Algorithm Engineering Challenge Report

Peter Atkinson - pan450 - 2555678
Chris Esterhuyse - cee600 - 2553295

September 12, 2019

1 Introduction

The task is to develop a program that can reliably give a good solution for the following problem: Given a set of compounds, and a set of machines, where each machine is able to transform one input compound into one output compound, and has an associated once-off cost, what is the optimal subset of machines that can be purchased such that all compounds can be created (using machines) from any one other compound at minimal cost?

One can view the problem in terms of a directed graph G, where every vertex is a compound, and every machine is an edge with a weight. The task is to find the minimum-weight subgraph G' of G where G' is *strongly connected*.

An exact solution to the problem should take into consideration the following important distinctions between classical ATSP and our problem:

1. One may traverse edges and visit vertices multiple times.
2. An edge only contributes to the cost the first time it is traversed (more on this point in section 3.5 - Implementation Issues and section 5 - Discussion).

The standard ATSP solution will not suffice, as it prohibits visiting vertices more than once, which could be beneficial. To sidestep this issue, the graph is altered such that an ATSP solution on the altered graph will result in a solution that allows for multiple edge traversals (distinction #1).

The finished algorithm was designed with the above points in mind, by treating the overall problem as a sequence of subproblems, as described in section 3.2 - Algorithm & Design).

Ultimately, the finished algorithm does not completely succeed in compensating for difference #2 from classical ATSP, resulting in suboptimal solutions for certain input problems (see 3.5 - Implementation Issues). While the algorithm fails to guarantee exactness, it returns exact solutions in *most* test cases (see 4 - Computational Results) and reaches a solution quickly in *all* test cases. This report outlines the development process of the inexact solution and includes some deliberations on possible steps toward a fast exact solution.

2 Previous work

The given task would likely have been difficult to approach had we not had prior exposure to the traveling salesman problem and its asymmetric variant. It becomes easier to understand with the support of related materials. Specifically, surveying existing literature about graph traversal problems, successes with related problems could inform the approach to the given problem. A number of materials that proved beneficial to the development of the solution are presented below.

1. Guten and Punnen's book The Traveling Salesman Problem and Its Variations details a number of TSP variants that require modified approaches to the classical TSP. One variant they mention allows for a solution that can involve visiting a vertex more than once. In some cases, this loosened constraint allows for cheaper circuits. Gutin and Punnen touch on the TSP with multiple visits (TSPM) problem, mentioning that it can be converted to a regular instance of the TSP "by replacing edge costs with shortest path distances in [the graph]" [3].

2. User cat_baxter responded to a question on StackExchange [7] about genetic algorithm solutions to the TSP with a moderately understandable exact solution based on the original branch and bound work of Little, Murty, Sweeney and Karel. Their code can be found at: `http://www3.cs.stonybrook.edu/~algorith/implement/syslo/distrib/processed/babtsp.p` [5].

3. TSPLIB is a useful resource for those interested in TSP solvers. The library contains a number of interesting test cases designed not only to stress solvers under development, but also to allow for universally comparable benchmark testing.

4. Little et al laid out the original branch-and-bound ATSP algorithm [5] in 1963 and it is still relevant today. Making use of this paradigm with a reduction heuristic, this algorithm is fairly effective. More advanced algorithms have been developed since [4].

5. Floyd and Warshall detail an algorithm for shortest path using an adjacency matrix [2]. Although not ideal for all scenarios, this polynomial-time algorithm is a simple, intuitive way to determine the minimal costs between each pair of nodes in a graph, and to be able to reconstruct the path that uses that minimal cost as needed.

3 Own contribution

3.1 Intuition behind the algorithm

Every node is a compound, and every machine is an edge. The problem asks for an algorithm that finds a minimum-weight edge set that visits all nodes

(compounds). This is similar to the asymmetric traveling salesman problem, but without the restriction that a node can only be visited once.

The advantage that revisiting nodes gives can be illustrated as follows:

Let there be an arbitrary problem instance which corresponds to a graph shaped like the wheel of a bicycle. In the center is the hub node, with several other nodes on the rim of the wheel. Every spoke represents two edges connecting the hub node to a node on the rim (one there and one back). Each rim node has 2 outgoing edges and 2 incoming edges from its two adjacent rim nodes (in addition to the spoke edges) which creates a path in a ring shape around the rim. Traversing the spokes is very cheap (say, a cost of 1 per edge) and traversing the rim of the wheel is very expensive (say, a cost of 1000 per edge). The best way to visit every node is to make "there-and-back" trips to each node via the hub. The asymmetric traveling salesman problem prohibits this, however, because the hub node can only be visited once in the ATSP scenario. With the ability to revisit nodes, the "there-and-back" circuit becomes possible.

