improves when compared to SuperJump increases with the q_0 value. Also, GreedyJump prevails over the Jump strategy, particularly in the smaller instances, and in the slow changing environments of the larger instances. In the more drastic changing and moderate-to-fast scenarios of the larger instances, GreedyJump tends to be not as good as SuperJump. This is likely because GreedyJump is overly greedy, and, given the degree of change and the restricted time available, the information becomes misleading.

A direct comparison with standard ACS variants c99 and c99rs reveals that, in the smallest instance (kroA100), the results are, on most scenarios, similar to the ones achieved by c99 - the best of the two configurations for this instance. They are also better or similar to the ones achieved by c99rs, except for the most severe change scenario (for comparison refer to Figure 5.2). For every scenario of all other instances, the average error of these GreedyJump configurations is comparable or inferior to both c99 and c99rs. As such, gj01_99, gj05_99 and gj10_99 are likely a more robust option than c99 or c99rs, since neither ACS variant is universally

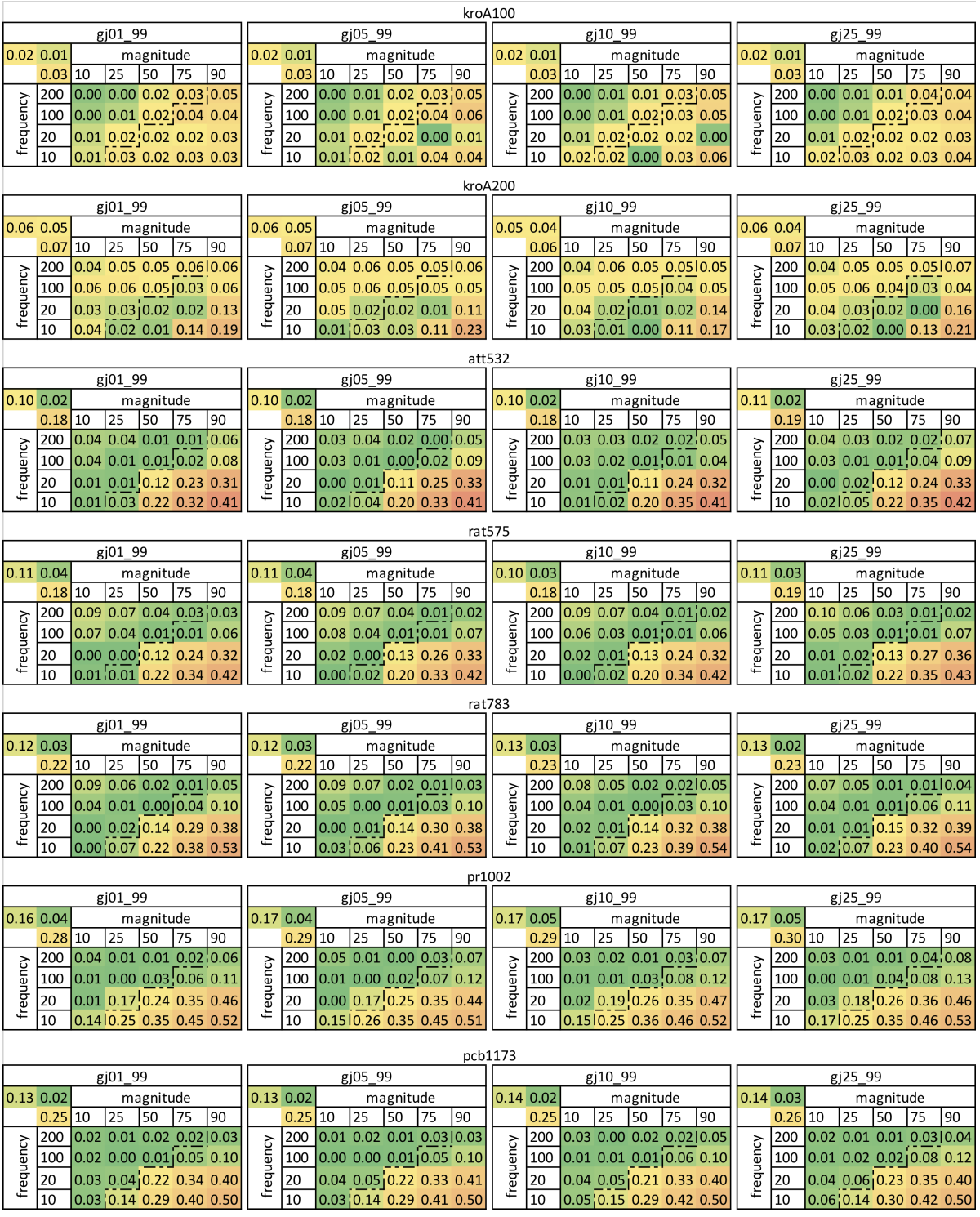

Figure 5.12: Offline performance on DTSP: GreedyJump dual-caste, one low and one high q_0

good, and the GreedyJump very rarely has a larger average error than both, being usually better than the worst of the two.

gj50_99, gj75_99, gj90_99 and gj95_99

The comparative performance of GreedyJump dual configurations with higher q_0 can be observed in Figure 5.13. Like lower q_0 GreedyJump configurations, we can see that the performance of this group, and particularly of gj50_99, is less extreme than standard c99 and c99rs. Configurations gj50_99 and gj75_99 favour average to larger instances, while gj90_99 and gj95_99 are particularly well suited for smaller instances. Configuration gj50_99 is well-balanced, being a little superior to GreedyJump configurations with lower q_0 in the smaller instances, but somewhat less successful in the fastest scenarios of the larger instances.

The advantage of using GreedyJump over SuperJump is clearer as q_0 increases. In fact, while gj50_99 is similar to sj50_99 on most scenarios, better in a few, but also worse in a couple of other ones, gj95_99 is similar to sj95_99 in most scenarios, but has a lower average error in almost 20% of the environments. When the q_0 values are high, GreedyJump castes sizes are noticeably different from those of SuperJump. As a rule, for a given instance and scenario, the size of the lower q_0 caste of gj95_99 is larger than the lower q_0 caste size of gj01_99, but considerably smaller than the lower q_0 caste size of sj95_99 (consult Figure 5.26 for an example of the average caste sizes). In a sense, gj95_99 behaves more like c99 and, as such, better than sj95_99 in those situations where c99 is also better than sj95_99 (e.g., on the larger instances).

On most scenarios, GreedyJump configurations are comparable or superior to the Jump ones when considering configurations with similar q_0 values. The advantage of GreedyJump is most visible on the slower changing environments. This is explained by the general rule that these environments benefit from higher q_0 castes, and those castes tend to be larger on GreedyJump configurations. The exception of the GreegyJump superiority over Jump configurations occurs in instance pcb1173. Here j50_99 is better suited for fast or drastic changing scenarios, where very low q_0 ants are specially beneficial, and the need for very high q_0 is smaller.

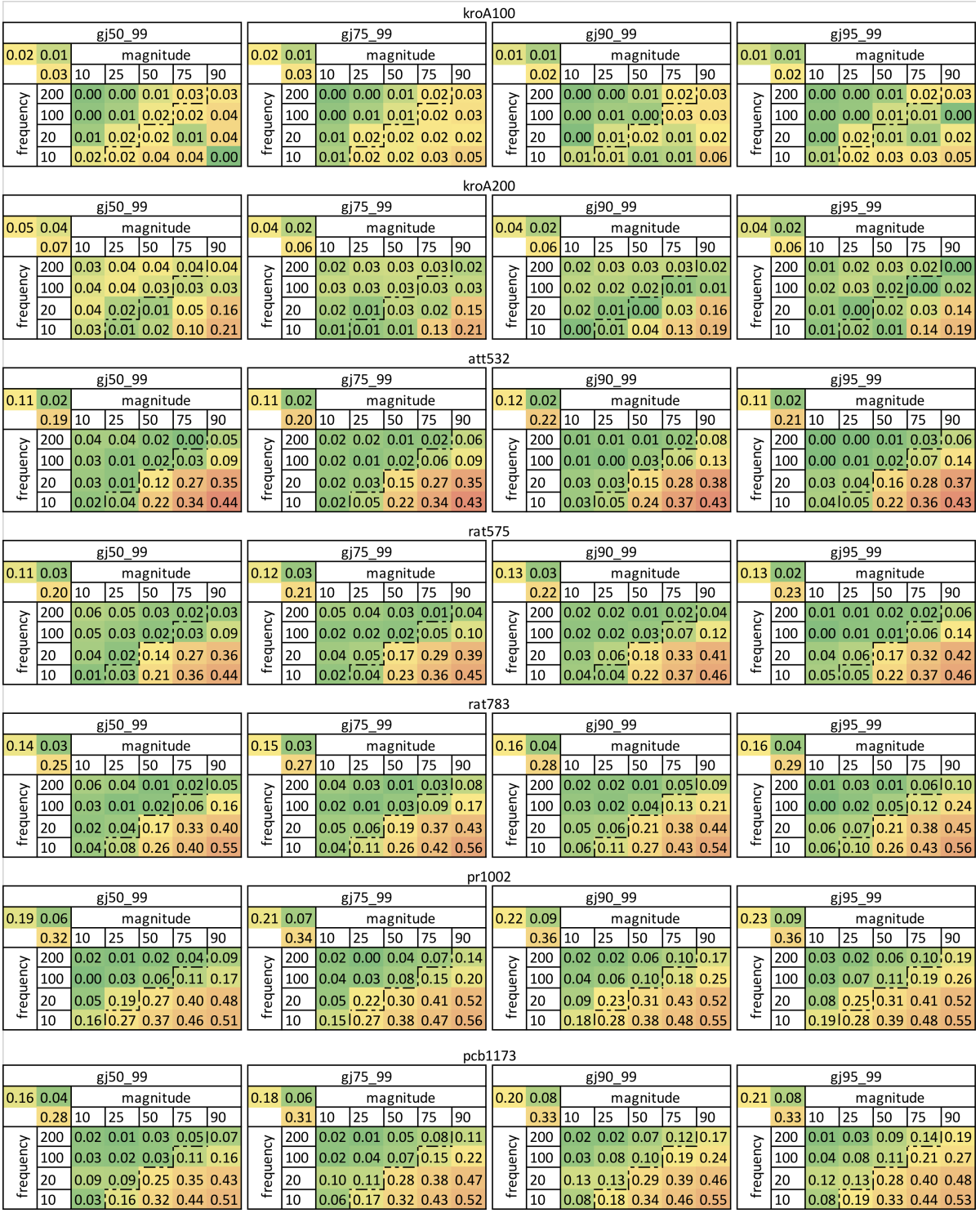

Figure 5.13: Offline performance on DTSP: GreedyJump dual-caste, q_0 moderate-to-high q_0

The performance of these GreedyJump configurations is superior than one or both of c99 and c99rs in the smaller instance kroA100, but then, the comparative performance declines with the instance size, and the q_0 value of the castes present. As a rule, configuration gj50_99 has a balanced outcome, surpassing c99rs in the scenarios where c99 is the best option. Generally, it also obtains results similar or better than c99 in those scenarios where c99rs is better suited.

gj50quads08, gj75quads08, gj50quads12, gj75quads12

The comparative results of GreedyJump with four castes are illustrated in Figure 5.14. This variant, particularly the configurations with 8 ants, is well suited for the smaller instance, kroA100, and for the slower, less drastic changing environments of the other instances. In more extreme or fast scenarios, the results are modest. Within this group of configurations, the performance of gj50quads is comparable to gj75quads, with a little advantage to gj75quads in the smaller instances. gjquads08 are, in average, similar or superior to gjquads12, except on the slower scenarios of the smaller instance, were the configurations with 12 ants perform better.

GreedyJump and Jump configurations with 8 ants are mostly similar, though gj75quads08 tends to perform better than j75quads08. The performance of gjquads12, when compared to jquads12, is similar or inferior, particularly on the slower scenarios of the larger instances. It is expected that the difference between GreedyJump and Jump is more noticeable when we have more ants in total: more ants can be moved, thus having a larger impact in the outcome. The scenarios where jquads12 are better than gjquads12 are those that need a low q_0 to face the change, but afterwards benefit from a higher q_0 to take advantage of the time available to exploit the heuristic informations and the updated trail. GreedyJump with 12 ants may have difficulty in adjusting the size of the castes once one of them becomes dominant, since it chooses the best ant of each caste.

5.4.2 Comparative Performance of Selected Configurations

In this section we present a statistic comparison of the results obtained by three selected configurations, c99, c99rs and gj01_99, to confirm if indeed gj01_99 is able to provide a more robust and balanced performance than that achieved by

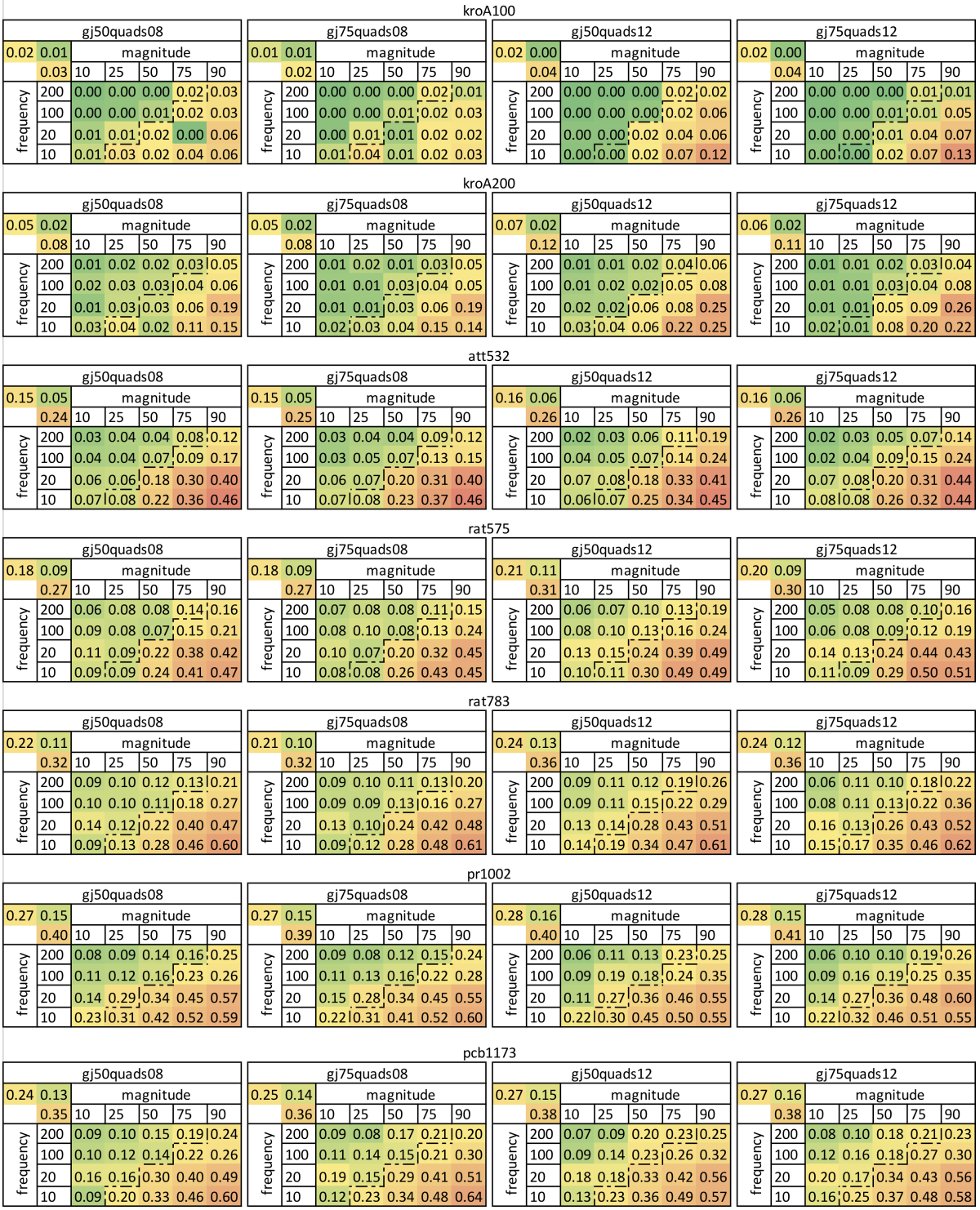

Figure 5.14: Offline performance on DTSP: GreedyJump quad-caste

the standard ACS configurations. Both c99 and c99rs excel on several scenarios, but, conversely, they have quite poor performance on some others. Our hypothesis is that gj01_99 is able to avoid poor results, regardless of the scenario.

