Developing Partition Crossovers for Combinatorial Optimisation Problems

Rúben Telmo Domingues Leal

1 2 9 0 Universidade de Coimbra

1 2 9 0

UNIVERSIDADE D
COIMBRA

Rúben Telmo Domingues Leal

Developing Partition Crossovers for Combinatorial Optimisation Problems

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professor Carlos M. Fonseca and presented to Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

This page is intentionally left blank.

Faculty of Sciences and Technology

Department of Informatics Engineering

Developing Partition Crossovers for Combinatorial Optimization Problems

Rúben Telmo Domingues Leal

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems advised by Prof. Carlos M. Fonseca and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

1 2 9 0

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Acknowledgements

First of all, I would like to thank my supervisor, professor Carlos Fonseca, for all the support and guidance, during this work and since I began working with him, before my master's degree. I would also like to thank him for his patience and understanding of all the mistakes I did and the difficulties I faced during this work. I am grateful for everything that I learned while working with him.

To my parents, I want to thank them for providing me all the means necessary to be able to follow my interests. I want to thank my mother for all the effort to make sure that I could focus as much as possible on my work. Furthermore, I want to thank all the people that helped me, in some way, to get through this work.

Finally, I wish to acknowledge that this work was financed by national funds through the Foundation for Science and Technology (FCT) within the scope of the project CISUC – UIDB/00326/2020 under research grant L.726690-Ref 2, and was partially carried out in the scope of project MobiWise: From Mobile Sensing to Mobility Advising (P2020 SAICTPAC/0011/2015), co-financed by COMPETE 2020, Portugal 2020 – Operational Program for Competitiveness and Internationalization (POCI), European Union's European Regional Development Fund (ERDF), and the Foundation for Science and Technology (FCT).

This page is intentionally left blank.

Abstract

Recombination operators play an important role in the performance of Evolutionary Algorithms. They generate a new solution by combining information from other two parent solutions. The Optimal Recombination Problem [8, 9] concerns the generation of the best possible offspring solution by a given operator. However, in many cases, this problem is NP-Hard. In particular, this is true for the Travelling Salesman Problem (TSP) when respectful, edge-transmitting recombination is considered [9].

Partition crossovers are deterministic recombination operators that solve or approximate the ORP. They do so by exploiting natural decompositions of the parents in order to generate high-quality solutions given those decompositions.

Partition Crossovers are usually combined with local search operators. The rules on which these operators operate define the neighbourhood structure of the search space. However, it is not known how Partition Crossovers relate to this neighbourhood structure. We show that indeed, all Partition Crossovers may be geometric under some distance and, for the particular case of current Partition Crossovers for the TSP, they are geometric for the bond distance.

Moreover, partition crossovers have been successfully applied in several optimisation problems. Despite the differences between problems, their implementation follows a common pattern that is generalisable to some extent. Thus, we propose an API for the development of partition crossovers that clearly identifies their basic operations, and separates a problem-dependent part of these operators from the rest of the operator, which is problem-independent.

Such an API brings focus to the relations between the components arising throught the decompositions of the solutions involved, and provide opportunities for improving existing partition crossovers. We present an experimental analysis of the GPX2 [27] partition crossover in the light of the ORP, and show how the proposed API could be used to improve it.

Keywords

Evolutionary Algorithms, Combinatorial Optimisation, Geometric Crossovers, Partition Crossovers, Optimal Recombination Problem

This page is intentionally left blank.

Resumo

Os operadores de recombinação desempenham um papel importante no desempenho de Algoritmos Evolucionários. Eles geram uma nova solução através da combinação de informação de outras duas soluções. O Problema da Recombinação Ótima (PRO) [8, 9] consiste na geração da melhor solução descendente segundo um dado operador. No entanto, em muitos casos este problema é NP-Difícil. Em particular, isto é verdade para o Problema do Caixeiro Viajante (PCV) quando se considera a recombinação com respeito e trasmissão de arestas [9].

Os Cruzamentos de Partição são operadores de recombinação determinística que resolvem ou aproximam o PRO, explorando as decomposições naturais dos pais tendo em vista a geração de soluções de elevada qualidade, dadas essas decomposições.

Geralmente, os Cruzamentos de Partição são combinados com operadores de procura local. As regras sobre as quais estes operadores funcionam definem a estrutura de vizinhança do espaço de procura. No entanto, não se sabe como é que os Cruzamentos de Partição se relacionam com esta estrutura de vizinhança. Mostramos que de facto todos os Cruzamentos de Partição podem ser geométricos sob alguma distância e que, para o caso particular dos Cruzamentos de Partição para o PCV existentes, eles são geométricos de acordo com a distância de *bond*.

Adicionalmente, os Cruzamentos de Partição têm sido aplicados com sucesso em vários problemas de otimização. Apesar das diferenças entre problemas, a sua implementação segue um padrão comum que pode ser generalizado até certo ponto. Portanto, propomos uma Interface de Programação de Aplicações (IPA) para o desenvolvimento de cruzamentos de partição, que identifica claramente as suas operações fundamentais e separa a parte dependente do problema destes operadores, do resto do operador que é independente do problema.

Esta IPA realça as relações entre componentes que surgem das decomposições das soluções involvidas e fornece oportunidades para melhorar os cruzamentos de partição existentes. Apresentamos uma análise experimental do Cruzamento de Partição GPX2 [27] à luz do PRO e mostramos como é que a IPA proposta pode ser usada para o melhorar.

Palavras-Chave

Algoritmos Evolucionários, Otimização Combinatória, Cruzamentos Geométricos, Cruzamentos de Partição, Problema da Recombinação Ótima

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

API Application Programming Interface. xv, 1, 2, 26–30, 32, 36, 40–42

APX Articulation Point Partition Crossover. 18

COP Combinatorial Optimisation Problem. 4, 5

DPX Dynastic Potential Partition Crossover. xv, 18–20, 26, 27

EA Evolutionary Algorithm. 1, 9, 10, 41

GAPX Generalised Asymmetric Partition Crossover. 15

GPX Generalised Partition Crossover. 13, 15

GPX$_e$ Exhaustive Search Generalised Partition Crossover. xvi, 33, 37–40

GPX2 Generalised Partition Crossover 2. xv, 15–17, 25, 27–33, 36–41

II Iterative Improvement. 7–9

ILP Integer Linear Programming. 33

ILS iterated local search. 7–9, 34

IPT Iterative Partial Transcription. 13–15

LKH Lin-Kernighan-Helsgaun. 7, 14

LS Local Search. 7–9, 11, 12, 23, 25

OEPR Optimal Edge Preserving Recombination. xvi, 10, 33, 34, 36, 38, 40

OREPR Optimal Respectful Edge Preserving Recombination. xvi, 10, 33, 34, 36, 38–40

ORP Optimal Recombination Problem. 1, 4, 10, 41

OTM Optimal Tour Merging. 10, 36

PBO Pseudo-Boolean Optimization. 13, 17, 19, 27

PII Probabilistic Iterative Improvement. 8

PX Partition Crossover. 1, 2, 4, 9–12, 14–16, 20, 22, 23, 25, 26, 32, 40–42

PXPBO Partition Crossovers for Pseudo-Boolean Optimisation. 17, 18

PXQAP Partition Crossover for the Quadratic Assignment Problem. 21, 22

QAP Quadratic Assignment Problem. 13, 21

RII Randomised Iterative Improvement. 8

TSP Travelling Salesman Problem. v, 1, 2, 4, 5, 7, 9–11, 13–17, 25, 26, 33, 34, 41, 42

VIG Variable Interaction Graph. 17–19

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

Chapter 1

Introduction

The recombination operators used in evolutionary algorithms are typically stochastic, mostly by analogy with biological recombination. This means that randomly chosen parts of two parents are combined to produce a new candidate solution, or offspring.

One of the challenges in the design and development of these operators concerns the generation of the best possible offspring. This is known as the Optimal Recombination Problem (ORP), which is generally NP-Hard [8, 9].

Partition Crossovers (PXs) are deterministic recombination operators that approximately solve the ORP by exploiting the structure and decompositions of the parents (*deterministically*). They partition two parents into several components, where each component has interchangeable partial solutions from each parent.

PXs have been applied to well known optimisation problems such as Travelling Salesman Problem (TSP) [26, 27, 28, 29], pseudo-Boolean optimisation [4, 25, 25], and the quadratic assignment problem [2]. Provided that the objective function is locally separable with respect to the structure of the parents, there might be more problems were they might be useful.

Moreover, little is known about the neighbourhood structure on which PXs operate and how they relate to other search operators such as the mutation operators of Evolutionary Algorithms (EAs). Using two properties common to all PXs, we show that these operators have the so-called *inbreeding properties* of geometric crossovers [19], so there may exist a distance under which these operators are *geometric* [18].

In the TSP, PXs are usually combined with local search operators with 2-opt movements, defining a 2-opt neighbourhood. With a view to understanding whether PXs for this problem are geometric under the 2-opt distance, we show that they are geometric under the *bond distance* [3], which bounds the 2-opt distance.

Despite the particularities of each problem, these PXs follow the same general structure of implementation, where its fundamental operations are the same regardless of the problem. This suggests that the development of these operators can be generalised, to a certain extent.

With that in mind and to facilitate the development of PXs, we propose an Application Programming Interface (API), that provides an abstraction of PXs that divides them in two parts: a problem dependent part and a problem independent part. In the first, we define the problem dependent parts of PXs; the problem independent part has the algorithm that decides from which parent the offspring inherits each component, which is common

to all PXs. Using this API, only the problem dependent part needs to be implemented, according to the specification. The problme independent part can be re-utilised, providing a way of defining PX operators for other problems.

The proposed API promotes a new way of thinking about the development of PXs based on the relationships that can be established between components. We present an experimental analysis of one PX of the TSP with respect to optimal recombination. We also analyse an idealised version of this operators, and show how this perspective can improve it.

The remainder of this thesis is organised as follows.

In Chapter 2 we present some background concepts required to understand the remaining chapters and to understand the broader scope of this work. Next, we elaborate on PXs and present the state-of-the-art. In Chapter 4, we discuss the geometricity of PXs. The API is defined in Chapter 5 and an experimental analysis of PXs for the TSP is presented in Chapter 6. Finally, we present some conclusions and possible directions for future work in Chapter 7.

This page is intentionally left blank.

Chapter 2

Background Concepts

In this chapter, we present some fundamental concepts required to understand the next chapters and to understand the broader scope encompassing this work. We describe Combinatorial Optimisation Problem (COP). Then we present some approaches to solve hard COPs, followed by introducing one type of operator that is used in some of this approaches and is the focus of this thesis, the crossover operators, in particular Partition Crossovers (PXs) and the broader problem that motivates them, the Optimal Recombination Problems (ORPs). We conclude by addressing the geometric properties crossovers.

2.1 Combinatorial Optimisation Problems

Combinatorial Optimisation Problems (COPs) are present in many areas, such as artificial intelligence, bioinformatics, and operations research, to name a few. They work on discrete, finite sets of objects where the goal is to find the optimal ordering, grouping or assignment of those objects under certain conditions [13].

2.1.1 Instances vs Problems

Before we move on, we should clarify the difference between a problem and an instance of a problem [20]. A problem is an abstraction of all the instances that might exist for this problem. For example, the Travelling Salesman Problem (TSP)[1] can be stated as: "Given a weighted connected graph with n vertices, find the minimum weighted path that begins and ends in the same vertex and passes through all the other vertices exactly once". Such path is called a tour. We cannot solve such problem as stated, since it does not have concrete data. We can only solve an instance of the problem. The instance is a concrete case of the problem, with the corresponding data. For example, an instance of the TSP would have a given weighted connected graph, with a given number of vertices. It could represent the map of the cities of a country, for example.

Formally, assuming minimisation, we can define an instance of an optimisation problem, given a set of feasible solutions S and an objective or evaluation function $f : S \to \mathbb{R}$, as finding a solution $s \in S$, such that [20],

$$f(s) \leq f(x), \forall x \in S \tag{2.1}$$

[1]The TSP is formally defined in the next chapter.

We call s the *globally optimal solution*, or simply *optimal solution*. If we wanted to maximise, equation (2.1) would become: $f(s) \geq f(x), \forall x \in S$.

COPs can be stated either as *maximisation* or *minimisation* problems. The choice is mostly dependent on the "nature" of the problem. For example, the TSP is more naturally formulated as a minimisation problem, since we want the tour with the *lowest cost*; for some other problems, a maximisation formulation is more suited. However, by considering the negative of the objective function, these two formulations are equivalent and can easily be converted to one another. As such, we will only consider minimisation problems from now on, without loss of generality.

2.1.2 Neighbourhoods

Each solution $s \in S$ has a neighbourhood $N : S \rightarrow 2^S$ of points that are close to s according with some distance measure. In COPs, the neighbourhood [20] of a solution $N(s)$ is the set of solutions that can be obtained by applying a single move to s, corresponding to an edit distance equal to one. For example, considering the TSP, this move could be the *2-opt move* [6].

Definition 2.1 (2-opt move)**.** Given a tour of an instance of the TSP, the *2-opt move* consists in removing two edges from the tour, and adding two other edges such that the new tour is also valid.

In this case, the neighbourhood of a solution $s \in S$ for the TSP would be defined as $N(s) = \{x \in S : x \neq s$ where x is obtained by applying a *2-opt move* on $s\}$.

Local Optima

For some important problems, such as those with which this work is concerned, it is challenging, or nearly impossible, to find a global optimum in a timely matter. However, in these cases it is usually possible to find local optima. Given an instance of an optimisation problem, with solution set S, an objective function f, and a neighbourhood structure $N(s), \forall s \in S$, a *local optimum* s^* is defined as [20]:

$$f(s^*) \leq f(s), \forall s \in N(s^*) \tag{2.2}$$

Moreover, if $f(s^*) < f(s), \forall s \in N(s^*)$, then s^* is a *strict local optimum*.

2.1.3 Types of Combinatorial Optimisation Problems

A COP can be stated in three different ways [20]. The TSP defined earlier is an example of a COP – its the *optimisation version* of the combinatorial optimisation problem. Generally speaking, in the *optimisation version* we want to find the best solution, i.e., the solution with the best objective value – the lowest value when considering minimisation. If we only want to know the optimal objective value, we call this the *evaluation version* of the problem. Finally, the *recognition* or *decision version* of a problem consists in asking if there is a solution with an objective value lower than some threshold b. The solution to this version is a *yes* or *no* answer. It is possible to convert an optimisation problem into a decision problem by stating a bound on the objective being optimised.

