Lancaster University

A Monte Carlo Tree Search algorithm to solve Kiwi's Traveling Salesman Challenge 2.0

Arnaud Da Silva

36471977

Supervisor

Ahmed Kheiri

A thesis submitted in fulfillment of the requirements for the degree of Master of Science Business Analytics

in the

Lancaster University Management School
Department of Management Science

September 2024

Declaration of Authorship

I, **Arnaud Da Silva**, hereby declare that this thesis entitled, **"Title"**, is all my own work, except as indicated in the text.

The report has been not accepted for any degree and it is not being submitted currently in candidature for any degree or other reward.

Signed: _____

Date: _____

Abstract

(TODO) Give a short (1 page) overview of the work. This should summarise (not advertise) your research project. After reading the abstract the reader should know what problem you are tackling, the techniques you are using, the results you have achieved.

Acknowledgements

(TODO) Thanking anyone who has helped you in any way

Contents

List of Figures

List of Tables

Abbreviations

AK Ahmed Kheiri

MS Management Science

Dedicate this to someone here.

Chapter 1

Introduction

(TODO) Set the scenes. Explain why you are doing this work and why the problem being solved is difficult. Most importantly you should clearly explain what the aims and objectives of your work are.

(TODO) Structure of the thesis. Academic publications produced (if any), including any achievements/highlights

1.1 Background

The number of flights connection keep increasing every year [4], more than have 38 millions flights have been scheduled in 2023 - hence it is a challenge for traveler's to find the best and cheapest flight connections for their specific journey, especially when one has to visit a big number of cities/areas. Consequently, traveler agencies have deployed online trip planner algorithm in order to find flights connection that match traveler's requirement. For example, Google Flights, skyscanner, Kayak and Kiwi.com.

These agencies have launched different challenges in order to create and build powerful trip planner algorithm. For instance, as mentionned in [5], OpenFlights.org launched the Air Travelling Salesman project and Kiwi.com has launched a project in 2017 called Traveling Salesman Challenge where the current algorithm used by Kiwi.com was developed and proposed a new challenge in 2018, the Traveling Salesman Problem 2.0 which is the focus of this study.

1.2 Research objectives

- Goal of this dissertation is to solve Kiwi's challenge with a method that has not been used yet
- No focus on the time limit - use Python and we then cannot compete against other participants - will be for later
- Try to find better solutions than the state of the art for some instances
- Big instances (instances I9,I10 etc) - not a major focus because they do not represent potential use case, but to study

1.3 Academic publication

- Discuss with Ahmed - implentation in C to speed up the code
- + think of new smart policies to solve the instances in more reasonnable time

1.4 Dissertation structure

Chapter 2

Literature Review

(TODO) Present a survey of your main approach and an overview of the approaches proposed previously for solving the problem dealt with in this work

(TODO) Identify the practical and research motivation of this work and the literature gaps

(TODO) How convincing is the authors' argument? (Critical response - comparisons with other research, strengths or weaknesses but in relation to your research)

2.1 Optimisation in Air Travel

In this section, we discuss some common challenges faced by airline companies and demonstrate the importance of optimisation in decision-making for the success and competitiveness of airline companies.

2.1.1 Fleet Assignment Problem

The Fleet Assignment Problem (FAP), as discussed in [6] involves assigning different types of aircraft, to flights based on their capabilities, operational costs, and revenue potential. This decision greatly influences airline revenues and is a vital part of the overall scheduling process. The complexity of FAP is driven by the large number of flights an airline manages daily and its interdependencies with other processes like maintenance and crew scheduling.

2.1.2 Crew Scheduling Problem

The Crew Scheduling Problem (CSP), as discussed in [7], involves assigning crews to a sequence of tasks, each with defined start and end times, with the primary objective of ensuring that all tasks are covered while adhering to regulations on maximum working hours for crew members.

This problem is particularly critical for low-cost airlines, for example in the United Kingdom in 2023, low-cost flights comprise 48% of the scheduled capacity (total number of seats offered) [8], which rely heavily on optimised crew schedules to maintain competitiveness. Efficient crew scheduling is essential not only for low cost carriers and for cost minimisation but also for ensuring operational reliability and flexibility in response to unexpected disruptions. [9]

2.1.3 Disruption Management

Disruptions in airline operations, as noted in [10], can occur due to various factors, including crew unavailability, delays from air traffic control, weather conditions, or mechanical failures. Given that flight schedules are typically planned months in advance [11], effective disruption management is crucial to minimise the impact on passengers and overall airline operations.

The two mains drivers of disruption management are aircraft and crew recovery.

- Aircraft recovery: Optimisation tools help manage the complex logistics of matching available aircraft with rescheduled flights, considering factors like airport availability and maintenance requirements.
- Crew recovery: Optimisation tools are used to adjust crew schedules, taking into account factors such as legal working hours, crew availability, and the need to cover all flights efficiently. These tools help in developing feasible and compliant crew rosters that adapt to the new flight schedules.

These optimisation strategies, supported by advanced software, for instance [12] and [13], are crucial for reducing the impact of disruptions and boosting operational resilience in the airline industry.

2.1.4 Airline adaptation to new demand

Airline companies must continuously adapt their schedules to meet evolving market demands, particularly with the growing dominance of leisure travel over business travel, which has introduced new patterns of demand as shown on Figure 2.1 in Europe. This seasonality poses a challenge for airlines as they have to balance high demand during peak seasons with the risk of underutilisation during off-peak times.

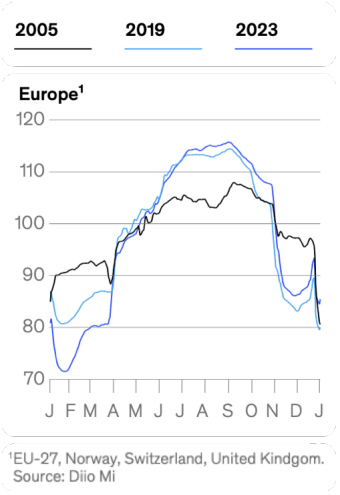


FIGURE 2.1: European demand seasonality [1]

Since travel demand varies throughout the year, airlines use a variety of techniques to achieve operational efficiency while maximising revenue [1]. For instances, airlines sell nearly 65% more seats. To ensure their operatios remain efficient during periods of heightened demand, airline companies make the required allowance for additional aircraft and crew by optimisation models that specify priority routes and requirements for additional flights, alongside effective crew rotation management.

In contrast, winter months pose a different type of problem where demand drops, which can potentially lead to underutilisation of aircrafts. To manage this, airlines are known to turn to ACMI leasing (agreement between two airlines, where the lessor agrees to provide an aircraft, crew, maintenance and insurance [14]) during periods of low demand to temporarily reduce fleet size by outsourcing their capacity. Alongside this, they also increase maintenance activities and incentivise crews to take holidays or undergo training to maximise productivity across the operation. Equally, on a year-round basis, airlines apply dynamic pricing algorithms to vary fares in reaction to real-time demand patterns. In high-demand summer months, fares are tactically set so as to maximise revenues from travelers willing to pay more, while in winter, pricing strategies are aimed at stimulating demand with fare reductions to fill seats that otherwise would have gone empty. Such

adaptive strategies are critical to the airlines for effectively beating the seasonal ebbs and flows in the travel industry.

2.2 Traveling Salesman problem and its adapation

The Traveling Salesman Problem is a well known problem in the Operational Research and Computer Science fields. A simple description of the TSP is to find the best roundtrip for a saleman that has to travel around a given number of cities while minimising the overall journey's distance. This problem is characterised as $\mathcal{NP}$-Hard [15]. This means that there is no known polynomial-time algorithm that can solve all instances of the problem efficiently . Regarding time complexity, if we were to solve it exploring all the possible solutions, the time complexity would have been $\mathcal{O}(\frac{(n-1)!}{2})$ where n represents the number of cities.

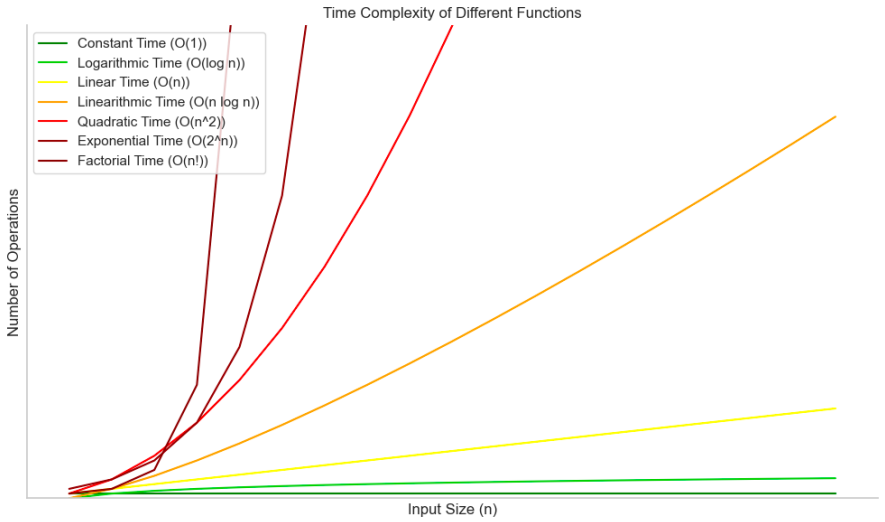


FIGURE 2.2: Time complexity of different functions [2]

On Figure 2.2, different time complexities are compared and demonstrates that the factorial time complexity is the worst. Therefore, these kinds of $\mathcal{NP}$-Hard problem are typically not solved exploiting all the search area but using heuristics algorithms. Heuristics solutions do not guarantee to find the absolute optimal solution but can find near-optimal solutions within more reasonnable timeframes.

The TSP has been studied extensively, and, many variants can be derived from it:

- Symmetric TSP (STSP): The distance between cities are symmetric, meaning that the distance to travel from city A to city B is the same as from city B to city A.
- Assymetric TSP (ATSP): The distance between cities are assymetric, meaning that the distance to travel from city A to city B is different than the distance to travel from city B to city A.[16]

- Multiple TSP (mTSP): Instead of one salesman, multiple salesman are starting from one city, they visit all the cities such that each city is visited exactly once. [17]
- Time Window TSP (TWTSP): Each city has to be visited in a defined time slot. [18]
- Price-collection TSP (PCTSP): Not all the cities have to be visited, the goal is to minimise the overall traveler's distance while maximising the price collected earned when visiting a city. [19]
- Stochastic TSP (STSP): The distances between the cities or the cost of travels are stochastic (ı.e random variables) rather than deterministic. [20]
- Dynamic TSP (DTSP): The problem can change over time, that means that new cities can be added or distances between cities can change while the salesman has already started his journey. [21]
- Generalised TSP (GTSP): The cities are grouped into clusters, the goal is to visit exactly one city from each cluster. [22]
- Open TSP (OTSP): The traveler does not have to end his journey at the starting city. [23]

Multiple algorithms have been developed to address these TSP variants, we can classify them into two categories:

- **Exact Algorithms**: These algorithms aim to find the optimal solution to the TSP by exploring all possible routes or by using mathematical techniques to prune the search space efficiently. Examples include:
  - **Branch and Bound**: This method systematically explores the set of all possible solutions, using bounds to eliminate parts of the search space that cannot contain the optimal solution. It is often used for smaller instances of TSP due to its computational intensity. [24]
  - **Cutting Planes**: This technique adds constraints (or cuts) to the TSP formulation iteratively to remove infeasible solutions and converge to the optimal solution. This approach is particularly effective for symmetric TSPs. [25]

  - **Dynamic Programming**: Introduced by Bellman, this approach breaks down the TSP into subproblems and solves them recursively, which is highly effective for specific TSP variants, though its complexity grows exponentially. [26]

- **Approximation and Heuristic Algorithms**: These algorithms are designed to find near-optimal solutions within a reasonable time frame, specifically for large-scale problems where exact methods are computationally infeasible. Examples include:

  - **Greedy Algorithms**: These algorithms make a series of locally optimal choices in the hope of finding a global optimum. An example is the Nearest Neighbor algorithm, which selects the nearest unvisited city at each step. [27]
  - **Genetic Algorithms**: Inspired by the process of natural selection, these algorithms evolve a population of solutions over time, using operations such as mutation and crossover to explore the solution space. [28]
  - **Simulated Annealing**: This probabilistic technique searches for a global optimum by allowing moves to worse solutions based on a temperature parameter that gradually decreases. It is particularly useful for escaping local optima. [29]
  - **Ant Colony Optimization**: This metaheuristic is inspired by the foraging behavior of ants and uses a combination of deterministic and probabilistic rules to construct solutions, which qre grqdually refined through updates based on pheromone trails. [30]

Some TSP problems (or its variants) have been solved using other algorithms.

2.3 The Monte Carlo Tree Search algorithm

The Monte Carlo Tree Search (MCTS) algorithm can be characterised as less traditionnal than the previously enounced methods in Section 2.2 because MCTS is typically used in games. MCTS' (and its variants) have been successfully implemented across a range of games, such as Havannah [31], Amazons [32], Lines of Actions [33], Go, Chess, and Shogi [34], establishing it as the state-of-the-art algorithm [35], [36], [37]. It is widely used in board games and is increasingly popular since Google DeepMind developed AlphaGo. AlphaGo is a software that was created to beat the best Go's player in the world.

Go is a board game from China where two players take turns placing black or white stones on a grid. The goal is to capture territory by surrounding empty spaces or the opponent's stones. Despite its simple rules, Go is a complex game, with countless possible moves and strategies. It is known for its balance between intuition and logic, hence why it has been a significant focus of artificial intelligence research ,[38]. In 2016, Lee Sedol [39] - the best Go's player in the world was been beaten by AlphaGo 4-1 [40].

MCTS with policy and value networks are at the heart of AlphaGo decision-making process, enabling AlphaGo's to pick the optimal moves in the complex search of Go. [41]

2.3.1 Overview

The MCTS' process is conceptually straightforward. A tree is built in an incremental and assymatric manner (Figure 2.3). For every iteration, a selection policy is used to determine which node to select in the tree to perform simulations. The selection policy, typically balances the exploration (looking into parts of the tree that have not been visited yet) and the exploitation (looking into parts of the trees that appear to be promising). Once the node is selected, a simulation - a sequence of available actions, based on a simulation policy, is applied from this node until a terminal condition is reached e.g no further actions are possible. [42]

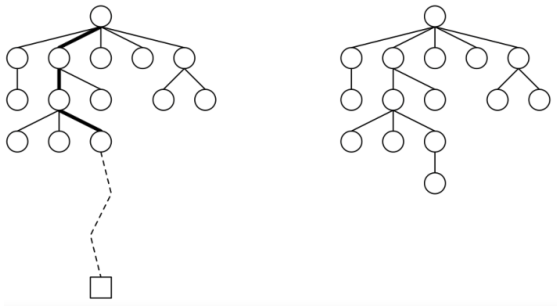


FIGURE 2.3: Assymetrical growth of MCTS - Simulation and Expansion - [3]

To ensure a clearer understanding of MCTS algorithm's stages, we will start by exploring a detailed example [43]. This example will illustrate each component of the algorithm in action. Furthermore, we will generalise the principles discussed, as the methodology of this paper is built on the application of the MCTS algorithm.

2.3.2 Example

Let's say we are given a maximisation problem. When beginning the game, you have two possible actions a_1 and a_2 from the node $S_0^{0,0}$ in the tree $\mathcal{T}$. Every node is defined like so: $S_i^{n_i,t_i}$ where n_i represents the number of times node i has been visited, t_i the total score of this node. Moreover, for every node - we can compute a selection metric, for instance the $UCB1$ value: $UCB1(S_i^{n_i,t_i}) = \bar{V}_i + 2\sqrt{\frac{\ln N}{n_i}}$ where $\bar{V}_i = \frac{n_i}{t_i}$ represents the average value of the node, n_i the number of times node i has been visited, $N = n_0$ the number of times the root node has been visited (which is also equal to the number of iterations).

