COP290 ASSIGNMENT2 SUBTASK2

PARTH GUPTA(2019CS10380)
ANANJAN NANDI (2019CS10325)

21 May 2021

1 The Problem

We are given a 2-D maze with dimensions $N \times N$ tiles. There are some blocked cells (maze walls) and some unblocked cells. We cannot pass through a blocked cell. An unblocked cell is either normal or broken as defined in the Notation section. There are M treasures in the grid. We need to move from our start point, take all the treasures and return to our start point. We need to minimize the cost of the path that we take. The cost of a path is defined in the Notation section. We imagine the problem as a Steiner Travelling Salesman Problem and use the Simulated Annealing heuristic to solve it. For our simulation, $N = 50$ and $M = 20$.

2 Notation

Any unblocked cell is either normal or broken. We define a weight for each cell C denoted by $w(C)$. If C is blocked then $w(C)$ is ∞, otherwise, if C is a broken cell then $w(C)$ is 10, else $w(C)$ is 1. The cost of a path $C_1, C_2, \ldots, C_n$ is 0 if $n = 1$ and $w(C_2) + w(C_3) + \ldots + w(C_n)$ otherwise. We will assume a path to be valid if and only if all the cells in the path are unblocked.

3 Maze Generation

3.1 Basic Structure

We take a 50×50 grid which has all cells blocked initially. We take all the cells which have both x-coordinate and y-coordinate even and unblock them. We need to ensure that all unblocked cells are connected to each other which we describe in next subsection. Two cells are adjacent if they share a cell border. Two unblocked cells C_1 and C_k are connected if we have a path $C_1, C_2, \ldots, C_k$ such that C_i and C_{i+1} are adjacent for $1 \leq i \leq k-1$ and C_j is unblocked for $1 \leq j \leq k$.

3.2 Algorithm

We have used a modified Kruskal MST algorithm to generate a random maze. We take the unblocked cells as vertices of a graph. Initially, there are no edges. We add edges in this graph. An edge $(v1, v2)$ can be added if the Manhattan distance($\mid v1.x - v2.x \mid + \mid v1.y - v2.y \mid$) is 2 and they have either the same x-coordinate or same y-coordinate. If we add an edge from $v1$ to $v2$, we unblock the cell that separates $v1$ and $v2$. We need a random maze, not a minimum spanning tree. So, we take edges at random rather than in sorted order (which we do in standard Kruskal's algorithm) and add the edge to the graph (corresponding to the MST) if the corresponding vertices are in different components. The maze generated has a tree-like structure, but we want more sparse (there should only be a small number of blocked cells) maze for our purpose. We randomly select some blocked cells and unblock them. The blocked cells selected are such that if they are unblocked, they are connected to the tree-like structure. This can be ensured by checking that atleast one neighbour is unblocked. We have now generated our required 2-D maze.

3.3 Random Elements of Maze

We select an unblocked cell at random as the position from where we start(the initial location).
We divide the maze into 25 blocks, each of dimension 10×10 tiles. In each block, we select 5 unblocked cells at random and break them. Further, in each block we select an unblocked cell randomly and place a treasure there. Out of the 25 cells of the second type, we randomly choose 20 of them to contain our treasure. This is done to prevent uniformity in the targets for TSP. So, there are 20 treasures and 125 broken cells.

3.4 Preprocessing of Shortest Path

We implement a multi-source Dijkstra for finding the shortest distance between all possible pairs of points. Using the shortest distances, we can find the shortest path from any point to any other point through backtracking. We store the shortest path between all vertices containing treasure. If we had defined the weight of a normal cell as 0, then we could've used 0-1 BFS (it would decrease the time complexity) but it was more suitable to define a positive weight to each cell for the purposes of backtracking.