Offline performance

The previous comparative analysis was based only on the average performance of each configuration, but did not consider the actual distribution of the values generated. It provided us with a trend of the comparative performance, but it lacked statistical validity. We now consider the two most successful ACS configurations, c99 and c99r, as for nearly every instance and scenario tested (the exceptions are some of the slow and slight changing environments of the smaller instances kroA100 and kroA200), they were the best ACS configurations. gj01_99 was selected as the previous analysis suggested that the coexistence of a very low q_0 caste combined with a greedy update strategy is a balanced combination, able to both exploit the information in more stable scenarios, but also to foster exploration of new trails when changes occur.

To compare the samples we used a repeated-measures ANOVA, as the samples are not independent. For each magnitude of change, m, we built 30 sequences of TSP instances (the sequential elements differed according to the value of m) as explained in section 5.2.2. We denote this sequence as A, B, C, and so on. Each configuration was given sequence A for the first run (try) of the algorithm, sequence B for the second one, and so on. When we perform the analysis of variance, the independent variable is the configuration (c99, c99rs or gj01_99) and the dependent variable is the offline performance, as defined in equation 2.15. This is a repeated-measures design because each sequence will be used by every configuration. Thus, the total variation in the performance will, in part be caused by the fact that some configurations perform better, and will, in part be caused by the fact that the sequence itself leads to tour lengths of a certain dimension.

One-way repeated-measures ANOVA requires that the assumption of sphericity must hold. To assess this assumption we use the Mauchly test with a significance level of 0.05. If the test is significant, the condition of sphericity is not met. In those cases where the sphericity assumption was not met, we used the

Greenhouse-Geisser correction and Huynh-Feldt correction p-values to access if any of the samples presented significant different means in offline performance, as recommended by [Field et al., 2012]. When there was a significant difference in the means of the samples, we conducted a post-hoc using the Tukey test when the sphericity assumption was clearly met, and the pair-wise t-test using the Bonferroni adjustment, as suggested by [Field et al., 2012], when that was not the case. The tests and post-hoc were done at a significance level of 0.05 using the 'ez', 'multcomp' and 'nlme' packages of the [R, 2011] software.

In Figure 5.15 we present the results of the comparisons for all instances considered in the study. The cells are shaded according to the number of configurations that had a significantly different mean offline performance with a lower (better) average performance. Hence, pink shading means that the other two configurations had a better performance than the current one, yellow means that only one of the other configurations was better, and green means that no other configuration had a significantly different mean with a lower average offline performance.

On instance kroA100 we can see that the restart strategy is a bad option for almost all scenarios, except those where the magnitude of change is extreme. On most scenarios the results of c99 and gj01_99 overlap and both configurations perform equally well. For instance kroA200, c99 is once again the best configuration, particularly in the slow changing environments and in the faster changing environments, as long as the change is not too acute. On the other side, c99rs is the best option when the change is very fast and intense, but, when the change is less marked, it performs bellow the other two configurations. gj01_99 has a balanced performance, excelling in the fast environments, as long as the change is not too extreme.

The results for instance att532 reveal an almost a perfect division between the scenarios where c99 is the best performing configuration, and those where c99rs is preferable. c99 has a very good performance in the scenarios where the intensity of change is smaller or the change is slower. It performs worse than the other two configuration when the change is more intense and frequent. When compared with the other configurations, c99rs is very good when the change is more intense (50 or higher), but quite bad in the other scenarios. Still, the favourable scenarios seen in kroA200 and kroA100 are kept, and the total number of scenarios were c99rs

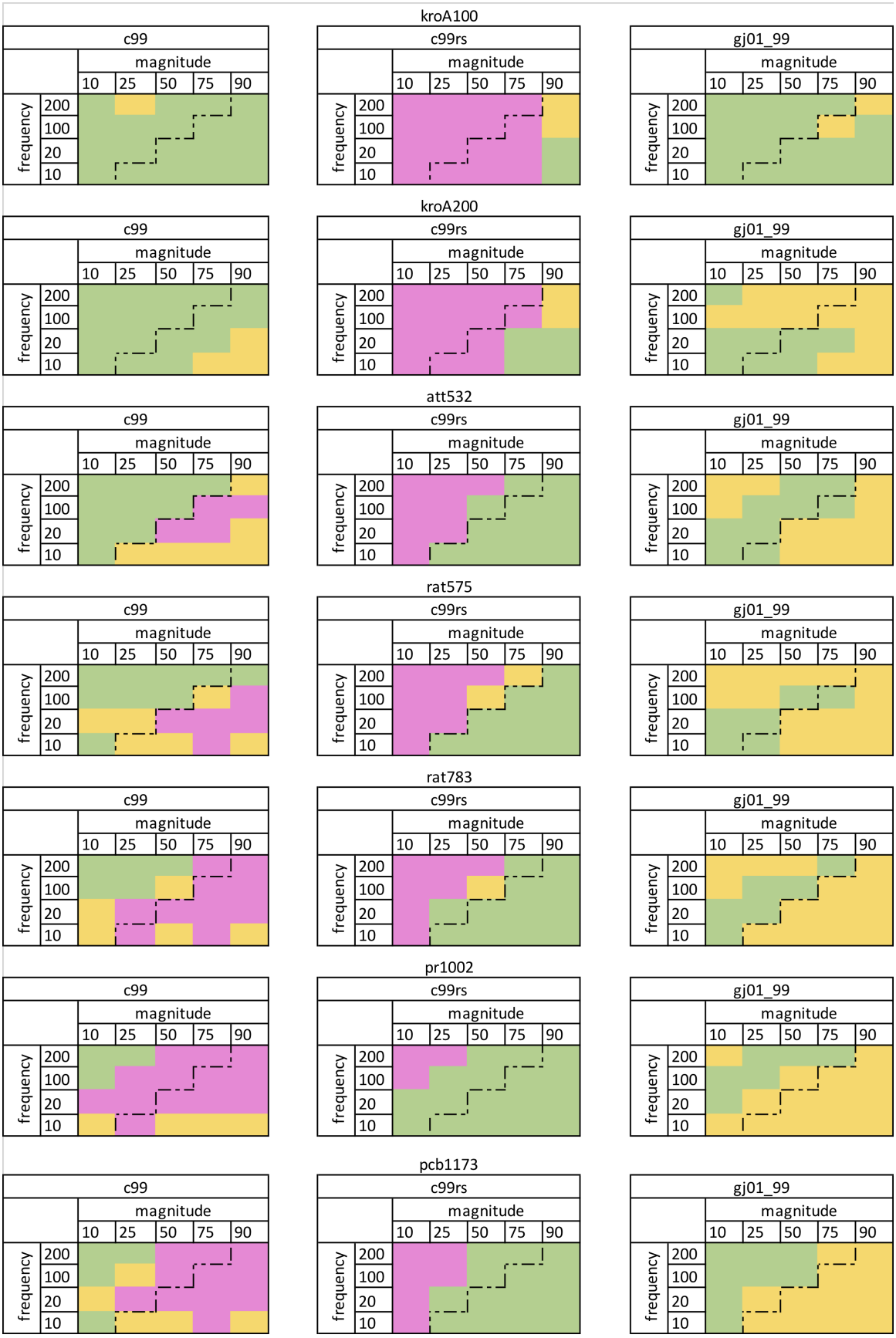

Figure 5.15: Performance comparison on DTSP: Selected Configurations

is the better option increases. gj01_99 outperforms c99 in those scenarios where forgetting previous information is important, while retaining the ability to exploit the information where c99rs fails. Summing up, it keeps a balanced performance.

The results for instance rat575 are alike those achieved for att532. The comparative performance of c99 degrades, only remaining the best option for slow scenarios or scenarios with less acute change. The range with fast and more intense changes presents a problem for this configuration, as a considerable part of the information is not reusable and is instead likely misleading. A general rule can be discerned by glancing at the tables: the number of scenarios were c99rs is the best option grows with the size of the instance, and the preferred scenarios are those were the change is larger and more frequent. Conversely, the scenarios better handled by c99 decrease with the size of the instance, and c99 keeps to the slower and less drastic changing corner of the table. Interestingly, in instance rat575 we can see some exceptions to this rule, as some of the scenarios where c99rs was considered an equally good option in att532 (e.g., m50f100 and m75f20) are now clearly best solved by c99. The number of scenarios where gj01_99 is the best option keeps diminishing, but the configuration is, nevertheless, able to avoid poor results.

The results on instance rat783 reveal that the comparative performance of c99 degrades further and is now the best option only for scenarios with slow and less marked changes. The number of scenarios where c99rs is preferable increase and, just like in the smaller instances, only if the magnitude of change is small (10%), it is clearly best to keep the trail. If the magnitude of change is moderate or high (25% to 50%), keeping the information is only advantageous if there is enough time between changes (100 or 200 iterations). gj01_99 performance is very similar to the one presented for instance rat575. The two largest instances pr1002 and pcb1173 follow this trend. Configuration c99 is only competitive in the most constant scenarios, c99rs excels in the more dynamic ones and gj01_99 keeps a balanced performance.

As a general conclusion, we notice an almost a perfect division between the scenarios where c99 is the best performing configuration from those where c99rs is. Both configurations can be very good in some of the environments, but tend to have extreme results: very good in some cases and quite poor in some others.

There are very few scenarios where the results of the two configurations overlap. Considering the 140 scenarios, c99 is ideal for 50, but the worst than the other two for 43; c99rs is ideal for 67 scenarios, but worst than both others on 62, gj01_99 is ideal for 61 scenarios and it is never worse than both the c99 and c99rs.

The desirability of a given configuration depends, not only on the size of the instance and the combination of frequency and magnitude of change, but also on the instance itself. Likely the structure of the instance is also a significant factor in determining if an instance is easy or hard to solve, as it is for the static case. Although some patterns can be observed, a perfect rule is impossible to devise, as there are also irregularities in those patterns.

Still c99 tends to perform comparatively better in smaller instances or when the frequency and magnitude of change are smaller. Conversely, c99rs prefers the opposite scenarios. Clearly, the more drastic the change, the smaller is the amount of useful information that remains. Also, if the frequency of change is high, or if the instance is large, there is less time to recover from suboptimal starts and the outdated trails become more problematic.

The results achieved by c99 and c99rs are distinct and they are particularly effective in complementary environments. Choosing one of them to apply to an unknown scenario might have a strong impact on the results, particularly if we select the configuration that is not well-suited to handle the current situation. This dillema is avoided by gj01_99. By being able to use or forget the information as needed, gj01_99 is a robust approach that presents a more balanced performance. In all tested scenarios, it achieved either the best or second best performance among the three configurations.

Average peak error

Another relevant aspect in dynamic problems is the stability of the algorithm in the presence of change. To measure it we recorded the peak after each change and computed the average peak with Equation 2.16. To lessen the natural variation of the average peak we consider the average peak normalised error, Q_{error}, as calculated by Equation 5.2:

$$Q_{error} = \frac{Q - Q^*}{Q^*} \tag{5.2}$$

Where Q represents the mean average peak for that configuration (Equation 2.16), and Q^* refers to the best (smallest) of all mean average peaks for that instance and scenario, considering configurations c99, c99rs and gj01_99. Results from the three configurations are presented in Figure 5.16. The shading is proportional to the value of the cell, when compared with all other values for that instance. The greener the cell, the smaller the average peak, the redder the cell, the higher the peak. In bold, we identify the lowest average peak of the three configurations presented. The peaks provide a good insight about the relevance of reusing the trail once the change occurred, as we have three distinct strategies to compare:

- c99 - It exploits both the old trail and the new heuristic information
- c99rs - It disregards the old trail and exploits the new heuristic information
- gj01_99 - It either exploits both the old trail and the new heuristic information (ants form the $q_0 = 0.99$ caste) or is loosely guided by both the old trail and the new heuristic information (ants from the $q_0 = 0.01$ caste)

In the case of the smaller instance kroA100, previous information provides an advantage when the magnitude of change is small (10%) to large (75%), but is detrimental if it is extreme. Configurations c99 and gj01_99 have similar results, but in the more extreme change scenarios, the performance of gj01_99 degrades. This is likely because the information provided by the trail tends to be less relevant and, since gj01_99 has fewer greedy ants, the probability of finding useful information is smaller.

Results for instance kroA200 clearly reveal that that reusing the trail is only beneficial when the amplitude of change is small (10%) to average (50%). If the magnitude of change is larger restarting the trail is preferable.

The average peaks on instance att532 are distinctive. In almost every scenario, the restart strategy is not a good option. This reveals that the heuristic information is not as accurate as in other instances, and relying solely on it is not ideal.

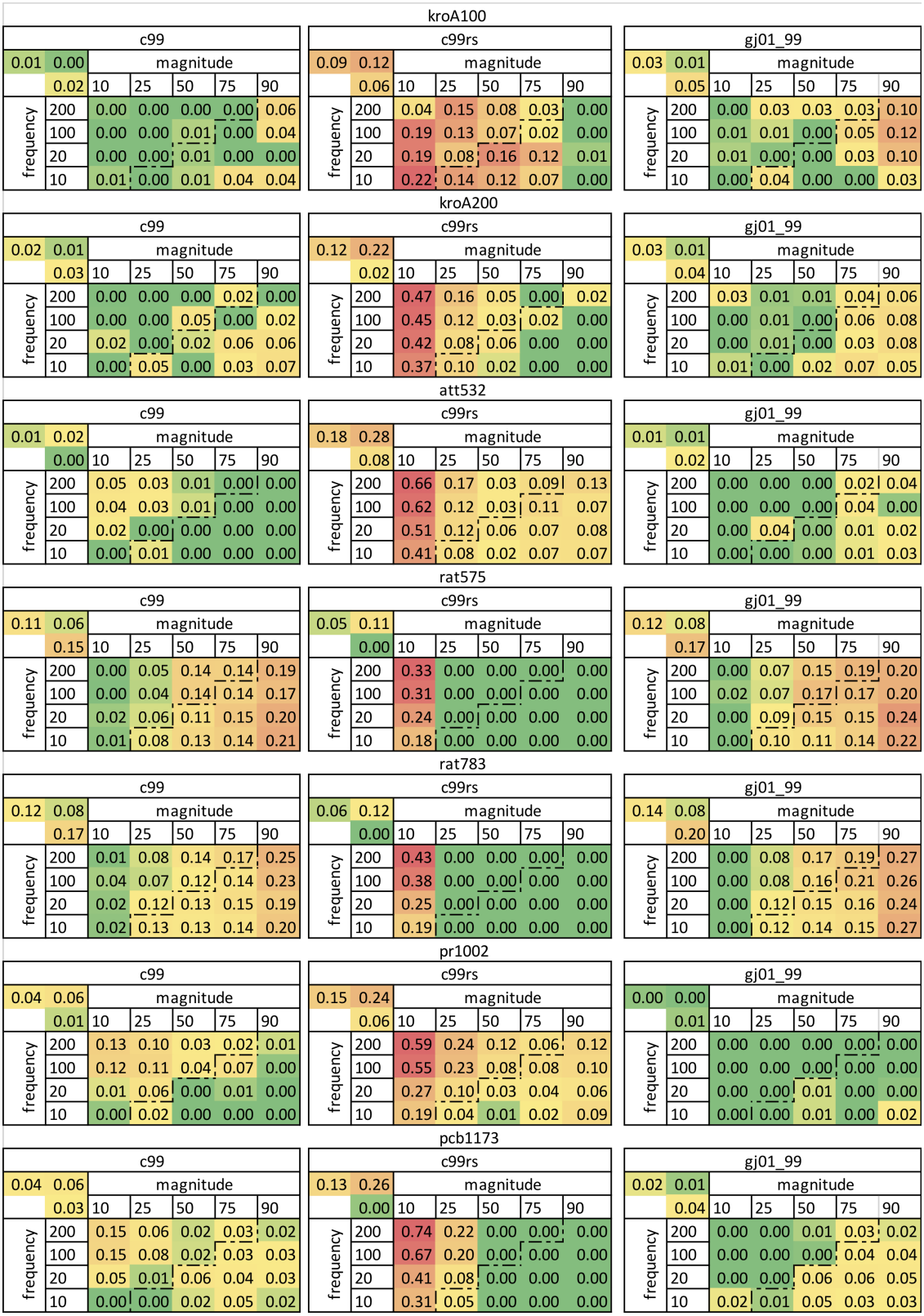

Figure 5.16: Average peak error comparison on DTSP: Selected Configurations.

