COP290 - Traveling Salesman Problem

Karan Aggarwal, Samanyu Mahajan

2019CS10699, 2019CS50446

1 Back Story

In continuation of our Harry Potter theme in sub-task 1, Harry Ron and Hermione are hunting *horcruxes*. 7 *horcruxes* are distributed at random locations within the maze. Since Harry had a tough time collecting the deathly hallows in sub-task 1, he doesn't want to go in the maze himself, so he decided to use the magic drone Dumbledore left for him in his will.

Assumptions

1. The Maze is initially generated as a Minimum Spanning Tree using a randomized *Prim's Algorithm.*
2. The *horcruxes* are placed at random points in the beginning.
3. The drone knows the structure of the maze and the position of *horcruxes* beforehand.

For further discussions, we will assume

$$\begin{aligned} n &= Number\ of\ nodes\ in\ the\ Graph \\ e &= Number\ of\ edges\ in\ the\ Graph \\ &= n-1\ (Since\ it\ is\ a\ tree) \\ s &= Number\ \ of\ horcruxes\ to\ collect \end{aligned}$$

In our simulation, $n = 300$, $s = 7$, $e = 299$.

Task

Calculate the shortest path from Node number 1 to Node number N (the destination) while collecting all the *horcruxes.*
Assume that the *horcruxes* are numbered from 1 to s and their node numbers are $n_1 ... n_s$.

2 Modelling

$$V = nodes\ of\ horcruxes + the\ starting\ point\ S + ending\ point\ T$$

For each node we calculate it's distance to all other nodes using Breadth First Search. Thus we construct a complete weighted graph $G = (V, E)$ where the

$$\begin{aligned} weight\ of\ each\ edge &= cost\ of\ travelling\ from\ one\ node\ to\ the\ other \\ &= distance\ between\ the\ two\ nodes \end{aligned}$$

In the maze, starting from S we need to visit each node (at least once) and end up at T.
Thus, we need to find a minimum-cost Hamiltonian $S - T$ path in G.

- **Traveling Salesman Path Problem (TSPP):** Find the cheapest Hamiltonian path starting at S and ending at T.

- **TSP:** Find the cheapest Hamiltonian Cycle (S to S).
  Note: This is simply $min[cost(\text{TSPP}(S,\, i)) + dist(i, S)]$ over all vertices i.

- **Metric TSP:** When the edges of the graph satisfy the Triangle Inequality: $\forall(i,j,k),\ d(i,j) \leq d(i,k) + d(k,j)$
  This is always true when the weights on the edges are actual distances (non-negative).

  Proof: Suppose it doesn't hold then for some nodes $i,j,k,\ d(i,j) > d(i,k) + d(k,j)$. However this means that to go from i to j, going through k gives us a shorter route which contradicts the fact that $d(i,j)$ is defined as the length of the shortest path from i to j.

Thus we can model our simulation to **Metric TSPP**.

Metric TSP is NP-hard, which means that we neither have nor know if there exists a polynomial-time algorithm to solve it. However we can use approximation algorithms which ensure us an answer which is within α times the optimal answer.

3 Algorithm used in the Simulation (Brute Force)

Algorithms

1. **Randomized Prim's Algorithm:** To generate the maze
2. **Breadth First Search:** To calculate shortest distances and paths

Data Structures

1. **Queue:** For BFS
2. **Vector**
3. **Graph:** Both, Adjacency Matrix and Adjacency List representations

Helper Functions

1. getDistances: Calculate shortest distances to a set of points from a fixed point in the Graph

Arguments

1. *src*: Source to start BFS from
2. *dests*: A vector of points to calculate distances from *src*

Algorithm Used: Breadth First Search

Time Complexity: For each *horcrux*, the distance to every other *horcrux* is calculated in a BFS starting from it. BFS is $O(e) = O(n)$.
$\implies O(s + n)$

