**AI Search Assignment**  
Maxi Clayton Clowes   
17/01/2015

Intro

Talk about the goals – best (shortest distance) tour in shortest time

Use the word heuristic

**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 select a valid input file, 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.

*Describe implementations -> overview*

*Focus on issues  
Specific details -> data structure choice, etc*

*Demonstrate understanding of each algorithm*

*ADD MST STUFF IN -> MST = Minimum(graph)*

**Modified Brute Force Search**

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 quality 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. However, the 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 from O(n) leading to big gains in terms of run time at the cost of the quality of the solutions produced. In table 1 the results of my initial experimentation can be seen. The modified brute force variations returned reasonable solutions

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input file** | **Optimal tour length** | **Modified Brute Force – *O(n2)*** | | **Modified Brute Force – *O(2n)*** | | **Modified Brute Force – *O(n)*** | |
| **-** | **-** | **Result** | **Time(s)** | **Result** | **Time(s)** | **Result** | **Time(s)** |
| AISearchfile012.txt | 56 | 56 | 144 | 66 | 24 | 66 | 12 |
| AISearchfile017.txt | - | 2791 | 289 | 3364 | 34 | 3451 | 17 |
| AISearchfile021.txt | - | 6036 | 441 | 6421 | 42 | 6599 | 21 |

Table - Initial experimentation result of brute force time-based modification

To improve the quality of the solutions it was also necessary to change the way the algorithm passed permutations. When the algorithm was run without an adapted permutation iterator, the algorithm was unable to run for long enough to see any major variation. 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.

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 - Example of permutation iteration

I then added shuffle?

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 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, in larger graphs this extra checking leads to a very small increase in run time which one may decide is not worth the potential pay off given the rarity of cases where lower bound is equal to optimal tour.

**(Repetitive) Nearest Neighbour Search**

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. complexity of O(n2).

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(n2)) is being run *n* times, meaning that the time complexity of the repetitive nearest neighbour search is O(n3).

Contrast results for O(n2) vs O(n3)

Why recursion?  
Recursion appeared an elegant execution of the algorithm given the algorithms minimal memory requirements.

As before, lower bound checking has been included, potentially saving time.

**Genetic Algorithm Search**

The most interesting solution I implemented xxxxx

**Results**

The first algorithm I chose to implement, due to both its simplicity of implementation (appropriate for getting

*Tabulated description of results*

* *lengths of best tours*
* *analyse quality*
* *run time*
* *better tours = better marks*

*Details of experiences running the implementations on different inputs*

*Comparative analysis*

*Fine tuning and experimentation -> improving performance*

|  |  |
| --- | --- |
| **Algorithm** | **Time Complexity** |
| Brute Force | O(n!) |
| Modified Brute Force | O(n) or O(n^2)??? |
| Nearest Neighbour | O(n3) |
| Genetic Algorithm | ? |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Algorithm run time (s)** | | |
| **Input file** | **Modified brute force** | **Nearest neighbour** | **Genetic** |
| AISearchfile012.txt | 12 |  |  |
| AISearchfile017.txt | 17 |  |  |
| AISearchfile021.txt | 21 |  |  |
| AISearchfile026.txt | 26 |  |  |
| AISearchfile042.txt | 42 |  |  |
| AISearchfile048.txt | 48 |  |  |
| AISearchfile058.txt | 58 |  |  |
| AISearchfile175.txt | 175 |  |  |
| AISearchfile180.txt | 180 |  |  |
| AISearchfile535.txt | 535 | ~3900 |  |

**Discussion**

The first algorithm I chose to implement, due to both it’s simplicity of implementation (appropriate for getting

A true brute force, given the required resources, guarantees an optimal solution and is probably the simplest and quickest algorithm to program, so xxxx

My modified brute force