Algorithmics SAT Part 3:
Advanced Algorithmic Design

1. Improved Algorithm design

A* Algorithm (SSSP)

A* is a pathfinding algorithm used to solve the Single Source Shortest Path problem (SSSP). It works by maintaining a priority queue of nodes to explore, starting from the initial node and expanding outwards. A* uses a heuristic function to estimate the remaining distance to the goal, which helps guide the search towards the target more efficiently than a simple breadth-first search. At each step, A* selects the most promising node (based on the sum of the cost to reach that node and the estimated cost to the goal) and explores its neighbors. This process continues until the goal is reached or all possible paths have been exhausted.

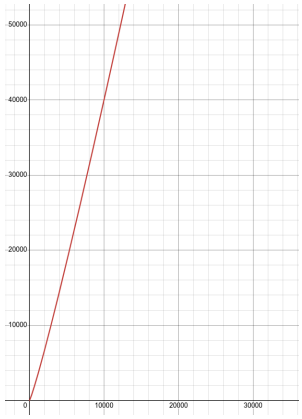
```
Algorithm AStar(graph, start, target)
// Input graph, start node, and target node for the A* algorithm.
// Initializes the parameters and performs the A* search to find the
shortest path.

    Class PathNode(node) // O(1) to init
        // A class to represent a node in the path with its cost values.
        initialize node to node
        initialize g (cost from start to node) to ∞
        initialize h (heuristic cost from node to target) to ∞
        initialize f (total cost) to ∞
    end Class

    // Initialization - O(1)
    Initialize graph to graph
    start_node := PathNode(start)
    target_node := target

    start_node.g := 0
    start_node.h := heuristic(start_node.node)
    start_node.f := start_node.g + start_node.h

    open := PriorityQueue to hold nodes to be evaluated
    closed := set to hold nodes already evaluated
```

```
    parent := dictionary to store the parent of each node

    path := empty list to store the final path
    path_found := False

    Function heuristic(node) // O(1)
        // Calculates the heuristic cost (h) from the current node to the
target node.

        return haversine distance between node and target_node
    end Function

    Function find_path()
    // Performs the A* search algorithm to find the shortest path from
start to target.

        insert start_node into open with priority start_node.f // O(log(V))
From queue lib

        while open is not empty do // O(V)
          current := node in open with the lowest f value // O(log(V)) From
queue lib

          if current.node equals target_node then
              path_found := True
              break
          end if

          add current.node to closed // O(1)

          for each neighbour in current.node.neighbours do // O(E/V) on
average depends on heuristic
              neighbour_node := PathNode(neighbour node from graph)

              if neighbour_node in closed then
                  continue to next neighbour
              end if

              g := current.g + distance between current.node and
neighbour_node
              h := heuristic(neighbour_node.node)
              f := g + h
```

```
            if f < neighbour_node.f then
                neighbour_node.g := g
                neighbour_node.h := h
                neighbour_node.f := f

                set parent of neighbour_node to current.node

                insert neighbour_node into open with priority
neighbour_node.f
            end if
        end for
    end while

    if path_found then
      // Reconstruct path - O(V) in worst case
      current := target_node
      while current is not start_node.node do
          insert current at the beginning of path
          current := parent[current]
      end while

      insert start_node.node at the beginning of path

      return path
    end if
    else
        return None
    end else

    end Function
end Algorithm
```

Analysis

1. The main loop (while open is not empty) can potentially iterate over all vertices in the graph. *O(V)*
2. Within each iteration of this main loop, we perform these key operations:
    a. Extract the node with the lowest `f` value from the open set (priority queue): *O(log(V))*
    b. Process each neighbor of the current node: *O(E/V)* on average, depending on the heuristic accuracy
    c. For each neighbor, we might insert it into the open set: *O(log(V))*



Combining these, for each iteration of the main loop, we have:
$O(log(V)) + O(E/V) * O(log(V)) = O(log(V) + (E/V) * log(V))$

Since the main loop iterates a maximum of `V` times we multiply this all by `V`:
$O(V) * O(log(V) + (E/V)log(V)) = O(Vlog(V) + E * log(V))$

The term E*log(V) dominates V*log(V) (since E ≥ V-1 in a connected graph), so we can simplify this to: $O(E * log(V))$