Interestingly the scenarios of att532 where c99 has smaller peaks are exactly those where its offline performance is worse, suggesting that the short peaks in fact lead to local optima that the greedy strategy could not escape.

The shape of the average peaks in instances rat575 and rat783 are similar. The differences between c99 and c99rs suggest that the modifications are so strong that they render the previous knowledge misleading. A few exceptions are visible when the intensity of change is low and the interval between changes generous. Once again we find scenarios where c99 (or c99rs) has smaller peaks than gj01_99, but where gj01_99 offline performance ends being better than that of c99 (or c99rs), again suggesting that the multi-caste strategy allowed to escape local optima.

On instance pr1002 the restart strategy is almost never a good option, being clearly the worst for most scenarios. As a rule, the best option is to occasionally forget all the information (gj01_99). This suggests that changes might have a decreased impact on the instance and that the trail likely leads to premature stagnation. When the magnitude of change is small to moderate (10% to 25%) and the interval between changes is large (100 or 200 iterations), c99 performs worse than gj01_99, possibly because c99 produces more marked trails and explores them too closely.

For instance pcb1173, c99rs is a bad option for low to moderate change (10% to 25%), and a good option for larger magnitudes of change. c99 is adequate for fast changing scenarios (10 or 20 iterations) and small or moderate amplitude (10% or 25%), while gj01_99 is better when the interval between changes is larger.

In general, the results depicted in Figure 5.16 reveal that the effect of change depends on the instance. For most instances, the magnitude is the crucial factor to determine the configuration that is better equipped to recover effectiveness immediately after change. In all situations, restarting the trail was not a good option when the intensity of change was small. Conversely, it was usually a good strategy when the intensity strength increases. For some instances, keeping the trail information is an advantage regardless the magnitude of change. This reveals that the ability to discover a good solution immediately after change is less dependent on the instance size, and more on the instances intrinsic features.

c99rs is ideal when the change is so large that the useful information in the trail almost vanishes. As for c99, it is better suited when the information remains

useful, but in those cases it risks premature stagnation. The dual caste gj01_99 is particularly effective when the information is reusable, but as the ants that are less greedy exploiting the trail, are also less greedy when using the heuristic information so, in average, the solutions immediately after the change may be less optimised. Still its less greedy nature is useful to avoid premature convergence.

5.4.3 Multi-caste ACS relative performance box-plots

To offer a clearer idea of the performance of the various configurations, we will present some examples, using box-plots. As there is a total of 140 scenarios tested, and since it would be impractical to show them all, we will select only a few representative ones. In order to cover the different possibilities, we selected 6 environments combining two frequencies of change (fast: every 10 iterations; slow: every 200 iterations) and three magnitudes of change (small: 10%; large: 50%; extreme: 90%).

For each of these scenarios we selected an instance in such a way that: for both fast and slow scenarios, we should use one small, one medium and one large instance; also, for a given magnitude of change, we should have either a small and large instance, or a small and a medium, or a medium and a large. Table 5.17 depicts the instances selected. It is worth noting that the scenarios depicted do not cover the full extent of possibilities, namely the moderate frequency and intensity of change scenarios of larger instances, where the multi-caste configurations exhibit a particularly good performance. Also, and since the number of configurations considered in the study (52) is too large to allow for a comfortable analysis, the performance box-plots will only include the best performing configurations: c99, c99rs, sj01_99, gj01_99, gj05_99, gj10_99 and gj50_99.

To compare the performances we normalised the results, so, instead of the actual $P_{offline}$ (Equation 2.15), we use P_n computed using Equation 5.3,

$$P_n = \frac{P_{offline}}{P*} \tag{5.3}$$

where $P*$ is the best $P_{offline}$ obtained by any of the configurations (c99, c99rs, sj01_99, gj01_99, gj05_99, gj10_99 gj50_99) for the present repetition of the experiment. We chose to normalise by repetition, since each used a different test

		magnitude		
		10	50	90
frequency	200	kroA200	pr1002	rat783
	10	pcb1173	rat575	kroA100

Figure 5.17: DTSP environments selected

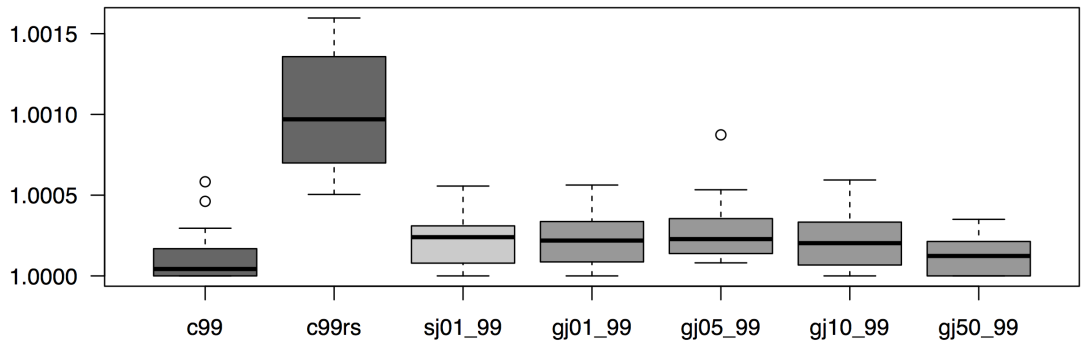

Figure 5.18: Offline performance: kroA100, change of magnitude 10, every 200 iterations

file, built as described in Section5.3 and, as consequence, the performance values achieved, varied sensibly according to the run.

Large interval between changes

kroA200 m10f200 On Figure 5.18 we may observe the performance achieved on the small instance kroA100, with change of magnitude 10, every 200 iterations. It is a slow changing environment, so, as expected, c99 is the best performing configuration, closely followed by the SuperJump and GreedyJump. The knowledge gathered justifies the difference between the c99 and c99rs. Multi-caste configurations, in spite of having one caste with a very low q_0, were able to retain the advantage provided by the previously acquired knowledge.

pr1002 m50f200 Figure 5.19 displays results achieved on the large instance pr1002, for a scenario with a change of magnitude 50, each 200 iterations. This

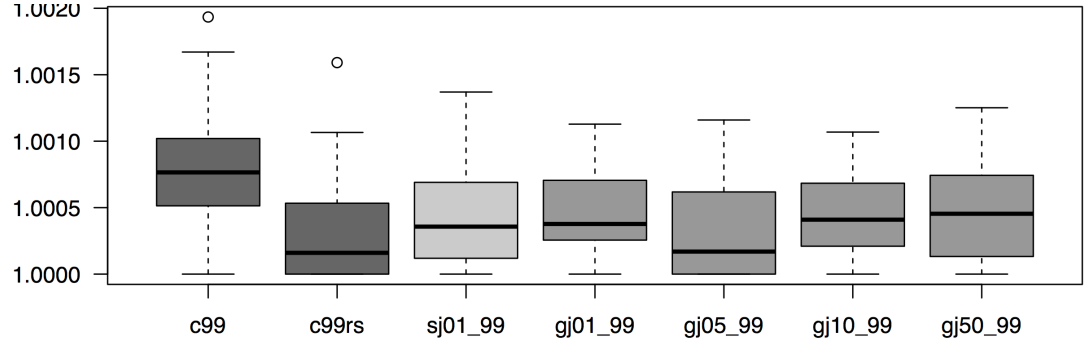

Figure 5.19: Offline performance: pr1002, change of magnitude 50, every 200 iterations

is a situation with a large magnitude of change and c99rs achieves better results than c99. Given the size of the instance, the interval between changes is not enough to allow for a proper resampling of the search space. Also, given the large changes, it is unlikely that the acquired knowledge is very useful and, indeed, can become detrimental by misleading the search efforts. The multi-caste approaches, by having one caste with a low q_0, can ignore the information to a certain degree, and as such, achieve a better performance than c99.

rat783 m90f200 The results on medium sized instance rat783, under a magnitude of change of 90, every 200 iterations can be observed on Figure 5.20. This is a scenario characterised by an extreme magnitude of change and the largest interval between modifications. Again, the magnitude of change is so large that it renders the previous knowledge irrelevant and even detrimental. In this case, c99rs is clearly superior, closely followed by the multi-caste configurations, while c99 is unable to reach the same results.

Small interval between changes

pcb1173 m10f010 Figure 5.21 displays the relative performance on the large instance pcb1173, with a magnitude of change of 10, every 10 iterations. This is an extremely fast scenario, albeit the magnitude of change is small. In this case c99 is better than c99rs, suggesting that the information gathered is beneficial. Multi-

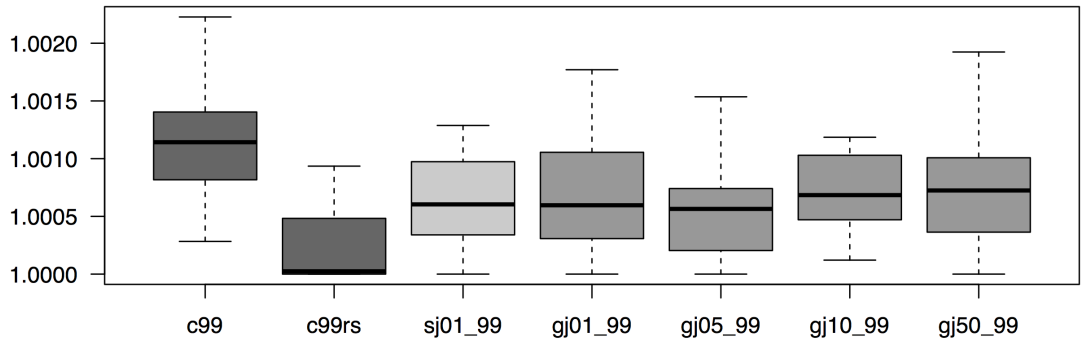

Figure 5.20: Offline performance: rat783, change of magnitude 90, every 200 iterations

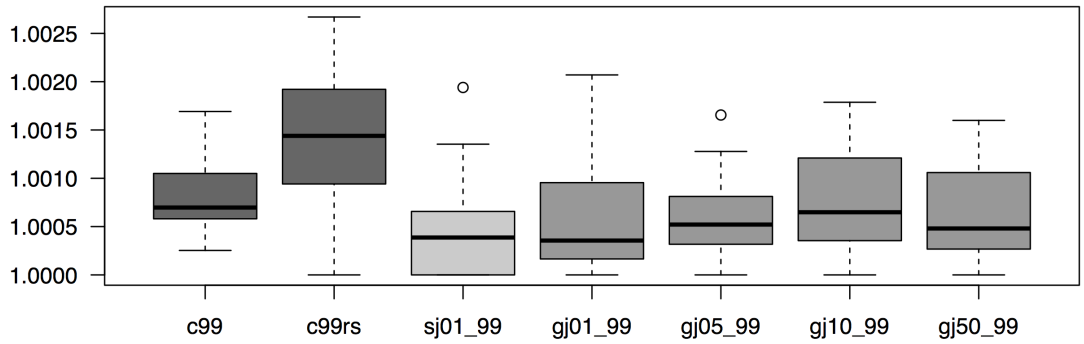

Figure 5.21: Offline performance: pcb1173, change of magnitude 10, every 10 iterations

caste configurations are able to perform even better, so being able to sometimes disregard the knowledge, seems to provide additional advantage. Configurations sj01_99 is particularly successful, indicating that a higher amount of movement between the castes is important (this fact is further confirmed by the very good results of j01_99).

rat575 m50f010 The relative performance on instance rat575 with a magnitude of change 50, every 10 iterations, is depicted in Figure 5.22. This is a medium instance, with a large magnitude of change and very small interval between changes. The large magnitude of change, together with an interval between changes too small to acquire sufficient information, makes this environment clearly

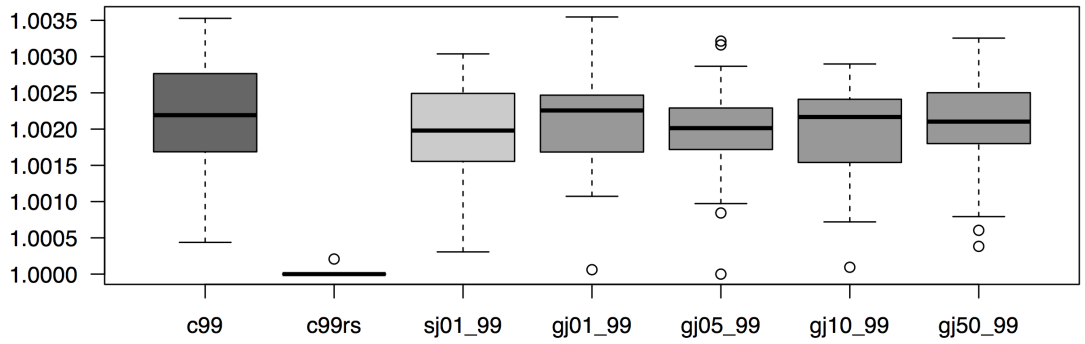

Figure 5.22: Offline performance: rat575, change of magnitude 50, every 10 iterations

better suited for c99rs. The multi-caste configurations gj05_99 and gj10_99 are a little better than c99, while the others are comparable to c99.

kroA100 m90f010 The results achieved in instance kroA100, under a change of intensity 90 every 10 iteration, can be observed in 5.23. Extreme change coupled with high frequency is the most challenging environment to the multi-caste configurations. Variant c99rs is the best for this environment, but, given the small size of the instance, all the configurations depicted have similar performances. The only exception is sj01_99 that is inferior to c99rs, but still considered similar to c99. This is likely due to the fact that, since the instance is small, and we are using an effective local search procedure, the influence of the specific configurations is less marked. On larger instances we can observe that multi-caste configurations have similar, or occasionally better outcomes, to c99, even though c99rs is clearly superior.

