Software Applications: AI Search Assignment

Matthew Dyson

Introduction

This report documents the development of my two algorithms, henceforth known as A and B, to solve the Travelling Salesperson problem presented in the form of ‘city-files’. I chose to produce an A* search and a genetic search algorithm to solve this problem in two entirely different manners. Contained within this report are the results of running my algorithms on the 10 problems provided, a brief explanation of the working of each algorithm – including the fine tuning performed to get the best results, as well as a comparative analysis of performance.

In measuring performance, I took the running time that each algorithm took to solve a particular problem to be a measure of efficiency. Obviously, the higher a run time, the less efficient an algorithm is. Throughout this report, n is taken to represent the size of the input being dealt with, and x is a city.

Results

Size	**A***		**Genetic**	
	Result	Time (ms)	Result	Time (ms)
12	98	10	78	53
17	1804	3	2341	47
21	3047	6	3924	71
26	1760	26	1981	89
42	1457	106	2101	284
48	18105	229	28697	487
58	32346	130315	70099	666
175	22947	820490	42537	9473
180	4350	117503	545030	8018
535	*Did not complete*		137073	103068

Algorithm ‘A’ – A* Search

Implementation

My implementation of A* Search relies heavily on an external class (AStarNode.class) for its running. An instance of AStarNode is comparable to another through the ‘Comparable’ interface, allowing for the efficient sorting of the list of fringe nodes (stored in a set) by analysis of each nodes predicted cost to finish (F score). The state of the path within this class is stored as a LinkedList of integers, each one representing a city on the tour.

The algorithm begins by generating a single state – the starting node (given as 1 for all cases for ease of use), which is then added into a TreeSet (acting as a store for the fringe nodes), which then is used to begin iteration. The current state is checked to see if it is a goal state (starts and finishes on the same node, and correct length), and returned as the correct answer if so. Otherwise, all potential additions to the state are considered, and added to the fringe as a new AStarNode. In creating this new node, the new F score is calculated, which in turn requires a separate algorithm.

In order for A* to be an admissible search function, the heuristic function $H(x)$ must also be admissible ($H(x) \leq x_i + x_{i+1} + \cdots + x_0$ where $F(x) = G(x) + H(x)$) – which is extremely hard to efficiently calculate. Instead, I opted to use a simple nearest-neighbour algorithm to calculate the remaining journey. This did not guarantee $H(x)$ to be admissible, however I deemed it to be a fair compromise between accuracy and speed, lowering runtime for the entire process at the expense of the result.

Adding a new AStarNode to the fringe TreeSet sorts every member of the fringe in order of ascending F value. This is done through the 'Comparable' interface as discussed before, and is done automatically – TreeSet inherits from the Java SortedSet class[1].

The AStarNode at position 0 in the fringe is chosen as the next state, then we iterate back to the start of the algorithm. A null return is given at the end of the algorithm – if no goal has been found once all fringe nodes have been examined then a path does not exist.

Fine tuning

Initially, I had begun developing the algorithm using a slightly different approach in terms of coding style. I was trying to reduce the memory footprint necessary for the program to run by avoiding use of a separate class, and for this reason used a HashSet and separate Comparator object combined with the Java Collections class in order to execute the ordering of the node list. However, I quickly discovered that this was vastly inefficient, and yielded massive penalties in terms of runtime. After experimenting, I found that using a TreeSet was much more efficient, and the memory overhead necessary for this approach did not exceed the allocated size of the Java Virtual Machine.

On examining the runtimes of this function (in comparison to a basic brute-force implementation not documented here), I determined that further adaptations were necessary to achieve better performance. To begin with, the heuristic function would drastically slow down the execution (as it is being run $\leq n$ times per fringe examination) of the algorithm. I therefore decided to opt for a simpler heuristic, which takes the mean path length of all paths that could be used to complete the tour. Testing this on several input cases, execution time was improved by up to 90%, at the expense of a ±15% increase in inaccuracy of results.

Despite my attempts at code optimisation, I could not get the algorithm to complete when using the $n = 535$ problem. I examined the runtime of various individual components of the algorithm on smaller sized problems, and found that that running the heuristic function was still causing a bottleneck. In order to generate an accurate heuristic value for any given node, huge amounts of calculations and loops were being done. In a best case scenario (the answer being generated in a linear progression, no other possibilities examined), over 150 million operations would be required to reach the end of the algorithm – an impossible task for any normal processor.

Algorithm 'B' – Genetic Search

Implementation

The basis of my Genetic Search algorithm is formed around another separate class (GeneticNode.class), which again is comparable to another instance of itself. Similarly to algorithm A, a LinkedList of integers is used to store the path. The class also contains a reproduceWith() function, which returns a child having performed the necessary operations to generate a valid solution from the current instance and the instance passed to it. This

[1] http://java.sun.com/j2se/1.4.2/docs/api/java/util/SortedSet.html

class is then implemented through use of another TreeMap, which contains the population of my solutions sorted by their respective cost (lowest to highest).