Before the first iteration, none node have been visited - $\forall i \in \mathcal{T}, S_i^{0,0}$. At the beginning

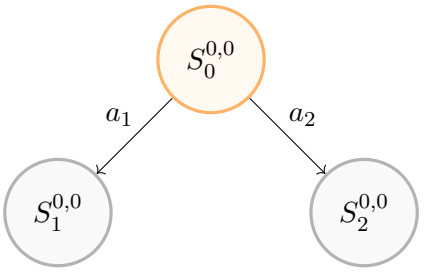


FIGURE 2.4: Selection - $I1$

of $I1$, we have to choose between these two child nodes (or choose between taking a_1 or a_2). After, we have to calculate the $UCB1$ value for these two nodes and pick the node that maximises the $UCB1$ value (as we are dealing with a maximisation problem). In

Figure 2.4, neither of these have been visited yet so $USB(S_1^{0,0}) = UCB1(S_2^{0,0}) = \infty$. Hence we decide to choose randomly $S_1^{0,0}$.

$S_1^{0,0}$ is a leaf node that has not been visited - then we can simulate from this node, which means selecting actions from this node based on the simulation policy to a terminal state as shown on Figure 2.5:

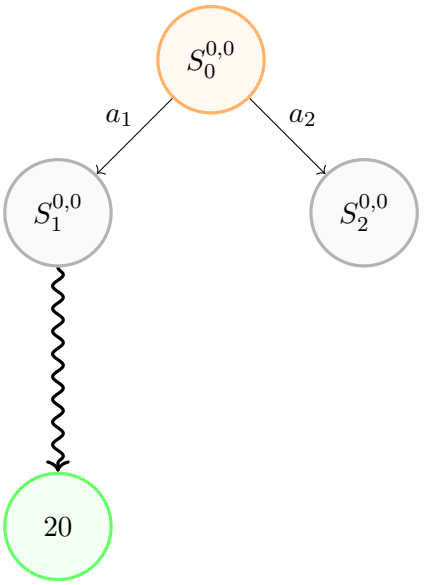


FIGURE 2.5: Simulation - $I1$

The terminal state has a value of 20, we can write that the rollout/simulation from node $S_1^{0,0}$ node is $\mathcal{R}(S_1^{0,0}) = 20$. The final step of $I1$ is backpropagation. Every node that has been visited in the iteration is updated. Let $\mathcal{N}_{\mathcal{R},j}$ be the indexes of the nodes visited during the $j-th$ iteration of the MCTS:

- Before backpropagation:

$$\forall i \in \mathcal{N}_{\mathcal{R},j}, S_{i,old}^{n_i,t_i} \tag{2.1}$$

- After backpropagation:

$$\forall i \in \mathcal{N}_{\mathcal{R},j}, S_{i,new}^{n_i+1,t_i+\mathcal{R}(S_{i,old}^{n_i,t_i})} \tag{2.2}$$

We can then define a backpropagation function:

$$\begin{array}{rccc} \mathcal{B} & : & \mathcal{N}_{\mathcal{R},j} & \rightarrow & \mathcal{N}_{\mathcal{R},j} \\ & & S_i^{n_i,t_i} & \mapsto & S_i^{n_i+1,t_i+\mathcal{R}(S_i^{n_i,t_i})} \end{array}$$

Then, back to the example on Figure 2.6 we update the nodes $\mathcal{B}(S_1^{0,0}) = S_1^{\mathbf{1,20}}$ and $\mathcal{B}(S_0^{0,0}) = S_0^{\mathbf{1,20}}$.

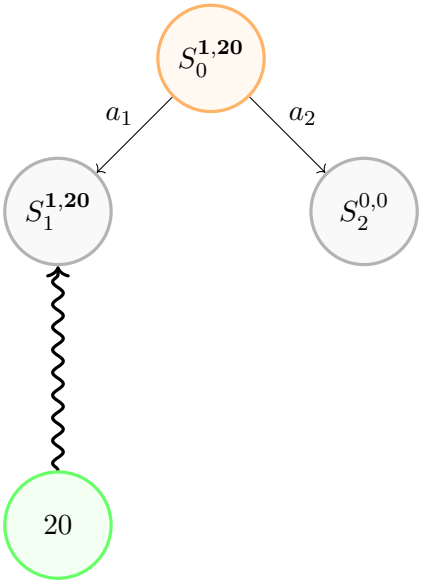
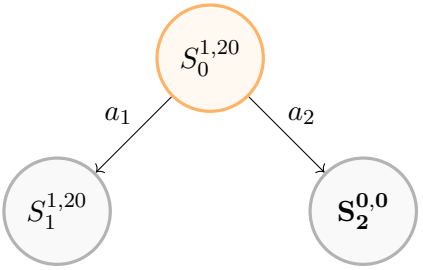


FIGURE 2.6: Backpropagation - I1

The fourth phase of the algorithm has been done for $I1$. Therefore, we can then start the 2^{nd} iteration $I2$. On Figure 2.7, we can either choose a_1 or a_2. When a child node



$$UCB(S_1^{1,20}) = 20 + 2\sqrt{\frac{\ln 1}{1}} = 20$$
$$UCB(S_2^{0,0}) = \infty$$

FIGURE 2.7: Selection - I2

has not been visited yet, you pick this node for the Selection or you can compute the $UCB1$ value, it leads to the same conclusion.

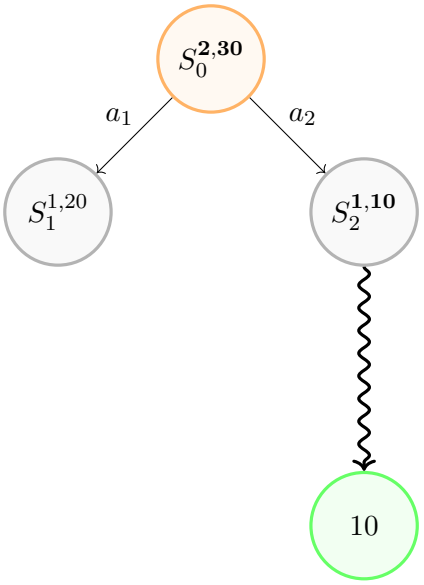


FIGURE 2.8: Simulation and Backpropagation - I2

We can simulate (Figure 2.8) from the chosen node $S_2^{0,0}$ and $\mathcal{R}(S_2^{0,0}) = 10$ and back-propagate all the visited nodes: $\mathcal{B}(S_2^{0,0}) = S_2^{1,10}$ and $\mathcal{B}(S_0^{1,20}) = S_0^{2,30}$. Next, we start the 3^{rd} iteration, based on the $UCB1$ score we decide to choose a_1.

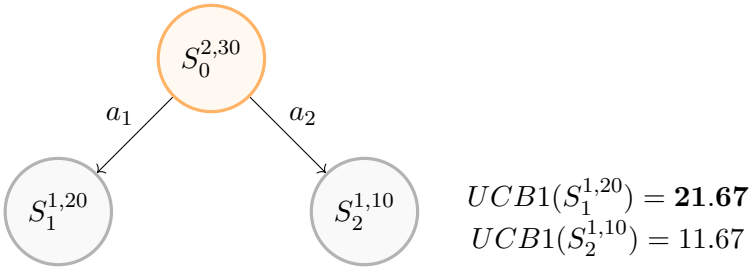


FIGURE 2.9: Selection - I3

$S_1^{1,20}$ is a leaf node and has been visited so we can expand this node.

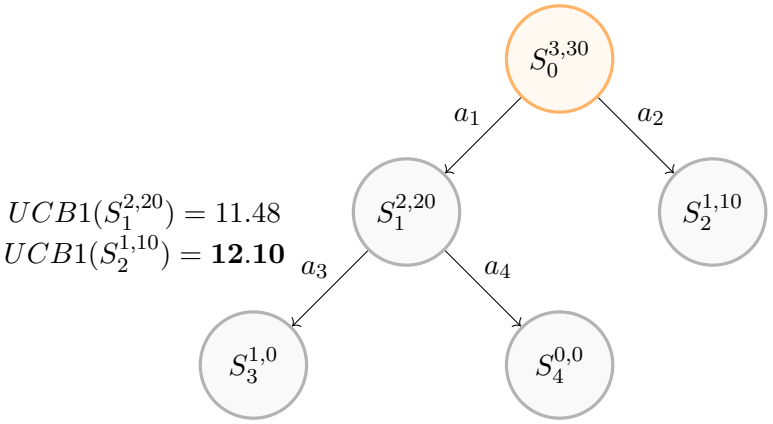


FIGURE 2.10: Selection and Expansion - I3

Based on $UCB1$ score we decide to simulate from $S_3^{0,0}$ on Figure 2.11

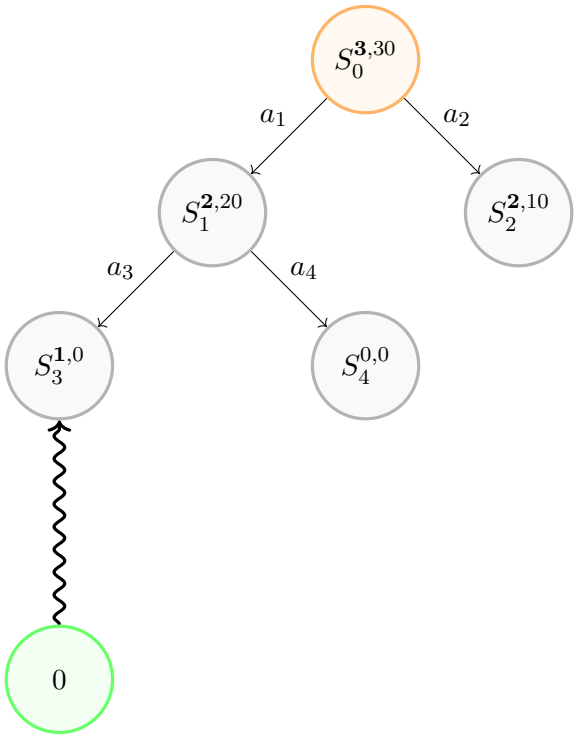


FIGURE 2.11: Simulation and Backpropagation - I3

This is the fourth iteration $I4$ represented on Figure 2.12:

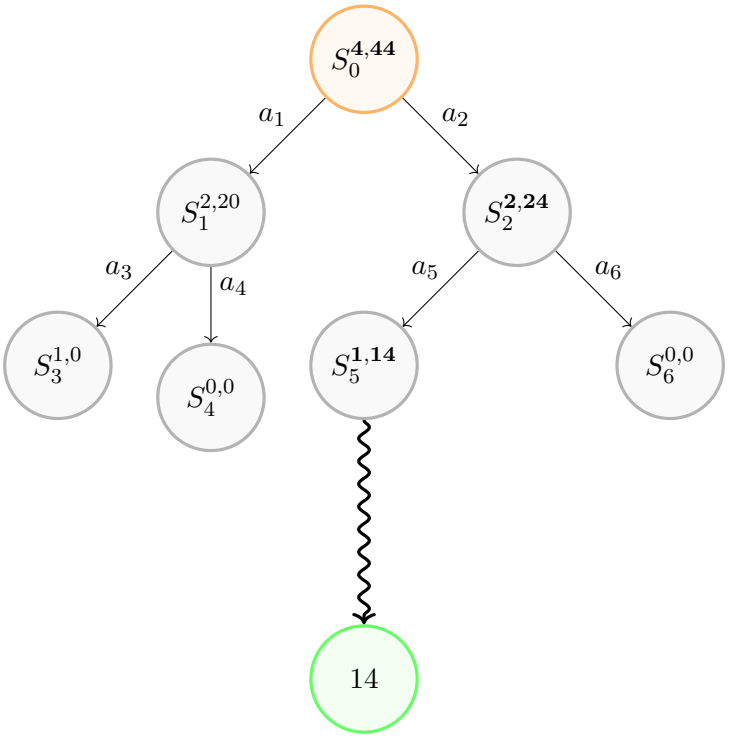


FIGURE 2.12: Selection - Simulation - Backpropagation - I4

The MCTS algorithm can either be stopped because you are running out of time or because you have no more available actions. For instance, if we were to stop at this stage of the algorithm, the best action to undertake is a_2 because it has the higher average value: $\bar{V}_1 = \frac{20}{2} \leq \bar{V}_2 = \frac{24}{2}$.

2.3.3 The different parameters in the MCTS

2.3.4 Selection policy

[44]

- **Upper Confidence Bound (UCB):**
  - *Description*: UCB is a popular selection policy in MCTS that balances exploration and exploitation using a mathematical formulation that considers both the average reward of a node and the uncertainty of that reward.
  - *Trade-offs*:

    - **Exploration vs. Exploitation**: UCB adjusts the balance between exploring less-visited nodes and exploiting nodes with high rewards.
    - **Parameter Sensitivity**: The performance of UCB depends on the constant C_p in its formula, which controls the level of exploration.
    - **Children's Count**: UCB can lead to varying numbers of children being explored, depending on the setting of C_p.

- **UCB1-Tuned:**
  - *Description*: A variation of UCB that dynamically adjusts the exploration term based on the variance of rewards, making it more adaptive to different scenarios.
  - *Trade-offs*:
    - **Adaptivity**: UCB1-Tuned can adapt to the variance in the rewards, potentially leading to better performance in complex environments.
    - **Computational Cost**: The added complexity in adjusting the exploration term may lead to higher computational costs.
    - **Children's Count**: This policy may result in fewer nodes being selected for expansion when the variance is low, focusing more on exploitation.

- **Single-Player (SP-MCTS):**
  - *Description*: A variant of MCTS specifically designed for single-player scenarios, incorporating a third term in the UCB formula to account for the uncertainty in node values.
  - *Trade-offs*:
    - **Exploration of Uncertainty**: SP-MCTS inflates the uncertainty term for less-visited nodes, which can lead to more thorough exploration in single-player settings.
    - **Focus on Strong Lines**: This policy tends to favor strong lines of play, potentially neglecting less promising but necessary exploratory paths.
    - **Children's Count**: Tends to explore a wider range of children nodes, balancing between known strong strategies and potential new solutions.

- **Bayesian UCT:**
  - *Description*: Bayesian UCT integrates Bayesian statistics into the selection policy, allowing for a probabilistic approach to balancing exploration and exploitation.

- *Trade-offs*:
  - **Probabilistic Exploration**: Bayesian UCT provides a more nuanced exploration strategy by considering prior knowledge and updating beliefs as more information is gathered.
  - **Complexity**: The Bayesian approach can be computationally expensive, especially in large search spaces.
  - **Children's Count**: The number of children explored under Bayesian UCT can be influenced by the prior distributions used, potentially leading to more focused exploration in areas of high uncertainty.

2.4 Selection Policies

2.4.1 Single-Player MCTS (SP-MCTS)

Single-Player MCTS (SP-MCTS) is an adaptation of the Monte Carlo Tree Search (MCTS) algorithm specifically designed for single-player games. SP-MCTS modifies the standard Upper Confidence Bounds (UCB) formula by adding a third term to account for the possible deviation of a node's value, adjusting the standard deviation for infrequently visited nodes to better estimate rewards.

The modified UCB formula used in SP-MCTS is:

$$rD\frac{\sigma^2}{n_i} + n_i, \tag{2.3}$$

where σ^2 is the variance of the node's simulation results, n_i is the number of visits to the node, and D is a constant that inflates the standard deviation for less frequently visited nodes.

Key enhancements in SP-MCTS include:

- **Heuristic-Guided Default Policy**: A heuristic is used to guide simulations, improving the efficiency of the search process.
- **Meta-Search**: To avoid getting stuck in local maxima, SP-MCTS periodically restarts with different random seeds and stores the best solution across runs.
- **Maximization Tracking**: By tracking maximum simulation results, SP-MCTS ensures that strong lines of play are not overshadowed by weaker ones.

2.4.2 UCB1-Tuned

UCB1-Tuned is an enhancement of the standard UCB1 algorithm, designed to fine-tune the balance between exploration and exploitation in MCTS. This approach adjusts the exploration term based on sample variance, providing a more accurate trade-off in dynamic environments.

