Master research Internship

Final report

Computing Genome Edit Distance

Bioinformatics - Performance

Author: Ian Jeantet

Supervisor: Dr Patrik Haslum
Decision Sciences Program
CSIRO Data61
Abstract: We consider the problem of calculating the edit distance between two genomes based on the same set of genes in terms of inversions, transpositions and inverted transpositions. As the problem can be formulated as a planning problem, there are various existing methods to solve it. However none of them were able to scale up to solve problems as large as we would like. In this document we will consider the problem as a state search problem. Hence, we will see various modified A* algorithms and pruning techniques coming from the optimal planning that could be applicable to enhance the existing optimal algorithm and could be able to increase the possible size of the genomes. In fact the idea is to attack the problem on two angles, the amount of memory and the time taken to stored all the generated nodes and the time taken by the heuristic computation by reducing the branching factor.

Contents

1 Introduction 1

2 Problem description 2
  2.1 Formulation as a planning problem ........................................... 2
  2.2 Various approaches to solve a planning problem ............................ 3
    2.2.1 As a Model Checking problem ........................................... 3
    2.2.2 As a Boolean Satisfiability problem .................................... 3
    2.2.3 As a Constraint Satisfaction Problem ................................... 3
    2.2.4 As a State Search Problem ............................................... 4
  2.3 Notations .............................................................................. 7

3 State of the art 7
  3.1 Modified A* algorithms .......................................................... 7
    3.1.1 Partial Expansion A* ......................................................... 7
    3.1.2 Partial Move A* ............................................................... 9
    3.1.3 Enhanced Partial Expansion A* .......................................... 10
    3.1.4 PrefPEA* ......................................................................... 10
  3.2 Pruning techniques ................................................................. 11
    3.2.1 Partial Order Reduction ...................................................... 12
    3.2.2 Symmetry Elimination ....................................................... 14

4 Experimentations 15
  4.1 Modified A* algorithms .......................................................... 16
    4.1.1 A* with buckets ............................................................... 17
    4.1.2 Partial Expansion A* ......................................................... 18
    4.1.3 PrefPEA* ......................................................................... 19
  4.2 Pruning techniques ................................................................. 23
    4.2.1 Symmetry Elimination ....................................................... 24
    4.2.2 Pruning Duplicate Nodes ................................................... 25

5 Conclusion 29
1 Introduction

The evolution of genomes through time was intensively studied to determine common ancestor or differences/similarities between a set of genomes. Considering a set of available operations $O$ and a certain cost for each of these operations, the genome edit distance (GED) problem or genome rearrangement problem consists in finding the smallest cost resulting of the sum of the cost of each operation from $O$ needed to obtain a genome $G'$ starting from a genome $G$.

Here we restrict ourselves to the generalized Nadeau-Taylor (NT) model (Nadeau and Taylor [1984]; Wang and Warnow [2001]). Two genomes $G$ and $G'$ match the model if they are a type of permutation of the same set $\{g_1, \ldots, g_n\}$ of genes. Hence they have the same size $n$. We consider $G$ and $G'$ as circular so every gene has exactly two neighbours. Each gene is also oriented in one of two directions which is indicated by a $\pm$ sign. Hence there are $2^n n!$ signed permutations for a set of $n$ integers and so $2^n(n-1)!$ possible states for this problem as there are $n$ ways to represent the same state (same sequence minus a rotation of the genes). The set of applicable operations on genomes that modify the order and orientation of the genes is composed of the three following operations:

- **Inversion (I):** The order of a stretch of genes is reversed. It negates the sign of each gene in the stretch. For example, $G' = (-g_{k-1}, \ldots, -g_1, g_k, \ldots, g_n)$ where $1 < k \leq n$ is an inversion of $G = (g_1, \ldots, g_n)$.

- **Transposition (T):** A stretch of genes is moved in the genome sequence. For example, $G' = (g_k, \ldots, g_l, g_1, \ldots, g_{k-1}, g_{l+1}, \ldots, g_n)$ where $1 \leq k, l \leq n$ is a transposition of $G$.

- **Inverted transposition (IT):** A composition of the two previous operations. For example, $G' = (-g_l, \ldots, -g_k, g_1, \ldots, g_{k-1}, g_{l+1}, \ldots, g_n)$ where $1 \leq k, l \leq n$ is an inverted transposition of $G$.

Some other models were used in the past to solve this problem. Two of them are the inversion only (IO) model (Kaplan et al. [2000]) and the transposition only model (Bafna and Pevzner [1998]). Based on simplified versions of the generalized NT model they are considered as non-optimal approaches and restrict their available operations respectively to only inversion and to only transposition. Several results with these different models were proved. If we consider the IO model, Caprara [1997] proved that sorting by reversals is NP-Hard on unsigned permutations and Bader et al. [2001] found a linear-time algorithm on signed permutations. If we consider only transpositions, Christie [1996] found a polynomial algorithm to solve the problem but using the mathematical meaning of transposition (ie not necessary consecutive blocks). By using only the transpositions allowed in the generalized NT model (ie of consecutive blocks), Bulteau et al. [2012] proved that the GED problem is NP-Hard. Thus it lets us think that the GED problem under the generalized NT model is also NP-Hard.

Also, operations of inversion and transposition often appear having the same cost in genome distance analysis. This is not necessarily true biologically, as transposition and inversion mutations do not naturally occur with the same frequency. Eriksen et al. [2001] found that the optimal weights introducing least bias are 1 for inversions and 2 for transpositions and inverted transpositions. Using this Eriksen [2002] gave a polynomial time algorithm able to compute an $(1+\varepsilon)$-approximation of the minimal weighted distance with $\varepsilon > 0$. 

1
The GED problem can be formulated as a planning problem which can be solved by several existing methods. The problem formulation and these methods will be developed in the first part of this document and we will focus on the state search method that corresponds to the scope of the subject.

The next step in this research project is to examine more advanced optimal search enhancements, such as symmetry reduction, partial expansion, branching strategies and others. The state of the art of these different techniques is presented in this document. These improvements are grouped in two parts. The first one is devoted to the explanations of various modified A* algorithms and the second one is focused on the presentation of different pruning techniques that exist in planning problem and that need to be adapted to the state search approach.

2 Problem description

2.1 Formulation as a planning problem

The GED problem was formulated for the first time as a planning problem by Erdem and Tillier [2005]. One of the genome is represented as the initial state and the second one as the goal state. The planner has to find a sequence of at most $k$ operations that leads the initial state to the goal state. They extend their planning formulation to a model with duplicate genes, involving inversions, transpositions, inverted transpositions, insertions and deletions (Uras and Erdem [2010]) but this goes beyond the scope of the subject. Also Haslum [2011] phrased the genome edit distance problem as a domain-independent planning problem.

There are two main formalisms for planning problems, STRIPS formalism (Fikes and Nilsson [1971]) and SAS$^+$ formalism (Bäckström and Nebel [1993]). STRIPS formalism is based on propositional logic when SAS$^+$ models states by multi-valued state variables. Note that there is a natural correspondence between STRIPS and SAS$^+$ formalisms (Helmert [2006]).

**STRIPS formalism:** In the STRIPS (Stanford Research Institute Problem Solver) formalism, a planning task is a tuple $\Pi = < P, O, I, G, C >$ where:

- $P$ is a set of propositions,
- $O$ is a finite set of operators. Each operator $o \in O$ has an associated set of preconditions $pre(o) \subseteq P$ and two sets of effects $add(o) \subseteq P$ (conditions made true by the action) and $del(o) \subseteq P$ (conditions made false by the action),
- $I$ is the initial state,
- $G \subseteq P$ is a set of propositions where a state $s$ is a goal state if $G \subseteq s$,
- $C : O \rightarrow \mathbb{R}^+$ is a non-negative cost function defined for each operators.

An operator $o$ is considered applicable to a state $s$ if $pre(o) \subseteq s$. If $o$ is applicable, the resulting state $s[o] = (s\setminus del(o)) \cup add(o)$. The transition graph over the state space contains all edge $< s, s[o]; o >$ from $s$ to $s[o]$ labelled with $o$ for each state $s$ and each operator $o$ applicable to $s$. A sequence of operators $\pi = < o_1, ..., o_k >$ is applicable to $s$ if it exists a path labelled by $\pi$ and starting from $s$ into the transition graph. If a such path exists, the end state is noted $s[\pi]$. If $s[\pi]$ is a goal state, $\pi$ is a plan for $s$. It costs is the cumulative cost of each operator in the sequence: $C(\pi) = \sum_{i=1}^{k} C(o_i)$. 


A plan for $s$ is called optimal if it has the minimal cost. The objective of optimal planning is to find an optimal plan for $I$.