Code Used

```
vector<int> Graph::getDistances(int src, vector<int> & dests){
        vector<bool> vis(n, 0);
        queue<int> q;
        q.push(src);
        vis[src] = 1;
        vector<int> dist(n);
        dist[src] = 0;
        while( !q.empty() ){
                int front = q.front();
                q.pop();
                for(auto neighbor : adj[front]){
                        if( !vis[neighbor] ){
                                q.push(neighbor);
                                dist[neighbor] = dist[front] + 1;
                                vis[neighbor] = 1;
                        }
                }
        }
        vector<int> result;
        for(auto u : dests){
                result.push_back(dist[u]);
        }
        return result;
}
```

2. getAdjMtr: Repeat *getDistances()* for each *horcrux* and store them in an adjacency matrix

Arguments

1. *points*: A vector of points of size s to calculate the minimum distance adjacency matrix of size ss

Time Complexity: $O(s \times (s + n))$

Code Used

```
vector<vector<int>> Graph::getAdjMtr(vector<int> points){
        vector<vector<int>> result;
        for(auto u : points){
                result.push_back(Graph::getDistances(u, points));
        }
        return result;
}
```

3. permute: Generate a matrix containing permutations of the numbers from 1 to s

Arguments

1. s: Number to permute upto

Time Complexity: Since this is a Brute-Force Algorithm, $O(s!)$

Code Used

```
vector<vector<int>> Graph::permute(int s){
        if(s == 1) return {{1}};
        auto small = permute(s-1);
        vector<vector<int>> result;
        for(auto u : small){
                for(int i=0;i<s;i++){
                        vector<int> temp(n);
                        temp[i] = s;
                        int ct = 0;
                        for(int j=0;j<i;j++){
                                temp[j] = u[ct++];
                        }
                        for(int j=i+1;j<s;j++){
                                temp[j] = u[ct++];
                        }
                        result.push_back(temp);
                }
        }
        return result;
}
```

4. getPath: Calculate shortest path from a source to a destination

Arguments

1. *src*: Source to start BFS from

2. *dest*: The end-point of the path to calculate

Algorithm Used: Breadth First Search

Time Complexity: BFS is $O(e) = O(n)$.
$\implies O(n)$

Code Used

```
vector<int> Graph::getPath(int src, int dest){
        vector<bool> vis(n, 0);
        vector<int> result;
        getPathbfs(src, vis, result, dest);
        return result;
}
void Graph::getPathbfs(int src, vector<bool> & vis, vector<int> & result, int dest){
        queue<int> q;
        q.push(src);
        vis[src] = 1;
        vector<int> parent(n);
        parent[src] = -1;
        while( !q.empty() ){
                int front = q.front();
                q.pop();
                for(auto neighbor : adj[front]){
                        if( !vis[neighbor] ){
                                q.push(neighbor);
                                parent[neighbor] = front;
                                vis[neighbor] = 1;
                                if(neighbor == dest) break;
                        }
                }
        }
        int vertex = dest;
        while(vertex != -1){
                result.push_back(vertex);
                vertex = parent[vertex];
        }
        reverse(result.begin(), result.end());
}
```

Main Function

1. set_stones: Calculate the shortest path from Node number 1 to Node number N (the destination) while collecting all the *horcruxes*.

Time Complexity:

1. The adjacency matrix is calculated (*getAdjMtr()*). $\implies O(s \times (s + n))$
2. s points are permuted (*permute()*). $\implies O(s!)$
3. For every permutation, s distances are added to calculate the shortest path. $\implies O(s \times (s!))$
4. Given the order of visiting each *horcrux*, the path path between every consecutive *horcrux* is calculated (*getPath()* is applied s times). $\implies O(s \times n)$

