Constraint Satisfaction for the Genome Edit Distance Problem

Joshua Nelson
supervised by Dr. Patrik Haslum

The Department of Computer Science
Australian National University

October 2013

Contents

Abstract

Calculating genome edit distances is a computationally difficult problem, but an important one in the field of biology. In this thesis, constraint satisfaction is applied to an instance of the problem with inversion, transposition and transversion operations. Positional and relational encodings are modelled with the MiniZinc CSP specification language, and it is found that the relational model is more natural and efficient.

In order to calculate distances for longer genomes, symmetry breaking and redundant constraints are implemented on these models. It is found that these additional constraints improve efficiency considerably, and there is much potential to find symmetry breaking constraints for both of these models.

A compression algorithm for circular genomes is presented, and it is found that real world mitochondrial genomes can often be compressed by a significant amount – most compressed genomes fall into classes of length 2-10, 18-22, and 32-37.

Since the problem is computationally difficult, approximations to this method are evaluated – particularly, inversion only distance, and the breakpoint distance. It is found that there are few differences between these approximations and the Inversion / Transposition / Transversion method, but these small differences are enough to influence phylogenetic tree construction, and so cannot be discounted.

Chapter 1

Introduction

In this chapter, the genome edit distance problem is introduced, and the biological background knowledge required is reviewed. Motivation for solving the problem and previous attempts at the problem are discussed.

1.1 Background

1.1.1 The motivation

For as long as humanity has existed, we have been fascinated with animals. Part of this fascination has led us to attempt to group the animals in some way – naming them, sorting them, and classifying them. This area of study is known as *Biological classification*.

One way of classifying species is through the construction of *Phylogenetic trees*. A phylogenetic tree represents evolutionary relationships between groups of organisms, called species. The tree is a hypothesis for the evolutionary ancestry of species.

Our interest in this thesis focuses on the relationships between different species. To classify with a phylogenetic tree, we require a measurement of how similar a pair of species are. Ideally, this measurement would precisely represent the evolutionary distance between two species. However, the exact history is unknown, so we must use the data we have to approximate the evolutionary distance. We use these distances to devise a hypothesis of the history of evolution in the form of a phylogenetic tree.

Apart from curiosity, construction of phylogenetic trees has many practical applications in the field of biology. When new species are discovered, and little is known about them, phylogenetic analysis can indicate a species' close relatives. The traits of these relatives are likely to be reflected in the new species if there is a short distance between them in the phylogenetic tree.

This has been useful for synthesising drugs – if an organism produces some substance useful for drug creation, it is likely that a close relative will also produce this. Phylogenetic study can reveal new pathways to synthesising drugs [1].

Knowledge of the phylognies of viruses can give us information about their behaviour, and predict the behaviour of future outbreaks [2]. For poisonous animals, it may be the case that closely related species' venoms are treatable by the same antivenom [3].

For these reasons and more, we seek to devise some measurement of distance between species, and from there, create a phylogenetic tree based on these distances that closely represents the evolutionary history of a set of organisms.

1.1.2 Biology background

An organism's traits are defined by its *genome*, which is the sum of all hereditary information encoded in the organism. This hereditary information is encoded in blocks called *genes*. A gene is a collection of DNA, but our analysis will take place at the gene level (DNA sequence comparison has been done in the past, but it is usually the case that gene order analysis is more accurate [4]).

Gene order is considered a good candidate for phylogenetic study due to the *neutral theory of genome evolution* [5]. This theory says that when comparing genomes of species, most gene order differences are fitness neutral, that is, they do not change the physical traits of the organism, and so there is no selective pressure on many of the mutations. We also know that the rate of mutation is constant over time. Therefore, the number of mutations can give a good indication of time since genetic divergence.

A genome is a string of genes, possibly connected circularly, as is the case with mitochondrial genomes [6]. Mitochondrial genomes contain 37 genes – 13 proteins, 2 ribosomal RNA's (rRNA), and 22 transfer RNA's (tRNA's) [7]. They are useful for genome sequence analysis and comparisons between species, as they are relatively small, and the DNA molecules are abundant and easy to isolate [6]. Mitochondrial genomes are also useful because genetic information is inherited directly from the mother. This removes sexual combination of genetic information, making mitochondrial genome comparison a more consistent measure of species divergence over time. This mutation rate for mitochondrial genomes is very slow, enabling us to study changes in species that diverged in the distant past.

Genes connected to each other may have one of two orientations – we will refer to these as the *positive* and *negative* orientations throughout the paper, and we will call this property the *sign* of the gene.

Over time, through the process of evolution, species diverge genetically. This divergence occurs through mutation. A mutation is an operation that transforms a genome in some way – for gene sequences, we usually consider some subset of the following operations:

- Transposition: a segment of genes is moved from one location to another in the

genome

- Inversion: the order and direction of the genes in a block are reversed
- Transversion: a combination of inversion and transposition on the same gene block
- Duplication: gene content is duplicated and placed in another location of the genome
- Deletion: genes are dropped from the genome

The mutation of a parent genome in an offspring genome is the main source of new genetic information during the course of evolution. However, there are other influences – for example, it has been observed that genetic data from other species can have an impact on gene content of the genome. This is known as *horizontal transfer*, as opposed to the parent/offspring transfer known as *vertical transfer* [8, 9]. This is most often the case for single celled organisms, such as bacteria.

1.1.3 Simplifying assumptions

Operation costs Some assumptions must be made to abstract the biological problem to a computational one. The major assumption that we make is some model of evolution – we assume that genomes mutate with a few set operations (some of the operations from § 1.1.2), and with a relative frequency (see § 4.4.1 for information on these costs). While there is good evidence that these are the operations that occur in nature, and they occur with this frequency, it is still an assumption that we make, and it should be remembered that other influences are possible (for example, horizontal gene transfer).

Duplication and deletion § 1.1.2 also mentions the operations of *Duplication* and *Deletion* – removing and adding gene content. However, in the models we use, we will ignore these operations, and only examine genomes of equal length and content. Finding the operations necessary to delete and duplicate content are computationally simple, and for computing a true distance between genomes of varying length and content, the costs of these operations should be considered. Our model will look at genomes of equal length and content (usually the case for mitochondrial genomes, or synthetically created genomes), and possibly assume that any necessary deletion and duplication operations that are required have already taken place. The cost of these additions and deletions are outside the scope of this paper.

Optimal path Another assumption that is made is that the evolutionary process will take the shortest path between two species. We assume in our models of the problem that the evolutionary distance between two species corresponds with the optimal evolutionary distance (the least cost path using a predefined set of operations). This assumption is known as the *maximal parsimony*, or *minimum evolution*, principle [10]. The principle is similar to Occam's razor, where the explanation with the least assumptions (or evolutionary steps) is preferred. This principle is used for tree construction in Chapter 9.

Horizontal transfer Horizontal transfer is another biological factor that may reduce the accuracy of our measures [8, 9, 11]. Our model assumes that the only source of genetic modification is through the operations transposition, inversion, and transposition, mutating a parent genome into an offspring's genome. However, as discussed in § 1.1.2, this is not necessarily always the case. It is a fair assumption, as the majority of genetic information mutates through vertical transfer, however, the accuracy of this measure of distance depends on the relative infrequency of horizontal transfer compared to vertical transfer. This should be kept in mind while assessing the accuracy of this genome edit distance measurement. This is particularly important for bacteria and other single celled organisms [9].

1.2 ITT model problem description

This section describes the evolutionary model that we use for the genome edit distance problem, and a specification of the problem that we use for the rest of the paper. It describes the problem independent of any encoding choices and solution methods.

1.2.1 Genomes

We are given two genomes, G_1 and G_2. Both G_1 and G_2 are a type of permutation of the same set, $\{g_1, g_2, g_3, ..., g_n\}$, where $|G_1| = |G_2| = n$.

Genomes G_1 and G_2 are circular in nature (See Figure 1.1). This means that a genome is defined relationally (that is, we can specify a genome by listing the neighbours of each gene).

Genes in each genome are also oriented in one of two directions. This orientation is indicated with a $\pm$ sign.

1.2.2 Operations

We define operations on genomes that modify the state and the order of the genes in some way. Each operation is a function that takes a genome, and produces another

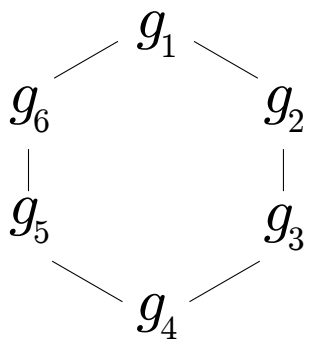


Figure 1.1: A genome of length $n = 6$.

modified genome. Additional arguments may be required to define the operation.

Inversion inversion(genome, block)
This operation takes a genome and mutates it, so that the ordering of genes in the selected block reversed. It also negates the sign of each gene in the block. (See Figure 1.2a)

Transposition transposition(genome, block, amount)
This operation takes a genome and mutates it, so that the selected *block* is moved by *amount* in the new genome. (See Figure 1.2b)

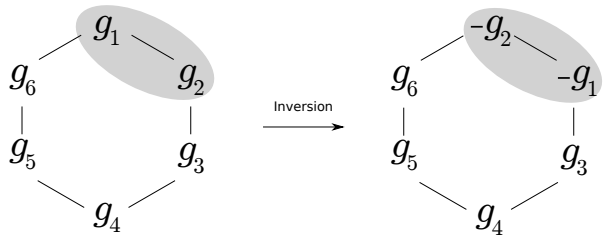
Transversion transversion(genome, block, amount)
Transversion is a composition of an Inversion operation, and a Transposition operation.

Note that the argument "block" is used – this "block" argument is intended to refer to some contiguous segment of the genome – however, the exact arguments used will depend on the encoding selected (with our positional encoding, we encode the block with a start and end *index*, and the block is shifted in some direction, arbitrarily chosen to be clockwise. With the relational encoding, we specify by the block by the start and end gene *value*).

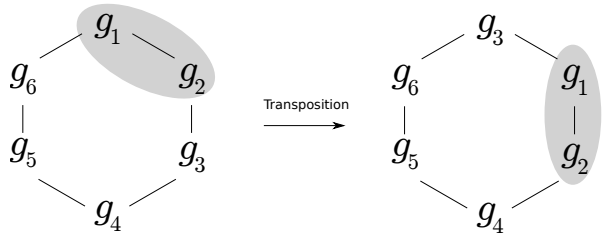
This model, with the Inversion / Transposition / Transversion operations, is referred to as the ITT model throughout the paper.

1.2.3 The edit distance problem

The task is to find a sequence of functions $f_1, f_2, ...$ and arguments that will mutate some genome G_1 into some other genome G_2. We assign some cost to the three



(a) A genome with the block g_1, g_2 being inverted. Note the change of sign for the inverted segment.



(b) A genome with the block g_1, g_2 being transposed by amount 1, clockwise

Figure 1.2: The inversion and transposition operations

types of mutation function – transposition, inversion, transversion. The additional requirement of optimality means that we seek to minimise $\sum_t \text{cost}(f_t)$

1.3 Previous attempts

We can classify the approaches for finding measurements of similarity between genomes into two classes – optimal and non-optimal approaches. An *optimal* approach is one that attempts to calculate the smallest cost distance between two genomes, assuming the model of evolution described in § 1.2. A *non-optimal* approach attempts to estimate this optimal distance with another, simpler to compute metric, possibly providing an optimal solution to a simplified version of the model.

1.3.1 Non-optimal approaches

Genome comparisons have been done for some time, however, not always with the ITT model of evolution. Some other models that have been used in the past are the Inversion Only (IO) model [12], and the *transposition only* model [13]. These models restrict possible operations to only the inversion and transposition operations, and each does not allow the other. Solutions to the IO model can be found in $O(n^2)$ time. The IO model is discussed further in § 6.2.

Additionally, we can perform a linear time static analysis on the gene with methods like breakpoint analysis, discussed in § 6.1.

Modifications of simple string edit distance calculations can also provide a rough approximation to the evolutionary difference between two genome sequences. This is discussed further in § 2.3.

1.3.2 Optimal approaches

Previous work on this model of genome evolution has been done with planning and with heuristic search algorithms. The planning approach is discussed in § 2.2.1.

A* heuristic search

Previously, A* heuristic search had been used to solve the ITT model of this problem [14]. The results show that an iterative deepening search is most effective, coupled with heuristics like breakpoint distance, and counting differences in gene signs.

A pattern database was used to guide the search. It was found that memory becomes an issue for the pattern database with patterns of genome size 9-10. Overall, the limits of the algorithm were found at genome length ≈ 15.

1.4 Research questions

This thesis seeks to answer the following questions about the genome edit distance problem.

What is the best way to phrase the genome edit distance problem as a constraint satisfaction problem? Is this a good way to approach the problem, what are the advantages and disadvantages? How can constraint satisfaction be better harnessed for this problem?
The relational model's formulation is less obvious, but ultimately more intuitive for specifying constraints. The relational model is more efficient than the positional encoding overall (See § 5.4).

A positional model was also trialled for the problem, with time states being represented by state variables, and actions being represented as constraints between consecutive time states (See Chapter 4). This model was found to be less efficient than the relational model, and overall more difficult to write constraints for due to operations that wrap over the boundary of the linear array.

Constraint satisfaction has potential for this problem, and there is research potential in finding more symmetry breaking constraints, and better models of the problem (possibly using dual modelling, see § 4.1.2).

What is the state of the art for solving the ITT genome edit distance problem? Which known algorithms procedures are the fastest, and which provide results

closest to ITT?
Attempts to solve this problem include PDDL planning, heuristic search, breakpoint analysis and IO algorithms. Breakpoint analysis and IO algorithms are faster as they are approximations to the ITT model. (See Chapter 2, Chapter 6)

Constraint satisfaction has potential for this problem, but the work done on the A* heuristic search algorithm provides the fastest solutions [14].

Approximations to the ITT model exist, and the distances correlate with ITT distance, but the structure of phylogenetic trees produced with each model varies, so they are not sufficient approximations if ITT is assumed to be the true model of evolution. (See Chapter 9).

Can we incorporate knowledge of the problem in the solver, or in the problem specification, with some heuristic?
We can incorporate knowledge of the problem by pre-processing the genomes, and adding constraints on plan cost that provide an upper bound for the solver. (See § 4.5)

What are the properties of real world biology genomes, and what can we learn about biological genomes to help us compute edit distances for them?
When comparing biological genomes, there is often a great deal of similarity. On average, mitochondrial genomes of length 37 can be compressed to length 17.0297 ± 11.2350, but many genome comparisons allow compression to lengths 4 and 19. Few genomes are compressed to the lengths 11-16. The smaller compressed genome set is within the range of optimal ITT solvers, but the larger group is out of range. (See § 8.3)

What impact does slight variation in the distances have on tree construction? Is the difference between approximations to ITT and the full ITT model's costs significant enough to warrant extra computation time?
It was found that there is a significant difference between phylognies produced with the different methods, meaning that IO distance and breakpoint distance are poor approximations to ITT for biological data (See Chapter 9).

How do the IO and breakpoint models compare with the ITT model, in terms of speed and accuracy?
It was found that the overall, the approximation distances correlate with the ITT distance, however, there is some unpredictable variability on some datasets. (See § 7.1, § 7.2)

IO and breakpoint algorithms can be executed in polynomial time, but there is no known polynomial time algorithm for calculating ITT distance, so the approximation methods are far superior in terms of speed.

Chapter 2

Literature review

This chapter presents a survey of current literature relating to constraint satisfaction and genome edit distances.

2.1 Constraint satisfaction

2.1.1 Constraint satisfaction for planning and scheduling problems

Barták and Salido [15] present multiple open issues for constraint satisfaction problems. They suggest that generating a good model for problems can be difficult, and that an appropriately expressive and simple language is necessary. In addition, it suggests the use of "open global constraints" for modelling planning problems. These are constraints on multiple variables, and they are advantageous for simple expression of complex concepts. Solvers can also interpret global constraints in a way that is efficient to the particular solver.

This indicates that availability of global constraints should be a criteria when choosing a modelling language for modelling the genome edit distance problem.

2.1.2 Revisiting Constraint Models for Planning Problems

Barták and Toropila [16] discuss the problem of translating a planning problem into a constraint satisfaction problem. This has an application in formulating a genome edit plan as a constraint satisfaction problem.

The paper mentions the problem of plans being of variable length, while constraint models must be of a fixed length. The proposed solution in this paper is to iteratively attempt to find plans of length $1, 2, ..., n$, until a satisfiable model is found.

The paper also suggests multiple approaches for improving the performance of solvers on the constraint model, specifically, approaches that cater to the nature of planning problems. The techniques discussed are *symmetry breaking*, *singleton consistency*, *nogoods*, and *lifting*.

Symmetry breaking is an important enhancement, and one that has an application

for the genome edit distance problem. When formulating an edit plan for a gene sequence, many choices do not interfere, and plans are symmetric. Actions that do not interfere with each other can be executed in any order, which inflates the search space unnecessarily. This can be avoided by imposing an arbitrary order on the actions.

These enhancements all seek to reduce the size of the search space. When looking for performance improvements of the constraint model, reducing the size of the search space in any way should be the goal.

2.1.3 Constraint Satisfaction Techniques in Planning and Scheduling

Barták et al. [17] discuss various methods of searching for constraint satisfaction solutions, including complete and incomplete searches. The paper mentions *incomplete search algorithms*, which are algorithms that may not return a solution if one exists – failure to find a solution does not imply that there are no solutions. However, it may be able to quickly find a non-optimal solution. Such methods may be worth examining for the genome edit distance problem – while not optimal, they can give a rough idea of the cost and length of an optimal plan.

The paper also introduces the technique of formulating a planning problem as a CSP model. It suggests a model where each time step of the plan is represented as a different state, and constraints are imposed on sequential states, depending on the action chosen at the different time steps.

2.1.4 Constraint Satisfaction Problems on DNA Strings

Bortolussi and Sgarro [18] examine the use of constraint satisfaction techniques for calculating hamming distances between DNA strings (the DNA word design problem). While most of the paper relates to the DNA word design problem, the problem shares some similarities with the genome edit distance problem. Both problems have large state spaces that inhibit the use of constraint satisfaction techniques.

