COVID Simulation of CoviWars

Shrey J. Patel
2019CS10400

Rahul Chhabra
2019CS11016

June 2021

1 Backstory

The Ruhan Lab is leaking a large amount of corona virus samples which have been taking over the world. It is of utmost importance to destroy the lab and stop the virus as it has left the world in a massive social and economical slump. Fortunately, our hero Pac-man has willingly accepted this responsibility.

But, surviving the pandemic is not easy, and reaching the lab is harder still. The pacman needs to be fully vaccinated to even reach the lab, for which he needs to collect various vaccines scattered randomly across the maze before reaching the lab. And we don't have much time left before the virus wipes off the world, so the pacman needs to complete this task in the minimum possible time.

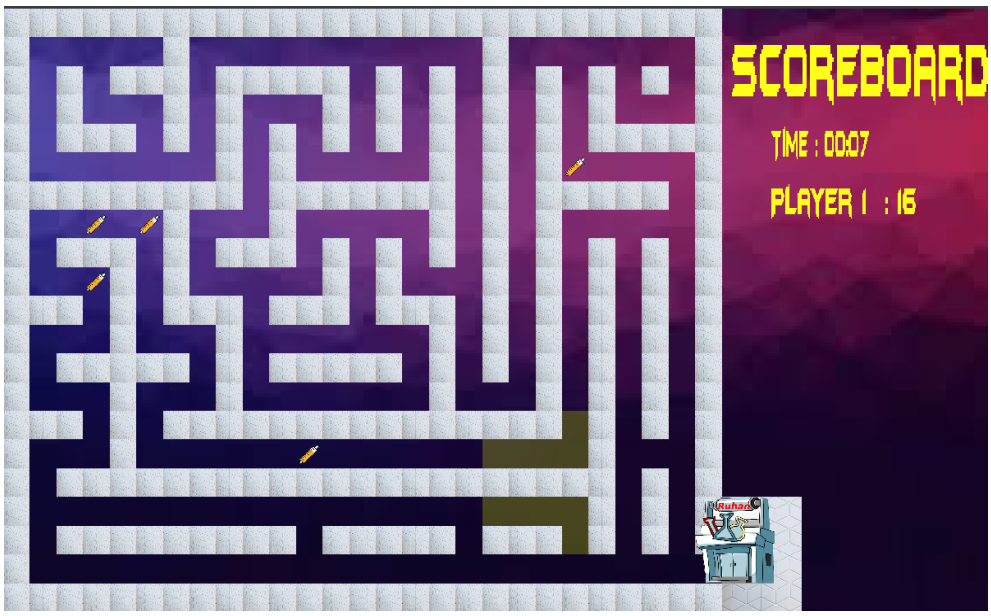
2 The Problem Statement

Formally, given a graph G(V,E), source vertex S, a destination vertex D and some other chosen vertices V_i, we need to find a path P of minimum length which starts from S, visits all the vertices V_i, and finally reaches D in minimum possible time. Note that, given that the speed of pacman is constant, the shortest path will be the same which takes minimum time to traverse. So, this is a direct application of the Travelling Salesman problem(TSP).

2.1 Structure of our simulation

We have used the structure of our game ***CoviWars*** for the simulation. The map of the world consists of a maze, and the graph of the world can be constructed with empty cells of the maze as vertices and the edges join the adjacent empty cells. All the edges are undirected and of equal weights of 1.

The initial position of the pacman (here, the top left corner of the maze) will be taken as the source vertex, while the Ruhan Lab will be the destination vertex (the bottom right corner). And the vaccines will be at the intermediate vertices which the pacman needs to visit.



2.2 Construction of maze

The maze is basically stored as a 37*21 matrix, whose cells store whether the corresponding cell of the maze is a wall or an empty cell. We have implemented a modified version of Depth First Search to create a maze from the matrix.

2.2.1 Algorithm: Randomised DFS

1. Initially, the whole matrix is unexplored, i.e. the whole maze consists only of walls and we choose a random starting vertex to start exploring. When the algorithm ends, the explored areas of the maze will be the empty cells through which the pacman can travel.

