Traveling Salesman Problem - Project Draft

Strand, Michael, mwstrand, 72048580
Van Riper, Brenda, bvanripe, 96124814
Fukutoku, Lisa, rfukutok, 54321338

COMPSCI 271P Artificial Intelligence

November 20, 2022

1 Stochastic Local Search

Problem introduction

In this section, we describe our application of stochastic local search to the traveling salesman problem, in which N cities must all be visited, without repeats, in a circuit such that the traveling distance is minimal. We also assume each city can be visited from any city other than itself. In this setting, we define the state space as the set of all such possible routes, of which there are $(N-1)!$. The objective function for this application is simply the cost, or distance, of the full route (state). We will use 4 different actions to randomly determine a new state: swapping two cities, moving a city to a new random position, inverting a sub route, and swapping the position of a sub route. We then consider the 'neighbors' of a state to be any new state that results from one of these simply transitions.

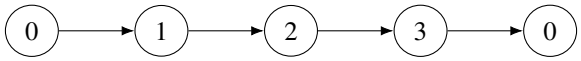
Simulated annealing (SA) is a type of stochastic local search useful for approximating the global minimum of a function. Each iteration, a new state is generated from the previous and if the cost of this new state is less then we store this new state and proceed. However, this algorithm can avoid becoming trapped in local minima by sometimes allowing worse state transitions to be accepted with some variable probability. This probability is primarily dependent on the algorithm's current *temperature*; the higher the temperature the more likely we are to accept 'poor' choices. The performance of SA depends largely on the choice of a temperature function (how temperature will decrease every iteration), the initial temperature, and the kinds of state transitions we implement.

Key features

- **Language and data structure:** Our implementation of simulated annealing is written in Python where the states are of type list of the form

  ```
  [0, 1, 2, 3, 0]
  ```

  where the above represents the path

- **Temperature initialization:** At the start of every search, generate a set of sample transitions and recursively find a T such that the acceptance probability is within a margin of error of our desired initial acceptance probability. Procedure taken from Walid, 2003[1]
- **State changes:** There are 4 different ways a path can be changed here: Swap two cities, Insert city, Swap subroute, Invert path between. Each iteration, one action is chosen uniformly at random to generate a new path. We consider the set of a state's neighbors to be all possible results from any 1 of these operations.
- **Complexity**
  - *space complexity:* $O(1)$ since we only need to reserve memory for the distance matrix, 3 fixed-length paths (new, current, and best) and scalars.
  - *time complexity:* $O(N^4)$ [2]

[1] Computing the Initial temperature of Simulated Annealing, Walid 2003
[2] Optimization by Simulated Annealing: A Time-Complexity Analysis, Sasaki 1987

- **Temperature function:** Every iteration, the initial temperature is scaled down by a factor of

$$\alpha(t) = \alpha_0^t = \left(1 - \frac{1}{\text{max\_it}^{X_0}}\right)^t \tag{1}$$

  - α_0: initial value around the order of 0.9999...
  - t: current iteration (0, 1, ..., max_it) where iteration count is used as a stopping criterion and controls runtime.
  - X_0: initial desired acceptance probability ("at the start, about how likely should it be for bad paths to be picked?")

  Then $\alpha(t)$ is a number initially very close to 1 which decreases geometrically at a rate dependant on the time we plan to run the algorithm (max_it), how desirable it is to randomly choose bad transitions (X_0), and the current run time (t).

  We calculate the probability of accepting a bad transition p as

$$p = e^{\frac{E}{T\alpha(t)}}$$

  where E is the difference in distance between the current path and newly generated path and T is the initial temperature. If $E < 0$ then the new path is longer than the current. This means that as $t \to$ max_it, $T\alpha(t) \to 0$ and thus $p \to 0$. The result is a high probability of jumping out of local minimums early on, but slows to 0 towards the end of run time.

Path changing actions

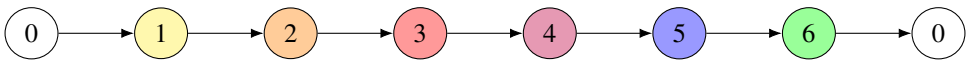


Figure 1: Initial Path

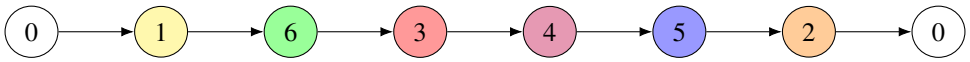


Figure 2: Switch two cities (2 & 6)

Figure 3: Invert path between two cities (2 & 6)

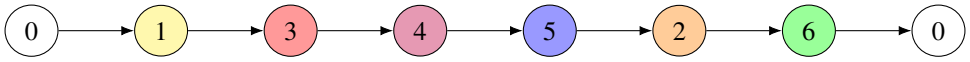


Figure 4: Insert city into random position (2 & 5)