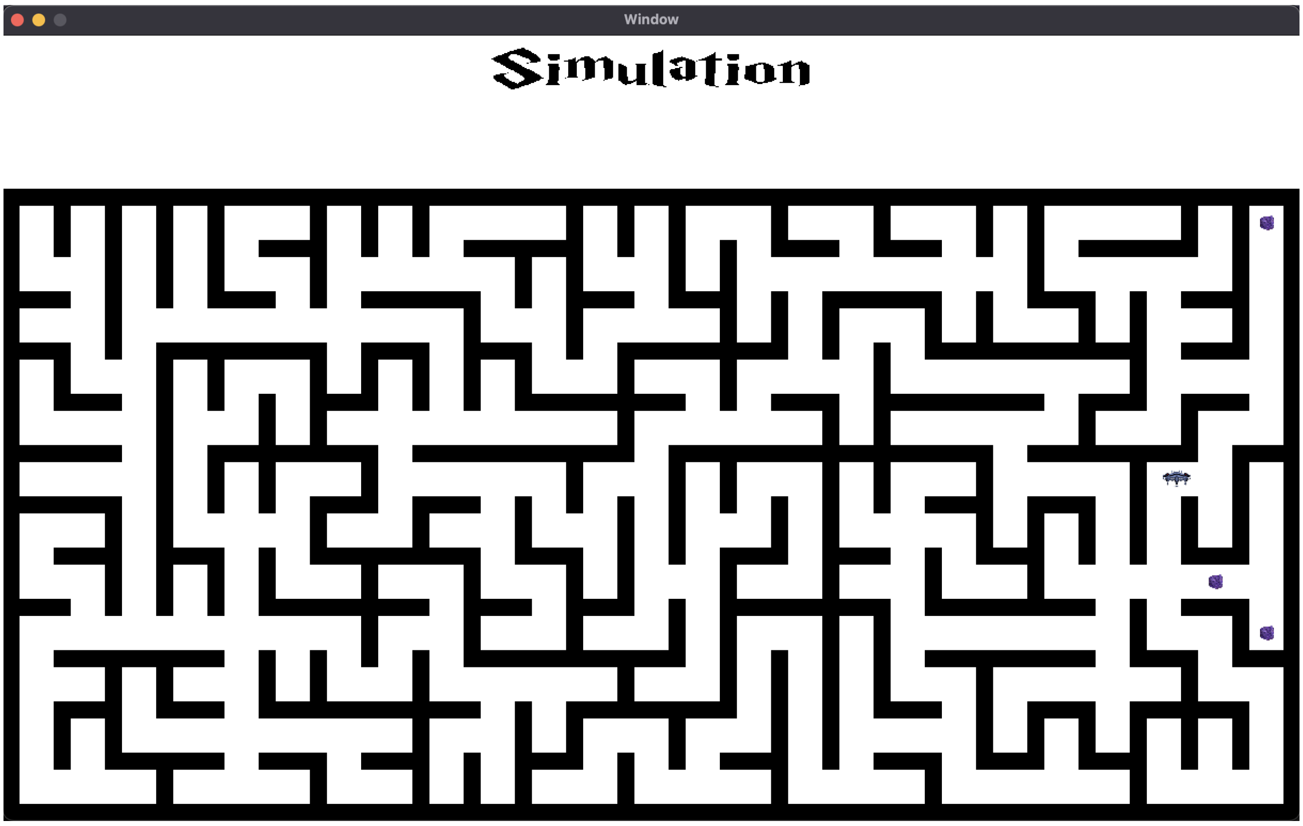
$\implies O(s \times (s! + n))$

Code Used

```
void Entity::set_stones(){
        vector<int> points;
        points.push_back(getBlock());
        for(auto & stone: * Game::entities->stones){
                points.push_back(stone->getBlock());
        }
        auto adj = Game::game_maze->graph.getAdjMtr(points);
        auto check = Game::game_maze->graph.permute(Game::entities->stones->size());
        vector<int> indices;
        int cost = INT_MAX;
        vector<int> destination = {Game::rows * Game::cols - 1};
        for(auto u : check){
                int temp_cost = adj[0][u[0]] + Game::game_maze->graph
                    .getDistances(points[u[u.size()-1]], destination)[0];
                for(int i=1;i < u.size();i++){
                        temp_cost += adj[u[i]][u[i-1]];
                }
                if(temp_cost < cost){
                        cost = temp_cost;
                        indices = u;
                }
        }
        vector<int> path_vector;
        path_vector = Game::game_maze->graph
            .getPath(getBlock(), points[indices[0]]);
        for(int i=1;i<indices.size();i++){
                auto temp_path = Game::game_maze->graph
                    .getPath(points[indices[i-1]], points[indices[i]]);
                for(int j=1;j<temp_path.size();j++){
                        path_vector.push_back(temp_path[j]);
                }
        }
        for(auto vertex: Game::game_maze->graph
            .getPath(points[indices[indices.size()-1]]
                , Game::rows * Game::cols - 1)){
                path_vector.push_back(vertex);
        }
        for(auto vertex : path_vector){
                path.push(vertex);
        }
        current = path.front();
        path.pop();
}
```

Animation

The animation was done with 60 *Frames per Second* so that it looks smooth.



A screenshot from the drone collecting all *horcruxes*

4 A Heuristics based Algorithm

We present a 2-approximate algorithm for metric TSPP ($O(s \times (s \times log(s) + n))$)