2. At every iteration of DFS, we check if the neighbours of the last visited cell are unexplored, and out of the unvisited neighbours(maximum 4), we randomly choose a vertex to explore next.

3. This algorithm continues until there are no more cells to visit. This is the normal version of randomised DFS.

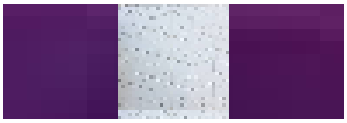
4. However, note that if we follow this algorithm then the whole matrix will be empty at the end. So, we modified the algorithm so that there are alternating rows of walls and empty cells at the end so that the final graph looks like a maze.

5. For that, whenever we check for neighbours of the last visited cell, we don't consider the immediate neighbours, but the ones at a distance of 2 in each direction. (See the figures below)

   Suppose the cells with purple background are already explored and the white cells are walls,

   Out of the above three cells, if the leftmost one is the last visited cell, then the third cell(i.e. the rightmost one in the figure) will be considered as its neighbours, not the middle one.

   So, the next explored cell will the third cell.

   But, to create a path from the last cell to the next cell, the middle wall is removed.

   Thus, at the end of this modified algorithm, not all walls are removed. Only the walls corresponding to the explored cells and the middle walls are removed as described above.

2.2.2 Removing Dead-ends

Because of DFS, the path formed is still a tree which is very difficult for the pacman to navigate, especially in the presence of enemies. So, we at the end of DFS, we keep track of dead-ends (i.e. those empty cells, which have only one open neighbour). And for such cells, we create an alternate path be removing one of walls in its neighbourhood.

3 Solution

We propose a solution to the above problem using the Dijkstras' algorithm.

3.1 Dijkstras' Algorithm

3.1.1 Algorithm

Given a graph G(V,E) with non-negative edges, and a source vertex 's', the Dijkstras algorithm finds the length of the shortest path from the source to every vertex. For this, we store these distances for every vertex in an array. During the algorithm, those vertices with finite distance values have been explored, while the unxplored vertices have ∞ distance value. Initially, only the source vertex has been explored, so only the distance value of 's' is finite (which is in this case zero as it is at zero distance from itself).

In this algorithm, we maintain two sets of vertices, a **"known region"** set, which contains vertices whose shortest distance value has been already determined and the rest of the vertices are contained in the **"unknown region"**. Initially, the known region consists only of the source vertex. In every iteration of the algorithm, we update the distance values of the neighbours of all the "known" vertices (i.e. for all the outgoing edges from all the vertices in the known region). This is also known as *edge relaxation*.

If we denote the known region with R, then $\forall$ u$\in$R, $\forall$ edges u$\rightarrow$v,

$$\textbf{dist[v] = min(dist[u] + w(u,v), dist[v])}$$

where w(u,v) is the weight of the edge u$\rightarrow$v

Thus, if d(s,t) denotes the correct shortest distance of t from s, and dist[t] is the value stored in the array at any point of the algorithm: $\textbf{d(s,t)} \leq \textbf{dist[t]}$ and only at the end of the algorithm, the equality holds for all vertices.

Additionally, after every such edge is relaxed, the unvisited vertex with the shortest distance value (from the unknown region) is included in the known region and all its edges are "relaxed" and so on. The algorithm terminates when for all the vertices, d(s,v) = dist[v], or in other words, the unknown region becomes empty.

3.1.2 Proof

To prove that the Dijkstras' algorithm finds all distances from a given source vertex correctly, it is sufficient to prove that at every step of the algorithm, correct distance is assigned to at least one vertex of the graph, particularly the one which is extracted from the unknown region i.e. the one with the shortest distance value in the unknown region.

Denoting the known region as R and the unknown region as U, we will prove by contradiction i.e. assume that for the vertex with the shortest distance value in the unknown region, say some v $\in$ U, d(s,v) $<$ dist[v].

