Solving Small TSP Instances

You'll Clean That Up Before You Leave

Shawn Brunsting, Ian Dimock, Yuguang Zhang
University of Waterloo

April 6, 2015

1 Introduction

Many algorithms have been developed for the travelling salesman problem. Asymptotic behaviour can be analysed for these methods, but methods that are superior asymptotically may turn out to be slower for small problems due to large overhead costs. For example, enumerating all possible tours may take $O(n!)$ operations, but for a problem with only $n = 5$ cities, could this be faster than the $O(n^2 2^n)$ Bellman-Held-Karp algorithm? This project explores such questions by implementing a few common algorithms for the travelling salesman problem, then comparing their run time for problems of varying sizes and types.

Section 2 outlines the algorithms that were implemented for this project, including some implementation details and optimizations. Section 3.1 describes the types of problems we ran the algorithms over. Section 3.2 describes the environment in which we tested the algorithms. Sections 4 and 5 present and discuss the results. Section 6 discusses some future work. Finally, Section 7 summarizes the project. The raw timing data as well as our code are contained in the appendix.

2 Methods

For our project, we implemented several methods for solving the TSP. They are enumeration with pruning, Held-Karp 1-trees, Bellman-Held-Karp dynamic programming, and subtour branch and cut. Bellman-Held-Karp dynamic programming has the best proven worst case run time for solving the TSP.

2.1 Enumeration

The enumeration method for solving the TSP generates all possible tours and iterates through them. The possible tours are generated through a recursive call that swaps the positions of cities in the tour. In each recursive call, the number of cities to permute is reduced by one. When there are no more cities to permute, the recursive function compares the current tour length against the best one found, saving it if it is better than the previous value.

To save time, three simple optimizations are implemented in our version. First, instead of computing the tour length by adding up the edges at the base case of recursive calls, the recursive calls keep track of the tour length so far. Second, if the tour length so far is greater than the best known tour, the search path is pruned. Third, a macro is used to swap cities in the tour saving two function calls before and after permuting the order of cities. These simple optimizations give this $O(n!)$ algorithm better performance than more sophisticated algorithms on the smallest instances [2].

2.2 Held-Karp 1-Trees

In a pair of articles published in 1970 and 1971 [5, 6] Michael Held and Richard Karp proposed a method for constructing lower bounds for the traveling-salesman problem using *1-Trees*. They described an ascent algorithm using these lower bounds which we have implemented.

If a graph G has nodes labelled $1, 2, \ldots, n$, then a 1-tree of the graph is formed by taking the node set and a subset of the edges such that the edges incident only with nodes $2, 3, \ldots n$ form a tree, with 2 additional edges incident with node 1. We can see that with this definition, a tour of the graph is a 1-tree.

While the TSP problem looks for a tour of minimum cost, we can similarly define the notion of a 1-tree of minimum cost (for this purpose the 1-tree need not be rooted at node 1). The core of Held and Karp's algorithm comes from the realization of what happens to the minimum cost tour and the minimum cost 1-tree when we alter the costs of the edges in the graph in a particular way.

Consider a vector $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots, \pi_n)$. We consider what would happen if we update the costs of the edges $c_{i,j}$ as follows:

$$c_{i,j} \leftarrow c_{i,j} + \pi_i + \pi_j$$

We note that since each node must have degree two in any tour, this modification of the edge weights changes the optimal tour cost by $2\sum_{i}^{n} \pi_i$, but does not change the tour itself. The minimum cost 1-tree however is not guaranteed to remain the same on the modified graph.

Held and Karp showed that from these minimum cost 1-trees we can produce lower bounds for the TSP of the from:

$$C^* \geq \sum_{(i,j)\in K^*(\boldsymbol{\pi})} c_{i,j} + \sum_{i} (2 - d_i)\pi_i = w(\boldsymbol{\pi})$$

Where $K^*(\boldsymbol{\pi})$ is the minimum cost 1-Tree defined on the graph with edges wieghts modified by $\boldsymbol{\pi}$ and d_i is the degree of node i in this 1-Tree.