**SAS$^+$ formalism:** In the SAS$^+$ formalism, a planning problem is defined as a tuple $\Pi = < V, O, S_I, S_G >$ where:

- $V = \{v_1, ..., v_n\}$ is a finite set of state variables. Each have a finite domain $D_v$. A fact is a couple $< v, d >$ where $v \in V$ and $d \in D_v$. A partial variable assignment is a set of facts, each with a different variable. A state is a partial variable assignment defined for each variables of the set $V$,
- $O$ is a set of operators specified with $(pre(o), eff(o))$ where both are partial variable assignments. It is the same definition than for the STRIPS formalism but all the effects of $o$ are regrouped in $eff(o)$,
- $S_I$ is the initial state,
- $S_G$ is the goal.

As these two formalisms are very close, it is possible to define a state transition graph in the same way as we did for the STRIPS formalism.

The following section will briefly summarize the existing methods available to solve a planning problem. And we will focus on the approach which considers the planning problem as a state search problem as it corresponds to the scope of the subject. Also we will explain in details the work done by Wong [2012] as it is the starting point for this work.

### 2.2 Various approaches to solve a planning problem

#### 2.2.1 As a Model Checking problem

One existing method is based on *model checking* and was introduced by Cimatti et al. [1997]. The idea is to transform the planning problem into a model checking problem and to use existing model checkers to solve it.

#### 2.2.2 As a Boolean Satisfiability problem

A similar method consists in transform the planning problem into a sequence of *boolean satisfiability problem* (SAT) (Kautz and Selman [2006]). A SAT solver will be able to decide if the planning problem can be solved in $N$ steps or not. By increasing the number of steps we will be able to determine the minimal value for $N$ to solve the problem. However SAT problems are NP-complete so it is hard to solve large instances.

#### 2.2.3 As a Constraint Satisfaction Problem

In his thesis Nelson [2013] tried to approach the genome edit distance problem as a *constraint satisfaction problem* (CSP) and to present the advantages and disadvantages of a such approach.
He found that constraint satisfaction is able to solve the problem with the advantage that using a CSP for modelling a problem allows to employ more general solving techniques, regardless of the problem domain. After having implemented two different constraint satisfaction models, he also worked on possible optimisations such as symmetry reducing constraints and genome compression algorithms. Finally he worked on alternative models using inversion-only distance and breakpoint distance. The CSP approach scaled slightly better, but still we are only able to find optimal solutions to problems of about half the size we need.

2.2.4 As a State Search Problem

Another approach consists to use state space search methods to solve planning problems. It corresponds to the approach that we want to improve and that was explored by Wong [2012]. It consists in finding a path (the sequence of operations of the planning problem) that links the initial state to a goal state.

This approach is based on a search algorithm and at least one heuristic function. Several search algorithms and heuristics exists but we will focus on those chosen by Wong. He used a standard A* search, with a state-of-the-art abstraction-based admissible heuristic. This, however, does not scale up to solve problems as large as we would like.

A* algorithm: A* is a best-first search algorithm, meaning that it solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance traveled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes \( f(n) = g(n) + h(n) \) where \( n \) is the last node on the path, \( g(n) \) is the cost of the path from the start node to \( n \), and \( h(n) \) is a heuristic that estimates the cost of the cheapest path from \( n \) to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open list. At each step of the algorithm, the node with the lowest \( f(x) \) value is removed from the queue, the \( f \)- and \( g \)-values of its neighbours are updated accordingly, and these neighbours are added to the queue. The algorithm continues until a goal node has a lower \( f \)-value than any node in the queue (or until the queue is empty). The \( f \)-value of the goal is then the length of the shortest path, since \( h \) at the goal is zero in an admissible heuristic.

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node’s predecessor is the start node.

The Figure 1 describes the A* algorithm. The initialisation phase consists mainly in putting the starting node in the open list. Then we have a loop over the open list that keeps going until the list is empty (No solution exists) or it finds the goal (Find path). The generation of the neighbours
uses a set of actions which is for the basic algorithm the set of all enabled actions on the current node. During the generations of all the neighbours it also calculates the heuristic value of each new node. This heuristic value is used in the next block to process the neighbours (add them to the open list by keeping the list sorted).

For the GED problem, the set of enabled actions is always the same as there is no preconditions to apply an action on a sequence. Hence we can see that it is possible to act on the set of actions to reduce the branching factor and on the heuristic to distinguish more precisely the different nodes. However we have to keep in mind that there often is a trade to do between the time taken to calculate the heuristic and the accuracy of this heuristic.

As the problem is very large and shallow, Wong needed to slightly modify the behaviour of his algorithm and used different techniques to rearrange the order of the nodes in the Open list to avoid to expand too many nodes.
Wong based his heuristic on a pattern database (PDB) heuristic (see the end of this section for the definition). The PDB is precomputed and stored in memory. To compute it he did a naive breadth first search from the goal sequence (which corresponds to the ordered sequence as we can always rename the genes to search the shortest path to this specific sequence) until no new node or shorter path are generated. For each generated sequence, the algorithm looks if it is a new one or not. If it is, it stores the cost of the actual path as smallest cost as there is no other path discovered so far. If it is not a new sequence, the algorithm update the cost if the cost of the actual path is smaller than the stored cost. In this case, the new node is added to the open list to be expanded or re-expanded with the new cost. Otherwise if the cost of the actual path is bigger than the one stored, we discard the node as it can’t lead to shorter paths than the previous node with the same sequence. The breadth first search stops when there is no node in the open list. However the size of the PDB grows exponentially with the number of genes and it is not possible the store in memory a PDB of a problem size greater than 9. The file that contains all the solution for a problem of size 9 contains more than 20 millions entries and weight 206MB. In fact Wong stored the rank of every permutation and their costs in the PDB but as the rank corresponds to the line where the entry is stored, we can store only the cost and reduce this file to 40MB. If the storage in memory is reduce, the number of generated sequences is the same so the limit size of the PDB is still 9.

To use this PDB on larger sizes of genome we need to extract a sub-problem of size 9 (9 genes) from the larger problem. To do so Wong used a sampling technique, the basic idea consists in taking randomly the number of genes corresponding to the size of the PDB among all the genes from the sequence and looking in the PDB for the exact cost to rearrange those genes. We can see that it exists \( \binom{n}{k} \) possibilities to extract \( k \) random genes in a sequence of \( n \) genes and if we take genes that are already ordered the heuristic value won’t be accurate. Hence Wong decided to repeat this random sampling a certain number of time with the hope to have a sample that contains the less possible number of following genes. By taking the maximal cost between all these samples we still have an admissible heuristic as the heuristic value for each sample is admissible.

Also it can be interesting to find a way to select the genes that are at the wrong place in the sequence as we have to move then at least once to reach the goal. The heuristic value given by the PDB will be more accurate. To do so Wong used different measures that can also be used as admissible heuristics themselves such as the breakpoint heuristic (see the end of this section for the definition) or the sign heuristic (see the end of this section for the definition). By taking the genes from each side of a breakpoint or a sign break we are sure to have at least one gene at the wrong place or in the wrong direction.

Wong compared all these heuristics and on a first sight, more samples equals a better heuristic estimation what implies less node expansions but he found a limit around 15 samples where after the gain in node expansions is almost zero. In addition the overall computational time increases after 15 samples. Indeed there is little or no gain in node expansions but there are more computations for the extra samples. In other scenarios the signed breakpoint heuristic shows better results than the other heuristics if the number of samples is small.

**PDB heuristic:** A Pattern Database stores a collection of solutions to sub-problems that must be achieved to solve the problem. Hence to compute an heuristic value of any state of the problem, we just have to extract a pattern that matches with a sub-problem in the PDB. The exact cost to
solve this sub-problem is a lower bound of the cost of the state and can be used as an admissible heuristic.

**Breakpoint heuristic:** The breakpoint heuristic (Blanchette et al. [1999]; Blin et al. [2004]) is based on the breakpoint distance. Given two genomes $G$ and $G'$ over the same set of signed genes and a pair of following genes $g_1g_2$ in $G$, we call breakpoint the fact that neither $g_1$ precedes $g_2$ nor $-g_2$ precedes $-g_1$ in $G'$. If $G$ contains $n$ genes ($|G| = n$), $g_n$ precedes $g_1$ as we consider only circular genomes. The breakpoint distance $D_B(G, G')$ is the number of breakpoints between $G$ and $G'$. One of the generalized NT model operations can fix at most 3 breakpoints (the transposition can) so we can use $h_B(G) = \left\lceil \frac{D_B(G, G_{goal})}{3} \right\rceil$ as an admissible heuristic.