Some methods discussed in the paper for reducing the size of the state space include adding constraints which attempt to break the symmetry of the problem, and introducing heuristics that guide the search toward better solutions, so that CSP search can be more effective (heuristics can lead to better results earlier, allowing more pruning and reducing the size of the search space).

2.2 Domain-Independent Planning

2.2.1 Computing Genome Edit Distances using Domain-Independent Planning

Haslum [19] phrases the ITT genome edit distance problem as a planning problem (using PDDL). Several formulations of the problem are examined; however, the most intuitive formulation was not the most efficient (the multi-step relational formulation performed better than the single-step positional formulation). The advantage of using domain-independent planning is the flexibility to change edit operations and their costs, but coming up with formulations of the problem such that current planners can solve it efficiently requires knowledge of the workings of the planner. The promise of a biologist being able to phrase models of mutation in a planning language for easy experimentation has not yet been fulfilled, and so it is concluded that better methods of finding high-quality plans are required.

2.3 Genomic distance measures

2.3.1 Gene Order Breakpoint Evidence in Animal Mitochondrial Phylogeny

Blanchette et al. [20] examine the use of genomic distance measures in the field of biology. They indicate that an important use of such measures is in the construction of *distance matrices*, that is, comparison of a particular genome with a set of other genes. This matrix can be used for the construction of a phylogenetic tree (using various algorithms discussed and compared in the paper).

The construction of this matrix requires many genomes to be compared for a useful result to be produced. Therefore, a computationally cheap algorithm for estimating genome edit distances is desirable. The paper proposes the method of *breakpoint analysis*, which has $O(n)$ running time in the length of the genome.

This indicates that low running time is an important characteristic of an algorithm for computing genome edit distances. Another useful research result would be an analysis of the accuracy of the estimations produced by *breakpoint analysis*. If the optimal gene edit distance is known, we can compare the results obtained by *breakpoint analysis*, and see how well correlated this measure is with the optimal edit distance. How breakpoint distance is calculated, and a comparison with the optimal distance, can be found in § 6.1.

2.3.2 Current approaches to whole genome phylogenetic analysis

Sawa et al. [8] provide an overview of whole genome analysis, as opposed to traditional DNA and protein sequence comparisons. These whole genome methods

attempt to provide a metric that correlates with the true evolutionary distance between genomes.

Methods examined include calculating distance by inversions only (which can be evaluated in $O(n^2)$ time), and the *breakpoint analysis* distance, discussed by Blanchette et al. [20]. Additional methods proposed include static analysis of gene content (the proportion of genes that are shared between two genomes). Statistical methods are also discussed (maximum likelihood, and Bayesian methods).

Comparisons of these metrics with an accurate measure of true genome edit distances would give an indication of which methods are the most correlated.

The paper also concludes that due to the changing nature of biological research at the moment (with uncertain knowledge of the mechanisms of genome evolution, and whole genomes being only partially sequenced), methods should be robust against inappropriate models being used, and flexible to adjust to new models as they are proposed, agreeing with the assessment given by Haslum [19].

2.3.3 Parametric genome rearrangement

Blanchette et al. [21] introduce a novel approach for calculating costs for the operations of transposition, inversion, and transversion. Often, operations of inversion and transposition are given 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.

The method used to calculate the probable weighting between the two operations is to trial different weights for the operations, and find the point at which the normalised number of moves required to sort various genome permutations (random, and human) changes the most. It is found that the trade-off point is in the range $2 < w_t < 2.5$ (where w_t is the weight assigned to the transposition operation), as in this range, the normalised number of moves required to permute the genomes increases abruptly. It is likely that this is a good value, as for w_t within this range, "many transposition operations are retained despite their elevated cost, [which suggests] that this may be a meaningful solution"

Ultimately, the paper suggests a value of somewhat more than twice the weight to transpositions as inversions, and questions the results of papers that assign the same weight to both operations. It also suggests that assigning different weights to operations over different amounts of genes may provide better gene edit distance results.

The implementation of this in the MiniZinc model is discussed further in § 4.4.1.

2.3.4 Conservation of genome form but not sequence

Franklin [4] look at the use of DNA sequence comparison rather than the higher level

gene order comparison. These methods calculate distances based on insertion and deletion of sequence content, rather than on gene order. Gene order based distance measures provide more accurate distances for species where gene order is preserved, but gene content is often not [4]. This is the case with some viruses, for example, the herpes virus [13], which features a high degree of gene rearrangement.

Chapter 3

CSP overview

In this chapter, a brief overview of constraint satisfaction is provided, which will provide a background for the discussion of the application of constraint satisfaction to the genome edit distance problem.

3.1 CSP overview

3.1.1 A definition of CSP's

A constraint satisfaction problem (abbreviated CSP) can be informally defined as a set of variables, each with a domain, and a set of constraints on these variables. Each variable can take any value from the specified domain. The constraints further restrict the valid values that variables can take, in various ways.

Problems are defined within a CSP by these constraints. *Binary constraints* are constraints that restrict only two variables; for example, a constraint may restrict the variables x and y with $x > y$. Complicated constraints, involving more than two variables, can be broken down into a collection of simpler, primitive constraints.

A global constraint is a constraint that affects a set of variables at all stages of the solving procedure. They are high level constraints that influence the domains of many variables. These constraints should be used as often as possible, as solvers can translate them into constraints as efficiently as possible, and the global constraints can be specifically catered for by solvers.

Solving CSP's

A solution to a CSP is an assignment of values to variables that satisfy all of the constraints on them. For some problems, any satisfying instantiation is considered a sufficient solution. Alternatively, we may seek the "best" solution, in some way. This may be determined by a cost function, depending on what is considered a desirable solution. A solver may seek the plan with the shortest length, or a solution that minimizes weight, price, time, or another measurement of solution quality.

Solutions to CSP's can be found via a backtracking search algorithm. A naïve solver could search the domain of every variable, and check to make sure that each constraint is satisfied. Solvers can make many improvements on this though, through a number of techniques. Some methods include branching on variables with the most constraints on them, and pruning the search tree.

The advantage of CSP's

The advantage of using a CSP for modelling a problem is that generalised solving techniques can be employed, regardless of the problem domain. The modelling language requires no knowledge of the solving techniques that will be used, and the solver does not require any knowledge of the problem domain. However, when writing the model for CSP, care must be taken to ensure that as much knowledge from the problem domain is incorporated into the model as possible, to allow solvers to make use of all of problem information that it can (this is done in the form of constraints, and adding search heuristics)

3.2 An introduction to MiniZinc

MiniZinc is a constraint satisfaction modelling language used to formulate these problems. MiniZinc aims to be a medium level modelling language, "high-level enough to express most constraint problems easily, but low-level enough that it can be mapped onto existing solvers easily and consistently" [1]. The high level nature of the language allows different solvers to interpret MiniZinc constraints in the most efficient way they can. It supports many global constraints, and is beginning to be adopted as a standard language for constraint satisfaction problems.

For more information on MiniZinc, see `http://www.minizinc.org/`.

[1] `http://www.minizinc.org/`

Chapter 4

Positional MiniZinc CSP model

In this chapter, a description of the MiniZinc CSP implementation of the ITT genome edit distance problem is described. A *positional* encoding is used. In addition, symmetry breaking and redundant constraints that improve solver performance on the model are described.

4.1 Problem encoding

4.1.1 Positional encoding

The genome edit distance problem was encoded as a constraint satisfaction problem using a positional encoding system. An array S_t was declared, encoding the state of the genome at each time step t. Another array, A_t specified the action taken from time step $t \rightarrow t+1$ (in practice this was represented with multiple arrays – see § 4.2.2). Constraints were then applied that restricted consecutive states from S_t to be valid transitions, according to the action array A_t.

The states in S_t were encoded as one dimensional arrays of genes, indexed by their position. That is, $S_t = [g_1, g_2, ..., g_n]$, where g_i is the gene at index i and time step t, and n is the length of the genome.

It must be remembered, however, that this state is actually circular in structure. This means that the last element of S_t is adjacent to the first element. Therefore, S_t is equivalent to the state $[g_2, ..., g_n, g_1]$, and all other rotations (n rotations total).

This must be kept in mind when writing constraints for the model.

4.1.2 Other encodings and dual modelling

Relational and reverse positional encodings

Haslum [19] proposed a *relational* encoding for plan states, rather than a *positional* encoding. This is discussed further in Chapter 5.

Other encodings exist, such as the "opposite" of the positional encoding – that is, rather than variables being the positions, and the domains being the genes, we could

encode the gene as the variable, and the possible positions as the domains of these variables.

Dual modelling

While it may seem like we must choose a particular encoding for our model, this is not necessarily the case. It is possible to use a *dual modelling* technique to use both models. Smith [22] suggests that this approach is well suited to permutation problems, which often have multiple options for problem encodings. After implementing both models as one model, we can introduce "channelling constraints" that link the two encodings of the problem, to force both to be in the same state at all stages of the search. Then, constraints can be intuitively expressed in whichever encoding is most natural, and inference on either model can be reflected in the other.

It is also only necessary to constrain one of the sub models, as the channelling constraints will link each version together to ensure consistency (however, for redundancy, it is better to include as many constraints as possible).

Combining these two approaches is a possible extension of the approaches implemented in this paper, and encoding with both may provide additional performance benefits.

4.2 Framing the problem in MiniZinc

This section provides an overview of the positional model that was created in MiniZinc, and includes information on the model's parameters, and main variables used.

4.2.1 Input parameters

Several input parameters help define a problem for the MiniZinc solver. These are

1. The initial genome
2. The goal genome
3. The penalties assigned to each operation
4. A pre-computed upper bound on the plan cost (explained further in § 4.5.2)

More control over how the solutions are found can be gained by modifying the model file directly.

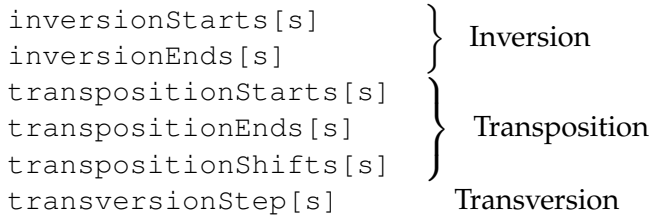


Figure 4.1: A list of variables specifying the parameters for each operation at a particular time step `s`

4.2.2 Variables

The state of the problem according to the positional MiniZinc model is uniquely defined by the contents of several variable arrays. These arrays specify which operations were performed at each step, and the parameters to these operations. The arrays that specify arguments are listed in Figure 4.1.

Each of these arrays are indexed by the plan step variable `s`. The inversion arrays specify the start and end positions for inversion operations, and the transposition arrays specify the start position, the end position, and the shift that we apply to the blocks of genes.

It can be seen that there is only one variable array for transversion, however. This is a Boolean array, and specifies at each step whether a transversion took place. If so, genes will be inverted as they are transposed, leading to a transversion. This allows transversion operations to "piggyback" on the transposition constraints, allowing for fewer constraints overall, and less decision variables, which helps performance.

It may also seem from this variable representation that we allow multiple operations at the same time step `s`. However, this is not the case. Constraints were added to the model that prevent multiple operations at a given time – if one operation has parameters set at a particular step, all other parameters at the step are set to a "no operation" value, which indicates that these operations did not occur at that step (in practice this was a negative value, which would not correspond to a valid operation parameter).

If all of the operation parameters at a particular step `s` are set to this "no operation" value, we say that no action occurred at that step at all. Allowing "no-actions" solves the problem of variable plan lengths. This is discussed further in § 4.5.1.

s					
1	4	-1	-3	-2	-5
2	1	-4	-3	-2	-5
3	1	2	3	4	-5
4	1	2	3	4	5

Table 4.1: An example of the `planStates` array, with inversion operations acting on indices 1-2, then 2-4, then 5. Indices are numbered from 1 to 5, left to right.

4.2.3 Plan states and goal states

Plan states

As discussed in § 4.1.1, a positional encoding was used to model the plan. This was represented by a two dimensional decision variable array in MiniZinc, named the `planStates[s][i]` array. This array is indexed by `s` - the current state index, and `i` – the gene index. Each row of this two dimensional array represents the genome at a different time step. So, for example, the `planStates[s][i]` array may look like Table 4.1.

Initial and goal states

The initial state for a particular problem is an input parameter, and is unchangeable. We are also given a goal genome, and the objective is to transform the initial genome into the goal genome. In the example given in Table 4.1, the initial genome is step `s=1`, and the goal genome is step `s=4`. We constrain the genes at these steps directly to the input parameters, and allow the solver to fill in the intermediate plan states according to the operation variables specified in Figure 4.1.

Remembering genome rotations however, we cannot constrain the `planState` variables *directly* to the input initial and goal genomes. Additional constraints must be created to allow rotations of the state, and possibly complete inversions of it too, as these are equivalent states.

The problem of transforming one genome into another is equivalent to sorting a permutation into the identity. This is true, as we can perform substitutions on the indices in the goal genome. For example, if the goal genome was $(-1, -4, 5, 3, 2)$, in the initial genome, we could replace -1 with 1, -4 with 2, 5 with 3, and so on, so that the initial genome has been modified, and the goal genome is now the identity.

This problem simplification is useful, as it eases implementation of some constraints, and it allows simpler calculation of breakpoint values. An optimisation discussed in § 4.5.2 works on the assumption that the goal genome is the identity permutation.

4.3 Modelling operations

4.3.1 Inversions

One of the actions that we allow is the inversion operation. This operation reverses a block of genes, and flips the sign of the genes in that segment.

$$S_t = (g_1, \boldsymbol{g_2, g_3, g_4}, g_5, ..., g_n) \overset{\text{inversion}}{\rightarrow} (g_1, \boldsymbol{-g_4, -g_3, -g_2}, g_5, ..., g_n) = S_{t+1}$$

Modular inversions

When encoding the constraints between consecutive steps where inversions have taken place, it was noted that no consideration was necessary for the cases where flips occur over the boundary of the array. This is a valid action, as the genome is circular. For example, inverting the following genome from index $n-1$ around to 2,

$$S_t = (\boldsymbol{g_1, g_2}, g_3, ..., g_{n-2}, \boldsymbol{g_{n-1}, g_n}) \overset{\text{inversion}}{\rightarrow} (\boldsymbol{-g_n, -g_{n-1}}, g_3, ..., g_{n-2}, \boldsymbol{-g_2, -g_1}) \quad (4.1)$$

is equivalent to inverting the range from 3 to $n-2$, and then inverting the order of the whole genome. Note that the genes we must invert are any genes that were not included in the original inversion operation.

$$S_t = (g_1, g_2, \boldsymbol{g_3, ..., g_{n-2}}, g_{n-1}, g_n) \overset{\text{inversion}}{\rightarrow} (g_1, g_2, \boldsymbol{-g_{n-2}, ..., -g_3}, g_{n-1}, g_n) \quad (4.2)$$

It may not appear to be equivalent due to the differences in sign and order between (4.1) and (4.2). However, (4.1) produces an inverted version of (4.2), which we consider an equivalent state. So if we allow any inversion operation to optionally invert the whole genome at any step, we can fully allow all valid genome operations without explicitly considering the case where inversion operations occur over the boundary of the array.

We allow this by letting the final goal state be possibly inverted (for most cases, this will be the inversion of the identity, or the identity).

Therefore, for any inversion over the boundary of the one dimensional array S_t, there exists an equivalent inversion operation that does not act over the boundary of the array. This means that we do not need to consider modular inversion operations when writing constraints.

4.3.2 Transpositions

A transposition is the movement of a block of adjacent genes to another location within the genome. For example,

$$S_t = (\boldsymbol{g_1, g_2, g_3}, g_4, ..., g_n) \overset{\text{transposition +1}}{\rightarrow} (g_4, \boldsymbol{g_1, g_2, g_3}, ..., g_n) = S_{t+1} \tag{4.3}$$

Modular transpositions

It was also found that it is unnecessary to create constraints that allow transposition of blocks that wrap around the boundary of the genome, as there is always an equivalent transposition that does not wrap around the boundary. For example, consider transposing the block from index $n-1$ to 1, shifting the whole block forward by 1 index (See (4.4)). Note that g_2 is displaced by the transposed segment, and afterwards, its position is immediately before the transposed segment.

$$S_t = (\boldsymbol{g_1}, g_2, g_3, ..., g_{n-2}, \boldsymbol{g_{n-1}, g_n}) \overset{\text{transposition +1}}{\rightarrow} (\boldsymbol{g_n, g_1}, g_3, ..., g_{n-2}, g_2, \boldsymbol{g_{n-1}}) \tag{4.4}$$

We now create an equivalent transposition that does not wrap around the boundary of the array.

Let

- b_{shift} = the amount we are shifting the block forward by
- b_{start} = the first index of the transposition block
- b_{end} = the last index of the transposition block

For our example in (4.4), these would be $b_{\text{shift}} = 1,\ b_{\text{start}} = n-1,\ b_{\text{end}} = 1$. These parameters define the original transposition.

For the new transposition, let

- $b'_{\text{shift}} = -b_{\text{shift}}$
- $b'_{\text{start}} = b_{\text{end}} + b_{\text{shift}} + 1$
- $b'_{\text{end}} = b_{\text{start}} - 1$

This takes all genes we were not going to change before (the genes outside the transposition block, and the genes that were not going to be displaced), and moves them backward by the amount we were going to shift by. Taking our example (4.4), the new transposition would become

$$S_t = (g_1, g_2, \boldsymbol{g_3, ..., g_{n-2}}, g_{n-1}, g_n) \overset{\text{transposition -1}}{\rightarrow} (g_1, \boldsymbol{g_3, ..., g_{n-2}}, g_2, g_{n-1}, g_n) \tag{4.5}$$

It can be seen that this is equivalent to (4.4), rotated forward by 1. This makes them equivalent states, and therefore equivalent transpositions.

