POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

An Approximate Graph-Similarity Algorithm based on Monte Carlo Tree Search

Supervisor
prof. Giovanni Squillero

Co-supervisor:
prof. Stefano Quer

Candidate
Mattia De Prisco

Academic Year 2018-2019

Chapter 1

Introduction

Graphs are a general and powerful data structure that can model a large variety of concepts in many fields:

- computer science (e.g. networks, data, code blocks)
- linguistics (e.g. lexical semantics, phrases)
- physics and chemistry (e.g. atoms, molecules, neurons)
- social sciences (e.g. rumour spreading)
- mathematics (e.g. topologies)

and these are only some of the many applications.

Thus, finding the similarity between two graphs and a measure of similarity between a set of graphs are crucial topics.

The similarity between graphs can be evaluated computing the *Maximum Common Subgraph* (MCS), which consists in finding the largest graph which is isomorphic to two subgraphs of two given graphs. MCS comes in two forms:

- *maximum common induced subgraph*, whose aim is to find a graph with as many vertices as possible which is an induced subgraph of each of the input graphs
- *maximum common partial subgraph*, where a common non-induced subgraph with as many edges as possible is found

Basically, in the non-induced variant, edges must be mapped to existing edges, but additional edges may be present in the computed subgraph. In this work, the induced variant will be discussed.

This problem has been widely discussed in literature since the seventies [1][2][3], but being NP-hard, it remains computationally challenging.

When working with unlabelled graphs, state of the art algorithms [4] that compute the MCS generally become computationally unfeasible with graphs of only 40 vertices (even if in some cases they can manage graphs of two orders of magnitude above). Thus, because of the many application fields, most of the efforts were directed towards finding the best practical approach to the problem.

In 2017, McCreesh, Prosser and Trimble proposed *McSplit* [4], a branch-and-bound algorithm to find MCS for various types of graphs (such as undirected, directed and labelled) with several labelling strategies.

McSplit is a recursive procedure based on two main ideas: the use of a smart invariant, and an effective bound prediction formula.

Another critical limitation of state of the art algorithms is that they work with two graphs at once, while it could be really useful to be able to determine the MCS among a set of graphs.

This work proposes a different approach to the problem, by not using an exact algorithm, but an approximate one, *Gamy*, inspired by the *Monte Carlo tree search*, a heuristic search algorithm for decision processes, mostly used in general game play.

Gamy is aimed at providing satisfying solutions in reasonable times, with its main feature being the ability to compute the MCS between a set of any number of input graphs.

1.1 Document Structure

The second chapter of this work will further dive into the current state of the art, focussing on the *McSplit* algorithm, while in the third one the *Monte Carlo tree search* algorithm will be presented, explaining how it works along with an example and taking a look at some of its variations.

In the fourth chapter, the *Gamy* algorithm will be presented and discussed in detail, with its implementation being shown in chapter five. The sixth chapter will illustrate the experimental results - both comparing *Gamy* against *McSplit* and both evaluating *Gamy's* performances on more than two input graph.

Lastly, the seventh chapter will summarise the obtained results and discuss the applicability of this new algorithm, along with various extensions that have already been detected, which will further improve its performances.

Chapter 2

Background

2.1 Definitions

Before diving into the state of the art, it is better to clarify some definitions that will be used throughout this work.

Subgraph Given two graphs, $G = (V_G, E_G)$ and $F = (V_F, E_F)$, where V_G and V_F are the set of their vertices, and E_G and E_F are the set of their edges, F is a *subgraph* of G if its set of vertices is a subset of G's set, i.e. if $V_F \subseteq V_G$.

Induced subgraph In order for F to be considered an *induced subgraph* of G, it must include all the edges $e \in E_G$ which have both the endpoints in V_F, otherwise it is called a *partial* or *non-induced subgraph*.

Common subgraph Given two graphs G and F, a *common subgraph* H is a graph that is simultaneously isomorphic to a subgraph of G and a subgraph of F.

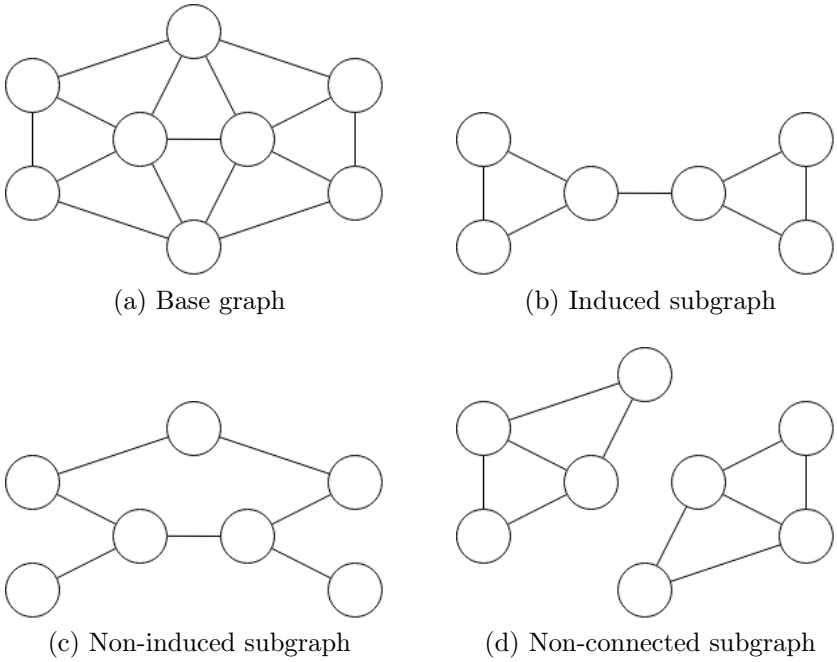
Maximum common subgraph The maximum common subgraph between two graphs G and F is the common subgraph with the highest number of vertices as possible.

Labelled graph A graph is *vertex-labelled* (*edge-labelled*) if each of its vertices (edges) have an associated label (which can be represented in various ways, e.g. a string, a number, etc.).

Directed graph A graph is *directed* if all of its edges are directed from one vertex to another. Instead, if the edges are bidirectional, i.e. they can be traversed starting from both its endpoints, the graph is *undirected*.

Connected graph A graph is *connected* when a path exists between every pair of vertices, i.e. starting from any node, there are no unreachable vertices. If the previous statement is not true, the graph is *disconnected*.

Figure 2.1 shows various types of subgraphs that can be obtained from a base graph, while figure 2.2 shows some maximum common subgraphs obtainable from two base graphs.



(a) Base graph (b) Induced subgraph

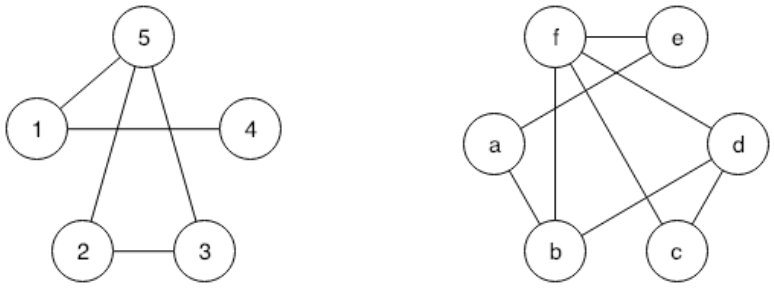
(c) Non-induced subgraph (d) Non-connected subgraph

Figure 2.1: An undirected base graph (a) and various types of subgraphs obtainable from it (b) (c) (d).

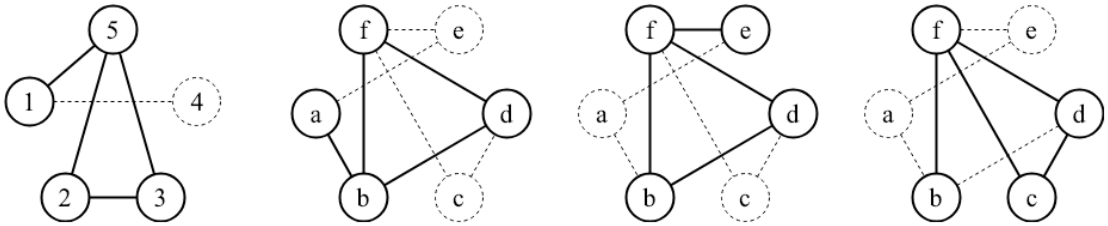
2.2 McSplit

McSplit [4] is a recently proposed algorithm proposed by McCreesh,Prosser and Trimble, and it is one of the most effective algorithm for the MCS problem.

It is a recursive algorithm based on a smart invariant and on an effective bound prediction formula that reduces the computational effort.



(a) Undirected graphs G and F



(b) Possible MCS for G and F

Figure 2.2: An undirected base graph (a) and various types of subgraphs obtainable from it (b) (c) (d).

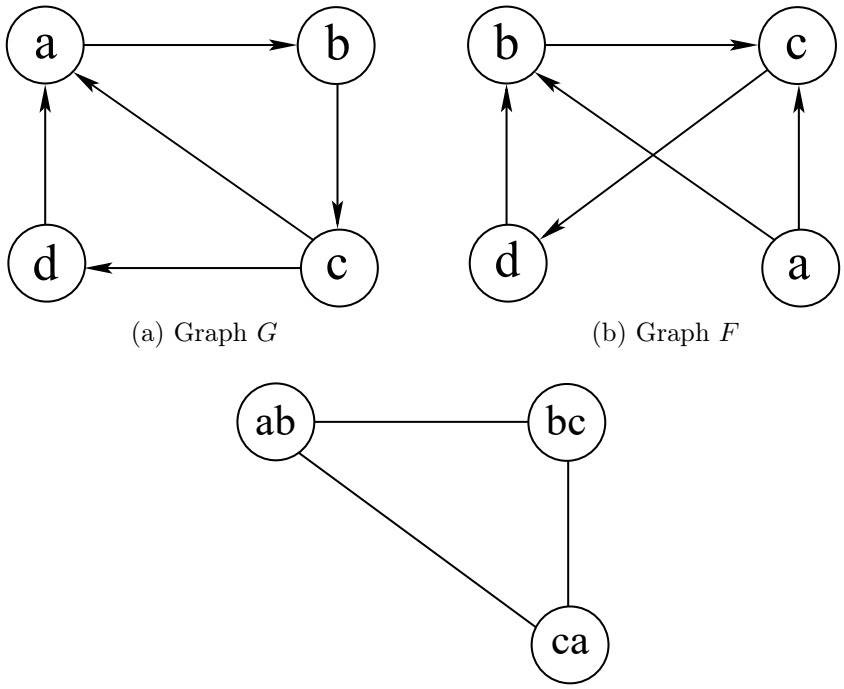
Invariant The algorithm builds a mapping between the vertices of the input graphs using a depth-first search, adding a new vertex pair (one for each graph) to the mapping set at each recursion level: after selecting a pair, a label is added to all the unmatched vertices according to whether they are adjacent to the just chosen vertex of their respective graph (for undirected graphs, adjacent vertices are assigned label 1, non-adjacent ones label 0). Each time a new pair has to be added to the set, only vertices that have the same label can be added.

Once no more vertices pairs can be chosen, the recursive procedure backtracks and tries to follow a different path that could lead to a longer set of pairs. After all the possibilities have been explored, the longest set of pairs found is the MCS of the input graphs.

Considering the input graphs G (in figure 2.3a) and F (in figure 2.3b) as undirected for the sake of simplicity, the *McSplit* algorithm computes the MCS shown in figure 2.3c.

In table 2.1 the whole process is shown: first the pair a, b is added to the mapping set M, and the non-mapped vertices are all assigned the label 1 since they all are connected with the selected vertices; then, the pair b, c is chosen, and only the vertex d of graph G receives a label 0 since it is not connected; lastly, the pair c, a

is selected, leaving M in a state in which no more pairs can be chosen, since the remaining vertices have different labels.



(a) Graph G

(b) Graph F

(c) MCS between G and F

Figure 2.3: Maximum common subgraph (c) computed among graphs G (a) and F (b) using *McSplit* algorithm.

Bound prediction The second main component that characterises the *McSplit* algorithm is the bound computation, which is used to prune the space search.

While parsing a branch, the following bound is evaluated:

$$bound = |M| + \sum_{l \in L} min(|\{v \in G : label(v) = l\}|, |\{v \in \mathcal{F} : label(v) = l\}|) \quad (2.1)$$

where $|M|$ is the cardinality of the current mapping and L is the actual set of labels. The algorithm will prune the current branch if the bound is smaller than the size of the current mapping since it means that there will be no possibility to find a match longer than the current one.

This operation allows to drastically reduce the computation effort.

G

Vertex	**Label**
b	1
c	1
d	1

F

Vertex	**Label**
c	1
a	1
d	1

(a)

G

Vertex	**Label**
c	11
d	10

F

Vertex	Label
a	11
d	11

(b)

G

Vertex	**Label**
d	101

F

Vertex	**Label**
d	101

(c)

Table 2.1: States of the labels on the non-mapped vertices of G and F as mapping proceeds. The state of the mapping set in each table is the following: (a) $M = a, b$ - (b) $M = ab, bc$ - (c) $M = abc, bca$

Since the MCS can be computed on various types of graphs, *McSplit* adapts its algorithm based on those types:

- In case of *directed* graphs, the adjacency matrix is modified: for each vertex pair $(v, u) \in G, F$ the adjacency matrix element $adj[v][u]$ takes the value 0 if v and u are not adjacent, 1 if the they share a single edge directed from v to u, 2 if they share a single edge directed from u to v, and 3 if there are edges in both directions. Thus, the algorithm splits the label class in four.
- In case of *vertex-labelled* graphs, the algorithm starts from a set of label classes, one for each different label belonging to the input graphs, and the invariant keeps this label information while it traverses the tree.

Chapter 3

Monte Carlo Tree Search

3.1 Introduction

Monte Carlo tree search (MCTS) is a heuristic search algorithm introduced by Rémi Coulom in 2006, used to calculate the most promising next move in a decision-making problem, combining the precision of tree search with the generality of random simulation.

The classic environment in which Monte Carlo tree search is used are discrete, deterministic games with perfect information:

- *discrete*: the set of moves and positions is finite
- *deterministic*: every move has a set outcome
- *game*: players competing against each other
- *perfect information*: both players see everything

Games like chess or Go fall under this definition, but while Deep Blue [5], a chess-playing computer by IBM, beat the world chess champion in 1997, no Go engine ever became close to human masters because of the combinatorial complexity of the game. That was until 2015, when Google's AlphaGo [6], an AI playing Go, exploited Monte Carlo tree search in combination with deep learning, becoming the first computer Go program able to beat a human professional Go player, with no handicaps and on a full sized 19x19 board.

Besides board games like Go, chess and shogi [7], Monte Carlo tree search has also been exploited in games with incomplete information like bridge [8] or poker [9], and as well as in real-time video games, like Total War: Rome II's implementation

in the high-level campaign AI [10].

