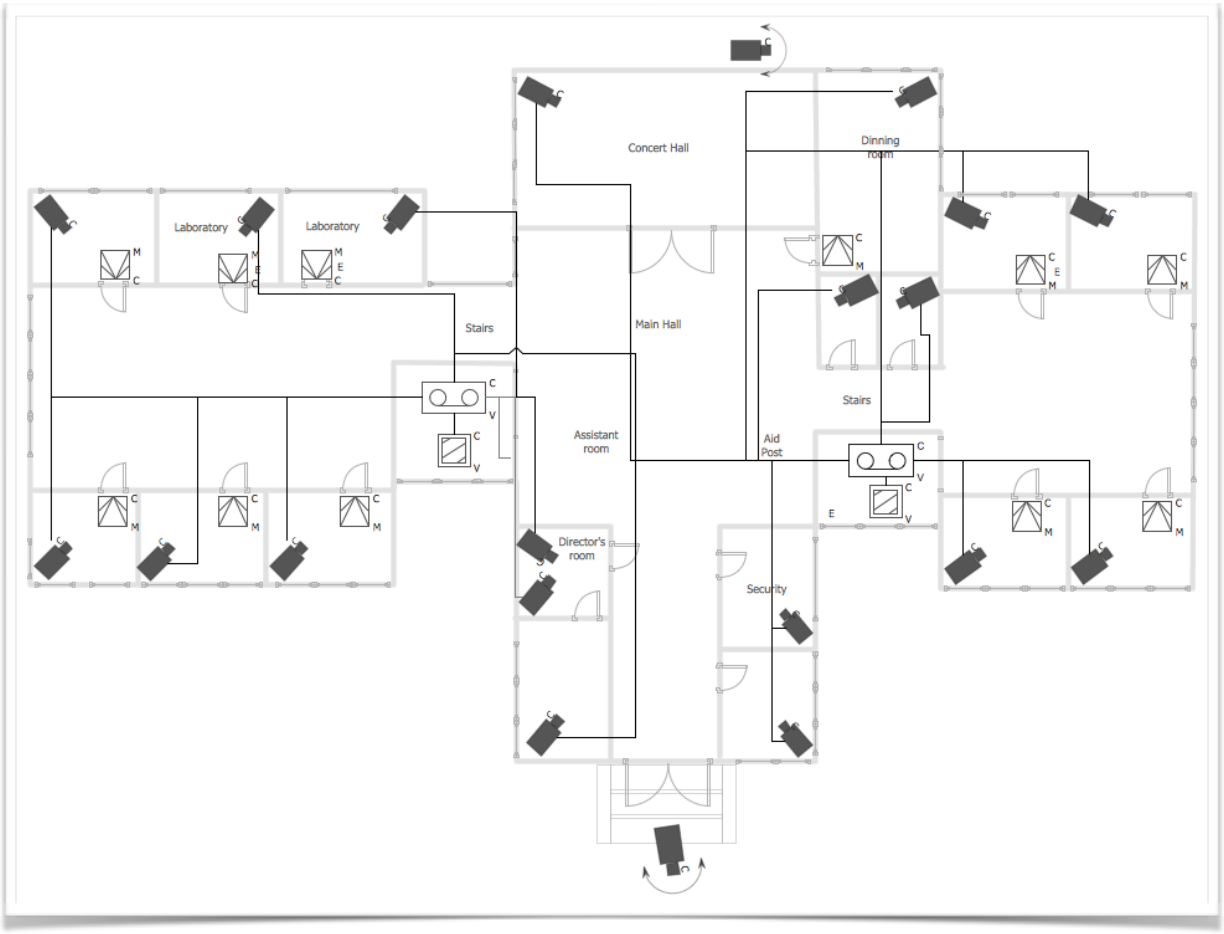
CSC361



SURVEILLANCE CAMERAS

Prepared for: Alanoud Alhakbani 436201421
Elaf Almahmoud 437200336

Introduction

Surveillance cameras are widely used for monitoring, however, installing a camera usually costs a lot. Therefore, in this project we aim to develop a program that finds the minimum number of cameras (and its location) needed to cover all spots in the room. Our program takes as input a text file that represents the map of a room as a grid. Using Best-first search algorithms (Greedy search & A*) and Local search algorithms (Hill climbing & Simulated annealing) to find the minimum number of cameras that covers all the cells.