1. Obtain the MST of G (using Prim's/Kruskal).
2. Obtain MST walk from S to T visiting all vertices, using each edge at most twice. (using DFS)

3. Shortcut the MST walk to obtain the required Hamiltonian path X.

Shortcutting
Skip to the next un-visited node in the path. Thus, each vertex is visited only once in a shortcut path. Example: path: 12324. Shortcut path: 1234. Since 2 was already visited we shortcut from 3 directly to 4.

Claim: shortcutting reduces the path cost.
Proof : By the Metric constraint of the TSP problem, the distances satisfy the Triangle Inequality. So $d(x, a_1, a_2, \ldots, y) > d(x, y)$.

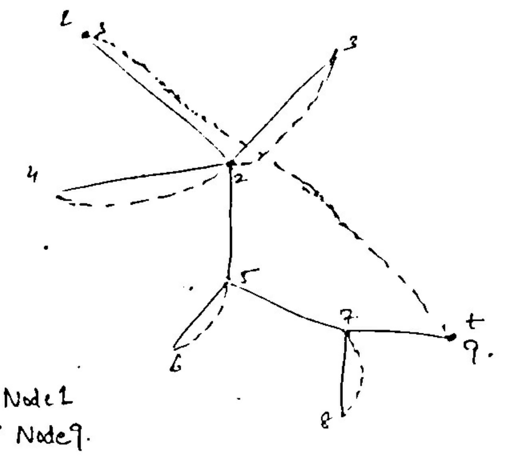
Proof of correctness of algorithm:

To show: $X \leq 2 \times OPT$. (*i.e.* $cost(X) \leq 2 \times OPT$)

- Cost of $MST \leq OPT$.
  If not, then $OPT < MST$. Remove an edge from OPT to get a spanning tree T then $T \leq OPT < MST$. This contradicts the minimality of MST.
- Cost of $MST\ Walk \leq 2 \times OPT$.
  since the walk uses each edge at most twice, $MST\ Walk \leq 2 \times MST \leq 2 \times OPT$
- $X \leq MST\ Walk$.
  This follows from the fact that X is a shortcut of MST Walk and shortcuts reduce cost.

Thus we have, $X \leq MST\ Walk \leq 2 \times OPT$.
$\implies X \leq 2 \times OPT$

We also need to prove that such an MST walk form $S - T$ exists.
To do so, we add a dummy edge of 0 cost from S to T and double every edge except those on the direct path from S to T to obtain a graph say G'. Note that every vertex (including those in the $S - T$ path) now have an even degree, so G' is *Eulerian* and therefore admits a Euler tour. In this tour, set S as starting node, T as ending node and drop the dummy edge to get an MST walk from $S - T$ which uses each edge at most twice. Thus X is a Hamiltonian $S - T$ path whose cost $\leq 2 \times OPT$.



s: Node 1
t: Node 9.
Double every edge except those on s-t path
and add a dummy edge of cost 0, from s to t

1 2 4 2 3 2 5 6 5 7 8 7 9 is the MST Walk.
Shortcut the MST walk to get

X: 1 2 4 3 5 6 7 8 9. as the required
hamiltonial s-t path.