Monte Carlo tree search is a hot topic right now in the AI field, with lots of research paper being written about it, suggesting variations, optimisations and enhancements.

3.2 Algorithm

Monte Carlo tree search is based on two concepts: that the value of an action can be approximated exploiting random simulation, and that these obtained values can be used to tune the policy in the direction of a best-first strategy. During MCTS execution, a game tree is built step-by-step, guided by the results provided by each iteration. The value of the moves is estimated using the tree, and the estimations become more accurate while the tree grows.

Before taking a more accurate look at how the algorithm works, it is necessary to clarify the meaning in this context of some terms that will be used:

- *Search*: a set of traversal down the game tree.
- *Game tree*: a tree in which every node represent a state of the game.
- *Traversal*: a path from the root node to a not fully expanded node (i.e. a node that has unvisited children).
- *Move*: a transition from a node to one of its children .
- *Root node*: the node representing the initial state of the game.
- *Terminal node*: a node that cannot have any children, meaning that the game has ended.
- *Expandable node*: a node that is not terminal and that has unvisited (unexpanded) children.

Each iteration of MCTS consists of four steps:

- *Selection*
- *Expansion*
- *Simulation*
- *Backpropagation*

Selection In the selection step, a traversal is performed: starting from the root node R, optimal child nodes are recursively selected until an expandable node L is reached. At the beginning of MCTS, only the root node R is in the tree, so it will be the first to be selected.

Subsequent selections will instead choose at each level of the tree the node that maximises some quantity (like in the multi-armed bandit problem where the player picks each turn a bandit that maximises the estimated reward), stopping when an expandable node is reached. The typical approach in selecting the most promising node is the UCT (Upper Confidence bounds applied to Trees) formula, which is the UCB (Upper Confidence Bound) formula applied to trees.

The UCT score for the node i (U_i) is computed in the following way:

$$U_i = X_i + c\sqrt{\frac{\ln N_p}{N_i}} \tag{3.1}$$

where X_i is the estimated value of the ith node, N_p is the number of times the parent of the ith node has been visited, N_i is the number of times the ith node has been visited and c is a constant. Note that a node's visits count increases only when a simulation starts from the node itself or from one of its descendants.

The left part of the formula, X_i, is called exploitation component, representing the accumulated reward for that node (e.g. in a game like tic-tac-toe, it estimates the win ratio of the node i). In competitive games the exploitation component is always computed relative to the player who moves at node i, meaning that the perspective changes while traversing the game tree, based on the node being traversed through: perspective is opposite for any two consecutive nodes.

If used alone, this component would lead the selection on a greedy path, favouring those nodes that result in a winning playout early in the search and abandoning the ones that were unlucky during the random playout resulting in a loss.

The right part of the formula, $\sqrt{\frac{\ln N_p}{N_i}}$, called the exploration component, fixes this behaviour by favouring those nodes that have been rarely visited, even more, if their parent has been visited a lot.

Lastly, the parameter $c \geq 0$ (which is usually set empirically) regulates the trade-off between choosing nodes that seem lucrative (when c is set to a low value) and rather unexplored nodes (c set to a high value).

Before nodes' UCT scores become reliable, the nodes need to be visited a certain number of times: their estimates will typically be unreliable at the start of a search,

but converge to more reliable ones after sufficient time, and even become perfect given infinite time.

Expansion In this step, the selected node L is expanded (unless it is a terminal node), creating one or more child nodes and choosing a node C among them. A particular version of Monte Carlo tree search, called *pure Monte Carlo game search*, expands all the possible child nodes of the selected node L and then run a playout from every one of them in the simulation step, but although this approach is more exhaustive, it is much slower since even low reward paths are expanded.

Simulation Simulation (or playout) is a single act of gameplay, meaning a sequence of moves starting at the expanded node C and ending in a terminal node: it consists in playing a game till the end starting from the state in the node C, according to a certain rollout policy function:

$$RolloutPolicy : s_i \rightarrow a_i \tag{3.2}$$

that given the state s_i produces the a_i move. Since lots of simulations need to be performed, the rollout policy has to be quick, so in practice, it is often a uniform random.

A simulation results in the evaluation of the reached state, that usually, for games, refers to a win, a loss or a draw, but in its simplest form a simulation does not have to end at a terminal node, in which case any other value specific to the particular application scenario is a legit result.

Backpropagation Backpropagation is a traversal from the leaf node C back to the root node R, performed once the simulation is over, propagating back the results, computing or updating specific statistics for all the nodes in the path from C to R, thus guaranteeing that every node's statistics reflect the results of the simulations started in all their descendants.

Termination The Monte Carlo tree search steps are performed cyclically many times, and the whole process will stop when a specific condition is met, usually when a timeout expires. At this point, the most promising move has to be chosen, and it typically is the one with the highest number of visits (N_i), since it means that most of the times it was considered the most lucrative node according to the UCT formula.

3.3 Tic-Tac-Toe Example

For this example, the exact UCT formula used becomes:

$$U_i = \frac{W_i}{N_i} + c\sqrt{\frac{\ln N_p}{N_i}} \tag{3.3}$$

where W_i is the accumulated value of the ith node (e.g. the sum of wins and losses from that node), thus the exploitation component represents the win ratio of the ith node.

In tic-tac-toe, the game can end in three different states: win, loss or tie, which will be valued +1, -1 and 0 respectively. It is important to note that in this example, W reflects whether the player using X won or lost, so during selection, when it is O's turn, the sign of W is flipped. Moreover, the *pure Monte Carlo game search* approach will be used.

In figure 3.1 the root node s_0 is chosen in the selection process since, at this point, no other node exists yet, and it is then expanded in all the possible combinations.

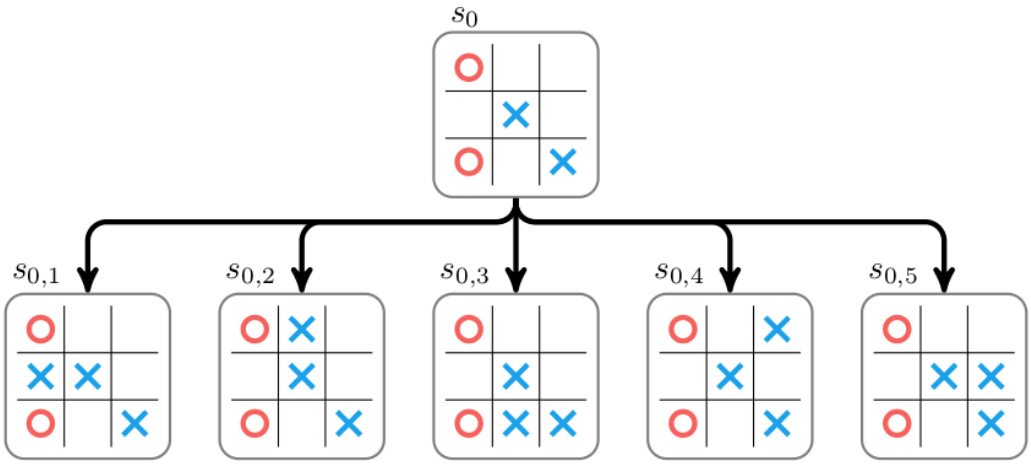


Figure 3.1: Selection of the root node s_0 and expansion of all the five possible children [11].

At this point a random simulation is run for all the child nodes expanded in the previous step: in figure 3.2 a sample playout from the node $s_{0,1}$ is shown, which ends in a win, thus increasing the value of $s_{0,1}$ by 1.

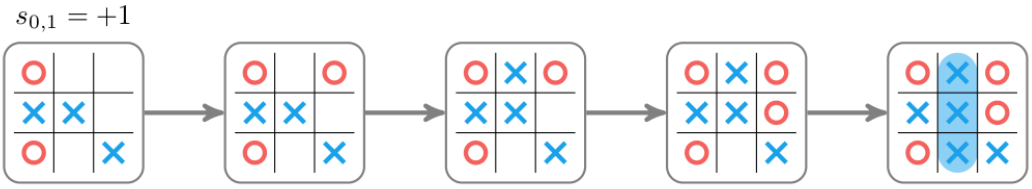


Figure 3.2: Simulation run for the node $s_{0,1}$ until an end state is reached [11].

Figure 3.3 shows the state of the tree once a playout is run for all the expanded nodes: for each node, two values are stored, N, the number of times that node has been visited and W, the accumulated value of wins and losses.

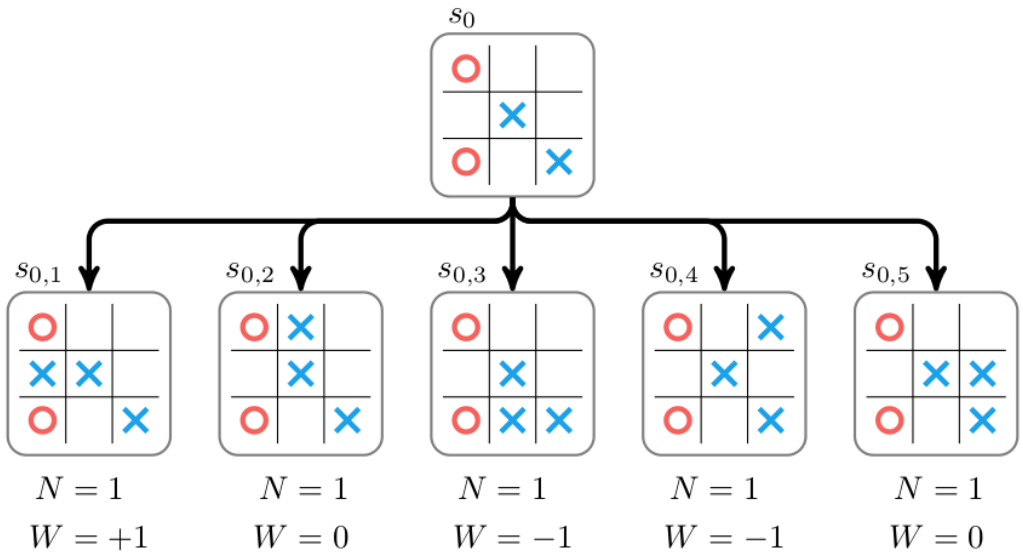


Figure 3.3: Results of the playouts of all the expanded nodes [11].

Once the simulation is run for all the child nodes, the results are propagated back up to the root node, reaching the situation showed in figure 3.4.

At this point, the process restarts, but now that more than one leaf node is present, the selection step will exploit the UCT scores to direct the traversal.

Figure 3.5 shows the UCT scores for the child nodes considering $c = 1$, thus node $s_{0,1}$ will be selected, which will be first expanded, then a simulation will be performed for all its child nodes, and lastly, the obtained results will be propagated back to the root node, obtaining the situation showed in figure 3.6.

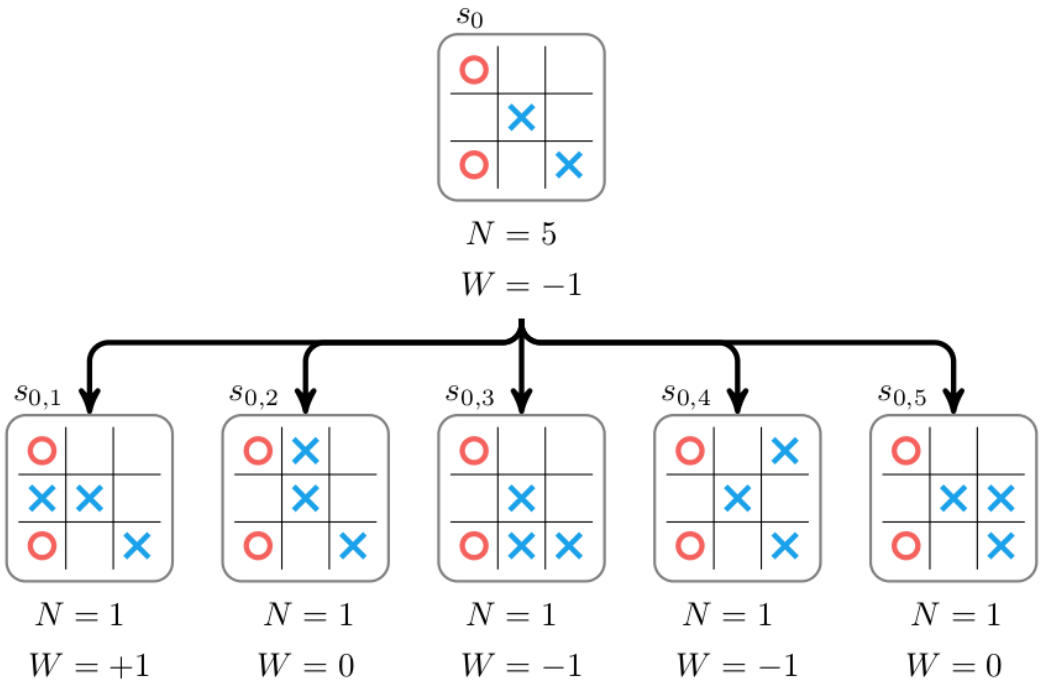


Figure 3.4: Backpropagation of the results until the root node s_0 [11].

This process continues until all the possible moves are expanded, obtaining the state showed in figure 3.7.