**Sign heuristic:** The sign heuristic is based on the number of consecutive gene segments that have a different sign between $G$ and $G'$. One of the the generalized NT model operations can only fix at most one of these segments so we can use this number as an admissible heuristic.

### 2.3 Notations

**Action** As an action is specified by the kind of action $A \in \{I, T, IT\}$ and 3 parameters that are a starting index $s$, a length $l$ of the modified section and an offset $o$ to determine the new location of the modified section, we will use the notation $A(s, l, o)$ for the rest of the document. For example $T(3, 1, 2)$ or $I(0, 2)$ (no need to precise the offset as it is always equals to 0 for an inversion) or $IT_6(1, 1, 4)$ where 6 indicates the size of the genome on which the action can be apply.

### 3 State of the art

#### 3.1 Modified A* algorithms

The main issue with the A* algorithm is that, considering the large branching factor of the GED problem, it generates many nodes that will be never expanded. It results in a significant costs in memory and computation times. As showed in the Figure 2, in this part we will focus on various modified A* algorithms trying to act on the generation and/or the processing of the neighbours.

##### 3.1.1 Partial Expansion A*

Yoshizumi et al. [2000] applied a modified A* algorithm to the multiple sequence alignment problem which is close to the genome edit distance problem due to its large branching factor.

**Principle:** This Partial Expansion A* algorithm (PEA*) aims to reduce the amount of memory required during the A* execution by putting only the promising nodes in the Open list. It acts on the neighbours processing.

Every node contains the usual f-value and contains also an F-value which is initialized to the f-value. This F-value is used to determines if a child node is promising or not. When a node is expanded, PEA* generates all of its children but puts into the Open list only those that have an f-value less than or equals to the F-value of the parent node plus a predefined and non-negative
Figure 2: The modified A* algorithms act on the node generation and processing cut-off value C. The remaining children are discarded and the expanded node is added back to the Open list with an F-value equals to the lowest f-value among all unpromising child nodes.

A* expands nodes in incremental order of f-value while PEA* expands nodes in incremental order of F-value.

Results: If $C = \infty$, this algorithm is identical to A* and if $C = 0$, only the best nodes with the optimal cost are stored. In this case, the search is performed with the same size of memory as the Closed list used by A* but it increases also the number of cumulative expanded nodes due to the possibility of re-expansion of a same node. And this increase of expanded nodes has a direct negative effect on the computational time.

In application to the multiple sequence alignment problem with $C = 0$, they achieved to align seven sequences (the best that A* can do in their configuration) with an average of only 4.7% of
the amount of memory required in A* but with more than 5 times of cumulative expanded nodes than A*.

3.1.2 Partial Move A*

Cazenave [2010] introduced another variant of A*, the Partial Move A* (PMA*) that aims to reduce not only the amount of memory required for a large branching problem but also the computational time. Instead to use the set of actions to apply on the current sequence, this algorithm acts on the neighbours generation by trying to avoid to generate all the child nodes.

**Principle:** To cut the computational time (i.e not generate all child nodes when a node is expanded) this algorithm splits a move that corresponds to the generation of a child node into incomplete moves and creates in consequence partial nodes. In multi-agent path finding for example, a partial move is moving one agent to one of its neighbour locations and a complete move is moving all the agents to one of their neighbour locations. In the genome edit distance problem, a partial move could consist in choosing or not if the action will modify the gene at a certain location of the sequence whereas a complete move could consist in choosing an action on the whole genome sequence. There are two stop conditions and one general case in PMA*:

- During an iteration, the f-value for the current partial node is computed and if this value is greater than the f-value of the current iteration, the path has greater cost and the algorithm stops.
- If not and if the current state completes a move which corresponds to a node then it stops if the goal is reached or it saved the node and starts a new move.
- In the general case, it calls recursively this algorithm for all the possible choices of next partial move.

In fact this algorithm uses a kind of dynamic programming to avoid to compute entirely all the child nodes. If the partial node is unpromising, all the children that can be created from it are not computed.

**Example:** Let be \( g = (1, 2, 3, 4, 5) \) a genome sequence that we need to expand. We will browse all the indices from 0 to 4 and construct partial nodes from the decisions that we will take for each index. The first one is to decide if an action that impact the first gene will is promising or not. Depending on the previous decisions, we will be able to take the same decision for the following indices or not. For instance we determined that an action on the first 2 indices lead to the goal but not an action on the 3 first. So there is no choice for the other indices, the action can’t act on them as it has to impact only following genes. This way we build only actions that are promising and prune the others. The difficulty here is to decide if a partial node is promising or not.

**Results:** Cazenave obtained better results than the regular A* regarding the number of generated nodes in two well-known problems, the multiple sequence alignment problem and the multi-agent path finding problem. Unfortunately he did not compare PEA* and PMA* in particular on the computational time side.
3.1.3 Enhanced Partial Expansion A*

Another recent modified A* algorithm called Enhanced Partial Expansion A* (EPEA*) was developed to avoid to generate all the surplus nodes. The surplus nodes are all the nodes that not contribute to find the optimal solution. As seen previously they are discarded but they still are generated and it costs on the computational time side. EPEA* was first introduced by Felner et al. [2012] and re-explained with more details by Goldenberg et al. [2014]. It seems to be applicable to IDA* with pattern databases that partially corresponds to the previous work done by Wong [2012] on the genome edit distance problem. So it could be useful to apply this modified algorithm to the GED problem.

Principle: The pseudo-code is similar to that of PEA* and the difference is in the use of a domain and heuristic specific Operator Selection Function (OSF). While PEA* generates all the child nodes and discarding all the unpromising, EPEA* uses the OSF to generate only those who are needed and to calculate the lowest F-value of these generated child nodes. Each node is generated at most once during the search process and no child is regenerated when its parent is re-expanded. To improve the IDA* we can augment it with an OSF and work with a EPE-IDA*. To do this we will have to find the right OSF.

There are two types of OSFs:

- Direct Function ($OSF_D$) which returns exactly the set of operators desired.
- Ordered by Operators ($OSF_D$) which iterates through all the applicable operators to the current state and decides if it will produce or not the desired f-value.

In the case where a direct-computation OSF is not available for a given domain and heuristic, we can still construct an $OSF_D$ for each node when it is expanded for the first time and use it for every re-expansion. On the other hand storing this function for every node in the Open list may cost a large amount of memory and have the opposite effect of what we are looking for with this novel algorithm.

Application: In their works Felner et al. [2012]; Goldenberg et al. [2014] built several OSFs for different well-known problems. They notably shown better results for the Rubik’s Cube problem with EPEIDA* including a PDBs-based OSF.

Finally they show that EPEA* is most effective if:

- The domain possesses a large branching factor.
- There are only few distinct possible f-values for the children of a given node.
- An $OSF_D$ is available what implies to be able to classify the operators to avoid to check all of them when a node is expanded.

3.1.4 PrefPEA*

In the same idea than EPEA* of not generating and evaluating all the successor nodes when a given node is expanded, Ivankovic et al. [2014] stage the node expansion by the preferableness of
successors. Instead of using an OSF, to determine the most promising child nodes, they use the information given by the heuristic when the parent node is generated. When the heuristic value of a node is computed, \textsc{PrefPEA*} stores with the node a set of preferred actions and when this node will be expanded, only preferred successors will be generated. This algorithm also prioritises expansion of node that have still unexplored preferred successors.

They showed that \textsc{PrefPEA*} is more effective if a node has many successors (as it is the case for the GED problem) and only few preferred ones. In the contrary they noticed that in a small number of cases, less than 2%, \textsc{PrefPEA*} may expand more nodes than \textsc{A*}.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Nodes generated</th>
<th>Nodes put into OPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textsc{A*}</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td>\textsc{PEA*}</td>
<td>All</td>
<td>Needed</td>
</tr>
<tr>
<td>\textsc{PMA*}</td>
<td>Partially</td>
<td>Needed</td>
</tr>
<tr>
<td>\textsc{EPEA*}</td>
<td>Needed</td>
<td>Needed</td>
</tr>
<tr>
<td>\textsc{PrefPEA*}</td>
<td>Preferred</td>
<td>Preferred</td>
</tr>
</tbody>
</table>

Table 1: Comparison of different modified A* algorithms

To conclude on this part, we saw four modified A* algorithms that aim to reduce the amount of memory required and the computational time by acting on the number of nodes that are generated and stored. The Table 1 summarizes the theoretical results for each of them. It seems to be possible to adapt them to the genome edit distance problem but we need to deeply explore pruning techniques to make them more efficient.