Simulated Annealing / 2-Opt (TSP)

A 2-Opt + Simulated Annealing algorithm combines local search with a probabilistic approach to solve optimization problems. It starts with an initial solution and iteratively applies 2-Opt moves, which involve swapping two edges to create a new path. The algorithm always accepts improvements, but also allows some non-improving moves based on a temperature parameter that decreases over time. This balance between exploration and exploitation helps the algorithm escape local optima and potentially find better global solutions. As the temperature cools, the algorithm becomes more selective, focusing on refinements to the current best solution.

```
Algorithm SimulatedAnnealing(graph, start, radius, initial_temp, stop_temp,
max_iterations, cooling_rate)
      // Initializes the parameters and performs the simulated annealing
algorithm to find an optimized path.

      // Initialization - O(1)
      Initialize graph to graph
      Initialize start to start
      Initialize nodes to BFS(graph, start, radius) // O(V+E)
      Initialize T to initial_temp
      Initialize stop_T to stop_temp
      Initialize cooling_rate to cooling_rate
```

```
      Initialize max to max_iterations

      Function acceptance_probability(best, candidate) // O(1)
      // Calculates the acceptance probability for a new solution.
      return exp((best - candidate) / T)
      end Function

      Function stats(path) // O(n) where n is the length of the path
      // Calculates the total distance and time for a given path.
      Initialize total_dist to 0
      Initialize total_time to 0
      for each pair (i, j) in path do
            total_dist := total_dist + graph.get_dist(i.name, j.name)
            total_time := total_time + graph.get_time(i.name, j.name)
      end for
      return total_time, total_dist
      end Function

      Function verify_path(path) // O(VE*log(V))
      // Converts a list of potentially disconnected nodes to a path with
virtual edges.
      Initialize verified_path to [path[0]]
      for each pair (i, j) in path do
            a_star := AStar(graph, i, j)
            Append a_star.find_path()[1:] to verified_path
      end for
      return verified_path
      end Function

      Function two_opt_swap(path, i, k) // O(1), List manipulations should
happen using pointers
      // Performs a 2-opt swap on the given path.
      Initialize new_path to path[0] to path[i]
      Append reverse(path[i] to path[k+1])to new_path
      Append remaining nodes in path to new_path
      return new_path
      end Function

      Function anneal()
      // Performs the simulated annealing algorithm to find an optimized
path.
      Initialize best_path_nodes to nodes
      Initialize best_path to verify_path(best_path_nodes)
```

```
      Initialize best_stats to stats(best_path)
      Initialize current_path_nodes to best_path_nodes
      Initialize current_path to best_path
      Initialize current_stats to best_stats
      Initialize iter to 0

      while T >= stop_T and iter <= max do // O(I), Runs Max Iterations
            i := random integer between 1 and length(current_path) - 2
            k := random integer between i+1 and length(current_path) - 1
            new_path_nodes := two_opt_swap(current_path_nodes, i, k)
            new_path := verify_path(new_path_nodes)
            new_stats := stats(new_path)

            if new_stats[0] < current_stats[0] then
                  current_path_nodes := new_path_nodes
                  current_path := new_path
                  current_stats := new_stats
                  if new_stats[0] < best_stats[0] then
                  best_path_nodes := new_path_nodes
                  best_path := new_path
                  best_stats := new_stats
                  end if
            else
                  if random() < acceptance_probability(current_stats[0],
new_stats[0]) then
                  current_path_nodes := new_path_nodes
                  current_path := new_path
                  current_stats := new_stats
                  end if
            end if

            T := T * cooling_rate
            iter := iter + 1
      end while

      return best_path, best_stats
      end Function

end Algorithm
```

Analysis

1. Initialization:
    a. BFS is performed once: $O(V + E)$
2. The main annealing loop can iterate up to *max_iterations* or until temperature reaches stop_T. We will let this be $O(i)$
3. Within each iteration of the main loop, we perform these key operations:
    a. Random number generation and two_opt_swap: $O(1)$
    b. verify_path: $O(n * E * log(V))$, where n is the path length (This is because A* is called potentially n times, each with complexity $O(E * log(V))$)
    c. stats: $O(n)$
    d. Comparisons and probability calculations: $O(1)$
4. Combining these, for each iteration of the main loop, we have: $O(1) + O(n * E * log(V)) + O(n) + O(1) = O(n * E * log(V))$
5. Since the main loop iterates I times, we multiply this by I: $O(I) * O(n * E * log(V)) = O(I * n * E * log(V))$
6. Combining the initialization with the main loop: $O(V + E) + O(I * n * E * log(V))$