The UCB1-Tuned formula is:

$$\sqrt{\frac{\ln n}{n_j}} \cdot \min\left(1, V_j(n_j)\right), \tag{2.4}$$

where:

$$V_j(n_j) = \frac{1}{2}\sum_{s=1}^{n_j}\left(X_{j,s}^2 - X_j\right) + \frac{\sqrt{2\ln t}}{s}, \tag{2.5}$$

and n_j is the number of times the machine j has been played, $X_{j,s}$ is the reward for each play, and t is the total number of plays.

Notable applications of UCB1-Tuned include:

- **Improved Exploration-Exploitation Trade-off**: UCB1-Tuned replaces the upper confidence bound term with one that accounts for variance, leading to better decision-making.
- **Domain Applications**: The UCB1-Tuned approach has been successfully applied in games like Go and real-time environments, showing superior performance compared to the standard UCB1.

2.4.3 Upper Confidence Bounds for Trees (UCT)

UCT is the foundational algorithm for MCTS, combining Monte Carlo simulations with the UCB1 algorithm for selecting actions during the search process. UCT ensures a balance between exploring new actions and exploiting known good ones.

The UCB1 formula used in UCT is:

$$X_j + C_p\sqrt{\frac{\ln N}{n_j}}, \tag{2.6}$$

where X_j is the average reward from action j, C_p is the exploration constant, N is the total number of simulations, and n_j is the number of times action j has been tried.

Core components of UCT:

- **Exploration-Exploitation Balance**: UCT employs the UCB1 formula to manage the trade-off between exploration and exploitation during tree growth.
- **Tree Growth Mechanism**: UCT builds a search tree dynamically, where nodes are expanded based on the outcomes of simulations, continuously refining the action-value estimates.

2.4.4 Bayesian UCT

Bayesian UCT introduces Bayesian methods into the UCT framework to improve the estimation of node values and their uncertainties, especially in scenarios with limited simulation trials.

The Bayesian tree policy can be represented as:

$$B_i = \mu_i + \frac{\sqrt{2 \ln N}}{n_i}, \tag{2.7}$$

where μ_i is the mean of an extremum (minimax) distribution P_i, and n_i is the number of visits to the node i.

An alternative Bayesian UCT formula is:

$$B_i = \mu_i + \sigma_i \cdot \sqrt{\frac{2 \ln N}{n_i}}, \tag{2.8}$$

where σ_i is the square root of the variance of P_i.

Key strategies in Bayesian UCT:

- **Bayesian Tree Policies**: Two tree policies are proposed—one that maximizes mean rewards and another that maximizes mean plus variance, with the latter often showing superior performance.
- **Improved Convergence**: Bayesian UCT demonstrates better convergence properties compared to standard UCT, particularly when accurate prior information is available.

2.4.4.1 Different nodes

Chapter 3

Problem Description

3.1 Overview

Kiwi's traveler wants to travel in N different areas in N days, let's denote A the set of areas the traveler wants to visit:

$$A = \{A_1, A_2, \dots A_N\}$$

where each A_j is a set of airports in area j:

$$A_j = \{a_{j,1}, a_{j,2}, \dots, a_{j,k_j}\}$$

where a_{j,k_j} being airports in area j and k_j is the number of airports in area j.

The traveler has to visit one area per day. He has to leave this area to visit a new area by flying from the airport he flew in. He leaves from a known starting airport and has to do his journey and come back to the starting area, not necessarly the starting airport. There are flight connections between different airports, with different prices depending on the day of the travel: we can write c_{ij}^d the cost to travel from $city_i$ to $city_j$ on day d. We do not necessarly have $c_{ij}^d = c_{ji}^d$ neither $c_{ij}^{d_1} = c_{ij}^{d_2}$ if $d_1 \neq d_2$. The problem can hence be characterised as an generalised, assymetric and time dependant TSP - as discussed in Section 2.2.

The aim of the problem is to find the cheapest route for the traveler's journey.

The problem itself had not been mathematically defined in previous research, and we found it particularly valuable in our study to rigorously formulate the problem mathematically, as it provided a clear framework to analyse and understand its complexities.

We can then formulate the problem as follow:

- $\mathcal{A} = \{1, 2, \ldots, N\}$: Set of areas.
- $A_j = \{a_{j,1}, a_{j,2}, \ldots, a_{j,k_j}\}$: Set of airports in area $j \in \mathcal{A}$.
- $\mathcal{D} = \{1, 2, \ldots, N\}$: Set of days.
- $U_d \subseteq \mathcal{A}$: Set of areas that have not been visited by the end of day d.

Parameters

- c_{ij}^d: Cost to travel from airport i to airport j on day $d \in \mathcal{D}$.

Variables

- x_{ij}^d: Binary variable which is 1 if the traveler flies from airport i to airport j on day d, and 0 otherwise.
- v_j^d: Binary variable which is 1 if area j is visited on day d, and 0 otherwise.

Constraints

1. Starting and Ending Constraints:

- The traveler starts at the known starting airport S_0.
- The traveler must return to an airport in the starting area on the final day N.

2. Flow Constraints:

- The traveler must leave each area and arrive at the next area on consecutive days, the next area has not been visited yet.
- Ensure that the traveler can only fly into and out of the same airport within an area.

- Ensure each area is visited exactly once.
- Update the unvisited list as areas are visited.

Objective Function

The goal is to minimise the journey's total travel cost:

$$\min \left(\sum_{d=2}^{N-1} \sum_{i \in \bigcup_{k=2}^{N-1} A_k} \sum_{j \in \bigcup_{k=3}^{N} A_k} c_{ij}^{d} x_{ij}^{d} + \sum_{j \in A_1} c_{S_0,j}^{1} x_{S_0,j}^{1} + \sum_{i \in A_N} \sum_{j \in A_1} c_{ij}^{N} x_{ij}^{N} \right)$$

Constraints

- Starting at the known starting airport S_0 at take an existing flight connection:

$$\sum_{j \in A_1} x_{S_0,j}^{1} = 1$$

$$\forall d \in \mathcal{D}, c_{S_0,j}^{d} \in \mathbb{R}^{+*}$$

- Visit exactly one airport in each area each day:

$$\sum_{i \in A_d} \sum_{j \in A_{d+1}} x_{ij}^{d} = 1 \quad \forall d \in \{1, 2, \ldots, N-1\}$$

- Ensure the traveler leaves from the same airport they arrived at the previous day:

$$\sum_{k \in A_d} x_{ik}^{d} = \sum_{k \in A_{d-1}} x_{ki}^{d-1} \quad \forall i \in \bigcup_{j=1}^{N} A_j, \forall d \in \{2, 3, \ldots, N\}$$

- Return to an airport in the starting area on the final day with an existing flight connection:

$$\sum_{i \in A_N} \sum_{j \in A_1} x_{ij}^{N} = 1$$

$$\forall (i,j) \in A_N \times A_1, c_{i,j}^{N} \in \mathbb{R}^{+*}$$

- Ensure each area is visited exactly once:

$$\sum_{d \in \mathcal{D}} v_j^d = 1 \quad \forall j \in \mathcal{A}$$

- Update the unvisited list:

$$v_j^d = 1 \implies j \notin U_d \quad \forall j \in \mathcal{A}, \forall d \in \mathcal{D}$$

- Ensure a flight on day d between i and j exists only if the cost exists and j is in the unvisited areas on day d:

$$x_{ij}^d \leq c_{ij}^d \cdot v_j^d \quad \forall i, j \in (\bigcup_{j=1}^{N} A_j)^2, \forall d \in \mathcal{D}$$

$$x_{ij}^d \leq v_j^d \quad \forall j \in \bigcup_{j=1}^{N} A_j, \forall d \in \mathcal{D}$$

- Binary variable constraints:

$$x_{ij}^d \in \{0,1\} \quad \forall (i,j) \in (\bigcup_{j=1}^{N} A_j)^2, \forall d \in \mathcal{D}$$

$$v_j^d \in \{0,1\} \quad \forall j \in \mathcal{A}, \forall d \in \mathcal{D}$$

3.2 Instances

3.2.1 Description

We are given a set of 14 Instances $I_n = \{I_1, I_2, ..., I_{13}, I_{14}\}$ that we have to solve. Every instances has the same overall structure.

For example, the first few lines of I_4 are:

13 GDN
first
WRO DL1
second
BZG KJ1
third
BXP LB1

That means that the Traveller will visit 13 different areas, he starts from airport GDN, that belongs to the starting area. Then we are given the list of airports that are in every zone. For example, the second zone is named second and has two airports: WRO and DL1.

After all the information regarding the areas and the airports we have the flight connections informations. In Table 3.1, few flights are displayed from I_6 for illustrative purposes.

TABLE 3.1: Flight connections sample I6

Departure from	Arrival	Day	Cost
KKE	BIL	1	19
UAX	NKE	73	16
UXA	BCT	0	141
UXA	DBD	0	112
UXA	DBD	0	128
UXA	DBD	0	110

For every instance I_i, we know what connections exist between two airports for a specific day and the associated cost. There might be in some instances flights connections at day 0, this means these connections exist for every day of the journey at the same price. Furthermore, we could have the same flight connections at a specific day but with different prices. Furthermore, we have to consider solely the more relevant connections i.e. the flight connection with the lowest fare, on 3.1 we only consider the flight from UXA to DDB with the associated cost of 110.

3.2.2 General formulation

We decided to formulate the problem mathematically because it was not done in the existing papers, and we found it useful to clearly understand the problem's instances and their characteristics.

An instance I_i can be mathematically defined as follows:

$$I_i = (N_i, S_{i0}, A_i, F_i)$$

where:

- **Number of Areas** N_i:

  $$N_i \in \mathbb{N}$$

  The total number of distinct areas in instance I_i.

- **Starting Airport** S_{i0}:

  $$S_{i0} \in \text{Airports}$$

  The starting airport of the traveller.

- **Airports in Each Area**:

  $$A_i = \{A_{i,1}, A_{i,2}, \dots A_{i,N_i}\}$$

  where each $A_{i,j}$ is a set of airports in area j for instance i:

  $$A_{i,j} = \{a_{i,j,1}, a_{i,j,2}, \dots, a_{i,j,k_j}\}$$

  with a_{i,j,k_j} being airports in area j and k_j is the number of airports in area j.

- **Flight Connections**:

  $$F_i = \{F_{i,0}, F_{i,1}, F_{i,2}, \dots, F_{i,N_i}\}$$

where each flight matrix $F_{i,k}$ represents the flight information of instance i on day k:

$$F_{i,k} = \begin{pmatrix} a^d_{i,k,1} & a^a_{i,k,1} & f_{i,k,1} \\ a^d_{i,k,2} & a^a_{i,k,2} & f_{i,k,2} \\ \vdots & \vdots & \vdots \\ a^d_{i,k,l_{k,i}} & a^a_{i,k,l_{k,i}} & f_{i,k,l_{k,i}} \end{pmatrix}$$

- **Columns**:
  - Departure Airport: $a^d_{i,k,j}$ (Departure airport for the j-th flight on day k)
  - Arrival Airport: $a^a_{i,k,j}$ (Arrival airport for the j-th flight on day k)
  - Cost: $f_{i,k,j}$ (Cost of the j-th flight on day k), where $j \in [1, l_{k,i}]$
- **Rows**: Each row corresponds to a specific flight on day k. The number of rows $l_{k,i}$ depends on the number of flights available on that day.

3.2.3 Kiwi's rules

When solving all the instances, Kiwi's defined time limits constraints based on the nature of the instance. We can summarise these constraints in the Table above:

TABLE 3.2: Time limits based on the number of areas and airports

Instance	nb areas	Nb Airports	Time limit (s)
Small	≤ 20	< 50	3
Medium	≤ 100	< 200	5
Large	> 100		15

All the useful information about the instances such as the starting airport, the associated area, the range of airports per area, the number of airports and the time limit constraints are defined in Table 3.3.

TABLE 3.3: Instances and their respective parameters

Instances	Starting Area - Airport	N° areas	Min - Max airport per area	N° Airports	Time Limit (s)
I1	Zona_0 - AB0	10	1 - 1	10	3
I2	Area_0 - EBJ	10	1 - 2	15	3
I3	ninth - GDN	13	1 - 6	38	3
I4	Poland - GDN	40	1 - 5	99	5
I5	zone0 - RCF	46	3 - 3	138	5
I6	zone0 - VHK	96	2 - 2	192	5
I7	abfuidmorz - AHG	150	1 - 6	300	15
I8	atrdruwkbz - AEW	200	1 - 4	300	15
I9	fcjsqtmccq - GVT	250	1 - 1	250	15
I10	eqlfrvhlwu - ECB	300	1 - 1	300	15
I11	pbggaefrjv - LIJ	150	1 - 4	200	15
I12	unnwaxhnoq - PJE	200	1 - 4	250	15
I13	hpvkogdfpf - GKU	250	1 - 3	275	15
I14	jjewssxvsc - IXG	300	1 - 1	300	15

Chapter 4

Methodology

(TODO) Describe implementation details

4.1 Monte Carlo Tree Search implementation

4.1.1 General flow

Based on the discussion in Chapter 2, the flow of the Monte Carlo Tree Search algorithm is summarised in Figure 4.1:

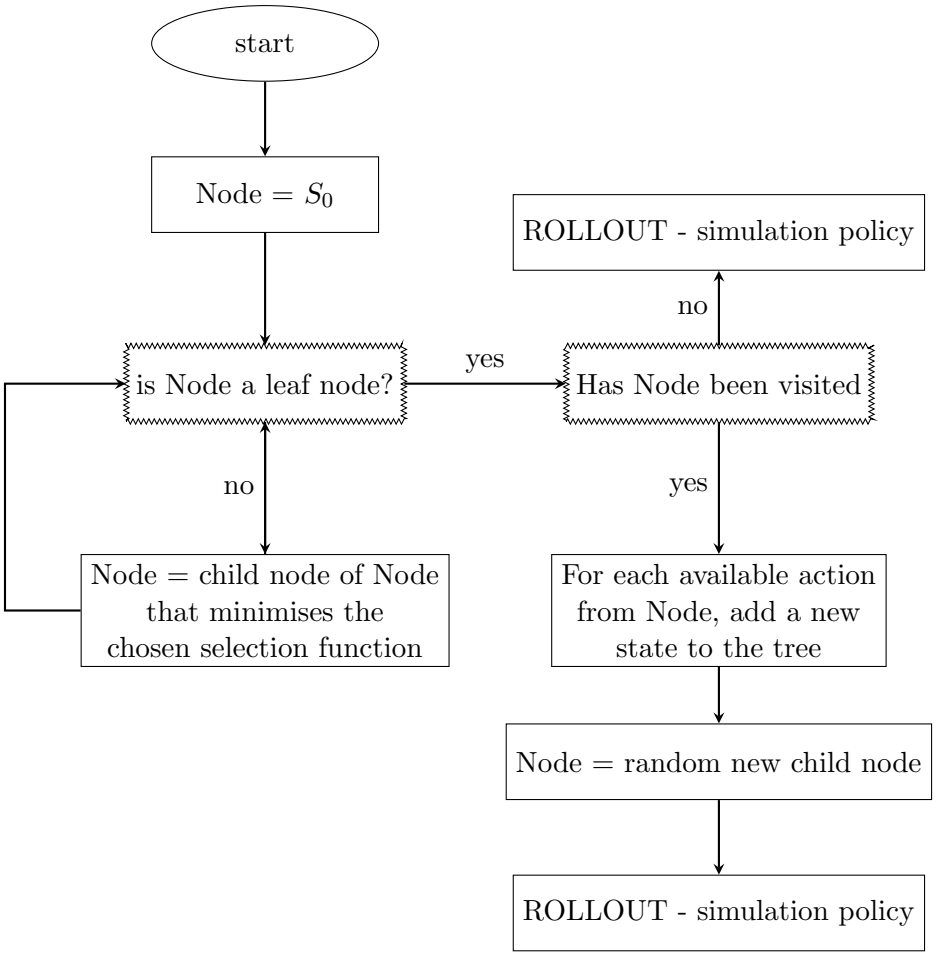


FIGURE 4.1: Flow MCTS

For every iteration of this algorithm - there are four different phases:

1. **Selection:** Starting from the root node (the starting airport S_{i0} for I_i), select successive child nodes (airports that are in unvisited areas) until a leaf node (the airport in the initial area - not necessarily the starting airport) is reached. Use the chosen Selection function to evaluate which node's is the most promising. In the illustrative example in Section 2.3.2, we used the UCB1 function for the selection

function. We were also dealing with a maximisation problem, hence we selected nodes with the highest UCB1 value. A contrario, in Kiwi's minimisation problem, nodes are evaluated based on the lowest value of the selection function.

