Algorithm Engineering:

Alchemy Challenge

Vrije Universiteit Amsterdam
Dimitri Diomaiuta - Hugo Brandão - Michael Olivari

Introduction

Background

In mathematics and computer science, an algorithm is a self-contained step-by-step set of operations to be performed in order to make calculations, process data, and/or evaluate automated reasoning tasks. Algorithms can be seen as a recipe; they can be as simple as incrementing a given variable by 1 or as complex as processing the human genome. In computer science specifically, the algorithm is the fundamental process in which an engineer can take a seemingly impossible problem as input, run it through the algorithm, and obtain a seemingly unobtainable solution which by hand could take millennia but through the power of the algorithm only mere moments. While this is the most desired outcome, a solution will only ever be as good as the algorithm it was generated by. In this sense, algorithmic engineering is the most fundamental aspect to computer science and it is determined by 2 main factors: correctness and efficiency.

For most problems, correctness is rather trivial. Generally speaking, a typical problem has many algorithms which it can be taken in as the input and these many algorithms will output the same correct solution every time, with varying degrees of complexity. For these problems, efficiency is key and if the correct answer can be computed in polynomial time, then the problem belongs to the set of **P** problems. However many problems do exist where there is no known algorithm in which the correct answer can be found in polynomial, but it can be verified in polynomial time. These classes belong to the set **NP** problems, with the most computationally difficult of these problems belonging to a subset of the **NP** class known as **NP-Hard**.

With an **NP-Hard** problem, most likely the only manner in which to guarantee to achieve the correct solution to the problem is through a complete enumeration of every possible solution in its solution set, or rather brute-forcing the solution. While in some cases this may be the only option an engineer has, for most cases the performance is too unacceptable. It is up to the engineer to determine what is most important for the outcome and their needs; a solution which is correct / nearly correct (depending on heuristical choices) but could possibly not be the most correct calculated with a reasonable performance, or guaranteeing the correct answer every time but suffering a huge performance penalty. The focus of this paper will be on solving the VU

alchemy challenge, an NP-Hard problem similar to the most well known NP-hard problem, the traveling salesman.

The Problem

The VU alchemy challenge consists of taking a set of compounds and a set of machines, with each machine being able to synthesize one compound into another with a given cost, and calculate the minimal total cost of the set of machines such that every compound can be synthesised from any given compound in order to create gold.

On first inspection, this seems to be a rather trivial problem; simply take each of the cheapest machines so that you can make each compound necessary, sum the costs and output the total sum. However, if you consider each machine an edge in a graph between 2 nodes (see: compounds), it's easy to imagine a scenario where the direct route (see: machine) from 1 compound to another is more expensive than traveling from the same initial compound to a third compound and finally to the original final compound. This clearly resembles the traveling salesman problem or vertex cover, with a few caveats. In this interpretation of the problem the graph is a directed, cyclic, near-but-incomplete graph whereas the traditional TSP problem is typically depicted in a complete, undirected graph. An example of the type of graph for the alchemy challenge is found in *Figure 1.1* below.

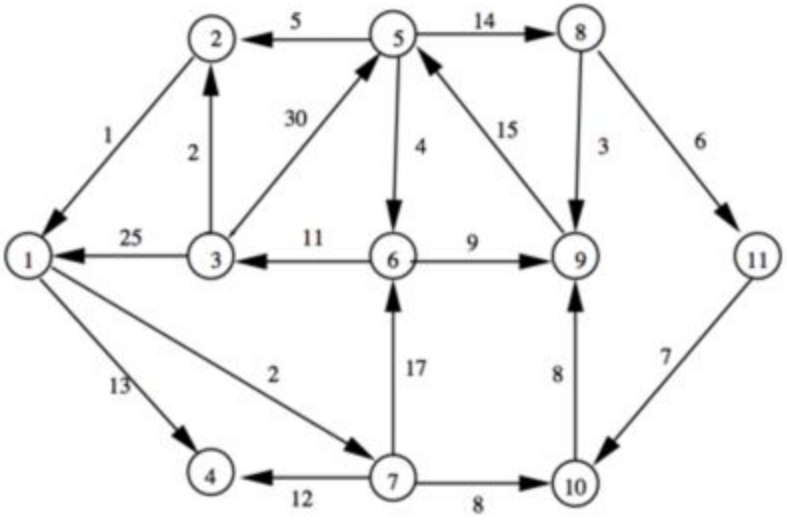

Figure 1.1 Graph Example

As this wasn't a typical TSP problem, unfortunately a lot of the known algorithms would not be usable, or would have to be heavily modified, in the case of the alchemy challenge. Knowing this, however, proved to be an excellent starting point in developing an algorithm to solve the challenge.