As Wong [2012] explained, there are several available admissible heuristics for our problem and depending on the number of samples taken, no particular heuristic clearly differs from the others. Another idea would be to consider all of those heuristics at the same time and to use meta reasoning techniques to speed up the time taken by the heuristic computations. Tolpin et al. [2013, 2014] suggested a Rational Lazy A* (resp. Rational Lazy IDA*) algorithm to easily reduce this time in \textsc{A*} (resp. IDA*). They found particularly good results if one heuristic is dominant but more costly than the others.

### 3.2 Pruning techniques

We saw on the first part some modified A* algorithms including pruning techniques but they may be not good enough. If a pure heuristic search based on A* algorithm is considered as one of the best approach to optimally solve classical planning problems, it is limited in the GED problem considering that heuristics lead to exponentially large state space. Hence, additional pruning techniques need to be used to efficiently solve the problem when the length of the gene sequence grows. This is why in this part we focus on recent pruning techniques for optimal planning such as partial order reduction or symmetry elimination. These pruning techniques will have to be adapted to the state space search method and they will act on the set of actions, as shown in Figure 3, in trying to reduce it but in keeping the optimality of the solution.
3.2.1 Partial Order Reduction

The partial order reduction (POR) is a technique for reducing the size of the state space by exploiting the commutativity of concurrently executed operations which result in the same state when executed in different orders. When two executions of actions sequences are sufficiently similar to each other then it is not necessary to investigate both of the executions. It has been extensively studied in model checking and has proven to be an enabling technique for reducing the search space and costs. Regardless of the method used for the POR, it needs to preserve completeness and optimality.

There are various POR methods such as ample set method, stubborn set method or sleep set method.

**Ample sets:** The ample set method relies on the notion of independence between transitions. The goal is to determine for each generated state the subset of successor states that need to be
explored. Using an ample set should lead to a significant smaller state graph but needs to include enough enabled operators to be sure to have correct results. Also the computation of an ample set should be doable in acceptable time.

**Independence:** Let be transitions $t_1$ and $t_2$ independent of each other such that $s \xrightarrow{t_1} s_1$ and $s \xrightarrow{t_2} s_2$ then it exists $s'$ such that $s_1 \xrightarrow{t_2} s'$ and $s_2 \xrightarrow{t_1} s'$. In other words, the execution of both results in a unique state regardless the order in which they are executed. Transitions that are not independent are dependent.

**Invisibility:** In using the SAS$^+$ formalism, a transition from a state $s$ is invisible if it has no effect on the partial variable assignment of each variables of $s$. As we define a state to its variable assignments, a transition $t$ is invisible if $s \xrightarrow{t} s'$.

The construction of an ample set revolves around 4 properties (Clarke et al. [1999]):

- **C0:** If a state has at least one successor in the full state space then it has at least one successor in the reduced state space ($\forall s \in S, ample(s) = \emptyset \iff enabled(s) = \emptyset$).
- **C1:** for all states $s$, a transition $t'$ that is dependent on a transition $t \in ample(s)$ cannot be executed without a transition from $ample(s)$ occurring first. Checking C1 is at least as hard as checking reachability for the full state space so it needs to use an approximating heuristic.
- **C2:** If $s$ is not fully expanded (i.e $enabled(s) \neq ample(s)$) then every transition $t \in ample(s)$ is invisible.
- **C3:** A cycle is not allowed if it contains a state in which some transition $t$ is enabled but never included in $ample(s)$ for any state $s$ on the cycle. A sufficient condition for C3 is that at least one state $s$ along each cycle in the reduced state graph is fully expanded (in this case $enabled(s) = ample(s)$).

**Persistent sets:** A set $T$ of transitions enabled in a state $s$ is persistent iff, for all non-empty sequences of transitions $(t_1, \ldots, t_n)$ from $s_1 = s$ such that $\forall i, t_i \notin T$, $t_n$ is independent in $s_n$ with all transitions in $T$.

**Stubborn sets:** The stubborn set method (Valmari [1989]; Valmari and Hansen [2016]) is defined through commutativity over sequences of actions. A set of operations $T(s)$ is weakly stubborn at $s$ if:

- **D0:** $\forall a \in T(s)$ and $\forall b_1, \ldots, b_n \notin T(s)$, if the execution of the sequence $(b_1, \ldots, b_n, a)$ is possible from $s$ and leads to $s'$, then the execution of the sequence $(a, b_1, \ldots, b_n)$ is possible and leads also to $s'$.
- **D1:** Either $s$ is a deadlock or $\exists a \in T(s)$ such that $\forall b_1, \ldots, b_n \notin T(s)$ the execution of $(b_1, \ldots, b_n, a)$ is possible from $s$.

Transitions in a stubborn set $T_s$ in a state $s$ can be either enabled or disabled in $s$. If we take only the enabled transitions from $T_s$ then the obtained set is a persistent set in $s$ (Godefroid et al. [1996]).
Wehrle and Helmert [2014] performed a comparison of different strategies for computing stubborn sets, using for instance envelope strategies.

**Sleep sets:** A sleep set is a set of transitions. The sleep set associated to a state $s$ is a set of transitions that are enabled in $s$ but will be not executed.

![Figure 4: Reducing the state space by using sleep sets](image)

As an example, we can see in Figure 4 that the sleep set $S_{s_2}$ for the state $s_2$ contains $t_1$ to avoid the exploration of $t_1$ in $s_2$ that leads to the same state than the transition $t_2$ form $s_1$.

### 3.2.2 Symmetry Elimination

In contrast to POR, symmetry elimination considers equivalence classes of symmetrical states and uses representative states of each equivalence class. In this case, A* with symmetry elimination prunes some of the resulting successor states and keeps only one per class. The optimal plan is preserved.

Let $\Pi = <P, O, I, G, C>$ be a STRIPS planning task as introduced in the Problem formulation section. A permutation $\sigma$ of $\Pi$ is a structural symmetry if:

- $\sigma(P) = P$
- $\sigma(O) = O$ and $\forall o \in O$:
  - $\text{pre}(\sigma(o)) = \sigma(\text{pre}(o))$
  - $\text{add}(\sigma(o)) = \sigma(\text{add}(o))$
  - $\text{del}(\sigma(o)) = \sigma(\text{del}(o))$
  - $C(\sigma(o)) = C(o)$
- $\sigma(G) = G$

This notion of structural symmetry allows directly reasoning on a compact representation of the state space called *orbit space*. The orbit of vertices can be computed in polynomial time and using graph theory, Pochter et al. [2011] presented a set of algorithms applicable to state based planners without serious harm to the performances. First we need some definitions:
Permutation group: $\Sigma = \{\sigma_1, ..., \sigma_n\}$ is said to generate a finite group $G$ if $G$ is exactly all permutations obtained by repeatedly composing elements of $\Sigma$.

Orbit space: The orbit of a vertex $s$ with respect to some subgroup $G$, denoted $G(s)$, is simply the set of vertices to which elements in $G$ map $s$. $G(s) = \{\sigma(s) | \sigma \in G\}$.

Point-stabilizer: The point-stabilizer of a vertex $s$ with respect to $G$, denoted $G_s$, is a subgroup of $G$ that contains all the permutations that fix $s$. $G_s = \{\sigma \in G | \sigma(s) = s\}$.

The goal of a such algorithm is to find a shortest path from $S_I$ to any of the nodes in $G_{S_I}(S_G)$. Once a such path is found to a node $\sigma(S_G)$, the real path from $S_I$ to $S_G$ is simply calculated by applying $\sigma^{-1}$ on the entire path. Indeed $\sigma \in G_{S_I}$ then $\sigma^{-1}(S_I) = S_I$. However, even if Shleyfman et al. [2015] proved that many heuristics are invariant under structural symmetry, git is not guaranteed that $h(\sigma(S_G)) = 0$ so the idea is to work with the symmetry group $G_{S_I,S_G}$ that stabilizes both start and goal state. They based their algorithms on the A* algorithm. The steps they added to A* are the following:

1. Before the search, find generator of $G_{S_I,S_G}$.

2. Whenever generating a state $s'$, search for a previously generated state $s$ such that $s' \in G_{S_I,S_G}(s)$. If $s$ exists, treat $s'$ exactly as if it was $s$ (as $s'$ is in the orbit of $s$), otherwise $s'$ represents a newly found orbit.

3. Stop the search when a goal state is expanded.

Also Wehrle et al. [2015] worked on a coupling approach of POR and symmetry elimination. They restricted the orbit space search to states generated by strong stubborn sets and it inherits the strengths of both approaches.

4 Experimentations

This part shows the modifications done on the A* algorithm to improve its efficiency regarding the state of the art presented in the previous part. Also we will explain why certain of these improvements were not developed because not applicable to the GED problem.