Best-First Search

REPRESENTATION

The problem will be represented using N*N matrix, where N is the room size. The matrix is of type String, and it's filled with "-", "o" or "v" , where it means uncovered cell, obstacle and covered cell respectively.

STATES

Rooms with camera location if any.

INITIAL STATE

The algorithm will start with an empty Room.

ACTION (GENERATING SUCCESSORS)

Adding a camera to the parent per level in a valid position where no camera or obstacle exist.

GREEDY SEARCH

Evaluation Function

Heuristic

To calculate the heuristic, first, we need to find the **maximum number** of cells that could be covered using one camera only. This number depends on the room representation.

Then the heuristic is calculated by dividing the number of uncovered cells over the maximum number.

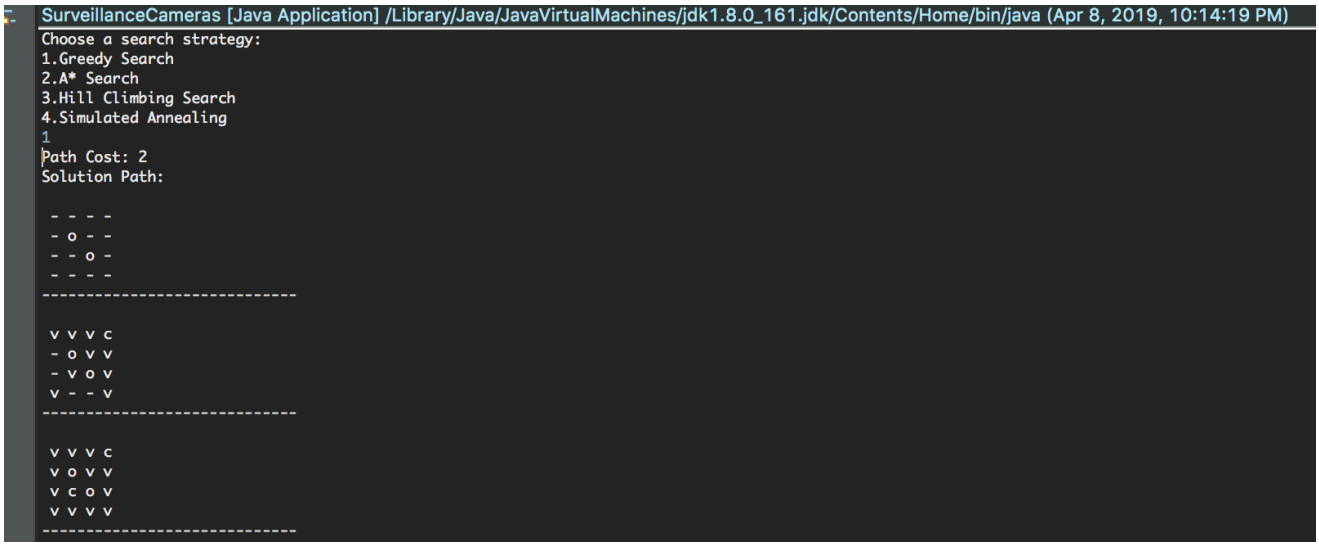
Design

Greedy Search
• greedySearch(): Node • greedyHeuristic(int[] positionOfCameras, int maximumNumber): double

Implementation

Greedy Search algorithm is done by implementing the greedySearch method. A frontier is implemented as a priority queue, so we expand states will lower heuristic value before others, and the explored set is represented as a Linked list to prevent re-expanding) . After that, the greedySearch method returns the goal state as a node.