For this reason, the ATSP step is preceded by a pre-processing step to translate the problem instance into terms ATSP can understand. The output of the ATSP module is then post-processed again into sets of machines that are the results of the paths that ATSP interpreted as edges. In the bicycle wheel example, a short path between two rim nodes through the hub would be represented as a direct connection between them, and then at the end once again translated into a composition of two distinct spokes that go through the hub node.

3.2 Algorithm & Design

The algorithm, currently implemented in Python, takes a number of steps to ultimately find an inexpensive subset of machines as described by the problem. These steps can be seen as smaller sub-problems that interface their outputs to the next step until the ultimate solution is found. This modular approach makes it possible to modify modules (or even select between several versions of the same module based on the attributes of the input) without affecting the functioning of any other step in the algorithm. The steps are detailed below:

1. Step 1: machine set input ⇒ graph

   The input is parsed and an internal data structure is built using nested hashmaps (Python dictionaries) to model the problem space and get fast lookups. In this step, machines that will never be used are discarded immediately and remaining machines are mapped from their provided names to a number that corresponds to an index in an adjacency matrix.

2. Step 2: graph ⇒ adjacency matrix

   The graph is transformed into an adjacency matrix by iterating through the nested dictionary and the compound maps mentioned in the previous step. In this form, the graph is in a common representation that is easy

to work with.

3. Step 3: adjacency matrix ⇒ shortest path matrix

   In order to accommodate the added freedom given by being able to revisit nodes, we build a new adjacency matrix, pretending that paths exist between every set of nodes. Using a shortest-path algorithm (currently Floyd-Warshall [2]), another matrix is created to represent the shortest paths between vertices. At the same time, for the purposes of actual path reconstruction, another matrix is created to store the steps required for these paths (referred to as the how_to_get_there matrix in the code).

4. Step 4: shortest path matrix ⇒ ATSP tour

   This step runs the an adapted version of Little et al's ATSP branch and bound algorithm on the shortest-path matrix. The output's path is in terms of direct connections between vertices, whether or not direct connections between those two vertices exist.

5. Step 5: ATSP tour ⇒ machine subset output

   Finally, the ATSP output paths are translated to actual paths in the original graph using the reconstruction matrix and that set of pairs of connections is then translated back into a set machines using the compound ID maps from step 1. Once all the paths have been reconstructed and the machines have been added to the solution set, the solution set and the cost of its corresponding machines are returned.

3.3 Relation to previous work

The finished algorithm makes use of most of the materials found through the literature study in some way. The most important of these was Gutin and Punnen's [3] note on creating a shortest-path matrix in order to solve the multiple-visits variant of TSP. It seems obvious in retrospect, but at the beginning it was the major sticking point.

Little, Murty, Sweeney and Karel, the originators of the term "branch and bound", designed their algorithm for the ATSP [5]. More advanced algorithms exist, but theirs is regarded as easy to understand and illustrate by mathematician Dieter Jungnickel as stated in his book Graphs, Networks, and Algorithms [4, Chapter 15.8], which is suitable to the educational objective of this challenge.

Upon reading Gutin and Punnen's note [3, p 7] and StackExchange user cat_baxter's [7] Python adaptation of the original Pascal code, construction of the prototype began, assembling the pieces in such a way that input and output of the final product suited the specifications of the challenge.

3.4 Engineering

The modular nature of the algorithm provides the benefit of being able to easily measure the performance of each of the modules, and as a consequence, easily measure how changes to one part affect both that particular module's performance, and the performance of the program as a whole. The implementation was done in a way to accommodate measurement and further improvement with minimal pain involved.

1. The input specification does not guarantee that machine names be sequential nor spaced at regular intervals. To accommodate arbitrary inputs and minimize speed penalties, machine and compound names are stored in hashmaps that map the names to array indices for fast lookup in a matrix. This mapping is done both ways: one after file reading to map to matrix indices, and the other way to recover the original input names at the end. The algorithm is currently implemented in Python, so the most evident choice was the use of built-in dictionaries, for their simplicity and respectable speed.