Heuristic determination: According to the work done by Wong [2012] the search algorithm used so far is a classical A* algorithm with a sorted open list that use a sampling technique to compute the heuristic value of a node. We will compare

Test conditions: Each point is an average for the heuristic computation on 500 random states. The PDB used contained sub-problems of size 7. Even if we have a PDB of size 9 we want also to compare the heuristic values for smaller sizes of sequences. The breakpoints and sign heuristics read the sequence exactly one time. The PDB heuristic with random sampling takes the biggest heuristic value among the values from 15 samples. Also we can take the subset of indices from the breakpoints or sign heuristics to extract a sample to determine the PDB heuristic. The idea is to take the indices of genes that we know they are not at the right place or in the right
orientation to maximise the PDB heuristic value. If the subset of indices is large enough we sample 15 times and we take the maximal value exactly like for the regular PDB heuristic. If the subset is too small (contains less index than the size of the PDB), we add arbitrary the first untaken indices to complete the subset to the wanted size. Hence in this case we take only one value in the database.

Then after further tests to compare the different heuristics, we can see on Figure 5 that the PDB heuristic with random sampling and based on breakpoints indices are greater than the others for small sizes of genome (better until 11 genes). For larger sizes of genomes, the breakpoint heuristic gives better results as we are limited with the PDB heuristic by the maximal value in the database. Also on the Figure 6 we can see that obviously the time taken for each variant of the PDB heuristic is greater than the time taken to compute the breakpoint end sign heuristic (15 samplings against one reading of the sequence).

However considering the tests conducted by Wong, we will use for all the different following algorithms and all further tests the PDB heuristic with a random sampling technique whose the number of samples is fixed to 15.

Figure 5: Comparison of the value of different heuristics

The machine where all the tests were conducted is an Asus Zenbook UX510UW-CN082T equipped with a processor Intel Core i5-7200U (Kaby Lake) cadenced at 2,5 GHz and with 8 Go of RAM.

4.1 Modified A* algorithms
As explained in the state of the art, the followings modified A* algorithms will act on the node generation and node processing.
4.1.1 A* with buckets

On large sizes of genome, the size of the open list grows very quickly and for each successor we check if it has already been generated with means that we have to browse the whole open list in the worse case (new node to add). Hence a first idea was to split this open list in buckets. Each bucket contains all the nodes for a specific f-value and is sorted accordingly to the h-value of the nodes. To take the first element of the list, we just have to take the first element of the no-empty bucket that contains the nodes with the smallest f-value. Also we don’t check if the node was already generated for the A* with buckets before we insert it in the open list. It’s only when we take the first element of this list that we check if it was already expanded. Otherwise the search works exactly the same way as for the classical A* algorithm.

Test conditions:

- Each line in an average on 500 random problems except for the last line which is an average on only 20 sequences as both algorithms sometimes fail to find the solution in less than 100s.
- The PDB size is 9 as we want to compare both algorithms on the largest possible problems when the size of the open list increases.
- The total time is split in 3 categories, the time taken to generate the nodes, to compute their heuristic values and to store them in the open list.

The Table 2 contains the average time taken for the classical A* algorithm and the A* with buckets. Here we have 11/20 failures for the classical A* and only 4/20 for the A* with buckets. The algorithm “fails” when it find the solution in more than 100s, in this case we stop the search and record the current statistics. It corresponds to a lower bound of the real values so the average is also lower than the real average value.
Table 2: Comparison between the classical A* and the A* with buckets

If we look into this Table we can see for the classical A* algorithm that computing the heuristic values takes the biggest part of the total time even if on large sizes of genome the storage takes longer than the computation of these h-values. As we said it happens when the size of the open list becomes too important and using buckets is a good way to reduce significantly the time taken for the storage.

Other tests with different PDB sizes lead to the same results, using buckets reduces the time taken to store the nodes and the total time in general.

4.1.2 Partial Expansion A*

To continue in this direction we can avoid to store too many nodes accordingly to Yoshizumi et al. [2000] by using a PEA* algorithm.

Test conditions:
- 500 random problems for sizes 10 and 11, 300 for 12 (the average stays almost the same between 300 and 500 random problems) and only 20 for 13 (6/20 failures for each variant).
- The PDB size is 9.
- The total time is split in 3 categories, the time taken to generate the nodes, to compute their heuristic values and to store them in the open list.
- We count 3 categories of nodes. The number of expanded nodes (whom the successors are generated), the number of generated nodes (stored or not) and the number of stored nodes.

Table 3: Time comparison between the A* with buckets and the 2 variants of PEA*

If we look at the Table 3 we can notice a reduction of the storage time for both variants of PEA* as expected. It comes from the significant reduction of stored nodes as we can see in the Table 4. When the difference between the size of the genome and the size of the PDB grows, the percentage of stored nodes decreases. It is around 1%. However, we can also noticed that the general time is not improved for any PEA* variant. In average the total time taken remains the same for the 3
algorithms. Indeed we generate as much nodes as for the A* algorithm and we have to compute their heuristic values to decide if we can avoid to store them. Or this is clear that most of the time taken by those algorithms is taken to compute the heuristics.

<table>
<thead>
<tr>
<th>Size</th>
<th>A* with buckets (number of nodes)</th>
<th>PEA* (number of nodes)</th>
<th>PEA* with buckets (number of nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expanded</td>
<td>Generated</td>
<td>Stored</td>
</tr>
<tr>
<td>10</td>
<td>5.086</td>
<td>2905.106</td>
<td>2905.106</td>
</tr>
<tr>
<td>12</td>
<td>95.120</td>
<td>96358.048</td>
<td>96358.048</td>
</tr>
<tr>
<td>13</td>
<td>464.571</td>
<td>604408.429</td>
<td>604408.429</td>
</tr>
</tbody>
</table>

Table 4: Node comparison between the A* with buckets and the 2 variants of PEA*

Also we measured the number of enqueued nodes and by comparing it to the number of expanded nodes we notice an exact difference of 1 (the last enqueued node is the goal so it is not expanded). With our precision it means that there is less than 1 problem over 1000 that enqueue a node already expanded. If it is not possible for the classical A* algorithm, it is for the variants that use buckets. Indeed if each bucket is sorted accordingly to the h-value of the nodes in it, to reduce computation time we insert them without checking if they were already generated before.

To reduce more the time the next step is to act on the node generation or at least to avoid to compute the heuristic value if we know that the node is not promising.

One way to do it is to use the PMA* variant but it requires to compute the f-value of the partial node at each step which implies to compute the heuristic value very often and as we want to reduce the number of heuristic computations we didn’t pursue in this way. Also it is not possible to link the effects of a sub-action to the effects a the full action. For instance, on the sequence $g = (1, -3, -2, 4, 5)$ The action that inverts the second gene (I(1,1)) doesn’t seem to be promising but is a sub-action of the action that inverts the second and third genes (I(1,2)) and this second action leads to the goal so should be promising. So this is not a good way to approach the problem.

Another variant we wanted to adapt is the EPEA* algorithm. It aims to avoid to generate all the successors of a given node and hence to avoid to compute all the heuristic values. However, it appeared difficult to find a problem based OSF as required for this algorithm (see section 3.1.3) so we also didn’t pursue with this variant.

4.1.3 PrefPEA*

Finally we tried to exploit the time taken to compute the heuristic to extract some indications on a set of preferred actions that can be used to partially expand a node with the hope that those preferred actions are going to lead to the solution. Of course if not, to conserve the completeness, we will have to totally expand the node as usual with the set of all enabled actions. Then as we can see on the Figure 3 this modified algorithm acts on the set of actions.

The idea is to store in the PDB file not only the exact cost for a specific sequence but also a set of preferred actions. This set of preferred actions is simple to determine when we construct the PDB, it corresponds to the last action of each path with the smallest cost that lead to this specific sequence starting from the goal. The goal corresponds to the ordered sequence as explained in the section that presents the previous work done by Wong. If a shorter path is discovered, we update the cost and clear the set of preferred actions before adding the new one. However, the
size of the PDB file was already important so we were able to store only one of the preferred actions and we decided arbitrarily to take the last action of the first shortest path that we find during the exploration. This way we were only able to generate a PDB with preferred actions of a size 7.

Also when we look at the heuristic value in the PDB, we can’t take the associated action as preferred action because the actions in the PBD are not on the same size of the genome so we need to adapt them to the real size of the current sequence. To compute the heuristic we took a sample of genes in the current sequence by taking random indices and depending on the locations of those indices we will be able to construct a set of possible preferred actions for the real sequence.