We note that because $\boldsymbol{\pi}$ defines a minimum cost 1-tree, we can consider our lower bounds on C^* - the cost of the optimal tour - as simply a function of $\boldsymbol{\pi}$, namely $w(\boldsymbol{\pi})$. By modifying $\boldsymbol{\pi}$ the ascent

algorithm attempts to produce minimum cost 1-trees that look more and more like tours, increasing the lower bound until the optimal tour is found.

The algorithm we implemented consists of three main components. The first is the computation of the minimum cost 1-trees. This is a fairly simple process which involves computing a minimum spanning tree on all but one node and then adding that node's two cheapest edges to the MST to produce a 1-tree. By repeating this process on each node, one of the 1-trees generated is guaranteed to be of minimum cost. The second component is a simple iterative algorithm to modify $\boldsymbol{\pi}$. This is done by increasing and decreasing π_i for nodes i in the 1-tree which have more or less than degree 2 respectively. The last component is a branching method. We used the technique used in homework 2 of forcing a particular edge into or out of the 1-trees. We kept our branching nodes in a priority queue so that as soon as we pop a 1-tree from our queue, and that is a tour, we know it is the optimal tour.

Over the development of the 1-tree algorithm, several optimizations were made. Many were programming details, but outlined here are a few more significant changes that together greatly increased performance. Firstly, we modified the Kruskal code developed for homework 1 to work on the subsets of our edge set, instead of building a subgraph every time we wanted to build an MST. Another optimization was to sort all the edges of the modified graph (modified weights) only once. In this way we can choose the two edges incident to our *ignored* node during our modified Kruskal's algorithm, and not need to re-sort when we consider the next node to ignore. Since we are branching on the smallest lower bound in our priority queue, we cannot declare a tour we've found optimal until it has been popped from the queue. We can however update our upper bound on the tour cost. This helps with the pruning that occurs during the $\boldsymbol{\pi}$ iterations. Lastly, a seemingly minor change was to update our maximum lower bound seen in the iterations of $\boldsymbol{\pi}$ when the bound was greater than *or equal* to our current maximum. Intuitively this makes sense because the iterations strive to produce 1-trees more resembling tours, so even if the bound itself is not increasing, we switch anyway because the 1-tree is hopefully more tour-like.

2.3 Bellman-Held-Karp Dynamic Programming

The Bellman-Held-Karp algorithm was discovered independently by both Richard Bellman [3] and Michael Held and Richard Karp [4] in 1962. It uses dynamic programming, which means it uses the solutions to smaller problems to create a solution to the larger problem.

2.3.1 Algorithm

The algorithm is best described with a recursive formula. First we pick some start city x. The choice of x does not matter, as long as it stays fixed throughout the algorithm. If t is a city, and S is a set of cities that includes t but not x, then we define *opt* to be our dynamic programming table where $opt(S, t)$ is the minimum cost of a path that starts at x, ends at t, and passes through every city in S exactly once (and does not pass through any cities not in S).

The base cases are simple to compute for this dynamic programming table. $opt(\{t\}, t)$ is simply the distance from x to t, since those are the only cities that can be in the path. We use $dist(x, t)$ to represent this distance. So we have

$$opt(\{t\}, t) = dist(x, t) \qquad \forall t$$

When S has more than one city, we define $opt(S, t)$ recursively. Let $q \neq t$ be some city in S. We let q be the second last city in the path, and we can find the optimal length by considering all possible values of q. We need the length of the shortest path to q, plus the distance from q to t:

$$opt(S, t) = \min_{q \in S, q \neq t} (opt(S \setminus \{t\}, q) + dist(q, t))$$

Finally, we need to calculate the optimal tour length. Following similar reasoning to the above formula, we know that the optimal tour must pass through every city in the problem, then return to x. So x is the last city, and we let t be the second last. If N is the set of all cities in the problem except for x, then the optimal tour length v^* is

$$v^* = \min_{t \in N} (opt(N, t) + dist(t, x))$$

The algorithm has a running time of $O(n^2 2^n)$, which gives it the best asymptotic bound that has been achieved so far [1].