Negative shifts The above uses negative shifts, which were also not encoded in the constraints. However, we can similarly show that any transposition with a negative shift has an equivalent transposition with a positive shift. In the same way as before, we let,

- $b'_{\text{shift}} = b_{\text{start}} - b_{\text{end}} + 1$
- $b'_{\text{start}} = b_{\text{start}} - b_{\text{shift}}$
- $b'_{\text{end}} = b_{\text{start}} - 1$

This takes s genes before the start of the old block, and shifts it forward by the width of the old block. Shifting the displaced genes forward has the same effect as shifting the original block backwards.

4.3.3 Transversions

Transversions allow a transposed block to be inverted, and then transposed. Unfortunately, due to transversions being the composition of two operations, there is no equivalent operation to a transversion that wraps over the boundary of the array – similarly for negative shifts. This means that constraints that specifically consider these cases were required for the model.

4.4 Operation costs

4.4.1 Transversion operation cost

As discussed in [21], and the relevant section of the literature review (§ 2.3.3), a weight of 1 was assigned to inversions, and a weight of 2 was assigned to transpositions. Blanchette et al. [21] do not discuss how to assign a weighting to transversions. This was set to a value of 2 (the same as transpositions) for our experiments. Some insight may be gained from looking at assigning different weightings to transversions; however, it *is* known that

$$w_{\text{i}} < w_{\text{tv}} < w_{\text{tp}} + w_{\text{i}} \tag{4.6}$$

where w_{tp} is the transposition weighting, w_{tv} is the transversion weighting, and w_{i} is the inversion weighting. This is true, as $w_{\text{tv}} < w_{\text{i}}$ would cause all inversion operations to be replaced with transversions (a transversion with a shift of zero is equivalent to an inversion). Also, we know that it must be cheaper than performing

a transposition and then an inversion, otherwise transversions would never occur in our solutions, as performing the operations separately would be cheaper. For this reason we know that w_{tv} is in the range given in (4.6), which with our assignments, is in the range $(1, 3)$

4.4.2 Variable operation costs

As suggested by Sawa et al. [8], operation costs dependent on the length of the number of genomes being operated on may provide better results. For this model, variable costs for operation lengths was not considered, however, it would not be difficult to modify the model to allow this.

It is uncertain from biological evidence whether mutations on longer genome segments occur more or less frequently. However, a similar stochastic approach to that described by Blanchette et al. [21] may be taken to investigate whether or not this may produce more realistic distance results.

4.5 Optimisations

4.5.1 Symmetry breaking

Symmetry breaking is an important aspect of modelling planning problems as constraint satisfaction problems. As discussed in § 2.1.2, symmetries in plans can lead to unnecessarily inflated search spaces, greatly increasing the time to find good solutions, making problems intractable.

There are ways to remove these unnecessary branches – we can eliminate symmetric branches from the search tree, since they lead to the same solution. An illustration of this can be seen in Figure 4.2.

Operation index ordering

The main symmetry breaking constraint that was added to the model was the operation index ordering constraints. This constraint forces the indices that each operation affects to appear in (almost) increasing order.

We cannot force operations to be in strictly increasing order. There are cases where optimal plans may involve moving backwards along the ring. An example of this can be seen in Figure 4.3 – the ordering of the inversion operations impacts the plan, even though the operations are not in increasing order.

To allow for this, the constraint that was added ensured that no operation at time step $t + 1$ can take place completely before an operation at time step t. However, it may *start* earlier than a previous operation, as long as there is overlap (as long as step $t + 1$'s *end* index is greater than t's start index.

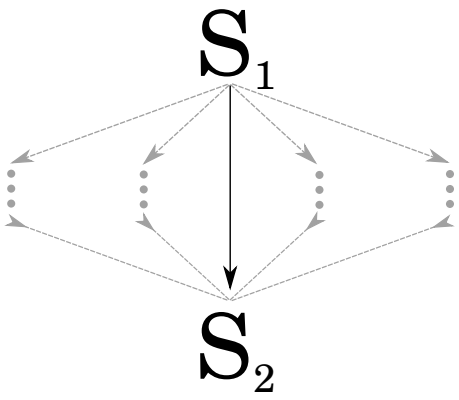


Figure 4.2: An illustration of the effect of symmetry breaking. S_1 and S_2 are states in the plan. Dashed arrows represent symmetric paths, and eliminating them reduces the search space, while preserving one route between them (the center arrow)

An illustration of this constraint can be seen in Figure 4.4. Here we can see three operations occurring sequentially along the genome – from g_1 to g_6. Operation 3 must occur after operation 1, since they are non-overlapping operations, and the plan must move forwards along the genome, or have some overlap. The ordering "operation 3, 1, 2" is eliminated, as it is equivalent to the ordering "operation 1, 2, 3".

The purpose of this constraint is to force a strict ordering on *independent* operations, but overlapping operations are *dependent*.

One exception to this rule is if operations occur over the boundary of the array. If this happens, then there will be modified gene content at the start of the array, giving us a reason to revisit the beginning.

The final constraint can be stated as something like this – if at some time step T_i, all modified genes precede all modified genes at time step T_j, then time step T_i occurred before time step T_j. No restriction is placed on the indices of modified genes if there is overlap between the operations at T_i and T_j, or if a modular operation took place.

We can know that this constraint does not eliminate valid plans, as we do not eliminate one of the orderings for the operations (Operation 1 is still allowed to occur before operation 3 in the example of Figure 4.4).

The MiniZinc code for this constraint can be seen in Listing 1.

No-action

A no-action operation was discussed in § 4.2.2. This operation is important, as it allows us to represent plans of any length, with a set plan state size. No-actions do not have any dependencies, so we can force each no-action to occur at the start

Positional model
File name: `genome.mzn`

```
%There must be some overlap / moving forward in the next operation
constraint forall(t in stepIndices) (
    noActionTookPlace(t) \/ noActionTookPlace(t+1) \/
    (
        (
            (firstGeneModified[t+1] < firstGeneModified[t]) /\
            (lastGeneModified[t+1] < firstGeneModified[t])
        ) -> (
         %We only allow this if a modular operation has occurred
         firstGeneModified[t] > lastGeneModified[t]
        )
    )
);
```

Listing 1: The operation index ordering symmetry breaking constraint.

...	3	(2	1)	...
...	(3	-1)	-2	...
...	1	(-3	-2)	...
...	1	2	3	...

Figure 4.3: An example of a plan that requires operations to move backwards along the genome. Genes inside parenthesis are reversed with inversions.

of the plan without consequence. This is a particularly useful constraint if the plan length is short as a proportion of genome length (we must allow plans to be as long as the genome itself for completeness, but it is likely that plans will consist of mostly no-action operations if this is the case)

The constraint that removes this symmetry can be seen in Listing 2.

4.5.2 Redundant constraints

Redundant constraints are constraints that are unnecessary for restricting solvers to valid solutions. They are useful for improving search time by allowing additional paths of inference for the solver, and they can help reduce variables' domains. Removing values from the domain of variables means that we do not need to branch on those variables / values.

When constructing these redundant constraints, it is useful to look at the properties of a valid plan, and come up with some constraints from these properties. A list of the redundant constraints that were used in this model follows.

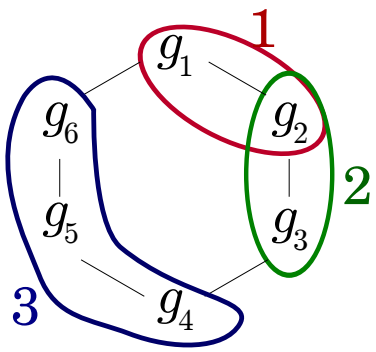


Figure 4.4: An illustration of symmetry breaking with index ordering. g_1 is designated as the start index, and we constrain operations $1, 2, 3$ to affect maximum indexes that are increasing in order (operations occur clockwise)

Positional model
File name: `genome.mzn`

```
%Do all noActions first
constraint forall(t in stepIndices) (
    forall(j in stepIndices) (
     %If no action happened at t, but it did at j,
     %then t happened first.
        (noActionTookPlace(t) /\
        (not noActionTookPlace(j))) -> t < j
    )
);
```

Listing 2: The no-action ordering symmetry breaking constraint.

Stationary genes constraint

A constraint was added to the model that specifies the circumstances necessary for a gene to remain stationary between steps. Most of the time, if a gene stays stationary between steps, it is because it is not involved in any operation. However, a gene may also remain in it's position if it is at the center of an inversion, so this was added as a special case. The code for this constraint can be found in Listing 8.

Blocks of modified genes

It was observed that genes are usually modified in whole blocks. By this we mean, the indices at which gene content changes between steps are usually adjacent indices. Again, an exception is the case when a gene is at the center of an inversion, hence stationary and not modified. Another exception is the case when transpositions wrap around the boundaries of the array. Apart from these two cases, genes are only modified in whole blocks.

The way that this was implemented as a constraint clarifies this property of valid solutions. Suppose the gene content at indices i_p and i_q has changed from time step t to $t+1$. Then, any index in between these two points, say i_r where $p < r < q$ must have been modified between t and $t+1$ as well. Otherwise, i_r is the center of an inversion, or a modular transposition took place – both easily checkable conditions.

The code for this constraint can be found in Listing 7.

Breaking apart sorted segments

Often genomes contain segments of genes that are not modified between the initial state and the final state.

This condition was added as a constraint. Suppose we have two genes which are adjacent in the goal genome, say g_1 and g_2. If these genes are at consecutive indices at some time step t, for all $t' > t$, g_1 and g_2 should still be consecutive. A consecutive arrangement may be g_1, g_2, or $-g_2, -g_1$ if the segment has been reversed. This prevents the pair from being broken apart.

Even though this constraint only explicitly deals with pairs of adjacent genes, it applies in all cases with consecutive sequences of genes. This creates a set of constraints that force larger, already sorted gene segments to stay together.

While this reduces symmetry, we can pursue this idea further by reducing the number of variables *before* any solving has been done. This can be achieved by *compression*, which is discussed in Chapter 8. Since the compression algorithm achieves the same goal, while eliminating additional variables and reducing their domains, this method of dealing with blocks of adjacent genomes was preferred, and so this constraint was left out of the final model.

Gene sign changes

Giving genes a sign doubles the size of their domain. To reduce the effect of this, constraints were added that specify under what circumstances negations of genes can occur.

The only circumstances under which a gene can change sign are:

1. The gene is inside an inversion operation

2. The gene is inside a transversion operation

If none of these cases hold, the gene must keep the same sign. This is often the case, for example, genes outside operations have signs preserved, and genes inside transpositions have signs preserved.

The code for this constraint can be found in Listing 9.

Breakpoint reduction

A redundant constraint was added that incorporates information about the breakpoint distance (introduced in the literature review (Chapter 2), with more details on how breakpoint distance is calculated in § 6.1). It was observed that an inversion operation can fix at most two breakpoints, and a transposition can fix at most three. It is also known that the breakpoint distance must be reduced to zero by the time we reach the goal state. Both of these properties are true, and if we can force them as constraints, then we can be more specific with our variable's domains, and reach unsatisfiability earlier. For example, if we know that the difference between breakpoint distance in consecutive steps is greater than 3, we will not be able to find operation arguments that transforms one genome into the other in one step.

Unfortunately, adding this required intermediate variables – one intermediate variable at each time step `s`, to track the breakpoint distance between that time step and the goal genome. This is the easiest way to perform complex inference in MiniZinc, but introduces some overhead.

The code for this constraint can be found in Listing 10.

Capping the plan cost

If we can come up with any plan, we may be able to improve search time by imposing a limit on the cost of the plan. The tighter this bound is, the more plans we will be able to invalidate.

We can use the IO distance (§ 6.2) to provide us with an upper bound on our plan, as an IO plan produces a valid plan for the ITT model also.

Chapter 5

Relational MiniZinc CSP model

There are alternatives to the positional encoding described in § 4.1.1. Haslum [19] uses a relational encoding with a planning approach to the problem. Using this encoding in a CSP model requires variables that store the neighbours of each variable, rather than variables storing the gene that is located at a particular position.

This relational encoding has a number of advantages over the positional encoding. The relational encoding allows simpler expression of some constraints on the system. For example, transpositions may be more naturally expressed as the modification of three variables between steps – the neighbour of the start of the transposed block, and the genes at either end of the transposed block.

There are a number of difficulties with implementing this encoding – there is no obvious way to encode inversions, without intermediate variables (Haslum [19] examines the possible approaches using the planning model). Approaches to overcome this in a CSP model are discussed in this chapter – gene sign can be used as a type of intermediate variable. The speed of the relational CSP model is also compared to the positional CSP model, and it is found that the relational CSP model has greater performance overall, but both have potential for speed improvements with improved symmetry breaking constraints.

5.1 Framing the problem in MiniZinc

5.1.1 Plan states

For the relational model, we use the same fundamental concept of linking plan states to operation parameters with constraints. However, we choose a different representation of both plan states, and operation parameters.

This representation is a *relational* encoding. This is different from the positional encoding, which uses a plan state array `planStates[s][i]`. The statement `planStates[s][i]=g` means that at state `s`, gene `g` is at position `i`.

The relational encoding is different, as it uses an array `leftNeighbour[s][g]`. This array encodes the left neighbour of the gene with absolute value `g` at each plan

g	1	2	3	4	5
leftNeighbour[g]	3	1	2	5	4

Table 5.1: A cycle present in the `leftNeighbour` array. The genome represented by this array would have disconnected cycles (1,3,2) and (4,5).

state `s`.

`leftNeighbour[s][g] = h` means that at state `s`, gene `g` is to the left of `h` in the circular genome. This has a number of advantages – it makes expressing transpositions simpler (only a few neighbours need to change, rather than a large block needing all indexes changed), and it allows simple expression of circular genomes (there is no "boundary" to the array, meaning no special allowances need to be made for operations that span this arbitrary boundary).

The `rightNeighbour[s][h]` array was also created, and a constraint enforced that the left neighbour array was consistent with the right neighbour array.

`rightNeighbour[s][g]=h` means that at state `s`, gene `g` is to the right of `h`. This allows simpler modelling of some operations, and either can be used wherever convenient.

The `leftNeighbour` and `rightNeighbour` arrays store the absolute relative positions of each gene, but do not track their signs. We place this information into a Boolean array `geneSigns[s][g]`, which gives the sign for each gene `g` at each time step `s`.

Ensuring valid plan states

An issue with using the relational encoding with the `leftNeighbour` array is that we may end up inadvertently splitting the genome – for an example, see Table 5.1.

To avoid this, we can use the `circuit` global constraint provided by MiniZinc. This constraint "constrains the elements of `x` to define a circuit where `x[i] = j` means that `j` is the successor of `i`" [1]. In this relational model, this appears as the constraint in Listing 3.

Relational model
File name: `genome-relational.mzn`

```
constraint forall(t in stateIndices) (
        circuit([leftNeighbour[t,i] | i in geneValues])
);
```

Listing 3: The circuit global constraint preventing cycles in the genome.

[1] MiniZinc 1.6 global constraint catalogue, see `http://www.minizinc.org/downloads/doc-1.6/mzn-globals.html`

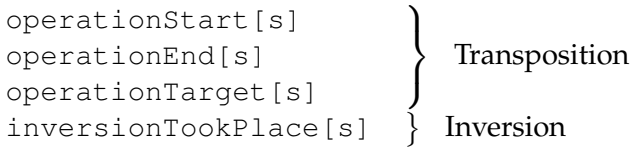


Figure 5.1: A list of variables specifying the parameters for each operation at a particular time step `s`, for the relational encoding

5.1.2 Operation parameters

Given that the plan state representation is relational, we need to express operation parameters in a relational way too. These new parameters are listed in Figure 5.1.

Rather than specifying the explicit parameters for each operation (Inversion / Transposition / Transversion), we use three arrays to specify any operation's boundaries – `operationStart[s]`, `operationEnd[s]` and `operationTarget[s]`.

`operationStart[s]` and `operationEnd[s]` specify the start and the end gene value for an operation at time step `s`. Note that this is gene value, rather than gene index, as was the case with the positional encoding.

These arrays provide parameters for a block to be transposed, and the `operationTarget[s]` array specifies where this block should be moved to. The block starting with `operationStart[s]` and ending with `operationEnd[s]` is moved to the left of the gene `operationTarget[s]`. Note that we specify a target rather than a shift to specify the movement of the transposed segment, as it is a more natural way of expressing transpositions relationally.

The `inversionTookPlace[s]` array is a Boolean array, specifying whether the operation at step `s` inverted the genes as well.

See Figure 5.2 for an example of how the parameters allow transpositions.

A no-action can be represented at step `s` if `operationEnd[s]` is already positioned to the left of `operationTarget[s]` (like a "stationary transposition"), and `inversionTookPlace[s]` is false. Alternatively, an inversion can be represented if `inversionTookPlace[s]` is true. Transversions occur if we have a non-stationary inversion, and transpositions occur if the segment is not stationary, and not inverted. These cases are formalised as predicates in Listing 4.

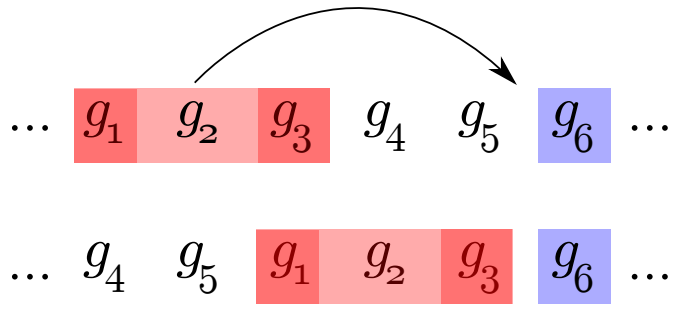


Figure 5.2: An example of the relational parameters `operationStart[s]`, `operationEnd[s]` and `operationTarget[s]`. In this example, g_1 is the operation start, g_3 is the operation end, and g_6 is the operation target – g_3 is moved to the left of g_6, and the block up to g_1 follows.