Note that steps 2 and 3 of the above algorithm can be clubbed together in implementation as follows:

We do a DFS of the MST ensuring that for any node x whose subtree contains T, we visit neighbors of x before x in the order in which we do DFS. This ensures that the subtree of T is the last to be visited.

Further in step 3, we do not shortcut T, since we need it to be the end point of the Hamiltonian path. This implies that T may occur any odd number of times in X. instead we can shortcut T the first time it occurs and then visit it from the last node. The proof for 2-approximation still holds.

Algorithms Used

1. **Kruskal's Algorithm:** To calculate the MST
2. **Depth First Search:** To calculate the order of visiting *horcruxes*

Data Structures

1. **Disjoint Set Union**
2. **Vector**
3. **Graph:** Both, Adjacency Matrix and Adjacency List representations

The Main Approximation Algorithm (*TSPP*)

Task: To calculate a nearly shortest path for the drone, bound by twice the length of the optimal path

Arguments

1. *adj*: The adjacency matrix of s *horcruxes* along with the start and end points of the drone

Return type: A vector of size $s + 2$ with the order of visit of *horcruxes*, under the constraint that the start and end node of the drone appear at the start and end of the vector respectively

Steps

1. Input the adjacency matrix. $\implies O(s^2)$
2. Use Kruskal's Algorithm with a Disjoint Set Union structure to get the MST. $\implies O(s^2 \times log(s))$
3. Run DFS on the MST to get the order of visit required. $\implies O(s)$

Time Complexity: $\implies O(s^2 \times log(s))$

Code Used (Implementation)

```
#include <bits/stdc++.h>
using namespace std;

int start = 0, last = 8, s = 9;
vector<vector<int>> adj;
vector<vector<int>> mst_adj(s, vector<int>(0));
vector<int> contains_end(s);
vector<int> walk;
vector<int> path;
vector<bool> vis(s);


bool comp(pair<int, int> &a, pair<int, int> &b) {
        return adj[a.first][a.second] < adj[b.first][b.second];
}

vector<int> parent, size;
void make_set(int n){
        parent.assign(n, 0);
        size.assign(n, 1);
        for(int i=0;i<n;i++) parent[i] = i;
}
int find_set(int v) {
        if (v == parent[v]) return v;
        return parent[v] = find_set(parent[v]);
}
void union_sets(int a, int b) {
        a = find_set(a); b = find_set(b);
        if (a != b) { if (size[a] < size[b]) swap(a, b); parent[b] = a; size[a] += size[b];}
}
int dfs_init(int vertex, int last){
        vis[vertex] = 1;
        if(vertex == last){
                contains_end[vertex] = 1;
                return 1;
        }
        int ans = 0;
        for(auto u: mst_adj[vertex]){
                if(!vis[u]){
                        ans = max(dfs_init(u, last), ans);
                }
        }
        contains_end[vertex] = ans;
        return ans;
}
void dfs(int vertex, int last){
        vis[vertex] = 1;
        walk.push_back(vertex);
        int left = -1;
        for(auto u:mst_adj[vertex]){
                if(!contains_end[u] and !vis[u]){
                        dfs(u, last);
                }
                else if(!vis[u]) left = u;
        }
        if(left != -1) dfs(left, last);
}
int tsp(vector<vector<int>> adj_param){
        adj = adj_param;
        vector<pair<int, int>> edges;
        for(int i=0;i<s;i++){
                for(int j=0;j<s;j++){
                        if(i < j){
                                edges.push_back({i, j});
                        }
                }
        }
        make_set(s);
        sort(edges.begin(), edges.end(), comp);

        for(auto edge:edges){
                if(find_set(edge.first) != find_set(edge.second)){
                        mst_adj[edge.first].push_back(edge.second);
                        mst_adj[edge.second].push_back(edge.first);
                        union_sets(edge.first, edge.second);
                }
        }
        dfs_init(start, last);
        vis.assign(s, 0);
        dfs(start, last);
        for(auto u:walk){
                if(u != last) path.push_back(u);
        }
        path.push_back(last);
        for(auto u:path){
                cout<<u<<" ";
        }
        cout<<endl;
        return 0;
}
```