2.3.2 Implementation

We tested 3 implementations of the Bellman-Held-Karp algorithm. First was our own implementation, which was written before looking at any other existing implementations. Writing this implementation ensured we had a solid understanding of the algorithm, along with some implementation ideas that were not influenced by what others had done.

The code for this first implementation can be found in `bhk.h` and `bhk.cpp`, and can be run using the `-m 3` flag with our main program. It has a few unique implementation ideas compared to the other two, the first of which was to store the dynamic programming table in a C++ map, rather than an array. This made it easier to implement, since we did not need to calculate the size of the table beforehand or figure out how to translate a subset into a position in that array. However, this method is expected to be slower than one which uses an array, since an array has $O(1)$ lookup time while the map would require $O(\log(s))$ time where s is the number of entries in the table.

Another unique idea was to generate all subsets of size k at once using a recursive method. This idea increases the amount of memory required, so it may not be ideal for large problems.

Finally, a subset of cities was initially represented as a vector of integers. Later, the code was changed to use an integer where each bit represented whether or not a particular city was included in the subset. This change was inspired by the second implementation, and although no rigorous testing was done, it seemed to significantly improve the running time.

The second implementation was taken directly from [1]. It is the recursive code that is first presented when that chapter describes the Bellman-Held-Karp algorithm. This code was put into `bhk2.h` and `bhk2.cpp`, and can be run using the `-m 5` flag with our main program.

The final implementation of the algorithm is also from [1]. The chapter describes various optimizations to improve the running time, which are all used in this implementation. The original C code for this implementation is in `tour_dp3.c`. This code was translated into C++ to fit with our program, so the version we used can be found in `bhk3.h` and `bhk3.cpp`. This implementation can be run with the `-m 6` flag.

2.4 Subtour Branch and Cut

The code we used for branch and cut was the code developed for homework 2, based on the board work and sample code seen in lectures. In particular, it uses subtour inequalities generated by finding disconnected components on the support graph generated from fractional LP solutions.

We made a few improvements from the code we submitted for homework 2. The first was branching on the *1-side* (force an edge to be included) before the *0-side* (forcing an edge to be excluded) in the branching procedure. Secondly, we improved our nearest neighbour tour initial upper bound by computing the nearest neighbour tour for every possible start node then taking the minimum of those. This change improved all methods which used pruning.

3 Problem Instances and Methods

3.1 Problems Instances

In order to compare the algorithms, we need to test them on travelling salesman problems of varying sizes. This section describes the properties of the problems that we used for testing. We implemented two ways of generating problems, but due to time constraints we only analyzed one of them.

A few properties were common to all test problems. Firstly, all the problems were symmetric. This means that if $dist(a, b)$ is the distance from city a to city b, then we have $dist(a, b) = dist(b, a)$. Secondly, all edge lengths were positive integers. This ensured that the algorithms would not be affected by floating point errors. Finally, all test problems were complete graphs, meaning there exists an edge between every pair of cities.

The method that we used to generate problems in this report first generated random points in a plane, which represented the locations of cities. These locations were truncated such that the x and y coordinates were integers. Furthermore, the problem generator ensured that each city location was unique. Finally, the length of an edge between two cities was calculated as the Euclidean distance between them, rounded to the nearest integer. This meant that all edge lengths were positive integers, and that the triangle inequality would hold. In other words, given three cities a, b, and c, we would have $dist(a, b) + dist(b, c) \geq dist(a, c)$.

The second method for generating problems (which is not analyzed in this report) did not guarantee the triangle inequality. Instead of generating random locations for the cities, random edge lengths were generated for every pair of cities. These edge lengths were positive integers. For some methods, such as enumeration and the Bellman-Held-Karp algorithm, we should expect similar performance results for problems generated by the two methods. Other algorithms, such as the 1-Tree method, can be sensitive to the distribution of edge lengths, so it was necessary to consider both types of problems.

Command line flags are provided to run these different problem-generating methods in our program. To run the first method, use the `-k` and `-b` flags to specify the number of cities and the grid size respectively. To run the second method (random edge lengths), use `-e` and `-l` to specify the number of cities and the maximum edge length respectively.

