AI Search Assignment

Maxi Clayton Clowes

22/01/2015

Using the code

Note: Code is in Python, version 3.4.2

To test the code, run *main.py*. This file will prompt the user to select one of 4 implemented algorithms:

- *"brute"* – a true Brute Force algorithm (included purely for academic discussion purposes)
- *"modified"* – a modified Brute Force algorithm, which returns the best found tour after *n* seconds
- *"nearest"* – a Nearest Neighbour algorithm
- *"genetic"* – a Genetic algorithm

Once an algorithm has been chosen, the user will then by asked to enter a valid input file (suffix included), which the program will then parse to the desired search function. Once the algorithm has completed running, it will produce at output file containing the best tour it discovered and that tour's length.

A) Modified Brute Force Search – *modified_brute_force.py*

The first algorithm I chose to implement, due to both it's simplicity of implementation (appropriate for getting to grips with the assignment) and the optimality of its results was a Brute Force search algorithm (contained in *brute_force.py*, found in the folder *pvxf29rest*).

The Brute Force algorithm iterates through each possible tour permutation and keeps track of which tour is the best. Permutations are generated when required and then forgotten rather than all being generated in one go, which would put huge stress on memory. Once all tours have been checked the best is returned and outputted.

This approach checks all possible tours and thus guarantees the discovery of the optimal tour, given enough time. One of the main problems with each algorithm is knowing when to stop; that is, the optimum solution may be the first tour tried, but it will keep running unnecessarily as it is not known that the optimum solution is reached.

However, there is a known lower bound of tours for a given graph, which is the smallest length any tour could potentially be for a given graph. This can by given by removing a node from the graph, generating a minimum spanning tree (through a technique such as Primm's algorithm) and reconnecting the removed vertex by the two smallest connecting edges, as I have implemented in the method *lowest_bound()* in *graph_tools.py*. Although it is not usual for the lowest bound to actually be a valid TSP tour, sometimes this is the case, as in *AISearchtestcase.txt*, where the first permutation is not only an optimal tour but is also equal in length to the lower bound. At this point the program stops, saving the program from iterating through another 40,319 tour permutations.

However, despite this added break point the worst-case time complexity of the algorithm is defined by the growth of the set of tour permutations. For a graph of size *n*, there will be *n!/(n-r)!* possible tours, making the time complexity of the brute force solution O(n!). For example, there are 40,320 possible tours of a graph with 8 nodes, 479,001,600 for 12, and 355,687,428,096,000 for 17; hence, the impracticality of a pure brute force implementation is clear.

In fact, my initial brute force implementation was unable to return solutions within a reasonable time frame for the test graphs of size 17 and above. To overcome this limitation, I then modified the brute force algorithm to include a time-based break point. By breaking the function after *n* seconds (where n= the number of vertices in the input graph), the time complexity of the function is vastly reduced to O(n) at the expense of the quality of the solutions produced. I also experimented with other time limits, such as *2n* and n^2, but found that the increase in tour quality didn't justify the signi increase in time.

1st Permutation: {1, 2, 3, 4, 5}
2nd Permutation: {1, 2, 3, 5, 4}
3rd Permutation: {1, 2, 4, 3, 5}
4th Permutation: {1, 2, 4, 5, 3}
5th Permutation: {1, 2, 5, 3, 4}
…

Figure 1 - Example of permutation iteration

The problem with limiting based on time is the method of iterating through permutations in the pure brute force approach leads to little variety across generated tours. Figure 1 shows the order in which the initial iterator would produce permutations. As can be seen, over the first few permutations, the tour always contains the edge {1,2}; if {1,2} were a particularly large edge then all permutations containing {1,2} would be of poor optimality. Therefore, ensuring variation across tours is key to maximal quality of solution in minimal time.

	Generated tour length	
Input File	**Non-random permutations**	**Shuffled Permutations**
AISearchfile012.txt	66	66
AISearchfile017.txt	3451	2148
AISearchfile021.txt	6599	4128
AISearchfile026.txt	2369	1904
AISearchfile042.txt	2666	2140
AISearchfile048.txt	45375	33339

To achieve this variety I used the pseudo-random method *shuffle()* which delivers permutations randomly rather than iterating in order. Although technically not brute force, given the imposed time constraints it delivers just as many tour permutations with far more variety and thus gives vastly improved results overall for no increase in run time, as can be seen in the above table. However, this does result in a non-deterministic algorithm and variations in tour outputs. The effect of this is that varying the time break ceases to offer any notable improvement in tour quality despite taking significantly longer.