5.4.4 Multi-caste ACS castes' size

We also examined the movements between castes to gain a better understanding of the various multi-caste migration strategies. Once more, due to the large amount of scenarios we will only depict a few of them that are, nevertheless, representative of the global trend. Since we are interested in the effects of various migration strategies on the castes' size, we will not only focus on the best performing multi-

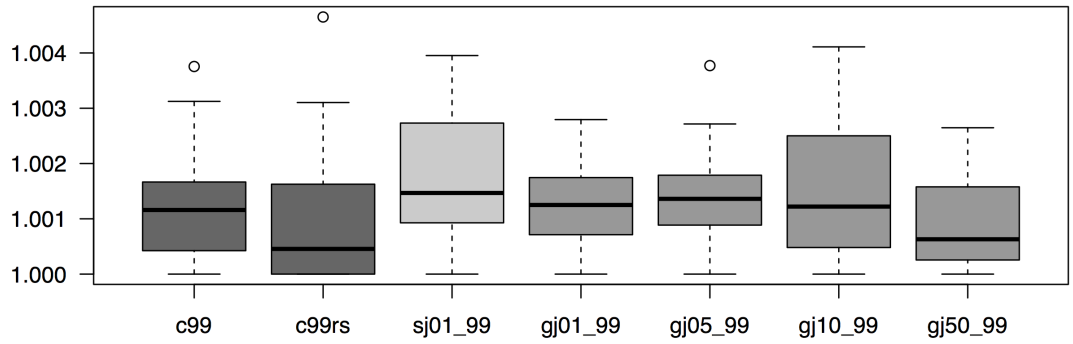

Figure 5.23: Offline performance: kroA100, change of magnitude 90, every 10 iterations

caste configurations, but also in some less successful ones. When referring to the best or worse performing configurations for a given instance and scenario, we will restrict ourselves to c99, c99rs and the multi-caste configurations being illustrated.

pcb1173 - m10f010 Figure 5.24 depicts the evolution of the size of the caste with lower q_0 for configurations j01_99, sj01_99 and gj01_99 when solving the large instance pcb1173. The scenario considered has very frequent changes (every 10 iterations) of small magnitude (10). Figure 5.25 refers to the same scenario when solved by configurations j95_99, sj95_99, and gj95_99.

Best configurations are j01_99 and sj01_99, with gj01_99 very close behind, then gj95_99 and sj95_99 and last j95_99. The best mono-caste is c99, which is similar to gj95_99, whereas c99rs is similar to j95_99 and sj95_99. The average size of the lower q_0 castes in the end of the optimisation is depicted in Figure 5.26.

In the 01_99 configurations (Figure 5.24) we can observe an alteration in the size of the castes with a $q_0 = 0.01$, immediately after each change. As usual, the alteration is more visible in the SuperJump and Jump migration strategies, but is also present in the GreedyJump. After each change, at first there is a small increase in the size of the less greedy caste, followed by a decrease, until it reaches a dimension close to its minimum size.

The variations in size in the 95_99 Jump and SuperJump configurations (Figure 5.25) are less directly related to the change, as the solutions obtained by each caste are not so markedly different. Then, there is less pressure to increase the

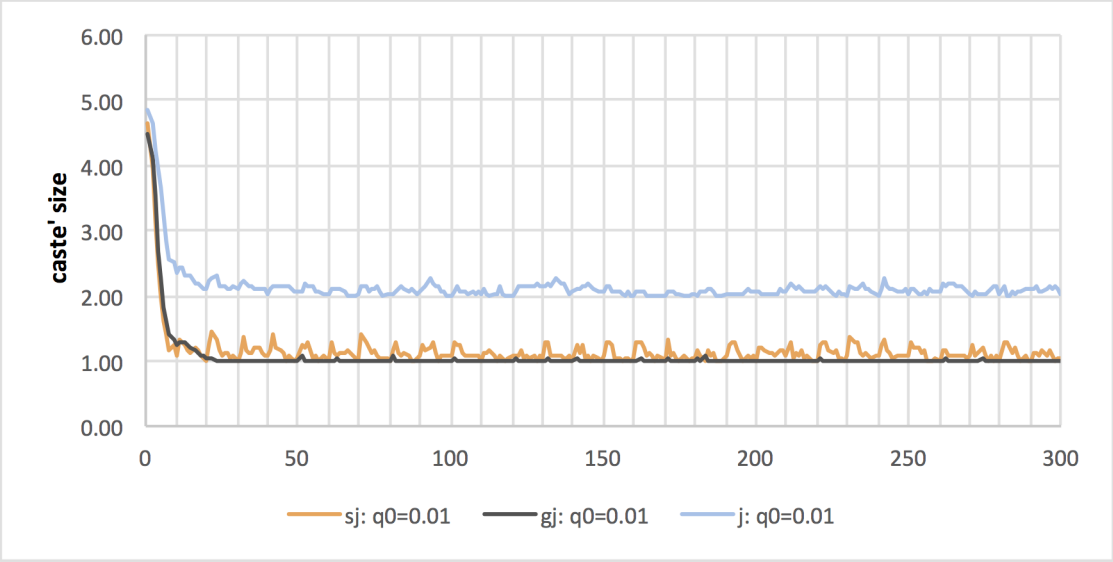

Figure 5.24: Evolution of the size of lower $q0$ caste on 01_99 configurations: pcb1173, change of magnitude 10, every 10 iterations

size of the lower q_0 caste after each change. Regardless of the migration strategy, the size of the caste with the smaller q_0 is larger in the 95_99 configurations than in the 01_99 counterparts, again because there is less pressure to increase the greediness once the change is surpassed.

pcb1173 - m90f100 The evolution of the size of caste with lower q_0 for configurations j01_99, sj01_99 and gj01_99, when solving instance the large instance pcb1173 can be observed in Figure 5.28. The scenario considered has an extreme magnitude of change (90) and large intervals between changes (100 iterations).

The best configuration is c99rs, followed by j01_99, then sj01_99 and gj01_99 with similar performances, and finally c99, worse than the rest. The average size of the caste $q_0 = 0.01$ in the end of the optimisation is depicted in Figure 5.29.

On all configurations, the lower q_0 caste size increases immediately following the change, but decreases again afterwards. GreedyJump has the smallest increase in the caste size and, in average, decreases after 2 iterations. The other two configurations have a larger increase and stay about 20 out of every 100 iterations above the minimum caste size. Also, for Jump and SuperJump, the change in

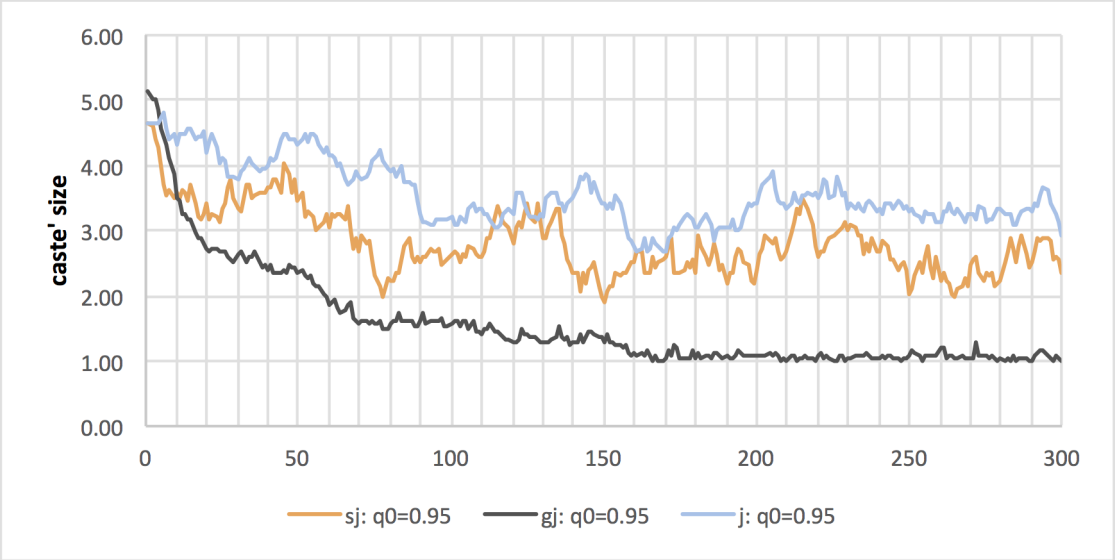

Figure 5.25: Evolution of the size of lower $q0$ caste on 95_99 configurations: pcb1173, change of magnitude 10, every 10 iterations

pcb1173 - m10f010 - average castes' size			
	j01_99:	sj01_99:	gj01_99:
q_0=0.01	2.14	1.15	1.06
	j95_99:	sj95_99:	gj95_99:
q_0=0.95	3.06	2.83	1.75

Figure 5.26: Average size of the castes with lower q0 at the end of the optimisation for configurations 01_99 and 95_99: pcb1173, change of magnitude 10, every 10 iterations

pcb1173 - m90f100 - average castes' size			
	j01_99:	sj01_99:	gj01_99:
q_0=0.01	2.02	1.03	1.00

Figure 5.27: Average size of the caste with lower q0 at the end of the optimisation for configurations 01_99: pcb1173, change of magnitude 90, every 100 iterations

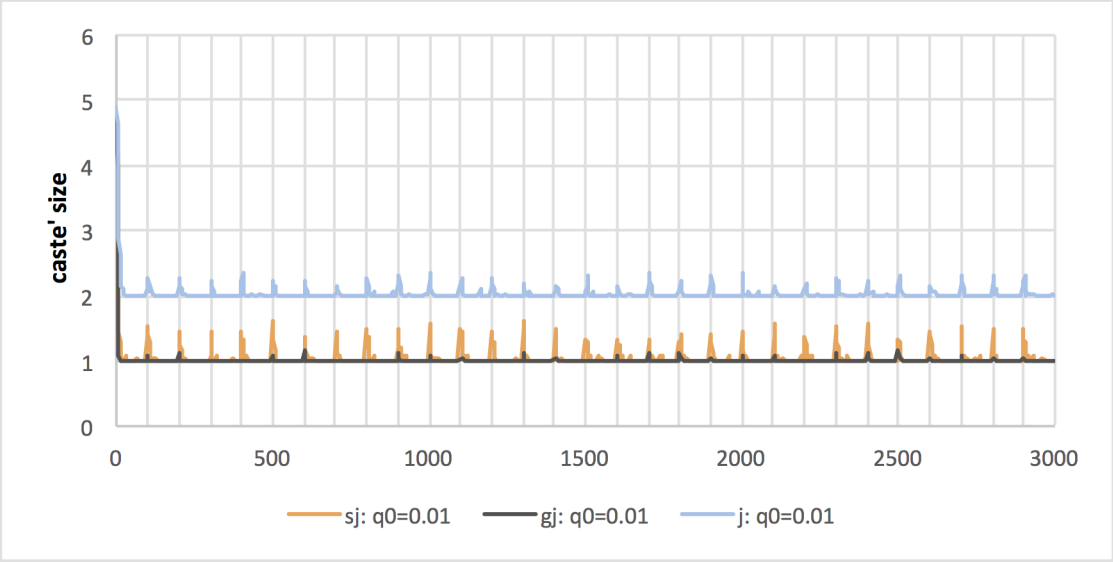

Figure 5.28: Evolution of the size of lower $q0$ caste on 01_99 configurations: pcb1173, change of magnitude 90, every 100 iterations

kroA200 - m25f010 - average castes' size			
	j01_99:	sj01_99:	gj01_99:
q_0=0.01	2.19	1.26	1.07

Figure 5.29: Average size of the caste with lower q0 at the end of the optimisation for configurations 01_99: kroA200, change of magnitude 25, every 10 iterations

the lower q_0 caste is not restricted to iterations immediately following the change. Clearly a larger number of very low q_0 value is important in this scenario, as shown by the superior performance of j01_99. Still, the simple existence of a very low q_0 caste, even if of smaller size, is also important.

kroA200 - m25f010 Figure 5.30 depicts the evolution of the size of the caste with lower q_0 for configurations j01_99, sj01_99 and gj01_99 when solving the small instance kroA200. The scenario considered has very frequent changes (every 10 iterations) of moderate magnitude (25). The best configurations are gj01_99, c99 and sj01_99, followed by j01_99, and finally c99rs. The average size of the caste $q_0 = 0.01$ in the end of the optimisation is depicted in Figure 5.29.

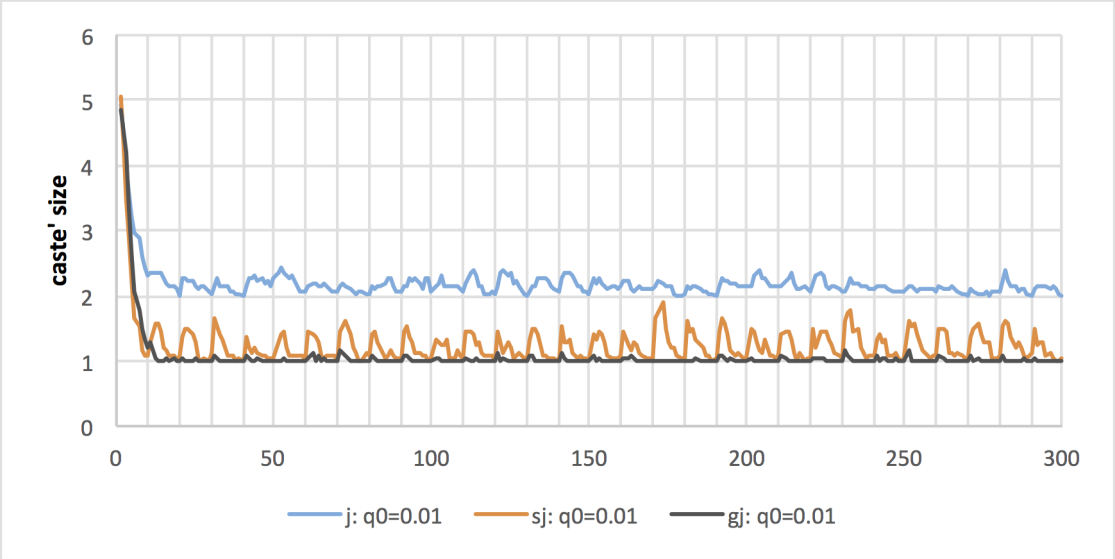

Figure 5.30: Evolution of the size of lower $q0$ caste on 01_99 configurations: kroA200, change of magnitude 25, every 10 iterations

The interval between changes is so small, given the magnitude of the change, that there is little time for the more aggressive caste to grow back after each change. This is specially patent on sj01_99 and j01_99, where the lower q_0 valued caste almost never reaches the minimum size and has a comparatively large average size. Even gj01_99 stays at the minimum size only about 75% of the time. For all migration strategies the largest alterations in the castes' size is immediately after each change. The fact that the best configurations is gj01_99 suggests that, having a caste with a lower q_0 is important but, in very fast environments, the ability to rapidly increase the size of the more aggressive caste is a crucial advantage.

rat575 - m50f020 Figure 5.32 (respectively, Figures 5.33 and 5.34) depicts the evolution of the size of each caste for quad-caste configurations with 8 ants using the Jump migration strategy (respectively, SuperJump and GreedyJump), while solving moderate sized instance rat575. The scenario considered has frequent changes (every 20 iterations) of moderate magnitude (50). The average size of each caste at the end of the optimisation is shown in Figure 5.31.

The best configuration is c99rs, then at distance c99, followed by sjquads08

rat575 - m50f020 - average castes' size				
	q_0=0.75	q_0=0.90	q_0=0.95	q_0=0.99
j75quads08:	1.15	1.37	1.72	3.76
sj75quads08:	1.07	1.22	1.69	3.96
gj75quads08:	1.01	1.17	1.56	4.27

Figure 5.31: Average size the castes on j75quads08 configuration at the end of the optimisation: rat575, change of magnitude 50, every 20 iterations

and gjquads08, and finally j75quads08. All charts depict a similar trend where caste $q_0 = 0.99$ increases in size until it reaches almost its maximum dimension. All other castes decrease in size and maintain a pattern where the size of the caste increases with the q_0 value, so $q_0 = 0.75$ is the smallest, $q_0 = 0.90$ is the second smallest, and so on.

The evolution of the castes' size of j75quads08 (Figure 5.32) shows a lot of movement between the castes, before stabilising at about 50 iterations. This movement is most evident immediately after a change, when the $q_0 = 0.99$ caste diminishes and the lower q_0 valued castes increase in size.