Now that the process is completed, the best possible move is the one with the highest visits count, since it means that most of the times it was considered the most lucrative node, which in this case is $s_{0,1}$

3.4 Advantages and Disadvantages

Monte Carlo tree search has various advantages with respect to traditional tree search methods:

- *Aheuristic*: it does not need any tactical or strategical knowledge about the domain in which it is operating in order to make reasonable decisions. The only things it needs to know are the permitted moves and the game-end conditions. This trait allows MCTS to be reused in different games with little modifications and to be effectively employed in general game playing.
- *Asymmetric*: the game tree grows asymmetrically (figure 3.8), since the algorithm tends to visits more promising nodes more often, thus focussing

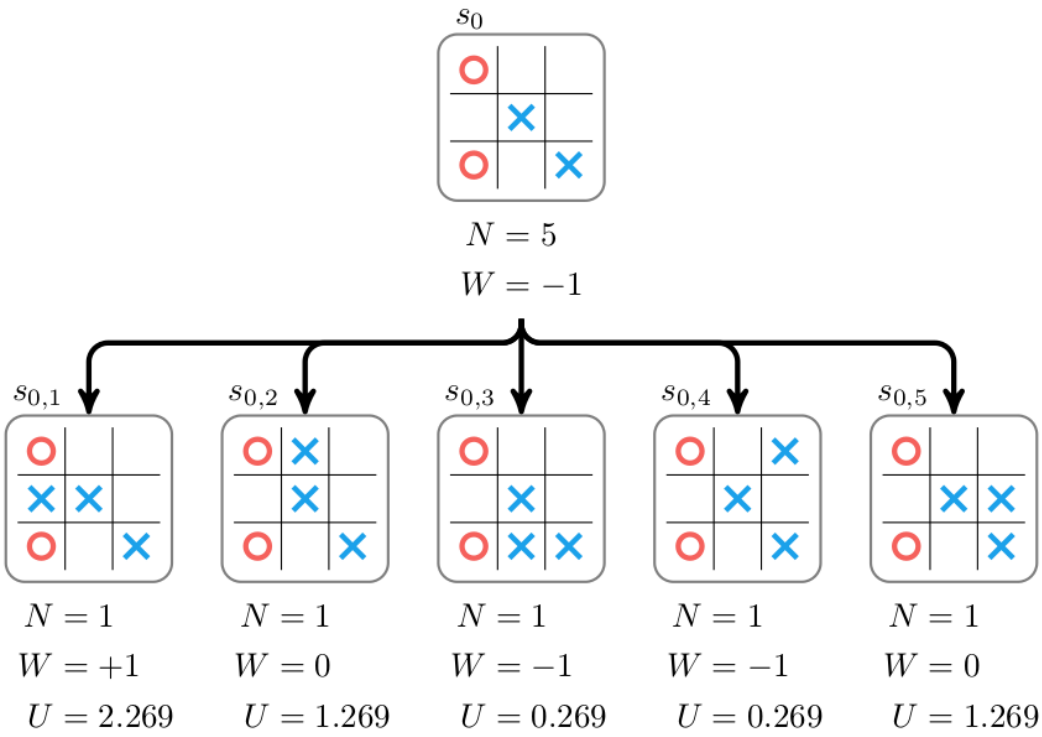


Figure 3.5: UCT scores with $c = 1$ [11].

most of the search time in the more relevant parts of the tree. For this reason, MCTS is well suited for games with large branching factors, where instead standard depth-based or breadth-based search methods struggle.

- *Anytime*: the algorithm can be stopped at any time, returning the most promising result found until that point.
- *Elegant*: the algorithm, especially in its base version, is straightforward to implement.

However, like any algorithm, it also has some drawbacks:

- *Playing strength*: in its basic form, the algorithm may fail to find good moves within a reasonable time, even for games of medium complexity. This is due to the size of the combinatorial move space and the fact that there may be key nodes that are not be visited enough times to give reliable estimates. For example, there may be one single move that leads to a win against an expert player, but given the nature of the algorithm, it may be overlooked.
- *Speed*: converging to a good solution may require many iterations, and this is

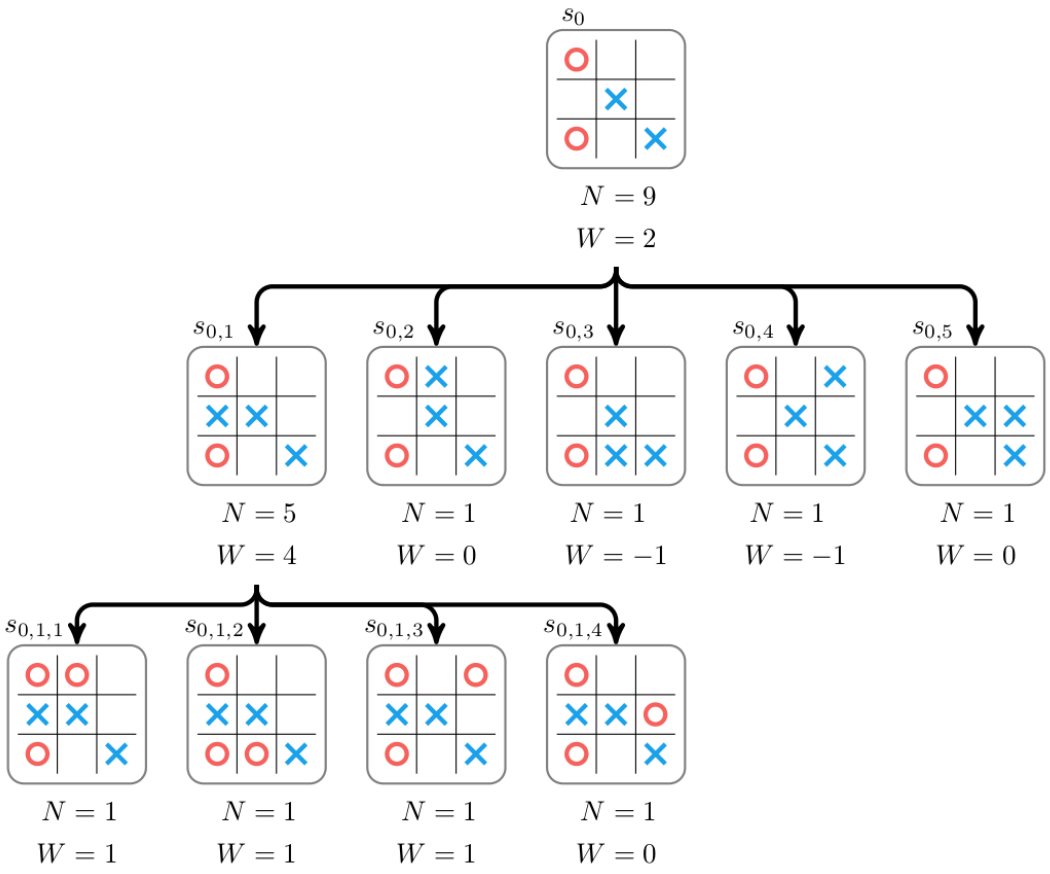


Figure 3.6: Selection of $s_{0,1}$, expansion, simulation and backpropagation [11].

especially true for more general applications that cannot be easily optimised.

3.5 Improvements

Several enhancements can be made to the algorithm, in order to improve its speed and efficiency.

Expert policies Monte Carlo tree search can employ either light or heavy playouts. Light playouts are essentially random moves, thus improving speed, while heavy playouts make use of various heuristics to compute the next move. These heuristics may exploit the results of the previous playouts or be based on expert knowledge of a specific game. This means that certain nodes are given a higher weight because they represent the move that an expert human player is more likely to make.

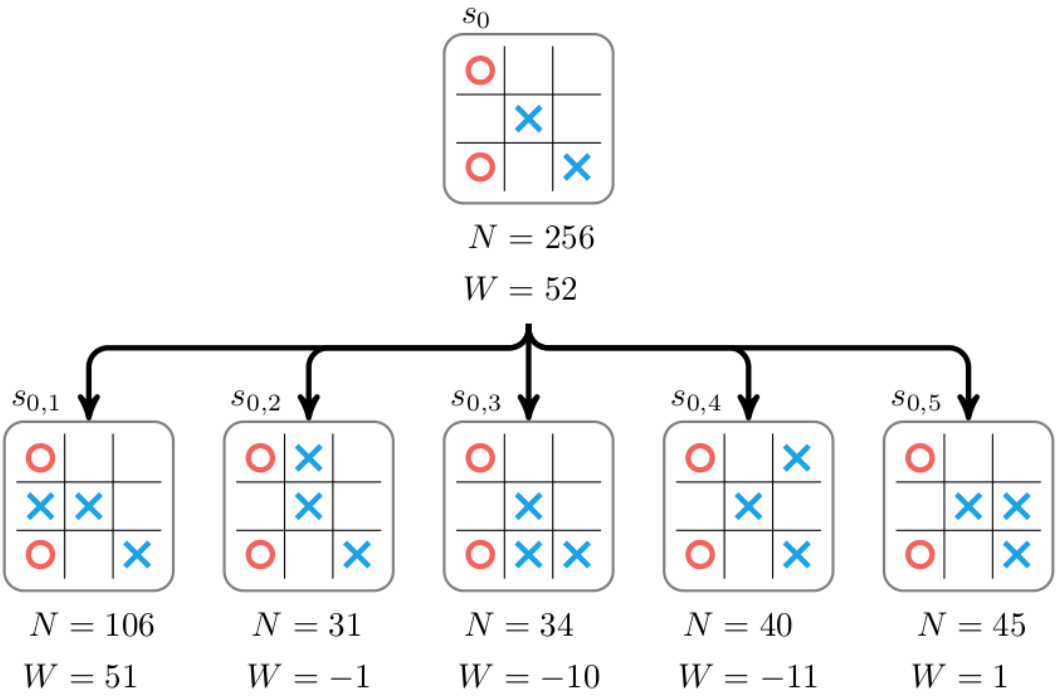


Figure 3.7: State of the tree after completely visiting it [11].

The UCT formula including expert policies becomes:

$$U_i = \frac{W_i}{N_i} + cP_i\sqrt{\frac{\ln N_p}{N_i}} \tag{3.4}$$

with $P_i = \pi(a_i|si)$, where, given a certain expert policy π, it represents the probability of choosing the ith action a_i from the state s_i.

As with the previous UCT formula, the score trades off between lucrative and unexplored nodes, but now, the expert policy is guiding node exploration, directing it towards moves that would more likely be made by an expert of that dominion. Thus, this addition makes playouts more realistic, obtaining reliable reward values in fewer iterations, but in exchange losing speed and generality. It is also important to consider that expert policies are very difficult to generate, and even then, it is hard to verify that they are optimal.

Going back to the tic-tac-toe example, figure 3.9 shows the probabilities of each move, given the current state s_0: an expert, seeing that two Os are on the same column, would almost surely place the X on that same column to prevent the

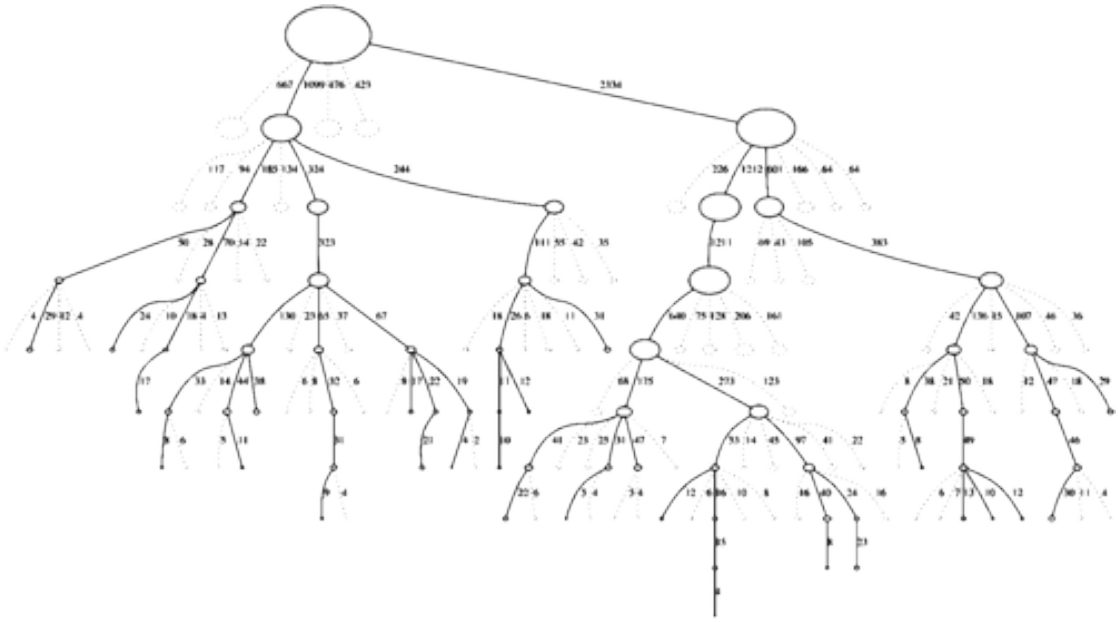


Figure 3.8: Sample showing how Monte Carlo tree search explores a tree asymmetrically [12].

opponent from winning the game.

Domain independent improvements Some other improvements are more general and can be applied to all problem domains. They are typically applied in the node selection, but others are also applied to the simulation (e.g. preferring some moves with respect to others). Since this kind of improvements, enhance the results without losing generality, most studies are focussed in this direction.

Parallelisation Monte Carlo tree search is perfectly suited for parallelisation, being indeed able of being parallelised on different levels:

- *Leaf parallelisation*: after the expansion phase, many playouts are run in parallel from the chosen node C.
- *Root parallelisation*: several game trees are built independently in parallel and moves are selected gathering statistic from all of them.
- *Tree parallelisation*: the game tree is built in parallel, managing synchronisation in order to avoid simultaneous writes.

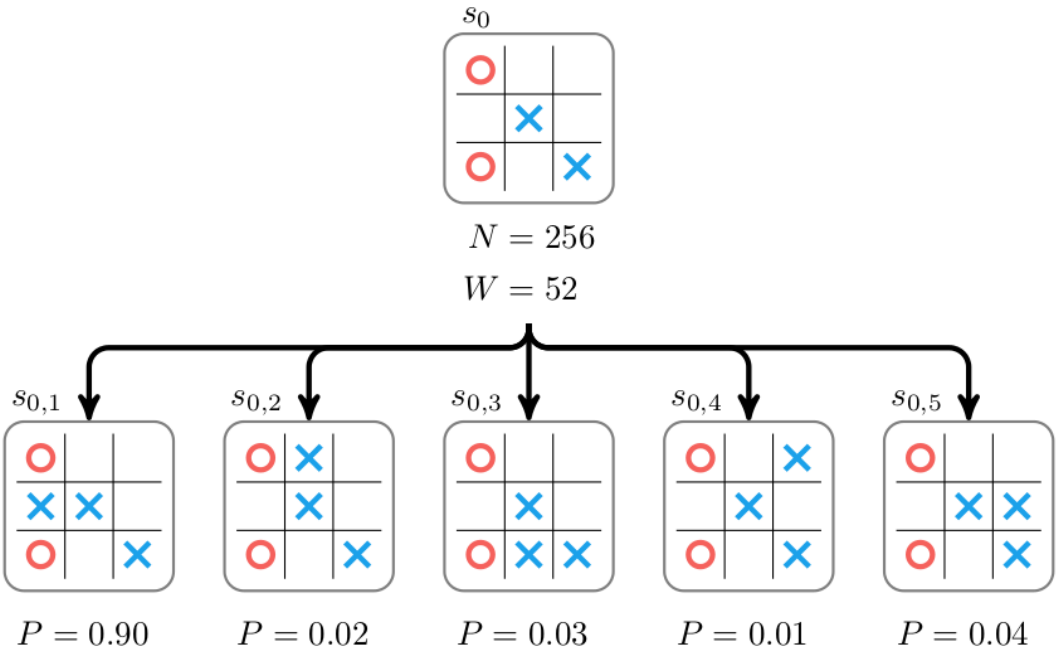


Figure 3.9: Probabilities P_i of choosing a certain move, based on the expert policy π [11].