5.1.3 Operation costs

Using the action predicates of Listing 4, we can implicitly define the costs associated with each operation. For the relational encoding, this means that the costs are linked to the plan state variables directly (through predicates), rather than being linked to the operation parameters, which are in turn linked to the plan states.

5.1.4 Input parameters and output

The input to the model is the same as in the position model (§ 4.2.1), allowing simple direct comparison between the two models. However, as the input is simpler to express in a positional format, we use a positional encoding for the input initial and the goal genome, and must translate this into a relational encoding before we begin. This is done by simply forcing the neighbours in the positional array to be neighbouring in the relational array (at the first and final time steps), and also extracting the sign information from each input gene. The constraint that implements this can be found in Listing 11.

Plan output Since the relational MiniZinc model outputs plan states as relational arrays, it is difficult to tell whether a plan state is valid intuitively. To aid debugging, an output formatter was created that translated these relational plans back into positional ones. This allows simpler visualisation of the operations and their effects.

Relational model
File name: `genome-relational.mzn`

```
predicate isLeftNeighbour(var int: t, var int: g1, var int: g2)
  = leftNeighbour[t,abs(g2)] = abs(g1);

predicate operationIsMoving(var int: t)
  = not isLeftNeighbour(t, operationEnd[t], operationTarget[t]);

%Vanilla inversion (no moving)
predicate inversionAction(var int: t)
  = ((not operationIsMoving(t)) /\
    inversionTookPlace[t]);

%Vanilla transposition (no inverting)
predicate transpositionAction(var int: t)
  = (operationIsMoving(t) /\
    (not inversionTookPlace[t]));

%Transversion (moving and inverting)
predicate transversionAction(var int: t)
  = (operationIsMoving(t) /\
    inversionTookPlace[t]);

%No action (not moving, not inverting)
predicate noAction(var int: t)
  = ((not operationIsMoving(t)) /\
    (not inversionTookPlace[t]));
```

Listing 4: Action predicates for the relational model. Each of these determine which operation took place based on the plan states.

5.2 Modelling operations

5.2.1 Transpositions

With the encoding of transposition parameters described in Figure 5.1, transposition operations are easily allowed with a simple constraint. Assuming a transposition operation occurred from time state `s` $\rightarrow$ `s` $+ 1$, we add the constraint that at time step `s+1`, the gene value `operationEnd[s]` is left of the gene value `operationTarget[s]`. We do not need to manually move any genes in the middle of the transposed blocks, as the `operationEnd[s]` will "drag along" the other genes in the segment. The genes inside the transposed segment keep their neighbours, and so we only need to modify the start and the end of the block.

We also add constraints that link `operationStart[s]` to the right of `operationTarget[s]`'s original left neighbour – this links the other end of the transposed segment, inserting it in between the gene left of `operationTarget[s]` and `operationTarget[s]`.

This is complicated by a special case, however. It might be the case that we perform an transposition, but the segment to transpose is *already* to the left of the

operationTarget[s] (a stationary transposition, i.e. a no-action). In this case, we do not restrict the left and right neighbours of operationStart[s] and operationEnd[s].

All of the other gene's relative positions stay the same, and this is enforced by the general rules constraint that says genes usually keep their neighbours between steps (except for the genes near the operationStart[s] and operationEnd[s] described above).

An example of this can be seen in Figure 5.2, and the constraint that implemented transpositions can be found in Listing 5.

5.2.2 Inversions

An inversion operation is represented in the model by setting the inversionTookPlace[s] bit to true for that step. Doing this means that all genes inside the operation block must have their left and right neighbours flipped.

Finding which genes to invert

This is not as simple as it was in the positional encoding, as we have no way of simply determining whether a gene is within the block. One way we can tell whether a gene g is in an operation block is by following the chain of neighbours from the start (operationStart[s]) to the end (operationEnd[s]) of the block, and looking for g.

This is difficult though, as chain inference to find this gene along the leftNeighbour[s][g] array would require intermediate variables, storing whether or not the gene is included within the operation block at that step. If we had this, we could say that the leftNeighbour of a gene inside the block is inside the block, and the operationEnd[s] gene is inside the block (until we reach operationStart[s]). Following this chain, the solver could calculate which genes are contained inside the operation. Adding intermediate variables is something we would prefer to avoid though, as intermediate variables will decrease the efficiency of constraint satisfaction.

However, it is fortunate that we are required to keep track of geneSigns[s][g] anyway. We can use this array to keep track of which genes are involved in an operation. If the sign for gene g changes between step s and step s+1, then we know that g was inside the operation block. The constraint that executes this chain inference on the geneSigns[s][g] array can be found in Listing 6.

To summarise this constraint, we first assert that operationStart[s] and operationEnd[s] have their signs change. We then constrain that anything to the left of operationEnd[s] has its sign change, and anything to the right of operationStart[s] has its sign change. This sign change is forced until the

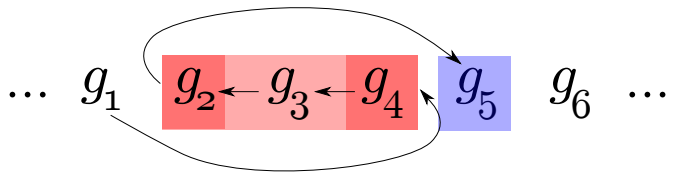


Figure 5.3: An example of a relational inversion. Genes g_2, g_3, g_4 are being inverted between steps s and $s+1$. g_2 is `operationStart[s]`, g_4 is `operationEnd[s]`, and g_5 is `operationTarget[s]`. The arrows on the diagram represent the "left neighbour of" relationship. So, g_1 becomes the left neighbour of g_4 at time step $s+1$, g_4 the left neighbour of g_3, and so on.

`operationStart[s]` and `operationEnd[s]` genes are reached (for inference in the *right* and *left* directions respectively).

Inverting the genes

We have already performed inference on the `geneSigns[s][g]` array, so we know which genes need their signs flipped. With this information, we can invert the order as well.

To reverse the order of a gene within a block, we must swap the left and the right neighbour of every gene in the block between steps, which is simple. However, we must specifically consider the cases for the genes `operationStart[s]` and `operationEnd[s]`. The gene to the left of `operationStart[s]` is now left of `operationEnd[s]`, and the gene to the right of `operationEnd[s]` is now right of `operationStart[s]`. `operationStart[s]`'s right neighbour becomes it's left neighbour, and `operationEnd[s]`'s left neighbour becomes it's right neighbour.

Figure 5.3 visualises this, and displays the changes we make in gene relationships between steps to invert the order of a gene segment.

5.2.3 Transversions

Few constraints were needed to cater for transversions, as a transposition with the `inversionTookPlace[s]` bit set to `true` allows a transversion. Inversions were essentially implemented as a special case of a transversion (a stationary transversion).

This is different from the positional encoding, which constrained the plan states differently depending on whether the `transversionStep[s]` bit was set. Implicitly allowing transversions simplified the model considerably.

5.3 Symmetry breaking constraints and redundant constraints

Many of the constraints from § 4.5 have analogies in the improved relational model, so they are not repeated here. Constraints that particularly impacted efficiency included the *no-action* ordering constraint (§ 4.5.1), as it eliminates a great deal of symmetry if plans are short as a proportion of genome length.

5.4 Speed result comparison with positional encoding

5.4.1 Speed result experiment

Because the positional ITT model and the relational ITT model produce plans of equal (minimum and optimal) costs[2], we wish to know which is most efficient in general.

Figure 5.4 shows the timing results on the Timing dataset. Note that a timeout of 6 minutes was set for each genome, and if it was not solved in this time, it received a time cost of 360 seconds. Results were computed on the system described in Appendix A. Results for both models were computed with the `minizinc -b lazyfd` solver, which was by far the fastest solver for both models.

Figure 5.4a shows the relative positional and relational timing results, as a function of genome length (note the logarithmic scale). We see here that the relational encoding performs better at all genome lengths. This relative advantage diminishes as we approach genomes of length 8 and higher. This may be due to the solvers having difficulty solving the problems within the time limit – 22/300 plans were unsolved by the positional model, and 14/22 were unsolved by the positional model.

Prior to adding symmetry breaking constraints, it was found that the positional model performed better at longer genome lengths.

Figure 5.4b also indicates that the relational encoding is faster than the positional encoding, but the results are broken down by the type of synthetic genome being used (see § B.1 for a list of the different generation modes). We can see that "Random" takes the longest time to solve for both models, which is consistent with our compression results [3].

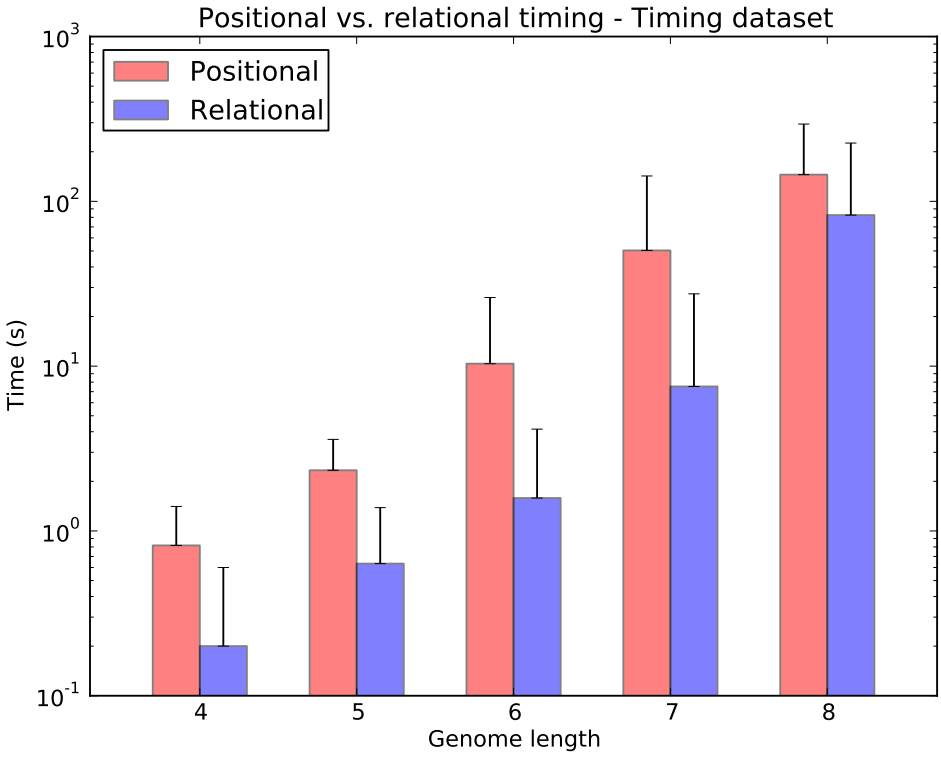
We see here that there are no significant changes between the two models and their abilities to sort synthetic genomes made with different operations. This tells us that neither method is better at finding plans that involve particular operations.

[2]This was verified through experimentation on the Short lengths and Timing dataset. While not a proof of equivalence, it gives some confidence that they are.

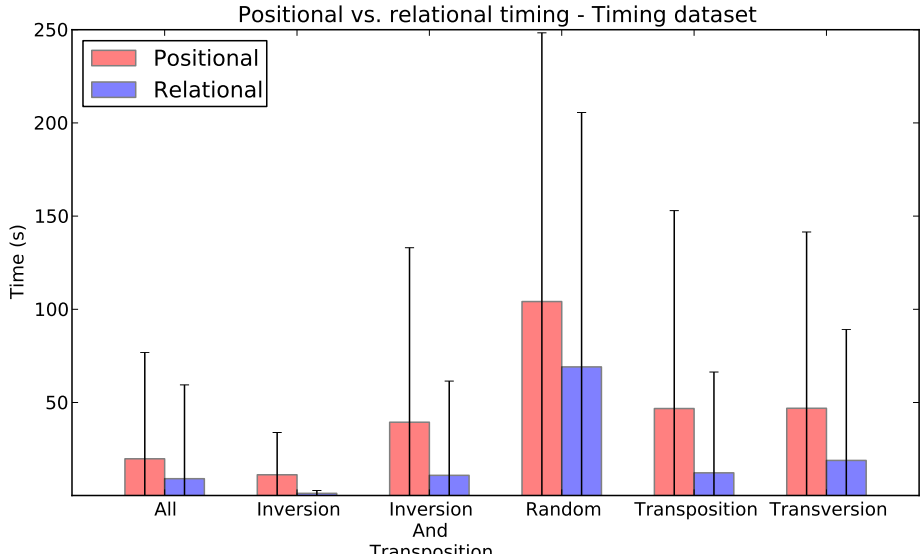
[3]Our compression results from § 8.2 show that the "random" mode is the most difficult to compress.

5.4.2 Speed results conclusion

From this experiment, it is clear that the relational encoding is superior in terms of efficiency. This is most likely due to the positional model's edge cases, and complicated constraints due to the problem of a circular genome being represented in a linear array. The relational model is recommended for the ease of expressing constraints, and for efficiency. It is believed that further research into symmetry breaking constraints for the relational model may yield even better timing results.



(a) Timing data for the positional and relational MiniZinc models, by input genome length. Note that the y axis is logarithmic. Error bars represent one standard deviation



(b) Timing data for the positional and relational MiniZinc models, by input genome type. Positional is the left bar, relational is the right bar. Error bars represent one standard deviation

Figure 5.4: Timing results for the positional and relational MiniZinc models. Positional is the left bar, relational is the right bar. Data taken from the Timing dataset.

Relational model
File name: `genome-relational.mzn`

```
%---------OPERATIONS---------
%----TRANSPOSITION----
constraint forall(t in stepIndices)
(
  (
    %The case where an inversion did not take place
    %(link the start and the end to the rest in the same orientation
    (not inversionTookPlace[t]) ->
    (
      %end left of target
      isLeftNeighbour(t+1, operationEnd[t], operationTarget[t]) /\

      %(left of target) is now left of (start).
      %Only bother if moving
      (operationIsMoving(t) ->
        (isLeftNeighbour(t+1,
                 leftNeighbour[t,operationTarget[t]],
                 operationStart[t]))) /\

      %(left of start) is now left of (right of end).
      %Only bother if moving
      (operationIsMoving(t) ->
        (isLeftNeighbour(t+1,
                 leftNeighbour[t,operationStart[t]],
                 rightNeighbour[t,operationEnd[t]])))
    )
  ) /\
  (
    %The case where an inversion did take place
    %(link the end in the start's position, and vice versa.
    %Caps in comments indicate changes from above
    inversionTookPlace[t] ->
    (
      %START left of target
      isLeftNeighbour(t+1, operationStart[t], operationTarget[t]) /\

      %(left of target) is now left of (END)
      (operationIsMoving(t) ->
        (isLeftNeighbour(t+1,
                 leftNeighbour[t,operationTarget[t]],
                 operationEnd[t]))) /\

      %(left of start) is now left of (right of end)
      (operationIsMoving(t) ->
        (isLeftNeighbour(t+1,
                 leftNeighbour[t,operationStart[t]],
                 rightNeighbour[t,operationEnd[t]])))
    )
  )
);
```

Listing 5: A transposition operation for the relational model (also accounts for possible inversions).

Relational model
File name: `genome-relational.mzn`

```
%The predicate we use to determine whether the gene is inside the block
predicate partOfInversion(var int: t, var int: g)
  = geneSign[t,g] != geneSign[t+1,g];

%First, we infer which blocks should be negated based on their signs
%We say that genes to the left and right of genes that are part of
%inversions become part of inversions (chain inference to the middle)
constraint forall(t in stepIndices) (
  inversionTookPlace[t] -> (
    forall(g in geneValues) (
      (
        (partOfInversion(t,g) /\ (not (g=operationStart[t]))) ->
          (partOfInversion(t,leftNeighbour[t,g]))
      ) /\

      (
        (partOfInversion(t,g) /\ (not (g=operationEnd[t]))) ->
          (partOfInversion(t,rightNeighbour[t,g]))
      )
    )
  )
);

%If an inversion took place, force it to change the sign of the start
%and the end, so we can chain inference into the middle (base case)
constraint forall(t in stepIndices) (
  inversionTookPlace[t] ->
    (
      partOfInversion(t,operationStart[t]) /\
      partOfInversion(t,operationEnd[t])
    )
);
```

Listing 6: Constraints to perform chain inference on the gene signs, and determine whether a gene `g` is involved in an inversion operation at time step `s`. The produced predicate is `partofInversion(t,g)`

Chapter 6

Polynomial time distance measures

In Chapter 4 and Chapter 5 we discussed the use of constraint satisfaction to solve the genome edit distance problem described in § 1.2, using the MiniZinc constraint modelling language. This method solves the problem (calculating minimum cost transformation plans for inversion, transposition, transversion operations with arbitrary weighting) optimally. However, it is conjectured that sorting by transpositions alone is NP-Hard [13], and the constraint satisfaction approach implemented in Chapter 4 is certainly not a polynomial time algorithm.

We look at other distance measures for two reasons. Firstly, if we can calculate a good upper bound on cost, and add this to the constraint satisfaction model, we can improve search time with pruning. Secondly, if we can compare the distances given by the optimal constraint satisfaction model with those given by polynomial time approximations on various genomes, we can attribute some level of accuracy to each of these approximation methods. We can then decide whether it is worthwhile to compute the optimal distance with inversions, transpositions and transversions.

6.1 Breakpoint distance

6.1.1 Calculating the breakpoint distance

Breakpoint distances can be calculated in linear time [20]. To calculate this distance, we require two genomes – call these genome A and genome B. A *breakpoint* is a pair of consecutive genes (g_1, g_2) in genome A that are not consecutive in genome B ("Consecutive" depends on the sign of the gene: g_1, g_2 are consecutive, and $-g_2, -g_1$ are consecutive). The number of these breakpoints is the breakpoint distance.

If we are calculating the breakpoint distance to the identity permutation, we can decide whether a pair of adjacent genomes are a breakpoint easily. Adjacent genes can be identified as breakpoints if $|g_1 - g_2| > 1$.

This algorithm is described in more detail by Kaplan et al. [12] and Blanchette et al. [20]. An example of a breakpoint distance calculation can be seen in Figure 6.1.