Machine	Factor	F value	p-value
1	Instance size	197.559	<2e-16
	Method	733.567	<2e-16
	Seed	2.523	8.13e-15
	Grid size	0.256	0.774
2	Instance size	198.096	<2e-16
	Method	664.248	<2e-16
	Seed	2.146	3.76e-10
	Grid size	1.331	0.264

Table 1: Reduced ANOVA table for execution times on two machines for all six methods on instances of less than 20 cities

3.2 Experiment Design and Setup

For our experiment, there were two treatment factors and two blocking factors. We measured the effect of these factors on the response, which is the run time. The treatment factors, which are factors of interest, are the number of cities and the methods used. Although the grid size and random seed also have an effect on the execution time, they are not factors of interest. We perform blocking to estimate the effect of these blocks and remove the effects of blocks on the response. For the grid sizes of 100, 1000, and 5000 we have a trial for each combination of the levels for our two treatment factors. For further blocking, we apply the same random seed to all methods with the same number of cities and grid size.

To gather experimental data of the execution times of our algorithms on these test instances, we used DataMill. Datamill is an infrastructure for performance evaluation and has a cluster of computers for running benchmarks. We selected two machines for our experiment. Machine 1 has an Intel Core i5-2500 CPU with 8GB of RAM. Machine 2 has an AMD Opteron 8378 Processor with 32GB of RAM. Due to time constraints, experiments were run on all CPU cores with one process for each core. The code is compiled with GCC 4.7.3 using optimization flag `-O2`. Because most of our algorithms are known to have exponential worst case running times, we set a time limit of 10 minutes for all our test runs. For each combination of the two treatment factors, 144 trials were executed on both machines with a total of 3 grid sizes.

4 Results

To analyze the results, the data is separated by number of cities. Data gathered for instance sizes of 5 to 19, 20 to 25, and 26 to 32 were partitioned into separate files for each machine. This produces better analysis of the effect of factors on run times. Methods that time out at a certain size are ignored, as missing data can result in biased ANOVA analysis.

The experimental results in Table 1 show that for small instances, different grid sizes have the same effect on the run time. This is indicated by a p-value much less than 95%. Thus, changing grid sizes for small instances has an uncertain effect on execution time. However, other factors all have a significant effect on execution time with a confidence interval greater than 99%. The same trend continues with instance size, method, and seed on instances of 20 to 25 cities in Table 2, with the grid size showing a significant effect on execution time.

Machine	Factor	F value	p-value
1	Instance size	87.530	<2e-16
	Method	385.404	<2e-16
	Seed	2.119	1.33e-09
	Grid size	10.686	2.35e-05
2	Instance size	382.141	<2e-16
	Method	1203.730	<2e-16
	Seed	1.892	3.45e-07
	Grid size	22.767	1.48e-10

Table 2: Reduced ANOVA table for execution times on two machines for all six methods except enumeration on instances of 20 to 25 cities

5 Discussion

We first consider Figures 1 and 2 which show the mean running times of our various algorithms on a log scale. The methods are

1. Enumeration
2. Held-Karp 1-tree
3. Bellman-Held-Karp
4. Branch and bound
5. BHK recursive
6. BHK non-recursive

From these plots we can see some clear thresholds on which one algorithm performs better than the others. Firstly we see that on problem instances up to and including 6 cities, the enumeration algorithm is the fastest. This is an encouraging result because it illustrates that a asymptotically slow algorithm can be faster on small problems because of a lower overhead. For bigger than 6 city problems we can see that the BHK dynamic programming algorithm performs better up until 14 cities. From 15 to 20 city problems we can see that either BHK or the Branch-Cut algorithm is superior depending on the test instances. From 21 cities onward we see that the Branch-Cut algorithm dominates all other algorithms. The 1-Tree algorithm is notably lacking here. We can see that its trend in run time is similar to those of the dynamic programming algorithms, but it is consistently slower and hence never appears as the fastest algorithm.