4 Simulated Annealing

4.1 Motivating Idea

Simulated Annealing exploits quantum mechanics and the law of entropy to find an optimal solution. The state space of the problem is modeled as the energy states of a metal. The "energy" of a state is defined by the scoring

metric of the problem, in our case it is the total cost of the path defined by the state. We then simulate the cooling down of the metal. We use the fact that as the metal cools, it tries to reach a lower energy state. Therefore, our cost should be lowered. From quantum mechanics, we know that minimizing the energy of a system is a combinatorial problem for discrete systems. Therefore, in the case of our problem, the state space is the 20! possible orders of visiting the vertices containing the treasures. We know from thermodynamics that the transition probability $P(T, e_1, e_2)$ of transition from a state having energy e_1 to a state having higher energy e_2 is $e^{\frac{(e_1-e_2)}{kT}}$ at temperature T where k is the Boltzmann constant. We accept a transition from state S to state T either if $cost(T) < cost(S)$ or with probability $P(T, score(S), score(T))$. After a few iterations, we consider the energy of the system to be lowered and reduce the value of T. This continues either until a target temperature is reached or the minimum cost converges.

4.2 Algorithm Implementation

First we generate a random order to take the treasure. Let the sequence generated be S and the associated cost be C. We define a variable T(for temperature) which is initially 1.0 and another variable $k = 10$. Then we run a loop with 50 iterations. In each iteration, we multiply T by 0.9 . After that, we run another loop of 500 iterations inside the first loop. In each iteration of the inner loop, we take 2 elements of S randomly and swap them. This gives us another sequence S' with cost C'. If $C' < C$ then we do $S = S'$. Otherwise, first we generate a random floating-point number r in [0,1]. If $e^{-\frac{C'-C}{k\times T}} > r$ then we do $S = S'$ else we do nothing and move onto the next iteration of the inner loop. When we exit the outer loop, we have obtained the optimal sequence S and the optimal cost C generated by the algorithm. We provide the pseudo code below:

```
S ← randomSequence(0, 1, . . . , M-1)
C ← cost(S)
for i ← 1 to COOLING_STEPS do
    T ← T × 0.9
    for j ← 1 to NUM_ITER do
        ind1 ← random(0,M-1)
        ind2 ← random(0,M-1)
        S' ← S
        swap(S'[ind1], S'[ind2])
        C' ← cost(S')
        r ← randomFloat(0,1)
        exponent ← e^(-(C'-C)/(k×T))
        if C' < C then
            S ← S'
            C ← C'
        else if exponent > r then
```

S $\leftarrow$ S'
C $\leftarrow$ C'