Note that if v is the vertex with the shortest distance value, then v has been explored in which case dist[v] is finite. And in other words, $\exists$ a path P of length dist[v] from s to v. From our assumption, this is not the shortest path i.e. $\exists$ some other path P' from s to v with length strictly less than dist[v]. There can be two cases for such a path:

1. There is a direct edge e from a vertex in the known region, say $u \in R$, to v, such that the shortest path P' includes this edge. Length of this path is d(s,u) + w(u,v). Also, $u \in R$, and so d(s,u) = dist[u], i.e. **A subpath of an optimal path is also optimal** (proof is trivial). So,

   $$\mathbf{len(P') = d(s,v) = dist[u] + w(u,v)}$$

   Note that because e is a direct edge between R and U, this edge would have been relaxed in the most recent iteration of this algorithm, i.e.

   $$\mathbf{dist[v] = min(dist[u] + w(u,v), dist[v])}$$

   $$\Rightarrow \mathbf{dist[v] = min(d(s,v), dist[v])}$$

   $$\Rightarrow \mathbf{dist[v] = d(s,v)}$$

   The last step follows from the invariant of the algorithm: $\mathbf{d(s,t) \leq dist[t]}$ for any vertex t. We have a contradiction since we assumed that d(s,v) < dist[v].

2. Consider a direct edge e' between R and U which connects some vertex $u \in R$ with some vertex $t \in U$ and a path Q from t to v. Such a path exists because v has already been explored. Let the shortest path P' be such that it includes the shortest path T from s to t and the path Q, i.e. $P' = T \cup Q$. So, len(P') = len(T) + len(Q).

   Now, from case 1, for vertex t, dist[t] = d(s,t) = len(T). Also, $len(Q) \geq 0$ as there are only non-negative edges. So,

   $$\mathbf{len(P') = d(s,v) = d(s,t) + len(Q)}$$

   $$\Rightarrow \mathbf{d(s,u) = dist[t] + len(Q)}$$

   $$\Rightarrow \mathbf{d(s,u) \geq dist[t]}$$

   Now, from our initial assumption, out of all the vertices in the unknown region, u has the shortest dist value, so $\mathbf{d(s,v) \geq dist[t] \geq dist[u]}$ which is a contradiction since we assumed that d(s,v) < dist[v].

Hence we proved that d(s,u) = dist[u] and so at every step, the Dijkstras' algorithm correctly sets the shortest distance value of at least one such vertex. And so at the end of at most $|V|$ iterations, the algorithm correctly finds the shortest distance value from s for every vertex.

3.1.3 Data Structures

1. **Adjacency List:** Used for storing the graph. Stored as an array of vectors for efficiently retrieving the neighbours of any vertex during edge relaxation.

2. A vector to store the current dist values of all the vertices.

3. A set to store the known vertices.

4. **Priority Queue:** For storing the "unknown" vertices. We have used a binary min heap based implementation of priority queue for efficient insertion, deletion and update of dist values all of which can be done in log(n) where n is size. While extracting the vertex with minimum dist value is a constant time operation. (Note that searching is expensive in this implementation)

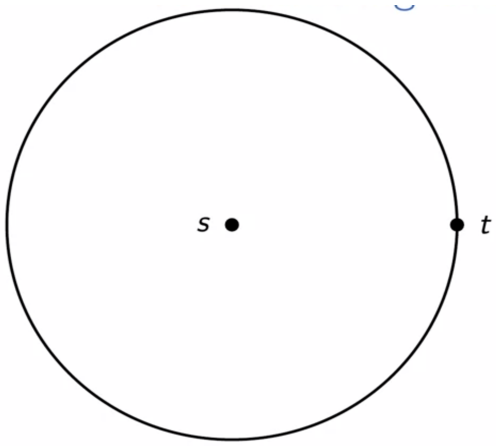
3.2 Implementation

Our problem statement requires us to find the length of the shortest path such that all vaccines are collected. For that, we have implemented a greedy strategy using the Dijkstras' algorithm. Given a set of intermediate vertices to visit, we can start the Djikstras' algorithm from the source vertex. But this time, we don't need to find the shortest distances to every vertex. It is sufficient to find one of the intermediate vertices. Once such a vertex is found, we travel to it from the source vertex and repeat the algorithm with the current vertex as the new source. This is repeated until we cover all the intermediate vertices.

The total length of the path will be the sum of all the paths constructed in this algorithm. We claim that this path will be the shortest (and therefore the fastest) path for covering all the points.