Genome A	-3	-2	-1	4	5	8	6	7
Genome B	1	2	3	4	5	6	7	8

Figure 6.1: An example of breakpoint distance between two genomes A and B. The vertical lines in Genome A represent breakpoints, and the breakpoint distance between A and B is 4.

See § 7.2 for a detailed analysis of calculated breakpoint distances, and how they compare with other edit distance models.

6.2 Inversion Only distance algorithm

In this section, we provide a very brief overview of the algorithm used to sort signed sequences by inversions. This model is referred to as the Inversion Only (IO) model. A naïve approach is described, and an intuition for the more efficient $O(n^2)$ algorithm is given. It is not the goal of this section to provide a comprehensive description of the algorithm, – for a detailed description of the algorithm, see Kaplan et al. [12].

6.2.1 Sorting by reversals algorithm

The naïve algorithm

One approach, presented by Bergeron [23], describes an algorithm to calculate inversion distances by finding *oriented pairs*, and choosing the inversion with the *maximal score*.

An *oriented pair* is a pair of consecutive integers, that is, (π_i, π_j) such that $|\pi_i| - |\pi_j| = +1$, and π_i and π_j have opposite signs. For example, consider sorting the sequence S to the identity,

$$S = (0, 3, 4, -2, 1, 5) \tag{6.1}$$

Equation 6.1 shows a sequence S with oriented pairs $(3, -2)$ and $(-2, 1)$. Oriented pairs correspond to inversions that help sort the sequence. The pair $(3, -2)$ tells us to perform the inversion on $\pi_2 = 3$, $\pi_4 = 4$ and the pair $(-2, 1)$ tells us to invert $\pi_5 = 1$. Doing either of these increases the number of consecutive integers, and so helps us sort the sequence. We choose the operation that leaves the most oriented pairs, execute it, and then repeat the procedure.

Doing so leaves a positive sequence with no more oriented pairs. For the sequence S of Equation 6.1, this sorts the sequence to the identity, but for others, we may be left with an unsorted positive sequence. Remaining operations aim to eliminate *hurdles*, which are intervals $[s, e]$ of the sequence S containing all integers between s and e

(not necessarily in order), and no sub-sequences of the interval with this property. Inversions can be selected that break these hurdles, which can then be reduced using the process described above.

For more details on this algorithm, see Hannenhalli and Pevzner [24] and Bergeron [23].

The $O(n^2)$ approach

The algorithm described by Hannenhalli and Pevzner [24] is complete, but we can use a more efficient $O(n^2)$ approach. This is the approach that was taken for calculating IO distances in this paper, and it was initially introduced by Kaplan et al. [12]. It is based on the same principle of reducing breakpoints and eliminating hurdles, however, it does this more efficiently with the use of a *interval overlap graph*, and it counts the cycles and breakpoints in with this graph, and uses this to derive the required number of inversions [12]. This algorithm has been implemented for signed sequences by Braga [25].

Chapter 7

Model comparisons

In this chapter, the raw distance results of the methods described previously are examined – the breakpoint edit distance, the IO edit distance, and the ITT edit distance. It is found that there is some correlation between the distance results, but the degree varies depending on the type of genome.

7.1 Inversion Only vs. Inversion/Transposition/Transversion

7.1.1 Distance comparison

Experiment aims and methodology

The IO model of evolution allows a much faster distance calculations than the full ITT model described in § 1.2. It would be interesting to know how well the IO model approximates the full model. To see what relationship the IO distance has with the ITT distance, both algorithms were run on the same datasets, and distance results were collected.

Genome datasets Several datasets were created for the comparison of these two distance measures. These datasets were synthetically created, and are designed to provide a wide range of data, to prevent a bias towards any one model. The following synthetic datasets were created for experimentation

- Completely random genomes
- Genomes created from random inversions
- Genomes created from random transpositions
- Genomes created from random transversions
- Genomes created from random inversions and transpositions
- Genomes created from random inversions, transpositions and transversions

More information on the creation of the data sets, and other datasets that were used for experimentation can be found in § B.1 (this section also includes information about real world biological data used for experimentation).

Pairs of genomes were created using each of these generation techniques, and were sorted back to the identity genome.

Due to the high running time of the ITT minizinc solver for long genomes, and to maximize the number of genomes tests could be run on, shorter genomes were used. The Short lengths dataset and the All lengths datasets were used for this experiment (Short lengths contains many short genomes, All lengths contains a fewer genomes of longer length. See Table B.2 for details).

Results

Semi-random genomes The ITT and IO algorithms were both run on the Short lengths and All lengths datasets. A visualisation of the result can be seen in Figure 7.1. The first thing to note from this graph is that the IO distance is always equal to or greater than the ITT cost. If this were not the case, it would indicate that our algorithm for solving with ITT was non-optimal, so the result is encouraging.

Further, we can see that a substantial amount of ITT and IO costs are equal. When there are inequalities, the IO distance cost is never more than 1 higher than the ITT cost for this dataset.

Another comparison of results can be seen in Figure 7.2. The first bars represent ITT and the second represent IO average cost, for various genome lengths. We can again see that the difference in plan costs for the IO model and the ITT model are small. However, at most, the average plan cost difference is less than 1. We can also see the error bars, representing one standard deviation, are approximately 1-2 for each genome length – often wider than difference between the average plan costs.

It was also found that the IO model produced different results depending on the type of semi-random generation method used. Figure 7.4b compares the IO, ITT and breakpoint distances over the Short lengths dataset, and we find that the IO model produces results that are closer when inversions are involved (inversion and transversion generation methods). The ITT model produces substantially smaller costs for the "Inversion and transposition" mode, and the "transposition" mode, indicating that the difference between the two models increases if we increase the proportion of transposition operations.

From these experiments, we can see that there is some difference between the IO distance and the ITT distance in terms of raw results, for genomes generated synthetically with these methods. However, the difference is small as a proportion of the lengths of the genomes.

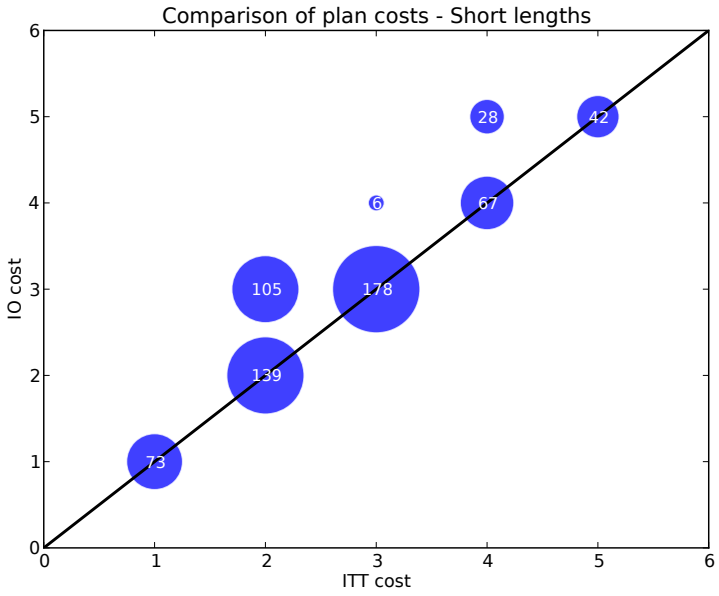


Figure 7.1: A comparison of the plan costs for all operations (the minizinc implementation) and the IO distance. Bubble size correlates to the number of generated genomes with the associated costs. The Short lengths dataset was used.

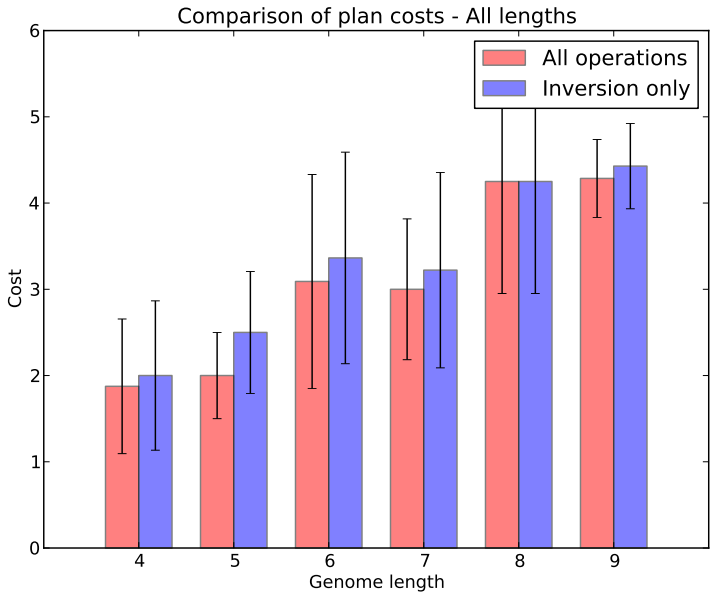


Figure 7.2: A comparison of genome length with average cost, for all operations and IO operations. Inversion / Transposition / Transversion costs are on the left, and Inversion Only costs are on the right. The All lengths dataset was used.

Length	ITTw	ITTuw	IO	ITTw-ITTuw	IO-ITTw	IO-ITTuw
4	0.533	0.450	0.541	0.083	0.008	0.092
5	0.633	0.448	0.640	0.185	0.007	0.192
6	0.700	0.452	0.700	0.248	0.000	0.248
7	0.691	0.455	0.703	0.236	0.011	0.248
8	0.643	0.467	0.643	0.176	0.000	0.176
All lengths	0.640	0.470	0.645	0.187	0.005	0.191

Table 7.1: The normalised average costs for the different models. Results were normalised by dividing total average cost as calculated by the model, and dividing by the length of the genome. Columns ITTw, ITTuw, IO are the normalised average costs for each method, and the final three columns are the difference between these values. Calculations were performed on the Random dataset (see Table B.2).

Completely random genomes Looking at the plan costs for semi-random genomes gives us an idea of how our results compute distances for realistic genomes (ones formed synthetically with a few random inversions / transpositions) but the selection of methods and mutation parameters is somewhat arbitrary. To prevent this bias, we can compare distance results between completely random genomes for different models (genomes generated by assigning random signs and positions to each gene). This will tell us how the algorithms perform on completely random genomes – giving an insight into how the algorithms measure distance for genomes as a function of length. However it will not necessarily produce realistic distance results – genomes are usually the product of a small number of mutations rather than completely random sequences [4].

From Table 7.1, we can see the results after calculating plan costs with three different models on the Random dataset. Results are calculated for the weighted ITT (ITTw) model and IO models, which have been introduced, and the ITTuw model, which is an unweighted version of the ITTw model – all operations are given a cost of 1. The results show the average normalised average plan cost for genome lengths 4 to 8.

The fact that ITTuw costs < ITTw costs confirms that the transposition operation is useful in sorting the genomes, as lowering the penalty for the operation lowers the cost of the results given by the model. However, comparing ITTw with IO, we see that there is minimal difference in the produced plan costs when we use the transposition weighting of 2, suggested by Blanchette et al. [21]. On average, the normalised plan cost difference is around 1%. This is consistent for all genome lengths.

The results for completely random genomes are similar to the ones for semi-random genomes. It appears as though for completely random genomes, the difference between the IO model and the ITTw model are even smaller.

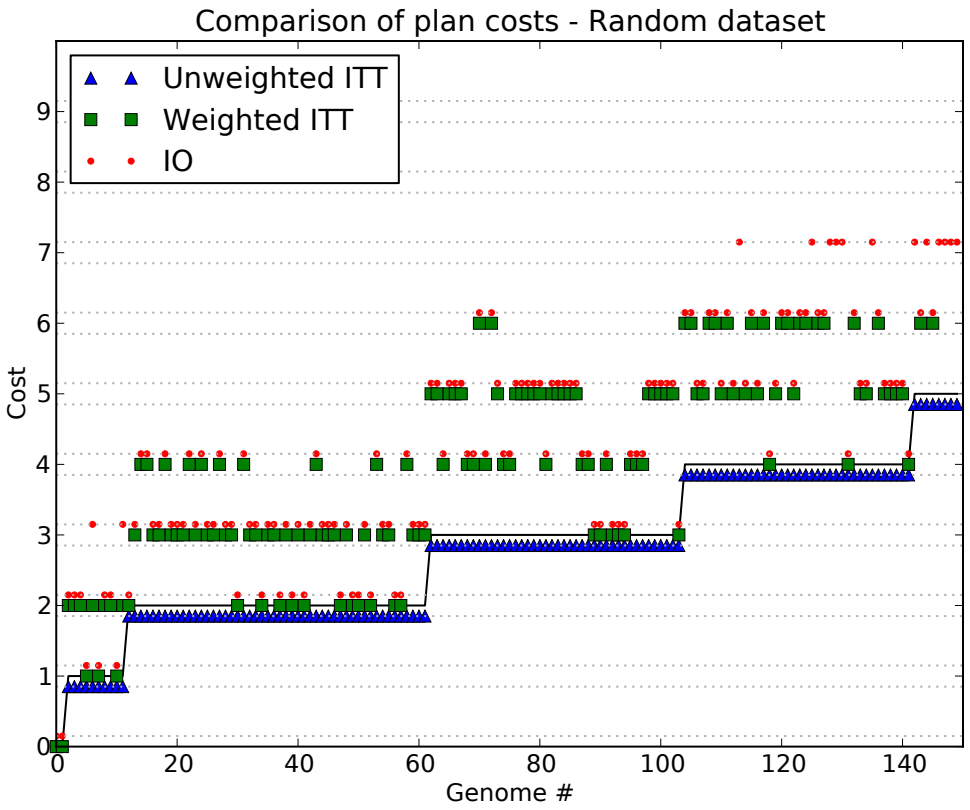


Figure 7.3: A comparison of the plan costs for the ITT model (unweighted and weighted) and the IO model. Note that symbols are shifted vertically for visibility – each model produces integer weights. Random problem instances are sorted on the x axis by unweighted ITT cost.

Another visualisation of this data can be seen in Figure 7.3. Rather than averaging and tabulating the results, each genome's cost was plotted using the IO, ITTuw and ITTw models. Genomes are sorted by their ITTuw costs.

We can see that ITTuw costs are always the lowest, usually by 1-2 (savings of one or two inversions, or a transposition / transversion). However, if we use the ITTw model, we stop using these plan length reducing operations, and tend to use longer plans that use inversions instead (the ITTuw model is essentially a plan length minimising model).

7.2 Breakpoint distance analysis

We now introduce breakpoint distance to the comparison. Figure 7.4 displays the breakpoint distance, compared to the IO and the ITT distances for the Short lengths dataset. In Figure 7.4a we can see a pattern that is similar to the one from Figure 7.2 – that is, there is a marginal difference between IO and ITT distances for each genome length, and this distance stays mostly constant as we vary the length of the initial and goal genome. The breakpoint distance is similarly larger than the IO and the ITT distances, by a cost of approximately 1 for each length. Note that this difference is not necessarily indicative of the breakpoint distance being a poor approximation to the IO or ITT distances, as we can only examine the relative increase and decrease in costs (breakpoint distance does not correspond to the weighted cost of a plan, while IO and ITT are directly comparable). Given that the difference is constant for all lengths, Figure 7.4a shows that genome length does not differentiate between the breakpoint and IO or ITT distances.

If we look at Figure 7.4b however, we can see that there *are* relative differences between the distances, if we compare by *genome generation type*. The x axis displays a number of semi-random synthetic genome creation methods, and we can see that the difference between the breakpoint distance and the IO / ITT distances varies considerably depending on generation method.

The first interesting aspect of Figure 7.4b is that breakpoint distance is almost equivalent to IO distance if we use just transpositions for generating our genomes. Since every transposition operation can be simulated by three IO operations, a transposition in the genome will usually increase plan length by about three (assuming the operations are independent). Independent transpositions will also increase the breakpoint distance by three, so the similarity between the distances for transpositions makes sense. It also makes sense that ITT distances are two-thirds that of IO and breakpoint, as a transposition operation in ITT can be performed with cost 2 rather than cost 3 [1].

An interesting side note is also the inversion data from Figure 7.4b. The ITT and IO distances are equal here, meaning that for this dataset, no random inversions created a transposition operation. This may indicate that random inversions leading to transpositions are rare, however, more trials would be needed to verify this claim.

7.2.1 Breakpoint distance applications and usefulness

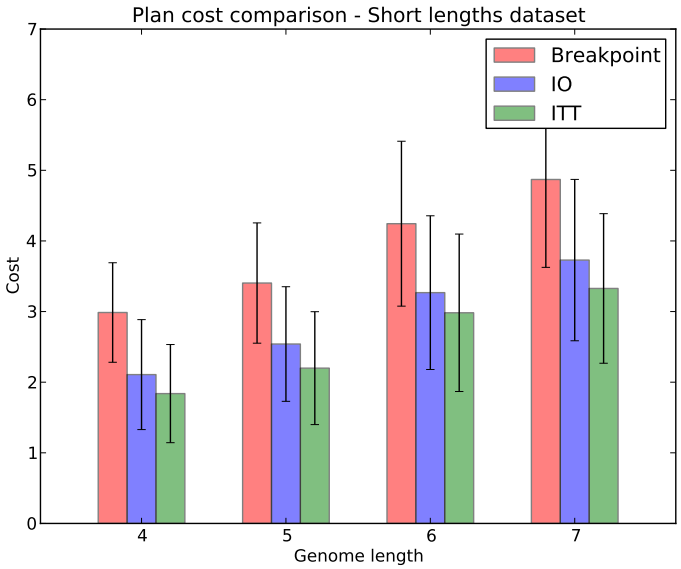
We can see from Figure 7.4 that there is a difference between breakpoint distance and IO / ITT distances, as expected, but critically, this distance is not uniform or predictable, as it varies depending on the operation chosen to mutate the genomes.