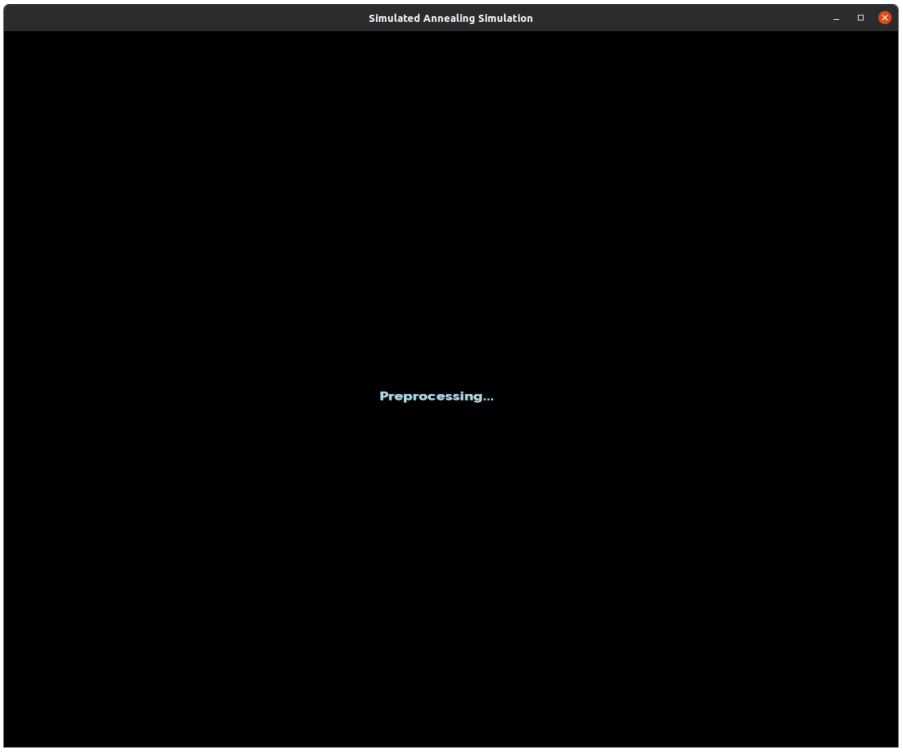
5 Simulation

We generate the maze with blocked cells encoded as 0, unblocked cells encoded as 1 and broken cells encoded as 2. Different textures are used to render different types of cells to visually show the difference. Dimensions of each texture is 20×20, so we scale each coordinate by 20 while rendering. We also use different sprites for the cells containing treasure chests. At all times, we display the current best score, current state score, temperature and iteration number on the screen. For each iteration, the simulation displays the path considered by the explorer to get the treasure in the following way. Let the path generated be $P_1, P_2, \ldots, P_n$. Then in the first frame for the iteration, the path from P_1 to P_2 is marked with crosses. In the next frame, the path from P_1 to P_3 through P_2 is marked with crosses. This continues until the entire path for the current iteration has been displayed, at which point the next iteration starts. This clearly shows the path and order of visiting considered by the explorer in the current iteration. If this state is accepted by the algorithm, the changes are reflected in the current state and otherwise they are discarded. To make the animation appear smooth and still convey the required information to the viewer, the simulation has been run at 20 FPS, using SDL_Delay. We have run 50 cooling steps of 500 iterations each. T is initialized with 1 and multiplied by 0.9 at each cooling step. Value of k is chosen to be 50. We have included options for the user to modify the number of cooling steps and iterations per step. We have included a functionality that pauses the simulation when a particular key has been pressed. This will allow the user to take his/her time viewing the current iteration.

6 Pictures and Video from the Simulation

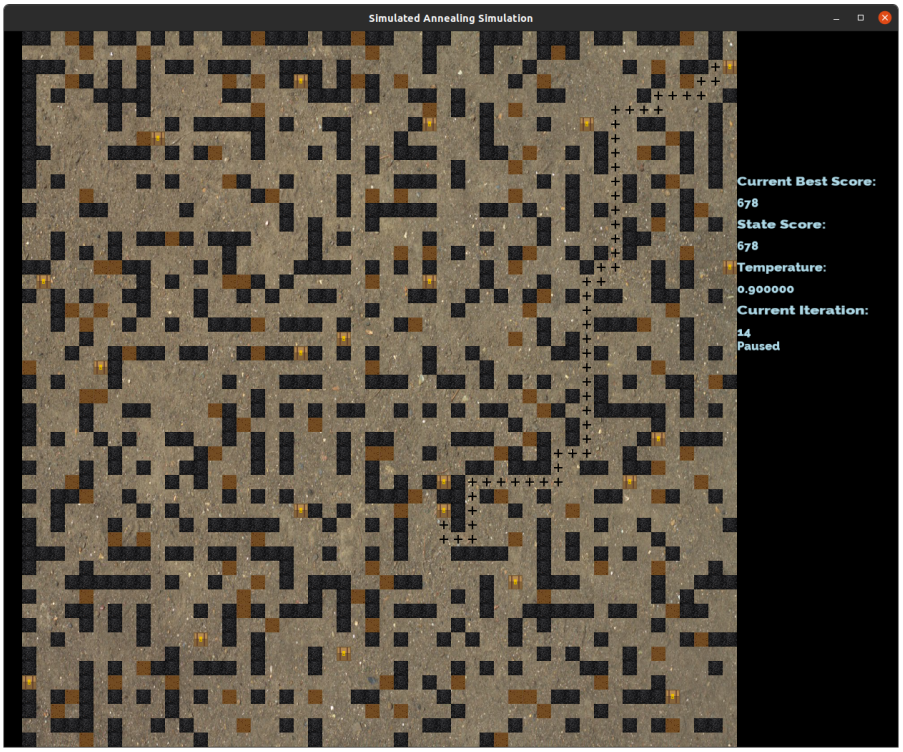
Link for a video of the simulation: Link (Click)