Assuming that the objective function is easy to compute, the decision version is not more challenging to solve than the evaluation version, which is not more challenging than the

optimisation version. If we find an optimal solution for the optimisation version, we can use its objective value to answer the evaluation version and, for any threshold b on the objective value greater than the objective value of the optimal solution found, we can answer the recognition version.

2.1.4 On the Complexity of COPs

The following complexity classes [20] are originally specified only for decision problems. However, as we saw in sub-section 2.1.3, the decision version of a problem is no harder than an optimisation version of that problem. Stated in another way, the optimisation version is at least as hard as the decision version. As a consequence, if the decision version is hard, the optimisation version will also be hard. This enables us to use decision problems to study the difficulty of optimisation problems.

P The first class that we present is the P, which stands for *polynomial* time. Decision problems that belong to this class can be solved by an algorithm in $O(n^k)$ – i.e. polynomial time – where n is the size of the instance and $k \in \mathbb{R}$ is a constant. Intuitively, for an instance of a problem in P we can quickly determine if the answer is *yes* or *no*.

NP Another class is *NP*, which stands for *non-deterministic polynomial* time. Consider an instance of a decision problem with answer *yes* and a certificate of that – i.e., the solution that gives the answer *yes* to the problem. If the certificate can be checked by an algorithm in time polynomially bounded by the size of this solution, then the problem belongs to the *NP* class. As such, $P \subset NP$. Whether $P = NP$, remains one of the most important unsolved problems.

The difference between the classes P and *NP*, is that decision problems in P can be *solved* in polynomial-time; decision problems in *NP* have solutions that can be *verified* in polynomial time.

***NP-Complete* and *NP-Hard* Problems** Suppose that we have a problem $A \in NP$. If any instance of any other problem $B \in NP$ can be transformed, in polynomial time, into A, then A is at least as hard as any other problem in NP and belongs to the class of NP-Complete problems. This means that whatever answer we get in A, we can use that answer to get the answer to B. However, if we do not know whether $A \in NP$ or know that it is not, the rest being equal, we say that A is NP-Hard.

NP-Complete and NP-Hard decision problems are considered intractable problems, in the sense that there are no known polynomial time algorithms to solve all of their instances. For the case of NP-Hard problems there are no known algorithms that can even verify their solutions in polynomial time.

As stated before, this "dificulty" also applies to the optimisation version of these problems, which are the problems that we deal with in this thesis. We note, however, that if the decision version of a problem is *NP-Complete*, then its optimisation version is *NP-Hard*.

As such, for the respective optimisation version of NP-Complete and NP-Hard decision problems, we need to be less ambitious and aim to at least have a good approximation of the solutions to these problems. This is the underlying motivation for the approaches discussed in the next section and for the work presented in this thesis.

2.2 Combinatorial Optimisation Algorithms

Another way to look at optimisation problems is as search problems, where we are searching for the best solution in the solution space. Due to the intractability of NP-Complete and NP-Hard problems, this is the main way to get good solutions for arbitrary instances of these problems[2]. However, the solution space of these problems grows at least exponentially with the size of the instance.

For example, the optimisation version of the TSP, informally stated earlier, can be interpreted as "find an ordering of the cities, starting on a city, such that passing through all the other cities in that order and returning to the first one, has the lowest possible cost". This can be represented as a permutation of the cities. Its search space is then all possible permutations of these cities. For an instance with n cities, we have a solution space of $n!$. The decision version of the TSP is NP-Complete, since given an *yes* instance of the problem there exists a solution (certificate) of that answer, where it is possible to verify if each city is present exactly once and calculate the total cost in $O(n^k)$. This means that its optimisation version is *NP-Hard*.

Searching such large solution spaces systematically or randomly is usually not a good strategy. As such, we need "smarter" search strategies. We briefly present some of these strategies.

2.2.1 Local Search Algorithms

Local Search (LS) [14] strategies are present in well known algorithms, such as Lin-Kernighan-Helsgaun (LKH) [12], for the TSP.

These approaches explore the search space of a particular instance of a combinatorial optimisation problem, following some rules local to the current solution, in order to move in the solution's neighbourhood.

In general, LS begins the search at some initial solution. Then, until some stopping criteria is achieved, moves iteratively through the neighbourhood of the current solution, until a local optimum solution is achieved.

The definition of a neighbourhood allows for the interpretation of the solution space as a graph, which reflects some of its proprieties. For example, if the graph is symmetric, the neighbourhood is also symmetric. This means that the LS algorithm will be able to reverse search steps, returning to the previous solution. Another relevant property is that the degree of a vertex corresponding to a given solution in the search graph is equal to the size of its neighbourhood. In some cases, all the vertices have the same degree.

Iterative Improvement

One particular type of LS, that is the basis for other LS algorithms that we will present, such as iterated local search (ILS), is Iterative Improvement (II)[3] [14].

II begins by randomly selecting a solution from the search space and tries to improve this solution iteratively, searching for better neighbours. The search stops when the neighbour-

[2]For some of these problems, there may exist specific classes of instances that can be solved exactly in polynomial time.

[3]This approach is also called *iterative descent* or *hill-climbing* (for maximisation).

hood of a solution is completely explored without finding a better solution.

This stopping criterion presents a clear disadvantage, because once a local optimum is found there is no way of escaping this solution and, as such, it is not possible to find better local optima. However, there are approaches to overcome this problem, and Randomised Iterative Improvement (RII) [14] is one of them.

In the RII, instead of accepting only improving neighbours, at each search step it chooses whether to take the step *only if the neighbour is better* or to take the step *to a random neighbour*. This makes it possible to accept worse solutions, preventing the search from becoming stuck at a local optimum.

Another approach is Probabilistic Iterative Improvement (PII) [14]. In PII, if no solution in the neighbourhood is better than the current solution, then it "tries" to accept the least bad neighbour, where the probability of accepting a worse solution is dependent of how bad it is when compared to the current solution. As such, the worse a neighbour is, the less likely it is to be selected. The accepting function [13] is given by:

$$p_{accept}(T, s, s') = \begin{cases} 1 & \text{if } f(s') \leq f(s) \\ exp(\frac{f(s)-f(s')}{T}) & \text{otherwise} \end{cases} \tag{2.3}$$

In the equation 2.3, s is the current solution, s' is a neighbour of s, and T is a constant.

Turning T into a variable, yields another approach called *Simulated Annealing*, where T refers to the temperature (see [14] for details).

Iterated Local Search

Iterated local search (ILS) [14] can also be seen as an extension of II. It is an hybrid approach, that combines two local searches: one to find good solutions efficiently and another to escape from local optima.

ILS has three main components: a *local search* function, a *perturbation* function, and an *accept* function.

As usual in LS, ILS begins by randomly choosing a solution from the search space. From this solution, a local search is performed until a local optimum is reached – this is the first candidate solution. Then, with this candidate solution, the algorithm enters a cycle where it performs the following steps, until a termination condition is reached:

1. A *perturbation* is applied to the candidate solution to escape the local optimum as far as needed.

2. A *local search* is performed starting at the new, perturbed solution, until another local optimum is found;

3. The best of the two local optima found in the present and the previous LS steps is *accepted*, from which the search continues with the perturbation step.

Regarding these three main components, the following should be considered. The size of the *perturbation* should be large enough to generate a solution away from the local optimum, so that the probability of returning to this local optimum is as low as possible, but not too far! This perturbation could be, for example, a random walk or a fixed sequence of simple

search steps on the same or on a different neighbourhood structure. The *local search* step, could be the same as the II, although more sophisticated methods can improve the ILS performance.

Until now, we only refered to LS that is performed on a single solution at each iteration. In the following, we present a class of algorithms that work on a set, or population, of solutions.

2.2.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) [7] are a class of iterative population-based approaches inspired in the natural evolution of biological species, whose goal is to solve hard combinatorial optimisation problems by applying the main principles of natural evolution: mutation, recombination, and selection. Additionally, the objective function – also called fitness or evaluation function – guides the search.

In general, an EA [7] starts with a population of random candidate solutions. This population is then evaluated to assess its quality, by evaluating each individual in it. The lower the objective value of a solution, the better[4].

Then, until a termination condition is satisfied, an EA works as follows:

1. *select* parents (candidate solutions) from the population, where the ones with the best objective value are more likely to be selected;
2. pairs of these parents are then *recombined*, to generate a new solution (offspring);
3. a *mutation* operator is applied to the offspring with some probability, to introduce some variation in the population;
4. the new population is created by replacing completely or partially the current population by the new offspring.

The recombination (or crossover)[5] operator merges some information from both parents, such that a new valid solution is generated (the offspring). The parts of each parent that are recombined, determined by the crossover points, are (usually) chosen randomly. This is not the case for Partition Crossovers (PXs), as we will see.

The goal of the mutation operator is to introduce variability in the population, so that the EA is able to explore other locations of the search space. This prevents the fast convergence of the population towards a particular solution sub-space. More concretely, we can think of mutation applied to a particular solution as performing a single search step in this solution's neighbourhood implicitly defined by that particular mutation operator and replacing this solution with one of its neighbours.

In the context of EAs, a candidate solution has several characteristics. In this work, the most relevant is the *allele*. Consider a candidate solution represented by a sequence of n numbers. In this candidate solution, an allele corresponds to the value in a given position. As another example, consider an instance of the TSP with n cities. If we represent each candidate solution (tour) as a graph with n edges, then an allele corresponds to i^{th} edge in this solution.

[4] If we are maximising, the best solutions have higher objective value.

[5] In this work, we will use the terms "recombination" and "crossover" interchangeably.

Having introduced the basic workings of a general EA, this work will focus on a particular type of crossovers. Next, we give a brief presentation of these crossovers and of the geometric interpretation of crossovers.

2.3 Crossover Operators

Most recombination operators are stochastic in the choice of the crossover points. Here we are interested in a type of crossovers that are deterministic in this choice, called Partition Crossovers.

In what follows we show the main motivation for the development of these operators and briefly introduce them.

2.3.1 Optimal Recombination Problem

One of the major challenges in the design of crossovers is producing an operator that deterministically generates the best possible offspring in a set of possible offspring. This is called the Optimal Recombination Problem (ORP) [8, 9]. For the TSP, one version of this problem is known to be NP-hard [9], since it is the same as solving a *regular* TSP on the graph of the union of two parent tours. We refer to this as the Optimal Tour Merging (OTM) problem. The OTM problem consists in finding the best possible solution made up of edges from the parents. Solving the OTM problem may also be referred to as, Optimal Edge Preserving Recombination (OEPR). OEPR does not introduce in the offspring edges that are not present in either parent. As such, we say that this it "transmits alleles".

However, another property that many recombination operators have is "respect" [22], i.e., edges that are common to both parents *must* be in the offspring. Optimal Respectful Edge Preserving Recombination (OREPR), a variant of OEPR, is both "respectfull" and "transmits alleles". That is, OREPR performs optimal recombination in the union of two parent tours, such that the resulting offspring must contain all the common edges between the parents.

Both OEPR and OREPR are known to be NP-hard [9], and thus impractical for large instances. For this reason, more efficient operators were developed that try to approximate the OREPR problem. Partition crossovers, the main theme of this thesis, are one of them.

2.3.2 Partition Crossovers

PXs [27, 28] are deterministic recombination operators, in the sense that they exploit the natural decompositions of the parents, deterministically, partitioning them into components. As such, for a given pair of parents, they always produce the same offspring. The components emerge through the "removal" of common, partial solutions between parents, and each resulting component has different, interchangeable partial solutions from each parent. The offspring is constructed by selecting the best partial solution in each component and adding to it the common partial solutions. As such, these operators are "respectfull" and "transmit alleles". For example, if we consider two parent solutions from an instance of the TSP, a component emerges through the "removal" of common sub-tours. Each component has two interchangeable sub-tours, one from each parent, both sub-tours start and end in the same city and traverse the same inner cities in different order. The

offspring is constructed by choosing the best sub-tour in each component and adding the common sub-tours.

The partitioning of solutions is particularly useful when the objective function can be locally decomposed as a sum of sub-functions, with respect to the parent decompositions. Continuing with the example of the TSP, an objective function that consists of going through the cities of the tour in the order presented, summing the distance between these cities, could be expressed as a sum of several sub-functions where each sub-function corresponds to a component.

The properties of "respect" and "allele transmition", in combination with the local separability of the objective function, enables PXs to return the best possible offspring given the partitioning [26, 27, 28, 29]. Often, if the parents are local optima with respect to the neighbourhood defined by a local search operator, the resulting offspring is also local optimum in the same neighbourhood [27]. This suggests that, by recombining two local optima, PXs are often able to move to another local optimum.

In the TSP, it was observed that high quality local optima tend to be closer to each other and surrounding a global optima, in that there is a direct relation between the distance of a solution to the global optimum and that the closer this solution is to the this global optimum, the closer their objective values are. The same direct relation exists from this solution to other local optima. This suggested that there is a big, central valley in the search space at the bottom of which is the global optimum. This is called the *Big Valley Hypothesis* [3]. However, it was found [11] that several heuristics become stuck at these local optima, in the Big Valley. This is due to the fact that these local optima are located in "funnels", that break down the valley as the algorithm approaches its bottom. The ability of an algorithm with PXs to move between local optima, when the parents are local optima, means that these "funnels" are smoothed out.

Local optima are characterised with respect to a neighbourhood structure defined by some LS operator. Thus, to be able to characterise the offspring generated by PXs, we need to know if they operate on the same neighbourhood structure. In order to do that, we introduce some concepts of *Geometric Crossovers* in the following.

2.3.3 Geometric Crossovers

Consider a solution space S connected through its underlying graph that results from the definition of the neighbourhood. Consider also that this neighbourhood is symmetric and, as such, this underlying graph is undirected[6]. Such a space is also called a *metric space* [18].