2. **Expansion:** If the selected node is not a terminal node, expand the tree by adding all possible child nodes.

3. **Simulation:** From the newly added node, perform a simulation (based on the simulation policy) until we reach a terminal node i.e we find a feasible solution.

4. **Backpropagation:** Update the values of the nodes along the path from the newly added node to the root based on the result of the simulation.

$$\mathcal{B}(S_i^{n_i,t_i}) = S_i^{n_i+1,t_i+\mathcal{R}(S_i^{n_i,t_i})} \tag{4.1}$$

where $\mathcal{R}(S_i^{n_i,t_i})$ is the cost of the solution found after performing a simulation from node $S_i^{n_i,t_i}$.

4.1.1.1 Data Preprocessing

In order to implement our MCTS' solution, the first thing to implement was a data_preprocessing class in order to preprocess the given instance. Kiwi's challenge has been solved using Python 3.10 on VS Code 1.92.2. Our Python code is structured using object-oriented programming following CamelCase's convention [45]. This data_preprocessing class is represented on Figure 4.2. The input is an instance I_i, as defined in Chapter 3:

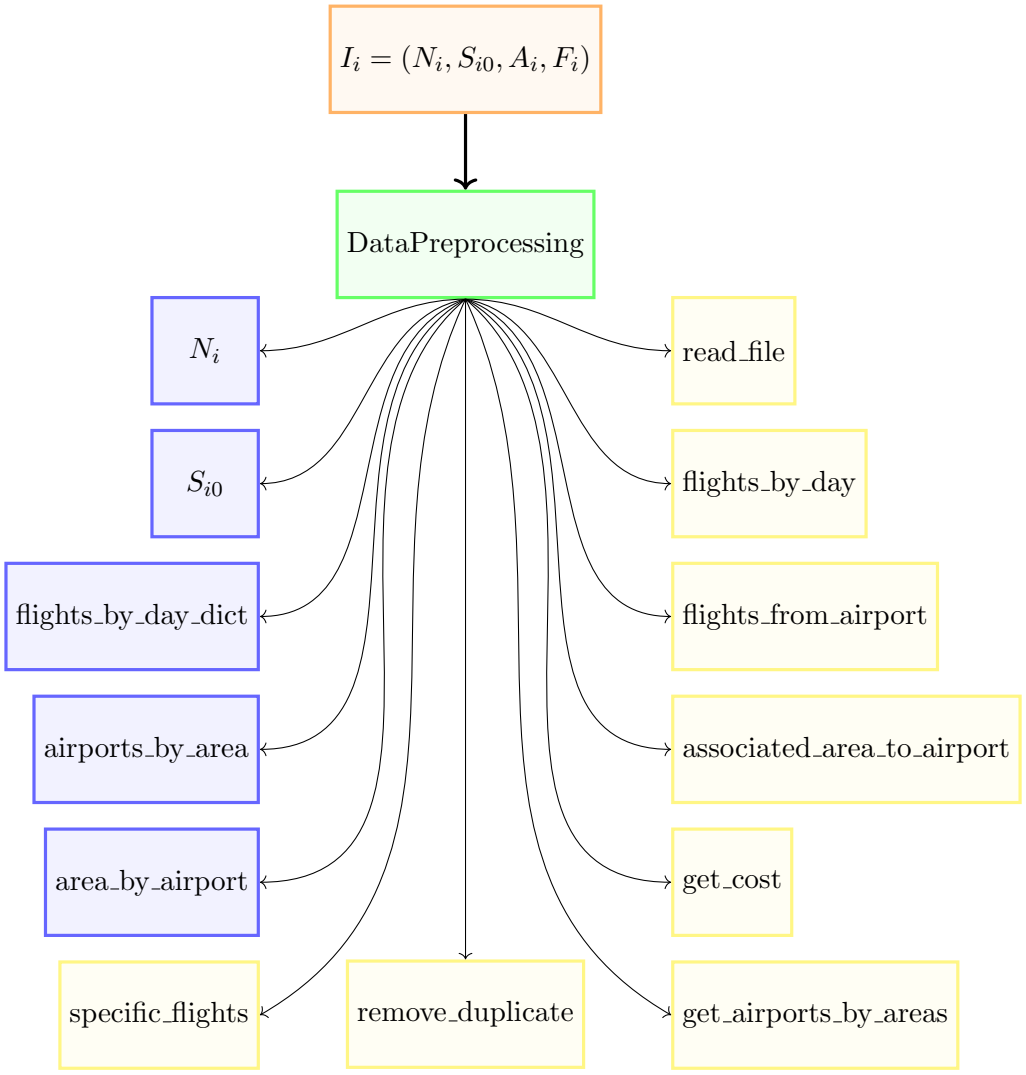


FIGURE 4.2: Explanation of the data preprocessing class

Different useful methods are implemented within the data_preprocessing class to compute and manage various attributes required for the problem at hand. These methods are designed to prepare and structure the data, making it easier to use in subsequent phases

of the algorithm. For example, the remove_duplicate method ensures that only the cheapest flight connections are considered between two airports if multiple flight connections exist at different prices on the same day. Other methods, such as flights_by_day_dict and get_airports_by_areas organise the data. The first method regroup all the flights by their respective days, creating a dictionary where each key represents a day and its corresponding value is a list of available flights. The second is regrouping all the airports present in the different areas. Finally, others methods like specific_flights will be helpful in the algorithm's development gives all the possible flight connections from a specific airport on a specific day considering the visited_areas, it hence gives you all the possibles actions from a node.

Given that Python is relatively slower in terms of computation compared to other programming languages, we opted to use as much as possible dictionaries. Dictionnaries allow for efficient data retrieval based on a key, with an average time complexity of $\mathcal{O}(1)$. This choice enhances the performance of the data preprocessing step, ensuring that the algorithm runs more efficiently despite Python's inherent limitations.

4.1.1.2 Node

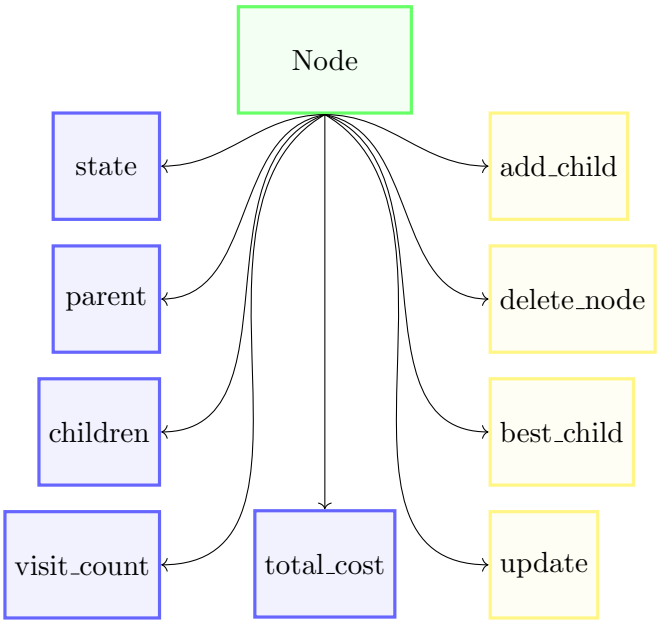


FIGURE 4.3: Explanation of the Node class

As mentionned earlier in Section 2.3.2, we use a Node structure in our algorithm, hence we implemented a Node class. Each Node has a reference to a parent node (unless it is the root node) and may have one or more child nodes (unless it is a leaf node). These relationships form a tree structure where each node can expand into potential future states, guiding the search process. The visit_count tracks the number of times a node has been visited during the MCTS process. This is crucial for evaluating the node's importance and for calculating score of the node with the selection function. The state is a dictionnary that contains node's current information:

- current_airport: The airport where the traveler is at this node.
- current_day: The day of the trip at this node.
- remaining_zones: The zones that still need to be visited to complete the journey.
- visited_zones: The zones that have already been visited to ensure that all zones are visited exactly once during the trip.
- total_cost: It represents the accumulated cost of the current solution path leading to this node.

Additionally, to manage the expansion of child nodes, the add_child method is defined. This method generates new nodes based on the possible actions available from the current node, the potential flight connections from the current airport on this specific day, given the current path taken so far. These new nodes represent the next possible states in the traveler's journey, allowing the search tree to expand and explore different travel routes. Finally, the delete_node method can be used to delete a node from the list of its parent's children.

4.2 The different policies

In the previous section, we outlined the general flow of the MCTS algorithm, focusing on two cores classes, DataPreprocessing and Node, that are central in MCTS' implementation.

In Section 2.4, we explored the various selection policies that guide the decision-making process within MCTS. Although there is a limited litterature review, we decided to parameterise not only the selection policy but also the simulation and expansion policy.

4.2.1 Simulation policies

When you simulate from a given node in the tree, the goal is to find a feasible combinaison of airport that could be a solution to our problem. Then from this chosen node for the simulation, you have the current state (defined in Section 4.1.1.2), so you have to chose for the remaining actions to find a simulated solution based on the simulation policy.

We decided to define three distinct simulation policies:

- Random policy: This policy selects a random action from the set of available actions, introducing variability and exploration in the simulation process.
- Greedy Policy: This policy selects the action that corresponds to the cheapest available flight connection, thus prioritising cost minimisation at each step.
- Tolerance Policy (with coefficient c): This policy selects an action randomly from a subset of actions that are within a certain tolerance level of the minimum cost action. The tolerance level is defined by a coefficient c, allowing for a balance between exploration and exploitation.

The tolerance heuristic is defined as follows:

- Identify the cheapest flight connection among the available actions c_{min}.
- Filter the actions to include only those with a cost less or equal than $c_{min}(1+c)$.
- Randomly select an action from this filtered set.

4.2.2 Expansion policies

When expanding a node, it's theoretically possible to expand all available child nodes i.e. add to the tree all the possible flight connections from this airport (that are in the available actions based on the visited areas). However, in practice, this can be computationally expensive and time-consuming, particularly in problems with a large number of possible actions. To address this, heuristic approaches often involve compromises that enhance the efficiency of the search process by selectively expanding certain nodes rather than all possible ones.

Firstly, we defined `number_of_children`, a parameter of our MCTS - it regulates the maximum number of children that can be expanded from any given node. This limitation controls the size of the search tree, expanding too many children for every selected node could make the algorithm computationally exhaustive.

In our implementation we defined two expansion policies:

- **Top-K Actions Policy**: This policy expands the nodes corresponding to the cheapest flight connections available. Specifically, it sorts all possible actions based on their associated costs and selects the top k actions with the lowest costs, where k is regulated by `number_of_children`. This approach ensures that only the most promising actions, in terms of cost efficiency, are considered during expansion. This policy narrow down the search space but can reach to local optima.
- **Ratio Best-Random Policy**: This policy takes a more balanced approach by combining the selection of the best actions with a degree of randomness. First, it calculates the number of top actions to select based on a predefined ratio, $c \in [0, 1]$, which reflects the proportion of Top-K Actions within the allowed `number_of_children`. After selecting these best actions, the policy randomly selects $(1 - c) * number\_of\_children$ actions from the remaining pool to reach the

desired number of children. This policy is designed to explore a broader range of possibilities while still prioritising cost-effective options.

4.2.3 MCTS' Pseudo-code

In this section, we go into more detail about how we implemented the algorithm in practice by examining the different functions of our MCTS class. The main idea is:

Algorithm 1 Monte_Carlo_Tree_Search

```
1: Initialise Root_Node with Initial_State
2: while Tree is not fully explored do
3:    Node ← Select(Root_Node)
4:    if Node is not fully expanded then
5:       Node ← Expand(Node)
6:    end if
7:    Cost ← Simulate(Node)
8:    Backpropagate(Node, Cost)
9: end while
10: return Best_Leaf_Node
```

The `Select` function returns two arguments: a boolean and a node. The boolean indicates to the expansion function whether expansion is necessary (True means no expansion needed, False means yes):

Algorithm 2 Select_Function

Input: $Node$
$Current \leftarrow Node$
while $Current.Children$ is not empty **do**
  if Current is not fully expanded **then**
    $UnvisitedChildren \leftarrow$ Children with $VisitCount = 0$
    if $UnvisitedChildren$ is not empty **then**
      $SelectedChild \leftarrow$ Randomly select from $UnvisitedChildren$
      return $True, SelectedChild$
    end if
  else
    $Current \leftarrow$ BestChild($Current$)
  end if
end while
if $Current.Children$ is empty **and** $Current.State$["$current\_day$"] == N_{Areas} **then**
  return $False, Current$
else if $Current.Children$ is empty **and** $Current.State$["$current\_day$"] == N_{Areas} **then**
  return $False, Current$
else if $Current.State$["$current\_day$"] == $N_{Areas} + 1$ **then**
  return $True, Current$
end if

We backpropagate the node using the update method of the node. The new node becomes the parent of this node, and we do that until `Node` is `None`, i.e., we have backpropagated all the information up to the root node.

Algorithm 3 Backpropagate_Function

while $Node$ is not $None$ **do**
  $Node.Update(Cost)$
  $Node \leftarrow Node.Parent$
end while

The transition function modifies the states of a node by updating the current airport, the visited zones, remaining zones, etc.

Algorithm 4 Transition_Function

1: $New\_State \leftarrow$ Copy of $State$
2: $New\_State.Current\_Day \leftarrow State.Current\_Day + 1$
3: $New\_State.Current\_Airport \leftarrow Action[0]$
4: $New\_State.Total\_Cost \leftarrow State.Total\_Cost + Action[1]$
5: Update($New\_State.Path$, $New\_State.Current\_Airport$)
6: Remove_Visited($New\_State.Remaining\_Zones$, $New\_State.Current\_Airport$)
7: Add_Visited($New\_State.Visited\_Zones$, $New\_State.Current\_Airport$)
8: **return** New_State

Finally, the Best Child function, defined in the Node class is based on the selection function UCB, UCB1_Tuned, SP and Bayesian, computes the score of the visited nodes and pick the one that minimises the selection function.

Algorithm 5 Best Child

Require: $Selection\_Function$
1: $Visited\_Children \leftarrow$ Children with $visitCount > 0$
2: $Choices\_Weights \leftarrow [Selection\_Function(child)$ for child in $Visited\_Children]$
3: $Best\_Child\_Node \leftarrow$ Child with minimum $Choices\_Weights$
4: **return** $Best\_Child\_Node$

Chapter 5

Results and performance

(TODO) Present the results and discuss any differences between the findings and your initial predictions/hypothesis

(TODO) Interpret your experimental results - do not just present lots of data and expect the reader to understand it. Evaluate what you have achieved against the aims and objectives you outlined in the introduction

5.1 Hypothesis

TABLE 5.1: State of the Art Solution

Instance	Kiwi's	RL	Best known	Best found
I1	1396	1396	1396	1396
I2	1498	1498	1498	1498
I3	7672	7672	7672	7672
I4	14024	13952	13952	-
I5	698	690	690	-
I6	2159	2610	2159	-
I7	31681	30937	30937	-
I8	4052	4081	4052	-
I9	76372	75604	75604	-
I10	21667	58304	21667	-
I11	44153	59361	44153	-
I12	65447	86074	65447	-
I13	97859	166543	97859	-
I14	118811	198787	118811	-

Table 5.2: I4

Expansion policy	Number childrens	Desired simulation policy	Desired selection policy	Ratio expansion	Cp	Best node cost	Time to find the solution	Mean	Median	Std
top_k	5	greedy_policy	UCB	0.5	1.41	31698	464.2422	31698	31698	0.00
top_k	5	greedy_policy	SP	0.5	1.41	31884	519.5144	32129.67	32131	129.24
top_k	10	greedy_policy	UCB	0.5	1.41	31924	353.9113	31924	31924	0.00
top_k	20	greedy_policy	UCB	0.5	1.41	32237	937.3664	32237	32237	0.00
top_k	15	greedy_policy	UCB	0.5	1.41	32265	942.9624	32265	32265	0.00

Chapter 6

Conclusion

(TODO) Explain what conclusions you have come to as a result of doing this work. Lessons learnt and what would you do different next time. Please summarise the key recommendations at the end of this section, in no more than 5 bullet points.

