Travelling Salesman Problem - DFS and BFS

John Murray

Computer Science

Speed School of Engineering

jmmurr02@louisville.edu

1. Introduction

The Travelling Salesman Problem (TSP) is a problem in which the goal is the find the most efficient path between all of the nodes such that the path creates a Hamiltonian path. This report discusses the results of implementing both depth-first and breadth-first algorithms on a directed graph such that a path is found from the start destination to the end-destination. In this report, we are dealing with a specialized version of TSP in which we do not have to visit all cities, the edges have a clear direction, and we wish to only get to the goal city from the start city.

2. Approach

The approach, probably quite obviously, were to write two algorithms for implementing breadth-first and depth-first searches (for directed graphs). Once the algorithms were implemented, metrics were collected on the number of transitions that occurred and the time it took to execute each algorithm.

Depth First Search

Just to give you a better understanding, the exact algorithm for depth first search that was implemented is as such:

```
def self.dfs(graph, opts = {})
  opts[:depth]     ||= 0      # current depth
  opts[:max_depth] ||= 11     # maximum depth
  opts[:node]      ||= 1      # current node
  opts[:goal]      ||= 11     # goal node

  if graph[opts[:node]] == graph[opts[:goal]]
    return opts[:node]
  end
```

```
    return nil if opts[:depth] == opts[:max_depth]

    graph[opts[:node]][:points_to].each do |pt|
      incr_trans
      result = dfs(graph, {
        depth:      opts[:depth] + 1,
        max_deph:   opts[:max_depth],
        node:       pt,
        goal:       opts[:goal]
      })
      if result
        return [result].flatten.unshift(opts[:node])
      end
    end

    nil
  end
```

You'll want to take notice of a couple of things:

- **`opts`** - Passed in so that the algorithm can be used for more generic problems other than the exact parameters that are specified and used (by default) for this project.
- **`incr_trans`** - A function that increments the number of transitions taken by the algorithm (and is used as a performance measure).

In this implementation, we are performing a recursive search such that if the current node is not the solution node, then the function is called recursively on all of the children in numeric order based on their node-number (eg. 1, 2, 3, 4, etc.).

We are also specifying a value `opts[:max_depth]` that keeps the algorithm from getting stuck in any type of loop. Since we have 11 nodes in total, for this particular problem, we can be sure that we will not need to exceed a depth of 11 to find an answer. So, if a depth of 11 is reached, then the algorithm returns with no solution (aka - it works itself back up the recursion tree and continues to search).

If no solution is found, then the value `nil` is returned. If a solution is found, then an array of node-numbers is returned that represents the path. This array is built up recursively by pushing each node to the front of the list, once a solution has been found. Thus, the `unshift` command.

Breadth First Search

And just to be thorough, the exact breadth first algorithm that was used is implemented as such:

```
  def self.bfs(graph, opts = {})
    opts[:start] ||= 1     # start node
    opts[:goal]  ||= 11    # goal node

    queue = []
```

```
    queue << [opts[:start]]

    graph[opts[:start]][:visited] = true

    until queue.empty?
      path = queue.shift
      node = path.last

      return path if node == opts[:goal]

      graph[node][:visited] = true
      incr_trans

      graph[node][:points_to].each do |pt|
        unless graph[pt][:visited]
          graph[pt][:visited] = true
          new_path = path.dup
          new_path << pt
          queue << new_path
        end
      end
    end

    nil
  end
```

You'll, once again, want to take note of a couple of things:

- **`queue`** - We're going to store the entire path into the queue so that once a solution is found, we can just return the last item that we had in the queue. So, we're building the return path as we go along, instead of working backwards once a solution is found. We can do this now because our state-space is so small. If we were working with a larger state-space we may want to conserve memory a little more and compute the path by working backwards.
- **`opts`** - Passed in so that the algorithm can be used for more generic problems other than the exact parameters that are specified and used (by default) for this project.
- **`incr_trans`** - Means the same thing for BFS as it did for the DFS implementation.

Unlike the previous DFS implementation, we are not implementing this algorithm recursively because the solution is simpler this way. In this implementation, we are using a queue to keep track of nodes that still need to be processed. If a node is found to be the solution, the the solution-path is returned, if it is not the solution, then all of that node's children are added to the back of the queue to be processed. Then the next item is popped off the front of the queue and the same processing is done again. This is repeated until either a solution is found or the queue is empty. If nothing is found, then `nil` is returned.

As we go along, we mark each node as `:visited`. If a node has been visited, then there is no need to process it again (or put it on the queue to be processed). So you'll notice that when

we are processing a node, we mark it and when we queue a node, we also mark it. We do it both ways because not doing so could result in duplicated computation if something is queued multiple times before it is processed.

What we are storing on the queue is not actually the node, it is the path to get to the node. Since we are not working recursively, we need a way to keep track of the path that we have taken to reach the current node. By storing the entire path, once a solution is found, we need only return the path. The path is built when it is pushed to the queue by taking the node's current path and appending the child-node to the end of it.

3. The Results

Data

The problem was solved using a TSP file with 11 data-points. These data-points were supplemented by a table that explicitly defined the directed edges between the points. Within the solution, the TSP data is parsed and then the edges are built manually (statically).

The Results

DFS
- Solution: [1, 2, 3, 4, 5, 7, 9, 11]
- Time (ms): 0.034715
- Transitions: 7

BFS
- Solution: [1, 3, 5, 8, 11]
- Time (ms): 0.025338
- Transitions: 10

4. Discussion

Speed and Transitions

While I would have imagined the time of the DFS algorithm to be faster (given that less transitions were made), I would attribute the minor difference to the cost of recursive calls (extra method-calls) in Ruby. However, the difference is very very small ($\simeq 9.3\mu s$) and can probably be ignored. I wouldn't consider this to be an issue unless observed on larger data-sets.

Path Optimality

Since neither depth-first nor breadth-first are optimal (except for BFS where all edges are equal), it is hard to discuss which one is more optimal than the other. However, given the nature of the algorithms, it is certain that the DFS algorithm will never be able to find a solution that is *better* (in terms of path length) than the BFS algorithm. It can, at most, find a solution that is equal.