The overall shape of the evolution of the size of the caste on sj75quads08 (Figure 5.33) is similar to that of j75quads08, but only the dimension of the $q_0 = 95$ caste is similar, as SuperJump favours the highest q_0, and have smaller castes of lower q_0. It also takes a little longer to stabilise and the movement tends to be smoother, so the variations in size are less marked.

gj75quads08 outcome is similar to the one of sj75quads08, but the evolution of the castes' size is different. For gj75quads08 (Figure 5.34) castes' size have much less variation and from iteration 450 onward the size of the castes remain mostly stable, with only small oscillations immediately after a change.

5.5 Conclusion

The multi-caste approach, together with four migration strategies (Const, Jump, SuperJump and GreedyJump), were applied to the dynamic, periodic, non cyclic, version of the TSP. The dynamic nature of this problem is the ideal scenario to confirm the enhanced robustness of the self-adaptive multi-caste framework. We

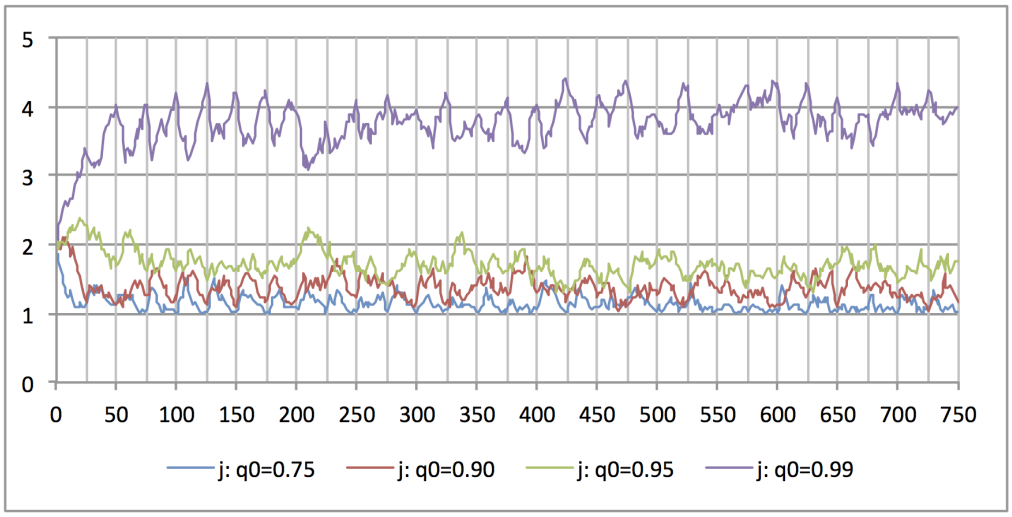

Figure 5.32: Evolution of the size the castes on j75quads08 configuration: rat575, change of magnitude 50, every 20 iterations

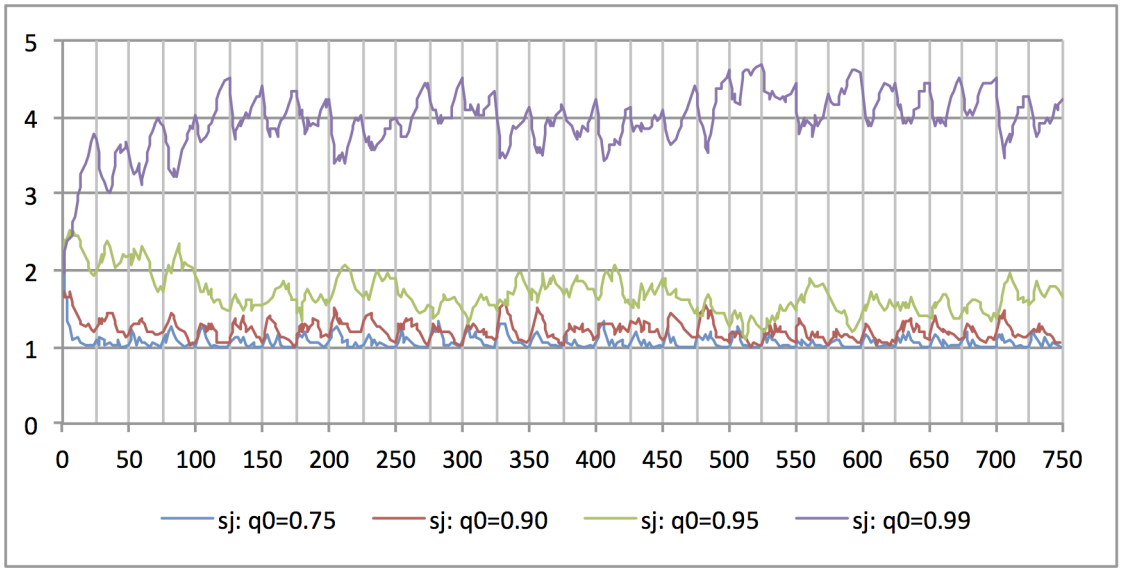

Figure 5.33: Evolution of the size the castes on sj75quads08 configuration: rat575, change of magnitude 50, every 20 iterations

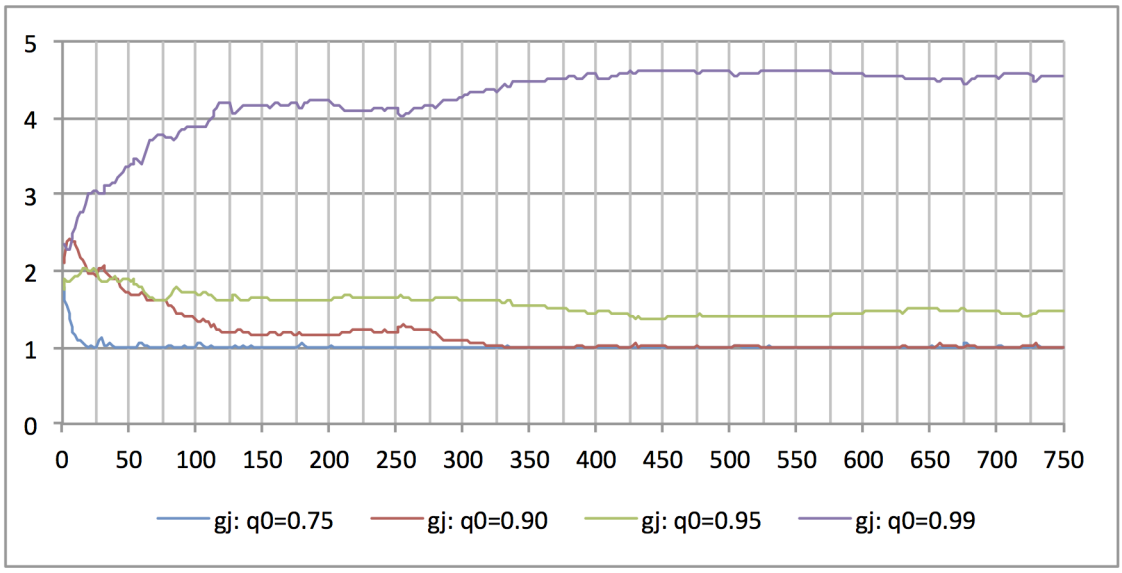

Figure 5.34: Evolution of the size the castes on gj75quads08 configuration: rat575, change of magnitude 50, every 20 iterations

tested our approach against the most successful conventional ACS configurations, including the one that restarts the trail after each change.

The overall best performing ACS configurations are those using a high q_0, and there is a almost perfect division between the cases where it is best to restart the trail, and those where it is best to re-use it. The information provided by the trail is important if the knowledge is still valid in following scenarios, but it can be misleading if the information is irrelevant or erroneous. Fast, radical changing scenarios benefit from restarting the trail, but that classification is a relative measure, it depends on the particular instance and in a real-world situation it might not be available.

Not surprisingly, the overall best performing multi-caste configurations were those that combined a caste with a very high q_0 with another with a very low q_0. Again, resembling the combination of the best performing ACS configurations, since ants with a very low q_0 will disregard the trail. Also, in the dynamic version of the TSP, the migration strategy becomes more relevant, and the most competitive configurations tend to use either the SuperJump or GreedyJump variant. Also, configurations with four castes are only competitive in the smaller instances, or in

scenarios where the change is less intense or less frequent.

Another interesting way to apply the multi-caste approach would be to use specific β (relative importance given to the heuristic information when compared with the trail), since a low q_0 means, not only disregard the trail, but also the heuristic information. Still, as mentioned before, using β would have the disadvantage of requiring a larger adaptation of the ACS. One other option would be to consider ρ (trail evaporation parameter). That would be a more drastic approach, as it could potentially allow for large portions of the trail to be erased. Also it would require deeper alterations of the algorithm, since ρ affects the trail, and not directly the ants behaviour. Possibly, it would need for more than one trail to be kept at all times.

6

Conclusion

Swarm Intelligence algorithms are effective methods for hard optimisation situations. They are inspired by the principles of biological self-organisation and collective intelligence and comprise a set of robust computational procedures able to tackle complex, noisy and dynamic environments. ACO is a relevant subset of swarm intelligence, comprising a collection of methods loosely inspired by the pheromone-based strategies of ant foraging.

It is widely accepted that an effective setting of parameters enhances the behaviour of stochastic optimisation methods. ACO algorithms are particularly sensitive to this issue. However, parameter tuning for a specific situation is difficult and it requires a deep understanding of both the algorithm properties and the problem being addressed. Also, the optimisation conditions change throughout the run and, as a consequence, the optimal parameter setting may also change. This is particularly relevant when addressing dynamic optimisation environments.

In this dissertation we proposed two novel self-adaptive ACO frameworks. Both approaches are able to autonomously adjust their search strategy by means of an online adaptation of parameters. Self-adaptation has two main advantages: it removes from the user the burden of carefully selecting an appropriate setting for a given problem and it grants the algorithm the ability to adjust its search behaviour to the different stages of the optimisation run. The effectiveness of the proposed approaches was assessed with a selected set of combinatorial optimisation problems. Both static and dynamic situations were considered in the experiments.

6.1 Main Achievements

We adopted different guidelines to develop the self-adaptive approaches proposed in this work. MC-Ant is a multi-colony ACO, where each group independently maintains its own trail and search behaviour. Adaptation occurs through migration and parameter sharing. Conversely, Multi-caste ACS defines a single colony with a shared knowledge pool. Ants self-organise in groups, each pursuing a different exploration strategy.

6.1.1 MC-Ant

In MC-Ant, multiple trails and different settings allow the coexistence of different search strategies. Self-adaptation is achieved by a moderate migration flow between colonies, coupled with periodic parameter sharing. The approach was tested on NPP, a topology optimisation problem that has practical application in the assignment of virtual topologies to optical networks.

Results presented in Chapter 3 clearly show that the existence of multiple colonies enhance the robustness of the algorithm and foster the discovery of promising solutions. The analysis reveals that the multi-colony framework is able to adjust the parameters to the specific situation being addressed, thereby removing the need to perform a careful settings prior to the optimisation. Also, MC-Ant adjusts the migration frequency to the different search stages, promoting a strong exchange of information in the beginning of the optimisation and then gradually decreasing as the search progresses.

MC-Ant has a few drawbacks, particularly, its high computational cost, which is mainly due to the need to simultaneously maintain several pheromone and heuristic matrices. To address this issue we developed Multi-caste ACS, an alternative self-adaptive ACO with low computational overhead.

6.1.2 Multi-caste ACS

In Multi-caste ACS, a colony is composed of several castes, *i.e.*, a group of ants that adopts a specific q_0 value. The value of this parameter determines the tendency of ants to either explore new areas of the search space or, instead, exploit existing

information. Several migration strategies allow for the adjustment of the castes size, so the overall behaviour of the colony can shift according to the search needs.

Multi-caste ACS and the TSP Several variants of Multi-caste ACS, both in what concerns to the migration strategy and the caste configuration, were tested on the TSP. Results show that standard ACS is sensitive to the q_0 value adopted. Also, the optimal setting for this parameter changes according to the instance. The optimal settings for the multi-caste approach also change with the instance being addressed but, particularly in situations where migration is possible, the framework is able to autonomously adapt its behaviour and exhibits a robust performance across all tests.

When compared to the two best performing classic ACS configurations, the multi-caste approach combining the q_0 values of the standard variants was able to perform just as well as the best ACS, for every tested optimisation scenario. The ability to combine the qualities of the individual q_0 values is also evident when looking at the relative error, as several multi-caste configurations exhibit a robust behaviour delimited by the two best mono-caste.

Experiments were performed both with and without local search. As expected, the application of local optimisation smoothed the performance of the algorithm. In any case, the general conclusions stand for both sets of experiments.

Multi-caste ACS and the DTSP The robustness of Multi-caste ACS was also tested in several periodic, non cyclic, DTSP instances, with varying frequency and magnitude of change.

Results obtained with the classic ACS show the impact of the chosen configuration. Standard ACS with a q_0 of 0.99, with and without restart mechanism obtains opposite outcomes, depending on the scenario. Frequent and strong changes are better tackled by the algorithm with restart, whereas slow and mild changes are better addressed by the variant that keeps the trail. This is a result that confirms how the effectiveness of standard ACS is dependant on the parameter settings. Multi-caste ACS is able to keep a balanced performance. Those configurations that combine a low and a high q_0 and that allow migration between castes are robust and able to adapt to different dynamic scenarios, being competitive with

the best ACS algorithms specifically designed to handle a given situation.

A direct comparison between two classic ACS and one of the most effective Multi-caste ACS configurations, gj01_99, was assessed in chapter 5. The results confirm our main claims, as they clearly show that the optimal ACS configuration depends on the instance and scenario, whilst the multi-caste configuration is more robust and it exhibits a visible adaptive behaviour.

A detailed analysis of the behaviour of the algorithm reveals that the oscillation in caste size depends, as expected, of the combination of q_0 values, migration strategy and change scenario. As a rule the greedier caste is the larger one and it stays close to the maximum size. The largest change in caste size occurs immediately after each change. In that occasion, the less greedy caste increases in size, decreasing again some iterations afterwards.

6.2 Future Work

There are several possible avenues for future work. One possibility is to extend the application of the proposed frameworks to other problems. Of particular interest are dynamic environments, possibly with different properties from those addressed in this dissertation.

Another interesting path to follow is to expand and improve the frameworks proposed in this work. MC-ANT maintains several independent pheromone matrices, thus incurring in a high computational overhead. This efficiency bottleneck compromises its application to large instances and one possibility to overcome this limitation is to develop a distributed MC-Ant framework. If we look at the second proposal from this work, we can see that Multi-caste ACS is the simplest possible framework, as it deals with a single parameter. Following this line of research it would be interesting to generalise the approach to other parameters. One possibility is to add ρ to the set of parameters that characterise a caste. As ρ rules evaporation, having a caste with low q_0 and high ρ could be advantageous in case of a pronounced change in the environment. Also, specially interesting for dynamic environments with pronounced change, could be the inclusion of a caste with the ability to disregard the trail but still consider the heuristic information.

Bibliography

[Angus and Hendtlass, 2002] Angus, D. and Hendtlass, T. (2002). Ant colony optimisation applied to a dynamically changing problem. In Hendtlass, T. and Ali, M., editors, *IEA/AIE 2002 Proceedings: Developments in Applied Artificial Intelligence*, pages 618–627, Cairns, Australia. Springer-Verlag Berlin Heidelberg.

[Angus and Hendtlass, 2005] Angus, D. and Hendtlass, T. (2005). Dynamic Ant Colony Optimisation. *APPLIED INTELLIGENCE*, 23(1):33–38.

[Angus and Woodward, 2009] Angus, D. and Woodward, C. (2009). Multiple objective ant colony optimisation. *Swarm Intelligence*, 3(1):69–85.

[Banerjee and Sarkar, 2001] Banerjee, S. and Sarkar, D. (2001). Hypercube connected rings: A scalable and fault-tolerant logical topology for optical networks. *Computer communications*, 24:1079.