3.6 Variations

Given its generality, Monte Carlo tree search can be applied in many flavours to better perform in each different dominion in which it is applied [13]:

Flat UCB In this variation, the leaf nodes of the trees are effectively treated as a single multi-armed bandit problem. It has been demonstrated that this approach maintains the adaptivity of the UCT formula while improving its regret boundaries in some worst cases, where UCT was observed to be excessively optimistic [14].

Bandit Algorithm for Smooth Trees (BAST) This is an extension of the flat UCB model that makes assumptions on the smoothness of the rewards to identify and prune suboptimal branches, which is in contrast with plain UCT where all the branches are indefinitely expanded [14].

Temporal Difference Learning Monte Carlo tree search learns based on the values of states or of the state-action pairs, using this information to decide which action to take, behaviour in common with temporal difference learning (TDL). A

significant difference is that TDL algorithms do not build trees, but under certain conditions, it can be equivalent to MCTS [15], i.e. when all the state values can directly be stored in a table. TDL learns the long-term values of the states, which will then guide the future behaviour, while MCTS estimates temporary reward values for the nodes to choose the next move. An approach combining both these algorithms has been proposed [16], which uses the concepts of permanent and transient memory to differentiate how MCTS and TDL handle the two types of state value estimation. TDL can learn heuristic value functions to influence the selection and simulation policies.

Single-Player MCTS (SP-MCTS) A variant for single players game, which adds a new term to the UCT formula representing the possible deviation of the node:

$$U_i = X_i + c\sqrt{\frac{\ln N_p}{N_i}} + \sqrt{\sigma^2 + \frac{D}{N_i}} \tag{3.5}$$

where σ^2 represents the node's i simulation results variance, and D is a constant. The term $\frac{D}{N_i}$ increases the standard deviation for nodes that have a low visits count, making rewards for said nodes more uncertain. Another difference between with respect to standard UCT is the usage of heuristically guided default policies during selection phase [17].

Multi-player MCTS When a game with more than two players is considered, the base idea of a player trying to maximise his reward and of an opponent trying to minimise it, cease to be valid. The simplest way to exploit Monte Carlo tree search in multi-player games is to use the max^n idea: a vector of rewards is stored for each node, and the selection step operates by trying to maximise the UCT score using the appropriate component of the reward vector [18].

Another addition in multi-player games is considering coalitions [19], i.e. a group of players playing together against other players/coalitions. In this case, the same max^n approach is used, but a rule is added so that simulations avoid making moves that would negatively affect players in the same coalitions, and rewards are differently computed, considering that said players belongs to the same coalition. Coalitions can be handled in several ways:

- *Paranoid UCT*: the player considers that all the other players are in the same coalition against him.
- *UCT with Alliances*: the coalitions are explicitly provided to the algorithm.

- *Confident UCT*: independent searches are performed for each possible coalition, and the move is chosen according to the most promising coalition.

Multi-agent MCTS The simulation phase of the standard UCT may be seen as a single agent playing against itself, but if instead, multiple agents are considered, some improvements can be made. More specifically, these multiple agents are obtained by assigning different priorities to the heuristics used, and it has been observed that using the right subset of agents, improves playing strength, because there is an increase in the exploration of the search space. The drawback is that it is computationally intensive to find the set of agents with the correct properties [20].

Real-time MCTS Traditional games are turn-based, usually giving players time to think about the next move. Unfortunately, this is not true for real-time scenarios, where there is constant progress without any waiting, so it is crucial to act swiftly. Real-time games are mostly represented by video games, which also often possess other features that increase the complexity of the problem:

- Uncertainty
- Massive branching factor
- Simultaneous moves
- Open-endedness

For all this reasons, developing an efficient algorithm for this dominion is challenging, but Monte Carlo tree search is well suited for it since it can stop at any time, thing that is crucial in this games where time is limited, and also because of its asymmetric exploration of the tree, allowing to better explore the space in the short time available.

Chapter 4

Gamy Approach

The *Gamy* algorithm has been developed with the main goal of allowing to compute the MCS among multiple input graphs simultaneously, overtaking the limit of comparing two graphs at a time. Its design also aims at providing an efficient solution to the problem when the graphs reach the thousands of vertices, situation in which exact algorithms often become unfeasible.

4.1 Overview

Finding the MCS between graphs is a complex problem that involves various difficulties, the main being the large branching factor, that grows swiftly with the number of vertices in the graphs.

Before continuing, it is essential to define the meaning of equivalence between vertices:

given two graphs, $G = (V_G, E_G)$ and $H = (V_H, E_H)$, and two vertices $a \in V_G$, with label L, and $b \in V_H$, with label K, a is equivalent b if and only if $L \equiv K$ and, if any of a's parents (or children) belongs to an equivalence class γ (meaning that it has been defined equivalent to another vertex), also b must have matching parents (or children) in that same equivalence class γ.

This definition applies in the same way to n different vertices from n different graphs, and it also applies to unlabelled graphs, with only the parents and children restrictions applying.

To get a better grasp of this definition, let us take a look at an example, where the goal is to prove the equivalence between two vertices.

Starting from the situation shown in figure 4.1, the goal is to prove that vertices d and w are equivalent. Since no other equivalence has been chosen yet, the only necessary condition is for the two vertices to have the same label, which in this case is true (both are labelled J): the two vertices are thus equivalent.

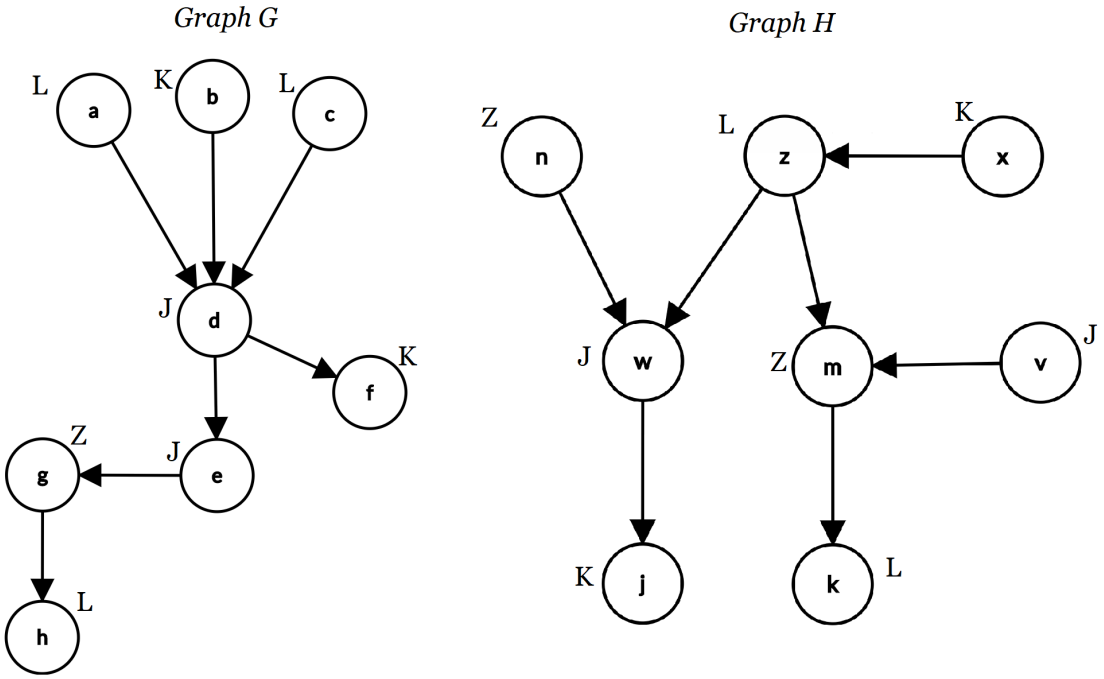


Figure 4.1: Starting state showing graphs G and H with no chosen equivalence, where right next to each vertex its label is shown.

Instead, if the situation was the one shown in figure 4.2, where an equivalence between vertices a and k has been chosen already, vertices d and w cannot belong to the same equivalence class any more, since d has a parent in an equivalence class (coloured in red), while w does not.

Lastly, in figure 4.3, an equivalence was chosen between vertices a and z, thus allowing d and w to be equivalent since they both have the same label and have parents belonging to the same equivalence class.

In order to apply an algorithm inspired by the Monte Carlo tree search to the MCS problem, several issues had to be addressed:

- How to handle graph vertices in order to choose equivalences among them conveniently?
- What do nodes in the Monte Carlo tree represent?

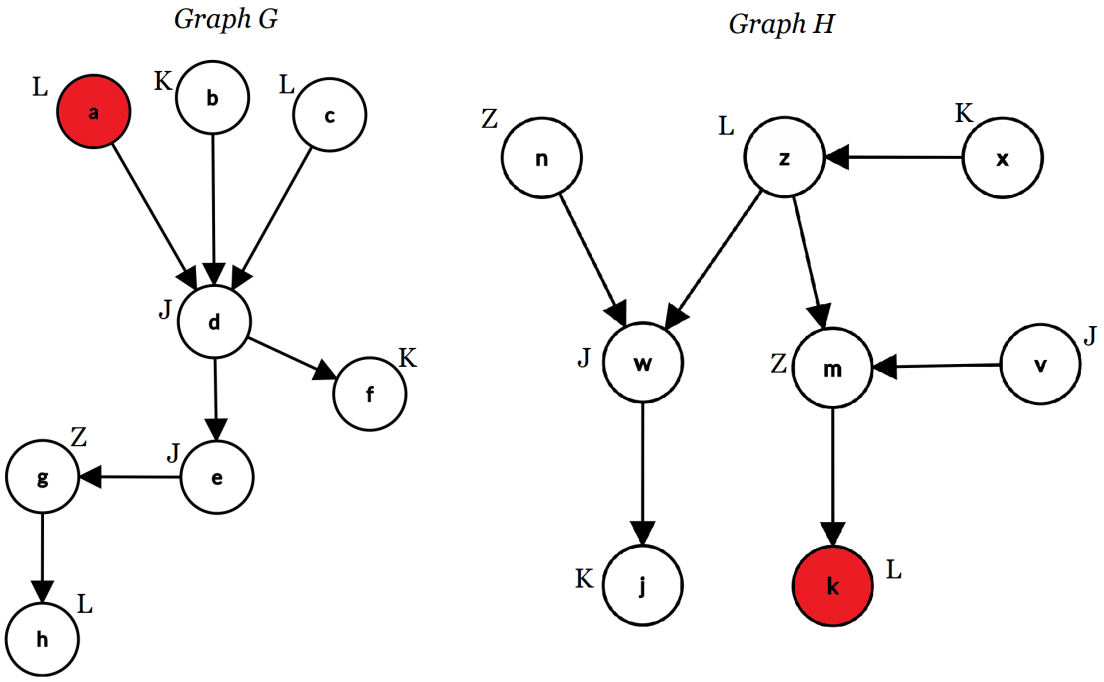


Figure 4.2: Graphs G and H with an equivalence found between vertices a and k, making the equivalence between d and w invalid.

- What is the estimated reward of a Monte Carlo node?
- Which policies have to be used to select the node to expand and to perform a simulation?
- Which values should be backpropagated?

Note: even though the term *node* can also be used to refer to a graph's vertices, only the latter will be used to refer to them, while *node* will always refer to nodes in the Monte Carlo tree.

4.2 Main Components

4.2.1 Hyper Partitions

One of the first addressed issues was how to manage the vertices of the input graphs in order to simplify and optimise (both in terms of speed and memory consumption) their storage and the choice of the equivalences. To solve this problem, the *hyper partitions* technique was conceived.

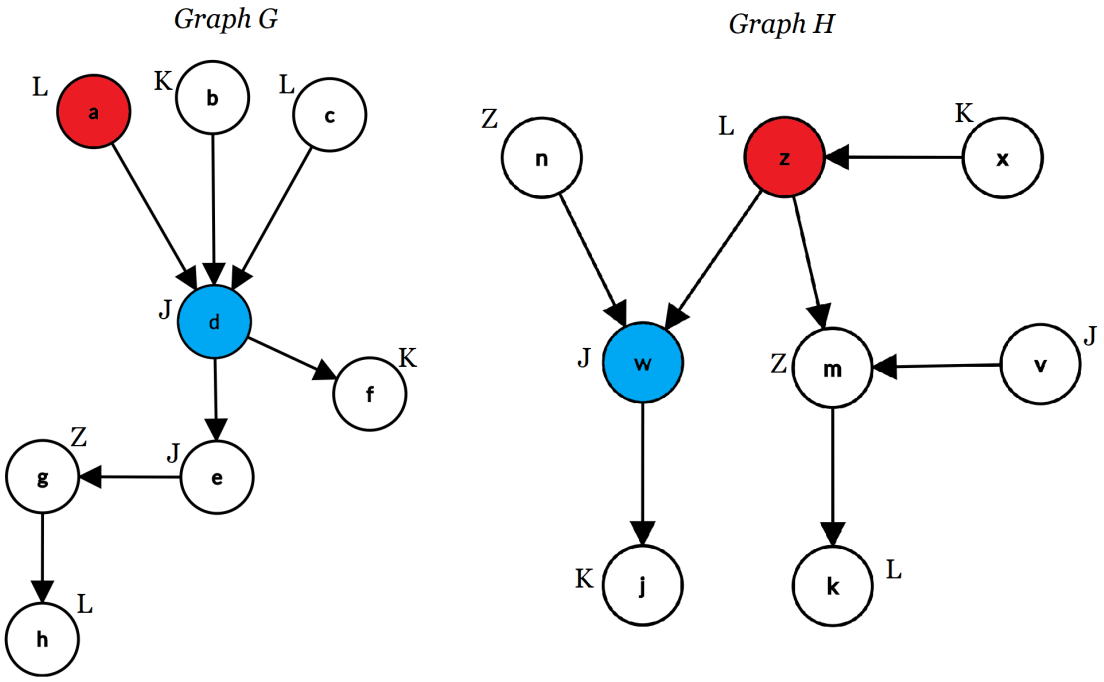


Figure 4.3: Graphs G and H with an equivalence found between vertices a and z, allowing to select the equivalence between d and w.

Split At the root of this technique, a split is performed: given some generic partitions containing vertices, a set S of some specific vertices is taken, and then each partition is split, removing from it the vertices contained in the set S, and placing them in a new partition. All the newly created partitions are then grouped under a new unique partition (hence the term *hyper partitions*): in this way, a node will have several *hyper partitions*, and it can choose one at random, pick a random vertex for each inner partition, and thus obtain a valid equivalence.

This technique allows choosing equivalences while maintaining a convenient data structure from which, given a *hyper partition*, it is always possible to safely choose new equivalences ranging from 2 to n vertices (based on how many inner partitions are then selected). Moreover, with this approach, it is also easy to discard vertices that cannot be any longer equivalent to any other vertex because of the restrictions applied after choosing an equivalence: in fact, given n input graphs, the *hyper partitions* which do not contain at least two non-empty partitions (i.e. that can obtain at least a two-vertices equivalence) can be discarded.