6.1 Summary of Work

6.2 Critics

6.3 Future Work

(TODO) The References section should include a full list of references. Avoid having a list of web sites. Examiners may mark you down very heavily if your references are mainly web sites.

Chapter 7

Progress and next steps

Selec policy	Exp policy	Simu policy	N° childrens	Ratio	Cp	Best cost	Mean	Std	T(s)
UCB	ratio k	tolerance	10.0	0.0	1.4	1396.0	1396.0	0.0	0.0

Chapter 8

Code Listings

8.1 Data preprocessing

```
import numpy as np
from copy import deepcopy


class data_preprocessing:
    def __init__(self, instance_path):
        self.instance_path = instance_path

        self.info, self.flights = self.read_file(f_name=self.instance_path)
        self.number_of_areas, self.starting_airport = (
            int(self.info[0][0]),
            self.info[0][1],
        )

        self.flights_by_day_dict =
self.flights_by_day(flight_list=self.flights)

        self.flights_by_day_dict = self.remove_duplicate(
            flights_by_day=self.flights_by_day_dict
        )

        self.list_days = [k for k in range(1, self.number_of_areas)]

        self.airports_by_area = self.get_airports_by_areas()
```

```
            self.area_to_explore = self.which_area_to_explore(
                airports_by_area=self.airports_by_area
            )
            self.area_by_airport =
    self.invert_dict(original_dict=self.airports_by_area)

            self.starting_area = self.associated_area_to_airport(
                airport=self.starting_airport
            )
            self.list_airports = self.get_list_of_airports()
            self.list_areas = list(self.airports_by_area.keys())
            self.areas_connections_by_day = (
                self.possible_flights_from_zone_to_zone_specific_day()
            )

        def read_file(self, f_name):
            dist = []
            line_nu = -1
            with open(f_name) as infile:
                for line in infile:
                    line_nu += 1
                    if line_nu == 0:
                        index = int(line.split()[0]) * 2 + 1
                    if line_nu >= index:
                        temp = line.split()
                        temp[2] = int(temp[2])
                        temp[3] = int(temp[3])
                        dist.append(temp)
                    else:
                        dist.append(line.split())
                info = dist[: int(dist[0][0]) * 2 + 1]
                flights = dist[int(dist[0][0]) * 2 + 1 :]
            return info, flights

        def flights_by_day(self, flight_list):
            # Create an empty dictionary to hold flights organized by day
            flights_by_day = {}

            # Iterate over each flight in the input list
            for flight in flight_list:
                # Extract the day from the flight entry
```

```
                day = flight[2]

                # Create a flight entry without the day
                flight_without_day = flight[:2] + flight[3:]

                # Add the flight to the corresponding day in the dictionary
                if day not in flights_by_day:
                    flights_by_day[day] = []
                flights_by_day[day].append(flight_without_day)

            return flights_by_day

        def flights_from_airport(self, flights_by_day, from_airport,
    considered_day):
            flights_from_airport = []
            for day, flights in flights_by_day.items():
                if day == considered_day:
                    for flight in flights:
                        if flight[0] == from_airport:
                            flights_from_airport.append(flight)
                    return flights_from_airport
                else:
                    return None

        def invert_dict(self, original_dict):
            inverted_dict = {}
            for key, value_list in original_dict.items():
                for value in value_list:
                    if value in inverted_dict:
                        inverted_dict[value].append(key)
                    else:
                        inverted_dict[value] = key
            return inverted_dict

        def get_cost(self, day, from_airport, to_airport):
            # Retrieve flights for the specified day and day 0
            flights_day = self.flights_by_day_dict.get(day, [])
            flights_day_0 = self.flights_by_day_dict.get(0, [])

            # Find the cost for the specified day
            cost_day = next(
```

```
                (
                    flight[2]
                    for flight in flights_day
                    if flight[0] == from_airport and flight[1] == to_airport
                ),
                float("inf"),
            )

            # Find the cost for day 0
            cost_day_0 = next(
                (
                    flight[2]
                    for flight in flights_day_0
                    if flight[0] == from_airport and flight[1] == to_airport
                ),
                float("inf"),
            )

            # Return the minimum cost if either exists, otherwise inf
            if cost_day == float("inf") and cost_day_0 == float("inf"):
                return float("inf")

            return min(cost_day, cost_day_0)

        def possible_flights_from_zone_to_zone_specific_day(self):
            areas_connections_by_day = {}

            for day, flights in self.flights_by_day_dict.items():
                areas_connections_list = []

                for flight in flights:
                    connection = f"{self.area_by_airport.get(flight[0])} to
    {self.area_by_airport.get(flight[1])}"
                    if connection not in areas_connections_list:
                        areas_connections_list.append(connection)

                areas_connections_by_day[day] = areas_connections_list

            return areas_connections_by_day

        def get_airports_by_areas(self):
```

```
        area_num = int(self.info[0][0])
        return {f"{i}": self.info[2 + i * 2] for i in range(0, area_num)}

    def get_list_of_airports(self):
        unique_airports = set()

        # Iterate through each sublist and add elements to the set
        for sublist in self.airports_by_area.values():
            for airport in sublist:
                unique_airports.add(airport)

        return list(unique_airports)

    def associated_area_to_airport(self, airport):
        return next(
            (
                area
                for area, airports in self.airports_by_area.items()
                if airport in airports
            ),
            "Airport not found",
        )

    def remove_duplicate(self, flights_by_day):
        for day, flights in flights_by_day.items():
            unique_flights = {}
            for flight in flights:
                flight_key = (flight[0], flight[1])
                if flight_key not in unique_flights:
                    unique_flights[flight_key] = flight
                else:
                    if flight[2] < unique_flights[flight_key][2]:
                        #
print(flight[0],flight[1],flight[2],flight_key,unique_flights[flight_key][2])
                        unique_flights[flight_key] = flight
                flights_by_day[day] = list(unique_flights.values())
        return flights_by_day

    def possible_flights_from_an_airport_at_a_specific_day(self, day,
from_airport):
        daily_flights = self.flights_by_day_dict.get(day, [])
```

```
            flights_from_airport = []
            for flight in daily_flights:
                if flight[0] == from_airport:

                    flights_from_airport.append([flight[1], flight[2]])

            return flights_from_airport

        def
    possible_flights_from_an_airport_at_a_specific_day_with_previous_areas(
            self, day, from_airport, visited_areas
        ):
            daily_flights = self.flights_by_day_dict.get(
                day, []
            ) + self.flights_by_day_dict.get(0, [])
            flights_from_airport = []
            for flight in daily_flights:
                # print(self.associated_area_to_airport(airport=flight[0]))
                if (flight[0] == from_airport) and (
                    self.associated_area_to_airport(airport=flight[1]) not in
    visited_areas
                ):

                    flights_from_airport.append([flight[1], flight[2]])

            return flights_from_airport

        def which_area_to_explore(self, airports_by_area):
            return list(
                {
                    key: len(value)
                    for key, value in airports_by_area.items()
                    if len(value) > 1
                }
            )
```

8.2 Node

```
import numpy as np
import random
from scipy.stats import (
kstest,
norm,
beta,
expon,
gamma,
lognorm,
weibull_min,
uniform,
pareto,
t,
chi2,
)


class Node:
def __init__(self, state, desired_selection_policy, cp, parent=None):
self.cp = cp
self.desired_selection_policy = desired_selection_policy
self.state = state  # State is a dictionary representing the current situation
self.parent = parent  # Parent node
self.children = []  # List of child nodes
self.visit_count = 0  # Number of times this node has been visited
self.total_cost = 0  # Total cost accumulated in simulations from this node
self.scores = []

def add_child(self, child_state):
child_node = Node(
state=child_state,
desired_selection_policy=self.desired_selection_policy,
cp=self.cp,
parent=self,
)
self.children.append(child_node)
return child_node
```

```
def is_fully_expanded(self):
# if self.parent is None:
#    return False
return len(self.children) > 0 and all(
child.visit_count > 0 for child in self.children
)

def update(self, result):
self.visit_count += 1
self.total_cost += result
self.scores.append(result)

def UCB(self, c_param):
epsilon = 0

visited_children = [child for child in self.children if (child.visit_count >
    0)]

sorted_children = sorted(
visited_children,
key=lambda child: child.total_cost / (child.visit_count + epsilon),
)
scores = {child: rank + 1 for rank, child in enumerate(sorted_children)}
total_scores = sum(scores.values())

def normalized_score(child):
return scores[child] / total_scores

choices_weights = [
normalized_score(child)
+ c_param
* (2 * np.log(self.visit_count) / (child.visit_count + epsilon)) ** 0.5
for child in visited_children
]

best_child_node = self.children[np.argmin(choices_weights)]

return best_child_node

def SP(self):
```

```
visited_children = [child for child in self.children if child.visit_count > 0]
D = 1

def sp_mcts_score(child):
mean_cost = np.mean(child.scores) if len(child.scores) > 0 else 0
variance = np.var(child.scores) if len(child.scores) > 0 else 0
possible_deviation = np.sqrt(variance + (D / child.visit_count))
return mean_cost - self.cp * possible_deviation

choices_weights = [sp_mcts_score(child) for child in visited_children]

best_child_node = self.children[np.argmin(choices_weights)]
return best_child_node

def Bayesian(self):
visited_children = [child for child in self.children if child.visit_count > 0]
N = self.visit_count

def bayesian_uct_score(child, use_variance=False):
mean_cost = np.mean(child.scores) if len(child.scores) > 0 else 0
exploration_term = np.sqrt(2 * np.log(N) / child.visit_count)

if use_variance:
variance = np.sqrt(np.var(child.scores)) if len(child.scores) > 0 else 0
exploration_term *= variance

return mean_cost + exploration_term

# Select which Bayesian UCT formula to use
use_variance = True  # Change this to `False` to use the first formula
choices_weights = [
bayesian_uct_score(child, use_variance=use_variance)
for child in visited_children
]

best_child_node = self.children[np.argmin(choices_weights)]
return best_child_node

def UCB1_tuned(self, c_param):
visited_children = [child for child in self.children if child.visit_count > 0]
```

```
def ucb1_tuned_score(child):
mean_cost = np.mean(child.scores) if len(self.scores) > 1 else 0
variance = np.var(self.scores) if len(self.scores) > 1 else 0
# UCB1-Tuned formula
exploration_term = np.sqrt(
(np.log(self.visit_count) / child.visit_count)
* min(
0.25,
variance
+ np.sqrt(2 * np.log(self.visit_count) / child.visit_count),
)
)
return mean_cost + c_param * exploration_term

choices_weights = [ucb1_tuned_score(child) for child in visited_children]

best_child_node = self.children[np.argmin(choices_weights)]
return best_child_node

def thompson_sampling(self, c_param):
visited_children = [child for child in self.children if child.visit_count > 0]

def best_fit_distribution(scores):
distributions = {
"normal": norm,
"beta": beta,
"exponential": expon,
"gamma": gamma,
"lognormal": lognorm,
"weibull_min": weibull_min,
"uniform": uniform,
"pareto": pareto,
"t": t,
"chi2": chi2,
}
p_values = {}
for dist_name, dist in distributions.items():
try:
params = dist.fit(scores)
d_statistic, p_value = kstest(scores, dist_name, args=params)
p_values[dist_name] = p_value
```

```
except Exception as e:
p_values[dist_name] = (
0  # Handle the error and skip this distribution
)
print(f"Skipping {dist_name} due to fitting issues: {e}")

best_dist_name = max(p_values, key=p_values.get)
best_p_value = p_values[best_dist_name]

if best_p_value < 0.05:
return None, None

best_dist = distributions[best_dist_name]
best_params = best_dist.fit(scores)

return best_dist, best_params

sampled_values = []
for child in visited_children:
if len(child.scores) > 1:
best_dist, best_params = best_fit_distribution(child.scores)
if best_dist is not None:
sampled_value = best_dist.rvs(*best_params)
sampled_values.append(sampled_value)
else:
return self.UCB(
c_param
)  # Fallback to UCB if no good distribution is found
else:
sampled_values.append(np.mean(child.scores))

best_child_node = visited_children[np.argmin(sampled_values)]
return best_child_node

def randomized_ucb(self, c_param, random_factor=0.1):
visited_children = [child for child in self.children if child.visit_count > 0]

def randomized_ucb_score(child):
mean_cost = np.mean(child.scores) if len(child.scores) > 0 else 0
exploration_term = np.sqrt(
(2 * np.log(self.visit_count) / (child.visit_count))
```

```
)
random_term = random_factor * np.random.rand()
return mean_cost + c_param * exploration_term + random_term

choices_weights = [randomized_ucb_score(child) for child in visited_children]

best_child_node = visited_children[np.argmin(choices_weights)]
return best_child_node

def epsilon_greedy(self, epsilon):
visited_children = [child for child in self.children if child.visit_count > 0]

if np.random.rand() < epsilon:
# Explore: randomly select a child
best_child_node = np.random.choice(visited_children)
else:
# Exploit: select the child with the best average cost
best_child_node = min(
visited_children,
key=lambda child: (
np.mean(child.scores) if len(child.scores) > 0 else float("inf")
),
)

return best_child_node

def best_child(self):
if self.desired_selection_policy == "UCB":
return self.UCB(c_param=self.cp)
if self.desired_selection_policy == "UCB1T":
return self.UCB1_tuned(c_param=self.cp)
if self.desired_selection_policy == "SP":
return self.epsilon_greedy(self.cp)
if self.desired_selection_policy == "Bayesian":
return self.Bayesian()

else:
raise ValueError(
f"Unknown Selection policy: {self.desired_selection_policy}"
)
```

```
def delete_node(self):
self.parent.children = [
child for child in self.parent.children if child != self
]
```