[Baransel et al., 1995] Baransel, C., Dobosiewicz, W., and Gburzynski, P. (1995). Routing in multihop packet switching networks: Gb/s challenge. *IEEE Network*, 9(3):38–61.

[Bentley, 1992] Bentley, J. L. (1992). Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing*, 4:387–411.

[Botee and Bonabeau, 1998] Botee, H. and Bonabeau, E. (1998). Evolving ant colony optimization. *Advances in complex systems*, 1:149–159.

[Brackett, 1990] Brackett, C. A. (1990). Dense Wavelength Division Multiplexing Networks: principles and applications. *IEEE JASC*, 8(6):948–964.

[Branke, 2002] Branke, J. (2002). *Evolutionary Optimization in Dynamic Environments*. Kluwer Academic Publishers.

[Bullnheimer et al., 1997] Bullnheimer, B., Hartl, R. F., and Strauss, C. (1997). A New Rank Based Version of the Ant System: A Computational Study. Technical report, Institute of Management Science, University of Vienna, Austria.

[Bullnheimer et al., 1998] Bullnheimer, B., Kotsis, G., and Strauß, C. (1998). Parallelization strategies for the ant system. *High Performance Algorithms and Software in Nonlinear Optimization Applied Optimization*, 24:87–100.

[Caro et al., 2004] Caro, G., Ducatelle, F., and Gambardella, L. M. (2004). AntHocNet : an Ant-Based Hybrid Routing Algorithm for Mobile Ad Hoc Networks. In *Parallel Problem Solving from Nature - PPSN VIII, 8th International Conference, LNCS 3242*, pages 461–470, Birmingham, UK. Springer Berlin Heidelberg.

[Caro et al., 2008] Caro, G. a. D., Ducatelle, F., and Gambardella, L. M. (2008). Theory and practice of Ant Colony Optimization for routing in dynamic telecommunications networks. In Sala, N. and Orsucci, F., editors, *Reflecting interfaces: the complex coevolution of information technology ecosystems*, pages 185–216. Idea Group, Hershey, PA, USA.

[Caro and Dorigo, 1998] Caro, G. D. and Dorigo, M. (1998). AntNet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9(August):317–365.

[Chu and Zomaya, 2006] Chu, D. and Zomaya, A. (2006). Parallel Ant Colony Optimization for 3D Protein Structure Prediction using the HP Lattice Model. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 177–198. IEEE.

[Colorni et al., 1991] Colorni, A., Dorigo, M., and Maniezzo, V. (1991). Distributed Optimization by Ant Colonies. *Proceedings of the first European Conference on Artificial Life*, 142(or D):134–142.

[Deneubourg et al., 1990] Deneubourg, J. L., Aron, S., Goss1, S., and Pasteels, J. M. (1990). The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3(2).

[Dorigo et al., 2006] Dorigo, M., Birattari, M., and Stützle, T. (2006). Ant colony optimization artificial ants as a computational intelligence technique. *IEEE Computational Intelligence Magazine*, 1(4):28–39.

[Dorigo and Blum, 2005] Dorigo, M. and Blum, C. (2005). Ant colony optimization theory: A survey. *Theoretical Computer Science*, 344(2-3):243–278.

[Dorigo and Gambardella, 1997] Dorigo, M. and Gambardella, L. M. (1997). Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(4):53–66.

[Dorigo et al., 1991] Dorigo, M., Maniezzo, V., and Colorni, A. (1991). Positive feedback as a search strategy. Technical report, Laboratorio di Calcolatori, Dipartimento di Elettronica - Politecnico di Milano.

[Dorigo et al., 1996] Dorigo, M., Maniezzo, V., and Colorni, A. (1996). Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41.

[Dorigo and Stützle, 2004] Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. A Bradford Book. MIT Press, Cambridge Massachussetts.

[Dorigo and Stützle, 2010] Dorigo, M. and Stützle, T. (2010). Ant Colony Optimization: Overview and Recent Advances. In Gendreau, M. and Potvin, J.-Y., editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 227–263. Springer US, Boston, MA.

[Eaton and Yang, 2014] Eaton, J. and Yang, S. (2014). Dynamic Railway Junction Rescheduling using Population Based Ant Colony Optimisation. In *UKCI'2014, 14th UK Workshop on Computational Intelligence*, University of Bradford, UK.

[Ellabib et al., 2007] Ellabib, I., Calamai, P., and Basir, O. (2007). Exchange strategies for multiple Ant Colony System. *Information Sciences: an International Journal*, 177(5):1248–1264.

[Escario et al., 2012] Escario, J. B., Jimenez, J. F., and Giron-Sierra, J. M. (2012). Optimisation of autonomous ship manoeuvres applying Ant Colony Optimisation metaheuristic. *Expert Systems with Applications*, 39(11):10120–10139.

[Escario et al., 2015] Escario, J. B., Jimenez, J. F., and Giron-Sierra, J. M. (2015). Ant Colony Extended: Experiments on the Travelling Salesman Problem. *Expert Systems with Applications*, 42(1):390–410.

[Eyckelhof and Snoek, 2002] Eyckelhof, C. J. and Snoek, M. (2002). Ant Systems for a Dynamic TSP. In *ANTS'2002, LNCS 2463, Ant Algorithms: Third International Workshop*, pages 88–99. 10.1007/3-540-45724-0_8.

[Field et al., 2012] Field, A., Miles, J., and Field, Z. (2012). *Discovering statistics using R*. SAGE Publications Limited.

[Fischer et al., 2005] Fischer, T., StuÌĹtzle, T., Hoos, H., and Merz, P. (2005). An analysis of the hardness of TSP instances for two high-performance algorithms. In *6th Metaheuristics International Conference (MIC)*, pages 361–367, Vienna, Austria.

[Floreano and Mattiussi, 2008] Floreano, D. and Mattiussi, C. (2008). *Bio-Inspired Artificial Intelligence*. MIT Press.

[Freire and da Silva, 2001] Freire, M. M. and da Silva, H. J. A. (2001). Performance Comparison of Wavelength Routing Optical Networks with Chordal Ring and Mesh-Torus Topologies. In *Networking- ICN 2001*, pages 358–367. Springer Berlin Heidelberg.

[Gaertner, 2004] Gaertner, D. (2004). *Natural Algorithms for Optimisation Problems Final Year Project Report*. Master's thesis, Imperil College, London, UK.

[Gambardella et al., 1999] Gambardella, L., Taillard, É., and Agazzi, G. (1999). MACS-VRPTW : A MULTIPLE ANT COLONY SYSTEM FOR VEHICLE ROUTING PROBLEMS WITH TIME WINDOWS. *New ideas in optimization*, pages 1–17.

[Gambardella and Dorigo, 1995] Gambardella, L. M. and Dorigo, M. (1995). Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem. In Prieditis and Russell, S., editors, *Proceedings of ML-95*, pages 252–260. Morgan Kaufmann.

[Ganz and Koren, 1991] Ganz, A. and Koren, Z. (1991). WDM passive star-protocols and performance analysis. In *IEEE INFCOM '91. The conference on Computer Communications. Tenth Annual Joint Comference of the IEEE Computer and Communications Societies Proceedings*, pages 991–1000 vol.3, Bal Harbour, Florida, USA. IEEE.

[García-Martínez et al., 2007] García-Martínez, C., Cordón, O., and Herrera, F. (2007). A taxonomy and an empirical analysis of multiple objective ant colony optimization algorithms for the bi-criteria TSP. *European Journal of Operational Research*, 180(1):116–148.

[Ghimire et al., 2014] Ghimire, B., Cohen, D., and Mahmood, A. (2014). Parallel cooperating ant colonies with improved periodic exchange strategies. In *HPC '14 Proceedings of the High Performance Computing Symposium*, pages 25–30, Tampa, Florida, USA. Society for Computer Simulation International.

[Goss et al., 1989] Goss, S., Aron, S., Deneubourg, J.-L., and Pasteels, J. M. (1989). Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76:579–581.

[Guinand and Pigné, 2010] Guinand, F. and Pigné, Y. (2010). Short and Robust Communication Paths in Dynamic Wireless Networks. In *ANTS'2010, LNCS 6234*, pages 520–527.

[Gunes et al., 2002] Gunes, M., Sorges, U., and Bouazizi, I. (2002). ARA-the ant-colony based routing algorithm for MANETs. In *Proceedings. International Conference on Parallel Processing Workshop*, pages 79–85. IEEE Comput. Soc.

[Guntsch and Middendorf, 2001] Guntsch, M. and Middendorf, M. (2001). Pheromone Modification Strategies for Ant Algorithms Applied to Dynamic TSP. In *EvoWorkshops (2001)*, pages 213–222.

[Guntsch and Middendorf, 2002a] Guntsch, M. and Middendorf, M. (2002a). A population based approach for ACO. In *Applications of Evolutionary Computing - EvoWorkshop 2002: EvoCOP, EvoISAP, EvoSTIM/EvoPLAN*, pages 72–81. Springer-Verlag.

[Guntsch and Middendorf, 2002b] Guntsch, M. and Middendorf, M. (2002b). Applying Population Based ACO to Dynamic Optimization Problems. In *ANTS'2002, LNCS 2463, Ant Algorithms: Third International Workshop*, pages 111–122.

[Guntsch et al., 2001] Guntsch, M., Middendorf, M., and Schmeck, H. (2001). An ant colony optimization approach to dynamic TSP. In *GECCO 2001: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 860–867. Morgan Kaufmann Publishers, 2001.

[Janson et al., 2005] Janson, S., Merkle, D., and Middendorf, M. (2005). Parallel Ant Colony Algorithms. In Alba, E., editor, *Parallel Metaheuristics: A New Class of Algorithms*, volume 47, chapter 8, pages 171–201. John Wiley & Sons.

[Johnson and McGeoch, 1997] Johnson, D. S. and McGeoch, L. A. (1997). The Traveling Salesman Problem: A Case Study in Local Optimization. In *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons.

[Katayama et al., 2007] Katayama, K., Yamashita, H., and Narihisa, H. (2007). Variable Depth Search and Iterated Local Search for the Node Placement Problem in Multihop WDM Lightwave Networks. In *IEEE Congress on Evolutionary Computation, 2007*, pages 3508–3515.

[Kato et al., 1999] Kato, M., Kawakita, R., and Nagamochi, H. (1999). A Reconfigurative Algorithm for Torus Lightwave Networks. *Electronics and Communications in Japan*, 82(6).

[Kato and Oie, 2000] Kato, M. and Oie, Y. (2000). Reconfiguration algortihms based on meta-heuristics for multihop WDM lightwave networks. In *Proceedings IEEE International Conference on Communications*, volume 3, pages 1638–1644. Ieee.

[Komolafe and Harle, 2003] Komolafe, O. and Harle, D. (2003). Optimal node placement in an optical packet switching Manhattan street network. *Computer Networks*, 42(2):251–260.

[Laptik, 2011] Laptik, R. (2011). Ant System Initial Parameters Distribution. *Electronics and Electrical Engineering*, 110(4):85–88.

[Leguizamón and Alba, 2013] Leguizamón, G. and Alba, E. (2013). Ant Colony Based Algorithms for Dynamic Optimization Problems. *Metaheuristics for Dynamic Optimization*, 433:189–210.

[Li et al., 2006] Li, C., Yang, M., and Kang, L. (2006). A new approach to solving dynamic traveling salesman problems. In *Simulated Evolution and Learning 2006, LNCS 4247*, pages 236–243. Springer-Verlag Berlin Heidelberg.

[Li, 2011] Li, W. (2011). A parallel multi-start search algorithm for dynamic traveling salesman problem. *Experimental Algorithms*, pages 65–75.

[Lin, 1965] Lin, S. (1965). Computer Solutions of the Traveling Salesman Problem. *Bell System Technical Journal*, 44(10):2245–2269.

[Liu, 2005] Liu, J.-l. (2005). Rank-based ant colony optimization applied to dynamic traveling salesman problems. *Engineering Optimization*, 37(8):831–847.

[Marsan et al., 1992] Marsan, M., Abertengo, G., Francese, A., Leonardi, E., and Neri, F. (1992). All-optical bidirectional Manhattan networks. In *[Conference Record] SUPERCOMM/ICC '92 Discovering a New World of Communications*, pages 1461–1467. IEEE.

[Mavrovouniotis et al., 2015] Mavrovouniotis, M., Müller, F. M., and Yang, S. (2015). An Ant Colony Optimization Based Memetic Algorithm for the Dynamic Travelling Salesman Problem. In *Proceedings of the 2015 on Genetic and*

Evolutionary Computation Conference - GECCO '15, pages 49–56, New York, New York, USA. ACM Press.

[Mavrovouniotis et al., 2017a] Mavrovouniotis, M., Muller, F. M., and Yang, S. (2017a). Ant Colony Optimization With Local Search for Dynamic Traveling Salesman Problems. *IEEE Transactions on Cybernetics*, 47(7):1743–1756.

[Mavrovouniotis et al., 2017b] Mavrovouniotis, M., Van, M., and Yang, S. (2017b). Pheromone modification strategy for the dynamic travelling salesman problem with weight changes. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE.

[Mavrovouniotis and Yang, 2010] Mavrovouniotis, M. and Yang, S. (2010). Ant Colony Optimization with Immigrants Schemes in Dynamic Environments. In *PPSN'10 Proceedings of the 11th international conference on Parallel problem solving from nature: Part II*, pages 371–380, Berlin, Heidelberg. Springer-Verlag.

[Mavrovouniotis and Yang, 2011a] Mavrovouniotis, M. and Yang, S. (2011a). A memetic ant colony optimization algorithm for the dynamic travelling salesman problem. *Soft Computing*, 15(7):1405–1425.

[Mavrovouniotis and Yang, 2011b] Mavrovouniotis, M. and Yang, S. (2011b). An Immigrants Scheme Based on Environmental Information for Ant Colony Optimization for the Dynamic Travelling Salesman Problem. In *EA'2011, Artificial Evolution: 10th International Conference, Evolution Artificielle*. Springer.

[Mavrovouniotis and Yang, 2011c] Mavrovouniotis, M. and Yang, S. (2011c). Memory-based immigrants for ant colony optimization in changing environments. In *EvoApplications 2011:Applications of Evolutionary Computation, Part I, LNCS 6624*, pages 324–333. Springer.

[Mavrovouniotis and Yang, 2013a] Mavrovouniotis, M. and Yang, S. (2013a). Adapting the pheromone evaporation rate in dynamic routing problems. In Esparcia-Alcazar, A., editor, *Applications of Evolutionary Computation, LNCS 7835*. Springer Berlin Heidelberg.

[Mavrovouniotis and Yang, 2013b] Mavrovouniotis, M. and Yang, S. (2013b). Ant colony optimization with immigrants schemes for the dynamic travelling salesman problem with traffic factors. *Applied Soft Computing*, 13(10):4023–4037.

[Mavrovouniotis and Yang, 2014a] Mavrovouniotis, M. and Yang, S. (2014a). Ant algorithms with immigrants schemes for the dynamic vehicle routing problem. *Information Sciences*, 294(October):456–477.

[Mavrovouniotis and Yang, 2014b] Mavrovouniotis, M. and Yang, S. (2014b). Ant Colony Optimization with Self-Adaptive Evaporation Rate in Dynamic Environments. In *Proceedings of the 2014 IEEE Symposium on Computational Intelligence in Dynamic and Uncertain Environments*, pages 47–54. IEEE.

[Mavrovouniotis and Yang, 2014c] Mavrovouniotis, M. and Yang, S. (2014c). Interactive and Non-Interactive Hybrid Immigrants Schemes for Ant Algorithms in Dynamic Environments. In *CEC 2014: IEEE Congress on Evolutionary Computation*, pages 1542–1549.