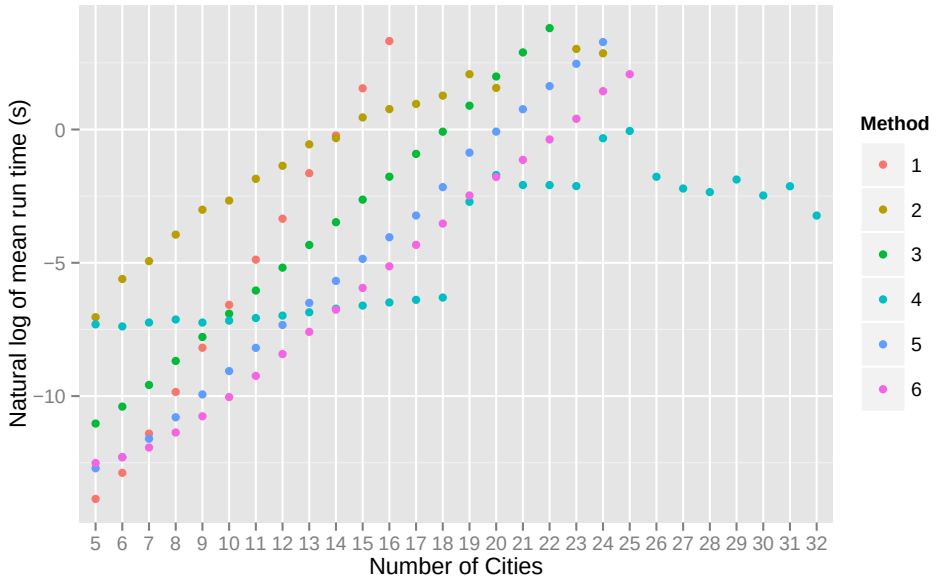

Figure 1: Mean running times of various algorithms on different sized geometric TSP problems. Computed on machine 1.

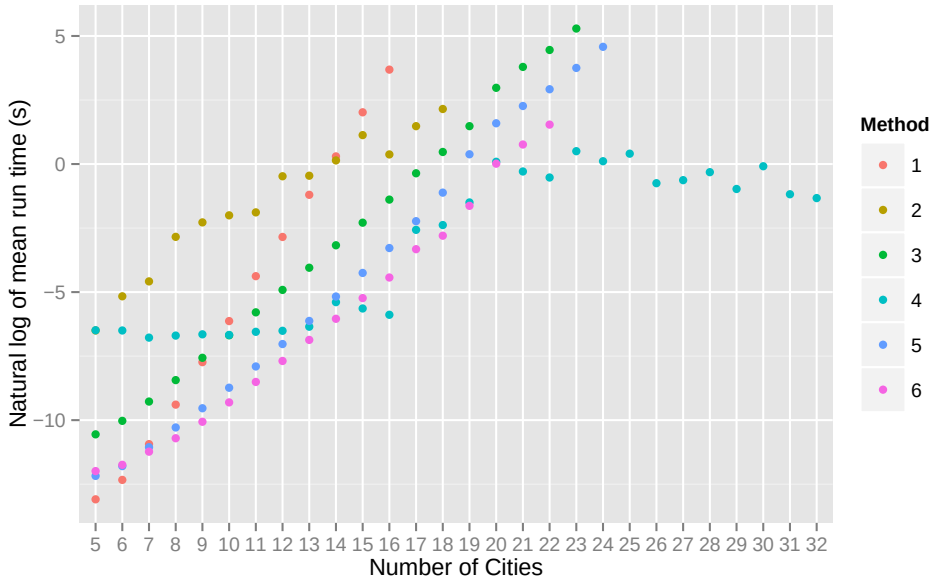

Figure 2: Mean running times of various algorithms on different sized geometric TSP problems. Computed on machine 2.

Figure 3 shows that for n=5, enumeration has the most wins. It is always the best algorithm. Branch and cut is within a factor 1000 of the best algorithm, and BHK is about a factor 3 of the best algorithm. The 1-Tree algorithm solves 70% of the problems in a factor of 1000 of the best algorithm.