Example: For the sequence \( g = (A, B, C, D, E, F, G, H) \) of size \( n = 8 \), we take the random sample of size 5 from the indices \{1, 3, 4, 5, 7\}. We obtain the sequence \( g_{\text{sampled}} = (B, D, E, F, H) \) that we rename in \( g' = (\alpha, \beta, \gamma, \delta, \epsilon) \). We can assume that if we look to the PDB, the preferred action for the sequence \( g' \) is the action \( T_5(0, 2, 2) \).

![Figure 7: Example of sampling with preferred action on the abstract sequence](image)

For each action we can split the genome in 3 sections as shown in the Figure 7. In red the moved section which corresponds to the section that is inverted and/or moved in the sequence. In green the offset section which is empty for an inversion. And in blue the rest of the sequence which is not modified by the action.

To adapt this preferred action from the abstract state to the real sequence we must include each extra gene strictly between two genes of a section of the action in this specific section. In the example we must include \( C \) in the moved section (in red) as \( B \) and \( D \) both belong to it. However, for the 2 remaining genes, \( A \) and \( G \), there are located between 2 different sections and it is possible to include them in one or the other. Hence we can build several actions from one preferred action on the abstract state. Here \( T_5(0, 2, 2) \) with this particular sampling becomes the set \( \{T_8(0, 4, 2), T_8(0, 4, 3), T_8(1, 3, 2), T_8(1, 3, 3)\} \).

In fact when we transform an action from the abstract state to the real state, do we keep the preferred character for each action in the generated set of actions?

**Experiment 1:** Precision of the set of preferred actions compare to the set of enabled actions and a subset of random actions of the same size of the set of preferred actions.

To determine the percentage of best actions that exists among all the enabled actions, the idea is to compute the exact cost of each successor for a given genome sequence. Then the number of successors that have the smallest cost equals to the number of best actions and we can easily calculate the corresponding percentage.
We can also determine the equivalent percentage only on a subset of actions and to measure the efficiency of the PrefPEA* algorithm, we calculated it on the set of actions generated when we compute the heuristic value (by adapting the preferred actions from the abstract states generated during the process).

More, we can take a random subset of actions with the same cardinality and do the same to see if it is better to take the subset based on preferred actions than a random subset.

<table>
<thead>
<tr>
<th>Size</th>
<th>Best f-value</th>
<th>Nb best actions</th>
<th>Best actions (%)</th>
<th>Best f-value</th>
<th>Nb best actions</th>
<th>Best actions/ Pref actions (%)</th>
<th>Best f-value</th>
<th>Nb best actions</th>
<th>Best actions/ Pref actions (%)</th>
<th>Best f-value</th>
<th>Nb best actions</th>
<th>Best actions/ All best actions (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2.46</td>
<td>15.896</td>
<td>13.951</td>
<td>3.96</td>
<td>0.725</td>
<td>19.84</td>
<td>4.224</td>
<td>0.392</td>
<td>13.993</td>
<td>2.452</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2.857</td>
<td>12.587</td>
<td>6.878</td>
<td>3.052</td>
<td>0.687</td>
<td>9.192</td>
<td>4.375</td>
<td>0.375</td>
<td>9.302</td>
<td>3.564</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3.239</td>
<td>22.924</td>
<td>7.944</td>
<td>4.025</td>
<td>1</td>
<td>8.84</td>
<td>4.358</td>
<td>0.375</td>
<td>8.710</td>
<td>3.547</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3.718</td>
<td>23.039</td>
<td>5.633</td>
<td>4.342</td>
<td>0.93</td>
<td>8.662</td>
<td>4.354</td>
<td>0.93</td>
<td>6.621</td>
<td>2.411</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Comparison between preferred and random actions

The Table 5 shows the results for these 3 cases for different sizes of genome and for an average on 1000 random sequences. The PDB used to generate the subset of preferred actions has a size equals to 5. For each case we have an average of the smallest f-value found among the f-values of the successors (Best f-value), an average of the number of successors that have this smallest f-value (Nb best actions) and the percentage that it represents among all the generated successors (Best actions/Pref actions). Also when we use a subset of actions we calculate the percentage of best actions found (Best actions/All best actions). For instance if there are 20 best actions and that the subset of actions contains only 5 of those actions then the percentage will be 25%. The average of this value is in the last column of each case where it is applicable. The colour code of each column title corresponds to the colour code in the Figure 8.

Figure 8: The different sets of actions

For both subsets we can see that the average of best f-values is greater than the one from all actions. It means that there are some cases where there is no best actions in those subsets. In fact we find in average less than 1 best action even if we slightly increase the percentage of best actions among the successors. For example on the first line, we can see that there is an average of 15.486 best actions per sequence and only 0.723 in the set of preferred actions. However, 15.486 represents 13.951% of the actions when 0.723 represents 19.63% of the preferred actions. As expected, if we
take a random subset of actions, we keep the same proportion of best actions. It’s only the average number that changes.

The goal is to maximize the red part in the Figure 8 with an ideal case where the set of preferred actions equals to the set of best action. Also, having $S_{\text{bestActions}} \subseteq S_{\text{prefActions}}$ and $S_{\text{prefActions}} \subseteq S_{\text{allActions}}$ is interesting since we prune several actions by being sure that we keep all the best ones. In this case we can discard definitively these actions and not put them aside temporary when we expand a node for the first time as we do in the PrefPEA*.

**Experiment 2:** Effects of different samplings on the set of preferred actions.

If the subset of preferred actions seems to give a slightly better probability to pick one of the best actions if we want to expand partially a node as we do in the PrefPEA*, we can try to increase even more this probability. In fact to generate the subset of preferred actions we use a random sampling and maybe other techniques can lead to more actions or to a greater proportion of best actions. We can try to take a subset of consecutive indices to avoid to have an extra gene to include inside one of the section of the action. For example, to avoid to have gene like $C$ in the Figure 7. In fact it acts like a sliding window that we can move on the sequence to fix local mutations in the genome. Another sampling could be to take the indices of genes in the current sequence that are following genes in the goal sequence. The idea is to find the best action to reorder genes that we know they are following in the goal sequence.

<table>
<thead>
<tr>
<th>Size</th>
<th>Best actions</th>
<th>Nb best actions %</th>
<th>Best actions/ Pref actions (%)</th>
<th>Best f-value</th>
<th>Nb best actions</th>
<th>Best actions/ Pref actions (%)</th>
<th>Best actions/ All best actions (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3.06</td>
<td>0.724</td>
<td>0.64</td>
<td>4.186</td>
<td>0.992</td>
<td>20.031</td>
<td>0.497</td>
</tr>
<tr>
<td>7</td>
<td>3.525</td>
<td>0.687</td>
<td>0.832</td>
<td>13.382</td>
<td>0.428</td>
<td>10.341</td>
<td>0.27</td>
</tr>
<tr>
<td>8</td>
<td>3.929</td>
<td>0.762</td>
<td>0.862</td>
<td>4.414</td>
<td>0.968</td>
<td>5.596</td>
<td>0.684</td>
</tr>
<tr>
<td>9</td>
<td>4.342</td>
<td>0.95</td>
<td>0.862</td>
<td>4.414</td>
<td>0.968</td>
<td>5.596</td>
<td>0.684</td>
</tr>
</tbody>
</table>

Table 6: Comparison between different samplings for preferred actions

The Table 6 shows the same results than in the Table 5 for the 3 different sampling techniques used to compute the preferred actions. The tests were conducted in the same conditions and on the same sequences. We can see that the sliding window seems to give better percentage than the random sampling but excepted for genomes of size 6, the average of f-value is greater with this sliding window. It means that we find the best f-value more often with the random sampling but when we find it with the sliding window, we generate more best actions. Also we have the same results with the indices of following genes, the average of best f-value is greater than with the random sampling.

**Experiment 3:** Runtime comparison between A* with buckets, PEA* and PrefPEA*.

Finally, after those tests we can see that from a best action on an abstract state, we generate only few best actions on the real state and that the best way to generate this set of preferred actions is by taking random samples of indices in the sequence. This is the configuration that we are going to take to compare the PrefPEA* algorithm with the A* with buckets.

As we could generate only a PDB file with preferred actions of a size 7, we can solve problems only with a maximal size equals to 11 and even in this case we sometimes had to stop the algorithm when it took more than 100s. The Table 7 shows the results for an average of 500 random sequences.
Table 7: Time comparison between the A* with buckets and the PrefPEA*

Here we can see that the total time is reduced. It is mainly because the time taken to compute the heuristic values decreased. The Table 8 shows different statistics concerning the management of the nodes during the search, they were calculated in the same conditions than the results of the Table 7. It explains why the heuristic time was reduced, we generate (and hence store) less nodes with the PrefPEA* than with the A*. This reduction is roughly between 20 and 40%.