Preprocessing:

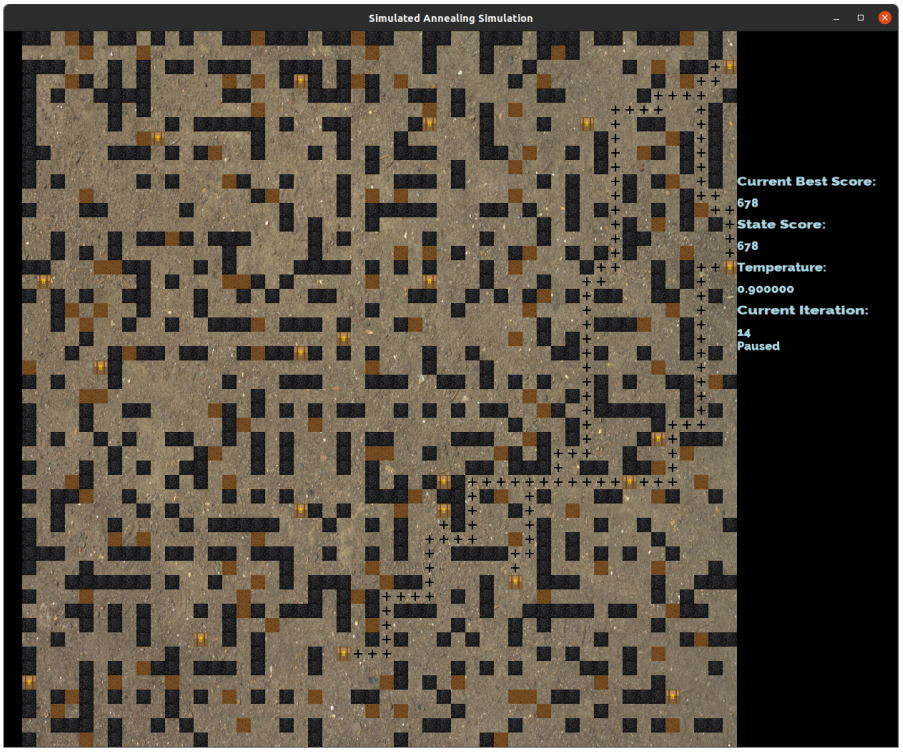


First Iteration:

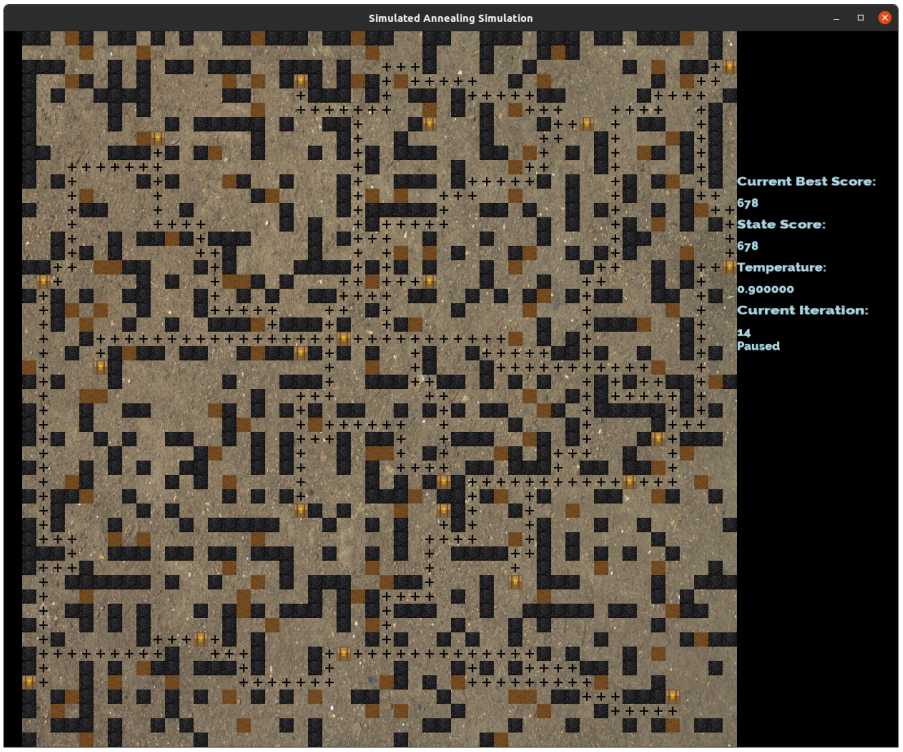
Start of an Iteration:



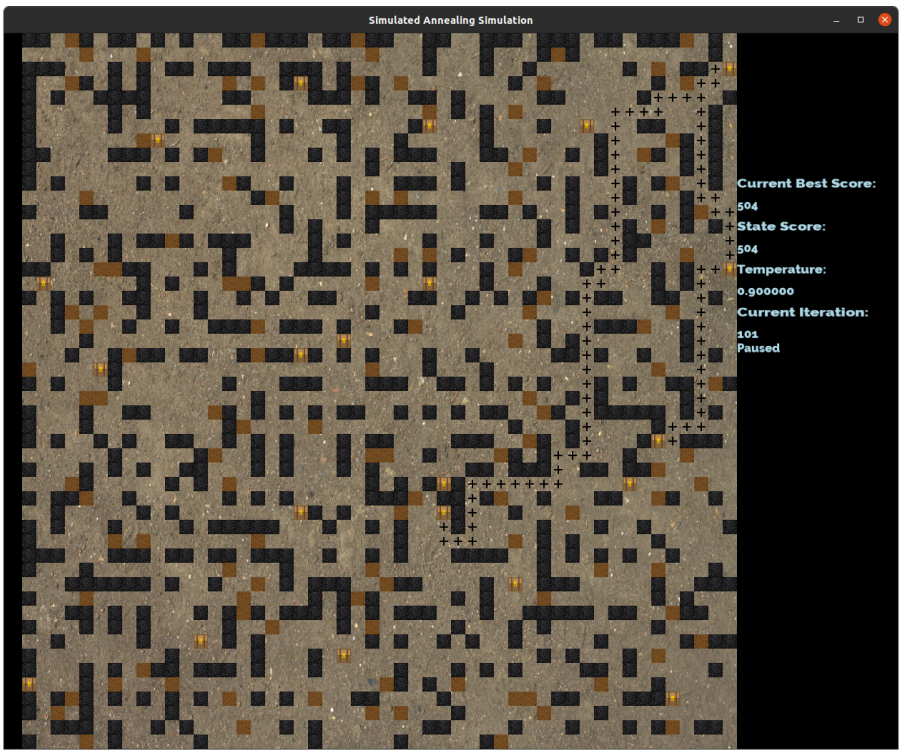
Middle of an Iteration:



End of an Iteration:



Decrease in Score:



7 Runtime Analysis and Data Structures

- Kruskal's Algorithm: We have a graph of $O(N^2)$ vertices and $O(N^2)$ possible edges(edges are added between vertices with Manhattan distance 2 and if they have same x or y coordinate). We use random_shuffle for randomizing the order of edges. To implement DSU(Disjoint Set Union) which is required, we use union by size and path compression techniques. The final algorithm runs in $O(N^2 \times logN)$.

- Dijkstra's Algorithm: We have implemented a Dijkstra's algorithm for each cell of the graph. The time complexity of a single run is $O(V \times logV + E) \equiv O(N^2 \times logN + N^2)$. Since we perform a run for each cell, the final time complexity is $O(N^4 \times logN + N^4)$. To make it a bit faster, we notice that we don't need to run this for blocked cells. The time complexity remains same, but it reduces the hidden constant factors. A set in the form of a red-black tree in the C++ implementation is used for this algorithm.

