Group 27: Heather Godsell, Jesse McKenna, Mario Franco-Munoz

CS325 Project: Traveling Salesman

2018/6/8

Contents

[Algorithm Analysis 2](#_Toc515540308)

[Greedy Solution 2](#_Toc515540309)

[Description 2](#_Toc515540310)

[Pseudocode 2](#_Toc515540311)

[Brute-force Solution 2](#_Toc515540312)

[Description 2](#_Toc515540313)

[Pseudocode 2](#_Toc515540314)

[Exact Solution: “*Branch And Bound”* 2](#_Toc515540315)

[Description 2](#_Toc515540316)

[Pseudocode 3](#_Toc515540317)

[Algorithm Selection 6](#_Toc515540318)

[Test Results 6](#_Toc515540319)

[Competition Results 6](#_Toc515540320)

[Bibliography 7](#_Toc515540321)

# Algorithm Analysis

## Greedy Solution

### Description

abc

### Pseudocode

abc

## Brute-force Solution

### Description

The brute-force solution is an exact solution, but with an extremely poor runtime. Put simply, given a starting vertex *s*, a brute-force algorithm will look at every potential path beginning and ending at *s*, ultimately returning the smallest path that it encountered. Another way to think of the potential paths is as permutations of the vertices to be visited, with the restriction that the permutations begin and end at *s*. [1]

However, the number of permutations of vertices is extremely large – O(*n*!), where *n* represents the number of cities on the map – so the brute-force algorithm becomes impractical even at an *n* of only 20. [2] Regardless of its impracticality at large values of *n*, the brute-force method is an exact algorithm, so it is worth considering as a baseline for the efficiency and accuracy of our other exact solution, branch and bound (discussed in the subsequent section).

### Pseudocode

The basic pseudocode for the brute-force solution is as follows:

find an initial Hamiltonian tour, called T

set the min tour to T

set the min distance to T.distance

while there are unchecked permutations of T, excluding start and end vertex *s*:

generate a new permutation of T, called T’

if T’.distance < min distance:

set the min tour to T’

set the min distance to T’.distance

[1]

The process of getting permutations is where the complexity is introduced, as there are O(*n*!) permutations of *n* cities. When getting permutations, we would also need to be careful to exclude the start/end vertex *s* from the permutation, since those points will never change. Thus, more specifically, there are (*n* – 1)! permutations of a path from *s* back to *s*, since *s* is excluded from the permutation process. We still need to take into account the distance from *s* to other vertices, though, so we can’t simply remove it from the calculations.

Simple pseudocode for generating the permutations is as follows:

permutations = []

permute(T, start, end):

if start == end:

permutations.add(T)

else:

for i from start to end:

swap(T[start], T[i])

permute(T, start+1, end);

swap(T[start], T[i]) // backtrack

[3]

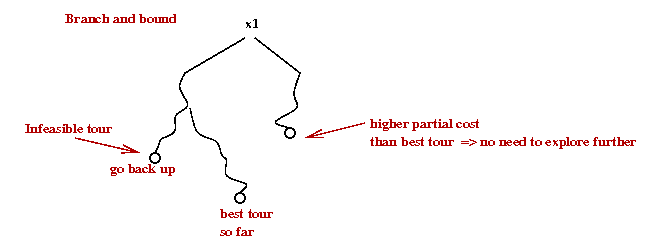
## Exact Solution: “*Branch And Bound”*

### Description

For determining the exact solution, the easiest approach to consider would be that of the “Naïve Approach” or “Brute Force Method” in which case every permutation is examined. A more efficient approach (amongst many) to determining the exact solution however involves either applying a *Branch and Bound* method.

The underlying idea behind the B&B method is that an optimal solution is found by forming a rooted tree of candidate solutions starting at the route node (starting city). From this point, each branch is checked against the upper and lower estimated bounds of the optimal solution, and the branch is discarded if it cannot produce a better solution than the best one found so far. [4]

Below is a simple graphical representation of the B&B method.

[5]

To determine this upper and lower brands, we first need to create an adjacency matrix for the graph in question. After which the matrix is reduced, and we obtain a cost of reduction. For each subsequent node we assign a “cost” value in the following manner:  
(The following matrixes are a direct quote/example taken from: <https://people.eecs.berkeley.edu/~demmel/cs267/assignment4.html>)

i\j 1 2 3 4 5 6 7

\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1 |Inf 3 93 13 33 9 57

2 | 4 Inf 77 42 21 16 34

3 | 45 17 Inf 36 16 28 25

4 | 39 90 80 Inf 56 7 91

5 | 28 46 88 33 Inf 25 57

6 | 3 88 18 46 92 Inf 7

7 | 44 26 33 27 84 39 Inf

[6]

* Reduce this new matrix to obtain an additional cost of reduction

i\j 1 2 3 4 5 6 7

\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Cost of reduction:

1 |Inf 0 83 9 30 6 50 3

2 | 0 Inf 66 37 17 12 26 4

3 | 29 1 Inf 19 0 12 5 16

4 | 32 83 66 Inf 49 0 80 7

5 | 3 21 56 7 Inf 0 28 25

6 | 0 85 8 42 89 Inf 0 3

7 | 18 0 0 0 58 13 Inf 26

Total Cost of Reduction: 84

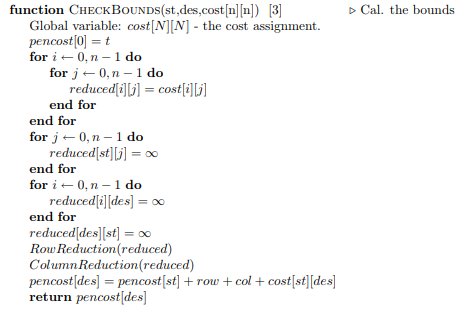
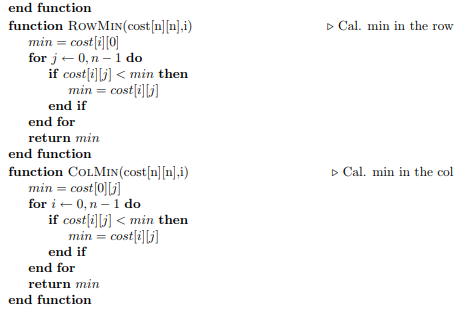
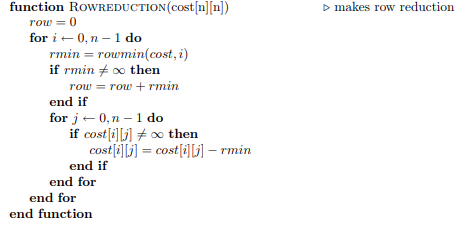
* Add the matrix reduction costs to the path cost C(Root Node, Node-k) + R+ Rk’  
  (Where C() is the distance between two nodes) R is the Root Cost of Reduction, and Rk’ is the cost of reduction of subsequently analyzed node k where subsequent matrix reductions are done per row AND per column.
* Repeat procedure for all connected nodes (obtain the cost of the children nodes, distance + R + R’)
* Select the node with the lowest cost
* Repeat process for that node

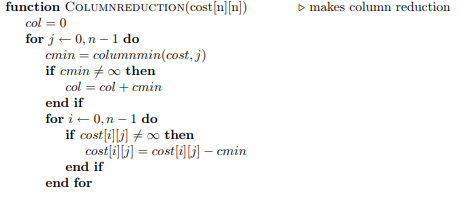
Process is repeated until we have visited all nodes (cities) and thus the optimal TSP path has been found.

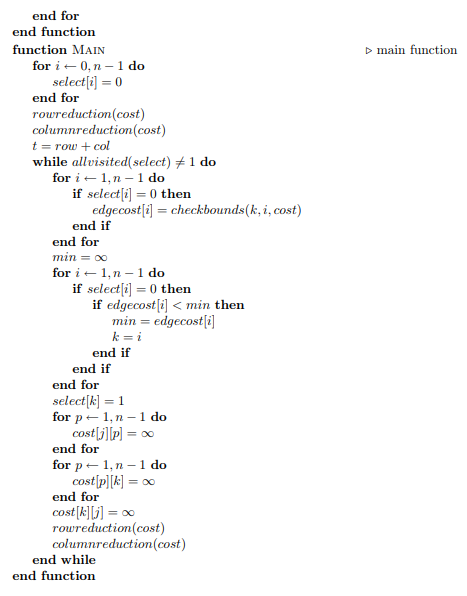
For a much more detailed explanation of the outlined procedure please visit <https://www.youtube.com/watch?v=1FEP_sNb62k> and/or <https://people.eecs.berkeley.edu/~demmel/cs267/assignment4.html>

### Pseudocode

(The following pseudo code is a direct quote from <http://cs.indstate.edu/cpothineni/alg.pdf> ) [7]





# Algorithm Selection

*A verbal description of the algorithm(s) your group implemented along with pseudocode. You may select more than one algorithm to implement.*

*A discussion on why you selected the algorithm(s).*

# Test Results

*Your “best” tours for the three example instances and the time it took to obtain these tours. No time limit.*

# Competition Results

*Your best solutions for the competition test instances. Time limit 3 minutes and unlimited time.*

# Bibliography

|  |  |
| --- | --- |
| [1] | T. Donaldson, "13. Case Study: Solving the Traveling Salesman Problem," 2012. [Online]. Available: http://www.cs.sfu.ca/CourseCentral/125/tjd/tsp\_example.html. [Accessed 31 May 2018]. |
| [2] | "Travelling salesman problem," [Online]. Available: https://en.wikipedia.org/wiki/Travelling\_salesman\_problem. [Accessed 31 May 2018]. |
| [3] | B. Jain, "Write a program to print all permutations of a given string," GeeksforGeeks.org, [Online]. Available: https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/. [Accessed 1 June 2018]. |
| [4] | "Branch and bound," [Online]. Available: https://en.wikipedia.org/wiki/Branch\_and\_bound. [Accessed 30 May 2018]. |
| [5] | "The Traveling Salesman Problem (TSP)," [Online]. Available: https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html. [Accessed 30 May 2018]. |
| [6] | "CS267. Assignment 4: Traveling Salesman Problem," 1 April 1996. [Online]. Available: https://people.eecs.berkeley.edu/~demmel/cs267/assignment4.html. [Accessed 5 May 2018]. |
| [7] | C. Pothineni, "http://cs.indstate.edu/cpothineni/," 13 December 2013. [Online]. Available: http://cs.indstate.edu/cpothineni/alg.pdf. [Accessed 30 May 2018]. |