8.3 MCTS

```
    import numpy as np
import random
from copy import deepcopy
import logging
import time
import os
import shutil
import glob

from Data_Preprocessing import data_preprocessing
from Node import Node


class MCTS(data_preprocessing):
    def __init__(
        self,
        instance,
        instance_number,
        number_childrens,
        desired_expansion_policy,
        ratio_expansion,
        desired_simulation_policy,
        desired_selection_policy,
        cp,
        number_simulation,
    ):
        self.instance_number = instance_number
        self.number_childrens = number_childrens
        self.desired_simulation_policy = desired_simulation_policy
        self.desired_expansion_policy = desired_expansion_policy
        self.ratio_expansion = ratio_expansion
        self.number_simulation = number_simulation
        self.desired_selection_policy = desired_selection_policy
        self.cp = cp

        self.expanded_nodes = []
        self.simulations_dict = {}

        self.start_time = time.time()
```

```
        super().__init__(instance_path=instance)
        self.end_time_data_preprocessing = time.time() - self.start_time
        self.simulation()
        # self.organise_log_files_in_folder(
        #     folder_path=os.path.dirname(self.instance_path)
        # )
        # self.collect_all_nodes()

    def configure_logging(self):
        log_file =
    f"{self.instance_path}_{self.number_childrens}_{self.desired_simulation_policy}_{self.desi
        log_file = self.get_unique_log_file(log_file)

        # Clear any existing handlers
        for handler in logging.root.handlers[:]:
            logging.root.removeHandler(handler)

        # Configure the logger
        logging.basicConfig(
            level=logging.DEBUG,  # Set the log level to DEBUG to capture all
    types of logs
            format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
            handlers=[
                logging.FileHandler(
                    log_file, mode="w"
                ),  # 'w' to overwrite the log file each run, 'a' to append
                # logging.StreamHandler(),  # Optional: to also print logs to
    the console
            ],
        )
        logger = logging.getLogger(__name__)
        return logger

    def get_unique_log_file(self, base_log_file):
        """
        Check if the log file exists and if so, create a new file with a
    unique suffix.
        """
        base_name, extension = os.path.splitext(base_log_file)
        counter = 0  # Start with 0 to have the first file as _0
        while True:
```

```
            new_log_file = f"{base_name}_{counter}{extension}"
            if not os.path.exists(new_log_file):
                return new_log_file
            counter += 1

    def organise_log_files_in_folder(self, folder_path):
        """
        Organize log files in the specified folder by moving files with the
    same base name into a dedicated directory.

        :param folder_path: Path to the folder containing the log files.
        """
        # Change to the target directory
        os.chdir(folder_path)

        # Find all log files in the directory
        log_files = glob.glob("*.log")

        # Track which files have already been moved to avoid duplication
        processed_bases = set()

        for log_file in log_files:
            # Extract the base name (up to the first '_')
            base_name = (
                log_file.rsplit("_", 1)[0]
                if "_" in log_file
                else log_file.rsplit(".", 1)[0]
            )

            if base_name not in processed_bases:
                # Mark this base as processed
                processed_bases.add(base_name)

                # Create a pattern to match all similar files
                pattern = f"{base_name}*.log"

                # Find all files matching this pattern
                matching_files = glob.glob(pattern)

                if matching_files:
                    # Create a directory for these files
```

```
                folder_name = os.path.join(folder_path, base_name)
                os.makedirs(folder_name, exist_ok=True)

                # Move each matching file into the directory
                for file in matching_files:
                    shutil.move(file, folder_name)

                # print(
                #     f"Moved files with base '{base_name}' into folder:
{folder_name}"
                # )

    def initialise_root_node(self):
        return {
            "current_day": 1,
            "current_airport": self.starting_airport,
            "remaining_zones": [
                x for x in self.list_areas if x != self.starting_area
            ],  # Exclude the starting area
            "visited_zones": [self.starting_area],  # Exclude the starting area
            "total_cost": 0,
            "path": [self.starting_airport],
        }

    def transition_function(self, state, action):
        new_state = deepcopy(state)
        new_state["current_day"] += 1
        new_state["current_airport"] = action[0]
        new_state["total_cost"] += action[1]
        new_state["path"].append(action[0])
        # self.logger.info(
        #     f"Airport {action[0]},
{self.associated_area_to_airport(airport=action[0])} to remove in
{new_state['remaining_zones']}"
        # )
        new_state["remaining_zones"].remove(
            self.associated_area_to_airport(airport=action[0])
        )
        new_state["visited_zones"].append(
            self.associated_area_to_airport(airport=action[0])
        )
```

```
        return new_state

    def random_policy(self, actions):

        if not actions:
            return None
        return random.choice(actions)

    def greedy_policy(self, actions):

        # self.logger.info(f"Actions: {actions}")
        if not actions:
            return None
        # Select the action with the lowest cost
        best_action = min(actions, key=lambda x: x[1])
        # self.logger.info(f"Chosen action based on heuristic policy:
{best_action}")
        return best_action

    def tolerance_heuristic_policy(self, actions):
        # self.logger.info(f"Actions: {actions}")

        if not actions:
            return None

        # Find the minimum cost
        min_cost = min(actions, key=lambda x: x[1])[1]

        # Filter actions within the tolerance level
        best_actions = [
            action
            for action in actions
            if action[1] <= min_cost * (1 + self.ratio_expansion)
        ]

        # Select a random action from the best actions
        best_action = random.choice(best_actions)

        # self.logger.info(f"Chosen action based on tolerance policy:
{best_action}")
```

```
        return best_action

    def get_unvisited_children(self, node):
        queue = [node]
        unvisited_children = []
        while queue:
            current_node = queue.pop(0)
            for child in current_node.children:
                if child.visit_count == 0:
                    unvisited_children.append(child)
                else:
                    queue.append(child)

        return unvisited_children

    def backpropagate(self, node, cost):
        while node is not None:

            node.update(cost)

            # self.logger.info(
            #    f"Backpropagating Node: {node.state}, Visit Count:
{node.visit_count}, Total Cost: {node.total_cost}, Scores: {node.scores}"
            # )

            node = node.parent

    def collect_all_nodes(self):
        nodes = []
        queue = [self.root]
        while queue:
            node = queue.pop(0)
            nodes.append(node)
            queue.extend(node.children)
        return nodes

    def get_final_nodes(self):
        day = self.number_of_areas + 1
        nodes = [
            node
            for node in self.collect_all_nodes()
```

```
            if node.state.get("current_day") == day
        ]

        # Initialize variables to track the best nodes for this day
        min_cost_child = None
        robust_child = None
        min_cost_robust_child = None
        secure_child = None

        # Values to compare against
        min_cost = float("inf")
        max_visit_count = -float("inf")
        max_secure_value = -float("inf")

        for node in nodes:
            # Min-Cost Child: Select the root child with the lowest total_cost
            if node.total_cost < min_cost:
                min_cost = node.total_cost
                min_cost_child = node

            # Robust Child: Select the most visited root child (visit_count)
            if node.visit_count > max_visit_count:
                max_visit_count = node.visit_count
                robust_child = node

            # Min-Cost-Robust Child: Among the nodes with the lowest
total_cost, select the one with the highest visit_count
            if node.total_cost == min_cost and node.visit_count >=
max_visit_count:
                min_cost_robust_child = node

            # Secure Child: Select the child that minimizes a lower confidence
bound
            if (
                node.visit_count > 0
                and node.parent is not None
                and node.parent.visit_count > 0
            ):
                secure_value = (node.total_cost / node.visit_count) - self.cp
* (
                    (node.parent.visit_count / node.visit_count) ** 0.5
```

```
                )
                if secure_value > max_secure_value:
                    max_secure_value = secure_value
                    secure_child = node

        # Logging the results for the current day
        if min_cost_child:
            self.logger.info("\n\n")
            self.logger.info(f"Best Node: {min_cost_child.state}")
        if robust_child:
            self.logger.info(
                f"Robust Child (Day {day}): State={robust_child.state}, Visit
Count={max_visit_count}"
            )
        if min_cost_robust_child:
            self.logger.info(
                f"Min-Cost-Robust Child (Day {day}):
State={min_cost_robust_child.state}, Cost={min_cost}, Visit
Count={min_cost_robust_child.visit_count}"
            )
        if secure_child:
            self.logger.info(
                f"Secure Child (Day {day}): State={secure_child.state}, Secure
Value={max_secure_value}"
            )

    def display_all_nodes(self, nodes):
        for node in nodes:
            print(
                f"State: {node.state}, Visit Count: {node.visit_count}, Total
Cost: {node.total_cost}"
            )
            self.logger.info(
                f"State: {node.state}, Visit Count: {node.visit_count}, Total
Cost: {node.total_cost}"
            )

    def print_execution_times(self):
        self.logger.info(
            f"\n\n\n Time to preprocess the data:
{self.end_time_data_preprocessing:.4f} seconds"
```

```
        )
        self.logger.info(
            f"\n\n\n Time to find the solution: {self.end_search_time:.4f}
    seconds"
        )
        self.logger.info(
            f"\n\n\n Total time:
    {self.end_time_data_preprocessing+self.end_search_time:.4f} seconds \n\n"
        )

    def get_simulation_policy(self):
        if self.desired_simulation_policy == "greedy_policy":
            return self.greedy_policy
        elif self.desired_simulation_policy == "random_policy":
            return self.random_policy
        elif self.desired_simulation_policy == "tolerance_policy":
            return self.tolerance_heuristic_policy
        else:
            raise ValueError(
                f"Unknown simulation policy: {self.desired_simulation_policy}"
            )

    def get_expansion_policy(self):
        if self.desired_expansion_policy == "top_k":
            return self.top_k_actions

        if self.desired_expansion_policy == "ratio_k":
            return self.ratio_best_random

        else:
            raise ValueError(
                f"Unknown expansion policy: {self.desired_expansion_policy}"
            )

    def top_k_actions(self, actions):
        sorted_actions = sorted(actions, key=lambda x: x[1])
        return sorted_actions[: self.number_childrens]

    def ratio_best_random(self, actions):
        # Determine the number of best actions to take based on the ratio
        ratio = self.ratio_expansion
```

```
        num_best = int(self.number_childrens * ratio)
        num_random = self.number_childrens - num_best

        # Sort actions to get the best ones
        sorted_actions = sorted(actions, key=lambda x: x[1])
        best_actions = sorted_actions[:num_best]

        # Select the remaining random actions from the remaining pool
        remaining_actions = sorted_actions[num_best:]

        # Ensure we don't try to sample more than available actions
        num_random = min(num_random, len(remaining_actions))

        # If num_random is zero or there are no remaining actions, we skip the
    sampling
        if num_random > 0 and remaining_actions:
            random_actions = random.sample(remaining_actions, num_random)
        else:
            random_actions = []

        # Combine the best actions and the random actions
        final_actions = best_actions + random_actions
        random.shuffle(final_actions)

        return final_actions

    def delete_node(self, node):
        if node.parent:
            for _ in node.parent.children:
                pass
                # self.logger.info(
                #     f"before deletion: {len(node.parent.children)},{_.state}"
                # )
            node.parent.children.remove(node)
            for _ in node.parent.children:
                pass
                # self.logger.info(
                #     f"after deletion: {len(node.parent.children)},{_.state}"
                # )

    def print_characteristics_simulation(self):
```

```
        self.logger.info(f"\n\nSimulation dictionnary:
    {self.simulations_dict}")
        self.logger.info(f"Number of childrens: {self.number_childrens}")
        self.logger.info(f"Desired expansion policy:
    {self.desired_expansion_policy}")
        self.logger.info(f"Ratio expansion: {self.ratio_expansion}")
        self.logger.info(f"Desired simulation policy:
    {self.desired_simulation_policy}")
        self.logger.info(f"Desired selection policy:
    {self.desired_selection_policy}")
        self.logger.info(f"Cp: {self.cp}")
        self.logger.info(f"Instance: {self.instance_number}")

    def simulation(self):
        for _ in range(self.number_simulation):
            self.logger = None
            self.logger = self.configure_logging()
            self.root = Node(
                self.initialise_root_node(),
                desired_selection_policy=self.desired_selection_policy,
                cp=self.cp,
            )
            self.best_leaf = None
            self.best_leaf_cost = float("inf")
            self.search()
            self.end_search_time = time.time() - self.start_time
            self.print_execution_times()
            self.get_final_nodes()
            self.print_characteristics_simulation()

    def select(self, node):
        self.logger.info("\nSELECTION\n")
        current_node = node
        self.logger.info(f"Starting selection at node: {current_node.state}")

        while current_node.children:
            self.logger.info(f"Current node: {current_node.state}")
            self.logger.info(f"Childrens: {current_node.children}")

            if not current_node.is_fully_expanded():
                # Select a random unvisited child if there are any
```

```
            unvisited_children = [
                child for child in current_node.children if
child.visit_count == 0
            ]
            self.logger.info(f"Unvisited children:
{len(unvisited_children)}")
            if unvisited_children:
                selected_child = random.choice(unvisited_children)
                self.logger.info(
                    f"Randomly selected unvisited child: {selected_child}"
                )
                return True, selected_child

        else:
            current_node = current_node.best_child()
            self.logger.info(f"Moving to best child: {current_node.state}")
            # return True, current_node

    if (not current_node.children) and (
        current_node.state["current_day"] == self.number_of_areas
    ):
        self.logger.info("Final day selected")
        return False, current_node

    elif (not current_node.children) and (
        current_node.state["current_day"] != self.number_of_areas
    ):
        self.logger.info(f"The node {current_node.state} has no children")
        return False, current_node

    elif current_node.state["current_day"] == self.number_of_areas + 1:
        return True, current_node

def expand_node(self, node):
    if node not in self.expanded_nodes:
        self.expanded_nodes.append(node)

        actions =
self.possible_flights_from_an_airport_at_a_specific_day_with_previous_areas(
            node.state["current_day"],
            node.state["current_airport"],
```

```
                node.state["visited_zones"],
            )

            if node.state["current_day"] == self.number_of_areas:
                node.state["visited_zones"] = node.state["visited_zones"][1:]
                node.state["remaining_zones"].append(
                    self.associated_area_to_airport(self.starting_airport)
                )

                actions =
    self.possible_flights_from_an_airport_at_a_specific_day_with_previous_areas(
                    node.state["current_day"],
                    node.state["current_airport"],
                    node.state["visited_zones"],
                )

            expansion_policy = self.get_expansion_policy()
            actions = expansion_policy(actions)

            if actions:
                self.logger.info("Start expansion")
                for action in actions:
                    self.logger.info(f"{action}")
                    new_state = self.transition_function(node.state, action)
                    node.add_child(new_state)
                self.logger.info("End expansion")
            else:
                self.logger.info(f"No actions possible")
                return None

            return node

        else:
            self.logger.info("INFINITE LOOP")
            return None

    def search(self):
        while True:
            node_to_explore = self.select(self.root)

            self.logger.info(f"Node to explore: {node_to_explore[1].state}")
```

```
        if node_to_explore[1].state["current_day"] == self.number_of_areas
+ 1:
            while not node_to_explore[1].parent.is_fully_expanded():
                # self.logger.info(
                #     "Node to explore is last day but all siblings have
not been visited yet"
                # )
                node_to_explore = self.select(self.root)
                self.logger.info(f"Node to explore:
{node_to_explore[1].state}")
                result = node_to_explore[1].state["total_cost"]
                self.backpropagate(node_to_explore[1], result)

            node_to_explore[1].state["visited_zones"].append(
                self.associated_area_to_airport(
                    airport=node_to_explore[1].state["path"][-1]
                )
            )
            return

        if not node_to_explore[0]:
            expanded_node = self.expand_node(node=node_to_explore[1])
            if not expanded_node:
                self.logger.info("Not unexpandable so deleted")
                node_to_explore[1].delete_node()
                # self.logger.info(f"Nodes in tree:
{len(self.collect_all_nodes())}")
                if len(self.collect_all_nodes()) == 1:
                    self.logger.info("Everything has been deleted to the
root node")
                    self.end_time_data_preprocessing = 0
                    self.end_search_time = 0
                    self.print_characteristics_simulation()
                    self.print_execution_times()
                    break
                continue
            else:
                self.logger.info(
                    f"{node_to_explore[1].state} has been successfully
expanded"
```

```
                )
                continue

        else:
            simulation = self.simulate(node_to_explore[1])

            if simulation[0]:
                self.logger.info(f"Result from simulation:
{simulation[0]}")

                key = str(node_to_explore[1].state["current_day"])
                value_to_add = simulation[0]
                if key in self.simulations_dict:
                    self.simulations_dict[key].append(value_to_add)
                else:
                    self.simulations_dict[key] = [value_to_add]

                self.backpropagate(node_to_explore[1], simulation[0])

            else:
                self.logger.info(
                    "Simulation failed to reach a valuable state - node
deleted"
                )
                self.delete_node(node_to_explore[1])
                if len(self.collect_all_nodes()) == 1:
                    self.logger.info("Everything has been deleted to the
root node")
                    self.end_time_data_preprocessing = 0
                    self.end_search_time = 0
                    self.print_characteristics_simulation()
                    self.print_execution_times()
                    break

def simulate(self, node):
    self.logger.info("\n\nSIMULATION")
    simulation_policy = self.get_simulation_policy()
    current_simulation_state = deepcopy(node.state)
    self.logger.info(f"Selected node for simulation
{current_simulation_state}")
```

```
        while current_simulation_state["current_day"] != self.number_of_areas:
            actions =
    self.possible_flights_from_an_airport_at_a_specific_day_with_previous_areas(
                day=current_simulation_state["current_day"],
                from_airport=current_simulation_state["current_airport"],
                visited_areas=current_simulation_state["visited_zones"],
            )

            action = simulation_policy(actions=actions)
            # self.logger.info(f"Action: {action}")
            if action is None:
                self.logger.info("Action is None")
                return False, False

            current_simulation_state = self.transition_function(
                current_simulation_state, action
            )
            # self.logger.info(f"Current simulation state
    {current_simulation_state}")

        if current_simulation_state["current_day"] == self.number_of_areas:
            current_simulation_state["visited_zones"] =
    current_simulation_state[
                "visited_zones"
            ][1:]
            current_simulation_state["remaining_zones"].append(
                self.associated_area_to_airport(self.starting_airport)
            )

            actions =
    self.possible_flights_from_an_airport_at_a_specific_day_with_previous_areas(
                day=current_simulation_state["current_day"],
                from_airport=current_simulation_state["current_airport"],
                visited_areas=current_simulation_state["visited_zones"],
            )

            if not actions:
                self.logger.info("No flight available to go back to the
    initial area")
                return False, False
            else:
```

```
                action = simulation_policy(actions=actions)

                current_simulation_state = self.transition_function(
                    current_simulation_state, action
                )
                self.logger.info(f"Current simulation state
    {current_simulation_state}")

                return current_simulation_state["total_cost"],
    current_simulation_state
```