Now consider a distance function, on this metric space, $d(s_1, s_2)$ between solutions $s_1, s_2 \in S$, where $d : S \times S \to \mathbb{R}$ defined according to the following axioms[18]:

- $d(s_1, s_2) \geq 0$;
- if $s_1 = s_2$ then, $d(s_1, s_2) = 0$;
- $d(s_1, s_2) = d(s_2, s_1)$ (symmetry)
- $d(s_1, s_3) \leq d(s_1, s_2) + d(s_2, s_3)$ (triangle inequality).

[6] See sub-sections 2.1.2 and 2.2.1.

A metric space is defined by (S, d).

Intuitively, we can think of the distance $d(s_1, s_2)$ as the minimum number of moves (or search steps) we need apply from s_1 to reach s_2. One example, is the 2-*opt distance*.

Definition 2.2 (2-opt distance)**.** Given two solutions s_1 and s_2, the 2-opt distance between s_1 and s_2 corresponds to the *minimum* number of 2-opt moves required to turn s_1 into s_2.

Definition 2.3 (Closed ball)**.** A *closed ball* in a metric space (S, d) is defined as $B(s; r) = \{s_1 \in S : d(s, s_1) \leq r \ \wedge s_1 \in S\}$, where $r \in \mathbb{R}_{>0}$ is the radius of the ball.

As such, making $r = 1$, a neighbourhood of a solution $s \in S$ can be defined as the closed ball $B(s; 1) = \{s_1 \in S : d(s, s_1) \leq 1 \ \wedge s_1 \in S\}$. We note that a mutation operator does a search in the ball $B(s; r)$, through the application of a move, once, on s.

Definition 2.4 (Line segment)**.** A *line segment* (also a closed interval) between two extremes $s_1, s_2 \in S$ is $[s_1, s_2] = \{s \in S : d(s_1, s)+d(s, s_2) = d(s_1, s_2)\}$, where $[s_1, s_2] = [s_2, s_1]$ and $l([s_1, s_2]) = d(s_1, s_2)$ is the length of the segment. A segment has exactly two extremes.

We can think of a segment between two solutions $s_1, s_2 \in S$ (*extremes*) as a sub-set (and a sub-graph) of S, with all the possible *paths* of length $d(s_1, s_2)$.

A *geometric crossover* under some distance d can be defined informally as a crossover that, given two parents p_1 and p_2, produces an offspring c that lies on a path in the segment between the two parents ($c \in [p_1; p_2]$), and thus $d(p_1, c) + d(c, p_2) = d(p_1, p_2)$.

It remains to be known if PXs are geometric crossovers under the same distance defined by the neighbourhood structured of the LS operators they are usually combined with. In this thesis, we address some of the geometric properties of PXs, but before that, in the next chapter we present the state of the art of these operators.

Chapter 3

Partition Crossovers

This chapter presents the latest work in partition crossovers. In particular, we present four different partition crossovers. The difference between them is how they work, concerning the representation of the solutions that they recombine and the problem of application. For example, Iterative Partial Transcription (IPT), Generalised Partition Crossover (GPX), and its variants work on the same problem, but on different solution representations. However, their underlying characteristics are the same.

In the following sections, partition crossovers for three combinatorial optimisation problems are presented: TSP, Pseudo-Boolean Optimization (PBO) problems, and Quadratic Assignment Problem (QAP).

3.1 Iterative Partial Transcription

The Iterative Partial Transcription (IPT) [17] was proposed with the aim of improving the performance of local search algorithms for the TSP.

Definition 3.1 (Travelling Salesman Problem)**.** Consider a complete graph $G(V, E)$, where $V = \{v_1, v_2, ..., v_n\}$, is the vertex set, and for every pair of vertices $v_i, v_j \in V$ there is an edge $e_{i,j} \in E$ with an associated cost $c_{i,j} \in \mathbb{R}_{>0}$ [27]. A feasible solution to the TSP is an Hamiltonian cycle on G denoted by $\vec{s} = [s_1, s_2, ..., s_n]$ in the set of feasible solutions S, with s_1 the starting and ending vertex. The TSP consists in finding a feasible solution $s \in S$ that minimises the function,

$$f(\vec{s}) = c_{s_n,s_1} + \sum_{i=1}^{n-1} c_{s_i,s_{i+1}} \tag{3.1}$$

Using partition crossover terminology, given two parent solutions, IPT works as follows ([24]):

1. the cities connected through common edges in both parents are removed;
2. recombining components are searched for – these are sub-sequences of vertices composed of the same vertices in both parents, with the same initial and final vertices;
3. offspring are created by selecting the best sub-sequence in each recombining component and adding the vertices connected by common edges.

The steps above describe, in general, how partition crossovers work: they search for common alleles (edges) in both parents and pass them to the offspring; then, they search for recombining components, and used them to complete the offspring and minimise or maximise its objective value. This similarity in operation classifies the IPT as a partition crossover [24].

In [17], the authors combined IPT with a local search procedure and called this combination IPTLS. The IPTLS applies IPT to local optima generated by a multi-start-local-search and, if the result is different from the parents, applies a local search to it, since there is no guarantee that it is locally optimal.

To give an example of an application of IPTLS from [17], and hence IPT, we briefly describe the simple version of IPTLS on a multi-start-local-search for the TSP. This application consisted of executing a local search procedure several times (trials), where each local search started from a random solution and ended in a local optimum. IPTLS recombines this local optimum with the last recombination result, except in the first trial. The local optimum achieved in the first trial is the starting point for the IPTLS results. It then uses this first result in the recombination with the second trial's local optimum. The process continues, recombining each local optimum with the last result of recombination. The same paper proposes other approaches, such as trying to improve on an archive of solutions – a parallel multi-trial-local-search –, as well as applying IPT in the Thermal-cycling approach (see [17]). Moreover, this operator is also applied in a version of the LKH algorithm, the LKH-2 [12].

IPT has a time complexity of $O(n^2)$ [27]. As it will be seen in the next section, this performance can be improved.

3.2 Partition Crossovers for the TSP

Partition Crossovers (PXs) [28] work on the graph representation of the TSP, and its process is similar to that of the IPT.

To facilitate the explanation of PX, Figure 3.1a depicts the graphs of two parent solutions and the Figure 3.1b shows the superposition of those solutions, resulting in the union graph G.

In general, the vertices of this graph have degree two, three, or four (not shown in the example). Those vertices that have degree two belong to a sub-tour common to both parents. As such, their incident edges are also common in both parents. If this sub-tour has two or more edges, PX replaces the common edges in the common sub-tour in both parents by surrogate edges (dotted lines), creating G' (Fig. 3.1c).

For the example in Figure 3.1, it is possible to separate G' by deleting the surrogate edges, revealing two components in G'. A partition of G' is said to have cost 2 if it is possible to partition the vertices in two non-empty (components) sets by removing precisely two (surrogate) edges. If there is no partition of cost 2, PX cannot be applied. In the example given, these components are recombining components – feasible for recombination – since they start and end in the same vertices in both parents and the associated sub-tours traverse the same vertices on each parent in a different order. Finally, an offspring is created by choosing, for each recombining component, the best sub-tour from either parent, and adding to it the sub-tours common to both parents that were replaced by surrogate edges earlier. This ensures the properties of "respect and "allele transmission". However, PX has some weaknesses. As stated before, if there is no partition of cost two, recombination is

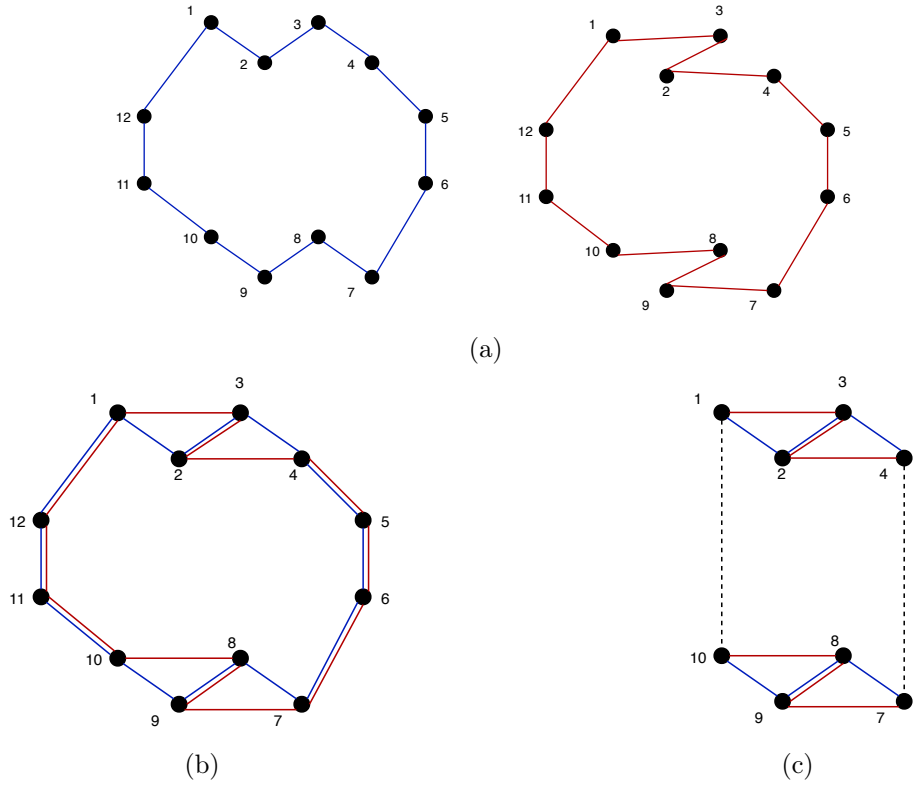


Figure 3.1: Recombination with PX. In a), we have two random parents (blue and red); b) shows the union of these parents; and in c) common paths are substituted with surrogate edges (dotted lines).

impossible. Additionally, it does not find all the possible components and can not create all possible offspring that obey the properties of "respect" and "allele transmition", only two of them can be created. If there were k components, then there would be $2^k - 2$ possible offspring [28], since in each component we can choose between two partial solutions.

A generalised version of PX, called GPX [11, 29], was later proposed. It is potentially able to identify more recombining components, and thus to produce more offspring. Improvements to GPX have been made since then in order to increase the number of components found, including a version of GPX for the asymmetric TSP, the Generalised Asymmetric Partition Crossover (GAPX). The most recent version is called Generalised Partition Crossover 2 (GPX2) [21, 23, 26, 27]. In addition to what GPX does, GPX2 [27] can find more components by splitting the vertices of degree 4 in G', considering components with more than one start and ending vertices, and fusing unfeasible components to hopefully create more components suitable for recombination, as explained in more detail bellow.

PX, GPX, GAPX, and GPX2 present an improvement to IPT. They have a time complexity of $O(n)$, instead of $O(n^2)$ as in the IPT. The GPX and later versions of PX are able to find more recombining components, and to produce more offspring. Moreover, if the parents are local optima, the offspring is also a local optimum with high probability [27]. As such, they have the ability to tunnel between local optima, providing the possibility to escape from them to reach a new, better one.

3.2.1 The Generalized Partition Crossover 2

As stated earlier, GPX2 distinguishes itself from the other PXs for the TSP by its ability to find more (recombining) components. As in the other PXs for the TSP, after generating the graph of the union of the parents, G, GPX2 removes the vertices of degree 2 and its incident edges. This creates the first *candidate* components. Then it splits all vertices of degree 4. For each vertex of degree 4, v, a ghost vertex v' is added immediately after v in both parents. This creates a new (common) edge (v, v') in both parents with weight 0. These new common edges are removed, creating additional candidate components. The vertex after which the ghost vertex is added is determined by the reading direction of the permutations of the parents. The reading direction must be such that, vertices that are next to each other in both parents, should be read in the same order. This might imply that one of the parents (permutation) must be read in the reverse direction. At this point, the initial candidate components are found. These candidate components are tested to determine whether they are recombining or non-recombining components[1].

Recombining components have an *even* number of portals (half entry, half exit). These portals are entry and exit vertices in the component that connect to other components through common edges. In a **recombining component**, both parents must enter and exit in the same portals, and traverse the same vertices (in different order) between these portals. These components are independent from all the others, such that any partial solution (parent) in it can be chosen without neither risking creating an invalid offspring, nor affecting the choices in other components. If these conditions do not apply, the candidate component is a **non-recombining component**. On their own, the sub-tours present in each of these components, if chosen, *may* cause the resulting offspring to be invalid.

After identifying the initial recombining and non-recombining components, two types of fusion, Type-1 and Type-2, are performed a fixed number of times in order to (hopefully) find more recombining components. In the implementation of GPX2 [23], it starts by performing Type-1 fusion three times followed by one execution of Type-2 fusion. In Type-1 fusion, a given non-recombining component is fused with at most two neighbouring non-recombining components. The Type-2 fusion, fuses non-recombining components that are nested and embedded inside other candidate components. Every time a fusion is completed, the new components are tested to see if new recombining components were created.

In the recombining components, all edges must be inherited from the same parent. As a consequence, non-recombining components that were fused together and originated a recombining component, must have the same chosen parent (sub-tour). This guarantees that the chosen sub-tour is always valid, without additional checking. However, as we will see in Chapter 6, there may exist valid, advantageous combinations of different parent sub-tours between fused non-recombining components, even when fusing them does not result in a recombining component. After all the fusions are done, the offspring can be created.

The remaining non-recombining components, that could not be made into recombining components despite the fusions, are joined in a component called "rest" that is necessarily a recombining component. The "rest" is independent of all the other components, because it contains all the components that could have a dependency with other components. Therefore, all the edges inside a recombining component must come from the same (chosen) parent. As such, there is no possibility of generating an invalid offspring from choices in the "rest".

[1] In [27], "non-recombining components" are called "infeasible components".

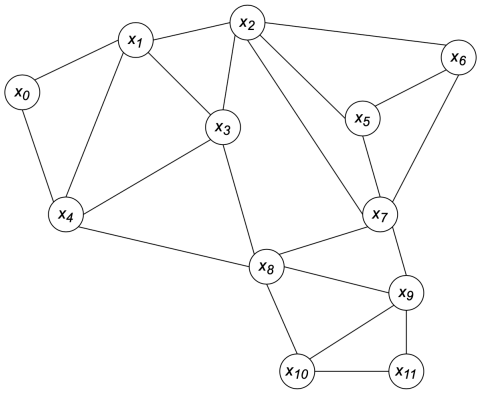


Figure 3.2: Example of a Variable Interaction Graph.