By implementing a few alterations, the impractical brute force approach has been changed into an algorithm that can generate tours of mediocre quality in a very quick amount of time. Although the tour quality is typically low, the fast tour generation offered is a bit advantage and it's linear growth makes it particularly useful, especially at larger problem sizes.

B) (Repetitive) Nearest Neighbour Search – *nearest_neighbour.py*
I next implemented a Nearest Neighbour Search algorithm. This approach initialises a tour at a starting node and visits the nearest vertex to the last node repeatedly until the tour is complete (ignoring edges that would break the cycle's validity). My recursive implementation returned a tour, within a very reasonable time.

I then improved upon this by changing it into a *repetitive* nearest neighbour implementation by running the nearest neighbour method on all possible starting nodes. The trade-off is that the simple nearest neighbour algorithm ($O(n^2)$) is being run *n* times, meaning that the time complexity of the repetitive nearest neighbour search is $O(n^3)$.

Recursion appeared an elegant execution of the algorithm given the algorithms minimal memory requirements. As only n tours are created from a huge size of potential tours, there was no point lower bound checking.

Once the best tour is returned, a tour improvement method runs through that tour, swapping 2 adjacent nodes in order, and only if the swap improves the tour length is the swap kept. This achieves small improvements for

C) Genetic Algorithm Search – *genetic.py*
My final algorithm was a genetic search algorithm. It begins by creating an initial random population of size *n,* and then adds in a tour generated by running nearest neighbour on the first node. This addition from nearest neighbour ensures at least one member of the population is of a reasonable quality. I experimented with creating the initial population entirely from repetitive nearest neighbour but the time required to run nearest neighbour followed by genetic was unnecessarily large.

The algorithm then randomly selects two parents from the population, weighted towards the top quarter of the population. I experimented with heavier weighting towards the best tours of the population but this failed to present any notable improvement and in some cases resulted in significantly worse results where the population will have become trapped within a local minimum (despite the additions of mutation).

Two children (c1 and c2) are generated from these two parents (p1 and p2) by taking a slice form the start of a parent (c1 takes from p1, etc.) and then filling the remaining spaces by in order checking the remaining parent and appending missing nodes.

Each child has a 50% chance of mutating, and is mutated by one of two methods: the first mutation reverses a slice of the child tour; the second runs a tour improvement on the child. This mutation is crucial for avoiding homogeny across the population and the population getting trapped in a local minimum. The final children are then added

to the population and the population is sorted before the worst 2 members of the population are removed.

The algorithm is set to run for *n* seconds, although I played with iterating for a certain number of generations and for longer periods of time. The quality of the outputted tours are generally very good for the limited time, performing better than nearest neighbour for the smaller sizes. However, the larger the graph becomes the more insufficient running for such a short period of time becomes and ideally the genetic algorithm should run for a longer period of time. In the results table, it is clear that from graph size of 48 onwards nearest neighbour performs a little better. However, genetic is running much faster at these larger sizes and were they to run for an equal period of time the genetic would likely give the better tour.

Results

Algorithm	**Time Complexity**
Brute Force	O(n!)
Modified Brute Force	O(n) - Artificial
Nearest Neighbour	$O(n^3)$
Genetic Algorithm	O(n) - Artificial

	Tour obtained		
Input file	**Modified brute force**	**Nearest neighbour**	**Genetic**
AISearchfile012.txt	66	**56**	**56**
AISearchfile017.txt	2148	1628	**1564**
AISearchfile021.txt	4128	3130	**2754**
AISearchfile026.txt	1904	1731	**1664**
AISearchfile042.txt	2140	1534	**1408**
AISearchfile048.txt	33339	**15172**	15509
AISearchfile058.txt	81629	**27208**	**27208**
AISearchfile175.txt	44203	**21951**	22296
AISearchfile180.txt	659840	**5420**	8890
AISearchfile535.txt	147349	**50029**	50270