- Shortest Path: To find the shortest path from cell A to B, we backtrack from B, and change B to one of its neighbours, which is closest to A. When B becomes equal to A, we stop. The path length can be of the order of $O(N^2)$ in the worst case. So, the time complexity of this function is $O(N^2)$. But in practice, it is very less than this bound due to the randomness and sparseness of the maze. We store the values in a matrix in the form of a vector of vectors.

- Simulated Annealing: First we generate a random order to visit the treasures and find the cost of that path. This takes $O(M)$ time, where M is the number of treasures. The procedure has loops in a nested order. We assume the outer loop runs for N_1 iterations and the inner loop runs for N_2 iterations. In each iteration of the inner loop, we perform a constant number of operations. We generate 2 random indices, swap them in sequence S and find the new cost in $O(1)$ time. The exponential term that we use for probability can also be computed in $O(1)$. We have precomputed the shortest path between all pairs of treasures and from each treasure to our source. We just use them for the animation and they have nothing to do with the algorithm. Therefore, overall complexity is just $O(N_1 \times N_2)$ assuming $(N_1 \times N_2)$ is larger than M. We store the current state sequence in a vector.

8 A Deterministic Approach

The most straightforward approach is taking all the possible permutations in which we can take the treasures. But we would need to calculate the cost for 20! different sequences. This is not feasible.
There is another approach which uses bitmask dp. Assign an arbitrary ordering to the treasure vertices from 0 to M-1, where M is the number of treasures. Then, for each state of the solution, construct a bitmask with 1 in the i^{th} $(0 \leq i \leq M-1)$ position if the i^{th} treasure has been visited in the current path and 0 otherwise. This can be seen as the binary representation of a M bit integer. We also need to know the treasure that we have taken last among all the treasures taken currently. So, for any position, we can define a dp state dp[i][j], where i is an integer that represents the treasures taken and j is the number of the treasure taken last in the current state.
Our recursion then is:
dp[i|(1<<s)][s]=min(dp[i|(1<<s)][s], dp[i][j]+distance(s,j)) for all treasures s not taken in the bitmask of i. Here distance(s,j) is the shortest distance between treasure s and treasure j. This is done for all possible i from 0 to $2^M - 1$ in increasing order.
This is because we can choose any of the currently unchosen treasures to be the next treasure. This corresponds to setting the bit for that treasure and that treasure becomes the treasure that we took last.
The base case is dp[i][j]=0 if i=0 and j is the source, otherwise it is ∞

Our final answer is min(dp[2^M-1][0]+val(0), dp[2^M-1][1]+val(1), ..., dp[2^M-1][M-1]+val(M-1)) where val(i) is the shortest distance from treasure i to the source.
This is because the last selected treasure can be any treasure, and we have to choose all treasures for a valid path.
Note that this approach has complexity $O(2^M \times M^2)$. This is because there are 2^M bitmasks of size M and atmost M possible last treasures for each bitmask. Therefore, there are atmost $2^M \times M$ possible states in the dp. From each state we have atmost M transitions. This gives us a time complexity of $O(2^M \times M^2)$. For M=20, it should run in less than 10 seconds.

9 Conclusion

We see that the simulated annealing method converges quite fast. We ran some simulations after disabling the rendering in SDL and observed that the score would regularly become almost half of the initial score even with only 10 cooling steps of 100 steps each. It is also very fast since the individual iterations themselves have constant cost. The time complexity is completely dependent on the number of cooling steps and iterations per step, which are under our control. The correctness of Simulated Annealing comes from statistical physics and Boltzmann statistics in particular. In a way, we are trying to reach the optimal entropy for the system. To reduce the probability of transitions from a state with lower metric to a state with higher metric, we can reduce the value of k and T in our program. This will ensure less "exploration" by the algorithm, but it may get the algorithm stuck in local minima of the metric. Finally, we saw that there exists a deterministic bitmask dp approach to this problem. However, it has exponential complexity, which is expected since our problem reduces to TSP, which is provably NP-hard. We could run the dp algorithm in parallel with simulated annealing for small M and check the accuracy of annealing. However, the dp approach is no longer feasible for large M. The idea behind simulated annealing was taken from The Algorithm Design Manual by Steven Skiena.