The GPX2 then goes through each recombining component (including the "rest") and chooses the best parent sub-tour in each case. The offspring is generated by adding to it the common edges along with the choices made in each recombining component.

Under certain conditions, partition crossovers can also be constructed for other combinatorial optimisation problems. The next sections present two such operators.

3.3 Partition Crossovers for pseudo-Boolean Optimisation Problems

The partition crossovers introduced so far are only suitable for the TSP. In this section, we present partition crossovers for a class of Pseudo-Boolean Optimization (PBO) problems where the pseudo-Boolean function is k-bounded.

Definition 3.2 (Pseudo-Boolean Optimisation Problem)**.** Optimise a pseudo-Boolean function $f : \mathbb{B}^n \to \mathbb{R}$, where $\mathbb{B} = \{0, 1\}$. If it is possible to express f as a sum of m sub-functions, where each sub-function depends on at most k of the n variables, we say that f is k-bounded.

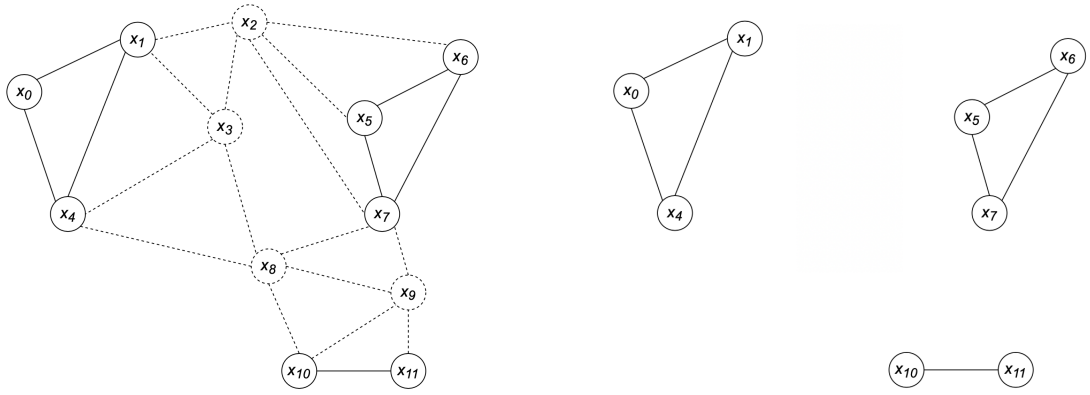
These partition crossovers [4, 5, 25] explore the Variable Interaction Graph (VIG) of the objective function, that represents the interactions among its variables.

Definition 3.3 (Variable Interaction Graph)**.** A VIG is a graph such that each vertex corresponds to a variable, and each edge represents a nonlinear interaction between the variables corresponding to the associated vertices.

Partition Crossovers for Pseudo-Boolean Optimisation (PXPBO) extract the VIG from the decomposition of the objective function into m sub-functions of at most k variables. Two variables have a nonlinear interaction if they are arguments of the same sub-function; alternatively, they have a nonlinear interaction if we take the Fourier transform of the objective function, and the term's coefficient with these two variables is not zero [5].

For example, suppose that the objective function can be decomposed in the following sub-functions, with $k = 3$:

$$
\begin{array}{lll}
f_0(x_0, x_4, x_1) & f_4(x_2, x_5, x_7) & f_8(x_8, x_{10}, x_9) \\
f_1(x_1, x_4, x_3) & f_5(x_2, x_6, x_5) & f_9(x_{10}, x_9, x_{11}) \\
f_2(x_2, x_3, x_1) & f_6(x_5, x_7, x_6) & \\
f_3(x_4, x_3, x_8) & f_7(x_7, x_8, x_9) &
\end{array}
$$



(a) (b)

Figure 3.3: Illustration of the recombining graph from the VIG. In (a), the dashed vertices and nodes will be deleted, because they correspond to variables with the same values in both parents. This originates the Recombination Graph in (b).

The arguments of each sub-function indicate that there are nonlinear interactions among those variables. From that, we construct the VIG in the Figure 3.2, that represents these nonlinear interactions.

PXPBOs split the VIG by eliminating the nodes, along with their incident edges, corresponding to variables whose value is the same in both parents – as it was for the common edges in the TSP –, resulting in the *recombination graph*. The recombination graph is composed of sub-graphs (or components), disconnected from one another. Each of these sub-graphs corresponds to a new sub-function, where nodes represent the variables and edges represent the nonlinear interaction between those variables, just as in the VIG. For example, the dashed edges and nodes in the Figure 3.3 are deleted, because the variables that the nodes represent have the same value in both parents, originating the Recombination Graph in the Figure 3.3b. Finally, PXPBO creates the offspring by going through each component in the Recombination Graph and selecting the best partial solution from one of the parents [25].

Articulation Point Partition Crossover (APX) [4], improves the crossover presented before by exploring articulation points in the recombination graph, i.e., on its sub-graphs, that correspond to variables that, once flipped in one of the parents, further split the recombination graph, thus finding more components (and more sub-functions) for recombination. In other words, it creates more common values between parents. To decide whether a variable corresponding to an articulation point and, therefore, the component should be split, APX evaluates the increase in contribution to the objective function of doing so and then applying the partition crossover.

Finally, Dynastic Potential Partition Crossover (DPX) [5] returns the best possible offspring from the recombination of two parents by exploring possible combinations of the values from either of the parents in each component where the number of nonlinear interactions between variables of the objective function is low, using dynamic programming.

The recombination operators introduced in this section, have the same properties of "respectfulness" and "allele transmission", as the other partitions crossovers introduced in earlier sections.

Next, the DPX operator is explained in more detail, due to its relevance for the work presented ahead.

3.3.1 Dynastic Potential Crossover

The Figure 3.4, shows the pseudo-code of the DPX [5].

Algorithm 1 Pseudocode of DPX

Input: two parents x and y
Output: one offspring z
1: Compute the Recombination Graph of x and y as in [6]
2: Apply Maximum Cardinality Search to the Recombination Graph [12]
3: Apply the fill-in procedure to make the graph chordal [12]
4: Apply the Clique Tree construction procedure [13]
5: Assign subfunctions to cliques in the clique tree
6: Apply Dynamic Programming to find the offspring (see Algorithm 2)
7: Build z using the tables filled by Dynamic Programming

Figure 3.4: The pseudo-code of the DPX [5].

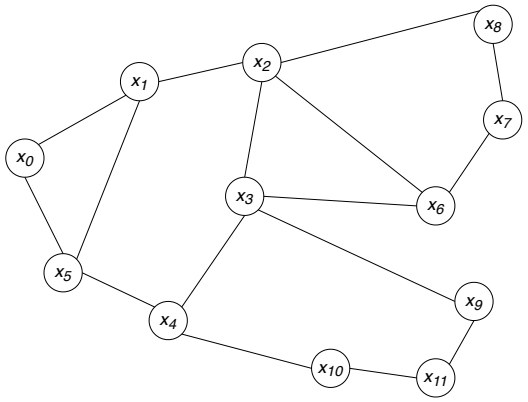


Figure 3.5: The Variable Interaction Graph of the example provided.

Consider a PBO problem instance, with $k = 3$, represented by the following sub-functions and corresponding VIG in Figure 3.5:

$$\begin{array}{lll} f_0(x_0, x_1, x_5) & f_5(x_5, x_4, x_0) & f_{10}(x_{10}, x_4, x_{11}) \\ f_1(x_1, x_2, x_5) & f_6(x_6, x_3, x_7) & f_{11}(x_{11}, x_{10}, x_9) \\ f_2(x_2, x_1, x_8) & f_7(x_7, x_8, x_6) & \\ f_3(x_3, x_6, x_9) & f_8(x_8, x_7, x_2) & \\ f_4(x_4, x_5, x_{10}) & f_9(x_9, x_3, x_{11}) & \end{array}$$

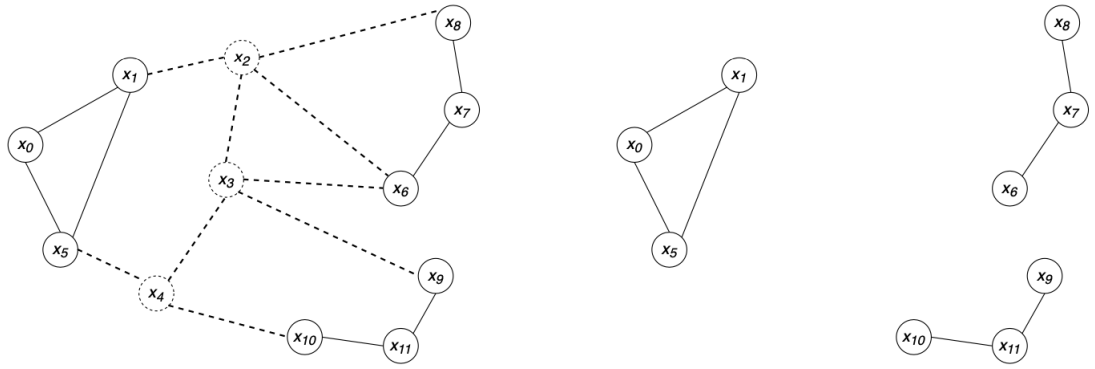
The nodes in the VIG (Figure 3.6a) representing the variables whose value is equal in both parents are then eliminated along with incident edges, resulting in the *recombination graph*, shown in the Figure 3.6b.

This *recombination graph* yields a new set of sub-functions as follows:

$$g_0(x_0, x_1, x_5) \quad g_1(x_7, x_8, x_6) \quad g_2(x_{11}, x_{10}, x_9)$$

This corresponds to the first step of the algorithm in Figure 3.4.

The resulting components (sub-graphs) of the *recombination graph* are the ones actually used in the recombination process. This process is performed in step 6 of the algorithm of Figure 3.4. In this step, for each of the components, the algorithm explores all possible combinations of values (of each parent), selecting the best one. This exploration is carried out by eliminating variables in a specific order using dynamic programming.



(a) (b)

Figure 3.6: Obtaining the recombination graph from the VIG. In (a), the dashed vertices and nodes will be deleted, because they correspond to variables with the same values in both parents. This originates the Recombination Graph in (b).

A prerequisite for this elimination to work is to determine an order of elimination. This is done in steps 2 through 4, resulting in a clique tree of a particular component – a tree where its nodes are cliques.

Then, in step 5, the new sub-functions are assigned to one, and only one, clique if and only if the variables of this clique are a super-set of the variables of the sub-function assigned to it.

Considering two cliques in the clique tree, C_i and C_j, where C_i is the parent (precedent) of C_j ($i > j$). We call the *separator* of C_j the set of variables that are both in C_i and C_j; and we call the *residue* of C_j the set of variables that are in C_j but not in C_i. The separator of the clique in the root of the tree is the empty set.

The algorithm of step 6 is shown in Figure 3.7. It traverses the clique tree in post-order, applying dynamic programming in each clique (node) and using the corresponding assigned sub-functions to evaluate each combination of parents.

In general, the algorithm in Figure 3.7 works as follows: for each clique in the clique tree, traversed in post-order, consider *all* combinations of parents for the variables in its residue, for each combination of parents for variables in its separator, and save the best combinations (in array `variable` in the Figure) and its corresponding objective value (in array `value` in the Figure 3.7). The final step is to use the combinations stored to reconstruct the offspring.

Avoiding Exponential Time

The PX presented here performs optimal recombination, however, it runs in exponential time in the worst case, since it tests all possible combinations. This can be impractical. To avoid the exponential factor, it was proposed in [5] to consider a constant β that defines a limit on the number of variables exhaustively explored in both the residue and the separator of each clique. The remaining variables are treated as one variable and jointly take the values in either parent. However, this constant turns DPX into a *quasi-optimal* recombination operator.

In the next section, we present yet another partition crossover that works on bipartite graphs.

Algorithm 2 Optimal Offspring Computation

```
1: for all cliques C_i of the clique tree in post-order do
2:    for x_{S_i} ∈ {0,1}^{|S_i|} do
3:       value[x_{S_i}] = −∞
4:       for x_{R_i} ∈ {0,1}^{|R_i|} do
5:          aux = 0
6:          for f ∈ F_{C_i} do
7:             aux = aux + f(x)
8:          end for
9:          for children cliques C' of C_i do
10:            aux = aux + value[x_{C'}];
11:         end for
12:         if aux > value[x_{S_i}] then
13:            value[x_{S_i}] = aux
14:            variable[x_{S_i}] = x_{R_i}
15:         end if
16:      end for
17:   end for
18: end for
```

Figure 3.7: The dynamic programming algorithm that computes the optimal offspring [5].

3.4 A Partition Crossover for the QAP

More recently, a Partition Crossover for the Quadratic Assignment Problem (PXQAP) was proposed [2]. It is based on a bipartite graph representation of the solutions.

Definition 3.4 (Quadratic Assignment Problem)**.** Considering a set of n plants, $P \subset \mathbb{Z}_{\geq 0}$, and a set of n locations, $L \subset \mathbb{Z}_{\geq 0}$, where there is a flow $f(p,q) \in \mathbb{R}_{\geq 0}$ between plants $p, q \in P$ and a distance $d(k,l) \in \mathbb{R}_{\geq 0}$ between locations $k, l \in L$, a solution can be represented by a permutation $\pi = (1, \ldots, n) \in \Pi$, the set of permutations of n elements. The goal of the QAP [2, 15] is to find a permutation $\pi \in \Pi$ that assigns each plant i to a location $\pi(i)$ that minimises the function:

$$f(\pi) = \sum_{p=1}^{n} \sum_{q=1}^{n} d(\pi(p), \pi(q)) f(p,q) \tag{3.2}$$

Although at first permutations represent solutions for the QAP, they have to be converted into a bipartite graph for recombination to be possible with the PXQAP.

Definition 3.5 (Bipartite graph)**.** A bipartite graph $G = (U, V, E)$ has two disjoint sets of vertices U and V, and an edge set E, such that each edge connects a vertex in U to one in V.