3.2.1 Proof

Intuitively, in Dijkstras' algorithm, we can think of the region of explored vertices i.e. known region as a circle of increasing radius with the source vertex at its centre. Thus the vertices with shorter distance from source are explored first. In our case, if our target vertex is t then we stop the algorithm as soon as t is explored, or in other words, t appears on the circumference of this circle.



So, the Djikstras' algorithm always explores the vertices in the order of non-decreasing distance values. Because, in every iteration of the algorithm, we always extract the vertex with the minimum distance value from the unknown region.

Formally, we need to prove that if if some vertex u is explored before v then d(s,u) $\leq$ d(s,v). For this, we must first prove a lemma.

Lemma: At any point of time in Dijkstras' algorithm, $\forall$ vertices x $\in$ R and $\forall$ vertices y $\in$ U, dist[x] $\leq$ dist[y].

Proof of Lemma: We will prove this by induction on the size of R (number of explored vertices).

1. **Induction Hypothesis:** Let the size of R be n. Then $\forall$ vertices x $\in$ R and $\forall$ vertices y $\in$ U, dist[x] $\leq$ dist[y]. Without loss of generality, we assume that both R and U are non-empty. Because the proof of such cases is trivial.

2. **Base Case:** Initially i.e. when n=1, R only contains the source vertex s where dist[s]=0. While for all other vertices dist value is ∞ and zero $< \infty$. Therefore dist[s] is less than dist values for all vertices in U. So, the induction hypothesis holds for n=1.

3. **Induction Step:** Let the current size of R be n. Now, in some iteration of Dijkstras' algorithm, one vertex with the smallest distance value from U will be included in R. So, the size of R will increase by 1 i.e. n+1. If we prove that the induction hypothesis holds even after this step, we are done.

   From induction hypothesis for n, we know that all vertices of R have lesser distance values than those in U. Let the next vertex to be added to R be z. So, z will have the smallest dist value in U i.e. dist[z] $\leq$ dist[y] $\forall y \in U, y \neq z$. So, before relaxation of outgoing edges of z, we have,

$$\textbf{dist[x]} \leq \textbf{dist[z]} \leq \textbf{dist[y]}\ \forall x \in R \textbf{ and } \forall y \in U, y \neq z$$

$$\Rightarrow \textbf{dist[x]} \leq \textbf{dist[y]}\ \forall x \in R' \textbf{ and } \forall y \in U$$

where R' = R $\cup$ z and U' = U - z.

Now, after the relaxation of all edges of z, for all vertices y $\in$ U, such that the edge z $\rightarrow$ y $\in$ E :

$$\textbf{dist[y] = min(dist[y],dist[z] + w(z,y)}$$

If dist[y] remains same, then we have the same case as when there was no edge relaxation in which the induction hypothesis holds. For the case when dist[y] = dist[z] + w(z,y), note that dist[y] $\geq$ dist[z], because w(z,y) $\geq$ 0, because we have assumed non-negative edge weights. So, again we have that all vertices of R' will have lesser distance values than those in U.

End of proof of lemma

Now, back to our original proof, where we need to show that the vertex which will be explored first will have shorter path from the source. Now, assume that the vertex u is explored before v. Then at point of the algorithm, u $\in$ R and v $\in$ U. Without loss of generality assume that the vertex v is the next vertex to be explored. From the lemma, we have dist[u] $\leq$ dist[v].

Note that, once explored the dist value of the vertex is the actual length of the shortest path i.e. dist[u] = d(s,u). Also, after exploring v, we have dist[v] = d(s,v). Thus, we proved that d(s,u) $\leq$ d(s,v).

Hence proved that our algorithm finds the optimal path to cover all the intermediate points in the minimum possible time.

3.3 Runtime Analysis

There are two major operations in every iteration of the Dijkstras' algorithm:

1. **Extracting the vertex with the minimum distance value from the unknown region**: The unknown region U is a binary heap based priority queue. So, retrieving the vertex is O(1) but deleting it from the queue is O(log n). Now, the maximum size of U is $|V|$, so the worst case time for extracting one such vertex is O(log $|V|$). Every vertex is explored exactly once by this method. So, the total time taken is O($|V|$ log $|V|$).

