Computing optimal genome edit distances under inversion and transposition

ANU/NICTA Summer Scholar Research 2012
Supervisor: Dr Patrik Haslum
Zhi Yang Wong

Background
In this summer research project the aim was to compute optimally the genome edit distance under transpositions, inversions and transversions. Currently, there is a polynomial time algorithm that computes this optimally under inversions only. However, no work has been done to compute this optimally under transpositions, inversions and transversions.

The main application for computing genome edit distances is that it could be used in comparative studies between different genomes. The shorter the edit distance, the closer the genomes are in the phylogeny. Optimality was the main focus in computing the edit distance, this was due to the fact that currently in the costs of each of the operations are unknown and having a sub-optimal algorithm would increase even further any inaccuracies in any comparative gene studies.

This combinatorial problem has various properties that makes it an interesting search problem to study. Due to the nature of the problem, the genome edit distance problem has a number of similarities with the pancake problem a famous search problem, this makes the pancake problem something to also look at when thinking about solving the genome edit-distance problem.

Methodology

Genome Edit Distance

In computing the genome edit distance we applied it only to the case of circular genomes, but could easily be modified for linear genomes. Also, each of the genes in the genome has an orientation, for simplicity the genome sequence was represented as a numerical sequence with each number representing a specific gene and the sign representing the orientation of the gene.

There are three operations that these genomes can undergo each with different weights that could be changed to more realistic values. For our studies we set the costs arbitrarily to 1. A transposition moves a part of a genome to a different position, an inversion inverts a segments order and the orientation that it was in. A transversion is simply a combination of a transposition and an inversion. For the purposes of testing in our case we have arbitrarily allocated a cost of 1 for each of the operations as the exact relative costs are not definite and depend on the problem at hand.

IDA* Search

A simple A* search was initially implemented to find the optimal edit distance without a heuristic. This was used as the base of the algorithm for the problem. However it was found to be not a suitable method when the number of genes in the genome increased. This is because the size of the open list for the A* search increased rapidly and the overhead computational cost related to it caused this search method to be too slow.

Instead an Enhanced Iterative-Deepening Search (Reinfeld and Marsland(1994)) was implemented. This search algorithm was used as it did not need to store the open list like in the A* search, this removed the limitation that was involved with having a large open list for the A* search. This is important as the branching factor for the problem is large and the actual search is very shallow in comparison.

The enhanced iterative deepening search was also the most practical as it only required a small amount of memory and also kept the algorithm optimal. In the Enhanced Iterative-Deepening Search, a transposition table was used to store any states that had already been explored, although this required extra memory, the time saved from not exploring that state outweighs the memory cost.

Heuristics

Pattern Database Implementation The main heuristic used for our search algorithm was a pattern database heuristic. This stored the shortest edit-distance for all different combinations for a certain sized problem. Initially an exhaustive A* search was implemented in order to build this pattern database until all heuristic values were found for all the permutations. However, due to the large branching factor the open list again grew too quickly and hence slowed down this process.
To overcome this problem, we made use of a bucket list method. This method works quite well for our problem due to the large branching factor for our problem and in addition to this there are only a few number of costs that each of the permutations could have. The bucket list method calculated the shortest edit-distance and stored them in 'buckets' based on the cost. Once the breadth first search had found all of the costs for a certain cost. These buckets were stored in a separate array and the next 'bucket' with a greater cost was calculated for, this process was repeated until all the permutations were found.

Pattern Database Limitations  It was found that the computation time for the states at a lower cost were found much faster as most of the possible permutations only required a small number of operations. Hence, most of the possible permutations were found at a low cost and only a small fraction of the pattern database was found at a high cost. Due to this, we found that if we just found a large proportion of all the pattern databases at the lower costs this would be sufficient for our heuristic as it was more useful to have a slightly incomplete pattern database for a large size than to have a complete pattern database with all the optimal costs found. This would greatly reduce the computation time for the pattern database as the last few permutations would take a long time to find due to the large branching factor. For the permutations that were not found the last bound was used to be the underestimate of the heuristic.

The main limitation in creating our pattern database was the amount of memory that was required. The size of the pattern database grew at an exponential rate with increasing number of genes. For our pattern database we were limited to a pattern database for a size of 9 genes. This pattern database already had around $2 \times 10^5$ heuristic costs that translated to about 200Mb in memory. For size of 10 this would increase the size in the memory to be about 2Gb which would be impractical for our purposes.

Lexicographic Ordering  Due to the number of heuristic calls it was found to be important to efficiently rank these permutations hence a lexicographic ordering system on the permutations was used. The algorithm used was described in (Bonet(2008)), it is a $O(n \log n)$ lexicographic ranking system. Bonet’s paper describes only for the case of positive permutations, to incorporate the signed aspect in our problem, a n-bit base 2 number was used to represent the signed aspect of the permutation.