Partition hope The most important property of a *hyper partition* is the *hope*, whose value represents how many equivalences can probably be obtained from that

hyper partition. It is computed in the following way:

$$PH = \sum_{i=0}^{k} j^{m_i} * x_i \tag{4.1}$$

where:

- k is the number of equivalences that may be chosen in the *hyper partition* in the best possible combination, i.e. longer equivalences are preferred.
- $j \geq 1$ is a constant called *equivalence modifier* which gives more value to longer equivalences the higher it is chosen ($j = 1$ gives the same value to equivalences of different lengths)
- m_i is the length of the ith equivalence
- x_i is a modifier representing the likelihood of the ith equivalence

In particular:

$$x_i = x_{i-1} * h \tag{4.2}$$

with $x_0 = 1$ and $h < 1$ being a constant called *partition hope reducer*. More specifically the x_i modifier starts at one because the first equivalence can surely be chosen from that *hyper partition*, but then decreases at each sum, since there will be less probability to choose subsequent equivalences because of the new restrictions arising at each choice.

4.2.2 Monte Carlo Nodes

A node in the Monte Carlo tree represents a state of the game, which in the MCS case, consists of a set of already chosen valid equivalences, and a set of vertices still available to be picked. To represent these structures and to calculate the node's estimated reward, several properties have been defined for a Monte Carlo node.

Partition Manager A structure used to store and manage all the *hyper partitions* belonging to the node, thus representing the pool to choose the equivalences from.

History A structure containing all the already chosen equivalences.

Hope The *hope* of a node, like for the *hyper partition*, is a value that represents how many equivalences can hopefully be obtained from that node:

$$H = (\sum_{i=0}^{l} j^{m_i} * x_i) * c \tag{4.3}$$

which is similar to the *partition hope* formula, but now l represents the number of all the equivalences in all the partitions (still in the best possible combination). The constant $c > 0$ is called *node hope modifier*, used to regulate the final value of the *hope*.

If the *hope* of a node was just the sum of the *partition hopes*, its value would not have accurately represented the decrease of likelihood of an equivalence, since in the *partition hope* formula the x_i modifier is reset after the calculation on each partition, while in this case, it keeps decreasing after the sum of each equivalence of each partition.

In relation to the UCT formula, the *hope* corresponds to the exploration component, since a node with a high hope is probably able to be expanded a lot, obtaining lots of equivalences.

Fact The *fact* represents the value of obtained equivalences, i.e. the combined value of all equivalences in the *history*:

$$F = (\sum_{i=0}^{p} j^{m_i}) * f \tag{4.4}$$

where:

- p represents the number of chosen equivalences
- $f > 0$ is a constant used to balance the final value of the node's *fact*.

Here the x_i modifier is missing, since there is no uncertainty on those equivalences, being them already picked.

With respect to the UCT formula, the *fact* corresponds to the exploitation component, since it represents the certain current value obtained by that node.

Both *hope* and *fact* are balanced by a constant, and usually, especially for graphs with a high number of vertices, f is set to a higher value than c. The reasoning behind this choice is that for the first iterations of the algorithm, nodes' *hope* will always be higher than their *fact* since the *hope* will start at its maximum value, while the *fact* will start at 0 and grow slowly.

Weight The *weight* of a node is its estimated value, and it is a combination of its *hope* and *fact*:

$$W = H + F \tag{4.5}$$

This value is used to indicate how worthy is a node, combining both how much it can still be expanded and how good results it has already obtained.

Lastly, it is essential to note that only equivalences with a length of at least two (i.e. that involve at least two graphs) contribute to the various parameters (*partition hope*, node's *hope* and *fact*).

Figure 4.4 shows the results of the calculations for all the properties of a sample Monte Carlo node, where a few important things can be noted:

- Partition HP_{00} has a *partition hope* of zero since no equivalence of length of at least two can be made.
- In the partition HP_{01} the best possible combination of equivalences is one of length 3, and two of length 2, which leaves a vertex unchosen, and which will not thus contribute to the *partition hope*.
- Since the *equivalence modifier* j is equal to 2, the partition HP_2 has a higher *partition hope* than HP_{01}, even if it can form one less equivalence, but being them both of length 3, its final value is higher.

4.3 Algorithm

The algorithm inspired by the Monte Carlo tree search was conceived to perform better in this particular application, with one of the main peculiarities being that in the MCS problem there are no multiple players competing against each other, and no negative valued moves can be performed, only more or less worthy.

4.3.1 Initialization

Initial partitions Initially, given n input graphs, n partitions are created, each containing all the vertices of a given graph and they are stored under a unique *hyper partition*, like showed in figure 4.5, which is referred to the graphs in figure 4.1. The vertices in the partitions are then sorted by degree.

First split A first split is performed, with the aim of separating all the vertices based on their labels, thus, considering a total of m different labels, generating at

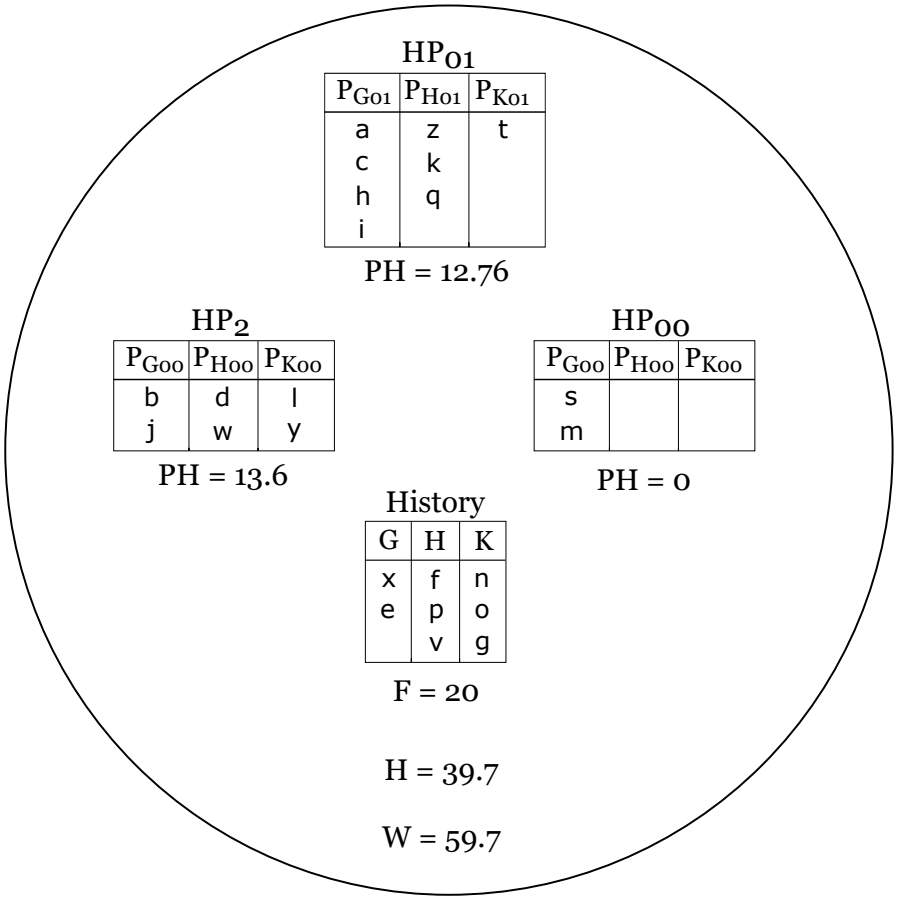


Figure 4.4: Example showing the values of the properties of a Monte Carlo node. For this example, the following constant values were used: $j = 2$, $h = 0.7$, $f = 1$, $c = 2$.

most $n * m$ separated partitions, and this is achieved by splitting the initial *hyper partition* m times, where at each iteration the set S_M contains all the vertices with a given label M. The result of this first split can be observed in figure 4.6.

After this initial split, partitions of different graphs which contains vertices with the same label, are grouped into separate *hyper partitions* (obtaining at most m different *hyper partitions*). This results in the possibility of choosing one *hyper partition*, then one random vertex for each inner partition, and always obtaining a valid equivalence.

This, and all the subsequent split that will be performed, will not alter the order of the vertices in the partitions: the sorting by degree will always be preserved.

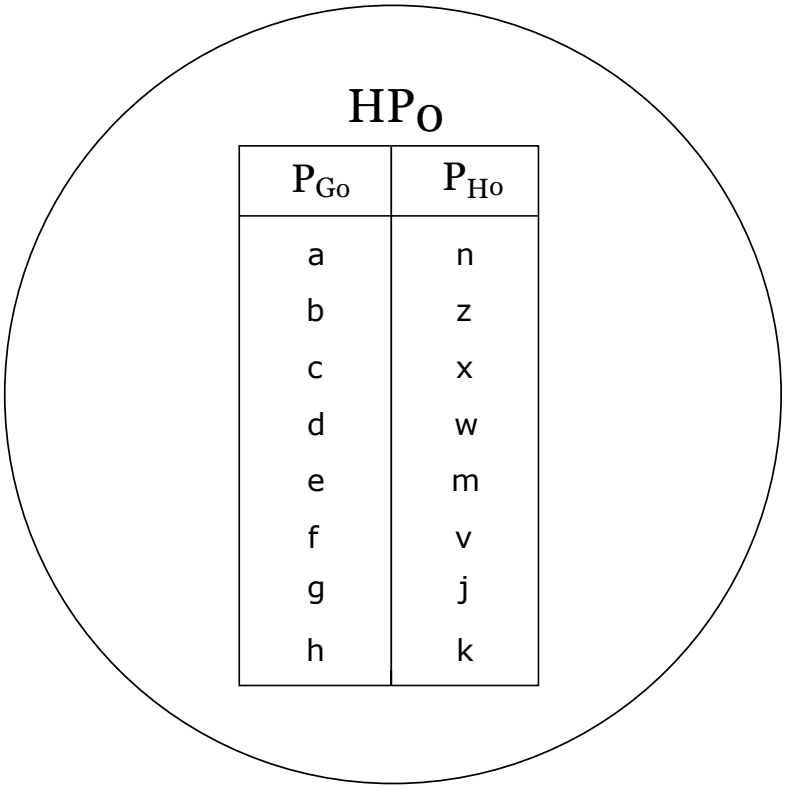


Figure 4.5: Initial state of the root node's *hyper partitions* after reading the input graphs.

Given this initial *hyper partitions*, the root node of the Monte Carlo tree is generated, calculating its *hope* (to which the *weight* will be equal since at this point its *fact* is zero), and it will be the starting point of the search.

4.3.2 Selection

At each iteration, a node to be expanded is selected. Besides the first iteration, where the only node that can be selected is a root node, the selection is made on a list of candidate nodes which possess two particular features:

- expandable: there is still at least one equivalence of two or more nodes that can be chosen
- promising: only nodes whose weight is higher than the *best fact* ever found are considered worthy of exploring

The only exception in admitting a non-promising node is when an input parameter that controls the minimum number of candidates available for expansion is set to

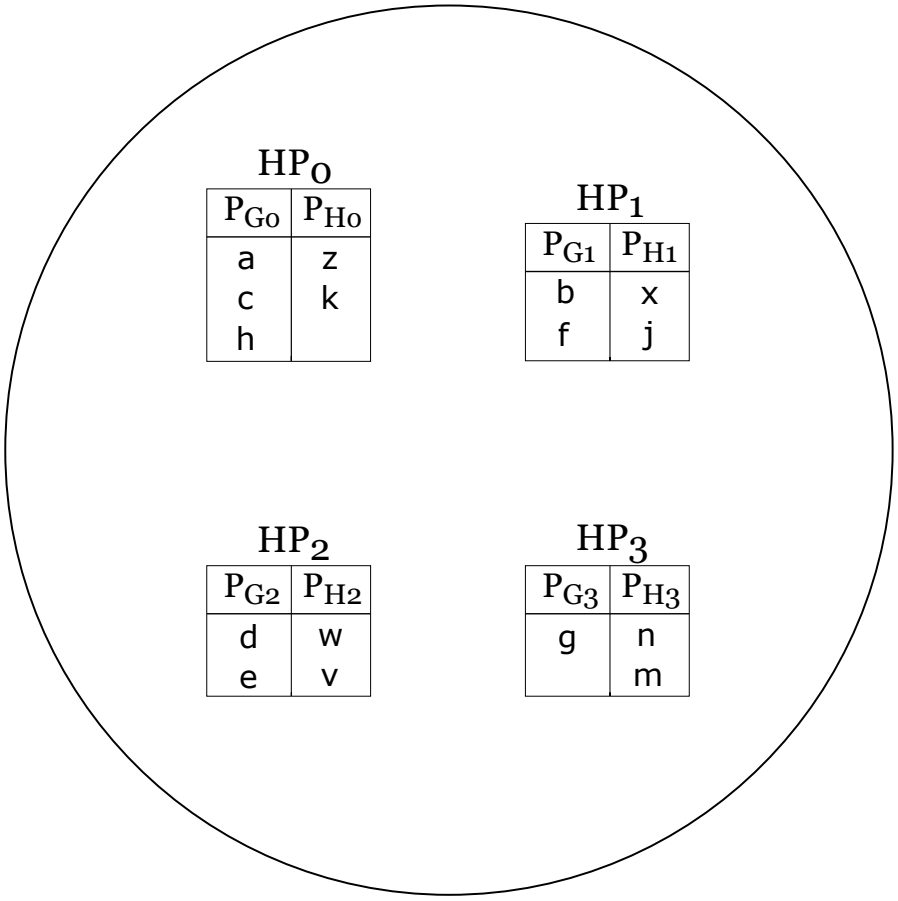


Figure 4.6: State of the root node after the first split based on vertices' labels.

a value greater than one: in this situation, if the number of candidates is not high enough, even if a node is not promising, it is considered a valid candidate.

To select a node to expand from the candidates, a fitness-proportionate selection algorithm is used, called *roulette wheel selection*, a genetic algorithm where each member of the population (in this case the candidate nodes) is allocated a section of an imaginary roulette wheel: each section of the roulette wheel has a different size, which is proportional to the individual's fitness. In this specific case, the fitness is represented by the *weight* of a Monte Carlo node, thus candidates with a higher *weight* will have a higher chance to be chosen when the roulette wheel is spun.

4.3.3 Expansion

Equivalence selection Once a node has been selected, a new equivalence has to be chosen in order to expand it. First, a *hyper partition* must be chosen, and this is again done using the *roulette wheel selection*, where the population is composed by the various *hyper partitions*, and the fitness is the *partitions' hope*. After spinning the wheel and picking the winning partition, an equivalence must be chosen: for each partition, a vertex is chosen using the tournament selection, which is another genetic algorithm used to select one individual among a population. In particular, several tournament rounds are run among a few individuals chosen at random, with the winner passing on to the next round. The winner is again the individual with the best fitness, and in this case, it corresponds to the vertex with the highest degree.