2. **Edge Relaxation**: Every edge is relaxed exactly once. And for relaxing one edge, the distance values either remain same or decrease. If they decrease, then we need to update the priority value of that vertex in the priority queue which takes O(log $|V|$) as mentioned in point 1. So, the total time in this case if O($|E|$ log $|V|$).

Thus, the total runtime of Dijkstras' algorithm for a given source vertex is O((|E| + |V|) * log |V|).

Now, in our case, we require multiple applications of this algorithm with the intermediate points as source vertices. Also, we require the running time in terms of the dimensions of our maze. Let the number of rows of cells be y and the number of column of cells be x. So, the vertices and edges both will be O(x*y). Now, if the number of intermediate points are z, then the total running time of our simulation is ***O(z * x * y * log(x*y))***.

4 Challenge: The Smart Virus

The scientists at Ruhan Lab have somehow been informed of our hero's efforts to overthrow their evil plan. The pacman may be smart, but the people at the lab are definitely smarter and so they have decided to reverse engineer a smarter virus to prevent the pacman from collecting all the vaccines. This virus is so smart that in addition to knowing the whole map(maze) like pacman, it also has the ability to always chase pacman wherever it goes by always choosing the optimal path (i.e. the shortest path between the pacman and the virus).

4.1 Problem Statement

In the first part of the simulation, we improved upon the AI of the pacman, now we improve the AI of the enemy.

Formally, we plan to use the Dijkstras' algorithm dynamically, in the sense that the enemy knows the optimal path towards the pacman at every instant of the simulation, because the pacman and the enemy both are moving, so the source and destination of the path are constantly changing. We need to be able to predict the shortest path for every pair of source and destination.

We will assume that the pacman and the enemy have the same travel speed. The same algorithm and explanation can be extended for different speeds with some modifications.

4.2 Solution

The most obvious solution of this problem is to implement the Dijkstras' algorithm at every fixed intervals of time when the enemy as well as the pacman have crossed one cell (or in other words, the source and destination of the algorithm change), with the enemy's position as the source vertex and the pacman's position as the destination vertex. *One important observation here which can be used later for optimising this algorithm is the fact that because all the edges are undirected, the source and destination are interchangeable.*

But even a single application of Dijkstras' algorithm is expensive and therefore running it after such small intervals will affect the smoothness of the simulation which will then appear highly mechanical. To avoid this, we propose the **pre-computation and storage of the shortest paths for every pair of**

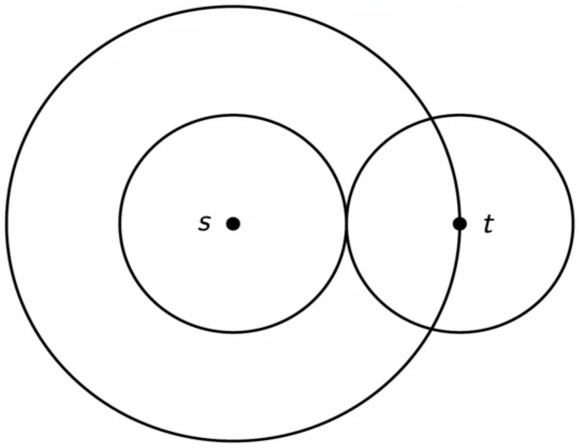
source and destination. By doing this, the enemy can simply provide its own and the pacman's position at any point of the simulation and in constant time we can obtain the path that the enemy needs to follow to reach the pacman in shortest time.

5 Improvements

At this point, both the algorithms work from the point of view of correctness. However, there is a lot of room for improvements in time as well as space complexity. Some of the possible optimisations are:

1. One of the most obvious observation that can be exploited to improve the Dijkstras' algorithm is the undirectedness of the graph. So, the shortest path from some vertex u to another vertex v is the same as the shortest path from v to u. So, we can use a **bi-directional variant** of the Dijkstras' algorithm in which instead of conducting a uni-directional search starting from the source vertex and ending on the target vertex, we conduct a bidirectional search: a forward search from source and a backward search from target to **"meet-in-the-middle"**.

   Following the same visualisation as before, we can think of the search proceeding as two circles increasing in radius, centered at the source and the target, until these circles meet i.e. touch each other.