Given a permutation $\pi \in \Pi$ that represents a given solution, the corresponding bipartite graph is $G_\pi = (P, L, E)$, where each edge $(i, \pi(i)) \in E \subset P \times L$ corresponds to the assignment of plant $i \in P$ to location $\pi(i) \in L$.

Figure 3.8 shows an example of two solutions in permutation form, π_1 and π_2. Each of these solutions has a corresponding bipartite graph G_{π_1} and G_{π_2}. The graph $G_{\pi_1 \pi_2}$ is the union of these graphs, whose edges are the union of the set of edges in G_{π_1} and G_{π_2}. In the Figure 3.9, we show $G_{\pi_1 \pi_2}$ for the example permutations presented in Figure 3.8. PXQAP uses this last graph to perform the crossover.

π_1	1	2	0	3	4	5	8	6	7
π_2	2	1	0	5	4	3	6	8	7

Figure 3.8: Example of the permutation representation of two solutions.

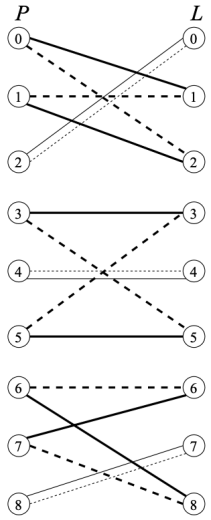


Figure 3.9: Bipartite Graph representation of the permutations in the Figure 3.8. The solid edges belong to parent π_1, and the dashed edges to π_2. Thinner edges are common edges in both parents.

In this graph, plants with the same location assigned in both parents are the common alleles, and the offspring inherits them "directly". Each of these plants connects to exactly one location, a common edge. The remaining edges and vertices (plants and locations) compose the recombining components.

Each recombining component is a bipartite graph $C_i = (U_i, V_i, E_i)$ composed of edges belonging alternately to each parent, forming a cycle. For example, in figure 3.9 there are three ($i \in \{1, 2, 3\}$) recombining components with the corresponding edge sets:

- $E_1 = \{(0,1), (1,1), (1,2), (0,2)\}$,
- $E_2 = \{(3,3), (5,3), (5,5), (3,5)\}$,
- $E_3 = \{(6,6), (7,6), (7,8), (6,8)\}$.

PXQAP constructs the offspring by preserving the parents' common assignments and, for all the plants of each recombining component choosing the best assigned locations of either parent. In the example of figure 3.9, one possible offspring might have edges from π_1 in C_1, π_2 in C_2, π_1 in C_3, and the common edges to both π_1 and π_2.

As the previous partitions crossovers, the PXQAP also has the properties of "respectfulness" and "allele transmission".

Having introduced the state of the art of PXs, next we present some of their properties with regards to geometric recombination.

Chapter 4

The Geometricity of Partition Crossovers

Having introduced Geometric Crossovers in Chapter 2, in this chapter we show some properties of PXs in the light of Geometric Recombination.

LS operators, by virtue of their rules, define the neighbourhood structure on which the search is performed. From the definition of this neighbourhood, a distance metric emerges, where the distance between two solutions corresponds to the minimum number of LS steps required to reach one of the solutions, starting from the other (see Sub-section 2.3.3).

PXs are usually used in a combination with some kind of local search operator. Proving they are geometric according with this same distance would show how they operate on the neighbourhood structure induced by such an LS operator. Such *unification* of these search operators would reveal other properties about PXs that are common to all geometric operators [18].

4.1 Necessary Conditions for Geometric Recombination

Consider two candidate solutions s_1 and s_2 of an instance of an optimisation problem and a distance d defined as the *minimum* number of movements that need to be applied to s_1 to generate s_2, such that $d(s_1, s_2) = d(s_2, s_1)$. A PX is geometric if, for all of its possible offspring o between two parents p_1 and p_2, $d(p_1, p_2) = d(p_1, o)+d(o, p_2)$ – i. e., the offspring lie on a shortest path between the two parents.

Proving that a crossover is geometric requires that we find a distance such that the definitions of geometric crossovers hold. On the other hand, in order to prove that a crossover operator is *non-geometric*, it is required to prove that this crossover is not geometric *for any distance*. However, one way to prove non-geometricity of a crossover operator is based on three properties common to all geometric crossovers, known as the *inbreeding properties* of geometric crossovers [19]. These properties naturally arise from the axiomatic definition of geometric crossover and are distance and representation independent. Failing to observe one of this properties implies that the given crossover is not geometric under any distance, or *non-geometric*.

In the following, we show that "respectfullness" and "allele transmission" imply that PXs may be geometric under some distance measures.

Consider an arbitrary partition crossover px and two parents p_1 and p_2. The recombination of these parents under this crossover, $px(p_1, p_2)$, gives the set of all possible offspring o. We show that px has the inbreeding properties.

Lemma 4.1.1 (Property of Purity)**.** *If px is a partition crossover, then the recombination of one parent with itself only produces the parent.*

Proof. Given that px is respectfull, the common alleles between both parents are directly passed to the offspring. Since $p_2 = p_1$, we have $px(p_1, p_2) = px(p_1, p_1)$, with all alleles common to both parents and no partitions exist. As such, $px(p_1, p_1) = \{p_1\}$. □

We note that the *property of purity* is also implied by the "allele transmition" property, since this requires that the offspring only contains alleles present in either of the parents.

Lemma 4.1.2 (Property of Convergence)**.** *If px is a partition crossover, then the recombination of one parent p_1 with one offspring o of p_1 and p_2 cannot produce the other parent p_2, unless $o = p_2$.*

Proof. Given an offspring $o \in px(p_1, p_2)$, where p_1 and p_2 have $k \in \mathbb{Z}_{>0}$ components, if $o \neq p_2$ then, by the properties of respect and allele transmition, o has alleles taken from p_2 in only $0 \leq j < k$ components and from p_1 in the $i = k - j$ components.

Taking $px(p_1, o)$, the common alleles between p_1 and o are: the alleles common to p_1 and p_2 from $px(p_1, p_2)$ and the alleles from p_1 that were inherited by o and that are not present in p_2. Thus p_1 and o lead to j components.

Then $\forall c \in px(p_1, o)$, c has the common alleles and the alleles from either p_1 or o (that came from p_2) in each of the j partitions.

If, for an offspring $c \in px(p_1, o)$, the alleles are selected from p_1, for any of the j components $c \neq p_2$. If all of them come from o, then $c = o \neq p_2$.

Finally, $c = p_2$ if and only if $j = k$, meaning that o inherits all components from p_2 and c inherits the components from o in all of the j components. □

Lemma 4.1.3 (Property of Partition)**.** *If px is a partition crossover, then $\forall o \in px(p_1, p_2)$, $px(p_1, o)$ and $px(o, px_2)$ can only produce o as common child.*

Proof. Consider any offspring $o \in px(p_1, p_2)$, generated from k components between p_1 and p_2, such that $o \neq p_1 \wedge o \neq p_2$.

Then there exists a set of components P_1 with $|P_1| < k$ components between p_1 and o which are inherited from p_2 by o in $px(p_1, p_2)$; everything else is common to o and p_1 – the alleles inherited from p_1 by o and the common alleles between p_1 and p_2. Then, by the respect property every offspring $g_1 \in px(p_1, o)$ inherits those common alleles "directly". Additionally, by the allele transmission property, for each partition in P_1, g_1 is composed of partial solutions from either p_1 or o.

The case is the same for any offspring $g_2 \in px(o, p_2)$, but in this case the partition set P_2 has $|P_2| = k - |P_1|$ and $P_2 \cap P_1 = \emptyset$.

If, for each partition in P_1 for g_1 and partition P_2 for g_2, only partial solutions from o are chosen, then $g_1 = g_2 = o$.

Otherwise, since $P_2 \cap P_1 = \emptyset$ and the common alleles between p_1 and o *and* between p_2 and o are different, then $\forall g_1 \in px(p_1, o)$ and $\forall g_2 \in px(o, p_2)$, if $g_1 \neq o \vee g_2 \neq o$ then, $g_1 \neq g_2$. □

Using only the properties of "respect" and "allele transmission", we showed that *all* PXs have the inbreeding properties, common to all geometric crossovers. As a consequence, it was not possible to prove their *non-geometricity*. This means that, for every PX there may exist a distance according to which it is geometric.

Indeed, for the particular case of PXs for the TSP, such as GPX2, there is at least one distance, namely *bond distance*, where this is the case.

4.2 Proving the Geometricity of PXs

Here, we prove that PXs for the TSP are geometric under the bond distance [3].

Definition 4.1 (Bond distance)**.** Given two tours p_1 and p_2, the *bond distance* between those tours, $b(p_1, p_2)$, is defined as the total number of cities, n, minus the number of common edges between p_1 and p_2.

Theorem 4.2.1 (PXs for TSP are geometric under the bond distance)**.** *If px is a PX for the TSP, p_1 and p_2 are two parent tours of an instance with $n \in \mathbb{Z}_{>0}$ cities, and $b(p_1, p_2) \leq n$, then $\forall o \in px(p_1, p_2)$, $b(p_1, o) + b(o, p_2) = b(p_1, p_2)$.*

Proof. Suppose that $b(p_1, p_2) = k < n$, such that there are $m > 0$ common edges, where $m = n - k \Leftrightarrow k = n - m$.

Then, $b(p_1, o) \leq n - m = k$ (analogously, $b(o, p_2) \leq k$) by the "respect" property of PXs. Furthermore, considering the "allele trasmission" property, suppose that o takes e_1 edges from p_1 and e_2 edges from p_2, where $e_1 + e_2 = k$. As such $b(p_1, o) = n - e_1 - m$ and $b(o, p_2) = n - e_2 - m$.

Thus,

$$\begin{aligned} b(p_1, o) + b(o, p_2) &= n - e_1 - m + n - e_2 - m \\ &= 2n - 2m - e_1 - e_2 \\ &= 2(n - m) - (e_1 + e_2) \\ &= 2k - k = k. \end{aligned}$$

Therefore, $b(p_1, o) + b(o, p_2) = k = b(p_1, p_2)$. □

Most LS algorithms for the TSP that are combined with PXs (or any crossover), such as [10], have a neighbourhood structure defined by the 2-opt distance. Furthermore, we know that the 2-opt distance $d(p_1, p_2)$ (Definition 2.2), is related to the bond distance as $b(p_1, p_2)/2 \leq d(p_1, p_2) \leq b(p_1, p_2)$ [3]. However, this does not provide enough information to prove that PXs for this problem are geometric under the 2-opt distance. As future work, it would be useful to know if PXs for TSP are geometric under this distance.

Having addressed some theoretical properties of PXs, in the next chapter, we address some the practical aspects.

Chapter 5

An Application Programming Interface for Partition Crossovers

In this chapter, we propose an Application Programming Interface (API) for PXs to facilitate the development of these operators for various combinatorial optimisation problems.

After analysing the applications of PXs in the literature (see Chapter 3), we noticed that all of them follow a common general structure, whose steps are roughly as follows:

1. Merge (or equivalent operation) the alleles of the two parents (according with their representation);
2. Find the alleles common to both parents and identify the components;
3. Select in each component the best partial solution from either parent;
4. Construct the offspring according to the selections made in the last step.

Given this structure, we noticed that some steps are *dependent* on the problem of interest and others are *independent*. In the listing above, steps 1 and 2 are dependent on the problem, since deciding what a component is depends on the representation of the solutions and on what a valid solution is in that representation – which depends on the problem. It also depends on the structure of the objective function.

For example, considering the TSP, we can represent a solution as a graph, as in many other problems. However this graph must fulfil some conditions in order to represent a valid solution, namely every node must have a degree of exactly two. As such, a valid solution cannot be any graph.

On the other hand, step 3 does not necessarily depend on the problem, since it consists of a selection: either choose the partial solution from one parent (0) or from the other one (1). As long as information about the existing components (and any dependencies between them) and the effect of each possible selection can be computed, step 3 can be performed in the same way *independently* of the problem. In the proposed API, step 3 might correspond to DPX, presented in Subsection 3.3.1. From now on, we refer to it as "algorithm".

This suggests that the development of PXs can be organised as two separate aspects: specifying what components are and how they can be evaluated on the one hand, and deciding from which parent solution to inherit for each component on the other hand. In the following we define an API that reflects this view. Python sytax is used for the sake of simplicity.

5.1 API Definition

A class `PX` is defined such that `px = PX(s1,s2)` instantiates an object that represents the union of two parent solutions. This is instantiated by the algorithm, which then calls the methods as needed. The following `PX` methods must be implemented according to the problem:

- `k = px.components()`
  Returns the number of components, `k`.
- `sf = px.subfunctions()`
  Returns a list of tuples of values between 0 and $k-1$, that represent the indices of the components that are the arguments of each sub-function. The number of sub-functions is given by `m = len(sf)`.
- `y = px.evaluate(i, t)`
  Returns the value $y \in \mathbb{R}$ of the *ith* sub-function given the values of the input variables. `t` is a tuple of zeros and ones representing the values of those variables.
- `o = px.offspring(t)`
  Returns the offspring constructed from the choices for each component in `t`, which is a tuple of zeros and ones, of length `k`.

5.2 Description of the API

The API proposed above is composed of three main operations that need to be defined according to the problem:

1. The identification of the components of the partition of the parents and their non-linear relations, to be represented as tuples (`px.subfunctions()`).
2. An evaluation method, that evaluates a given set of component choices according to a given sub-function.
3. A method that constructs the offspring, given the choices made for each component.

Identification of Sub-Functions

What a component is, depends on the problem and how its solutions are represented. As such, a way to identify the candidate components and how they relate to each other must be provided.

This is the most important step, since identifying what the components are and how (and if) they depend on other components, will determine the "level of granularity" of the decisions made by the algorithm, which impacts the quality of the offspring generated and the overall performance of the crossover.

For example, in the case of GPX2, the recombining components are independent, as such the sub-functions have only one variable (component) and there are as much sub-functions as there are components. As a consequence, the decisions are made in linear time. For PBO problems, the sub-functions may correspond to the sub-functions of the recombination graph. In this case, the decisions are performed in the same way as in the DPX.

Evaluation Function

The evaluation function must give feedback to the algorithm on the quality and validity of its choices, considering that it produces real objective values. To indicate the quality of the choices, this function must be able to evaluate a partial solution represented by the set of dependent components and choices of parent for each component. Due to component dependencies, some choices may not be valid. If the choice being evaluated is valid, the respective objective value is returned. On the other hand, if the choice is invalid, a very large value can be returned. This causes those choices to be rejected.