2. The data being read from the input file is stored in nested dictionaries, where the top level contains all the compounds as keys, and those compound keys map to all the other compounds reachable from that one with the associated machine and its cost. This dictionary is transformed into an adjacency matrix for further processing. Initially storing the information in dictionaries offers the benefit of fast lookups as we read the input file, ignoring expensive machines whose job is already done by a less expensive machine. It also facilitates the creation of the ID maps and of the adjacency matrix. Simply put, it's a structure that's ideally suited for transformation into other structures.

3. The pre-processing of the adjacency matrix into a shortest-path matrix is done using the Floyd-Warshall algorithm. Despite this not being the fastest shortest path algorithm, the contribution of this step to the time for the algorithm was such that improvements in this part of the algorithm were deemed low-priority. The choice of this algorithm was based on its ease and simplicity. To facilitate future expansion and optimization, this step was written to be modular so that it might easily be changed, or even decided on dynamically, based on the specific properties of the input. For example, Johnson's algorithm in $O(|V|^2 \log |V| + |V||E|)$ is faster in sparse graphs[1].

4. For the ATSP implementation as first described by Little et al [5], branch-and-bound is used to greatly improve the performance of the ATSP step. Coupled with the reduction heuristic, considerable time is saved and the algorithm becomes feasible for a more respectable number of compounds.

   Even though the worst case time complexity of the ATSP step has factorial cost, with an effective heuristic, in practice it tends to be significantly

faster. Another strength of the particular implementation of ATSP solver, is that each branch of the tree takes linear space. All information directly related to individual matrix elements are kept in-place. Not only does this help with space in general, but avoiding the creation and iteration over numerous copies of matrices also ends up saving time.

The heuristic used for the bounding step of the ATSP algorithm seeks to identify a *best edge* to traverse at each branch of the tree. Intuitively: This heuristic relies on the fact that in a tour, all vertices will have to be entered and exited once. Finding edges that are comparatively cheap to the alternatives is a good way of guessing edges that are likely to minimize the cost of the tour. This is done by comparing the out-costs and in-costs of the vertices involved. Entering a vertex at low cost will prohibit the vertex from being entered later at a high cost.

5. To try to maximize what the Python language can provide in terms of performance, the code is compiled rather than interpreted at run-time. This minor change was noticed to decrease each execution time by a small amount.

3.5 Implementation issues

In our zeal to create a working prototype quickly, we cut several corners in the research and development phases of this challenge.

The increased cognitive load as the prototype came closer to completion as a consequence of cutting corners in the coding process made it such that small problems were difficult to solve. Two notable such problems were:

1. Reconstructing the actual path between two compounds using the solution returned by the ATSP step and the how_to_get_there (path reconstruction) matrix.

2. Translating pairs of internally-used matrix indices back to machine names.

Both of these problems stemmed from the use of Python's in-line list comprehension feature in an excessively-nested manner with a bewildering amount of if/else conditions and from the fact that development the system's components were developed side-by-side as opposed to sequentially, which gave rise to sloppily-made patches made to enable continued development.

Once the hastily-made prototype was completed, we ran several tests with inputs with known solutions and assumed a false sense of security when they all returned exact solutions. As it turns out, these tests were not exhaustive. Upon proofreading this document, it occurred to us that shortest path matrix + ATSP may in some cases result in either machines being purchased more than once in a solution (a problem that is easy to solve with our current implementation) or the ATSP module making a 'wrong' choice (since ATSP counts repeat traversals, whereas this problem does not). Upon further investigation, it was discovered that this problem appeared in a Facebook puzzle challenge

named "Facebull" (`http://web.archive.org/web/20100108032420/http://www.facebook.com/careers/puzzles.php?puzzle_id=1`), and is formally named the minimum spanning strong sub(di)graph problem. The popularity of this puzzle made it such that a trove of inputs could be found (ex: `https://github.com/mlbright/puzzles/tree/master/facebull/tests`). Of all the inputs we tested, only 5 of 94 resulted in inexact solutions. 88 of 94 resulted in an exact solution and the remaining one did not finish in a reasonable amount of time. Potential approaches to a reasonably fast exact solution are discussed in section 5.2 - Future development.

With a slower, more organized approach, all of these issues could have been avoided.

4 Computational results

With a functioning algorithm and implementation, it became possible to run benchmark tests and observe the speed and quality of the program's response individual instances of the problem. The five logical steps of the algorithm were timed individually for each test input in order to gain a deeper understanding of which steps were contributing the most to the running time. The tests were run with input provided by one of the administrators of the course.