Chapter 9

Test Instances

The instances can be found on the following website: https://code.kiwi.com/articles/travelling-salesman-challenge-2-0-wrap-up/

Chapter 10

Simulations results

10.1 Instance 1

Selec policy	Exp policy	Simu policy	N° children	Ratio	Cp	Best cost	Mean	Std	T(s)
UCB	ratio k	tolerance	10.0	0.0	1.4	1396.0	1396.0	0.0	0.0
UCB	top k	tolerance	10.0	0.0	2.8	1396.0	1396.0	0.0	0.1
UCB	top k	tolerance	5.0	0.0	1.4	1396.0	1396.0	0.0	0.1
UCB	ratio k	tolerance	5.0	0.0	2.8	1396.0	1531.0	133.8	0.1
UCB	top k	tolerance	15.0	0.8	1.4	1396.0	1577.8	133.0	0.2
UCB	ratio k	tolerance	15.0	0.0	1.4	1396.0	1396.0	0.0	0.2
UCB	top k	tolerance	5.0	1.0	2.8	1396.0	1572.6	117.8	0.2
UCB	top k	tolerance	15.0	0.0	1.4	1396.0	1396.0	0.0	0.2
UCB	top k	tolerance	5.0	1.0	1.4	1396.0	1666.6	148.8	0.2
UCB	ratio k	tolerance	5.0	1.0	1.4	1396.0	1521.7	114.1	0.2
UCB	ratio k	tolerance	15.0	1.0	1.4	1396.0	1643.6	132.6	0.2
UCB	top k	tolerance	5.0	0.0	2.8	1396.0	1396.0	0.0	0.2
UCB	ratio k	tolerance	15.0	0.0	2.8	1396.0	1396.0	0.0	0.2
UCB	ratio k	tolerance	10.0	0.0	2.8	1396.0	1396.0	0.0	0.2
UCB	ratio k	tolerance	10.0	0.8	1.4	1396.0	1596.4	92.2	0.2
UCB	top k	tolerance	10.0	0.0	1.4	1396.0	1396.0	0.0	0.3
UCB	top k	tolerance	15.0	0.0	2.8	1396.0	1396.0	0.0	0.3
UCB	top k	tolerance	10.0	0.5	2.8	1396.0	1555.9	87.7	0.4
UCB	top k	tolerance	5.0	0.3	1.4	1431.0	1518.0	43.2	0.0
UCB	ratio k	tolerance	10.0	0.8	2.8	1431.0	1622.4	159.4	0.1
UCB	top k	tolerance	15.0	0.5	2.8	1431.0	1561.9	95.8	0.1

UCB	ratio k	tolerance	15.0	0.8	1.4	1431.0	1582.4	130.4	0.1
UCB	top k	tolerance	15.0	1.0	1.4	1431.0	1597.3	124.3	0.1
UCB	top k	tolerance	15.0	0.3	0.0	1431.0	1815.8	189.8	0.3
UCB	top k	tolerance	10.0	0.3	2.8	1457.0	1614.4	173.1	0.0
UCB	ratio k	tolerance	5.0	0.3	2.8	1457.0	1566.0	106.9	0.0
UCB	ratio k	tolerance	5.0	0.5	2.8	1457.0	1600.6	113.8	0.1
UCB	top k	tolerance	5.0	0.3	2.8	1457.0	1533.8	86.8	0.2
UCB	ratio k	tolerance	5.0	0.0	1.4	1458.0	1586.0	97.9	0.0
UCB	top k	tolerance	10.0	1.0	1.4	1458.0	1581.1	137.5	0.0
UCB	ratio k	tolerance	10.0	0.5	2.8	1458.0	1570.0	81.5	0.0
UCB	ratio k	tolerance	5.0	1.0	2.8	1458.0	1596.3	100.2	0.1
UCB	ratio k	tolerance	15.0	0.5	2.8	1458.0	1586.6	91.9	0.1
UCB	top k	tolerance	15.0	0.3	1.4	1458.0	1516.2	49.7	0.1
UCB	top k	tolerance	10.0	1.0	2.8	1458.0	1589.1	95.9	0.1
UCB	top k	tolerance	15.0	1.0	2.8	1458.0	1618.7	137.1	0.1
UCB	ratio k	tolerance	15.0	1.0	2.8	1458.0	1624.0	133.5	0.1
UCB	top k	tolerance	15.0	0.5	1.4	1458.0	1654.1	146.8	0.1
UCB	ratio k	tolerance	15.0	0.8	2.8	1458.0	1666.0	108.3	0.1
UCB	top k	tolerance	5.0	0.8	1.4	1458.0	1625.7	105.6	0.2
UCB	top k	tolerance	10.0	0.5	1.4	1458.0	1547.4	78.2	0.2
UCB	top k	tolerance	10.0	0.8	2.8	1458.0	1626.9	129.6	0.2
UCB	top k	tolerance	5.0	0.8	2.8	1458.0	1588.2	79.4	0.2
UCB	top k	tolerance	15.0	0.3	2.8	1472.0	1523.3	56.3	0.2
UCB	top k	tolerance	15.0	0.8	2.8	1472.0	1615.0	99.0	0.2
UCB	top k	tolerance	10.0	0.3	1.4	1472.0	1507.8	38.9	0.2
UCB	ratio k	tolerance	5.0	0.3	0.0	1472.0	1913.6	199.6	0.3
UCB	top k	tolerance	5.0	1.0	0.0	1472.0	1775.0	165.7	0.4
UCB	ratio k	tolerance	15.0	1.0	0.0	1472.0	1786.7	200.4	0.6
UCB	top k	tolerance	15.0	0.8	0.0	1472.0	1787.3	179.2	0.7
UCB1T	top k	tolerance	15.0	0.5	1.4	1472.0	1827.4	227.2	1.0
UCB1T	ratio k	tolerance	10.0	0.8	1.4	1472.0	1819.2	166.9	1.1
UCB	top k	tolerance	10.0	0.8	0.0	1472.0	1798.6	177.2	1.4
UCB1T	top k	tolerance	15.0	0.3	0.0	1472.0	1828.1	215.2	1.8
UCB1T	top k	tolerance	15.0	0.0	2.8	1472.0	1819.1	204.2	2.0
UCB1T	top k	tolerance	5.0	0.5	1.4	1472.0	1785.3	162.4	5.5
UCB1T	top k	tolerance	5.0	0.5	2.8	1472.0	1757.0	223.9	5.9
UCB	ratio k	tolerance	10.0	0.3	1.4	1479.0	1522.3	35.4	0.1
UCB	ratio k	tolerance	5.0	0.3	1.4	1479.0	1528.1	92.9	0.1
UCB	ratio k	tolerance	15.0	0.3	1.4	1479.0	1551.9	60.1	0.1
UCB	ratio k	tolerance	10.0	0.3	2.8	1479.0	1590.3	112.1	0.1
UCB	ratio k	tolerance	15.0	0.3	2.8	1479.0	1624.9	134.5	0.1

UCB1T	ratio k	tolerance	5.0	0.0	0.0	1490.0	1798.1	164.7	0.1
UCB	ratio k	tolerance	10.0	0.5	1.4	1490.0	1628.4	87.8	0.2
UCB	ratio k	tolerance	5.0	0.5	1.4	1490.0	1596.1	72.6	0.2
UCB	top k	tolerance	10.0	0.8	1.4	1493.0	1618.6	91.7	0.1
UCB	top k	tolerance	5.0	0.5	1.4	1493.0	1573.0	68.2	0.1
UCB	ratio k	tolerance	5.0	0.8	2.8	1506.0	1610.3	81.3	0.1
UCB	ratio k	tolerance	10.0	1.0	1.4	1521.0	1629.3	72.7	0.1
UCB1T	top k	tolerance	15.0	0.8	1.4	1521.0	1811.4	188.3	2.0
UCB	ratio k	tolerance	10.0	1.0	2.8	1522.0	1627.1	90.9	0.2
UCB	ratio k	tolerance	5.0	0.8	1.4	1522.0	1641.8	73.5	0.2
UCB	top k	tolerance	5.0	0.5	2.8	1526.0	1577.7	43.3	0.2
UCB	ratio k	tolerance	5.0	0.5	0.0	1529.0	1877.8	199.1	0.2
UCB	ratio k	tolerance	10.0	0.5	0.0	1529.0	1847.4	160.9	0.3
UCB1T	top k	tolerance	15.0	0.8	2.8	1529.0	1811.1	140.9	0.5
UCB1T	top k	tolerance	10.0	0.3	1.4	1529.0	1970.6	254.1	0.8
UCB1T	top k	tolerance	15.0	0.3	1.4	1529.0	1732.0	178.4	1.1
UCB1T	top k	tolerance	5.0	0.3	0.0	1529.0	1780.9	164.4	1.3
UCB1T	ratio k	tolerance	5.0	0.8	2.8	1529.0	1809.0	137.8	1.4
UCB	ratio k	tolerance	15.0	0.3	0.0	1529.0	1956.8	208.3	1.6
UCB1T	ratio k	tolerance	10.0	0.3	0.0	1529.0	1864.2	213.2	1.7
UCB1T	top k	tolerance	5.0	0.0	2.8	1529.0	1824.0	147.9	6.8
UCB	ratio k	tolerance	15.0	0.5	1.4	1530.0	1585.4	55.7	0.1
UCB1T	top k	tolerance	15.0	1.0	0.0	1533.0	1834.1	199.4	2.6
UCB1T	ratio k	tolerance	10.0	1.0	1.4	1533.0	1869.4	181.3	2.9
UCB1T	ratio k	tolerance	5.0	1.0	1.4	1533.0	1850.6	172.3	3.8
UCB1T	top k	tolerance	15.0	0.0	0.0	1540.0	1795.3	202.5	0.2
UCB1T	ratio k	tolerance	10.0	1.0	2.8	1540.0	1851.2	221.6	0.3
UCB	ratio k	tolerance	10.0	0.0	0.0	1540.0	1791.9	160.8	0.5
UCB1T	ratio k	tolerance	10.0	0.5	2.8	1540.0	1800.6	203.9	0.5
UCB	ratio k	tolerance	5.0	1.0	0.0	1540.0	1874.2	157.6	0.6
UCB1T	ratio k	tolerance	15.0	0.0	1.4	1540.0	1815.7	127.5	0.7
UCB	top k	tolerance	15.0	0.0	0.0	1540.0	1885.4	190.7	0.8
UCB	top k	tolerance	10.0	1.0	0.0	1540.0	1880.1	308.8	1.0
UCB	top k	tolerance	10.0	0.3	0.0	1540.0	1876.0	172.9	1.1
UCB	top k	tolerance	10.0	0.5	0.0	1540.0	1938.9	187.7	1.1
UCB1T	ratio k	tolerance	15.0	0.5	1.4	1540.0	1760.2	132.8	1.3
UCB1T	top k	tolerance	15.0	1.0	1.4	1540.0	1887.1	187.8	1.4
UCB1T	ratio k	tolerance	15.0	0.0	2.8	1540.0	1813.4	195.5	1.5
UCB1T	top k	tolerance	10.0	0.5	0.0	1540.0	1893.8	221.5	1.6
UCB1T	top k	tolerance	10.0	0.3	2.8	1540.0	1843.9	155.5	1.6
UCB	top k	tolerance	15.0	0.5	0.0	1540.0	1881.9	220.3	1.8

UCB	top k	tolerance	10.0	0.0	0.0	1540.0	1886.0	166.7	1.8
UCB	ratio k	tolerance	5.0	0.8	0.0	1540.0	1862.7	184.8	2.0
UCB1T	top k	tolerance	5.0	0.8	1.4	1540.0	1813.8	145.1	2.4
UCB1T	top k	tolerance	15.0	1.0	2.8	1540.0	1855.0	159.3	2.6
UCB1T	ratio k	tolerance	15.0	1.0	2.8	1540.0	1799.9	176.5	2.9
UCB1T	ratio k	tolerance	10.0	0.3	2.8	1540.0	1915.6	218.8	3.1
UCB1T	top k	tolerance	10.0	1.0	1.4	1540.0	1938.8	188.1	3.1
UCB1T	top k	tolerance	5.0	0.0	1.4	1540.0	1806.1	151.8	4.2
UCB1T	ratio k	tolerance	5.0	1.0	0.0	1540.0	1830.2	200.5	5.4
UCB	ratio k	tolerance	10.0	0.8	0.0	1544.0	1864.8	240.7	1.1
UCB1T	top k	tolerance	10.0	0.3	0.0	1546.0	1857.7	188.7	2.3
UCB	top k	tolerance	5.0	0.3	0.0	1551.0	1772.9	88.5	0.3
UCB	ratio k	tolerance	10.0	0.3	0.0	1551.0	1826.8	142.9	0.6
UCB1T	ratio k	tolerance	5.0	1.0	2.8	1551.0	1839.2	206.1	1.6
UCB1T	ratio k	tolerance	15.0	0.5	0.0	1551.0	1882.0	170.6	1.7
UCB1T	top k	tolerance	5.0	1.0	1.4	1551.0	1868.1	259.4	2.4
UCB	ratio k	tolerance	5.0	0.0	0.0	1553.0	1936.8	207.6	0.6
UCB	ratio k	tolerance	15.0	0.8	0.0	1553.0	1881.0	203.4	0.9
UCB1T	top k	tolerance	10.0	0.0	0.0	1564.0	1877.2	181.4	0.3
UCB1T	ratio k	tolerance	15.0	0.5	2.8	1564.0	1956.2	224.0	0.3
UCB1T	ratio k	tolerance	10.0	0.8	2.8	1564.0	1814.7	190.9	1.0
UCB1T	top k	tolerance	5.0	0.8	2.8	1564.0	1872.6	147.0	1.9
UCB1T	top k	tolerance	15.0	0.5	2.8	1564.0	1799.9	218.4	2.2
UCB1T	ratio k	tolerance	15.0	0.8	2.8	1564.0	1780.3	168.9	2.3
UCB1T	ratio k	tolerance	5.0	0.8	0.0	1564.0	1849.1	139.3	2.7
UCB1T	ratio k	tolerance	10.0	0.5	0.0	1564.0	1828.4	197.4	2.8
UCB1T	top k	tolerance	5.0	0.3	2.8	1564.0	1812.8	148.3	4.4
UCB1T	top k	tolerance	5.0	1.0	0.0	1564.0	1806.7	129.6	5.7
UCB1T	top k	tolerance	5.0	0.3	1.4	1564.0	1856.6	158.2	6.5
UCB1T	top k	tolerance	5.0	1.0	2.8	1564.0	1861.3	142.9	6.7
UCB1T	top k	tolerance	15.0	0.0	1.4	1573.0	1879.4	206.1	1.8
UCB	top k	tolerance	5.0	0.8	0.0	1578.0	1843.2	130.6	0.5
UCB1T	top k	tolerance	15.0	0.8	0.0	1578.0	1813.1	164.8	1.4
UCB	top k	tolerance	5.0	0.5	0.0	1578.0	1875.3	178.6	1.9
UCB1T	ratio k	tolerance	5.0	0.5	0.0	1615.0	1832.4	183.6	1.5
UCB	ratio k	tolerance	15.0	0.5	0.0	1620.0	1867.1	158.9	0.2
UCB1T	ratio k	tolerance	15.0	0.3	0.0	1620.0	1828.3	161.8	0.7
UCB1T	ratio k	tolerance	5.0	0.3	1.4	1622.0	1847.3	153.3	1.5
UCB1T	ratio k	tolerance	5.0	0.3	2.8	1658.0	1871.2	111.7	0.3
UCB1T	ratio k	tolerance	15.0	0.8	1.4	1662.0	1929.2	225.2	1.3
UCB1T	top k	tolerance	10.0	0.0	2.8	1663.0	1880.1	183.9	1.3

