Guided Local Search (GLS) by Riley Kraft

According to my cursory research, the best options for solving the Traveling Salesman Problem (TPS) tend to be heuristic algorithms that utilize a construction algorithm to find a locally optimal path, followed by an improvement algorithm to "fine tune" the locally optimal path into a globally optimal one.

In my search for the "best" known heuristic algorithm to solve the TSP, I came across several research papers detailing the Guided Local Search (GLS). According to some researchers, GLS is still considered an experimental method for the TSP, however it has shown tremendous promise in being the "best" method. Some testing [1] has even proven that GLS outperforms the Lin-Kernighan (LK) algorithm, which was specifically designed to solve the TSP, in both computing time and mean excess.

GLS is not itself a heuristic algorithm for solving the TSP, but a helping algorithm that utilizes penalties to increase the speed and optimality of repeatedly running a construction algorithm. Thus, we still need to choose our construction and improvement algorithms. However, for now I will define how GLS works and why it is the key factor in my chosen algorithm.

First off, the GLS runs as many times as defined by the programmer, whether that be a maximum number of iterations or maximum time. GLS starts with an initial path, either found randomly, heuristically or even in its original order. The algorithm then calls on our local search (construction) algorithm to find a locally optimal path. The locally optimal path is defined by making "greedy" choices, starting at the first city in the path, to minimize the total distance (i.e. cost). The GLS then increases our cost (i.e. total distance) by increasing the local costs (i.e. adding penalties) of some of the graph's edges. All edges begin with a penalty record of 0. GLS chooses which edges to adjust by calculating what's called the ***utility*** of an edge. The utility of an edge is calculated by dividing the cost of traveling from point A to point B by (1 + the penalty record for that edge). For each iteration of the GLS, the edge(s) with the highest/maximum utility, which is the edge(s) with the highest cost, is penalized by adding 1 to its penalty record. Thus, we continuously adjust the edge(s) until it becomes optimal (or approximately optimal). The greater a penalty gets the lower the edge's utility becomes, hence the closer we get to an optimal solution. After the penalties are assigned, GLS calls on the construction algorithm to run again for the most recent locally optimal path with its augmented costs, distance + (penalty * ***lambda***).

Lambda is the most critical value in ensuring the effectiveness of GLS since it directly affects the costs of our edges. Lambda is calculated by multiplying ***alpha*** by the cost of the original path, divided by the total number of cities (i.e. the average edge distance). Alpha is a constant that enables us to "tune" the effectiveness of our algorithm by adjusting lambda. Voudouris and Tsang [1] found that less effective "local search heuristics such as 2-Opt require higher [alpha] values than more effective heuristics such as 3-Opt and LK. This is because the amount of penalty needed to escape from local minima decreases as the effectiveness of the heuristic increases and therefore lower values for [alpha] have to be used to allow the local gradient to affect the GLS decisions." Their experiments found that the alpha sweet-spot for 2-Opt was between 1/8 <= alpha <= ½. As well, if alpha was less than 1/8 then the GLS needed more penalty cycles to escape the local minima, and if it was greater than ½ then the GLS would be more focused on just removing the longest edges rather than finding the locally optimal path. As critical as alpha is, it is most often chosen at the programmer's discretion through experimentation. For the purposes of my algorithm, I will be deferring to Voudouris and Tsang's [1] experimental results. There have also been cases of researchers who have dynamically calculated lambda [2], however their results, in my opinion, did not present enough of an advantage in order to justify the additional cost of computing time.

The purpose of the GLS is to progress towards finding the globally optimal path by escaping the local optima [2]. Based on where the initial path starts (i.e. which city), we could end up with very different locally optimal paths if we run the algorithm several times, starting in a different city each

time. GLS works to curb the differences in optimal answers based on starting location by continuously adjusting the locally optimal path without having to run our entire program over-and-over again.

```
GuidedLocalSearch(max_global_iterations/time, initial_path, lambda, max_local_iterations/time){
   optimum_path = initial_path
   for i = 0 to size of initial_path (number of cities){
      initialize city's penalty record to 0
   }
   for  j = 0 to max_global_iterations/time{
      cost+penalties = the sum of each edge's distance + (lambda * penalty)
      candidate_path = FastLocalSearch(optimum_path with cost+penalties, max_local_iterations)
      if cost(without penalties) of candidate_path < cost(without penatlies) of optimum_path{
         optimum_path = candidate_path
      }
      call function to calculate each city's utility in optimum_path
      update penalty for city(ies) with maximum utility in optimum_path
   }
   return optimum_path
}
```

Now that we understand how GLS works, we need to choose the local search heuristic that the GLS is going to aid. Again, I will be differing to Voudouris and Tsang's research [1] which declared that Fast Local Search (FLS), while inefficient on its own, paired with GLS resulted in the most optimal solutions compared to several well known TSP heuristic algorithms.

