CS 3510[*] – TSP Solver

Henry Harris, Molly Williams

Spring 2020

Contents

[*]Georgia Institute of Technology

1 Pseudocode

```
create an array of tuples that represents a node and holds level, path, a visited array, heuristic, and

func tsp_helper(heuristic, lower_bound, level, path, visited):
    if at a leaf node
      calculate the cost to return to start node
      check to see if the current value is less than the final value, if so
        set the final value to be the current value
        update the final path
        return

  for all the vertices i = 0 to n
      if i has not been visited
        calculate the cost from the previous node to the current node
        add this cost to the total cost of the previous node
        calculate the new lower bound
        new visited array = curr visited array

if the total cost + lower bound is less than final cost:
          mark i as visited
          push this new node onto the heap
        end if
      end if
  end for
end func

func tsp():
  create a current path that starts at root node
  create a visited array that shows if a node has been visited on the current path

  add the root node to the current path, and set this node to visited
  compute your initial lower bound by looking at the minimum edges around each node

  push this node onto the heap

  while the heap is not empty:
    pop a value off the priority queue
    if the heuristic + the lower bound is less than the final cost:
      call tsp_helper
    else:
      continue
    end if
  end while
end func
```

2 Rationale

To create our algorithm, we based it off the idea of the Branch and Bound algorithm. The way we did this was mainly through a tsp initialization function and a helper tsp method. The tsp function takes in the initial data and sets up the initial current path and visited array. Then, it creates an array of first mins and second mins, and at position i in these arrays it stores the cost of the minimum edge and second minimum edge, respectively, leaving i. These values will be used to calculate the lower bound, which is a bound on the best possible solution. You can choose any arbitrary node to be the root of the tree without loss of generalization.

The way we chose to go through the levels and nodes was with a min heap for priority purposes. We push the first node and all its relevant information onto the heap first to start the process. Also, for optimization, we only call tsp helper if the current cost at the node plus the lower bound is less than the final cost. If the cost is greater than the current final cost, then there is no reason to keep exploring that path.

When the cost is less and tsp helper is called, the function moves to the helper function that takes in the heuristic, lower bound, level, and current path as well as a visited array showing which nodes have already been explored on this level. In this function we first check to see if the node we are on is a "leaf" node in the tree, meaning it is the final node the salesman will visit. If this is true, the steps to process the node are slightly different. When this node is reached, you first want to calculate the cost of returning to the root node, because this will be included in the total cost of the path. Then, you want to add that edge to the current total cost and check and see if the current value is less than the currently stored final value. If so, then you want to update the final value and path to be the current path, which we did by creating an update final stats function that sets the current information to be the final information.

However, if we are not at a leaf level, then we go through all the nodes at the current level and process them. To do this, we first check to make sure the node has not already been visited, if it has then we just continue to the next node. If it has not been visited, then we need to find the cost at this node in the current path. To do this, we first calculate the cost of the edge from the previous node in the path to the current node then add this to the heuristic of the previous node. Next, we have to calculate the new lower bound for the node that we are on. We do this by taking the lower bound from the previous node and subtracting an edge weight such that the lower bound remains as tight as possible. The way to do this is to take the minimum edge of the 2 nodes and divide this number by 2, since this is the smallest value the edge could be. Once the total cost of this node at the current level and the new lower bound is determined, then we check to see if the calculated cost is less than the final cost we currently have stored. If it is greater than the cost, we disregard this path because it is not the correct one. If it is less than the cost, we add the new node to the heap so it will get sent back into the queue and the next level in the path can be processed. This continues until all the nodes have been processed.

3 Reasoning

In order to settle on this algorithm, we researched a few different approaches to the problem and tried to determine which would be the best to implement based on time efficiency and ease of implementation. First, we looked into simple algorithms such as nearest neighbor, which has a quicker runtime however will not always give the most optimal solutions, and works better on smaller sets of data. Since we want our algorithm to work on larger sets we ruled this one out. Next, we looked into Chistofides' algorithm which runs in polynomial time, however it is a 1.5 approximation algorithm, so we ruled this out because we required ¡ 1.10 approximation. Then, we came across the Branch and Bound algorithm. We started reading about it and learned it finds an optimal solution by building a tree and setting bounds so that paths in the tree can be cut off and there were many opportunities to improve the running time of the base algorithm. When you decide a path won't work at node i, you cut out a subtree of depth x and in turn save you having to process O(2x) nodes. Also, because this algorithm can remove non-feasible branches early on, we believed it would work well on large sets of data.

After we thought we fully understood the branch and bound algorithm, we attempted our first implementation of it. We attached a matrix to each node in each of the different paths and tried to use matrix reduction to find the total cost at all the nodes. However, once we attempted to run this, due to the deep copy of a 2D array in our code, the runtime was way too long. To optimize our solution and reduce the runtime, we stepped away from using matrices entirely and instead implemented a lower bound into the function. This lower bound represented the lowest possible cost of each path. We set this lower bound at each node based on the remaining nodes still needing to be reached in that particular path and the cheapest way to reach these nodes. This way, we could add this number to the total cost from the path at the node and determine if this path was worth exploring. By using a lower bound instead of the matrix approach, we were able to remove the deep copy from our code and remove additional paths early on so we did not waste time exploring them. We also had our code try the best possible route first in order to reduce our max bound as early as possible. We did this by using a min heap to store the nodes so that the current path with the lowest cost would be explored first and allow us to discredit the high cost paths early.