This equivalence selection step is repeated if one of the followings happen:

- given the selected *hyper partition*, if an equivalence formed by at least two vertices is not found
- the new node which will be formed with the new equivalence, has already been expanded (this may happen because different choices along the tree could lead to the same result)

If this happens, different partitions are selected from the same node until there are no more available partitions to pick vertices from: at this point, another hyper partition from the same node and its partitions are explored again. If no valid equivalence that leads to a new node can be found, the node is marked as *exhausted*, meaning that it has been fully explored, and it is thus removed from the candidates' list; the selection step is then performed again.

Instead, once a valid equivalence is found, the new node has to be expanded, thus the first step is to generate its new *hyper partitions*.

Subsequent splits The split technique is performed twice each time an equivalence is chosen because at this point the vertices' parents and children restrictions deriving from the new equivalence also need to be considered: the inner partitions are firstly split using as set the one containing all the parents of the vertices just picked for the equivalence, and then using the set containing all their children vertices. Lastly, the newly generated partitions are grouped into new *hyper partitions*. Considering again the graphs showed in figure 4.1, suppose the equivalence between vertices d and w has been chosen: the first split will be done using as set all the parent vertices of d and w, i.e. a, b, c, n and z, obtaining the result of figure 4.7, the second one on the set containing all their child vertices, i.e. e, f and j,

obtaining the final state showed in figure 4.8.

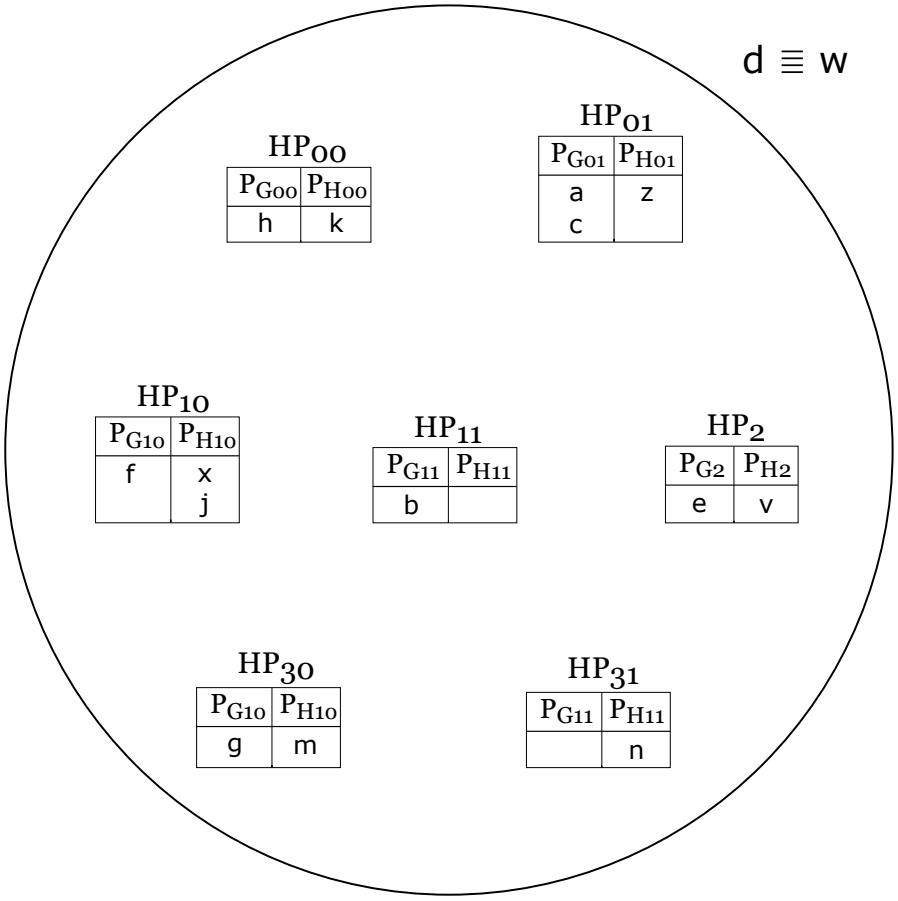


Figure 4.7: After choosing the equivalence between d and w, the *hyper partitions* are split based on the parent vertices of d and w.

At this point, the new node is initialised, and some conditions are checked to decide its course.

Exhausted First, there is a check on the node's *hope*: if $H = 0$ it means that no more equivalences can be obtained from this node, and it is thus marked as *exhausted* (it will not be included in the candidates list).

Bad weight The Monte Carlo tree keeps track of the highest *fact* reached by nodes, and uses this parameter to decide if the node is worthy to be explored, in which case the node is marked as *closed*, meaning that it was still expandable, but not considered valuable to.

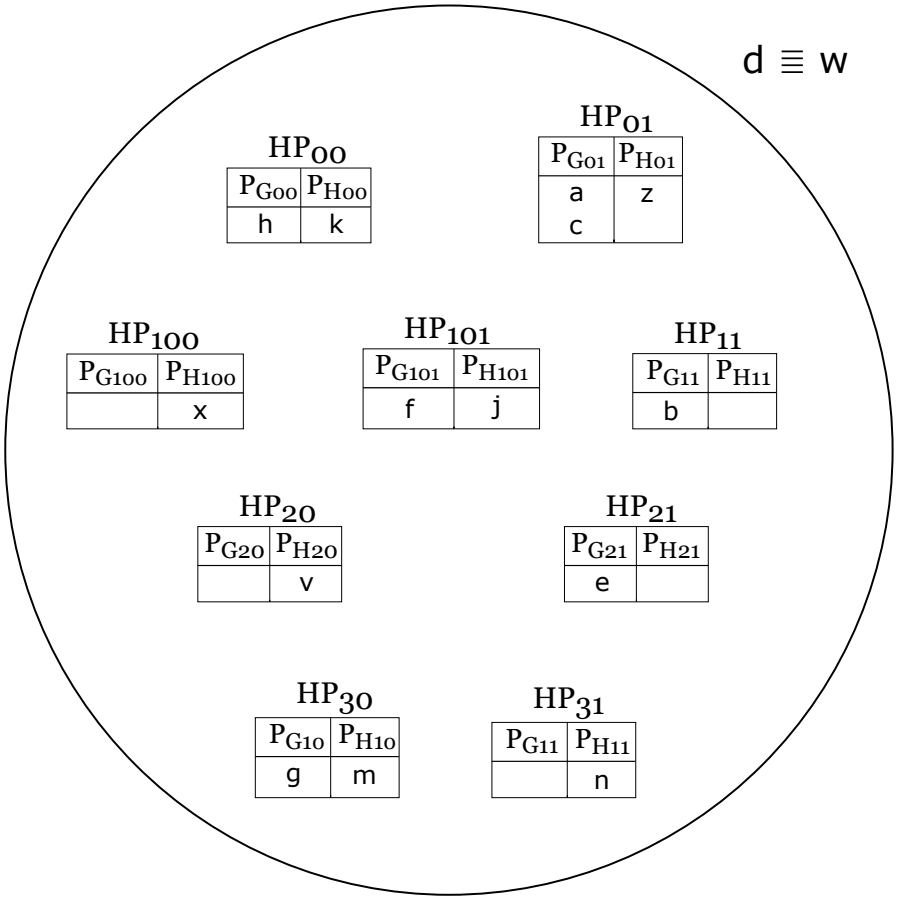


Figure 4.8: Right after splitting on their parents' vertices, another split is performed on the children of d and w, obtaining the final state of the Monte Carlo node.

If it is true that:

$$W * b < F_{max} \tag{4.6}$$

the node is not considered valuable, and it is *closed*. The constant $b > 0$ is called *bad weight ratio* and, based on its value, increases or decreases the ratio of *closed* nodes.

There are two cases in which even a node with *bad weight* can be inserted among the candidates:

- a minimum number of candidates is set, and it is currently not reached

- there is a node among the candidates whose *weight* is lower than the one of the just expanded node

In the second case, the former node is removed from the candidates and marked as *closed*, and the one just expanded is inserted in the candidate list.

It is important to note that since at each expansion a new *best fact* may be found, when the selection process starts and a candidate is selected, before expanding it, a *bad weight* check is performed on it, that if fails will exclude the node from the candidates (except if the minimum number of candidates is not met), mark it as closed, and proceed in selecting another candidate.

4.3.4 Simulation

Contrary from the classic MCTS algorithm, no simulation is performed in this case, since reaching an end state for every expanded node has been found to be counterproductive in such a scenario with a high branching factor, and even simulating for a finite number of levels (proceeding as in the expansion phase) did not improve the results. Thus the chosen approach is to expand nodes one-by-one.

4.3.5 Backpropagation

Each time a new node is created, the difference between its hope and the one of its parent is computed. If this value is negative, this difference is propagated back up till the root node:

$$P = H - H_p \tag{4.7}$$

where:

- P is the initial value to be propagated (if $P \geq 0$ there is no propagation)
- H is the hope of the newly created node
- H_p is the hope of its parent

This value is backpropagated in the following way:

$$H_{pi} = H_{pi} + P * s \tag{4.8}$$

where:

- H_{pi} is the *hope* of the ith parent node, climbing the tree back to the root node
- s is a variable used to reduce the propagated value as it goes up, starting at $s = v$ and being modified in this way: $s = s * r$ (with $v \leq 1$, *propagation initial value*, and $r < 1$, *propagation reducer*, being constants) after each propagation

The reasoning behind the propagation conditions is that if the number of vertices in the examined graphs is very high (and also if the *partition hope reducer* h is set to a low value), the values of the nodes' *hope* is flattened, thus if expanding a node, i.e. selecting an equivalence and splitting its hyper partitions, results in a new node whose *hope* is still the same as its parent's, it means that probably that is a path which will lead to lots of expansions, thus valuable.

Figure 4.9 shows an example of backpropagation after the expansion of a new node.

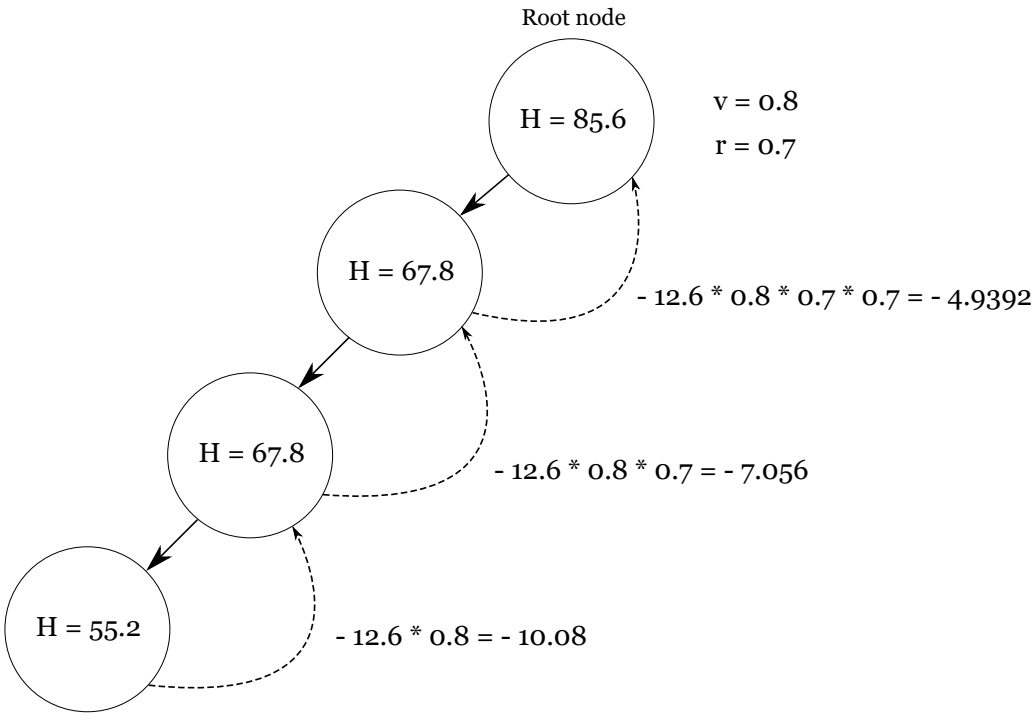


Figure 4.9: Example showing how P is calculated and backpropagated after the expansion of a new node, considering $v = 0.8$ and $r = 0.7$ (for the sake of simplicity only the affected path of the tree is shown).

4.3.6 Termination

There are two termination conditions:

- Timeout: once the timeout given as input expires, the program stops expanding the tree and compute the final results
- Exhaustion: all the nodes in the tree have been either marked as closed or exhausted

It is important to note that when the program is terminated by exhaustion, it does not mean that all the possible existing nodes of the Monte Carlo tree have been explored, but only those who could be generated given that particular input parameters: by changing them, there could be a much bigger exploration (which could even lead to worse results) or a much smaller one.

Once the search is over, all the statistics are computed, among which the final result that is represented by an equivalence index e, where $0 \leq e \leq 1$. This index represents how similar the input graphs are, and it is computed comparing the obtained results with the best possible case, i.e. given n input graphs, x equivalences of length n (x being the number of nodes of the smallest graph), y equivalences of length $n-1$ (y being the number of nodes of the second smallest graph minus x), and so on.

4.3.7 Parameters

Several input parameters can be set (some of which have already been introduced) in order to influence the tree search:

- *Timeout*: defines after how many seconds the search should stop. If set to zero, the search will continue until exhaustion.
- *Equivalence modifier*: influences the value of equivalences. Set to one makes equivalence of different length the same value, while it gives more value to longer equivalences the higher it is set.
- *Hope reducer*: reduces the value of subsequent equivalences in the calculation of the *partition hope* and node's *hope*, representing the decrease in the likelihood of the feasibility of an equivalence because of the various restrictions applying each time one is chosen.
- *Node hope modifier*: modifies the value of a Monte Carlo node's *hope* after it has been computed.
- *Fact modifier*: modifies the value of a Monte Carlo node's *fact* after it has been computed.

- *Bad weight ratio*: a value with whom the *weight* of a node is multiplied when it is compared with the *best fact*. A value greater than one closes fewer nodes, lesser than one, increases the number of closed nodes.
- *Propagation initial value*: a multiplier for the *hope* difference between a new node and its parent (the value to be propagated) that defines the initial value that is actually propagated.
- *Propagation reducer*: reduces the value to be propagated each time it is propagated up one level of the tree.
- *Minimum candidates*: sets the minimum number of candidates to be expanded, thus accepting even nodes with a *bad weight* in case this minimum is not satisfied.
- *Vertex labels*: a boolean value that if set to one enables the labels on vertices, if set to zero disables them. In the latter case, there are no restrictions based on labels to form equivalences, only those based on the edges between vertices.

Additional details on these parameters are available in table 5.1.