[Mavrovouniotis et al., 2014] Mavrovouniotis, M., Yang, S., and Yao, X. (2014). Multi-Colony Ant Algorithms for the Dynamic Travelling Salesman Problem. In *Proceedings of the 2014 IEEE Symposium on Computational Intelligence in Dynamic and Uncertain Environments*, pages 9–16, Orlando, Florida. IEEE.

[Maxemchuk, 1987] Maxemchuk, N. (1987). Routing in the Manhattan Street Network. *IEEE Transactions on Communications*, 35(5):503–512.

[Maxemchuk, 1985] Maxemchuk, N. F. (1985). Regular Mesh Topologies in Local and Metropolitan Area Networks. *AT&T Technical Journal*, 64(7):1659–1685.

[Melo, 2009] Melo, L. (2009). Multi-colony ant colony optimization for the node placement problem. In *GECCO 2009: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2713–2716, Montreal, Québec, Canada. ACM.

[Melo et al., 2009] Melo, L., Pereira, F., and Costa, E. (2009). MC-ANT: a multi-colony ant algorithm. In Collet, P., Monmarché, N., Legrand, P., Shoenauer, M., and Lutton, E., editors, *EA'2009 LNCS 5975, Artificial Evolution: 9th*

International Conference, Evolution Artificielle, pages 25–36. Springer Berlin / Heidelberg.

[Melo et al., 2011] Melo, L., Pereira, F., and Costa, E. (2011). Multi-caste Ant Colony Optimization Algorithms. In Antunes, L., Pinto, H. S., Prada, R., and Trigo, P., editors, *EPIA 2001: 15th Portuguese Conference on Artificial Intelligence - Local Proceedings*, pages 978–989, Lisbon.

[Melo et al., 2013a] Melo, L., Pereira, F., and Costa, E. (2013a). Effective Multi-caste Ant Colony System for Large Dynamic Traveling Salesperson Problems. In Lutton, E., Schoenauer, M., Hao, J.-K., Monmarché, N., and Legrand, P., editors, *EA'2013 LNCS 8752, Artificial Evolution: 11th International Conference, Evolution Artificielle*.

[Melo et al., 2013b] Melo, L., Pereira, F., and Costa, E. (2013b). Multi-caste ant colony algorithm for the dynamic traveling salesperson problem. In Tomassini, M., editor, *ICANNGA 2013, LNCS 7824*, pages 179–188. Springer, Heidelberg.

[Melo et al., 2014] Melo, L., Pereira, F., and Costa, E. (2014). Extended experiments with Ant Colony Optimization with heterogeneous ants for Large Dynamic Traveling Salesperson Problems. In *CPS ICCSA14 proceedings*.

[Michel and Middendorf, 1999] Michel, R. and Middendorf, M. (1999). An ACO Algorithm for the Shortest Common Supersequence Problem. In *New ideas in optimization*, chapter 4, pages 51–62. McGraw-Hill Ltd., UK, Maidenhead, UK, England.

[Middendorf et al., 2002] Middendorf, M., Reischle, F., and Schmeck, H. (2002). Multi Colony Ant Algorithms. *Journal of Heuristics*, 8(3):305–320.

[Mukherjee, 1992a] Mukherjee, B. (1992a). WDM-based local lightwave networks. I. Single-hop systems. *IEEE Network*, 6(3):12–27.

[Mukherjee, 1992b] Mukherjee, B. (1992b). WDM-based local lightwave networks. II. Multihop systems. *IEEE Network*, 6(4):20–32.

[Pedemonte et al., 2011] Pedemonte, M., Nesmachnow, S., and Cancela, H. (2011). A survey on parallel ant colony optimization. *Applied Soft Computing*, 11(8):5181–5197.

[Pellegrini et al., 2006] Pellegrini, P., Favaretto, D., and Moretti, E. (2006). On MAXâĂŞMIN Ant System's Parameters. *ANTS'2006, LNCS 4150, Ant Colony Optimization and Swarm Intelligence*, 4150:203–214.

[Psaraftis, 1988] Psaraftis, H. (1988). Dynamic vehicle routing problems. In *Vehicles Routing: Methods and Studies*, pages 223–248. Elsevier Science Publishers.

[R, 2011] R, D. (2011). R: A language and environment for statistical computing.

[Randall and Lewis, 2002] Randall, M. and Lewis, A. (2002). A Parallel Implementation of Ant Colony Optimization. *Journal of Parallel and Distributed Computing*, 62(9):1421–1432.

[Reinhelt, 1995] Reinhelt, G. (1995). {TSPLIB}: a library of sample instances for the TSP (and related problems) from various sources and of various types. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[Ridge, 2007] Ridge, E. (2007). *Design of experiments for the tuning of optimisation algorithms*. Phd thesis, Department of Computer Science, University of York, U.K.

[Ridge and Kudenko, 2008] Ridge, E. and Kudenko, D. (2008). Determining whether a problem characteristic affects heuristic performance. In Cotta, C. and van Hemert, J., editors, *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, volume 153 of *Studies in Computational Intelligence*, chapter 2, pages 21–35. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Sammoud et al., 2009] Sammoud, O., Solnon, C., and Ghedira, K. (2009). A New ACO Approach for Solving Dynamic Problems. In *EA'09, 9th international conference on Artificial Evolution - local proceeding*.

[Schoonderwoerd et al., 1997] Schoonderwoerd, R., Holland, O. E., Bruten, J. L., and Rothkrantz, L. J. M. (1997). Ant-Based Load Balancing in Telecommunications Networks. *Adaptive Behavior*, 5(2):169–207.

[Simões and Costa, 2011] Simões, A. and Costa, E. (2011). Memory-based CHC algorithms for the dynamic traveling salesman problem. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*, page 1037, New York, New York, USA. ACM Press.

[Simões and Costa, 2013] Simões, A. and Costa, E. (2013). Extended virtual loser genetic algorithm for the dynamic traveling salesman problem. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO '13*, page 869, New York, New York, USA. ACM Press.

[Siu and Chang, 2002] Siu, F. and Chang, R. K. C. (2002). Effectiveness of optimal node assignments in wavelength division multiplexing networks with fixed regular virtual topologies. *Computer Networks*, 38:61–74.

[Sörensen et al., 2018] Sörensen, K., Sevaux, M., and Glover, F. (2018). A History of Metaheuristics. In *Handbook of Heuristics*, pages 1–18. Springer International Publishing, Cham.

[Strąk et al., 2017] Strąk, Ł., Skinderowicz, R., and Boryczka, U. (2017). Adjustability of a discrete particle swarm optimization for the dynamic TSP. *Soft Computing*.

[Stützle, 1998] Stützle, T. (1998). Parallelization Strategies for Ant Colony Optimization. *Parallel Problem Solving from NatureâĂŤPPSN V*.

[Stützle, 2002] Stützle, T. (2002). {ACOTSP}: A software package of various ant colony optimization algorithms applied to the symmetric traveling salesman problem. URL:http://www.aco-metaheuristic.org/aco-code/.

[Stutzle and Hoos, 1997] Stutzle, T. and Hoos, H. H. (1997). Max-Min Ant System and Local Search for the Travelling Salesman Problem. In Piscataway T. Bäck, Z. M. and Yao, X., editors, *IEEE International Conference on Evolutionary Computation*, pages 309–314. IEEE Press.

[Stützle and Hoos, 2000] Stützle, T. and Hoos, H. H. (2000). MAX-MIN Ant System. *Future Generation Computer Systems*, 16(8):889–914.

[Stützle et al., 2011] Stützle, T., López-Ibáñez, M., and Dorigo, M. (2011). A Concise Overview of Applications of Ant Colony.

[Stützle et al., 2010] Stützle, T., Lopez-Ibanez, M., Pellegrini, P., Maur, M., de Oca, M. M., Birattari, M., and Dorigo, M. (2010). Parameter Adaptation in Ant Colony Optimization. Technical report number tr/iridia/2010-002, IRIDIA, Bruxelles, Belgium.

[Talbi et al., 1999] Talbi, E., Roux, O., and Fonlupt, C. (1999). Parallel ant colonies for combinatorial optimization problems. In *Parallel and Distributed Processing*, volume 1586, pages 239–247.

[Toyama et al., 2008] Toyama, F., Shoji, K., and Miyamichi, J. (2008). An Iterated Greedy Algorithm for the Node Placement Problem in Bidirectional Manhattan Street Networks. In *GECCO 2008: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 579–584, Atlanta, Georgia, USA. ACM.

[Wang et al., 2016] Wang, Y., Xu, Z., Sun, J., Han, F., Todo, Y., and Gao, S. (2016). Ant Colony Optimization with Neighborhood Search for Dynamic TSP. In *International COnference on swarm intelligence*, pages 434–442.

[Yonezu et al., 2007] Yonezu, M., Funabiki, N., Kitani, T., Yokohira, T., Nakanishi, T., and Higashino, T. (2007). Proposal of a Hierarchical Heuristic Algorithm for Node Assignment in Bidirectional Manhattan Street Networks. *Systems and Computers in Japan*, 38(4).

[Zar, 2009] Zar, J. H. (2009). *Biostatistical Analysis*. Pearsons, 5 edition.

[Zhou et al., 2003] Zhou, A., Kang, L., and Yan, Z. (2003). Solving dynamic TSP with evolutionary approach in real time. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 2, pages 951–957. IEEE.

Appendices

A

Performance boxplots for the TSP without local search - selected configurations

The boxplots of the performance relative error for some selected configurations while not using local search can be consulted in Figures **??**, A.2, A.3, A.4, A.5, A.6, and A.7.

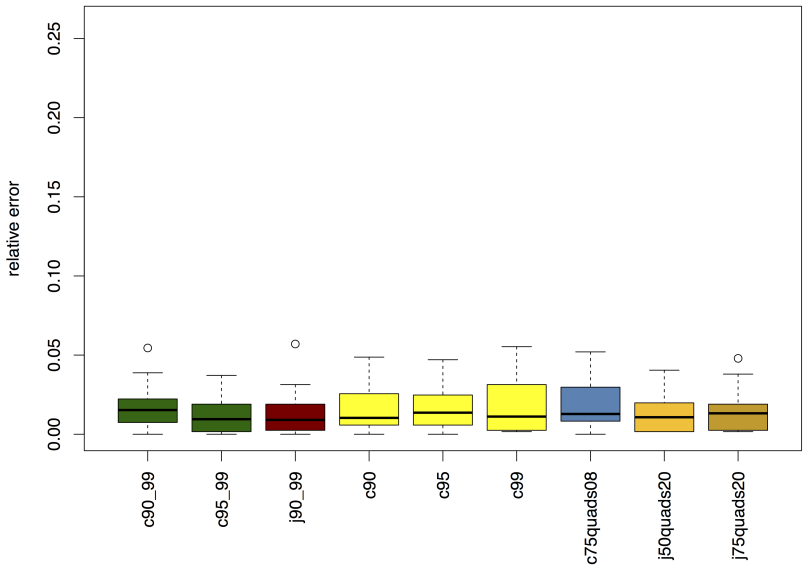

Figure A.1: TSP instance rat99, selected configurations relative error, no local search

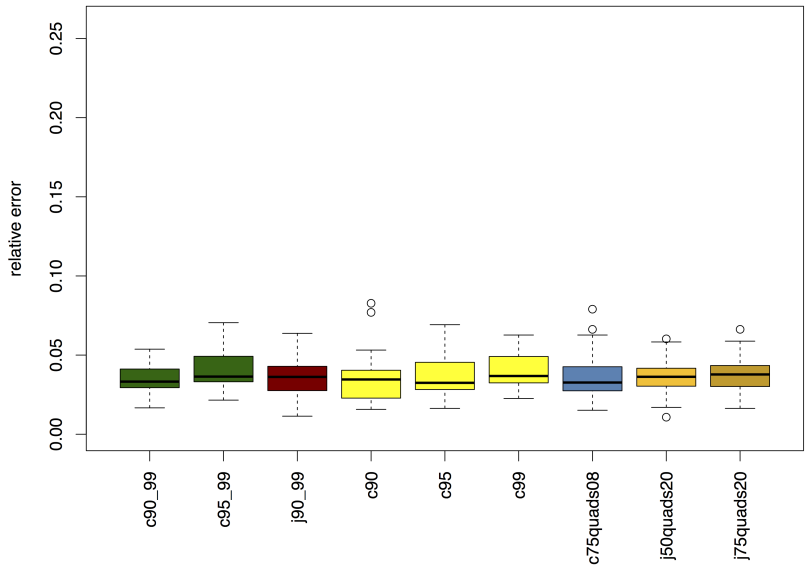

Figure A.2: TSP instance d198, selected configurations relative error, no local search

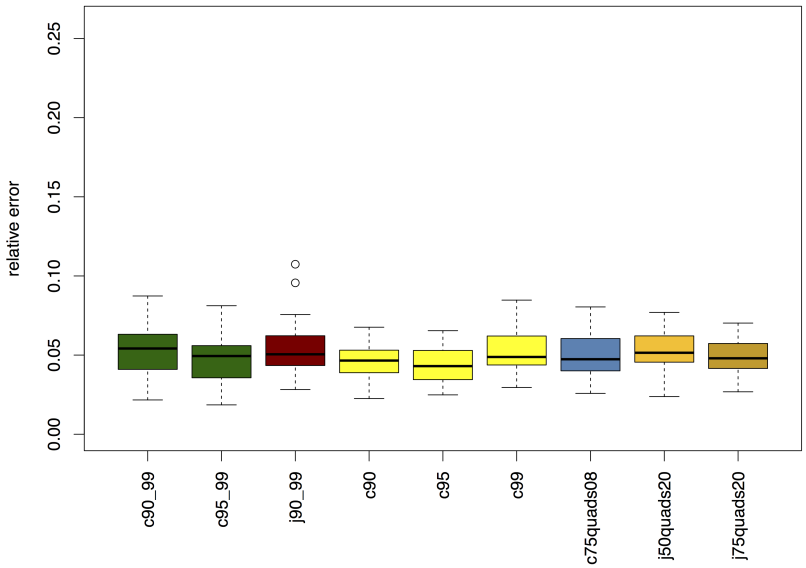

Figure A.3: TSP instance fl417, selected configurations relative error, no local search

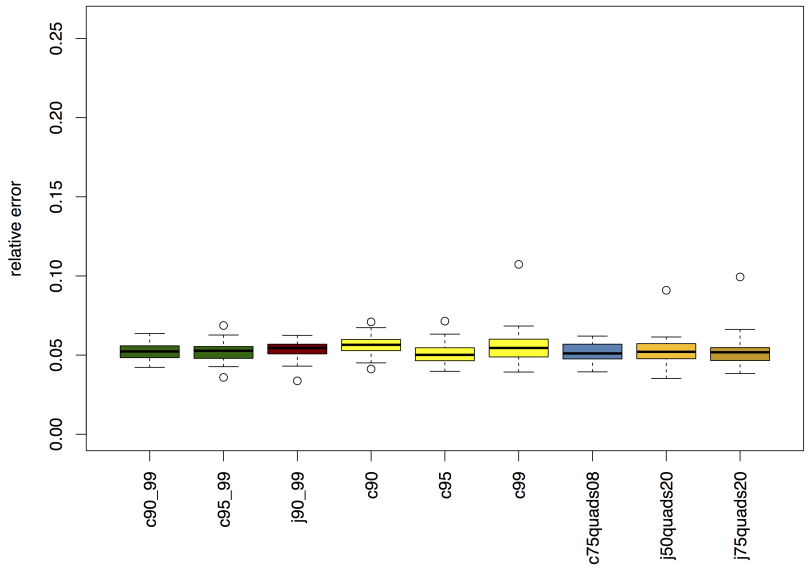

Figure A.4: TSP instance rat783, selected configurations relative error, no local search