Previous Works

In order to better understand the challenge main aspect of the challenge, graph traversal, the team referenced a lot of material over adjacency matrices, Hamiltonian paths, Hamiltonian circuits, shortest path traversal in a graph, and most importantly known efficient algorithms for the traveling salesman problem. Skiena's Algorithm Design Manual was the main source the team used in formulating a better understand of the various algorithmic heuristics used in which TSP has been "solved", from the simple greedy approach of nearest neighbor, the dynamic programming paradigm, to backtracking, to more complex solutions such as simulated annealing.

Contribution

Algorithm Design

As the alchemy challenge is in NP-Hard, the first point of contention for the team was deciding on which was most important: guaranteeing correctness or reducing temporal complexity. This choice would ultimately define the heuristics which would be used in the final algorithm constructed.

On one hand, the main point in attempting to generate a solution for such a problem is to generate a solution which is as accurate as possible. Finding a solution is ok, finding the best solution is optimal. However, in the case of the alchemy challenge, much like TSP, guaranteeing the best solution is found every time would result in an algorithm that effectively runs in worst case complexity, a brute force algorithm. Brute forcing would indeed solve our problem, and on small scale scenarios could be ideal, however on a large scale the performance drain is unacceptable.

Ultimately it was decided to emphasize the reduction of the temporal complexity of the algorithm against attempting to guarantee to find the best solution, as a near best solution seemed acceptable to the team. Looking at it from an economical point of view, it seemed more suitable for the lab to begin producing gold in a shorter amount of time while still maintaining a near minimal, even if not the lowest, overhead for the costs of the machines. If the lab had to spend so much time invested in picking machines rather than outputting gold, the money is lost anyway in the form of lost profit from the non-generated gold.

Implementation and Heuristics

After considering many different approaches, the team determined that utilizing the recursive paradigm of backtracking in combination with a nearest neighbor greedy heuristic was suitable for the algorithm design formulated. The main recursive loop works as follows:

- From the starting compound, choose cheapest machine to next compound. The total tour cost is initially the cost of this machine.
- Each subsequent compound chooses the cheapest machine, its nearest neighbor, to another unvisited compound. Each machine choosen adds its cost to the total tour cost.
- The final compound chooses the machine back to the original starting compound. If no machine exists, the total tour cost is set to infinity and returned up in the recursion. Otherwise this machine's cost is added and total cost is returned up in the recursion.
- The returned cost is compared to the previously found cost, and is reset to the returned cost if it is lower than the previously found cost.

Limitations

The main drawback of this implementation of the algorithm is that the best solution is not always found due to the greedy heuristic of choosing the cheapest path from one compound to the next each time. As discussed in the introduction, a greedy choice ignores the possibility of there being a path between the two nodes via another node which is cheaper than the direct path as there is no triangular equality, Path(a,b) ≤ Path(a,c) + Path(c,b), guaranteed in the challenge's graph. A smarter approximation heuristic, such as a modified Christofides' algorithm, would likely provide better results than the nearest neighbor greedy selection does. This however would increase the temporal complexity of the algorithm designed by the team as we value speed (slightly) over finding the best solution every time.

Results

Recorded below in *Table 1.1* are the results of running multiple input files through our algorithm.

Input File	Expected Output ($)	Actual Output ($)	Performance Difference (%)
06_0.IN	15083	15083	0%
06_8.IN	14358	14358	0%
08_0.IN	20254	20254	0%
08_3.IN	19033	21120	10.9652%
12_3.IN	27458	28544	3.9551%
16_0.IN	37782	37782	0%
20_0.IN	49443	49443	0%

20_8.IN	41989	44831	6.7684%
		Average Performance Difference	2.7108%

From this limited testing, it is apparent that the algorithm does not guarantee the best solution is generated each time, however with an average difference of 2.7% from the expected output shows that the greedy heuristic used is an acceptable choice when considering the temporal complexity of the created algorithm vs. a brute force algorithm to guarantee the correct solution is generated each time. From these results, the original goal of the team to design an efficient algorithm to find a best or near best solution was achieved.

Conclusion

There are multiple algorithmic approaches that could have been taken to solve the VU alchemy challenge. The solution the team found is just one which performs quickly but other heuristics could have been chosen to further minimize the output difference between the generated output of the algorithm and the expected output. From the testing done by the team, one could feel confident that this algorithm is an efficient solution to the challenge, however as the input scales so too does the possibility for imperfect solutions. Thus, a more refined approach likely would be necessary on graphs with a high amount of nodes and density. All in all, an effective algorithm was developed with quite acceptable results, thus the backtracking greedy algorithm can be considered if the lab accepts and agrees with the algorithm design principles the team established in this paper; a fast good solution is better than a slow perfect solution.