The term $i * n * E * log(V)$ dominates $V + E$, so we can consider the overall time complexity to be: $O(i * n * E * log(V))$

2. Advanced Algorithms VS Naive Algorithms

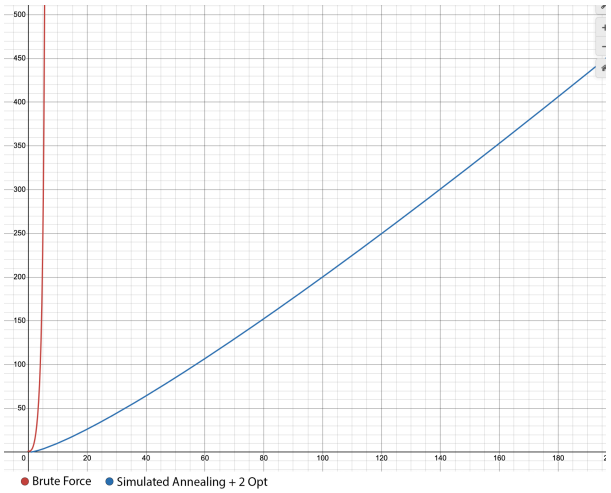
Comparison of TSP Algorithms

To improve upon the previous brute force algorithm a combination 2-Opt and Simulated Annealing are used. The original brute force method utilized the Floyd-Warshall algorithm for all-pairs shortest paths to create a set of “Virtual Edges” -- Making the graph act as if it was complete -- before generating all possible permutations to find the optimal tour. This approach guarantees finding the best solution but at a significant computational cost. Its time complexity is $O(n^3 + n!)$, making it impractical for all but the smallest problem instances, typically limited to 10-12 nodes. The memory usage is more modest, with a space complexity of $O(n^2)$ primarily due to the distance and predecessor matrices.

In contrast, the second algorithm combines Simulated Annealing with the 2-opt heuristic, offering a more scalable approach to the TSP. This method sacrifices the guarantee of optimality for the ability to handle much larger problem instances. In most cases though it finds a path that is optimal or near optimal. The time complexity of this algorithm is $O(i * n * E * log(V))$, where i is the number of iterations, n is the number of nodes, E is the number of edges, and V is the number of vertices. This complexity arises from the combination of the 2-opt swap operations,

path verification using A* algorithm, and the overall iterative nature of Simulated Annealing. While still computationally intensive, this approach is far more manageable than the factorial growth of the brute force method, especially for larger problem sizes.

The key differences between these algorithms lie in their applicability and performance characteristics. The brute force method is ideal for situations where finding the absolute best solution is critical and the problem size is very small. It's also more straightforward to implement and verify. On the other hand, the Simulated Annealing approach is far more flexible and can handle much larger datasets. It allows for a trade-off between solution quality and computation time through parameter tuning, making it suitable for a wide range of practical applications.



Comparison of SSSP Algorithms

The A* and Dijkstra's algorithms differ significantly in their approach to pathfinding. A* employs a heuristic function to estimate the cost from the current node to the target, which allows it to make informed decisions about which nodes to explore next. This heuristic-guided search often results in A* being more efficient for finding a path between two specific points, especially in large graphs. Dijkstra's algorithm, on the other hand, does not use a heuristic and explores nodes based solely on their distance from the start.

In terms of efficiency, A* has a similar time complexity to A*, being $O(E * log(V))$ as stated above, in comparison to $O(V + Elog(V))$. This means that A* generally outperforms Dijkstra's algorithm when searching for a single target, as it can often reach the goal while exploring fewer nodes. However, this efficiency comes at the cost of potentially higher memory usage in worst-case scenarios. In practice, Dijkstra's algorithm, while potentially slower for single-target searches, is more memory-efficient and excels at finding shortest paths from one node to all others, although this isn’t relevant to the SSSP problem.

