Randall Fowler

EEC 289Q

10 March 2024

Homework 3: Traveling Salesman Problem

As the traveling salesman problem (TSP) is well known, there are many known algorithms to solve or estimate the best solution. Given that time is a constraint, solving for the best solution is not practical, and a meta heuristic should be used to get close to the best solution within a 15-minute time frame. To do this, the Lin-Kernighan (LK) algorithm was chosen for having a reasonable run time with great results [1].

The LK algorithm is an improved heuristic to the k-Optimal (k-Opt) heuristic. For the k-Opt, the runtime is O(n^k) as k edges are evaluated for switching at every node. As each node is iterated through, close nodes are evaluated to switch the edges along the path [2]. 2-Opt and 3-Opt are common implementations to consider 2 and 3 edges for switching, and the runtimes would be O(n^2) and O(n^3) respectively. LK investigates the possibilities of switching edges with more costly paths and then performs a k-Opt algorithm. When possible improvements are found, the value of k will grow to perform different k-opt moves [3].

Implementation of LK are described in the flowchart and algorithms below. In figure 1, the algorithm will initialize a path, or tour, for the traveling salesman to take. This starting path is given directly as it travels to nodes in order. While the algorithm is improving, it will continue to call the improving function, and for each iteration of improvements, the new solution will be stored. These solutions can be found in the Solutions folder on GitHub.

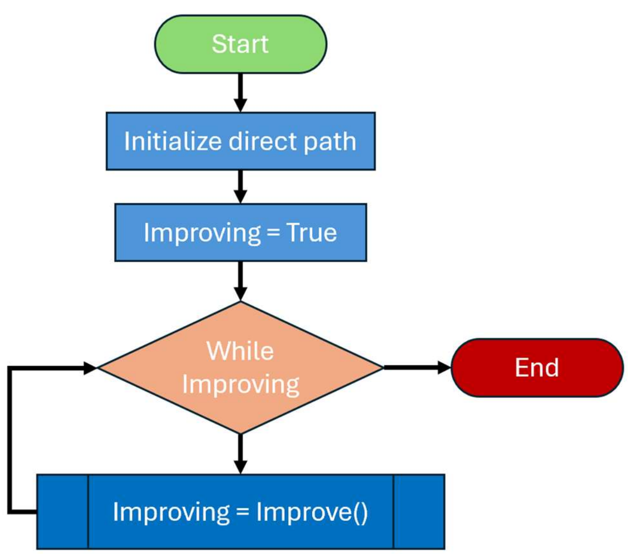

Fig. 1: Optimization Flowchart

The method for improving the tour involves iterating through every node along the path and evaluating the nodes before and after the current node. For naming convention, the nodes before and after the current node will be notes as “prenode.” Since the path will be the same length for traveling either direction, defining which node is before and which is after does not matter. Gain and distance are terms that will be used interchangeably, and the gain calculated between the current node and prenode will be the reference point. To find a potential improvement, the closest nodes to a prenode are found and evaluated. For the closest nodes with decrease in distance, an attempt will be made to remove the current node in exchange for the closer node. If it fails, it will try the next closest node. Five attempts will be made to remove the current node and replace it with a close node, but if all 5 attempts fail, the current node will iterate to the next node in the tour. However, if the node is successfully removed, the improve method will return true.

Algorithm 1: Improve method for Lin-Kernighan

```
Output1: True if improved, False if not
foreach node along the path do
    foreach prenode, before and after the current node do
        gain = distance between current node and prenode
        removed_edges = (current node, prenode)
        close_nodes, reduced_gains = closest(node, gain, removed_edges)
        attempt = 5
        foreach close_node in close_nodes do
            if close_node is not a prenode then
                added_edges = (prenode, close_node)
                if sucessfully remove_edge(node, close_node, reduced_gain,
                  removed_edges, added_edges) then
                    return True
                end
                attempt -= 1
                if attempt == 0 then
                    break
                end
            end
        end
    end
end
```

To find the closest node to a given target node, algorithm 2 describes this operation. All neighboring nodes to the target will be assessed for potential gain. Each neighbor will have a calculated reduced gain, and if the reduced gain is positive without being removed already or in the tour, the prenodes will be looked at. For each prenode (near node), if not already assessed, the potential gain is calculated as the difference between distances of prenode to neighbor and target node to neighbor. This potential gain will then be stored in a dictionary and sorted by potential gain.

