## **Resource Constrained Shortest Path Problem**

Joshua Broeksteeg, Monash Winter Research 2023

**Problem Description**

The input consists of a set of N locations, the first of which represents a depot and the others representing houses. Each location contains a position on the Cartesian Plane, opening and closing times, an unload time, a load and a bias. Additionally, each input contains a maximum load. The task is to find a cycle of minimum cost that satisfying the following conditions:

* ]
* Has either not been visited in the path, or is the depot and is the final node of the path.

When traversing an edge, the variables are updated following:

Where

* Are initially 0
* Is the Euclidean distance
* Are all non-negative
* Is a real number.

**Naive Algorithm**

The naive solution to this problem involves traversing the graph of possible States with a basic graph search algorithm. In this case, Breadth First Search is clearly suited as there are an exponential number of states and so exploring them in a breadth first manner is heuristically more likely to terminate. While could be stored as a list, a trivial optimisation is to replace it with a bitset.

In order to dramatically reduce the number of explored states, we will define State to be strictly better than (dominate) State if:

Therefore, if we find an efficient way to check whether it is worth exploring a State further the search space will be significantly reduced. The naive method of accomplishing this is simply storing every state that has reached a given node, and to only push a state to the BFS queue if no other dominating states are already in it. While other methods of checking dominance were explored, the high proportion of *potentially* dominating states (around 40% have less time, load and cost than the current state on average) means that any overhead imposed by alternate algorithms and data structures cause it to be slower than naive checking. In practice, the use of an ND-Tree and similar tree data structures caused up to around 10 times slower runtime, and as such they were ruled as inapplicable to this problem. Therefore, the research became around trying alternate traversal orders.

**Time Optimisation**

The motivation for ordering the traversal by time was the assumption that states with earlier time would be “better” than those of later time as in the average case they would have less load, cost and nodes visited. Therefore, exploring these nodes early would in theory have a higher chance of causing other suboptimal states to be dominated and thus further reduce the search space.

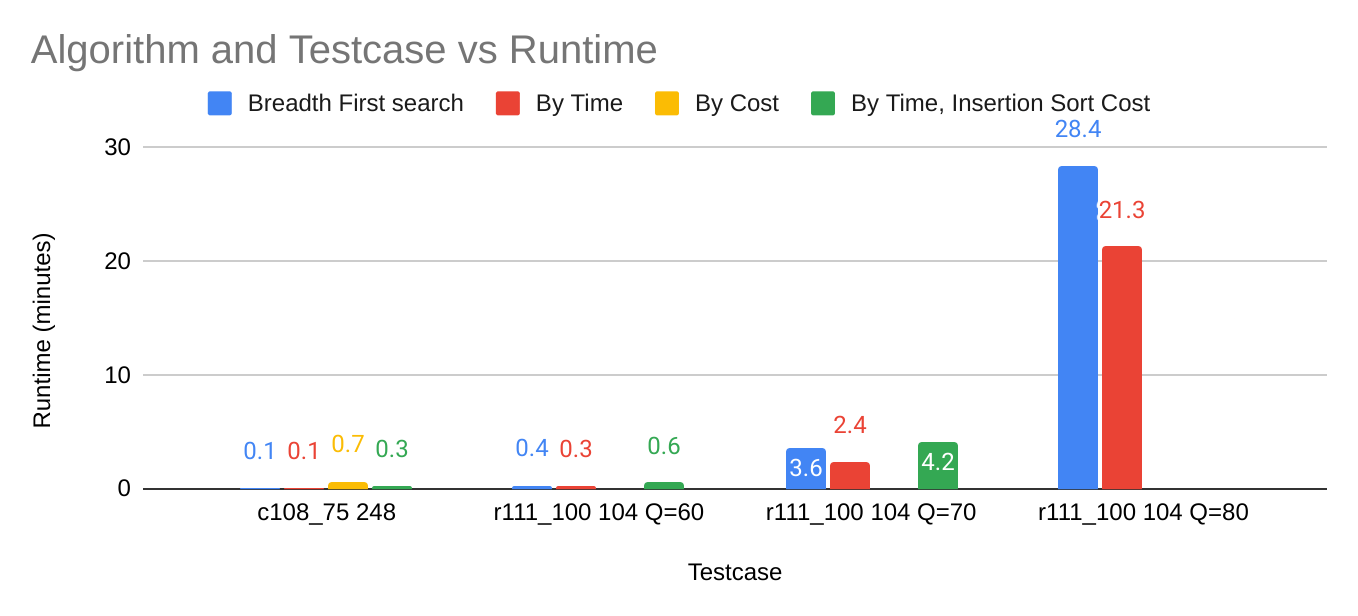
This is implemented by simply replacing the queue with a priority queue, traversing the states by time in ascending order. Since time is strictly increasing, the assumption can be made that states will be traversed in ascending order of time (but **not** that all the previous states that have been stored for dominance checks will have lower time).

**Other Strategies**

Two other strategies were tested: ordering by cost in ascending order, and extending this by storing the previously visited states in ascending order of time. The motivation for this ordering was that the goal is to find the paths of least cost, and therefore exploring them first would find them quickly and dominate other states. Furthermore, the motivation for ordering the previously visited states is that a suffix of them do not have to be checked, as the time will be too high to dominate.

The implementation for these is similar to the previous ordering, adding a priority queue and using insertion sort when inserting new states to the list of visited ones (this was significantly faster than using an ordered data structure).

**Results**



As seen in the results, the only significant increase in speed was achieved by processing the states in increasing order of time, resulting in about a 25% decrease in run-time. Note that missing bars on the graph indicate that the algorithm either did not terminate or ran out of memory of that testcase. Therefore, it can also be seen that while there is motivation for ordering by cost and using insertion sort in practice it leads to significantly slower run-time. In conclusion, the results suggest that for this problem the naive algorithm is actually one of the best and show that simple optimizations such as changing ordering can grant big decreases in run-time where complex data structures and algorithms fail.

**Further Research**

An interesting observation is that if the states are processed in increasing order of time, every node is in every other node’s adjacency list for a single contiguous period of time during the run-time of the program. Therefore, there is the possibility of further optimization by precomputing these to avoid unnecessary adjacency list checks. However, in practice cache locality made this significantly hard to achieve. Further research could involve developing an efficient way to precompute this.

**Source Code**

The source code for this research is available at [here](https://github.com/jbroeksteeg0/WinterResearch). While this repo does not contain the input data, anyone reading this probably has access to it.