4.1 Easy problems

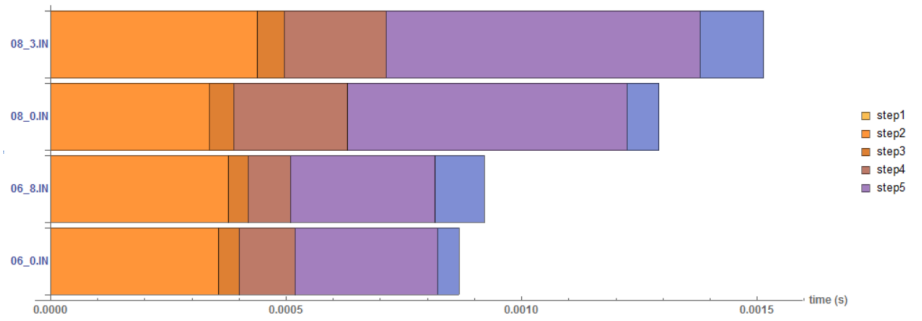

Figure 1: Time taken for all 5 steps of small vertex-count graph inputs

For four smallest inputs, there were a small number of machines and compounds. A considerable amount of time was spent on step 1 (machine set input $\Rightarrow$ graph), and step 4 (ATSP). With such small graphs, I/O taking a large chunk of the total time should come as no surprise. ATSP being a contributing factor to the runtime is simply a product of it doing the bulk of the work in any case. For these easier problems no parts of the algorithm are really being stressed yet, and the runtimes are still very short.

4.2 Adding intermediate problems

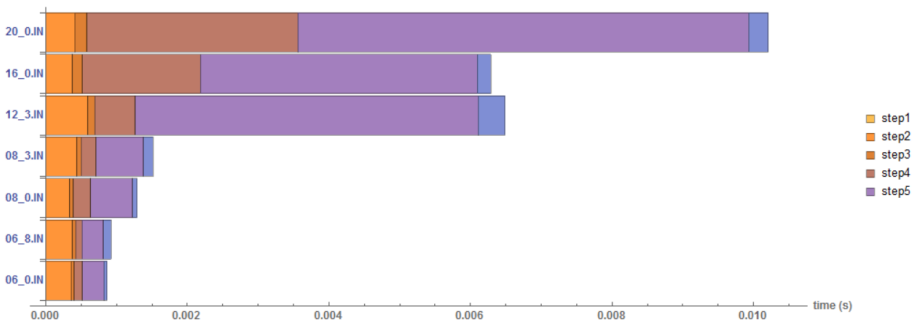

Figure 2: Time taken for all 5 steps of small and medium vertex-count graph inputs

Adding the three next larger inputs, the ATSP step begins to overshadow the other steps. Naturally, this is due to it having the largest worst-case time complexity, so this is to be expected. With some inputs, step 3 (adjacency matrix $\Rightarrow$ shortest path matrix) is also taking strain. Again, this is because it is also expected to be relatively expensive, in the order of $O(n^3)$.

4.3 All provided problems

With the addition of the the larger problems, the easier problems' times are barely visible. The final two tests show that with large numbers of vertices, the time contribution of steps 1,2 and 5 become increasingly negligible. The input 40_0 serves as a notable demonstration of the effectiveness of the branch and bound technique. The input consists of 40 machines that form a ring-shaped graph, so one would expect the solution to be easy to find with a sufficiently effective algorithm. The Floyd-Warshall algorithm's ($O(n^3)$ where n is the number of compounds in the input) contribution to the total time is visibly comparable to that of the ATSP part of the solution (where a brute-force solution would be in $O(n!)$) due to the number of branches in the search that can be pruned early-on.