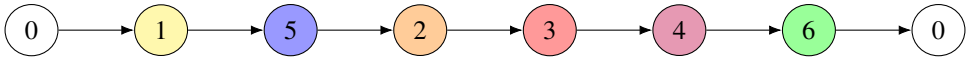


Figure 5: Swap sub route to random position (route 2-4 to 4th position)

Algorithm 1 Simulated Annealing

function ANNEAL(distance_mat, mat_it, X_0)
  $T \leftarrow$ INIT_TEMP(X_0) ▷ Compute initial temperature
  current_path $\leftarrow$ INITIAL() ▷ $[0, 1, 2, ..., N, 0]$
  current_dist $\leftarrow$ OBJECTIVE(current_path) ▷ Calculate path distance
  best_dist, best_path $\leftarrow$ current_dist, current_path

  while stopping criteria not reached **do** ▷ We use max_it
    new_path $\leftarrow$ CHOOSE_APPLY_CHANGE(current_path)
    new_dist $\leftarrow$ OBJECTIVE(new_path)
    $E \leftarrow$ current_dist - new_dist

    if new_dist $<$ best_dist **then**
      best_path, best_dist $\leftarrow$ new_path, new_dist
    end if

    if $E > 0$ **then**
      current_path, current_dist $\leftarrow$ new_path, new_dist
    else

      $p \leftarrow e^{E/T\alpha(t)}$ ▷ where $\alpha(t)$ is as in 1

      if BINOMIAL$(1, p) = 1$ **then**
        current_path, current_dist $\leftarrow$ new_path, new_dist
      end if
    end if
  end while

  return best_path, best_dist
end function

The above performs the simulated annealing algorithm given 3 required parameters:

- distance_mat: a symmetric $N \times N$ matrix of distances, where the $(i, j)^{th}$ element represents the distance to travel from city i to j.
- max_it: the maximum number of iterations before the algorithm terminates. Also serves to determine α_0, the initial temperature scaling value. Larger max_it will produce larger α_0 and thus the temperature stays 'hot' for longer if we are going to run for awhile.
- X_0 : the initial acceptance probability as described in Walid 2003. This value between 0 and 1 informs the initial temperature algorithm and produces starting temperatures that will, in expectation, accept bad routes $X_0 \cdot 100\%$ of the time during the actual search. We also use this value to further influence the starting α_0, such that larger X_0 leads to a higher α_0.

Several subfunctions are also defined within and contribute:

- INIT_TEMP(X_0): *described below*
- INITIAL(): Simply returns the initial path [0,1,2,...,N,0] for an N node problem.
- OBJECTIVE(path): Calculates the path distance by summing over path transition indices of the distance matrix.
- CHOOSE_APPLY_CHANGE(path): chooses with uniform probability one of the 4 switch actions and apply it to the passed path. Returns the neighbor path generated by this operation as a list.
- BINOMIAL(1,p): A Bernoulli random variable with probability of success (1) p.

Algorithm 2 Find initial temperature: problem simulation

function INIT_TEMP(X_0)
  $\omega \leftarrow$ SAMPLE_SIZE(dist_mat) ▷ Choose appropriate sample size
  $\Omega \leftarrow$GEN_SAMPLE(ω, X_0) ▷ Costs of ω simulated bad transitions
  $T \leftarrow n > 1$ ▷ Initialize temperature for recursion

  while $|X_T - X_0| < \epsilon$ **do** ▷ for some $\epsilon > 0$
    $T \leftarrow T \left(\frac{\ln X_T}{\ln X_0} \right)$
    $X_T \leftarrow \frac{\sum_{i \in \Omega} e^{\left(-E_{a_i}/T\right)}}{\sum_{i \in \Omega} e^{\left(-E_{b_i}/T\right)}}$ ▷ a_i, b_i are costs after and before i^{th} trans.
  end while

  return T
end function

This procedure, as described in Walid 2003, calculates an initial temperature capable of providing a smooth temperature decline tailored to specific distance matrices. Since different problem instances may have very different settings (number of nodes, average city distance, variance of distances) the difference in path costs can vary greatly. For this reason, temperature is not a one-size-fits-all and we need a robust way of initializing this important variable.

To find this temperature, the above pseudo code describes a simulation process where we generate ω poor transitions (new distance longer than old distance) and record the difference in costs before and after in Ω through GEN_SAMPLE(ω, X_0). We then recursively find our initial temperature T by scaling down according to the log ratio of current acceptance probability X_T and desired probability X_0. The further off X_T is with the current T the more we adjust until we reach a specified margin of error ϵ. We typically choose a sample size ω of max_it$/10$, such that the first 10% of the runtime has about an acceptance probability of X_0.

2 Brand-and-Bound

Problem introduction

We now consider the brand-and-bound approach to solving the traveling salesman problem. This method is similar to stochastic local with a few key differences. The problem setting and rules are the same, but now we will define a state to be a partial path with neighboring states being the same state but with a new, unvisited city at the end.