The main purpose of FLS is to increase the speed of our algorithm by breaking up our path into "neighborhoods". We know that the greater the number of cities in our path the longer our algorithm is going to take to find an optimal solution. GLS requires us to continuously find a new local optimum after issuing penalties, which would lead us to incur a large cost in computing time if our local search algorithm ran through all the possibilities for each iteration of GLS. This is how FLS will help speed up our algorithm, by only searching the "neighborhoods" that have been penalized, rather than all "neighborhoods". Fast local search relies on an improvement algorithm to define its "neighborhoods". Typical choices for this algorithm are 2-Opt, 3-Opt and Lin-Kernighan. Deferring to Voudouris and Tsang, I have chosen 2-Opt since, when paired with GLS + FLS, it produced the fastest and most optimal results in their research [1].

GLS determines the edges that are assigned penalties based on their calculated utilities, and only the edge(s) with the maximum recorded utility is penalized. FLS then calls on 2-Opt to adjust only the "neighborhoods" that have incurred penalties from GLS. This way, we do not spend the time adjusting low cost "neighborhoods". Like GLS, FLS adjusts the path for a maximum count of iterations. However, whenever we find a more optimal path after invoking 2-Opt, the count is set back to 0. Increasing the number of iterations for FLS could provide better results, but it would also drive up the cost of computation time. The iteration limit is left to the discretion of the programmer.

```
FastLocalSearch(current_path with cost_penalties, max_local_iterations){
   count = 0
   do{
      candidate_path = 2-Opt(current_path)  // call function to run 2-Opt on current_path
      get candidate_path cost+penalties
      if candidate_path cost+penalties < current_path cost+penalties{
         count = 0;
```

```
            current_path = candidate_path
        }
        else{
            count++
        }
    }while(count <= max_local_iterations)
}
```

Finally, now that we have GLS and FLS working together we need to define our improvement algorithm that actually adjusts the edges in our path. As stated before, I chose 2-Opt to work with FLS. Like any improvement heuristic, 2-Opt takes an already locally optimal tour and makes small adjustments to create a more optimal tour. In the case of this algorithm very small adjustments, as our 2-Opt will only consider "neighborhoods" that have been penalized in the GLS. 2-Opt chooses a city and finds 3 of its closest neighbors. The algorithm then compares the sum of distances between cities A and B + the distance between cities C and D, and the sum of distances between cities A and C + the distance between cities B and D. Only if the first sum of distances is greater than the second do we then delete the original edges and create the new, shorter edges. The edges between cities A and B, and C and D are gone, and city A now shares an edge with city C, and city B now shares an edge with city D.

To make 2-Opt more time efficient when working with GLS and FLS we need to add some parameters to the 2-Opt algorithm. As stated before, we start our swapping search only with cities with edges that have been penalized. We will indicate this separately by assigning a bit to each city in which cities that have been penalized are set to 1, and cities that have not been penalized are set to 0. 2-Opt chooses a penalized city and conducts its standard evaluation. If a swap did occur, then we must set all the cities' penalty bits to 1. If no swap occurred, then we set the base city's penalty bit to 0. The algorithm loops until all penalty bits are set to 0.

Adjusting 2-Opt in this way ensures we will only consider high cost neighborhoods, thus cutting down our computing time and allowing GLS + FLS to take charge in finding the most optimal path in the time allowed.

```
2-Opt(current_path){
    do{
        optimal = true
        for i = 0 to current_path.size() - 2{
            if current_path[i] has a penalty{     // if a city with a penalty is found
                optimal = false    // Then we  do not have the optimal path
                current = distance(current_path[i], current_path[i+1]) + distance(current_path[i+2],
                                current_path[i+3]) – distance(current_path[i], current_path[i+2]) +
                                distance(current_path[i+1], current_path[i+3])
                if(current > 0)  // if the first path is longer
                    // we swap the edges
                    swap1 = current_path[i+1]
                    swap2 = current_path[i+2]
                    current_path[i+1] = swap2
                    current_path[i+2] = swap1

                    // set penalty records
                    current_path[i].penalty = 1
                    current_path[i+1].penalty = 1
                    current_path[i+2].penalty = 1
```

```
                current_path[i+2].penalty = 1
            }
            else{  // Otherwise, the locally optimal paths already exist
                current_path[i].penalty = 0   // set city's penalty record to 0
            }
        }
    }while optimal = false          // repeat until no penalties exist
}
```

I think this algorithm is a good choice for finding an approximate, if not optimal, solution to the TSP for small to large problem sizes in a reasonable amount of time. The sub-algorithms of this method are simple enough to understand and all computing is done in polynomial time. More than that, it is easy for the programmer to "tune" the driving variables of this algorithm to find an approximate solution based on the size of the problem and the time alloted to solve it.

REFERENCES

[1] http://cswww.sx.ac.uk/CSP/papers/VouTsa-GlsTsP-Ejor98.pdf

[2] https://ieeexplore.ieee.org/document/6256155/

[3] https://www.sciencedirect.com/science/article/pii/S0167637796000429

[4] https://ocw.mit.edu/courses/sloan-school-of-management/15-053-optimization-methods-in-management-science-spring-2013/lecture-notes/MIT15_053S13_lec17.pdf

[5] https://optimization.mccormick.northwestern.edu/index.php/Traveling_salesman_problems

[6] http://web.tuke.sk/fei-cit/butka/hop/htsp.pdf

[7] http://www.lia.disi.unibo.it/Staff/MicheleLombardi/or-tools-doc/user_manual/manual/metaheuristics/GLS.html

[8] http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/