Chapter 5

Implementation

5.1 Main Data Structures

5.1.1 Partition Manager

The *hyper partitions* of a node are managed by an object called *Partition Manager*, which contains the *data* vector, a vector of all the vertices (represented by integers) still available in that particular node, and a vector of *Partitions*.

Each *Partition* is composed of a *partition hope*, that estimates the value of that partition, and a vector of *partition heads*, which is at most of length n, where n is the number of input graphs.

These structures are organised so that each value in the *partition heads* vector represent a certain vertex and it also corresponds to an index on the *data* vector, which is another vertex belonging to the same partition as the head. The just found vertex will behave the same way, representing both a particular vertex and the index for the next one, thus forming a chain containing all the vertices of a graph belonging to a certain partition. The end of the chain is specified by the special value -1. An example of this structure is shown in figure 5.1, which also includes a view of the node as a Venn diagram.

This kind of structure allows to obtain a very good trade-off between memory consumption and access times.

5.1.2 History

A *History* contains a set of vectors of integers. The integers represent the vertices, and the vectors of vertices constitute the chosen equivalences. The choice of the

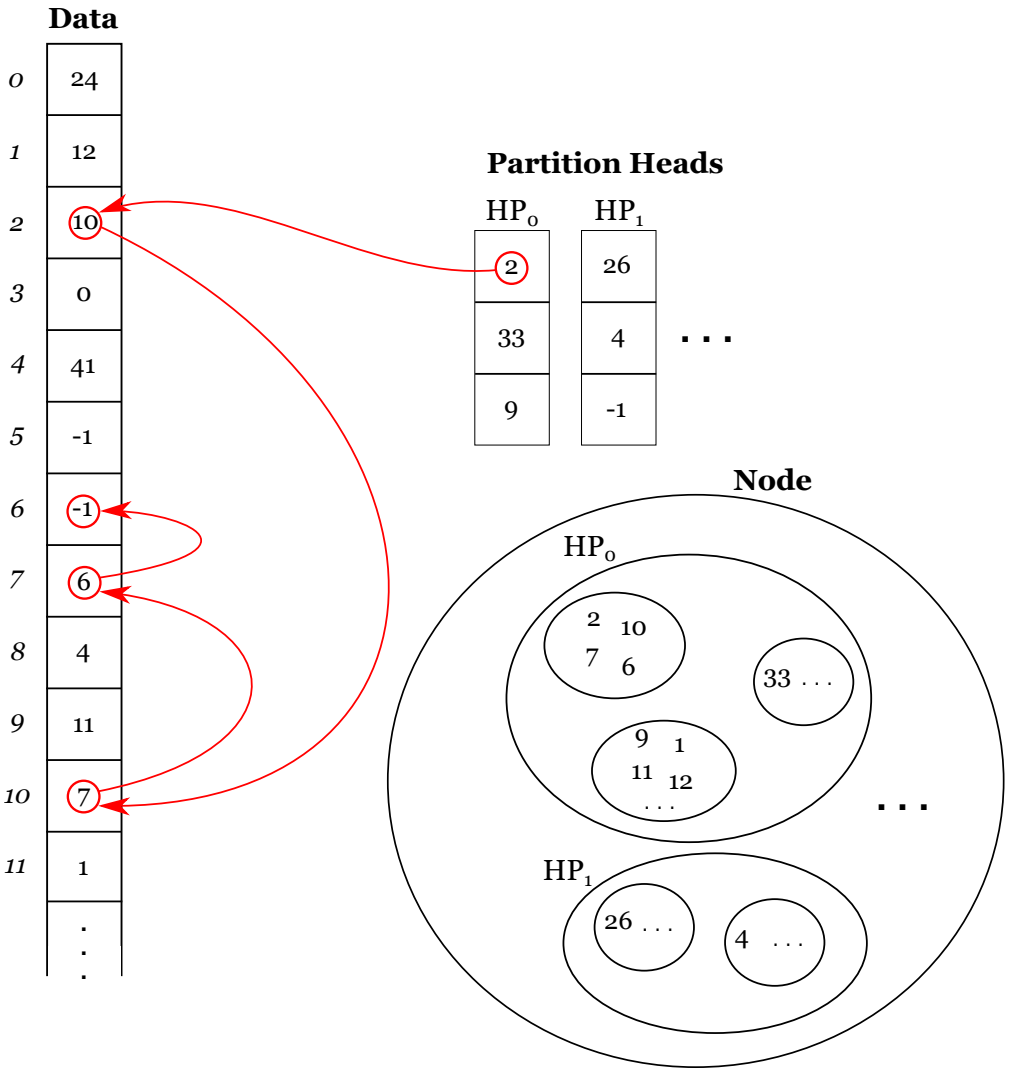


Figure 5.1: Example showing how the chains of data are linked (for the sake of clarity only one chain is explicitly showed). It is also worthy to note that while equivalences of length 3 can be obtained selecting partition P_0, one of the heads of partition P_1 is equal to -1, thus that particular graph has no vertices on that partition, so at most equivalences of length 2 can be obtained from P_1.

set was due to the fact that in this way the equivalences are ordered according to a certain criterion (it is not important which particular one), thus it is possible to detect Monte Carlo nodes that reached the same equivalences even if they followed different paths on the tree, thus avoiding duplicate nodes.

5.1.3 Node Selector

The *Node Selector* is composed by a vector of pointers to the nodes that can be selected for expansion and a double value that is the sum of all their weights. It is responsible for selecting a node to expand among the valid candidates: a random number x, with $0 < x < 1$ is selected, and then the normalized weight of each candidate is subtracted from it: as soon as x goes below 0, the node that whose weight was just subtracted is chosen as candidate.

5.1.4 Monte Carlo Node

A Monte Carlo node simply contains a *Partition Manager* relative to its hyper partitions, a pointer to its *History* (which is stored in the Monte Carlo tree), a vector of pointers to its child nodes and two more pointers, one towards its parent node and another towards the Monte Carlo tree.

Lastly, it contains three double values: *hope*, *fact* and *weight*, which have already been discussed in section 4.2.2.

5.1.5 Monte Carlo Tree

The Monte Carlo tree contains a vector of the input graphs, a *Node Selector*, pointers to the root node and to the one with the best *fact*, and a map containing the key-value pairs *History-pointer to the node*.

It also contains a *Monte Carlo Tree Helper* which is in charge of handling computation, storage and output of all the statistics.

5.2 Input Parameters

Below a table is shown which contains all the input parameters of *Gamy* (only excluding the input graphs and the results folder paths).

For each parameter data type, value constraints and direct effects (as the parameter value is increased) on the various properties are shown.

Table 5.1: Settable input parameters.

Parameter	**Type**	**Constraints**	**Effects on growing**
Timeout t	unsigned int	$t \geq 0$	/
Equivalence modifier em	double	$em \geq 0$	*partition hope* ↑ node's *hope* ↑ *fact* ↑
Hope reducer hr	double	$0 < hr < 1$	*partition hope* ↓ node's *hope* ↓
Node hope modifier nhm	double	$nhm > 0$	node's *hope* ↑
Fact modifier fm	double	$fm > 0$	*fact* ↑
Bad weight ratio bwr	double	$bwr > 0$	/
Propagation initial value piv	double	$0 < piv \leq 1$	node's *hope* ↓
Propagation reducer pr	double	$0 < pr < 1$	node's *hope* ↓
Minimum candidates mc	unsigned int	$mc \geq 1$	/
Vertex labels l	unsigned int	$l == 0 \vert 1$	/

5.3 Pseudocode

Main

Input: *search_params*, *input_graphs*
Result: initialise and start Monte Carlo tree search
read *input_graphs*;
create *initial partitions* with graphs' vertices;
sort *initial_partitions* by vertices degree;
`//vertices with more edges are first in the partitions`
get a list *labels_list* of all the labels;
for *label l in labels_list* **do**
    get a set *vertices_l* of all the vertices with label *l*;
    split *initial_partitions* on *vertices_l*;
end
create root node *root_node* using *initial_partitions*;
initialise Monte Carlo tree with *root_node* and *search_params*;
start **Monte Carlo tree search**;

Search

Object: Monte Carlo Tree
Result: perform Monte Carlo tree search
while *timeout IS NOT expired AND tree IS NOT exhausted* **do**
    if *size of candidates > 0* **then**
        select a node *n* using roulette wheel algorithm on candidates;
        `//candidates with higher weight have higher chances of being selected`
        if *n HAS NOT* ***bad weight*** **then**
            expand *n*;
            if *n IS exhausted* **then**
                remove *n* from candidates;
            end
        else
            remove *n* from candidates;
        end
    else
        exhausted = TRUE;
        `//no more candidates means that the tree is exhausted`
    end
end

Expand

Object: Monte Carlo Node
Input: node n to expand
Output: is n still expandable
Result: expand the selected node

```
copy node's hyper partitions in hps_copy;
while size of hps_copy > 0 do
    select an hyper partition hp using roulette wheel algorithm;
    //hyper_partitions with higher hope have higher chances of
      being selected
    while TRUE do
        create new empty equivalence new_eq;
        for partition in hp do
            if partition IS NOT empty then
                select vertex in partition using the tournament selection
                 algorithm;
                remove vertex from partition;
                //this does not modify the original structure since
                  it is done on a copy
                insert vertex into new_eq;
            end
        end
        if size of new_eq < 2 then
            break;
            //an equivalence of at least two can't be made from this
              hyper partition so another one is chosen
        end
        create a copy of the node's history history_copy;
        add new_eq to history_copy;
        if node with history history_copy DOES NOT exist in the tree then
            create new node;
            return TRUE;
            //node is still expandable
        end
    end
end
return FALSE;
//node can't be expanded anymore
```

Split

Object: Partition Manager
Input: set *vertices_set* of vertices to split on
Result: hyper partitions split according to *vertices_set*
create new empty hyper partition *new_hp*;
for *hyper partition current_hp in hyper partitions* **do**
    create new empty partition *new_part*;
    for *partition head part_head in current_hp* **do**
        set *part_head* as *current_vertex*;
        while *current_vertex IS NOT chain end* **do**
            if *current_vertex IS IN vertices_set* **then**
                insert *current_vertex* into *new_part*;
                relink the modified data chain excluding *current_vertex*;
            end
            current_vertex = next vertex in the chain;
        end
    end
    if *new_part IS NOT empty* **then**
        calculate *new_part* hope;
        insert *new_part* into *new_hp*;
    end
    if *current_hp IS NOT empty* **then**
        recalculate *current_hp* hope;
        `//hp hope needs to be recalculated since some of its vertices have been removed`
    else
        discard *current_hp*;
    end
end
add *new_hp* into the node's hyper partitions list;

Create New Node

Input: node to expand *n_exp*, new equivalence *new_eq*
Result: create a new node in the tree
copy n_exp partitions in *new_partitions*;
for *vertex v in new_eq* **do**
  remove *v* from *new_partitions*;
end
create empty list *from_vertices*;
create empty list *to_vertices*;
for *vertex v in new_eq* **do**
  insert all parent vertices of *v* into *from_vertices*;
  insert all children vertices of *v* into *to_vertices*;
end
split *new_partitions* on *from_vertices*;
split *new_partitions* on *to_vertices*;
create *new_node* using new_partitions;
copy history of *n_exp* into *new_history*;
calculate *new_node* hope, fact and weight;
if *new_node HAS NOT* ***bad weight*** **then**
  add *new_eq* to *new_history*;
  set *new_node* history equal to *new_history*;
  set *new_node* parent equal to *new_node*;
  insert *new_node* into nodes list;
  add *new_node* to *n_exp* children;
  if *new_node IS the new best solution* **then**
    update best fact with *new_node* fact;
  end
  if *new_node hope IS NOT EQUAL TO 0* **then**
    insert *new_node* into candidates;
  end
end

Bad weight

Object: MonteCarloTree
Input: node n
Output: has n bad weight
Result: determine if n has a bad weight, i.e. if it is not worthy to expand

```
if n IS IN candidates then
    if size of candidates < MIN_CANDIDATES OR fact of n IS EQUAL TO
     best fact in the tree OR weight of n * BAD_WEIGHT_RATIO < best
     fact in the tree then
        return false;
    else
        return true;
    end
else
    if weight of n * BAD_WEIGHT_RATIO < best fact in the tree then
        for candidate c in candidates do
            if weight of c < weight of n then
                remove c from candidates;
                add n to candidates;
                return false;
            end
        end
        return true;
    else
        return false;
    end
end
```

Chapter 6

Experimental analysis

6.1 Graphs Sets

Several experiments were run in order to evaluate the performance and the quality of *Gamy*. In particular, tests were performed on four sets of graphs:

- $\mathcal{S}$: small graphs with at most 100 vertices, with an edges to vertices ratio ranging from 10:1 to 300:1
- $\mathcal{B}$: larger graphs with a number of vertices ranging from 2710 to 4882, with an average edges to vertices ratio of 2:1
- $\mathcal{G}$: graphs with a number of vertices ranging from 700 to 4000, with a high average edges to vertices ratio, ranging from 173:1 to 1658:1
- $\mathcal{M}$: graphs generated from a base graph of 2710 vertices (taken from set $\mathcal{G}$), to which vertices and edges were added, maintaining the same edges to vertices ratio

All the graphs are in *dimacs* format, and they are *vertex-labelled* and *directed*.

Graphs belonging to the $\mathcal{B}$ set (and thus to $\mathcal{M}$) are graphs that represent Android applications. An example of a small *undirected* graph of this type is shown in figure 6.1.

6.2 Tests

The tests that were run are classified in the following way:

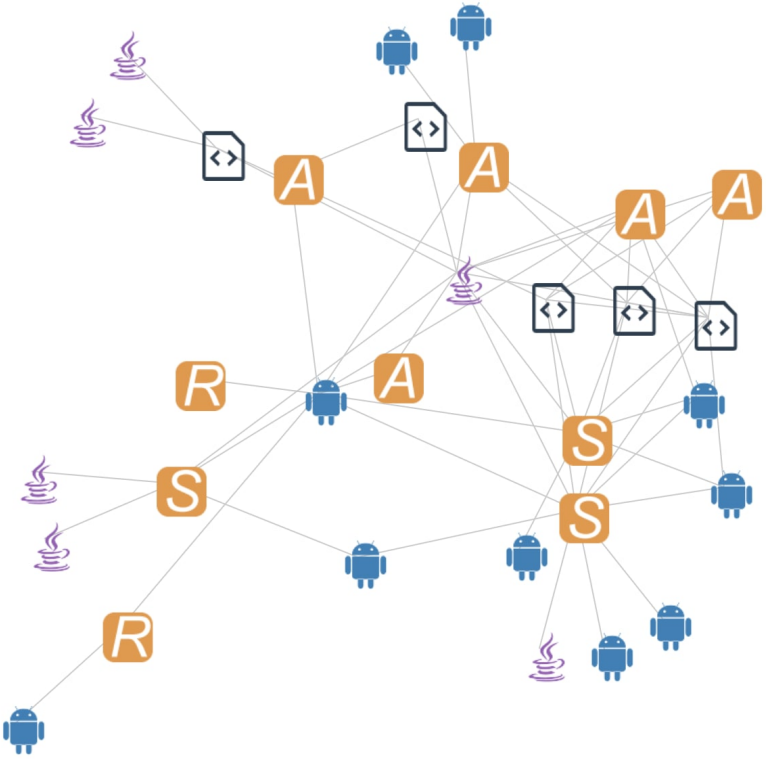


Figure 6.1: Graph representing an Android program, where the various icons represent vertices with a certain label.