[1] See § E.1 for an explanation of how

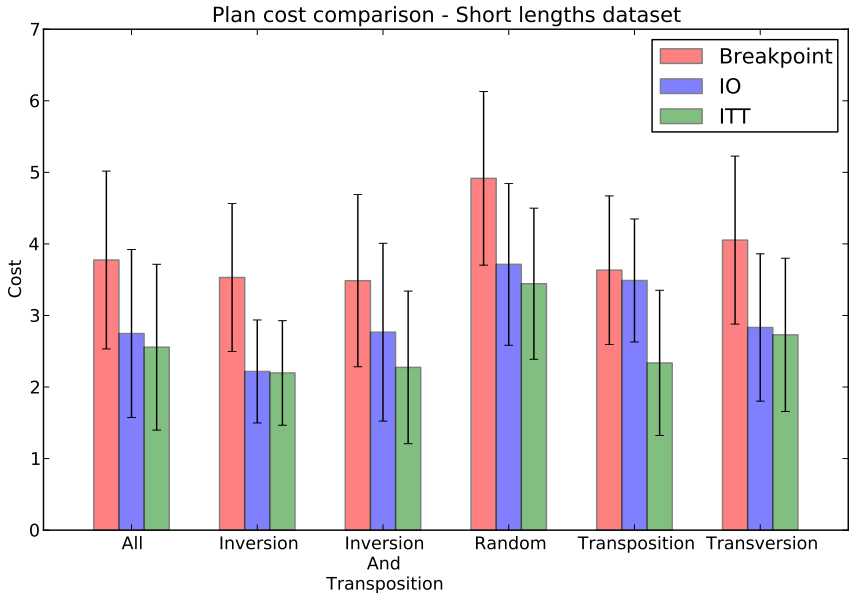
This may mean that breakpoint distance is a poor approximation to both models, but to quantify whether the distance is significant, a tree construction experiment must be performed. This experiment with the trees constructed from breakpoint distances can be found in Chapter 9

7.3 Usefulness of raw distance comparison

Raw distance comparison, as performed in this section, can tell us some things about the solving algorithms – which algorithms produce the best costs, how close the IO model costs are to the ITT model costs, and how the costs of these models relate to the length of the genome they are computed on. However, it is not certain whether this difference is substantial enough to warrant the increased solving time. To know this, we must look at the primary use of these distance methods – phylogenetic tree construction. This comparison can be found in Chapter 9.



(a) Comparison of the breakpoint costs with the ITT and IO costs, by length



(b) Comparison of the breakpoint costs with the ITT and IO costs, by genome generation type. Generation methods are listed in the order (All, Inversion, Inversion and Transposition, Random, Transposition, Transversion)

Figure 7.4

Chapter 8

Compression of genomes

As there is a significant time cost associated with some models of the genome edit distance problem, it is worth saving time any way that we can. One way that time can be saved is through the use of *compression*. Compression takes two genomes, finds matching subsequences, and ideally, produces shorter genomes for us to calculate edit distances with. We would like to quantify how useful this addition is to the process, both for synthetic datasets, and for biological data.

8.1 Description of compression algorithm

The algorithm used to compress the sequences is simple, so it will only be explained briefly. Any compression algorithm may be used on the genomes, remembering that genomes are modular, and so a sequence spanning the boundary of a one dimensional array can be identified as present in another genome, and compressed. We must also remember that it is possible to compress segments of a genome that have been inverted. Algorithm algorithm 1 contains the pseudo-code for this function.

The algorithm finds pairs of genes that are adjacent in both genomes, and compresses them. This is continued until no pairs of genes can be found adjacent in both genomes. This will compress the genomes as efficiently as possible, as compressing multiple pairs of genes is equivalent to compressing a large block of genes. It is not the most efficient implementation of the compression algorithm, but it is a simple way of dealing with issues like compression wrapping around the border of the genome, reversed gene segments, and finding the largest block of adjacent genes to compress.

8.2 Compression of synthetic genomes

We determined the average compressed genome length for the Compression stats dataset, for each genome length in the range [4,13], and for each generation mode (see Table B.1 for a description of these modes). Compression was calculated between

Data: A, B: One dimensional arrays of genes to compress
Result: A_c, B_c: One dimensional arrays of genes that have been compressed
$A_c \leftarrow A$;
$B_c \leftarrow B$;
while *we made a change last iteration, or it is the first iteration* **do**
  for $i \in 0..|A_c|$ **do**
    iNext = (i+1)%$|A_c|$;
    if $A_c[i]$ *and* $A_c[iNext]$ *are adjacent in* B_c **then**
      iLocationB $\leftarrow$ location of |i| in B_c;
      iNextLocationB $\leftarrow$ location of |iNext| in B_c;
      if $A_c[i]$ *and* $A_c[iNext]$ *appear reversed in* B_c **then**
        sign $\leftarrow -1$;
      else
        sign $\leftarrow 1$;
      end
      newGene $\leftarrow$max(abs(A_c[i], A_c[iNext]);
      Delete A_c[iNext];
      Delete B_c[iNextLocationB];
      $A_c[i] \leftarrow$newGene;
      B_c[iLocationB] $\leftarrow$sign $\times$ newGene;
    end
  end
end

Algorithm 1: Genome compression algorithm. The algorithm finds pairs of genomes that match in A and B, and merges them into a single unit. This process is repeated in the `while` loop.

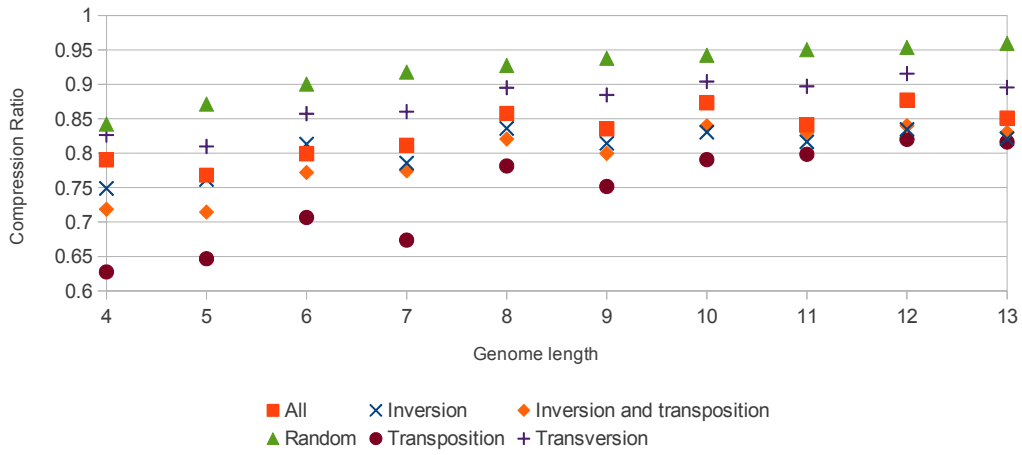


Figure 8.1: The average compression ratio for each generation mode, by genome length.

each pair of genomes in the same category (genomes generated with the same length and mode).

The result of this experiment can be found in Figure 8.1. The graph shows average compression ratios (compressed length / uncompressed length) compared with genome lengths. Firstly, we can see that all compression ratios are between 0.6 and 1. We can also see that as genome length increases, there is an overall increase in compression ratio, for all generation modes.

While all generation modes trend upwards, there is a noticeable difference between them, and we can see there is an ordering of the generation modes, ranked by the difficulty of compressing genomes created with these methods. According to the graph, the ranking is: Random > Transversion > All > Inversion > Inversion and transposition > Transposition.

It makes sense that the most difficult mode for compression is completely random genes – these would have the least structure, and for longer genomes (such as length 13 in this example) it becomes very unlikely that genes will be adjacent in any given pair of genomes. After this, we can see that transversion is more difficult to compress than inversion, and inversion is more difficult to compress than transposition. The combination modes, "All" and "Inversion and transposition", lie between their component mode's ratios.

8.3 Compression of real genomes

Similar compression experiments were conducted on the Biology mitochondria dataset. This dataset was taken from the NCBI database (See § B.1). The dataset includes 3,634 mitochondrial genomes, from a wide range of different species. A sample of 500 ran-

dom genomes was taken from this dataset, and pairs of genomes were selected for compression ($\frac{500^2}{2} = 125,000$ compressions).

The results of this experiment gave the average compression ratio for pairs, and the average length of compressed genomes (uncompressed length for genomes were usually 37). These were found to be:

Average compression ratio: 0.4911 ± 0.3270
Average length: 17.0297 ± 11.2350

This tells us that a significant amount of compression is possible. However, we get a better idea than raw values if we look at the distribution. Figure 8.2 shows us that the distribution is not uniform. While the average compressed genome length may be 17.0297, we can see that there are distinct peaks at lengths 4 and 19 (these correspond to the peaks at ratio 0.1 and 0.5). Besides this, a large number fall into the length range 33-35. This tells us that our dataset has three mostly distinct categories – genomes that are almost identical, apart from a few genes (compressed length of around 4), genomes that have different structures, but are somewhat compressible (compressed length of around 19), and genomes that are almost entirely different in structure. We can see that there are virtually no genomes with compressed lengths in the range 11-16.

Note that this structure is not due to gene content differences. Only genomes with equal gene content were compared.

The genomes in the first peak, of compressed length 3-11, are in the range of the capabilities of the algorithms discussed in this thesis (ITT model with constraint satisfaction, and heuristic search), while the genomes in the second and third groups are only be comparable with simplified models (IO, breakpoint analysis).

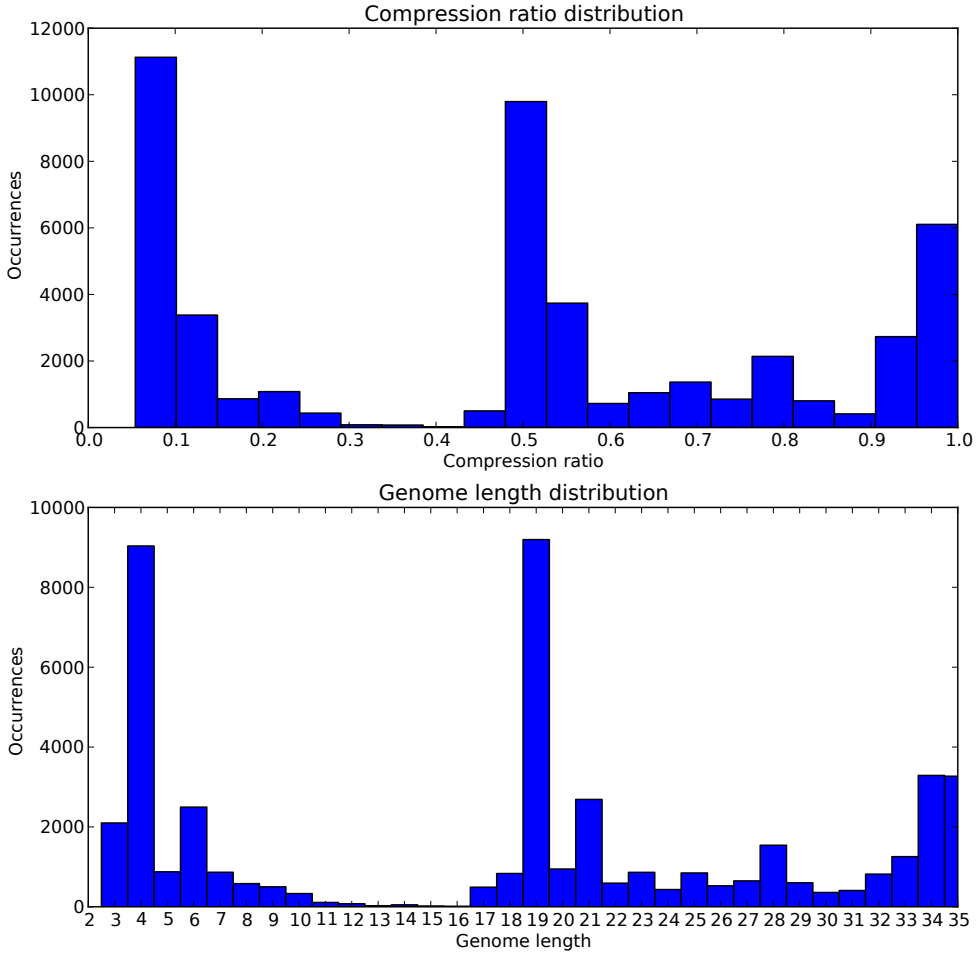


Figure 8.2: Compression for the Biology mitochondria dataset. Graph based on a random sample of 500 genes from the dataset. Categories with less than 5 occurrences are hidden. The top graph represents the compression ratio distribution, and the bottom graph displays the compressed genome length distribution.

Chapter 9

Tree construction

In Chapter 6 we looked at alternative methods to ITT for computing the distance between two genomes, and compared them in Chapter 7. We have seen that the distances do correlate to a degree (on the order of 10% average plan cost difference between IO and ITT), but there are differences in some cases (depending on the operations used to mutate the genome). We cannot tell from raw distances whether the difference is significant enough to warrant the extended computation time of using constraint satisfaction and the ITT model. Therefore, we look at one of the primary uses of these distances – tree construction (as mentioned in § 1.1.1), to see if the difference in distance dramatically alters the structure of the trees.

9.1 An overview of tree construction methods

9.1.1 Neighbour joining overview

The Neighbour Joining method of tree construction [26] was designed for efficient creation of phylogenetic trees from distance data. For our purposes, the input to the algorithm is a distance matrix of similarity between genomes, and the output is an approximation of the tree that minimizes total branch length between the nodes.

First, we define the term *neighbour*. A pair of neighbours is "a pair of operational taxonomic units connected through a single interior node in an unrooted, bifurcating tree" [26]. Here, an "operational taxonomic unit" (abbreviated OTU) is a node in the tree, or a group of nodes (See Figure 9.1).

The algorithm works by iteratively combining nodes of the tree. The initial tree is an unrooted tree, with all nodes joined to one interior node (See Figure 9.2). The algorithm seeks to minimise total sum of branch lengths, and so designates two nodes as neighbours if combining them into one operational taxonomic unit reduces the total sum of branch lengths by the largest amount.

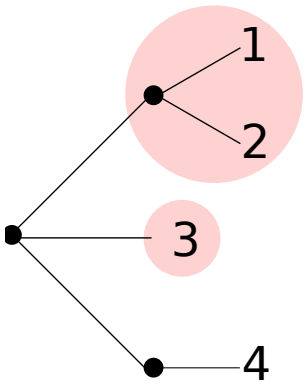


Figure 9.1: An example of an operational taxonomic units in a tree. Black circles indicate interior nodes within this tree, and an operational taxonomic unit is any node, or group of nodes, within the tree. In this example, two possible OTU's are shaded – the unit containing 1,2, and the unit containing just 3. Further, the shaded OTU's are neighbours in this example, as they are connected by only a single interior node. While 4 is an OTU by itself, it is not neighbours with 3, as there are two interior nodes separating them.

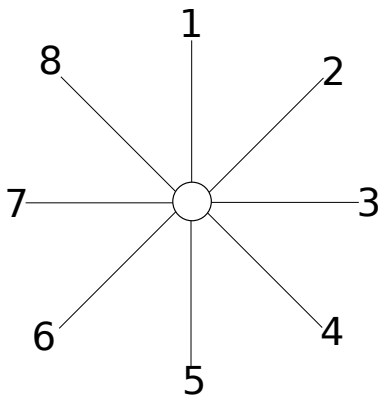


Figure 9.2: An illustration of the initial tree configuration of the neighbour joining algorithm

Neighbour joining discussion

While it is not always possible to know which nodes are truly neighbours in the tree [26], the algorithm can produce a reasonable approximation using this method. This algorithm also does not necessarily produce the minimum evolution, or maximum parsimony, tree. A minimum evolution tree is one that assumes the least amount of evolutionary deviation from one genome to another. The algorithm can produce the correct tree for purely additive trees, but may not for models that allow backward and parallel substitution (such as horizontal gene transfer). [26]

However, it should be noted that the minimum evolution tree may not be the true tree anyway. The minimum evolution tree "often has an erroneous topology, and the maximum-parsimony method of tree making is not always the best in recovering the true topology." [26]

Comparisons with other algorithms (UPGMA, DW, ST, LI, MF) by Saitou and Nei [26] through computer simulations indicate that this method is close to equivalent for practical purposes, and since it operates in polynomial time, is a good choice for testing phylogenetic tree construction.

9.2 Tree construction comparison with synthetic data

Edit distances were calculated using the ITT, IO and Breakpoint models on the Tree synthetic 6 dataset (A dataset of random genomes of length 6. See Table B.2). Edit distances were calculated for each pair of genomes, and a distance matrix was computed for each model (ITT, IO, breakpoint) on the same data. These distance matrices were input into an implementation of the Neighbour Joining algorithm [27, 28], and were visualised by the *iTOL* software package [29, 30].

Figure 9.3 shows the three trees produced by this algorithm. Displayed branch lengths are proportional to the actual branch lengths the Neighbour joining algorithm produces.

9.2.1 Discussion of the synthetic tree construction comparison

IO vs. ITT We can see that for this synthetic data (in this case, completely random genomes), there is a great deal of similarity between the two models. At the lower levels of the tree, the genomes are structured identically. We must look higher in the tree to see any difference – the ITT model groups the (7,4) sub-tree as neighbours of the (9,1,8,6,3) sub-tree, while the IO model assigns distinct groups. Additionally, the IO model groups (10,2,5) as a direct neighbour of (11,12), but the ITT model assigns an internal node as the parent of both sub-trees.

This corresponds with our raw distance results from § 7.1.1. We see that the small difference in raw distances (on the order of 1%) does have a small impact

on the resulting tree for raw genomes. It should be emphasised that this does not necessitate a small difference for *all* genomes, as this experiment was performed with purely randomly generated synthetic data. With random data, we would not expect distances to yield any real tree structure anyway. This may be the reason for the small difference between constructed trees from the ITT and IO computed distances (for the same experiment with biological data, see § 9.3).