Construction of the offspring

The construction of the offspring is the final step. Constructing the offspring consists in taking the parent choices for each component, "translating" this information to the context of the problem, and returning a new solution that can be evaluated and used by other search operators.

5.3 The GPX2 according with the API

In this section, we present three approaches to the implementation of the GPX2 (see Subsection 3.2.1) according with the proposed API, in ascending level of "granularity". This discussion is focused on how the components are found, how they can be represented as tuples (sub-functions), and how each decision can be evaluated.

5.3.1 First Level

The first approach we present is to consider the final recombining components used by the original GPX2, to make the recombination decisions. As such, an implementation of this approach would work as it is originally proposed for the GPX2 in [27].

In this approach, after the fusions are made (if any), the final recombining components are identified, and the non-recombining components are merged into the "rest", we proceed to the creation of the list of tuples that represent the components and their relations. Since all recombining components are independent, each one (including the "rest") has a corresponding tuple with just one value representing that component.

Figure 5.1 illustrates this case, with 6 components. Non-recombining components are represented in red and the recombining components in green. The solid lines indicate that the connected components were fused (are related), but no recombining components resulted from this fusion. As such, these components were added to the "rest", identified by the circle in green. The list of tuples would then be $[(0), (1), (2)]$, where 0 and 1 represent the components $C5$ and $C6$, respectively, and 2 the component "rest". The corresponding recombination graph would be a graph with 3 nodes, each representing a component, and no edges.

Since no dependencies exist, the algorithm chooses the best parent (partial solution) in each component separately, in a way similar to the original implementation of the GPX2. The method `px.evaluate(i, t)`, that evaluates the choices for each component, only returns the objective value of the chosen sub-tour. If the component is the "rest" it returns

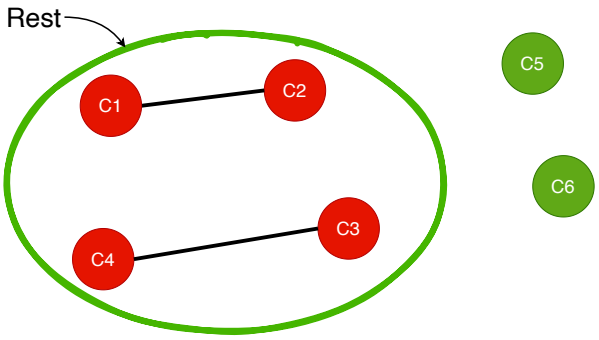


Figure 5.1: Dependencies according with the First Level approach to the implementation of GPX2 under the API. The recombining components are represented in green, the non-recombining components in red. Solid lines between components represent their dependencies. The green circle represents the "rest", which contains non-recombining components.

the objective value of the sum of the sub-tours in each non-recombining component for the chosen parent.

This approach, however, does not take much advantage of implementing under the API. Namely, the ability to consider the relations between components that are fused in the current approach. Recall that, when two components (or more) are fused, the chosen parent in each of these components must be the same.

5.3.2 Second Level

Another approach would be to *expose* the relations between fused, recombining or non-recombining components. In this approach, fusions are identified but are not actually made[1]. We keep the components initially identified before any fusion, and whenever two components are chosen to be fused, instead of fusing them, we store the information that these two components are dependent on each other. This information, along with the actual components, is used to create the list of tuples to be returned by `px.subfunctions()`.

In general, the resulting recombination graph would have edges between nodes (components) that would have been fused in the original implementation, indicating that choices made in one of the components affects the other directly and indirectly connected components.

However, the definition of the tuples is not so direct. The tuples should be defined as follows:

- One tuple of one element for each component.
- Tuples of two elements for each pair of components that are dependent on each other. These tuples indicate that its components are dependent and, thus, share an edge in the recombination graph.

The rest of the work must be done in the evaluation method, `px.evaluate(i, t)`. For tuples of just one element, this function returns the objective value of the partial tour of the indicated parent. For tuples of two elements (components), they are mainly used

[1]This is the main idea. In practice, fusions are still made on a different set of components. See Sub-section 5.3.4 for more information.

to give the feedback to the algorithm about the validity of the choices for two dependent components. If the choice is valid, the partial objective value returned is the objective value of the common edges between the components, since different components are connected through common edges. If it is invalid, it returns a very large objective value, that makes said choice much worse than what is possible in the instance being solved.

Working Like GPX2

We can make this approach work like the original implementation of GPX2.

Recall, from Subsection 3.2.1 that candidate components that are non-recombining (after going through the fusions), are joined in a final component called "rest", implying that the same parent must be chosen for all of these components, even though these components are not related to one another. By adding these components into the same component, an artificial dependence was created between components that were originally independent. This dependence between components is also represented by tuples of two elements, that are handled by the evaluation function in the same way as the tuples that represent real dependencies. However, 0 should be returned for valid choices whenever these components have no common edges.

We do not need to add an artificial relation to every pair of components that are not "naturaly" related. Considering that tuples with two components are edges of the recombination graph, we just need to add enough dependencies (edges) so that every component (node) in the recombination graph that would have gone to the "rest" has a path that connects it to every other component also in the "rest".

The Figure 5.2 illustrates this case. This is the same example as in Figure 5.1, except that instead of having the "rest", we added an artificial dependence between $C1$ and $C4$ (dashed line).

Considering this figure, the list of tuples, `sf`, would be:

$$[(0), (1), (2), (3), (4), (5), (0, 1), (0, 3), (2, 3)].$$

The recombination graph on the algorithm will be equal to the one in Figure 5.2.

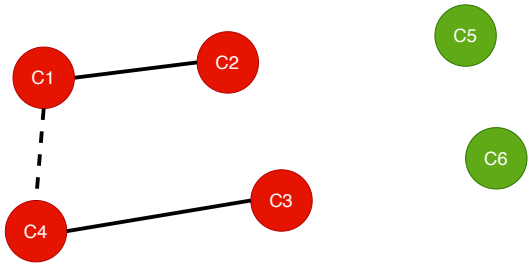


Figure 5.2: Dependencies according to the Second Level approach of implementing GPX2 under the API. The recombining components are represented in green, the non-recombining components in red. Solid lines between components represent the "natural" dependencies. The dashed line represents an artificial dependency.

To have this implementation working like the original implementation of GPX2 we just need to define in the evaluation function what an invalid choice means, for tuples with two components. An invalid choice would, then, be one where different parents are chosen for two mutually dependent components, and would lead to the evaluation function returning a very large value.

In the list of tuples above, this applies for the tuples $[(0,1),(0,3),(2,3)]$. Even if, for example, the tuples (0) and (1) are evaluated with different parents separately, when those choices come to $(0,1)$, the result of the evaluation invalidates the choices for the individual tuples, by the nature of dynamic programming.

5.3.3 Third Level

One final approach, is to allow different choices between related components that, when fused would have resulted in recombining components. Since the fusion resulted in a recombining component, then we are certain that any choice of parents between these components would not interfere with the choices in other, unrelated components. This way, we only need to test the validity of the resulting partial solution and not the whole final solution. Here, an invalid choice for a pair of related components, is one that would result in an invalid partial solution with respect to these components. In this case, suppose that the components $C1$ and $C2$, in Figure 5.1, once fused would have resulted in a recombining component and as such, would not be added to the "rest". Then, we would allow different choices of parents between these components, and an invalid choice would be one that resulted in the respective invalid partial solution.

As we will see in the Chapter 6, this approach adds the potential to explore more possible offspring, increasing the possibility of generating higher quality solutions, when compared with the GPX2.

5.3.4 Practical Considerations for GPX2

In the approaches presented in the sub-sections 5.3.2 and 5.3.3, we stated that no fusions are made, in order to simplify the explanation of the (general) idea. In practice, fusions should still be made, because for any two components that are fused, additional dependencies can be found that could not be found before. What should be done instead is to store a copy of the initially identified set of original components and have a data structure to store the dependencies found through fusions. We refer to this set as the set of copied components; these components do not change throughout the algorithm.

Whenever a fusion is made in the set of original components, the information about the respective dependency in the set of copied components is stored in these data structures. However, care should be taken to identify the real dependencies. Consider the (overly) simple example in Figure 5.3, with three components, where the lines represent one or more common edges connecting the respective components.

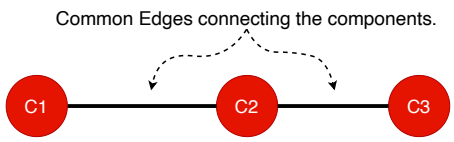


Figure 5.3: An example of three dependent non-recombining components.

Suppose that components $C1$ and $C2$ are fused, creating the component $C12$. Since $C1$ and $C2$ were fused in the set of original components, the found dependency is stored in the aforementioned data structure. Now, suppose that given the new component $C12$ in the original set of components, a possibility for fusion was found between $C12$ and $C3$. Before storing the newly found dependency, we should search for the component fused into $C12$ that is directly connected to $C3$, in the set of copied components. We see that this

component is $C2$. Then, the stored dependency is between $C2$ and $C3$. The set of copied components and the stored dependencies are then used to define the sub-functions.

5.4 Concluding Remarks

With this API we pretend to generalise the development of PXs. By identifying the common structures and operations whose specifics differ in each problem, and factoring out what is common, we exposed the problem of recombination by PXs into a pseudo-boolean-like problem, independently of the domain of application.

Motivated by the definition of this API, in the next chapter we present an analysis of the GPX2 in comparison to other, less restrictive operators, with a view to identify opportunities for improvement. Finally, we show how the proposed API could be used to implement these improvements.

Chapter 6

An Experimental Analysis of GPX2

In this chapter, we present an analysis of GPX2 with respect to two forms of optimal recombination: OREPR and OEPR (introduced in Sub-section 2.3.1); as well as with respect to a less restricted version of GPX2, provisionally called Exhaustive Search Generalised Partition Crossover (GPX_e). GPX2 was introduced in the Sub-section 3.2.1. Next, we explain in more detail the other operators.

The GPX_e is a modified implementation of GPX2 that does not perform fusions and searches exhaustively for the best combination of choices in each candidate component insted. In more detail, after partitioning the graph of the union of the parents into candidate components, GPX_e identifies the recombining and non-recombining components and exhaustively tests all possible combinations of choices, returning the best one.

The OEPR was implemented by solving the following Integer Linear Programming (ILP) formulation for the TSP adapted from [16]. For a given pair of parent tours and a TSP instance of n cities, where $V = \{i : i \in \mathbb{Z}_{\geq 1} \wedge i \leq n\}$ is the set of cities, s the starting and final city, $E \subseteq V \times V$ is the set of edges present in both parent solutions, and $d_{ij} \in \mathbb{R}$ $(1 \leq i \neq j \leq n,\ i, j \in \mathbb{Z})$ is the distance matrix:

$$
\begin{aligned}
\min \quad & \sum_{(i,j)\in E} d_{ij}x_{ij} \\
\text{s.t.} \quad & \\
& \sum_{(i,j)\in E} x_{ij} = 1, && \forall j \in V \\
& \sum_{(i,j)\in E} x_{ij} = 1, && \forall i \in V \setminus \{s\} \\
& u_i - u_j + (n-1)x_{ij} \leq n-2, && i,j \in V \setminus \{s\}, (i,j) \in E, i \neq j \\
& u_i - u_j \leq n-2, && i,j \in V \setminus \{s\}, (i,j) \notin E, i \neq j \\
& x_{ij} \in \{0,1\}, && 1 \leq i \neq j \leq n \\
& u_i \in \mathbb{R}, && 1 \leq i \leq n
\end{aligned}
\tag{6.1}
$$

OEPR starts by creating a new graph as the union of both parent tours, resulting in the corresponding set of edges E. Then, it proceeds by solving a regular TSP instance until it reaches the global optimal solution, that is made up only of edges present in either parent.

The OREPR was implemented in a similar manner as OEPR, the only difference being an added constraint $x_{ij} + x_{ji} = 1$, $\forall (i,j) \in F$, where F is the set of *forced edges*, which

correspond to the common edges between parents.

The general goal of these experiments was to assess the performance of these operators, and observe how it varied with the bond distance (see Section 4.2) between the solutions being recombined. In the following section, we show how the experiments were conducted, in the experimental setup.

6.1 Experimental Setup

The experimental setup is based on solutions generated from three TSP instances from [1], the four recombination operators introduced earlier in this chapter, and sets of solutions generated from an ILS implementation.

6.1.1 Instances and Solutions

An ILS algorithm for the TSP, with 2-opt neighbourhood and double-bridge move as perturbation, was used to generate local optima from three TSP instances from the TSPLIB repository [1] with 52, 150, and 200 cities. The names of these instances are, respectively, `berlin52.tsp`, `ch150.tsp`, and `kroA200.tsp`.

These three somewhat small instances sizes were chosen for computational reasons. Larger instances would make it harder to get good quality local optima – nearly as good as the global optimum – without needing to generate a larger number of solutions. Given that one of the steps (see Sub-section 6.1.3) requires the computation of the bond distances between all pairs of unique solutions of one or more runs, this could could imply calculating these distances for too large a number of pairs, generating a vast amount of data to process, which could be impractical due to time and memory limitations. Additionally, since we use two exact operators, OEPR and OREPR, larger instances would take too long to solve, which would be impractical.

Using instances of different sizes allowed running the experiments, in increasing order of instance sizes. Moreover, instances of different sizes also enable us to have a broader view of the quality of each operator for the same bond distance in different contexts, since the number and structure of the possible offspring of pairs of parents at a given distance on instances with sizes, for example, 52 and 200 is different, and so are the opportunities for recombination.

For each instance, 40 independent executions of the ILS were run, where each run was executed in two "stages": until it reaches the known global optima (available in [1]) and, then, until it finds 2000 more local optima.