   The circles in the visualisation roughly enclose the number of vertices which are explored before the shortest path is determined. The total area enclosed in bi-directional search is half the amount of area enclosed in unidirectional search suggesting that it is only twice as fast as the earlier algorithm which is misleading, since this is the case only with the graphs which are arranged as circles in terms of edges. In very dense graphs like a

social network, where the edges span across multiple layers of such circles, this algorithm does much better than in just half the time.

2. There is a still better algorithm which works on the principle of **"Directed Search"**. Note that in a general weighted graph, there is no sense of direction which means that when we start searching for the target vertex from the source, we are not sure which direction to proceed, i.e. there is no directional relation between the source and the target until the search reaches the target. Thus the analogy to growing circles.

   But what if we assign some artificial sense of direction to the vertices. In such cases, we can eliminate the exploration of many unnecessary vertices and proceed only in some predicted direction with some margin of error until we reach the target. This is called directed search. This can be done with the help of potential functions assigned to each vertex where the value of this function quantify that vertex's proximity to the target. This is actually the graph-search algorithm that is used by most navigation services as it uses very less space and time.

3. All of the above optimisations were targeted at optimising the Djikstras' algorithm. But our main target in the second part of our simulation is to precompute and store shortest paths between every pair of source and destination. Now, let's compute the space complexity in this algorithm:

   Total number of source-destination pairs = $O(|V|^2)$
   Maximum length of any path = $O(|V|)$
   Total space complexity = $O(|V|^3)$

   Now, if we consider the side n of maze as our parameter, then the space complexity scales as $O(n^6)$ which is massive. This makes it obvious that we cannot store the whole path. Fortunately, we don't need to and even incorrect if we do so. Recall that because of the fact that the enemy and our hero are constantly moving, the source and destination and therefore the whole path will change before the enemy has the chance to travel through the whole path. So, why then, store the whole path? It is **sufficient to just store the direction that the enemy has to take (i.e. the next vertex that the enemy needs to visit) to get closer to the pacman**, because the next shortest path may change very drastically and so it would be a waste if we would have stored the whole path. The space complexity in this case is $O(|V|^2)$ which is asymptotically better.

   Proof:
   We need to prove that through this strategy, the enemy approaches the enemy or remains at the same distance until the next cell i.e. in no case does the distance between the enemy and the pacman increases.

Consider that initially the enemy is at the vertex s and the pacman is at vertex t. And the next cell that the pacman visits is t' while the one that enemy visits is s'. Formally we need to prove that d(s,t) $\geq$ d(s',t').

Consider that the initial shortest path is P and the shortest path afterwards is P'. Note that s,t $\in$ P and the algorithm guarantees that s' $\in$ P. Now there can be two cases:

(a) P' = P - s $\cup$ t'. Then,

$$\textbf{len(P')} = \textbf{len(P)} + \textbf{w(t,t')} - \textbf{w(s,s')}$$

$$\Rightarrow \textbf{len(P')} = \textbf{len(P)} \Rightarrow \textbf{d(s',t')} = \textbf{d(s,t)}$$

This holds in our case because all the edges in our maze have the same weight.

(b) Let some path K = P - s $\cup$ t'. Then K $\neq$ P'. Note that for P' to be the shortest path in the new case, len(P') has to be less than or equal to the length of the path K or else K will be the shortest path instead. Thus,

$$\textbf{len(P')} \leq \textbf{len(K)}$$

$$\Rightarrow \textbf{len(P')} \leq \textbf{len(P)} \Rightarrow \textbf{d(s',t')} \leq \textbf{d(s,t)}$$

Hence proved for both cases, that the virus approaches the pacman.

4. Now, another obvious optimisation which improves the time complexity of the pre-computation comes from the fact that**any subpath of the optimal path is also optimal**. So, if we have found an optimal path P between any two vertices s and t then we have also indirectly determined the shortest path between any intermediate vertices on path P between s and t. More formally, if len(P) = d(s,t) and there is some path P' with u as the source and v as the destination such that P' $\subseteq$ P, then len(P') = d(u,v).

   So, while computing the path for the pair s,t we can compute the same for the pair u,v as a sub-problem. This heuristic saves many redundant computations.