Sampling Techniques  In order to obtain a heuristic from the pattern database, a sample from the current state has to be taken in order to generate a heuristic. From this sample the state is to be compared to the goal state and the permutation can then be ranked lexicographically and corresponding heuristic can be found in the pattern database. In order to get a satisfactory heuristic from the sample, multiple samples were taken from the state such that the heuristic would be the best estimate possible. However a limitation of this is that as the number of samples increases the computational cost of this also increases as more computation is required in finding a random sample and ranking that sample to the pattern database. It was found that looking up the heuristic took up most of the time when solving for the shortest edit distance.

Due to this limitation, one of the approaches was to include some intelligent sampling techniques such that the number of random samples could be reduced hence also hopefully decrease the computation time. One of the methods that was implemented was to keep the previous sample combination that produced the best heuristic and to sample this combination again.

Breakpoint Heuristic  Another heuristic that was considered was the breakpoint heuristic. This was based off the idea from the pancake problem where a gap heuristic was used (Helmert(2010)). The definition for breakpoints is defined in (Blanchette et al.(1999)Blanchette, Kunisawa, and Sankoff). Consider two genomes $A = a_1...a_n$ and $B = b_1...b_n$ over the same set of genes $\{g_1,...,g_n\}$, where each gene is signed. We say $a_i$ precedes $a_{i+1}$ and $a_n$ precedes $a_1$. If gene $g$ precedes $h$ in $A$ and neither $g$ precedes $h$ nor $-h$ precedes $-g$ in $B$, that defines a breakpoint in $A$. As we can rename our values such that in our goal genome order to be numerically increasing 1, ..., $n$, we just need to find which parts in $A$ that do not have $a_n = a_{n+1} - 1$.

From these breakpoints we can calculate the minimum number of moves that are required. Out of the three operations at most one operation can fix only 3 breakpoints. Hence the breakpoint heuristic can be the number of breakpoints divided by 3, this value is rounded up as the number of operations must be a whole number.

Breakpoint Sampling  Another intelligent sampling method was to sample each of the values in between the breakpoints and use those values to lookup into the pattern database for an accurate heuristic. If the number of breakpoints is less than the size of the pattern database the other values can be arbitrarily sampled to obtain the permutation.

Sign heuristic  The number of consecutive segments genes that have the wrong sign was also counted up. This can be used as another heuristic as one inversion or transversion can at most fix one consecutive stretch of genes that have the wrong sign.

Results  There were 4 different heuristic methods that were to be tested. Recording the previously best heuristic sample, breakpoint heuristic, sampling between breakpoints
and the sign heuristic. Due to time constraints these tests were tested with random genome sequences with a length of 13, with a pattern database of size 9.

From Figure 1 it can be seen that as the number of samples for the heuristic increases, the number of node expansions decreases. This is because as the number of samples increases, this means that the heuristic value that will be obtained will be a better estimate, resulting in less node expansions. It can also be seen that this effect decreases exponentially as there most likely is a certain limit at around 15 where any increase in the number of heuristic samples would not make a change in the heuristic output as the maximum heuristic has already been obtained.

From the graph of the nodes expanded for test case 1, it can be seen that only sampling between breakpoints has a significant effect on the number of node expansions and the time that it takes to solve for the optimal solution. This effect decreases as the number of samples used for the heuristic increases, this is most likely because as the number of heuristic samples increases there is no need for any intelligent sampling as the heuristic value would most likely already be at the maximum. Also, this means that increasing the number of heuristic samples would not only have no effect on the number of node expansions but increase the time taken, due to the extra computation as seen from the graph of the time taken.

For test case 2 it can be seen that there is a much greater variation in the number of nodes expanded for the different scenarios. It can be seen that this variation is mainly due to the scenarios involving the breakpoint heuristic. This effect is could be due to the breakpoint heuristic causing the algorithm to go a certain path that is not the solution path, hence increasing the number of nodes expanded. This reason is further reinforced by the variability of the number of nodes expanded for the other test cases. For the other scenarios the number of nodes expanded jumps between around 200 to 1200. This may be due to one important decision that is made early in the computation and depending on the path taken would result in more node expansions. At 10 samples it can be seen that the signed breakpoint heuristic has the least number of node expansions, this could be an indication that this heuristic does help the solution.

Future Works

It may seem that sampling between breakpoints and the signed breakpoint heuristic are good additions to the pattern database heuristic, whilst the breakpoint heuristic could be detrimental. However, these relationships has only been tested for a few different test cases. A more extensive investigation on these heuristics needs to be completed in order to truly determine which heuristic methods have a positive or negative effect. Investigations on the actual type of permutation could also be done and how to identify these permutations.
Conclusions

• Currently the algorithm produced is unable to efficiently solve all genome problems as some problems have sizes up to 30-40 genomes.

• Insufficient memory space to increase the pattern database for a large sized permutation

• Breakpoint sampling has a positive effect on the algorithm for most test cases

• There is evidence that there are crucial steps in finding the shortest path that could increase or decrease the number of node expansions significantly

References