6.1.2 Filtering of ILS Solutions

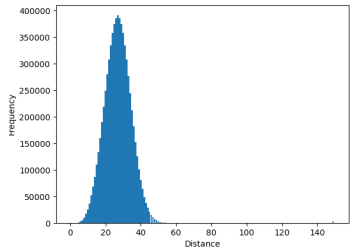
Due to the large number of solutions generated and time restrictions, only a fraction of the local optima that were found was used. Supporting this decision was the evidence that the distribution of the bond distances of *unique* solutions within each run is approximately equal from the distribution of the bond distances of *unique* solutions between runs. As such, we can select a subset of runs without losing the representativeness of each possible distance. This is illustrated in Figure 6.1, for the instance `ch150.tsp`, for three sets of solutions: unique solutions of run 0, unique solutions of run 1, and unique solutions of

both runs 0 and 1.

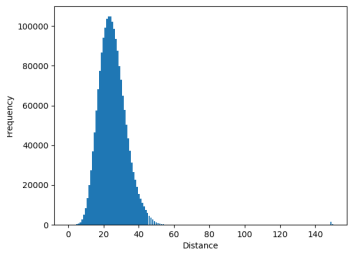
By *unique*, we mean that in order to get this information, for each of the sets of solutions presented, we filtered the repeated solutions, keeping only the first one found, where two solutions are equal if the corresponding permutation of one of them can be made equal to the other by rotation or inversion.

Figures 6.1a and 6.1b show the distribution of the distances all pairs of unique solutions in runs 0 and 1, respectively. Figure 6.1c shows the bond distances between all pairs of unique solutions in both runs 0 and 1. For this instance, we choose three runs, so that the number of solutions is as close as possible between instances; big enough to be useful, but not so big that it is not be manageable.

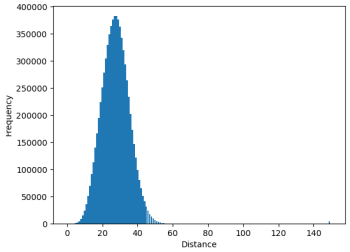
The distribution of the bond distances between all pairs of unique solutions among the chosen runs is shown in Figure 6.2. As it is possible to see, the distribution is similar to those of the Figure 6.1.



(a) Bond distances distribution in run 0.



(b) Bond distances distribution in run 1.



(c) Bond distances distribution in runs 0 and 1.

Figure 6.1: Bond distances distributions between pairs of unique solutions in runs 0 and 1.

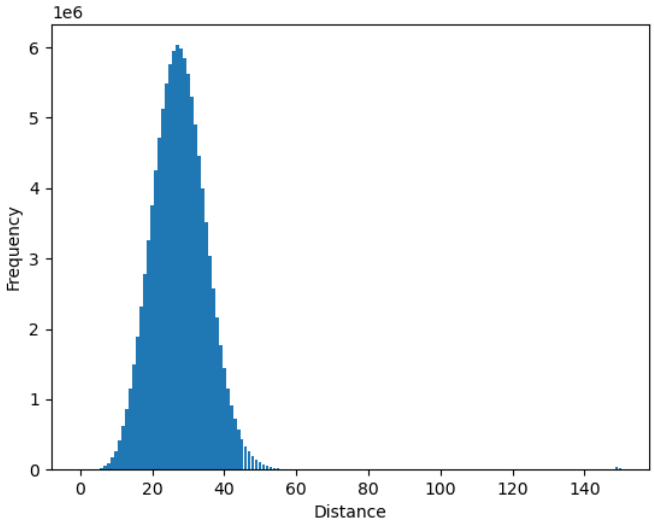


Figure 6.2: Bond distances distribution for the 3 runs selected for instance `ch150.tsp`

This is also the case for the instances `berlin52.tsp` and `kroA200.tsp`. For the former, we considered 20 runs, for the latter 1 run was considered.

After this analysis, the selected sub-set of runs for each instance were merge and the respective solutions were filtered in the same way as above. We note that the final set in each of the instances also includes the initial, random solutions, since they are unique).

6.1.3 Sampling, Recombination, and Data Handling

In the next step, we calculated the bond distances for each unique pair of solutions, from the filtered solutions. Up to 1000 pairs of tours were selected for recombination by uniform sampling according to the bond distance. The sampling was not completely uniform because some distances have fewer corresponding pairs than what would be required. The sampled pairs were, then, recombined with each recombination operator. For each pair of parents, the objective values of the offspring of each recombination operator were normalised, based on the objective value of the optimal tour (OEPR) and that of the best parent, as shown in the Equation 6.2.

In this equation, $f(o_r)$ is the value of the offspring o for a given recombination operator r, whose parents are at some distance d; $f(p_{best})$ is the best of those parents; $f(o_{oepr})$ is the objective value of the (optimal) offspring for the same parents; and $f_{norm}(o)$ is the objective value of o normalised. When dividing by 0, $f_{norm}(o) = 0$ by default.

According with Equation 6.2, if $o_r = o_{oepr}$, the optimal offspring, $f(o_r) = f(o_{oepr})$, and $f_{norm}(o) = 0$; if $o = p_{best}$ and $p_{best} \neq o_{oepr}$, then $f_{norm}(o) = 1$. None of the operators presented produces an offspring the is worse than its best parent.

$$f_{norm}(o) = \frac{f(o_r) - f(o_{oepr})}{f(p_{best}) - f(o_{oepr})} \tag{6.2}$$

The normalised values were grouped by distance, to be used for the main analysis.

The table 6.1 summarises the information presented in this and the preceding sections.

Instance	Size	# Runs considered	Total solutions	# sampled solutions in total
berlin52.tsp	52	20	12276	909
ch150.tsp	150	3	13830	909
kroA200.tsp	200	1	11273	905

Table 6.1: Data about the instances used. The first column indicates the name of the instance; the second, the respective number of cities; the third, the number of independent runs considered and how many solutions those runs have; and the last column, the number of sampled solutions.

In the next section, we present an analysis of the results.

6.2 Analysis and Discussion

In this section, we present an analysis of the operators mentioned before with respect to the bond distance, identify opportunities to improve the GPX2, and show how the API proposed in Section 5 can be used to that end.

The Figures 6.3 to 6.5 show where each of the operators is with respect to optimal recombination, represented by OEPR, as a function of the distance.

Going from the most optimal, besides OEPR, to the least optimal, we observe in the case of the OREPR enforcing the "respect" property – preserving the common edges between parents – reduces the quality of the offspring produced. This is expected, since the offspring that can be generated is restricted to those that have *all* the common edges between the two parents, and the OTM tour might not have all common edges.

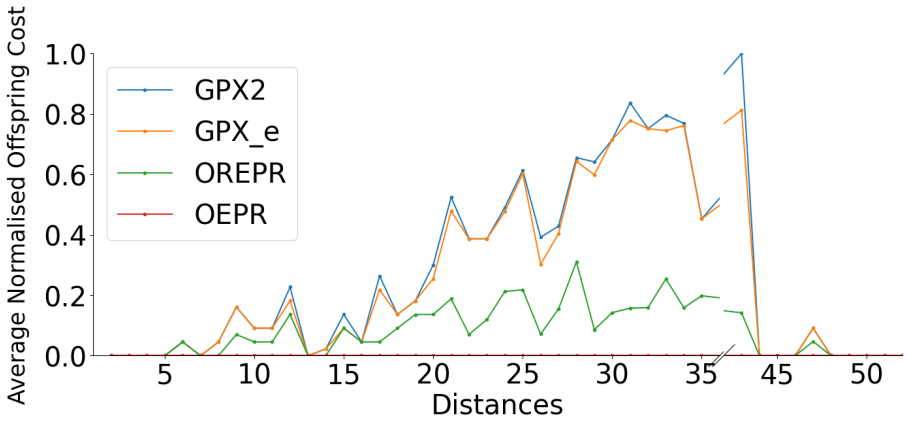


Figure 6.3: Instance berlin52.

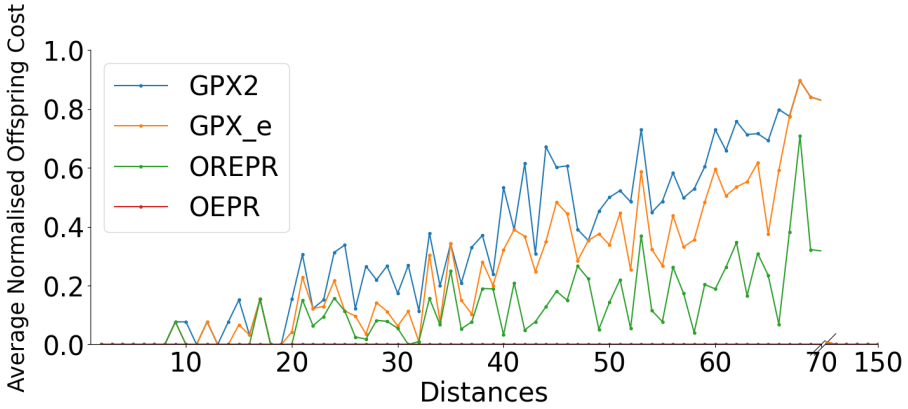


Figure 6.4: Instance ch150.

The quality of the offspring generated is further reduced by restricting the choices to the components, only as it is done in the GPX$_e$ operator since, for all the edges of each candidate component, we are forcing the edges to come from the same parent. This also reduces even more the number of offspring that can possibly be generalised.

In GPX2, fusing the non-recombining components reduces even further the quality of the solutions generated. This happens because fusing non-recombining components, results in less but bigger components. In some sense, the restriction is the same as in GPX$_e$ but there are larger sets of edges (components) whose parent choice must be the same.

Additionally, we note that in the three instances, for both very large and very small distances, the quality of the offspring is the same – optimal.

When the distance is small, both parents have a large number (if not all) of common edges that are directly inherited by the offspring. This means that GPX2 and GPX$_e$ find very few (or no) candidate components to be used in the recombination. Due to the high frequency of common edges, the few components found have a high probability of being recombining components from the beginning. This means that no fusion is required, in the case of GPX2. Moreover, the recombining components are independent from each other and, as such, there are no additional combinations to explore by the GPX$_e$. These conditions place all possible offspring very close to the optimal offspring, and all that is needed, as far as GPX2 and GPX$_e$ are concerned, is to choose the best parent in each of the few components identified.

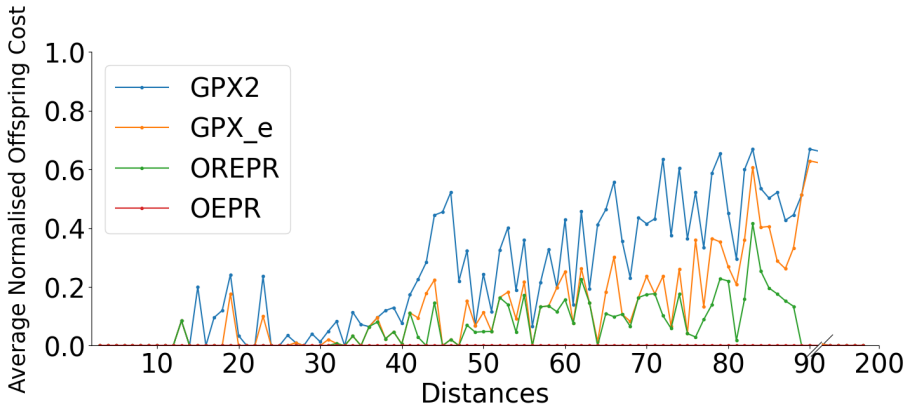


Figure 6.5: Instance kroA200.

When the distances between parents are large, the number of common edges is small (or none) and very few candidate components are found, by GPX2 and GPX$_e$, and the ones that are found might be very big (with respect to the instance size). When no components are actually found, we can think of both parents as belonging to the same, large component. Then, for each of the few candidate components found, all it takes is to choose the best parent in each one.

In both situations, OREPR and OEPR operators will choose the best tour solution anyway. Moreover, if GPX2 and GPX$_e$ find the optimal solution *while keeping the common edges*, so does the OREPR because it is not restricted by the notion of components.

A Concrete Example

We can have a glimpse of what causes these differences in performance between operators in the example solutions in Figures 6.6 to 6.10, from the instance `berlin52.tsp`.

The Figure 6.6 shows the graph of the union of two parent tours and the respective common edges. The boxes in green and yellow emphasise the components identified by GPX2 (before the fusions and the same as in GPX$_e$), inside of which are the edges that belong to each component. Additionally, the component to which an edge belongs is shown in its label. Components with boxes of the same colour mean that these components are fused in GPX2. As such, in this operator, after the fusion steps, there are only two components: one with the (initial) components A, B, C, E, and F (the green components); and the component D (the yellow component), to which no fusion was applied.

We also note that, the fact that GPX2 merges the green components, indicates that initially these components were not recombining components. On the other hand, no fusion was made with the component D, which means that this component was identified in the beginning as a recombining component. This also means that, in practice, the GPX$_e$ operator only performs exhaustive search in the green components.

The remaining Figures 6.7 to 6.10 show the tour generated by each operator. We begin by the solution of GPX2. As mentioned above, components highlighted in the same colour were fused into a larger component in GPX2 and, as can be seen in the Figure 6.7, the same parent was chosen in the components highlighted in the same colour. Namely, it selected the blue parent for the resulting component from the fusion of the green components and the red parent for the yellow component.

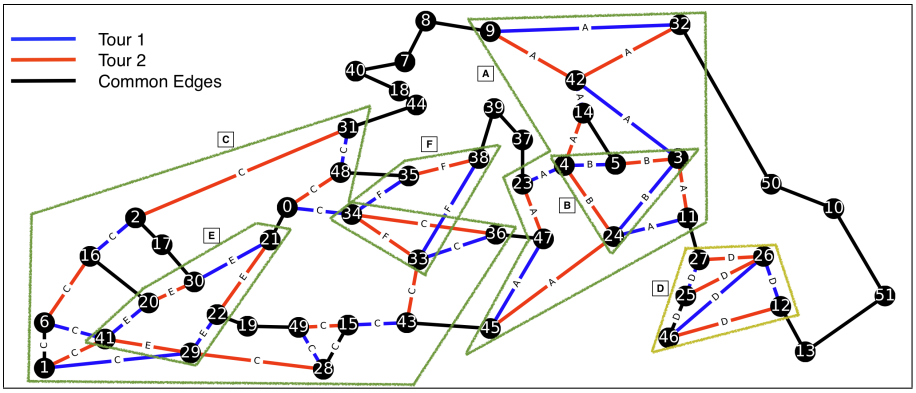


Figure 6.6: Union of two parent solutions.