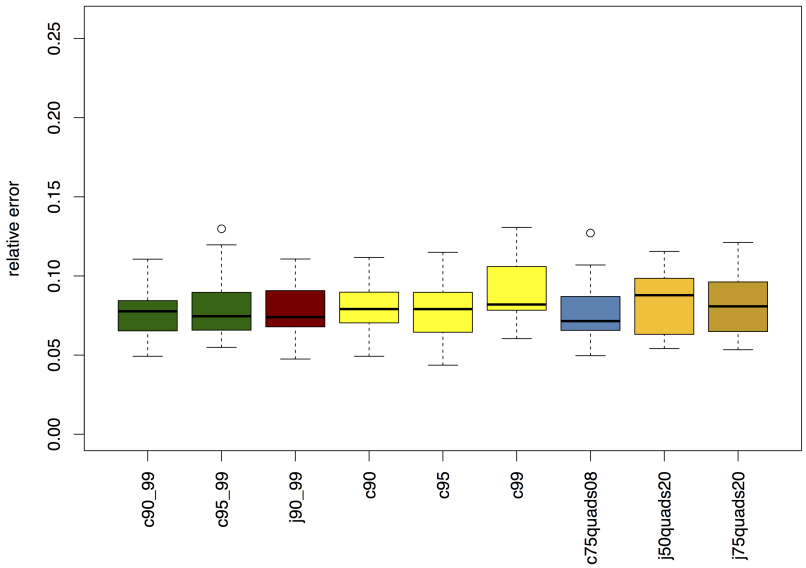

Figure A.5: TSP instance fl1577, selected configurations relative error, no local search

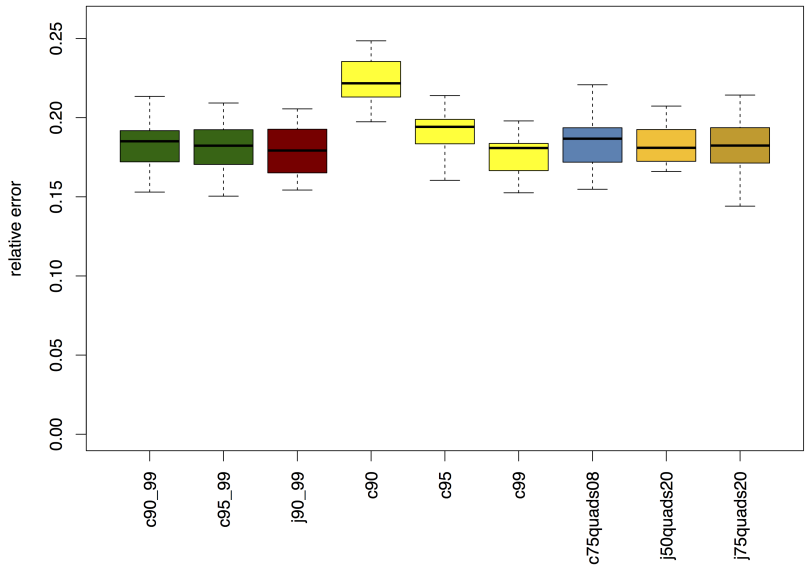

Figure A.6: TSP instance pcb3038, selected configurations relative error, no local search

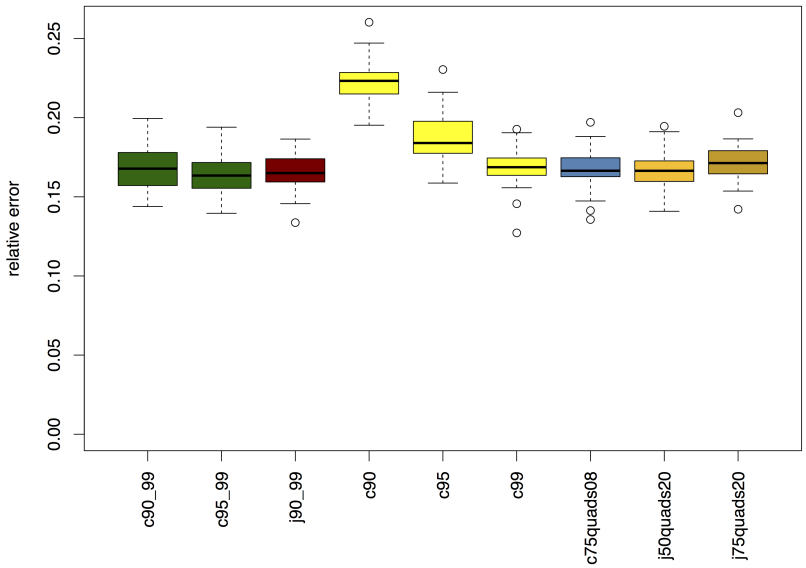

Figure A.7: TSP instance rl5934, selected configurations relative error, no local search

B

Performance boxplots for the TSP with local search - selected configurations

The boxplots of the performance relative error for some selected configurations while using local search can be consulted in Figures B.1, B.2, B.3, B.4, B.5, B.6, and B.7.

I

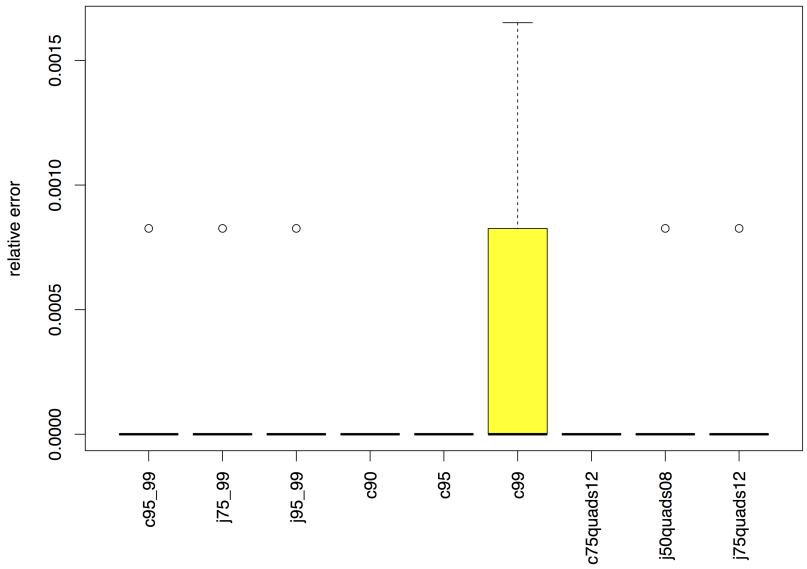

Figure B.1: TSP instance rat99, selected configurations performance, with local search

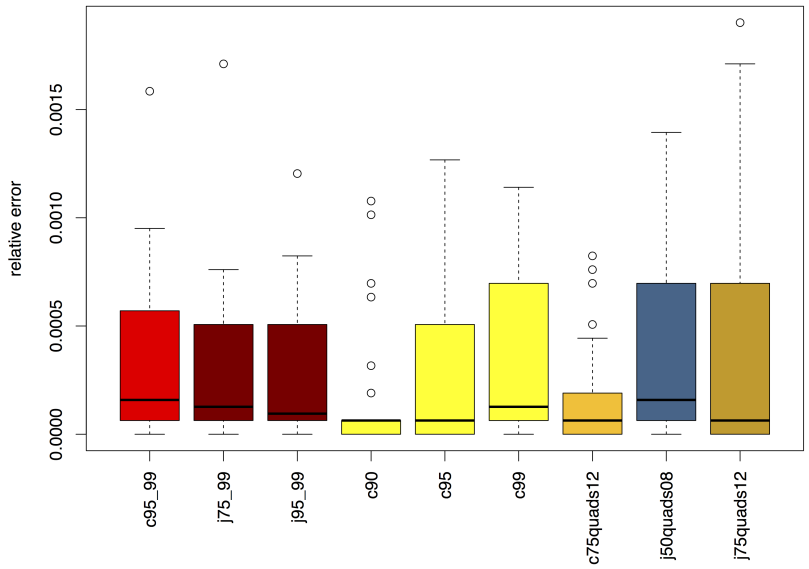

Figure B.2: TSP instance d198, selected configurations performance, with local search

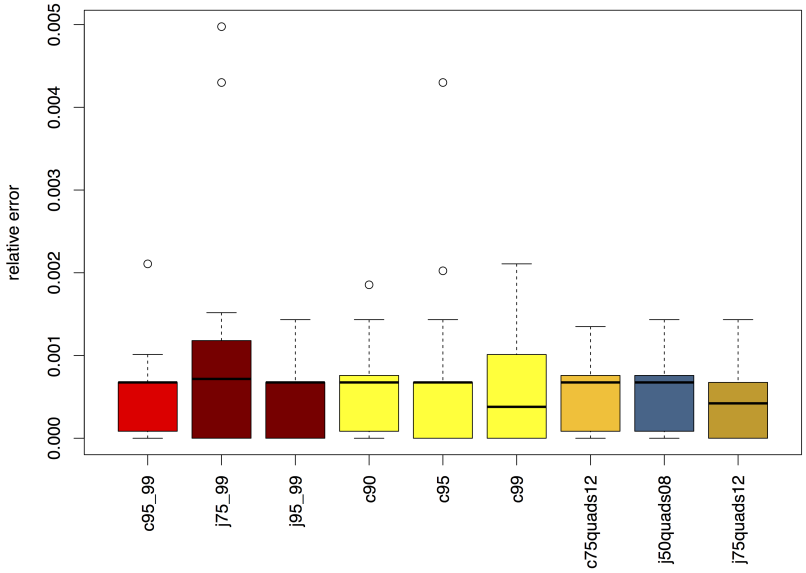

Figure B.3: TSP instance fl417, selected configurations performance, with local search

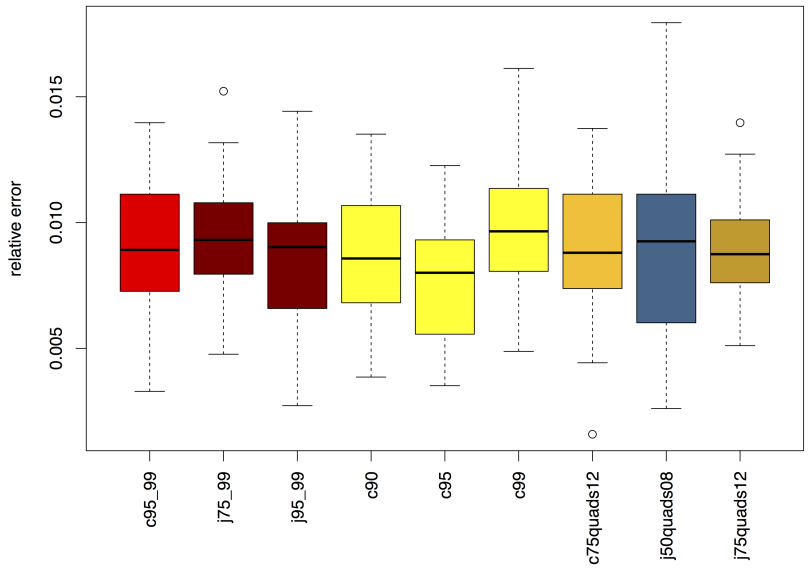

Figure B.4: TSP instance rat783, selected configurations performance, with local search

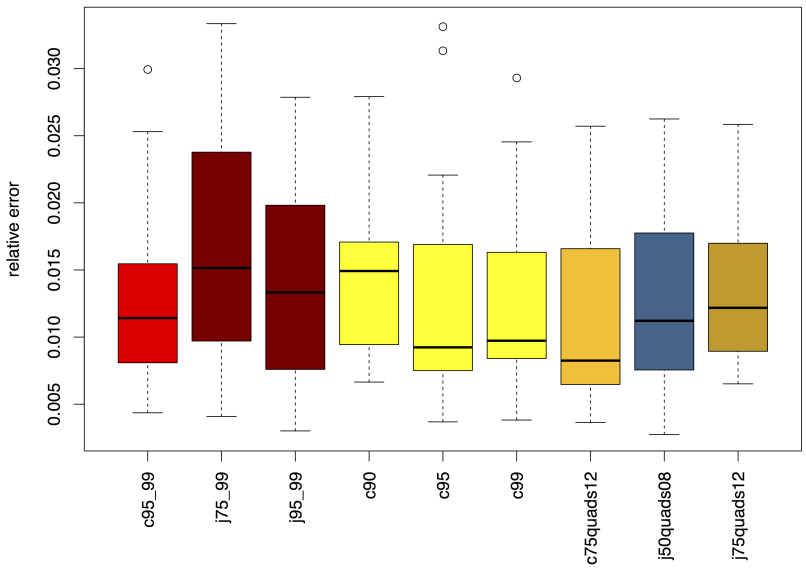

Figure B.5: TSP instance fl1577, selected configurations performance, with local search

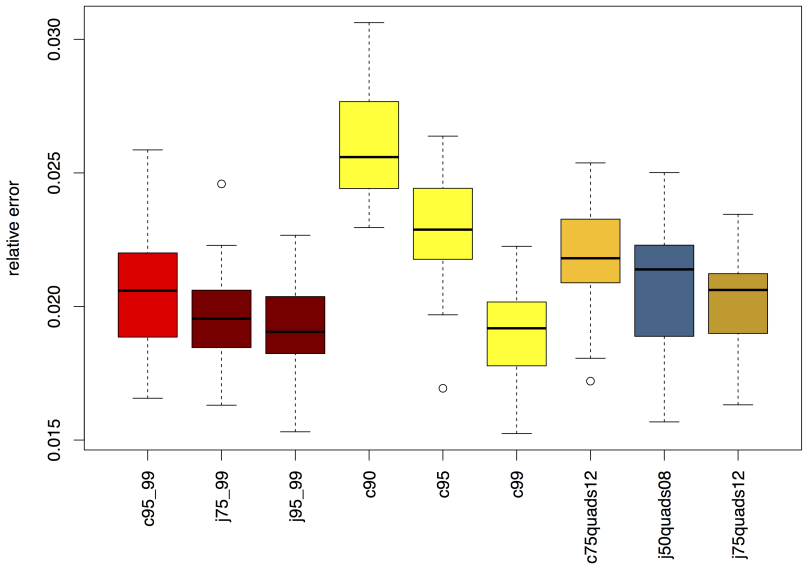

Figure B.6: TSP instance pcb3038, configurations performance, with local search

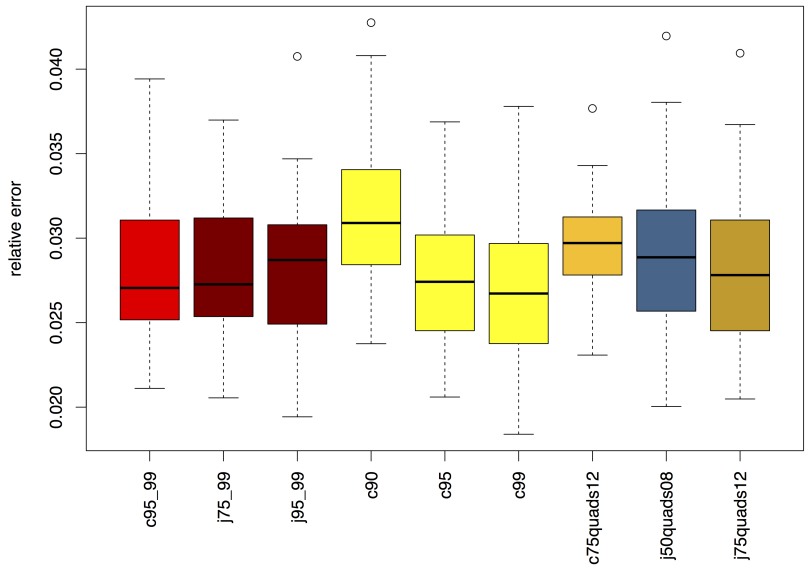

Figure B.7: TSP instance rl5934, selected configurations performance, with local search