Algorithm 2: Closest method for Lin-Kernighan

```
Input  : target node, gain, removed edges, added edges
Output: Sorted Dictionary of nodes with potential and reduced gains
foreach neighbor near target node do
    reduced_gain = gain - distance between target node and neighbor
    if reduced_gain > 0, (target_node, neighbor) is not in removed_edges
      and edges of the path then
        foreach near_node do
            if (near_node, neighbor) is not in removed_edges or
              added_edges then
                potential_gain = (distance between near_node and
                  neighbor) - (distance between neighbor and target node)
                if neighbor in neighbors and potential_gain > gain to
                  closest neighbor then
                    save potential_gain in the neighbor dictionary
                else
                    save potential_gain and reduced_gain in the neighbor
                      dictionary
                end
            end
        end
    end
end
return sorted neighbor dictionary
```

Removing and adding edges proves to be the most difficult operation for this algorithm. For removing an edge, only 5 attempts will be allowed to reduce time complexity. For each prenode to the close node, if the edge of the prenode and close node is not in added or removed edges, a new tour will be created with an edge between target node and prenode. This would remove the edge between the original prenode to the target node with the prenode of the close node. If this tour is valid, it will store a new tour and return true. If not valid, it will attempt at adding another edge to fix it.

Algorithm 3: Remove edge method for Lin-Kernighan

```
Input  : node, close node, gain, removed edges, added edges
Output: True if successfully removed, False if not
Check how many edges have been removed from the path, and only
  allow up to 5 edges to be removed.
foreach near_node do
    current_gain = gain + distance between close_node and near_node if
      edge is not in added_edges and removed_edges then
        added = added_edges + (node, near_node)
        removed = removed_edges
        new_gain = current_gain - distance between node and near_node
        valid = create new tour with added and removed if valid or
          added length is less than 3 then
            if new tour is a known solution then
                return False
            end
            if valid then
                save new tour as current path for TSP
                return True
            else
                return add_edge(node, near_node, current_gain, removed,
                  added_edges)
            end
        end
    end
end
return False
```

To add an edge, the closest nodes to a prenode are gathered. If the number of edges removed is greater than 2, only one neighbor of the prenode will be assessed. This is due to reducing the time complexity and only allowing a certain number of additional edges. Otherwise, the 5 closest neighbors will be assessed. For each of the closest neighbors, the edge between the prenode and neighbor node will be removed to attempt fixing the tour. If successful, the method will return true, but if all attempts fail, it will return false.

Algorithm 4: Add edge method for Lin-Kernighan

```
Input  : node, near node, gain, removed edges, added edges
Output: True if successfully added, False if not
close = closest(near_node, gain, removed_edges, added_edges)
foreach close_node with reduced_gain do
    added = added_edges + (near_node, close_node)
    if remove_edge(node, close_node, reduced_gain, removed_edges,
      added) then
        return True
    end
end
return False
```

It's difficult to evaluate the LK algorithm performance without another comparison. While improvements were made, comparison with another student led to the assumption that this algorithm performed well. This will be further addressed as other students share their results with different heuristics.

Since this algorithm requires more time for each iteration than other heuristics, the number of cycles performed in **15 minutes** was **2037** and **1458 cycles** for graphs **A** and **B** respectively. The cost of the best cycle for graph **A** was **3606** and graph **B** was **234**.

GitHub:

Th3RandyMan/TSP-Meta-Heuristic (github.com)

References:

TSP-JohMcg97.pdf (ubc.ca)

11 Animated Algorithms for the Traveling Salesman Problem (stemlounge.com)

Implementing Lin-Kernighan in Python (maheo.net)