Initially, the algorithm adds an initial population of 100 potential solutions, generated at random, and then begins the iterative loop. The algorithm first chooses two random numbers between 0 and the population size (p), with the probability of choosing a lower number having exponentially larger probability than that of choosing a higher number, in order that better solutions are chosen for reproduction. These two numbers are taken to be solutions in the population by their array index within the TreeMap. These solutions then are used as parents to form two new child solutions, which can be represented as the following:

$$c_1 =< p_1[1] , p_1[2] \dots\ p_1[y], p_2[y+1] \dots p_2[n]\ >$$

$$c_2 =< p_2[1] , p_2[2] \dots\ p_2[y], p_1[y+1] \dots p_1[n]\ >$$

Where y is another randomly generated number. The children are then tested to see if they contain any repeated values in their path, which are replaced so the children are still valid solutions to the search problem. The children are then given the chance to mutate (with a given, low, probability), which switches the index of two random values in the list.

The children are then assessed to see which is the best solution to the search problem (lowest path length), and the best suited child is added into a secondary population. Once this iteration has run p times, the primary set is replaced with the secondary set, and the algorithm runs again. This whole process repeats as many times as necessary, before returning the best solution that has been generated at any stage.

Fine tuning

To begin with, the number of iterations of the algorithm was based on a static number, which was the same for all problems. I discovered that keeping this number static caused smaller problems to take a long time to run, and larger problems to return poor answers, even after finding a fair trade-off between speed and efficiency (found to be ~10,000 iterations). For this reason, I adapted the algorithms terminating conditions to wait for a ‘plateau’ to be found in the generated results. If the optimal path found does not change after 10,000 iterations, the algorithm halts.

Obviously, the relationship between the size of the problem and the number of iterations needed to get an admissible solution is linear, and therefore I chose to further develop the efficiency of the algorithm by linking these two factors. After experimentation, I found that using a value of $100n$ instead of 10,000 for the plateau gave a decrease in running time of up to 70% on smaller problems (which brings the runtimes much closer to that of Algorithm A, although it is worth considering that at this point we are measuring improvements on the scale of hundredths of seconds!), with minimal effect on results. Obviously, for larger problems where $100n > 10{,}000$ the running time was increased, however this did vastly improve the quality of results returned. For such a wide range of values of n, it is difficult to find a value to govern the number of iterations that is both quick on small inputs and optimal on large inputs.

Originally, I thought it unfair on the testing procedure to modify the plateau value artificially depending on the scale of the input, however I believed that for any values where $100n > 10{,}000$, the latter should be used for the sake of speed efficiency, as in quite a lot of cases the increase in iterations will result in minimal improvements. I ran a series of experiments using the case $n = 535$, and found that by limiting the plateau to 10,000 rather than the 53,500 it would be using $100n$, the run time was halved, and the average increase in returned path length was only 5% - a small sacrifice for such a massive improvement on runtime. In light of these results, and after further experimenting to find an acceptable peak value, I changed the algorithm to cap the value of the plateau variable at 10,000 iterations.

The probability of mutation was initially set at 5%, which seemed to give good results for a wide variety of input values. Setting this value any higher caused small problems to be solved through a rather chaotic fashion, akin more to guesswork than actual computation! However, for large problems the process of mutation becomes even more important, otherwise it is possible to continue iterating over the same poor solutions, returning a very bad result. I found 5% to be a fair trade off between these two extremes.

Comparative Analysis

Both of these algorithms are similar in the respect that in programming them, the creator is given the choice of optimizing solutions versus runtime, a choice that would be affected in the real world by the situation it were to be placed in. In a game, where speed is of the essence, then an algorithm that could run quickly and efficiently would be called for, whereas in a piece of software where an optimal solution was called for, speed could be sacrificed. This difference in purpose is represented quite accurately within my algorithms. Algorithm A gives near-optimal solutions (albeit not guaranteed to be optimal), at the expense of runtime, to the end that the algorithm will fail to complete for large problems. In contrast, algorithm B will return a ‘quick and dirty’ solution that is often far from optimal. However – it is worth noting that in certain cases algorithm B can return a better solution than algorithm A, which essentially is put down to the probability of landing on a better path through random generation, as each run of algorithm B will return a different result due to its nature.

In terms of memory footprint, both algorithms are not ideally suited to running in a heavily restricted system, as for larger problems the number of solutions held at any given time can run to thousands, if not tens of thousands. Algorithm A is worse in this respect, as the fringe is not restricted in size at all, and in a worst case scenario could be holding $n!$ solutions at the same time. Algorithm B does have a restricted size, and hence will be much more memory efficient – another reason why it would be better suited in a gaming environment.

Both algorithms have been optimised using a single 2.66GHz processor core, running in an unmodified Java Virtual Machine. Obviously, running the algorithms on a lower specification machine would increase the running times, and vice versa. For this reason, further modifications could be made to assess the environment the algorithm is running in, and adapt the variables involved to suit this. For instance, running a lower number of iterations on algorithm B on a less powerful processor would give a similar runtime at the expense of accuracy. This example could also be worked in reverse for higher specification machines, even moving onto multithreading to run several different population simulations concurrently, bringing them together finally to reproduce children from the best solutions of each simulation. Algorithm A does not lend itself to multithreading due to the nature of the calculations involved.

Conclusion

In conclusion, the algorithms that I have created to solve the problems given have their respective individual strengths and weaknesses, lending themselves to different applications if they were to be used in a live environment. Within the scope of this project, I have done as much testing and modifying as possible, however I believe that the sky is the limit in terms of how these algorithms could be further tweaked in order to get the desired balance between performance and accuracy, which is the fundamental issue the programmer must consider when attempting to solve these problems.