- T_1: couples of all the graphs combination from set $\mathcal{S}$ with a timeout of 100 seconds
  - T_{1_1}: directed edges, labels on vertices enabled
  - T_{1_2}: undirected edges, labels on vertices enabled
  - T_{1_3}: directed edges, labels on vertices disabled
  - T_{1_4}: undirected edges, labels on vertices disabled
- T_2: couples of all the graphs combination from set $\mathcal{B}$ with a timeout of 100 seconds
  - T_{2_1}: directed edges, labels on vertices enabled
  - T_{2_2}: undirected edges, labels on vertices enabled
  - T_{2_3}: directed edges, labels on vertices disabled

  - T_{2_4}: undirected edges, labels on vertices disabled

- T_3: couples of all the graphs combination from set $\mathcal{G}$ with directed edges, labels on vertices enabled and a timeout of 10 seconds
- T_4: given any couple of graphs $(G, F) \in \mathcal{B}$ tests on the following combinations:

  (G, F)

  (G, G, F, F)

  (G, G, G, F, F, F)

  (G, G, G, G, F, F, F, F)

  with directed edges, labels on vertices enabled and a timeout of 100 seconds
- T_5: tests using as input 2 to n graphs from set $\mathcal{M}$, with n being the number of graphs belonging to $\mathcal{G}$, with directed edges, labels on vertices enabled and a timeout of 200 seconds

Tests T_1, T_2, T_3 were run on both *Gamy* and *McSplit*, to compare the differences between the results obtained by the two algorithms.

More specifically, tests T_1 were performed to observe how *Gamy* performs in a scope in which it is not optimised to run since exhaustive algorithms perform better on smaller graphs.

The most significant tests are T_4 and T_5, since no other algorithm is able to compute the MCS between multiple graphs simultaneously. For this reason, there will be no other algorithm that can be used to compare the results obtained by *Gamy* in these tests.

All the tests run using *Gamy* were run multiple times with different input parameters, selecting then the best results obtained.

6.3 Results

Tests on small graphs In T_1 tests, *Gamy* obviously never managed to find more equivalences than *McSplit*, but surprisingly it always got really close to it, with just 1-3 fewer equivalences found, and in some cases even finding the same number.

Even if *McSplit* is more reliable with this graphs, it needs a correct timeout to be set in advance, since setting it too low may not result into finding many equivalences, while setting it too high leads to lot of wasted computational time. On the other

hand *Gamy* prunes the search space, stopping itself in very short times and finding a solution which is really close (or the same) to the one found by *McSplit*.

This is emphasised in tests T_{1_2}, T_{1_3} and T_{1_4}, where *McSplit* almost always reaches the timeout, while *Gamy* terminates.

The averages of equivalences found and of computational time of both *Gamy* and *McSplit* are shown in the following table:

Table 6.1: Averages of equivalences found ($\overline{Eqs}$) and of computational time ($\overline{T}$) running T_1 tests.

Test	**Gamy** $\overline{Eqs}$	**McSplit** $\overline{Eqs}$	**Gamy** $\overline{T}$	**McSplit** $\overline{T}$
T_{1_1}	6,24	7,54	0,23	528,31
T_{1_2}	8,04	9,99	0,30	50031,78
T_{1_3}	9,72	11,39	0,35	20313,29
T_{1_4}	6,33	7,54	0,25	527,54

As already said, *McSplit* found more equivalences on average, but it also spent lot more time running. Reducing the timeout for the tests on *McSplit* would have probably left the number of equivalences found almost unaltered, but this is not something that can be known a priori.

Tests on big graphs Like shown in table 6.2, also on graphs belonging to the $\mathcal{B}$ set, *McSplit* provides better results on average, while *Gamy* remain faster in the execution time.

Actually, taking a look at the actual results in table 6.3, it is possible to notice that when the difference of vertices between two graphs becomes high, *Gamy* is able to obtain more equivalences, often in a very short time. More tests were made, raising the timeout for *McSplit* to 3000 seconds for those same input graphs, but it was not able to find more equivalences than it already had.

Table 6.2: Averages of equivalences found ($\overline{Eqs}$) and of computational time ($\overline{T}$) running T_2 tests.

Test	**Gamy** $\overline{Eqs}$	**McSplit** $\overline{Eqs}$	**Gamy** $\overline{T}$	**McSplit** $\overline{T}$
T_{2_1}	1878,47	2307,20	56317,07	60374,87
T_{2_2}	1894,40	2338,20	38347,47	62228,73
T_{2_3}	1887,80	2378,40	49831,80	60388,40
T_{2_4}	1907,20	2307,20	36232,87	60375,40

Table 6.3: Extract of results obtained from tests T_{2_4}, where rows in which *Gamy* obtains better results than *McSplit* are highlighted.

		Gamy			***McSplit***		
G_0	G_1	**N° Eqs**	**Eq index**	**Time**	**N° Eqs**	**T**	**Eq Diff**
2762	2710	1884	0,695203	100000	2550	100007	-666
2765	2710	1878	0,692989	42570	2574	100006	-696
2765	2762	1869	0,676684	1426	2761	306	-892
2766	2710	1859	0,685978	10810	2574	100006	-715
2766	2762	1885	0,682476	1474	2761	133	-876
2766	2765	1889	0,683183	9133	2765	96	-876
4882	**2710**	**1919**	**0,708118**	**21038**	**1509**	**100456**	**410**
4882	**2762**	**1914**	**0,692976**	**100001**	**1574**	**100402**	**340**
4882	**2765**	**1962**	**0,709584**	**1924**	**1559**	**101637**	**403**
4882	**2766**	**1951**	**0,705351**	**10372**	**1574**	**101687**	**377**
2762	2710	1876	0,692251	40942	2548	100007	-672
2762	2762	1920	0,695148	1641	2762	103	-842
2762	2765	1923	0,696235	100088	2761	125	-838
2762	2766	1896	0,686459	100134	2761	261	-865
2762	**4882**	**1983**	**0,717958**	**1940**	**1575**	**100399**	**408**

Tests on large dense graphs Table 6.4 shows the averages for tests T_3: here the gap in equivalences found is much shorter, and while the one in computational time also is, *McSplit* went in timeout almost in all cases, while *Gamy* didn't.

The results on this tests confirmed that *Gamy* is able to find more equivalences when the difference in vertices between the input graphs is bigger, but highlighted that it is able to find more also when the graphs are not that large but are really dense.

Table 6.4: Averages of equivalences found ($\overline{Eqs}$) and of computational time ($\overline{T}$) running T_3 tests.

Test	**Gamy** $\overline{Eqs}$	**McSplit** $\overline{Eqs}$	**Gamy** $\overline{T}$	**McSplit** $\overline{T}$
T_3	14,78	15,48	7775,714	9857,04

Tests with the same graphs multiple times in input This tests (T_4) were aimed at analysing the stability of *Gamy* on the feature that makes it unique, i.e. being able to work on multiple graphs as input. In particular, the goal was to obtain a similar equivalence index when running the algorithm on the various

combinations of the same couple of graphs: the results confirmed this hypothesis since all the tests on different couples, have a really low variance among the equivalence indexes of the various combinations.

Tests with large graphs sharing a common base This last group of tests (T_5) were performed on graphs belonging to set $\mathcal{M}$, that share a common base of 2710 vertices: this means that given any combination of input graphs, the number of longest equivalences - which is equal to the number of input graphs - is guaranteed to be at minimum 2710 (additional vertices and edges were randomly added to the graphs).

Table 6.5 shows the obtained results, and a fair amount of the minimum guaranteed equivalences were found, with an average of 66,36%.

Table 6.5: Extract of results from tests T_5, where **LE** represents the number of longest equivalences found, **PT** the percentage of **LE** over the total of 2710, **TE** is the number of total equivalences found (also considering the other lengths) and **EI** represents the equivalence index.

Input graphs vertices	LE	PT	TE	EI
3010 - 3610	1911	70,5	1911	0,635
3010 - 3410 - 3610	1807	66,7	2107	0,610
2810 - 3010 - 3410 - 3610	1764	65,1	2193	0,633
2810 - 3010 - 3410 - 3610 - 3710	1765	65,1	2318	0,624
2810 - 3010 - 3410 - 3610 - 3710 - 4010	1767	65,2	2450	0,621
2810 - 3010 - 3210 - 3410 - 3610 - 3710 - 4010	1776	65,5	2517	0,622

Concluding, the tests that were run also showed that *Gamy* has a low average RAM usage (considering the problem it is facing) of around 35 MB (for both small and larger graphs), with the only outsiders being the tests on dense graphs, which were characterized by an average RAM usage of 88 MB.

Chapter 7

Conclusions

Gamy is still an immature algorithm, but the results showed its potential and applicability.

Even when employed in cases for which it was not specifically designed for, like comparing small graphs, it obtained fair results in excellent times, even being able to overtake *McSplit* on larger or denser graphs.

In its main application scenario, i.e. comparing any number of graphs simultaneously, it obtained great, consistent results.

It is also worthy mentioning that *Gamy* is a really flexible algorithm, since tuning its parameters correctly, it can employ a greedy approach, that provides highly approximated results, but with a minimal computational time and memory usage, or a more exhaustive one, obtaining a behaviour closer to exact algorithms, which obtains more precise results, but requiring longer times and more memory.

7.1 Future work

The proposed algorithm has large room for improvement, and many areas that can be enhanced have already been identified.

7.1.1 Input Parameter Calibration

The algorithm uses lots of input parameters that influence the search. Although they are strongly linked among themselves, each of them influences unique aspects of the search, and for each set of input graphs, various combinations of them were tried. Still, since there were lots of graphs combinations, specifying many values

for each input parameter would have lead to a drastic increase in testing time. Considering:

- 9 input parameters (timeout is not included)
- 3 different values for each input parameter
- 20 different graphs
- graphs tested only in couples

each couple of graph would have been tested 19683, so with 190 different combinations (for 20 graphs) and a timeout of 100 seconds for each graph, this would have resulted in a test time of 103882.5 hours, which are almost 12 years.

A more focused study will be performed in order to better understand the relationship among them and with the input graphs structure, possibly performing a prior analysis to tune the parameters dynamically based on some features of the graphs that have to be analysed.

7.1.2 Simulation

The simulation phase is not included in the algorithm, and the selected node is just expanded of one level, using the newly generated node as new solution.

In fact a well-performing simulation phase could help to direct the search towards lucrative paths, avoiding less worthy ones.

A valid solution may be using a greedy algorithm in the simulation phase using three stop conditions:

- time: a certain amount of time is passed
- nodes: a certain number of nodes down the tree have been explored
- exhaustion: the simulation path can't be expanded any more

Work will be made in developing and analysing this approach.

7.1.3 Parallelisation

The algorithm has been implemented in single thread, but it can drastically improve its performances since it can be parallelised on different levels (as already discussed in paragraph 3.5):

- *Leaf parallelisation*: given an implemented simulation phase, several different ones can be run in parallel from the selected node, gathering more significant information about that path.
- *Root parallelisation*: several independent search trees are built, and information about branches, equivalences and nodes' statistics are exchanged among them, so that more thoughtful choices can be made during the search.
- *Tree parallelisation*: different nodes are selected and then expanded in parallel, speeding up the search.

Acknowledgements

To my professors, Giovanni Squillero and Stefano Quer for all the support they gave me during these months.

To my parents and my brother, who were always there to help me on everything, never leaving me alone.

To my friends, the old ones from my home town, and the new ones I made here in Turin, who were with me during every good and bad moment of my path, and will always be.

Without all of these people, I would not be where I am and who I am, so I would like to truly thank you from the bottom of my heart.

Contents

List of Tables

List of Figures

Bibliography

[1] Harry G. Barrow and Rod M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. Inf. Process. Lett., 4(4):83–84, 1976.

[2] Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). Commun. ACM, 16(9):575–576, 1973.

[3] R Morpurgo. Un metodo euristico per la verifica dell'isomorfismo di due grafi semplici non orientati. Calcolo, 8(1):1–31, 1971.

[4] Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. In Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI'17, pages 712–719. AAAI Press, 2017.

[5] Deep Blue, IBM. `https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/`.

[6] AlphaGo, DeepMind. `https://deepmind.com/research/alphago/`.

[7] David Silver (2017). *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*.

[8] Stuart J. Russell, Peter Norvig (2009). Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall.

[9] Jonathan Rubin, Ian Watson (April 2011). *Computer poker: A review*.

[10] AI Game Dev. Retrieved 25 February 2017. *Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI*.

[11] Tim Wheeler (2017). *AlphaGo Zero - How and Why it Works*.

[12] UCT for Games and Beyond. `http://mcts.ai/about/`

[13] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton. A Survey of Monte Carlo Tree Search Methods. IEEE Trans. Comp. Intell. AI Games, 4(1):1–43, 2012.

[14] R. Coquelin, Pierre-Arnaud and Munos, "Bandit Algorithms for Tree Search," in Proc. Conf. Uncert. Artif. Intell. Vancouver, Canada: AUAI Press, 2007, pp. 67–74

[15] D. Silver, "Reinforcement Learning and Simulation-Based Search in Computer Go," Ph.D. dissertation, Univ. Alberta, Edmonton, 2009.

[16] D. Silver, R. S. Sutton, and M. M¨ uller, "Sample-Based Learning and Search with Permanent and Transient Memories," in Proc. 25th Annu. Int. Conf. Mach. Learn., Helsinki, Finland, 2008, pp. 968–975.

[17] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J.-B. Chaslot, and J. W. H. M. Uiterwijk, "Single-Player Monte-Carlo Tree Search," in Proc. Comput. and Games, LNCS 5131, Beijing, China, 2008, pp. 1–12.

[18] N. R. Sturtevant, "An Analysis of UCT in Multi-Player Games," in Proc. Comput. and Games, LNCS 5131, Beijing, China, 2008, pp. 37–49.

[19] "Multi-player Go," in Proc. Comput. and Games, LNCS 5131, Beijing, China, 2008, pp. 50–59.

[20] L. S. Marcolino and H. Matsubara, "Multi-Agent Monte Carlo Go," in Proc. Int. Conf. Auton. Agents Multi. Sys., Taipei, Taiwan, 2011, pp. 21–28.