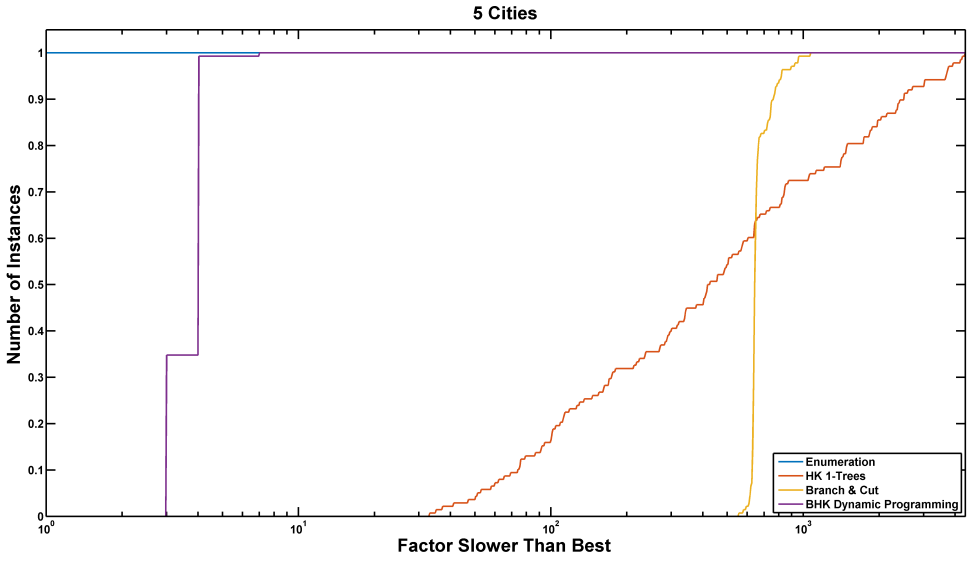

Figure 3: Performance profiles of algorithms on instances of size 5.

Figure 4 shows that for n=10, enumeration and branch and cut are on par solving within a factor 200 of the best algorithm. However, BHK dynamic programming is the overall winner.

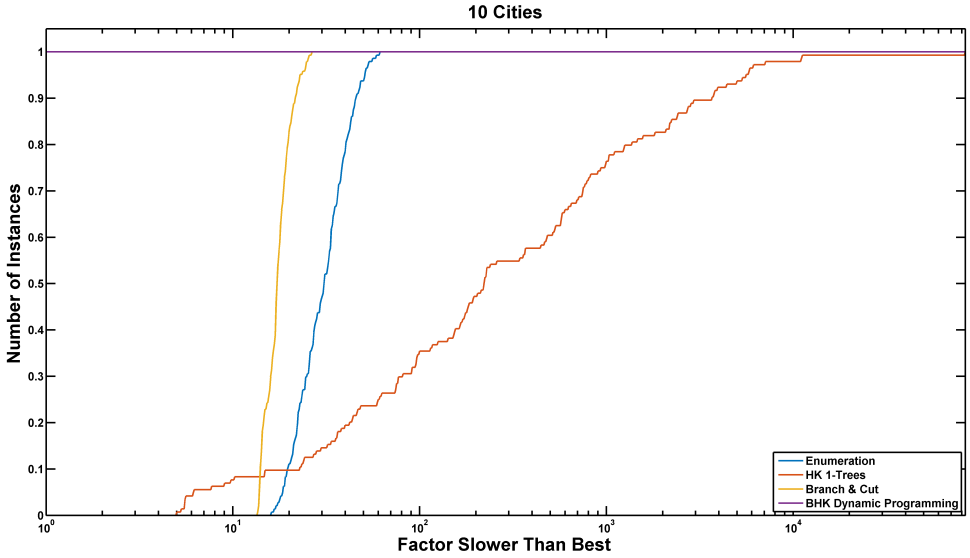

Figure 4: Performance profiles of algorithms on instances of size 10.

Finally, for n=15 branch and cut becomes has the most wins. The probability that it is the best algorithm is about .98. BHK dynamic programming is a factor of 2 slower than the best for 90% of the problems as show in Figure 5.