Both algorithms guarantee optimal paths under certain conditions. A* finds the optimal path if its heuristic function is admissible (never overestimates the cost), while Dijkstra's algorithm always finds the optimal path to all nodes. This makes Dijkstra's algorithm useful where an admissible Heuristic doesn’t exist.

3. Conclusions in terms of Pangobat Infestation

Traveling Salesman Problem

In the context of a Pangobat infestation in Victoria and the need to efficiently distribute vaccines for the INS disease, we can draw the following conclusions about which algorithm would be more suitable.

Simulated Annealing algorithm with 2-opt would likely be the better choice for this real-world scenario. Victoria has a large number of towns and cities spread across a large area, which would translate to a very large number of nodes in our graph. The brute force algorithm's factorial time complexity would make it impractical for handling a problem of this scale, potentially taking years to compute a solution.

The brute force algorithm, with its factorial time complexity O(n!), quickly becomes intractable as the number of locations increases. For instance, with just 20 towns, there are about $2.43 * 10^{18}$ possible routes to evaluate. This number grows astronomically with each additional location, making the problem computationally infeasible for any realistic number of towns in Victoria.

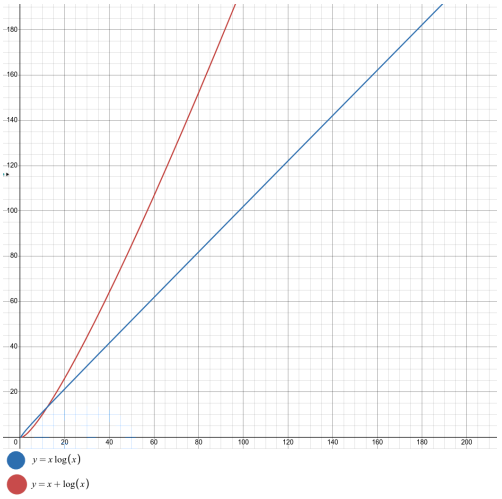
The Simulated Annealing approach, on the other hand, trades optimality for tractability. While it doesn't guarantee finding the optimal solution, it can find good approximate solutions in polynomial time. Its time complexity of $O(i * n * E * log(V))$ makes it much more tractable for larger problem instances. The potential inaccuracy of this approach is negligible in comparison to the time it may take to find an exact solution.

For these reasons, Simulated Annealing is a no-brainer. There is almost 0 advantages to the exact brute force method, which may take many decades to complete on a large number of nodes.

Single Source Shortest Path

The Pangobat infestation and spread of Itchy Nose Syndrome requires a fast response to contain each outbreak, for this use, the A* algorithm emerges as the superior choice for routing response teams. Its heuristic-guided approach allows for quick path finding, which is crucial in emergency situations where response time is critical to contain threats. While Dijkstra's algorithm guarantees the absolute shortest path, the speed and efficiency of A* in large-scale scenarios make it more practical for real-world application.

Both A* and Dijiksta's algorithms have similar time complexities, being $O(E * log(V))$ and $O(V + Elog(V))$ respectively. This means that they should theoretically run in roughly the same time. However, in practice, A often performs significantly better, especially in large-scale scenarios like the potential nationwide Pangobat infestation.



The key to A*'s superior performance lies in its heuristic function. While the worst-case time complexity of A* matches that of Dijkstra's algorithm, a well-designed heuristic can dramatically reduce the number of nodes that need to be explored. In the context of geographical pathfinding, the Haversine distance heuristic provides a good estimate of the remaining distance to the goal, allowing A* to focus its search more efficiently.

A* adaptability is a key advantage. Its heuristic function, currently using Haversine distance, can be easily modified to account for various factors such as terrain, road conditions, or even infestation density. This flexibility allows for more nuanced and realistic path planning, which may be significant if the infestation were to spread from Victoria to other regions or even overseas.

This all cultivates in A* being significantly faster to complete the problem on the given graph, as well as being much more scalable if the infestation were to spread outside of Victoria. This means that the obvious choice for which algorithm to use is A*.

4. Notes

1. The TSP algorithm can be improved by a refactor of the A* code using Dynamic programming to avoid recomputations