Breakpoint vs. ITT Again, we can see much similarity in the trees produced – the breakpoint model groups (10,2,5) and (11,12), and it's upper level hierarchy is identical to that of the ITT model for this sub tree (as opposed to the IO model, which excluded an internal node). (9,1) and (7,4) are neighbours in the breakpoint tree, but these two groups are not direct neighbours in the ITT model's tree. The group (8,1,6,3) contains the same structure in both trees.

Impact of raw distances on phylogenetic tree construction

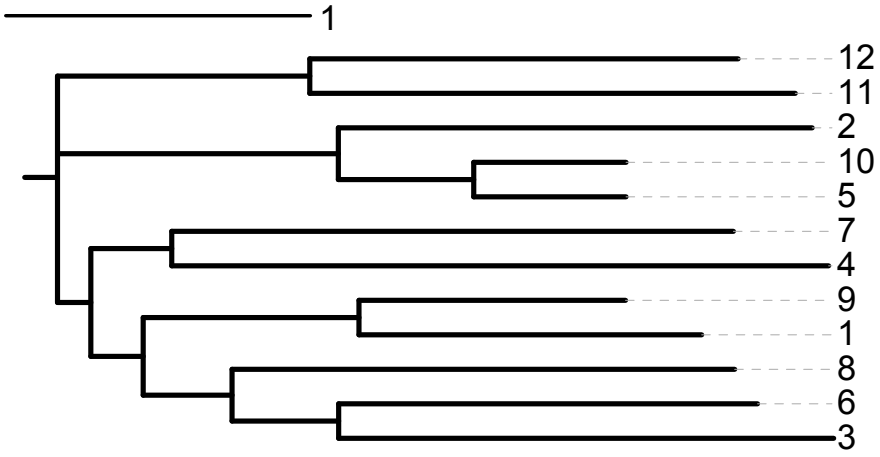
From this, we can see that there is some difference between the trees produced from the distance measurements for randomly generated synthetic genomes. It is difficult to tell which of the IO or breakpoint distance tree are closest to the ITT tree without some metric for tree comparison, but it is enough to know that there is a structural difference.

This means that we cannot ignore the minimal difference in raw results from Chapter 7. Since tree construction must create discrete categories for genomes (ignoring the variable tree lengths displayed in the diagrams), a minimal difference in raw distance may lead to a different categorisation. For these trees, the difference in categorisation is also minimal – the structure of the trees are fundamentally the same. We cannot definitively say that this difference is negligible though, as the difference may be important in some biological applications (but it could easily be unimportant).

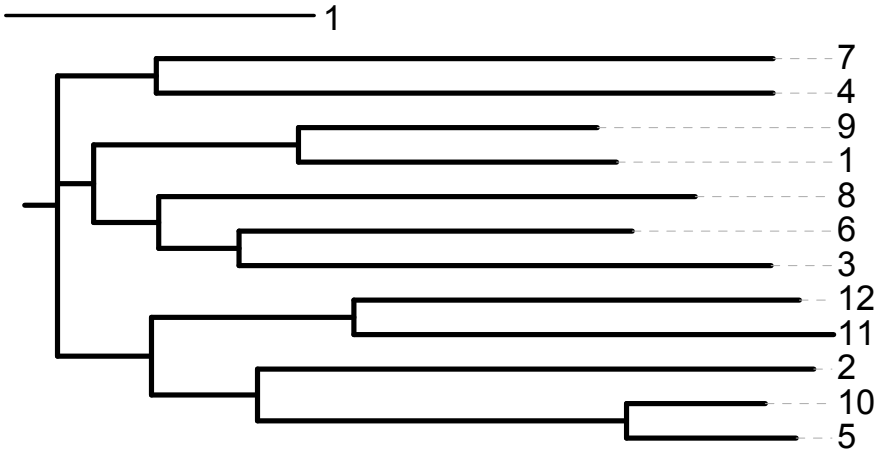
Neither can we judge which model produces trees closer to the biological reality based on this experiment, as the genome data has been artificially generated. The same experiment, with biological data, can be found in § 9.3.

9.3 Tree construction comparison with biological data

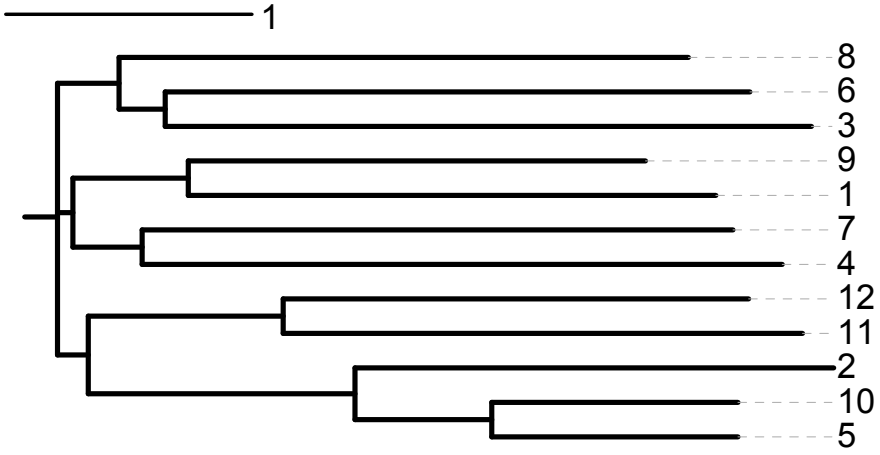
As in § 9.2, edit distances using the ITT, IO and breakpoint models were calculated for the same set of data. In this case, the Biology mitochondria dataset was used. This dataset contains 11 genomes of mitochondrial genomes, randomly selected to have pairwise compressed lengths of 5-7 (from 37). All genomes in this subset have equal gene content. More information on the dataset can be found in § B.2.



(a) Inversion Only phylogenetic tree.



(b) Inversion / Transversion / Transposition phylogenetic tree.



(c) Breakpoint distance phylogenetic tree.

Figure 9.3: Phylogenetic trees generated by the Neighbour Joining algorithm, from synthetic data. Line lengths represent branch length, scale present in the top left hand corner. Data taken from the Tree synthetic 6 database.

Genome number	Accession number	Scientific name	Description
1	NC 005826	Dromiciops gliroides	Monito del monte (Bush monkey, marsupial)
2	NC 013606	Coloconger cadenati	Congro (Short-tail eel)
3	NC 020586	Tragopan temminckii	Temminck's Tragopan (Pheasant)
4	NC 016119	Nanorana pleskei	Tibetan frog (Frog)
5	NC 017606	Dendrophysa russelii	Goatee croaker (Ray-finned fish)
6	NC 006082	Chinemys reevesi	Reeves's turtle (Turtle)
7	NC 004700	Chalceus macrolepidotus	Pinktail chalceus (Tropical fish)
8	NC 011191	Halichoeres tenuispinis	Chinese wrasse (Fish)

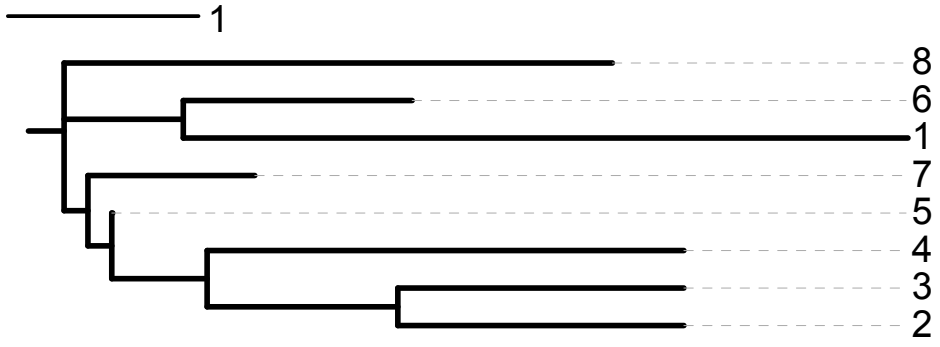
Table 9.1: The names of animals for the biological genome tree construction comparison (Figure 9.4)

The selection of animals for this experiment may be of use in comparing the validity of each method biologically, and is of general interest, so they are listed in Table 9.1.

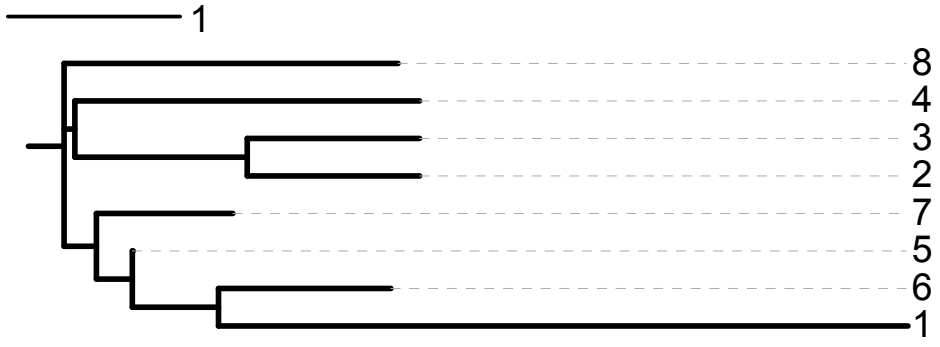
IO vs. ITT The most important observation from the generated trees is that they are similar, but not exactly equivalent. The ITT model groups (7,5,6,1), but the IO model splits (7,5) and (6,1) into separate groups. There are some similarities though – (4,3,2) has the same structure in both trees, and (8) is given its own top level branch in both trees. Additionally, in both, we can see that genome (1)'s distance from all others is very high (given that (1) is a marsupial, and most others are aquatic animals, this is probably accurate).

Breakpoint vs. ITT Again, there are some similarities between the breakpoint edit distance and the ITT tree, but there is a substantial difference between the structures. (2,3) and (1,6) are still grouped in the breakpoint tree, but these are the only significant similarities – the breakpoint tree has little internal structure, and lists many genomes as children of the root internal node. It appears as though the IO tree is a better approximation to the ITT tree (although this judgement is only visual).

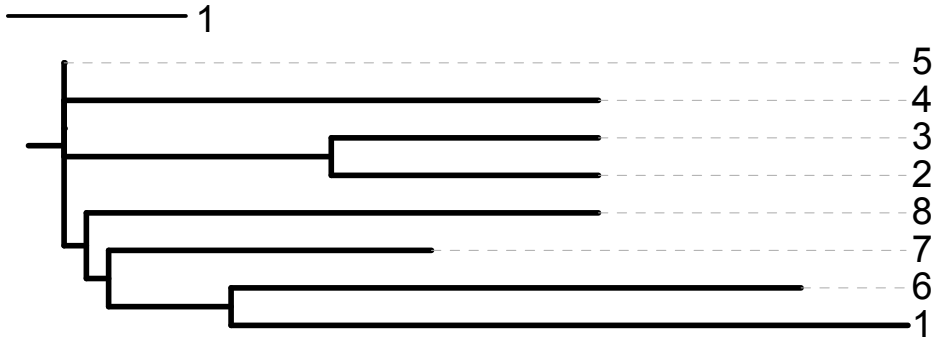
Note that most animals selected for these phylogenetic trees are from similar



(a) Inversion Only phylogenetic tree



(b) Inversion / Transversion / Transposition phylogenetic tree.



(c) Breakpoint distance phylogenetic tree.

Figure 9.4: Phylogenetic trees generated by the Neighbour Joining algorithm, from biological data. Line lengths represent branch length, scale present in the top left corner. Data for each genome is taken from the Biology mitochondria dataset.

groups. They were randomly chosen to satisfy the following properties:

1. Gene content was sufficiently similar to not require dropping many genes (at most 4 genes were dropped between comparisons in this set)

2. It was also required that compressed genome was small enough for calculation of ITT distances – this naturally forces the set to be similar animals (in this example, aquatic animals).

From this comparison, we can see that the IO and ITT model's do produce different phylogenetic trees, meaning that the small and rare differences between ITT and IO raw edit distances is significantly high for tree construction on biological data to produce different results.

Chapter 10

Conclusion

10.1 Summary of results

The goal of investigating the use of constraint satisfaction for the genome edit distance problem has been achieved. It has been found that constraint satisfaction is able to calculate distances for the Inversion / Transposition / Transversion model of genome evolution, however, it is not yet competitive with previously attempted approaches, such as planning and heuristic search.

Two MiniZinc constraint satisfaction models have been implemented – a relational model, and a positional model. The positional encoding may seem more intuitive due to the ability to simply express inversion operations, but this simplicity is complicated by edge cases, and operations that span the boundaries of the linear array. Additionally, with sufficient symmetry breaking constraints, it was found that the relational MiniZinc model out-performs the positional MiniZinc model.

Given the symmetric nature of this problem, and the benefits that the symmetry reducing constraints introduced in this paper give, we conclude that there is potential to expand the range of problems constraint satisfaction can solve in reasonable time through additional symmetry reducing constraints.

The application of this algorithm to real world problems was tested against the NCBI mitochondrial genome database, and it was found that compression algorithms are often useful for real world data. It was found that some genomes can be compressed by up to 90% of their original size, and most can be compressed by 50% or more. There exist some genome pairs that cannot be compressed at all. While the compression algorithm reduces genome length, and consequently solving time substantially, compression alone is not enough to bring all biological genome comparisons into the range of tractability for an ITT model.

Alternative models of genome evolution with polynomial time solutions were trialled, including inversion-only distance, and breakpoint distance. It was found that while the distance results are well correlated, and very similar in most cases, when these alternative distances are used for phylogenetic tree construction, the difference becomes non-trivial.

10.2 Future work

Performance improvements

It is believed that there is potential to expand the reach of constraint satisfaction for this problem through the use of stronger redundant constraints, and more symmetry breaking constraints.

Performance benefits may also be achieved by linking the two encodings of the problem into a dual-model [22].

Stronger upper and lower bounds on plan cost, stronger than the IO distance, may also improve constraint satisfaction's performance for the Inversion / Transposition / Transversion edit distance problem.

Other improvements

Beyond performance improvements, to increase the quality of our distances, it would be useful to look at variable operation costs (dependent on the length of transposed / inverted segments), and examine how this impacts tree construction. This model modification could then be assessed biologically, and the plausibility of this new model decided.

Similarly, it has been assumed that transversions should occur with the same frequency as transpositions, however, this is not necessarily the biological reality. Experimentation with different transversion weights may show that this weight should be adjusted.

A simple language or interface for biologists to manipulate models of evolution, and construct plan costs and phylognies based on these produced plan costs could be constructed. This would extend the benefits of flexibility from a constraint satisfaction system to biologists, allowing them to perform these experiments themselves.

Appendix A

Technical details

A.1 Benchmark computer details

CPU	Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz
L1 Cache	256KB, 4-way set associative, write-through
L2 Cache	1024KB, 8-way set associative, write-through
L3 Cache	8192KB, 16-way set associative, write-back
Main memory	6GB DDR3 1333Mhz
Mflops/s	64.424 Mflops/s

Table A.1: The relevant specifications for the computer used to benchmark the algorithms presented in the paper.

All calculations were carried out to `double` precision.

Appendix B

Datasets

B.1 Synthetic datasets

For testing and experimentation, we must have genome data to compare and compute distances for. The algorithms discussed in this thesis were tested on these datasets.

Different methods of generating synthetic genome data were experimented with, as the method of generation may be an important factor in the lengths and costs of plans (algorithms may perform better sorting sequences generated by random operations rather than completely random sequences). A complete list of algorithms used to generate these genomes can be seen in Table B.1.

Table B.2 describes the named datasets created from the different genome generation algorithms. For each algorithm listed in Table B.1, and for each length in the range [*length start*, *length end*], *iteration* genomes are created with the given genome generation algorithm.

B.2 Biological datasets

Table B.2 also contains genomes interpreted from real biological data. This data was scraped from the NCBI site [31] [32] (see `http://www.ncbi.nlm.nih.gov/genomes/OrganelleResource.cgi?taxid=33208`).