Overall Algorithm

1. The adjacency matrix is calculated (*getAdjMtr()*). $\implies O(s \times (s+n))$
2. The Approximation Algorithm is used (*tsp*). $\implies O(s^2 \times log(s))$
3. Given the order of visiting each *horcrux*, the path path between every consecutive *horcrux* is calculated (*getPath()* is applied s times). $\implies O(s \times n)$

$\implies O(s \times (s \times log(s) + n))$

5 Extras

1.5-approximate algorithm: (*Christofides' Algorithm*) $O(s^3)$

Recall what we did in proof for the 2-approximate algorithm

1. add some edges(including a dummy $S-T$ edge) to MST of G to obtain a graph G' which is Eulerian.
2. obtain a $S-T$ MST walk from Eulerian tour of G' by removing the dummy edge
3. Shortcut this walk (ensuring that T remains the end point) to obtain the required solution path X

To obtain the Eulerian graph G we need the degree of each vertex to be even. To accomplish that we simply doubled each edge of the MST, which gave rise to the factor of 2 in the algorithm.

Instead we now only even out/ fix the degree of all odd vertices (vertices with odd degree). Note that in any graph the sum of the degree of all vertices is even, hence the number of odd vertices is also even.

This enables us to construct a min cost perfect matching of all these odd vertices, say M, then $G' = MST + M + dummy\ edge$ has the degree of every vertex as even.

Christofide's Algorithm does precisely that by claiming that the matching $cost(M) \leq 0.5\ OPT$ thus we get the 1.5-approximation.

Optimal Algorithm using DP $O(s^2 \times 2^s)$

[We saw the $O(s!)$ naive algorithm.]

Let us index the vertices form $1...s$. S (starting node) is indexed to 1. T (ending node) to s.

Define:

$$f(i) = cost\ of\ Min\ cost\ Hamilitonian\ path\ from\ 1\ to\ i.$$

To find: $f(s)$

$DP(A, i) = cost\ of\ min\ cost\ path\ visiting\ every\ vertex\ in\ a\ set\ A$(which includes 1 and i), starting from 1 and ending at i.

If $size(A) = 2$, then A is simply $\{1, i\}$, $DP(A, i) = d(1, i)$

Else if $size(A) > 2$, then $DP(A, i) = min[DP(A - \{i\}, k) + d(i, j)]$ where j is in A, and is different from 1 and i.

Number of Sub-problems: Let the $size(A) = x$ then there are $\binom{(s-2)}{(x-2)}$ possible choices. (we exclude 1 and i). Summation of this over x gives $O(2^s)$

Each problem takes $O(s)$ time to solve

Time complexity: $O(s^2 \times 2^s)$

6 Analysis

Run-time: The optimal algorithms are exponential and the approximate ones are polynomial.

Optimality: Both approaches were compared for the given values of the constants, in about 20 random trials.

$$Average\ cost\ percentage\ increase\ of\ the\ approximation = 4.9705\ \%$$

This is far better than the theoretical bound of 100 % increase as proved for the 2-approximate algorithm. This is simply because in the simulation $s = 7$, is small.

7 References

- https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15859-f11/www/notes/lecture19.pdf
- http://www.cs.tufts.edu/~cowen/advanced/2002/adv-lect3.pdf