Depth first branch and bound search uses a last in first out (LIFO) queue using backtracking search which we implement again using Python lists. Depth first search starts at a root node and will explore as far as possible down the graph till it reaches a goal state or, if it does not, then it will backtrack and follow down another path. We also consider the set of neighboring nodes to be the frontier.

A lower and upper bound heuristic will determine how we prune out paths from the graph structure. At each new node we will compute a heuristic of the lower bound made by $L(n) = g(n) + h(n)$ where $h(n)$ is the heuristic (admissible estimated distance to solution) through the node n and $g(n)$ is the cost of the current path from the root to node n in the graph. Upper bound, U, is the true cost of our current best route – if $L(n) \leq U$ we will prune off that path, so we avoid expanding paths that are costly and focus on paths that are promising. If a node is not pruned then it's neighbors will be added to the frontier stack and it is popped from the queue.

Key Features

- **Language and data structure**: We again use Python for this implementation and lists as our data structure. Every iteration of the algorithm we track the paths (current and best) & costs and overall queue. The LIFO queue is a list of lists. For example, if there are 4 cities then the queue may look like

  ```
  [[0,1], [0,2], [0,3], [0,3,1], [0,3,2]]
  ```

  which indicates the order of partial paths to try and expand next. Objective function costs (distance + heuristic) are calculated against these partial paths to determine pruning actions.

- **Complexity**

  - *space complexity:* $O(N)$ because we only store frontier paths and this list grows lineraly as one city is popped at a time.

  - *time complexity:* $O(N^2)$ using a nearest neighbor heuristic[3]

- **Heuristic: Nearest neighbor**[4] This is a simple heuristic for putting an admissible lower bound on the estimated distance remaining from a current state. It is as follows:

  1. From the current city in the current path, find the shortest distance to the next unvisited city
  2. Repeat while there remain unvisited cities

  This is a greedy algorithm that proceeds towards the cheapest path at every step. This estimated cost is used in addition to the real cost when deciding which neighbors to prune. The use of this heuristic makes it so we are much less likely to visit the global minimum but the algorithm can complete in a realistic amonunt of time.

[3] An Evaluation of the Traveling Salesman Problem, Neoh, Wilder-Smith, Chen, Chase 2020
[4] Nearest neighbor heuristic

Algorithm 3 Branch-and-Bound

```
function BNB_DFS(distance_mat,start_city = [0])
    N ← LENGTH(distance_matrix)
    frontier ← [start_city]
    U ← U_initial                                   ▷ We take any very high number, ex 10^9
    best_dist, best_path ← current_dist, current_path

    while frontier ! = [ ] do                       ▷ frontier is not an empty list
        new_path ← frontier.POP()
        path_cost ← OBJECTIVE(new_path)

        if path_cost > U then
            Continue

        else if LENGTH(new_path) == N then
            U ← path_cost
            best_path, best_dist ← new_path, new_dist

        else
            frontier ←GEN_FRONT(new_path, frontier)

        end if
    end while
    return best_path, best_dist
end function
```

Above is the main portion of the branch-and-bound algorithm. Assuming we always start at city 0, it needs only a $N \times N$ symmetric distance matrix. There are many functions within which are responsible for carrying out this search:

- LENGTH(array): Simply returns the length of the array passed. For the distance array, it returns the number of cities in the circuit. Paths will have returned the number of cities in the path.
- POP(): A stack-pop operation to remove the latest element. Facilitates the LIFO nature of DFS.
- OBJECTIVE(path): Calculates the current total path distance by summing over transition indices of the distance matrix. The objective function also makes a call to NN(); our heuristic function described below. The return is thus $g + h$ for the current path cost g and estimated heuristic cost h.
- GEN_FRONT(path, frontier): This functions pushes onto the frontier stack all new neighbors (as lists) generated by popping a city. It will return the previous frontier with all new neighbors appended on.

Algorithm 4 Nearest Neighbours Heuristic

```
1: function NN(distance_matrix, state)
2:     cost ← 0
3:     next ← state.CURRENT_CITY(state)
4:     iter ← 0

5:     while iter ≤ N − 1−LENGTH(state) do        ▷ repeat once for each unvisited city, excluding last
6:         distance_matrix ← MASK(distance_matrix)
7:         min_index ← distance_matrix[next,:].ARGMIN()        ▷ City with minimum transition distance
8:         cost ← cost + distance_matrix[next, min_index]
9:         iter ← iter + 1
10:        next ← min_index
11:    end while
12:    return cost + distance_matrix[next, start_city]        ▷ Add on distance to return to start
13: end function
```

The sub functions are

- CURRENT_CITY(state): returns the last city held in the list
- ARGMIN(): Finds index of minimum element in a list
- MASK(matrix): Returns a masked version of the distance matrix such that visited cities and routes diagonal routes are not considered for the coming minimization problem.

Nearest neighbors is a simple heuristic to give an estimate for a lower bound of the remaining path distance. We scan through the distance matrix looking for the smallest transition distance corresponding to the current city we are at in the path. The sum of these distances is our heuristic and we use it in the calculation of the objective function.