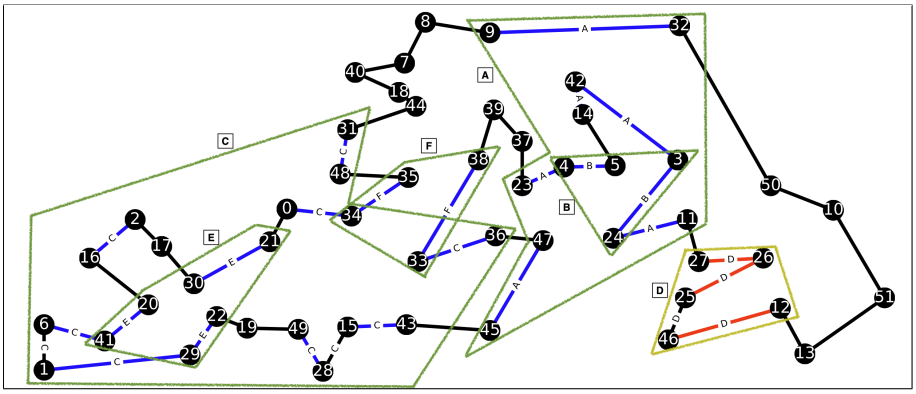


Figure 6.7: GPX2 solution.

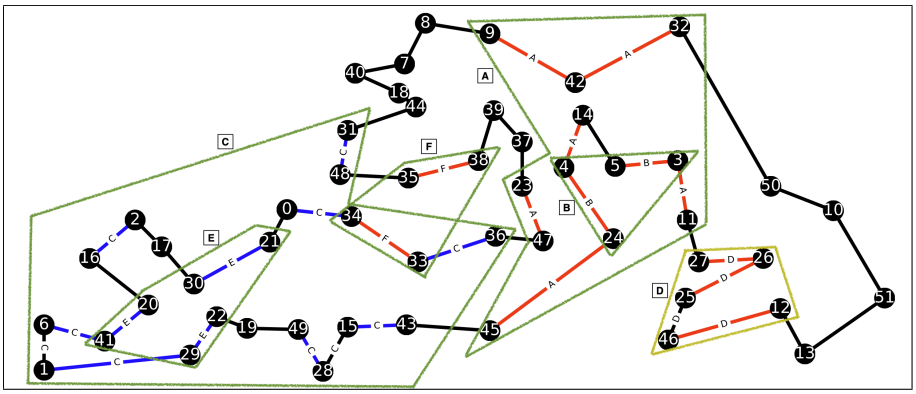


Figure 6.8: GPX$_e$ solution.

However, since GPX$_e$ does not perform any fusion, it was able to find a better solution by choosing a different parent for some of the green components, as seen in Figure 6.8. Comparing with the solution of GPX2, we see that in the green components F, A and B a different parent was selected from the one in the other green components. This shows that exposing the relations between non-recombining components adds to the potential of finding better offspring.

Reducing even more the restrictions so that it only keeps the common edges, as done in the OREPR operator, without using components, creates even more opportunities to find better solutions. In the Figure 6.9, we observe that the OREPR was able to find a better solution by selecting different parents for edges within of what would be the green components A and B.

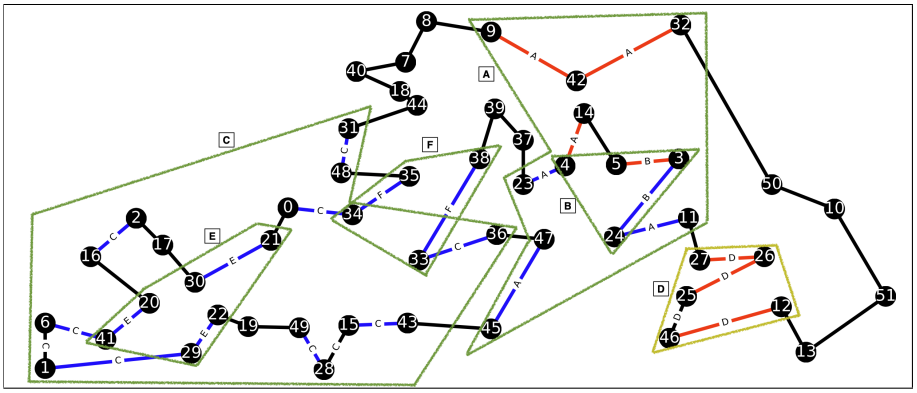


Figure 6.9: OREPR solution.

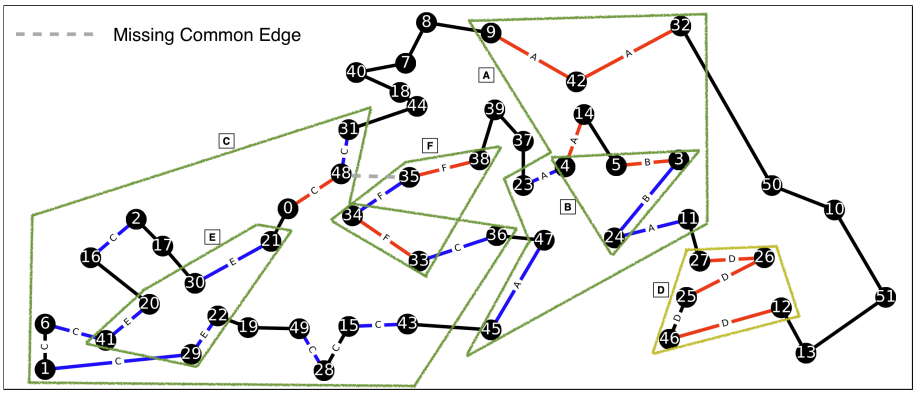


Figure 6.10: OEPR solution.

However, removing all the restrictions above, of using components and/or common edges, except the "allele transmition", enables, as expected, to find even better solutions – in this case, the true optimal solution. In Figure 6.10, we observe that the OEPR found the optimal solution by, among other things, ignoring the common edge $(48, 35)$.

Some Remarks

The OREPR and OEPR are known NP-Hard problems [9]. The GPX_e is also impractical for a large number of non-recombining components. However, there is room for improvement.

The results show that by taking into account the relationships between non-recombining candidate components, that would result in recombining components if fused, it should be possible to create a new PX that produces strictly better solutions than GPX2.

The API proposed provides the basis to make these improvements, because it provides mechanisms and ways of thinking that motivate and facilitate the exploration of the relations between candidate components. Under the proposed API, we can go from a PX that produces the same results as GPX2, by considering the final recombining components, without any relations to other components; to one that produces the better results, by considering the relations between components; to something in between.

Chapter 7

Conclusions and Future Work

Recombination operators are crucial to the performance of EAs and meta-heuristics. Good recombination operators can improve significantly the performance of these algorithms. However, the difficulty (in general) of the ORP presents an obstacle for the improvement of their performance.

PXs are efficient recombination operators that can often produce good quality offspring. However, they only provide an approximation to the ORP. For the TSP, we showed that the most recent partition crossover, GPX2, still has some room for improvement towards optimal recombination that is respectful and transmits alleles.

We proposed an API aimed at facilitating this improvement by promoting a new way of thinking not only about partition crossovers, but also about partitions. By providing the means to deal with the relations between components, recombination operators implemented under this API would be able to find more, better offspring, possibly at the cost of some performance.

By separating the development of PXs into two parts – partitioning and decision – and identifying its basic operations, developers can focus only on what is really necessary for the problem at hand. Moreover, improvements made to the decision part would benefit partition crossovers developed for any problem.

Additionally, we shed some light on the geometric interpretation of PXs. By proving that all PXs have the inbreeding properties of geometric crossovers, we showed that all PXs may be geometric under some distance. Indeed, we showed that all PXs for the TSP are geometric under the bond distance.

7.1 Future Work

One limitation of this work was the lack of an experimental analysis of several PXs developed according with the API. Such an analysis would allow the real increase in execution time of PXs developed with this API to be evaluated. Such implementations could be done in an time efficient programming language, such as C.

Another future line of work would be to develop a PX for the TSP, according with the proposed API, with the aim of improving the quality of the offspring, compared with GPX2. Such improvement could be done by considering the relationships between component that, if fused, would have resulted in a recombining component (the “third level” in Section 5.3).

Improvements to the algorithm that performs the decisions in the API or other search methods could also be done. This would ultimately benefit all PXs implemented under the API.

Finally, another question worth exploring is of whether PXs for the TSP are geometric according to the 2-opt distance. First, experimentally understand if that is possible – if a counter-example is found, then it is not possible –, and then by trying to prove it formally, if no counter-example is found experimentally.

References

[1] TSPLIB. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html`. Accessed: 2021-10-12.

[2] O. Abdelkafi, B. Derbel, A. Liefooghe, and D. Whitley. On the Design of a Partition Crossover for the Quadratic Assignment Problem. In T. Bäck, M. Preuss, A. Deutz, H. Wang, C. Doerr, M. Emmerich, and H. Trautmann, editors, *Parallel Problem Solving from Nature – PPSN XVI*, volume 12269, pages 303–316. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

[3] K. D. Boese, A. B. Kahng, and S. Muddu. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Research Letters*, 16(2):101–113, 1994.

[4] F. Chicano, G. Ochoa, D. Whitley, and R. Tinós. Enhancing partition crossover with articulation points analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 269–276, Kyoto Japan, July 2018. ACM.

[5] F. Chicano, G. Ochoa, D. Whitley, and R. Tinós. Quasi-Optimal Recombination Operator. In A. Liefooghe and L. Paquete, editors, *Evolutionary Computation in Combinatorial Optimization*, volume 11452, pages 131–146. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science.

[6] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[7] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2015.

[8] A. Eremeev and J. Kovalenko. Optimal recombination in genetic algorithms for combinatorial optimization problems: Part I. *Yugoslav Journal of Operations Research*, 24(1):1–20, 2014.

[9] A. Eremeev and J. Kovalenko. Optimal recombination in genetic algorithms for combinatorial optimization problems: Part II. *Yugoslav Journal of Operations Research*, 24(2):165–186, 2014.

[10] D. Hains, D. Whitley, and A. Howe. Improving lin-kernighan-helsgaun with crossover on clustered instances of the tsp. In C. A. C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, and M. Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII*, pages 388–397, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[11] D. R. Hains. GENERALIZED PARTITION CROSSOVER FOR THE TRAVELING SALESMAN PROBLEM. Master's thesis, Colorado State University, 2011.

[12] K. Helsgaun. An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, Oct. 2000.

[13] H. H. Hoos and T. Stützle. 1 - introduction. In H. H. Hoos and T. Stützle, editors, *Stochastic Local Search*, The Morgan Kaufmann Series in Artificial Intelligence, pages 13 – 59. Morgan Kaufmann, San Francisco, 2005.

[14] H. H. Hoos and T. Stützle. 2 - sls methods. In H. H. Hoos and T. Stützle, editors, *Stochastic Local Search*, The Morgan Kaufmann Series in Artificial Intelligence, pages 61 – 112. Morgan Kaufmann, San Francisco, 2005.

[15] T. Koopmans and M. J. Beckmann. Assignment problems and the location of economic activities. Cowles Foundation Discussion Papers 4, Cowles Foundation for Research in Economics, Yale University, 1955.

[16] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer Programming Formulation of Traveling Salesman Problems. 7(4):326–329.

[17] A. Mobius, B. Freisleben, P. Merz, and M. Schreiber. Combinatorial Optimization by Iterative Partial Transcription. *Physical Review E*, 59:4667–4674, Apr. 1999. arXiv: cond-mat/9902034.

[18] A. Moraglio and R. Poli. Topological Interpretation of Crossover. In K. Deb, editor, *Genetic and Evolutionary Computation – GECCO 2004*, Lecture Notes in Computer Science, pages 1377–1388, Berlin, Heidelberg, 2004. Springer.

[19] A. Moraglio and R. Poli. Inbreeding properties of geometric crossover and non-geometric recombinations. In C. R. Stephens, M. Toussaint, D. Whitley, and P. F. Stadler, editors, *Foundations of Genetic Algorithms*, pages 1–14, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[20] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover Publications, Mineola, N.Y, 1998.

[21] O. Quevedo de Carvalho, R. Tinos, D. Whitley, and D. Sipoli Sanches. A New Method for Identification of Recombining Components in the Generalized Partition Crossover. In *2019 8th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 36–41, Salvador, Brazil, Oct. 2019. IEEE.

[22] N. J. Radcliffe and P. D. Surry. Fitness variance of formae and performance prediction. volume 3 of *Foundations of Genetic Algorithms*, pages 51 – 72. Elsevier, 1995.

[23] R. Tinos. rtinos/gpx2, July 2019. original-date: 2018-04-11T11:33:19Z.

[24] R. Tinós, K. Helsgaun, and D. Whitley. Efficient recombination in the lin-kernighan-helsgaun traveling salesman heuristic. In A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete, and D. Whitley, editors, *Parallel Problem Solving from Nature – PPSN XV*, pages 95–107, Cham, 2018. Springer International Publishing.

[25] R. Tintos, D. Whitley, and F. Chicano. Partition Crossover for Pseudo-Boolean Optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII - FOGA '15*, pages 137–149, Aberystwyth, United Kingdom, 2015. ACM Press.

[26] R. Tinós, D. Whitley, and G. Ochoa. Generalized asymmetric partition crossover (GAPX) for the asymmetric TSP. In *Proceedings of the 2014 conference on Genetic and evolutionary computation - GECCO '14*, pages 501–508, Vancouver, BC, Canada, 2014. ACM Press.

[27] R. Tinós, D. Whitley, and G. Ochoa. A New Generalized Partition Crossover for the Traveling Salesman Problem: Tunneling between Local Optima. *Evolutionary Computation*, 28(2):255–288, June 2020.

[28] D. Whitley, D. Hains, and A. Howe. Tunneling between optima: Partition crossover for the traveling salesman problem. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, page 915–922, New York, NY, USA, 2009. Association for Computing Machinery.

[29] D. Whitley, D. Hains, and A. Howe. A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover. In R. Schaefer, C. Cotta, J. Kołodziej, and G. Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, pages 566–575, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.