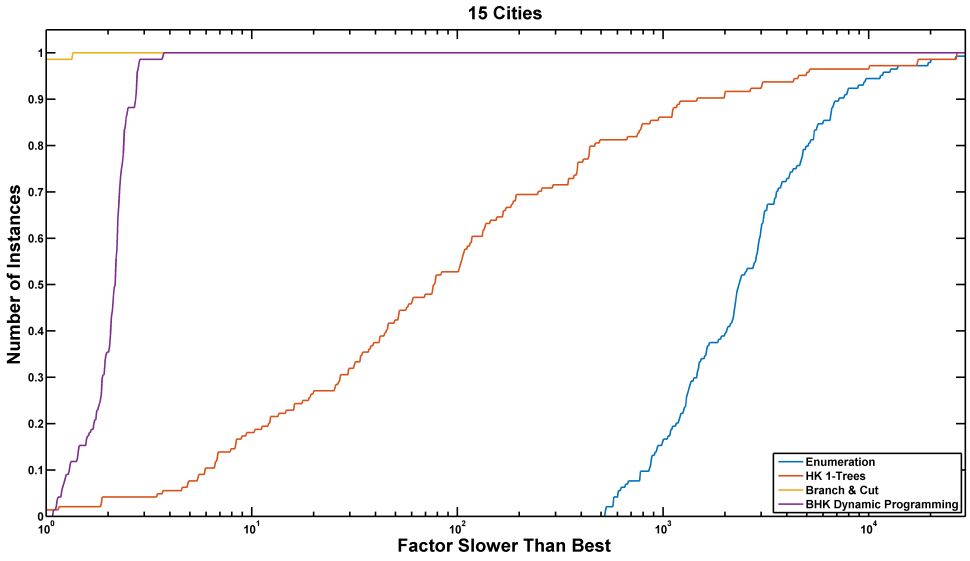

Figure 5: Performance profiles of algorithms on instances of size 15.

6 Future Work

After a careful analysis of the data collected from experiments, we found that several seed and grid sizes were repeated in our experiment. This is due to the fact that the random numbers were generated between 1 and 100. For future experiments, we would like to ensure that no two random seeds are the same in our experiment.

In addition, for an ideal experiment design we would like to randomly assign grid sizes and city sizes during the experiment rather than iterating through them in order. Iterating through them in order may in fact be causing extra disk swapping. Once some algorithms reach n=20, they may consume more memory than the available physical RAM. Therefore, they may time out due to previous experiment runs that were swapping memory to disk.

7 Conclusions

For this project, we implemented four algorithms for solving the TSP and collected experimental results of their run times. The experimental data shows that the instance size, method, and seed have a significant effect on run times. As the number of cities increases from 5 to 25, enumeration is the best, succeeded by BHK dynamic programming, followed by branch and bound linear programming. Our performance profiles illustrate that these algorithms are the clear cut winner for 5, 10, and 15 cities.

References

[1] David L Applegate, William J Cook, Sanjeeb Dash, and David S Johnson. *A Practical Guide to Discrete Optimization*, chapter 7: Dynamic Programming. 29 December 2014. Draft.

[2] David L Applegate, William J Cook, Sanjeeb Dash, and David S Johnson. *A Practical Guide to Discrete Optimization*, chapter 1: The Setting. 7 August 2014. Draft.

[3] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, January 1962.

[4] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[5] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[6] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25, 1971.

Appendix

Result Tables

Results for the benchmark-40 and server-1 machines are in `b40.csv` and `s1.csv` files respectively.

Code

Relevant code files are as follows:

- Main program driver, Nearest-Neighbour, Branch-Cut code: `tour.cpp`
- Graph constructs: `graph.cpp, edge.cpp, tnode.cpp`
- 1-Tree Code: `onetree.cpp`
- BHK Dynamic Programming: `bhk1.cpp, bhk2.cpp, bhk3.cpp`
- Datamill scripts: `collect.sh, run.sh, setup.sh, tour.sh`