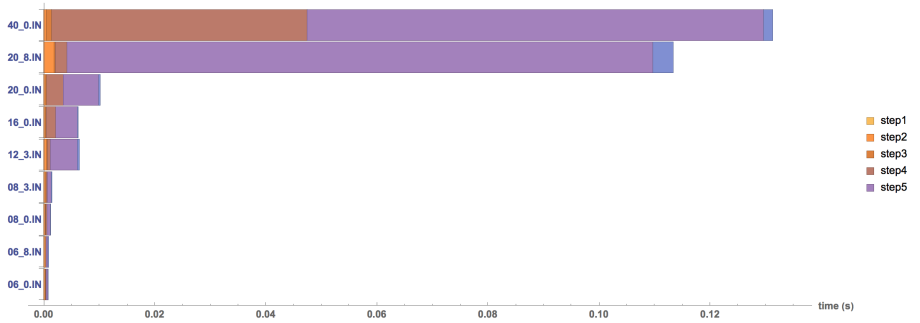

Figure 3: Time taken for all 5 steps of small, medium, and large vertex-count graph inputs

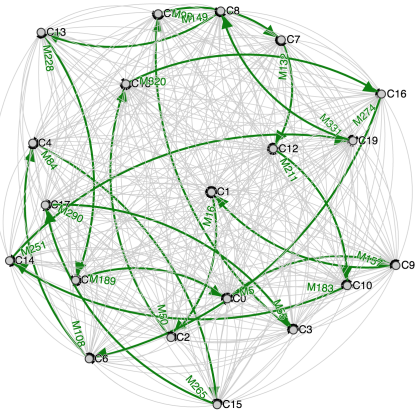

Figure 4: Visualized solution for 20_8.IN, a graph with 20 vertices of large degree

5 Discussion

5.1 Observation of interest

Throughout the development of the algorithm used in this paper, the solutions found for test cases were consistently optimal with generally respectable runtimes. Very late in the process, some test cases were found in which the algorithm does *not* find the optimal solution. Figure 5 shows an example of such an input.

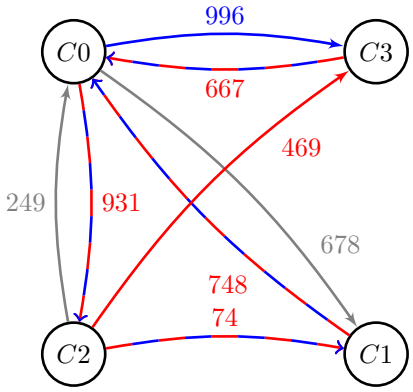

Figure 5: Example input for sub-optimal algorithm output. Red edges denote the optimal solution and blue edges denote our solution.

The ATSP-algorithm determines that the 996-cost machine is the cheapest way from C0 to C3, as C0 to C3 via C2 costs $931 + 469$. The ATSP branch-and-bound is unaware that costs between compounds are potentially affected by purchasing machines to traverse between other compounds. In our example, by purchasing the 931-cost machine from C0 to C2, the cost from C0 to C3 via C2 becomes substantially cheaper, and becomes preferable to the 996-cost machine directly connecting C0 to C2.

5.2 Future development

StackOverflow user cat_baxter contributed to a discussion on solving graph problems with strongly connected components and explains a potential method of detecting when the ATSP solution is inexact and how to proceed to an exact solution [6]. Further research would need to be done regarding the cited Hungarian method to solve the Assignment Problem, but this approach seems to hold promise.

Other steps might also be taken to swap in more efficient algorithms for the sub-components when the problem instance is deemed to be the weakness of that algorithm. For example, using Johnson's algorithm for shortest path in the case of sparse graphs rather than Floyd-Warshall. As the runtime is a function of the problem instance, finding a smart way of classifying problem cases could be a whole field of exploration.

As Python is not known for its speed, rewriting the algorithm in more lean language (like C) would also potentially be of benefit depending on the application. In addition, to really squeeze the best times out of the program, further optimizations such as splitting the graph into subproblems where possible and taking multi-threaded approach might be considered.

References

[1] Paul E Black. *Dictionary of algorithms and data structures*. National Institute of Standards and Technology, 2004.

[2] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[3] G. Gutin and A.P. Punnen. *The Traveling Salesman Problem and Its Variations*. Combinatorial Optimization. Springer US, 2007.

[4] Dieter Jungnickel. *Graphs, Networks and Algorithms*. Springer Publishing Company, Incorporated, 2nd edition, 2005.

[5] John DC Little, Katta G Murty, Dura W Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations research*, 11(6):972–989, 1963.

[6] user cat_baxter. Answer to: Minimum cost strongly connected graph. `http://stackoverflow.com/questions/1536067/minimum-cost-strongly-connected-digraph/10835399#10835399`, 2012-05-12. Accessed 2017-01-02.

[7] user cat_baxter. Answer to: Tackling the tsp with a genetic algorithm. `http://www.codereview.stackexchange.com/a/13340`, 2012-07-05. Accessed 2016-12-10.