Name	Description
Random	A completely randomly generated sequence
Inversion	A genome created from random inversions. The number of inversions is equal to half of the genome length, and the inversions are initially performed on the identity genome
Transposition	A genome created from random transposition, in the same way as inversions
Transversion	A genome created from random transversions, in the same way as inversions
Inversion and transposition	A genome created from both random inversions and random transpositions. The split between the amount of each operation is random, and the number of total operations performed is equal to half of the genome length
All	A genome created from random inversions, transpositions, and transversions. The split between the amount of each operation is random, and the number of total operations performed is equal to half of the genome length
Real data	Data from biology literature – not synthetically generated

Table B.1: A table of the modes of generating synthetic datasets. By default, the number of random operations performed on a genome of length n was $\lfloor n/2 \rfloor$

Name	Description	Modes	Minimum Length	Maximum Length	Number of genomes
All lengths	A dataset with many lengths, but few iterations of them. Intended to give a good range of types of genomes for wide testing.	All modes	4	10	2
Short lengths	A dataset with shorter lengths, but with many iterations of them. Also intended to give a good range of types of genomes for wide testing.	All modes	4	7	30
Benchmark	A small dataset used for benchmarking sorting algorithms	All modes	6	8	5
Random	A dataset comprised of completely random sequences only (Only uses the *Random* generation method of Table B.1)	Random only	4	7	30
Tree synthetic 6	A dataset comprised of completely random sequences only (like the Random dataset). Contains problems for each pair of genomes generated (for distance matrix creation)	Random only	6	6	12 ($\frac{12^2}{2}$ total)
Compression stats	A dataset of genomes used for compression experiments, not used for solving.	All modes	4	13	100
Timing	A dataset of genomes used for benchmarking different models.	All modes	4	8	10
Biology mitochondria	A dataset of real mitochondrial genomes collected from the NCBI database [31] [32].	Real data	37	37	3634

Table B.2: A table of datasets used to perform experiments. Modes are modes of generation. N problems were generated for each length in the range [Minimum length, Maximum length], where $N =$ The value from the column "Number of genomes"

Appendix C

MiniZinc model code

Positional model
File name: `genome.mzn`

```
%Genes are usually modified in contiguous blocks
constraint forall(t in stepIndices) (
  noActionTookPlace(t) \/(
    forall(i in geneIndices) (
      forall(j in geneIndices) (
        (
          %we reason about i+1, which is between i and j.
          %Will only select groups as big as 3
          (
            geneMovedThisStep(t,i) /\
            geneMovedThisStep(t,j) /\
            (i+1 < j)
          ) ->
          (
            geneMovedThisStep(t,i+1) \/
            inversionPivotPoint(t,i+1) \/
            (
              transpositionTookPlace(t) /\
              (
                transpositionStarts[t] > transpositionEnds[t]
              )
            )
          )
        )
      )
    )
  )
);
```

Listing 7: The contiguous gene blocks constraint. If a gene between two other genes that change stays stationary, there are only a few possible reasons why (modular transposition, or the center of an inversion)

Positional model
File name: `genome.mzn`

```
%If a gene stays in the same position between steps
%then it may be the center of an inversion,
%or it's not in an action
constraint forall(t in stepIndices) (
  noActionTookPlace(t) \/ (
    forall(i in geneIndices) (
      (
        (planStates[t,i] = planStates[t+1,i]) \/
        (planStates[t,i] = -planStates[t+1,i])
      ) <-> (
        (
          (
            %Case 1: not in a transposition
            transpositionTookPlace(t) /\
            (
              (i < transpositionStarts[t]) \/
              (i > (transpositionEnds[t]+transpositionShifts[t]))
            )
          )\/
          (
            %Case 2: not in an inversion
            inversionTookPlace(t) /\
            (
              (i < inversionStarts[t]) \/ (i > (inversionEnds[t]))
            )
          )\/
          %Case 3: in an inversion, but at the center
          inversionPivotPoint(t,i) \/
          %Case 4: the transposition is wrapping around
          (
            transpositionTookPlace(t) /\
            (transpositionStarts[t] > transpositionEnds[t])
          )
        )
      )
    )
  )
);
```

Listing 8: The stationary genes redundant constraint. If a gene stays in the same position between steps, then it may be the center of an inversion, or it's not in an action

Positional model
File name: `genome.mzn`

```
%Genes in consecutive steps should only have signs changed if
%1. inside an inversion 2. inside a transversion.
%Also, means an action had to take place
constraint forall(t in stepIndices) (
  forall(i in geneIndices) (
    forall(j in geneIndices) (
      (planStates[t,i] = -planStates[t+1,j]) ->
      (
        (
          (
            inversionTookPlace(t) /\
            (i >= inversionStarts[t]) /\
            (i <= inversionEnds[t])
          ) \/
          (
            transversionTookPlace(t) /\
            (i >= transpositionStarts[t]) /\
            (i <= transpositionEnds[t] + transpositionShifts[t]))
        )
        /\
        (not noActionTookPlace(t))
      )
    )
  )
);
```

Listing 9: The gene sign change constraint. If a gene has changed sign in consecutive steps, we can infer some things about what has occurred.

Positional model
File name: `genome.mzn`

```
%Constrain the change in breakpoints to be at most the operation
%that we did between the steps (trans* fix 3, inversions fix 2)
constraint forall(t in stepIndices) (
  (
    transpositionTookPlace(t) ->
    (
      (breakpointCount[t]-breakpointCount[t+1]) <= 3
    )
  ) /\
  (
    inversionTookPlace(t) ->
      (
        (breakpointCount[t]-breakpointCount[t+1]) <= 2
      )
  )
);
```

Listing 10: The breakpoint reduction constraint. We know there is a maximum number of breakpoints that can be fixed by any operations. The `breakpointCount` array is constrained based on the gene content at each step `t`.

Relational model
File name: `genome-relational.mzn`

```
%initialGenome must be reflected in leftNeighbour array
constraint forall(g in geneValues) (
    isLeftNeighbourConstant(1,initialGenome[g],initialGenome[g+1])
) /\ (
    isLeftNeighbourConstant(1,initialGenome[genomeLength],initialGenome[1])
);

%goalGenome must be reflected in leftNeighbour array
constraint forall(g in geneValues) (
    isLeftNeighbourConstant(max(stateIndices),
                goalGenome[g],
                goalGenome[g+1])
) /\ (
    isLeftNeighbourConstant(max(stateIndices),
                goalGenome[genomeLength],
                goalGenome[1])
);

%Also read in the signs
%Initial
constraint forall(g in geneValues) (
    ((initialGenome[g] < 0) -> (geneSign[1,abs(initialGenome[g])] = -1)) /\
    ((initialGenome[g] > 0) -> (geneSign[1,abs(initialGenome[g])] = 1))
);
%Final
constraint forall(g in geneValues) (
    ((goalGenome[g] < 0) ->
      (geneSign[max(stateIndices),abs(goalGenome[g])] = -1)) /\
    ((goalGenome[g] > 0) ->
      (geneSign[max(stateIndices),abs(goalGenome[g])] = 1))
);
```

Listing 11: The constraint that translates from a positional input to a relational output for the relational model.

Relational model
File name: `genome-relational.mzn`

```
%--ORIENTATION--
%If a gene is part of an inversion, usually the left and right
%neighbours need to be swapped between steps (edge cases too)
%Complicated by special case of a width one operation
%(in this case, don't swap the neighbours twice,
%it's both the start and the end which makes
%things confusing and wrong)
constraint forall(t in stepIndices) (
  forall(g in geneValues) (
    partOfInversion(t,g) -> (
      %don't flip anything if it's a single gene
      %(in terms of orientation anyway, we'll change the sign)
      (
        widthOneOperation(t) -> (
           %we know g is the one that is the width one operation
           %since it inverted.
          (leftNeighbour[t,g] = leftNeighbour[t+1,g]) /\
          (rightNeighbour[t,g] = rightNeighbour[t+1,g])
        )
      )/\
      %only do the rest if it's not a width one operation
      (not widthOneOperation(t)) -> (
        %if part of an inversion, swap neighbours, unless edge case
        (
          (g=operationStart[t]) -> (
            %start moves to old end's position
            (rightNeighbour[t+1,g] = rightNeighbour[t,operationEnd[t]]) /\
            (leftNeighbour[t+1,g] = rightNeighbour[t,g])
          )

        )/\
        (
          (g=operationEnd[t]) -> (
            %end moves to old start's position
            (leftNeighbour[t+1,g] = leftNeighbour[t,operationStart[t]]) /\
            (rightNeighbour[t+1,g] = leftNeighbour[t,g])
          )

        )/\
        (
          ((g!=operationStart[t]) /\ (g!=operationEnd[t])) -> (
            %anything in between has both neighbours flipped
            (leftNeighbour[t+1,g] = rightNeighbour[t,g]) /\
            (rightNeighbour[t+1,g] = leftNeighbour[t,g])
          )
        )
      )
    )
  )
);
```

Listing 12: The constraint that inverts the order of genes in the `leftNeighbour[s][g]` array, if it is part of an inversion.

Appendix D

Source files

Excerpts from the positional and relational MiniZinc model are contained in this document, but the complete files were too large to fully list. These files are available for download from `https://github.com/cyberdash/genome-edit`

This repository contains

- The positional and relational MiniZinc model files
- A program to convert and display relational plan output as a positional plan for debugging
- The scraped mitochondrial genome dataset from the NCBI website, including scripts to convert this data into `.dzn` files
- Code for the algorithm that was used to compress genomes

Appendix E

Other notes

E.1 Simulating a transposition with three inversions

It is possible to simulate a transposition operation with three inversion operations. Let

- b_{start} be the start of the transposed block
- b_{end} be the end of the transposed block
- b_{shift} be the amount we shift the block $[b_{\text{start}}, b_{\text{end}}]$ by.
- b_{width} be $|b_{\text{start}} - b_{\text{end}}|$

We then invert with

1. $[b_{\text{start}}, b_{\text{end}} + b_{\text{shift}}]$
2. $[b_{\text{end}}, b_{\text{end}} + b_{\text{width}} - 1]$
3. $[b_{\text{start}}, b_{\text{start}} + b_{\text{shift}} - 1]$

And we are left with the same result. For example, say we transpose

$$1, 2, (3, 4, 5), 6, 7, 8 \rightarrow 1, 2, 6, 7, (3, 4, 5), 8$$

We would then invert

s								
	1	2	(3	4	5	6	7)	8
1	1	2	-7	-6	(-5	-4	-3)	8
2	1	2	(-7	-6)	3	4	5	8
3	1	2	6	7	3	4	5	8

Appendix F

Glossary

ITT	Inversion / Transposition / Transversion
IO	Inversion Only
ITTuw	Unweighted ITT
ITTw	Weighted ITT
PDDL	Planning Domain Definition Language
OTU	Operational Taxonomic Unit

List of Tables

List of listings

List of Figures

Bibliography

[1] Feng Zhu, Chu Qin, Lin Tao, Xin Liu, Zhe Shi, Xiaohua Ma, Jia Jia, Ying Tan, Cheng Cui, Jinshun Lin, Chunyan Tan, Yuyang Jiang, and Yuzong Chen. Clustered patterns of species origins of nature-derived drugs and clues for future bioprospecting. *Proceedings of the National Academy of Sciences of the United States of America*, 108(31):12943–8, August 2011. ISSN 1091-6490. doi: 10.1073/pnas.1107336108. URL http://www.pnas.org/cgi/content/long/108/31/12943.

[2] David A Bader, Bernard M Moret, and Lisa Vawter. Industrial applications of high-performance computing for phylogeny reconstruction. In *ITCom 2001: International Symposium on the Convergence of IT and Communications*, pages 159–168. International Society for Optics and Photonics, 2001.

[3] Catharine E. Pook, Ulrich Joger, Nikolaus StÃČÂijmpel, and Wolfgang WÃČÂijster. When continents collide: Phylogeny, historical biogeography and systematics of the medically important viper genus echis (squamata: Serpentes: Viperidae). *Molecular Phylogenetics and Evolution*, 53(3):792 – 807, 2009. ISSN 1055-7903. doi: http://dx.doi.org/10.1016/j.ympev.2009.08.002. URL http://www.sciencedirect.com/science/article/pii/S1055790309003145.

[4] N C Franklin. Conservation of genome form but not sequence in the transcription antitermination determinants of bacteriophages lambda, phi 21 and P22. *Journal of molecular biology*, 181(1):75–84, January 1985. ISSN 0022-2836. URL http://www.ncbi.nlm.nih.gov/pubmed/3157001.

[5] M Kimura. Evolutionary rate at the molecular level. *Nature*, 217:624–626, 1968. URL http://www.eebweb.arizona.edu/Courses/Ecol426_526/Kimura_1968.pdf.

[6] J W Taanman. The mitochondrial genome: structure, transcription, translation and replication. *Biochimica et biophysica acta*, 1410(2):103–23, February 1999. ISSN 0006-3002. URL http://www.ncbi.nlm.nih.gov/pubmed/10076021.

[7] S Bensch and H Anna. Mitochondrial genomic rearrangements in songbirds. *Molecular biology and evolution*, 17(1):107–13, January 2000. ISSN 0737-

4038. URL http://www.ncbi.nlm.nih.gov/pubmed/10666710http://mbe.oxfordjournals.org/content/17/1/107.short.

[8] George Sawa, Jo Dicks, and Ian N Roberts. Current approaches to whole genome phylogenetic analysis. *Briefings in Bioinformatics*, 4(1):63–74, 2003. URL http://www.ncbi.nlm.nih.gov/pubmed/12715835.

[9] Christopher M Thomas and Kaare M Nielsen. Mechanisms of, and barriers to, horizontal gene transfer between bacteria. *Nature reviews. Microbiology*, 3 (9):711–21, September 2005. ISSN 1740-1526. doi: 10.1038/nrmicro1234. URL http://www.ncbi.nlm.nih.gov/pubmed/16138099.

[10] WM Fitch. Toward defining the course of evolution: minimum change for a specific tree topology. *Systematic Biology*, pages 406–416, 1971. URL http://sysbio.oxfordjournals.org/content/20/4/406.short.

[11] Gabriel L. Wallau, Mauro F. Ortiz, and Elgion L. Loreto. Horizontal Transposon Transfer in Eukarya: Detection, Bias, and Perspectives. *Genome Biology and Evolution*, 4(8):801–811, January 2012. ISSN 1759-6653. doi: 10.1093/gbe/evs055. URL http://dx.doi.org/10.1093/gbe/evs055.

[12] Haim Kaplan, R Shamir, and RE Tarjan. A faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal on Computing*, 2:1–8, 1997. URL http://epubs.siam.org/doi/pdf/10.1137/S0097539798334207.

[13] Vineet Bafna and Pavel a. Pevzner. Sorting by Transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, May 1998. ISSN 0895-4801. doi: 10.1137/S089548019528280X. URL http://epubs.siam.org/doi/abs/10.1137/S089548019528280X.

[14] Zhi Yang Wong. Computing optimal genome edit distances under inversion and transposition. *ANU/NICTA Summer Scholar Research 2012*, 2012.

[15] Roman Barták and Miguel a. Salido. Constraint satisfaction for planning and scheduling problems. *Constraints*, 16(3):223–227, May 2011. ISSN 1383-7133. doi: 10.1007/s10601-011-9109-4. URL http://www.springerlink.com/index/10.1007/s10601-011-9109-4.

[16] Roman Barták and Daniel Toropila. Revisiting Constraint Models for Planning Problems. pages 582–591.

[17] Roman Barták, MA Salido, and Francesca Rossi. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufactur-*

ing, pages 1–12, 2010. URL http://www.springerlink.com/index/n738j8333205134n.pdf.

[18] Luca Bortolussi and Andrea Sgarro. Constraint satisfaction problems on DNA strings. *WCB05*, 2005. URL http://www.dimi.uniud.it/dovier/WCB05/wcb05.pdf#page=17.

[19] Patrik Haslum. Computing Genome Edit Distances using Domain-Independent Planning. *ICAPS 2011 Scheduling and Planning Applications Workshop*.

[20] M Blanchette, T Kunisawa, and D Sankoff. Gene order breakpoint evidence in animal mitochondrial phylogeny. *Journal of molecular evolution*, 49(2):193–203, August 1999. ISSN 0022-2844. URL http://www.ncbi.nlm.nih.gov/pubmed/10441671.

[21] M Blanchette, T Kunisawa, and D Sankoff. Parametric genome rearrangement. *Gene*, 172(1):GC11–7, June 1996. ISSN 0378-1119. URL http://www.ncbi.nlm.nih.gov/pubmed/8654963.

[22] Barbara M. Smith. Dual models of permutation problems. *Principles and Practice of Constraint Programming - CP 2001*, pages 615–619.

[23] A Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. *Lecture Notes in Computer Science*, 0(0), 2001. URL http://link.springer.com/content/pdf/10.1007/3-540-48194-X_9.pdf.

[24] Sridhar Hannenhalli and PA Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM (JACM)*, 46(1):1–27, January 1999. ISSN 00045411. doi: 10.1145/300515.300516. URL http://portal.acm.org/citation.cfm?doid=300515.300516http://dl.acm.org/citation.cfm?id=300516.

[25] MDV Braga. baobabLuna: the solution space of sorting by reversals Documentation. 2009. URL http://biom3.univ-lyon1.fr/software/luna/doc/luna-doc.pdf.

[26] N Saitou and M Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, 4(4):406–25, July 1987. ISSN 0737-4038. URL http://www.ncbi.nlm.nih.gov/pubmed/18343690.

[27] Bertrand Neron, Herve Menager, Corinne Maufrais, Nicolas Joly, Julien Maupetit, Sebastien Letort, Sebastien Carrere, Pierre Tuffery, and Catherine Letondal. Mobyle: a new full web bioinformatics framework. *Bioinformatics*, 25(22):3005–3011, 2009. doi: 10.1093/bioinformatics/btp493.

URL http://bioinformatics.oxfordjournals.org/content/25/22/3005.abstract.

[28] Kevin Howe, Alex Bateman, and Richard Durbin. Quicktree: building huge neighbour-joining trees of protein sequences. *Bioinformatics*, 18(11):1546–1547, 2002. doi: 10.1093/bioinformatics/18.11.1546. URL http://bioinformatics.oxfordjournals.org/content/18/11/1546.abstract.

[29] Ivica Letunic and Peer Bork. Interactive tree of life (itol): an online tool for phylogenetic tree display and annotation. *Bioinformatics*, 23(1):127–128, 2007. doi: 10.1093/bioinformatics/btl529. URL http://bioinformatics.oxfordjournals.org/content/23/1/127.abstract.

[30] Ivica Letunic and Peer Bork. Interactive tree of life v2: online annotation and display of phylogenetic trees made easy. *Nucleic Acids Research*, 39(suppl 2):W475–W478, 2011. doi: 10.1093/nar/gkr201. URL http://nar.oxfordjournals.org/content/39/suppl_2/W475.abstract.

[31] Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. GenBank. *Nucleic acids research*, 37(Database issue):D26–31, January 2009. ISSN 1362-4962. doi: 10.1093/nar/gkn723. URL http://nar.oxfordjournals.org/cgi/content/long/37/suppl_1/D26.

[32] Eric W Sayers, Tanya Barrett, Dennis A Benson, Stephen H Bryant, Kathi Canese, Vyacheslav Chetvernin, Deanna M Church, Michael DiCuccio, Ron Edgar, Scott Federhen, Michael Feolo, Lewis Y Geer, Wolfgang Helmberg, Yuri Kapustin, David Landsman, David J Lipman, Thomas L Madden, Donna R Maglott, Vadim Miller, Ilene Mizrachi, James Ostell, Kim D Pruitt, Gregory D Schuler, Edwin Sequeira, Stephen T Sherry, Martin Shumway, Karl Sirotkin, Alexandre Souvorov, Grigory Starchenko, Tatiana A Tatusova, Lukas Wagner, Eugene Yaschenko, and Jian Ye. Database resources of the National Center for Biotechnology Information. *Nucleic acids research*, 37(Database issue):D5–15, January 2009. ISSN 1362-4962. doi: 10.1093/nar/gkn741. URL http://nar.oxfordjournals.org/cgi/content/long/37/suppl_1/D5.