UCB1T	top k	tolerance	10.0	0.5	1.4	1663.0	1865.8	119.5	1.4
UCB1T	top k	tolerance	5.0	0.0	0.0	1674.0	1808.0	85.7	7.4
UCB1T	ratio k	tolerance	10.0	0.8	0.0	1678.0	1862.6	163.0	1.1
UCB	ratio k	tolerance	10.0	1.0	0.0	1678.0	1915.4	132.6	1.5
UCB1T	top k	tolerance	10.0	0.8	1.4	1690.0	1909.8	166.0	1.0
UCB1T	ratio k	tolerance	15.0	0.8	0.0	1690.0	1897.4	127.5	1.8
UCB1T	ratio k	tolerance	5.0	0.3	0.0	1695.0	1941.2	167.6	1.0
UCB1T	top k	tolerance	5.0	0.5	0.0	1695.0	1946.6	148.7	4.7
UCB	ratio k	tolerance	15.0	0.0	0.0	1697.0	1921.8	177.6	1.0
UCB	top k	tolerance	5.0	0.0	0.0	1698.0	1866.7	118.9	3.6
UCB1T	top k	tolerance	15.0	0.3	2.8	1699.0	1920.2	146.3	0.4
UCB1T	ratio k	tolerance	5.0	0.5	1.4	1705.0	1856.6	114.4	1.0
UCB1T	top k	tolerance	10.0	0.8	0.0	1705.0	1851.3	112.4	1.8
UCB1T	top k	tolerance	10.0	1.0	2.8	1708.0	1920.3	181.4	2.6
UCB1T	top k	tolerance	10.0	1.0	0.0	1710.0	1948.2	164.5	0.3
UCB1T	top k	tolerance	10.0	0.5	2.8	1715.0	1934.2	144.1	2.4
UCB1T	ratio k	tolerance	5.0	0.5	2.8	1720.0	1934.9	110.9	2.0
UCB1T	ratio k	tolerance	15.0	1.0	1.4	1720.0	1882.9	123.4	2.6
UCB1T	ratio k	tolerance	5.0	0.8	1.4	1729.0	1884.0	116.8	2.8
UCB1T	top k	tolerance	10.0	0.0	1.4	1739.0	1959.0	148.3	0.6
UCB1T	top k	tolerance	5.0	0.8	0.0	1741.0	1880.4	96.7	7.7
UCB1T	ratio k	tolerance	5.0	0.0	1.4	1745.0	1973.6	173.2	0.6
UCB1T	ratio k	tolerance	10.0	0.0	2.8	1745.0	1987.0	165.1	0.9
UCB1T	top k	tolerance	10.0	0.8	2.8	1751.0	1954.7	190.1	1.1
UCB1T	ratio k	tolerance	15.0	0.3	1.4	1773.0	1920.4	126.9	0.3
UCB1T	ratio k	tolerance	10.0	1.0	0.0	1778.0	1916.1	74.0	0.3
UCB1T	ratio k	tolerance	10.0	0.5	1.4	1778.0	1973.1	125.8	0.6
UCB1T	ratio k	tolerance	5.0	0.0	2.8	1783.0	1972.2	184.5	0.3
UCB1T	ratio k	tolerance	10.0	0.0	0.0	1784.0	1963.9	148.1	1.1
UCB1T	ratio k	tolerance	15.0	0.0	0.0	1786.0	1934.2	125.5	2.5
UCB1T	top k	tolerance	15.0	0.5	0.0	1793.0	1935.1	120.6	2.2
UCB	top k	tolerance	15.0	1.0	0.0	1795.0	1954.8	117.0	0.8
UCB1T	ratio k	tolerance	10.0	0.3	1.4	1798.0	1969.7	198.4	0.6
UCB1T	ratio k	tolerance	15.0	0.3	2.8	1798.0	1944.9	87.6	0.6
UCB1T	ratio k	tolerance	15.0	1.0	0.0	1833.0	2005.4	124.3	2.1
UCB1T	ratio k	tolerance	10.0	0.0	1.4	1856.0	2001.7	118.2	1.6

10.2 Instance 2

Chapter 11

Best solutions

Instance 1

- Starting airport:'ABO'
- Solution = ['AB0', 'AB7', 'AB4', 'AB9', 'AB1', 'AB6', 'AB2', 'AB8', 'AB3', 'AB5', 'AB0']
- Associated cost = 1396

Instance 2

- Starting airport:'EBJ'
- Solution = ['EBJ', 'NBP', 'OMG', 'NCA', 'NUJ', 'OHT', 'GSM', 'EFZ', 'QKK', 'SSC', 'TKT']
- Associated cost = 1498

Instance 3

- Starting airport:'GDN'
- Solution = ['GDN', 'SZY', 'WMI', 'LD3', 'LB1', 'PD1', 'KRK', 'SA1', 'WRO', 'IEG', 'POZ', 'BZG', 'OSZ', 'OSP']
- Associated cost = 7672

Instance 4

Instance 7

Instance 8

Instance 9

Bibliography

[1] Jaap Bouwer Ludwig Hausmann Nina Lind Christophe Verstreken and Stavros Xanthopoulos. Air travel is becoming more seasonal. what steps can airlines take to adapt to the new shape of demand. *McKinsey*, January 8, 2024. URL `https://www.mckinsey.com/industries/travel-logistics-and-infrastructure/our-insights/how-airlines-can-handle-busier-summers-and-comparatively-quiet-winters#/`.

[2] Hennie de Harder. Np-what? complexity types of optimization problems explained. *Towards Data Science*, August 17, 2023.

[3] Hendrik Baier and Peter D. Drake. The power of forgetting: Improving the last-good-reply policy in monte carlo go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2:303–309, 2010. URL `https://api.semanticscholar.org/CorpusID:13578069`.

[4] Not mentionned. Number of flights performed by the global airline industry from 2004 to 2023, with a forecasts for 2024. `https://www.statista.com/statistics/564769/airline-industry-number-of-flights/`, 2024.

[5] Yaroslav Pylyavskyy, Ahmed Kheiri, and Leena Ahmed. A reinforcement learning hyper-heuristic for the optimisation of flight connections. pages 1–8, 07 2020. doi: 10.1109/CEC48606.2020.9185803.

[6] Hanif D. Sherali, Ebru K. Bish, and Xiaomei Zhu. Airline fleet assignment concepts, models, and algorithms. *European Journal of Operational Research*, 172(1): 1–30, 2006. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2005.01.056. URL `https://www.sciencedirect.com/science/article/pii/S0377221705002109`.

[7] J.E. Beasley and B. Cao. A dynamic programming based algorithm for the crew scheduling problem. *Computers and Operations Research*, 25(7):567–582,

1998. ISSN 0305-0548. doi: https://doi.org/10.1016/S0305-0548(98)00019-7. URL https://www.sciencedirect.com/science/article/pii/S0305054898000197.

[8] Deirdre Fulton. Unstoppable lccs - growth indicates a new norm. https://www.oag.com/blog/unstoppable-lccs-growth-indicates-new-norm, 2023.

[9] FranceTV Slash / Enquêtes. Ryanair: Y-a-t-il un rh dans l'avion? enquête sur les conditions de travail du géant du low-cost, 2024. URL https://www.youtube.com/watch?v=4TOsoX6aPiA. Accessed: 2024-07-05.

[10] Jens Clausen, Allan Larsen, Jesper Larsen, and Natalia J. Rezanova. Disruption management in the airline industry—concepts, models and methods. *Computers and Operations Research*, 37(5):809–821, 2010. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2009.03.027. URL https://www.sciencedirect.com/science/article/pii/S0305054809000914. Disruption Management.

[11] Allison Hope. The complex process behind your flight's schedule. *CNTraveler*, 2017. URL https://www.cntraveler.com/story/the-complex-process-behind-your-flights-schedule#:~:text=Flight%20schedules%20are%20mapped%20out,affect%20departure%20and%20arrival%20times.

[12] Not mentionned. Advanced decision support for aviation disruption management. https://www.inform-software.com/en/lp/aviation-disruption-management#:~:text=Proper%20aviation%20disruption%20management%20means,the%20schedule%2C%20while%20minimizing%20costs., 2024.

[13] Not mentionned. A modern cloud platform to optimize end-to-end airline operations and crew management. iflight drives unmatched efficiencies, cost-savings, and productivity for the world's top airlines. https://www.ibsplc.com/product/airline-operations-solutions/iflight, 2024.

[14] Not mentionned. What is acmi leasing? ACC Aviation, 2024.

[15] Lark Editorial Team. Np hard definition of np hardness. *Lark*, 26 December, 2023.

[16] Roy Jonker and Ton Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2(4):161–163, 1983. ISSN 0167-6377. doi: https://doi.org/10.1016/0167-6377(83)90048-2. URL https://www.sciencedirect.com/science/article/pii/0167637783900482.

[17] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209–219, 2006. ISSN 0305-0483. doi: https://doi.org/10.1016/j.omega.2004.10.004. URL https://www.sciencedirect.com/science/article/pii/S0305048304001550.

[18] Snežana Mitrović-Minić and Ramesh Krishnamurti. The multiple tsp with time windows: vehicle bounds based on precedence graphs. *Operations Research Letters*, 34(1):111–120, 2006. ISSN 0167-6377. doi: https://doi.org/10.1016/j.orl.2005.01.009. URL https://www.sciencedirect.com/science/article/pii/S0167637705000295.

[19] Pieter Vansteenwegen, Wouter Souffriau, and Dirk Van Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 209(1):1–10, 2011. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2010.03.045. URL https://www.sciencedirect.com/science/article/pii/S0377221710002973.

[20] Roberto Tadei, Guido Perboli, and Francesca Perfetti. The multi-path traveling salesman problem with stochastic travel costs. *EURO Journal on Transportation and Logistics*, 6(1):3–23, 2017. ISSN 2192-4376. doi: https://doi.org/10.1007/s13676-014-0056-2. URL https://www.sciencedirect.com/science/article/pii/S219243762030087X.

[21] Aviv Adler. The traveling salesman problem under dynamic constraints. *Massachusetts Institute of Technology*, Feb 2023.

[22] Petrică C. Pop, Ovidiu Cosma, Cosmin Sabo, and Corina Pop Sitar. A comprehensive survey on the generalized traveling salesman problem. *European Journal of Operational Research*, 314(3):819–835, 2024. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2023.07.022. URL https://www.sciencedirect.com/science/article/pii/S0377221723005581.

[23] Hung Chieng and Noorhaniza Wahid. *A Performance Comparison of Genetic Algorithm's Mutation Operators in n-Cities Open Loop Travelling Salesman Problem*, volume 287, pages 89–97. 01 2014. ISBN 978-3-319-07691-1. doi: 10.1007/978-3-319-07692-8_9.

[24] Malik Muneeb Abid and Muhammad Iqbal. Heuristic approaches to solve traveling salesman problem. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 15:390–396, 09 2015. doi: 10.11591/telkomnika.v15i2.8301.

[25] Bernhard Fleischmann. A cutting plane procedure for the travelling salesman problem on road networks. *European Journal of Operational Research*, 21(3):307–317, 1985. ISSN 0377-2217. doi: https://doi.org/10.1016/0377-2217(85)90151-1. URL https://www.sciencedirect.com/science/article/pii/0377221785901511.

[26] Not specified. Travelling salesman problem using dynamic programming. *Geeksforgeeks*, 19 April, 2023.

[27] Daniel Rosenkrantz, Richard Stearns, and Philip II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6:563–581, 09 1977. doi: 10.1137/0206041.

[28] Zakir Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometric and Bioinformatics*, 3, 03 2010. doi: 10.14569/IJACSA.2020.0110275.

[29] Lei Yang, Xin Hu, Kangshun Li, Weijia Ji, Qiongdan Hu, Rui Xu, and Dongya Wang. *Nested Simulated Annealing Algorithm to Solve Large-Scale TSP Problem*, pages 473–487. 05 2020. ISBN 978-981-15-5576-3. doi: 10.1007/978-981-15-5577-0_37.

[30] Yong Wang and Zunpu Han. Ant colony optimization for traveling salesman problem based on parameters optimization. *Applied Soft Computing*, 107:107439, 2021. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2021.107439. URL https://www.sciencedirect.com/science/article/pii/S1568494621003628.

[31] Wikipedia. Havannah (board game) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Havannah%20(board%20game)&oldid=1240631485, 2024. [Online; accessed 18-August-2024].

[32] Wikipedia. Game of the Amazons — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Game%20of%20the%20Amazons&oldid=1235225698, 2024. [Online; accessed 18-August-2024].

[33] Wikipedia. Lines of Action — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Lines%20of%20Action&oldid=1198717858, 2024. [Online; accessed 18-August-2024].

[34] Wikipedia. Shogi — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Shogi&oldid=1240175752, 2024. [Online; accessed 18-August-2024].

[35] Joris Duguépéroux, Ahmad Mazyad, Fabien Teytaud, and Julien Dehos. Pruning playouts in monte-carlo tree search for the game of havannah. volume 10068, pages 47–57, 06 2016. ISBN 978-3-319-50934-1. doi: 10.1007/978-3-319-50935-8_5.

[36] Richard J. Lorentz. Amazons discover monte-carlo. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 13–24, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87608-3.

[37] Mark Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte carlo tree search in lines of action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2:239 – 250, 12 2010. doi: 10.1109/TCIAIG.2010.2061050.

[38] Wikipedia. Go (game) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Go%20(game)&oldid=1239511822`, 2024. [Online; accessed 18-July-2024].

[39] Wikipedia. Lee Sedol — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Lee%20Sedol&oldid=1234296689`, 2024. [Online; accessed 11-August-2024].

[40] Google DeepMind. Alphago - the movie / full award-winning documentary. Youtube, 2020.

[41] Not mentionned. Explain the role of monte carlo tree search (mcts) in alphago and how it integrates with policy and value networks. EITCA, 2024.

[42] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 03 2012. doi: 10.1109/TCIAIG.2012.2186810.

[43] at the University of Strathclyde John Levine for his class CS310: Foundations of Artificial Intelligence. Monte carlo tree search, 2017. URL `https://www.youtube.com/watch?v=UXW2yZndl7U&t=385s`. Accessed: June, 2024.

[44] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, S. Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions*

on Computational Intelligence and AI in Games, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

[45] Rowaina Abdelnasser. Python naming conventions: 10 essential guidelines for clean and readable code. https://medium.com/@rowainaabdelnasser/python-naming-conventions-10-essential-guidelines-for-clean-and-readable-code-fe8~:text=For%20class%20names%20in%20Python,or%20behavior%20of%20the%20class., June 6, 2023.