Table 8: Node comparison between the A* with buckets and the PrefPEA*

Table 9: Time comparison between the PEA* and the PrefPEA*

We saw in the previous subsection that the PEA* algorithm gives the same results than the A* in terms of time taken but considerably reduces the number of stored nodes. In the Tables 9 and 10 we compared the PEA* and PrefPEA* to see which one reduce the most the number of nodes generated and stored. At first glance, the PEA* seems to be better as it stores less nodes than the PrefPEA* but in fact it also generates more. And this is when we generate a node that we compute the heuristic value so finally in average the PrefPEA* takes less time than the PEA*.

Table 10: Time comparison between the A* with buckets and the PrefPEA*

4.2 Pruning techniques

As seen in the last section, the PrefPEA* algorithm gives better results than other A* variant and by avoiding to generate all the successors of a node, it acts in a way on the branching factor. So reducing the branching factor seems to be a promised way to explore to improve the efficiency of the algorithm.
4.2.1 Symmetry Elimination

As saw in the section 3.2.2, the goal here is to cluster the same states with a different representation. As the genome is circular, we have $n$ different representations for the same sequence. However in our case, we take the canonical representation of each state that we have to generate. The canonical representation of a genome is the sequence that starts with the smallest number used to represent a gene without considering the sign. It's a rotation of every other sequences that represent the same genome. For example, the canonical representation of $g = (−2, 4, 3, −5, 1)$ is $g_c = (1, −2, 4, 3, −5)$. Hence the state space contains exactly one representation of every genome. This way we already do a symmetry reduction and the previous modifications that need to be apply to our algorithm won’t reduce more this state space. Indeed, to generate $G_{S_i, S_G}$, we need to find the permutations that stabilizes both the start and the set of final states. And as we said, we have exactly one representation of the final state and there is no operation that stabilizes the goal sequence so $G_{S_i, S_G} = \emptyset$ and every state is in a different orbit so we have to consider all of them.

Knowing this we can determine the current branching factor. In the generalized NT model there are 3 different possible actions on the genome, Inversion (I), Transposition (T) and Inverted Transposition (IT). Let be $n \geq 1$ the size of the genome.

For each possible start of an inversion ($n$ possibilities) and each possible length ($n$ possibilities), there is a distinct successor in the canonical representation except for the inversions with a length equals to $n$. If $l = n$, the start as no influence and every inversion leads to the same state in its canonical representation ($\forall i, j \in [0, n − 1], I(i, n) \sim I(j, n)$). Hence there are $nb_I(n) = \sum_{s=0}^{n-1} \sum_{l=1}^{n-1} 1 + 1 = n(n − 1) + 1$ inversions.

A transposition moves a section of the genome to another place in the sequence. So the first constraint on the parameters $s$, $l$ and $o$ is $l + o < n$ with $l \geq 1$ and $o \geq 1$. Indeed, as we saw previously, a transposition separates the sequence in 3 distinct sections (moved, offset and free section) and each of them needs to contain at least one gene and also the sum of the length of each section can’t be superior to $n$. More if we consider the same 3 sections for an action, the 3 transpositions than move one of the sections between the 2 others lead to the same state in the canonical representation as we can see in Figure 9. As we need to keep the first gene at the beginning of the sequence to have the canonical representation we can manage to keep it every time in the free section to avoid to rotate the genome after applying the action. It corresponds to the transposition $T_3(2, 2, 1)$ in the example of the Figure 9. So doing this we have more constraints such as $s \geq 1$ and $s + l + o \leq n$ as we can’t move the first gene, $s \leq n − 2$ as we need to have at least 2 genes after the start (1 for the moved section and 1 for the offset section), $l \leq n − s − 1$ as we need to have at least 1 gene in the offset section. Hence there are $nb_T(n) = \sum_{s=1}^{n-2} \sum_{l=1}^{n-s-1} \sum_{o=1}^{n-s-l} 1 = \frac{n(n-1)(n-2)}{6}$ transpositions.

### Table 10: Node comparison between the PEA* and the PrefPEA*

<table>
<thead>
<tr>
<th>Size</th>
<th>A* with buckets (number of nodes)</th>
<th>PEA* with buckets (number of nodes)</th>
<th>PrefPEA* (number of nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expanded</td>
<td>Generated</td>
<td>Stored</td>
</tr>
<tr>
<td>7</td>
<td>3.086</td>
<td>565.738</td>
<td>565.738</td>
</tr>
<tr>
<td>8</td>
<td>4.006</td>
<td>1126.686</td>
<td>1126.686</td>
</tr>
<tr>
<td>9</td>
<td>12.012</td>
<td>4913.908</td>
<td>4913.908</td>
</tr>
<tr>
<td>10</td>
<td>68.528</td>
<td>39130.488</td>
<td>39130.488</td>
</tr>
<tr>
<td>11</td>
<td>760.14</td>
<td>586068.94</td>
<td>586068.94</td>
</tr>
</tbody>
</table>
For every transposition there are exactly 3 inverted transpositions, we can invert one of the 3 different sections of the transposition. So there are $nb_{IT}(n) = 3 \cdot nb_{T}(n)$ inverted transpositions.

Finally the branching factor equals to

$$b_n = nb_{I}(n) + nb_{T}(n) + nb_{IT}(n) = \frac{n(n-1)(2n-1)}{3} + 1.$$  

We saw in the state of the art that applying one of the techniques of partial order reduction relies on the idea of independency between enabled actions. However, it is impossible to separate the set of actions described before in 2 independent subsets. Clearly in the GED problem, if 2 operations act on a common part of the sequence (at least one common index) then they are dependent. Hence the action that reverses the all genome, $I(0,n)$, is dependent with every other operations (it acts on all indices). This constraint on the set of actions prevents us to use the techniques of POR but we can try to go further in the symmetry reduction by trying to prune duplicate paths.

4.2.2 Pruning Duplicate Nodes

A way to detect and prune duplicate paths is described by Taylor and Korf [1992]. It aims to generate a Finite State Machine (FSM) that remembers the sequence of actions applied and prohibits actions that lead to a node that can be generated by a shorter path. In fact this FSM maps the state space by allowing only one path to each node by exploiting symmetric rules in the problem description.

**Principle:** For the GED problem, as every mutation modifies only the position of the genes and not the genes themselves, if two sequences of actions from a given node lead to a same node (different or not from the first) then these two sequences are symmetric for every starting node and it appears interesting to prune one of the path. The idea is to precompute a set of duplicate paths and to use this set during the search. In our case, we precompute only the duplicate paths of a length equal to 2. So it will be easy to prune them by keeping in every node the previous action and when a node is expanded we ignore all actions that are the second action of a duplicate path.

The Figure 10 presents an example of pruning action. We assume that the set of duplicate paths includes the path $(t_3,t_4)$. The initial state $s$ is fully expanded as it has no previous action. It generates among others the state $s_2$ through the action $t_3$. If $s_2$ needs to be expanded then we know that it is useless to apply to it the action $t_4$ as we know that there exists another state generated at the same time as $s_2$ (here $s_1$) and an enabled action in this state $s_1$ (here $t_2$) such as $t_2(s_1) = s' = t_4(s_2)$ and that the cost of the path $(t_1,t_2)$ is lower or equals to the duplicate path $(t_3,t_4)$.

We can see in the Table 11 the number of paths of length 2 and the number of duplicate paths of same length that on different sizes of genome. If the proportion of duplicate paths in regards of
Figure 10: Reducing state generation by pruning duplicate paths

Table 11: Number of generated duplicate paths

<table>
<thead>
<tr>
<th>Size</th>
<th>Nb paths of length 2</th>
<th>Nb duplicate paths</th>
<th>Duplicate paths (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>33498</td>
<td>25264</td>
<td>75.4</td>
</tr>
<tr>
<td>8</td>
<td>78961</td>
<td>56533</td>
<td>71.6</td>
</tr>
<tr>
<td>9</td>
<td>167281</td>
<td>114487</td>
<td>68.4</td>
</tr>
<tr>
<td>10</td>
<td>326041</td>
<td>214597</td>
<td>65.8</td>
</tr>
<tr>
<td>11</td>
<td>594441</td>
<td>253104</td>
<td>42.6</td>
</tr>
</tbody>
</table>

the total number of paths of length 2 drops from 75% to 40% it still allow us to reduce by 40% the branching factor on a size of genome equals to 11.

However, it takes a long time to precompute all the duplicate paths (between 1 and 2h for a size of the genome equals to 11). Indeed, the naive method that we used is a breadth-first search that explores the state space by increasing the depth. Each time we generate a node, we check if it was already explored. If it is, we compare two costs to reach it (old and new node) and we add in consequence the most expensive sequence of actions to the set of duplicate paths. To have all the sequences of 2 actions we set the maximal depth to 2. In this case, with a branching factor $b_n = \frac{n(n-1)(2n-1)}{3} + 1$, we generate $b_n^2$ paths that we have to be processed ($O(n^6)$).

To use the same technique of duplicate paths on larger sizes of genome, the idea is to take a duplicate path on a smaller size and to transform its actions in enabled actions on the larger size. One way to extend a genome of size $m$ to a genome of size $n > m$ is to take one gene on the first genome and to consider it as a block of several genes. Hence all we have to do to transform an action is to know the index of the duplicate gene, determine in which part of the action this gene is (unmodified section, moved section or offset section) and modify the action in consequence (see the example in Table 12).

**Example:** We want to extend the action $T_5(0, 2, 1)$ on the genome $g = (1, 2, 3, 4, 5)$ by duplicating the gene at the index $i$. The Table 12 shows different cases that imply different modifications on the action.
Doing this, if two paths lead to the same node, they are going to lead to the same node if we consider the same gene as a block for all the actions in both paths (not the same index but the same gene). Indeed, for a specific index, 1 action gives 1 and only 1 action after transformation, then 1 path gives 1 and only 1 path. This operation is injective.

Also we can consider other ways to take this gene and the size of the block, it gives us several alternatives. 4 of them are following:

- As explained before the simplest way is to take 1 fixed index and to consider the gene at this location as 1 block of \( n - m + 1 \) genes.

- Also there are \( m \) possibilities to chose an index in the sequence, we could improve the first alternative by moving the fixed index to every possible one and then theoretically (if all the generated paths are different) it can generate \( m \) duplicate paths on a size \( n \) from 1 duplicate path on a size \( m \) (for a total of \( m \times \#S_{\text{duplicatePaths}}(m) \) paths).

- Another way is to take randomly 1 gene \( n - m \) times (with the possibility to pick the same genes several times) and to consider these picked genes as blocks of a size equals to the number of times they were picked plus 1. Hence we have \( m \) choices for \( n - m \) pickings which gives theoretically a total of \( m^{n-m} \times \#S_{\text{duplicatePaths}}(m) \) paths.

- A fourth way to increase the theoretical number of generated paths is to increase step by step the number of gene in the genome (by one every time). It gives us \( m \) choices to pick a gene for the first step and \( n - 1 \) choices for the last step. Hence we obtain theoretically \( m \times \#S_{\text{duplicatePaths}}(m) \) paths after the first iteration, \( m \times \#S_{\text{duplicatePaths}}(m) \times (m+1) \) paths after the second one and \( m \times \#S_{\text{duplicatePaths}}(m) \times (m+1) \times \ldots \times (n-1) = \frac{(n-1)!}{(m-1)!} \times \#S_{\text{duplicatePaths}}(m) \) paths in total.

The only problem of the last 3 alternatives is that if we take different indices to generate different paths, we can generate many times the same paths and possibly generate that is known as critical path for another duplicate path.

**Example:** In the Figure 11, we consider the paths \( P_1 = I_5(1,1)I_5(2,1) \) and \( P_2 = I_5(1,1)I_5(2,2) \) where \( P_1 \) is a duplicate path and \( P_2 \) a critical one, then by using the index \( i = 2 \) we generate the path \( P_1' = I_6(1,1)I_6(2,2) \) from \( P_1 \) and by using the index \( i = 4 \) the path \( P_2' = I_6(1,1)I_6(2,2) \) from \( P_2 \). Hence we have \( P_1' = P_2' \), or \( P_2' \) was the only known alternative path to the path in blue so it is possible that we prune the only path leading to \( s_{22} \) that corresponds to the state \( s_2 \) where we duplicated the gene at the index 4 in the state \( s \).
In fact we can see that we generate and prune several critical paths when we count the number of paths generated by these advanced alternatives. For duplicate paths on size 7 that we want to extend to duplicate paths on size 8, we keep 56859 paths by using every possible index to transform the gene in a block (the second alternatives presented before) or in the file of duplicate paths on size 8 there are only 56533 different ones as we can see in the Table 11 so we kept at least 326 critical paths.

We are sure to prune only duplicate paths only by using the first alternative and in this case we generate exactly the same number of paths than the number of duplicate paths in the file. For example if we use the duplicate paths file on size 7, as we can see in the Table 10, if we have a problem of size 9 or 10 we generate 25264 paths. Hence the proportion of generated paths in regards of the number of duplicate paths decrease down to 10% if we have a problem of size 11 (4% of the paths of length 2). However, it takes only an average of 30ms to generate them so it seems still useful to consider them during the search.

**Experiment:** Runtime comparison between A* with buckets and PrefPEA* with pruning or not.

**Test conditions:** The tests are conducted with a PDB of size 7 and a duplicate paths file of size 7 too. For each size of genome we take an average on 500 random sequences except for 11 genes where it’s an average on only 100 repetitions as, again, the different algorithms sometimes fail to find the solution in less than 100s.

Applying this technique gives better results than the classical A* algorithm as we can see in Table 13 but we can notice an increase in the number of expanded nodes in Table 14. Those results can be explain if on two paths that lead to the goal we expand the first part of the pruned path. For example, on the Figure 10, assuming that the state $s'$ is the goal and all actions have the same cost, when $s$ is expanded $s_1$ and $s_2$ are generated and have same f-value (both verify $g = 1$ and
So there is no reason to have $s_1$ before $s_2$ in the open list to be the next node to expand. Depending on this order, it’s possible that we expand $s_2$ first and as we prune $t_8$, we are not able to reach the goal. We have to continue the search and hence expand $s_1$ and generate the goal through $t_2$. Finally we expanded 3 nodes ($s, s_2, s_1$) while we would have expanded only 2 nodes without any pruning ($s, s_2$).

Table 13: Time comparison between different variants with pruning

<table>
<thead>
<tr>
<th>Size</th>
<th>A* with buckets (ms)</th>
<th>A* with buckets and pruning (ms)</th>
<th>PrefPEA* with buckets and pruning (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generation</td>
<td>Heuristic</td>
<td>Storage</td>
</tr>
<tr>
<td>7</td>
<td>3.086</td>
<td>565.738</td>
<td>565.738</td>
</tr>
<tr>
<td>8</td>
<td>4.06</td>
<td>1141.86</td>
<td>1141.86</td>
</tr>
<tr>
<td>9</td>
<td>12.647</td>
<td>5173.487</td>
<td>5173.487</td>
</tr>
<tr>
<td>10</td>
<td>58.847</td>
<td>33602.447</td>
<td>33602.447</td>
</tr>
<tr>
<td>11</td>
<td>744.752</td>
<td>574205.158</td>
<td>574205.158</td>
</tr>
</tbody>
</table>

Table 14: Node comparison between different variants with pruning

Finally, the Figure 12 shows the average time taken by all the pruning variants that we studied for different sizes of the genome for 500 random sequences. The PDB size is 7 and the duplicate paths size 7. We want to focus on the improvements at the limit size which is 11 genes in the genome. We can conclude that the PrefPEA* with pruning is the fastest variant among all we tried during this study.

5 Conclusion

To conclude, all the variants of the A* algorithm seen in the state of the art were not explored considering certain constraints of the GED problem. After running some tests on this classical algorithm, we saw that most of the time is spend to compute the heuristic values and to stored the nodes when the size of the open list starts to be important. The first idea to reduce the time spend to store the nodes was to split the open list in buckets. Also some variants like PEA* decreased considerably the number of stored nodes which reduced even more the storage time. However, it still generates all the nodes so the second approach was to try to reduce the number of heuristic computations. The PrefPEA* managed to decrease it and hence to reduce the total time taken by the algorithm which pushed us to go further in this direction. The last improvement that we did was to prune a set of duplicate paths of length 2. Finally we managed to decrease the rate of failures (execution in more than 100s) of the algorithm but due to the complexity of the GED problem, we still can’t execute the algorithm in reasonable time on larger sizes of genome.

Table 14: Node comparison between different variants with pruning

Future work: To pursue in this direction we could experiment the 2 other alternatives presented to generate the set of duplicate paths of length 2. Also we could study the effects of the consideration of only a subset of actions with the characteristic that they cannot create a new breakpoint. The
bounds of the moving section must be 2 breakpoints as well as the location where this section is going to be moved. It might reduce more and more the branching factor as number of breakpoints decreases during the search. Also it is not obvious that the edit distance implies the absence of creation of new breakpoints so we need to be careful to keep the completeness of the algorithm. On the other hand, finding a better heuristic than the PDB heuristic (which means at least as good and calculable in less time) could cut the time taken by the heuristic computation which is as we already said the part that takes most of the time of the search.

References