Results



Since at each level the algorithm will add one camera, the cost represents the number of cameras we have.

A* SEARCH

Evaluation Function

Heuristic

To calculate the heuristic, first, we need to find the **maximum number** of cells that could be covered using one camera only. This number depends on the room representation. After that, the heuristic is calculated by taking the **ceiling** of the division of the number of uncovered cells over the maximum number. Which will give us an assumption of how many extra cameras we need to reach the goal state

Cost

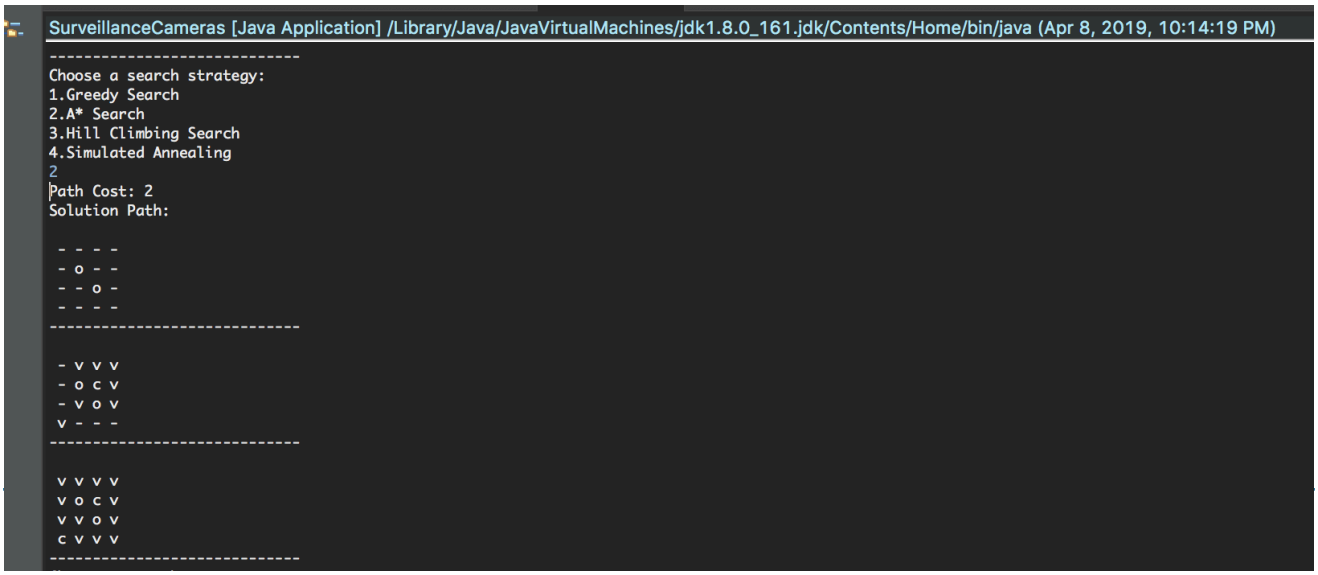
Number of cameras.

Design

A* Search
• A(): Node • AHeuristic(int[] positionOfCameras, int maximumNumber): int

Implementation

A* algorithm is done by implementing the A* method and using the Heuristic method to help it determine the obtained solution. A* is admissible since it never overestimates the actual number of cameras needed to cover the whole room.



Results

Local Search

REPRESENTATION

We are going to use the same representation described in best-first search section.

OBJECTIVE FUNCTION

Number of cameras and uncovered cells in the room. The goal is to minimize this number.

HILL CLIMBING

Initial Solution

We will generate the initial solution by randomly selecting a position for the first camera.

Termination Criterion

Predefined number of iterations (maxIteration), or if there is no improving neighbor.

Move

1- Deleting a camera

2- Adding a camera

Candidate neighbor selection

We will select the first improvement neighbor. By first improvement we mean the first neighbor we found by applying the chosen move with an objective function value less than the current solution.

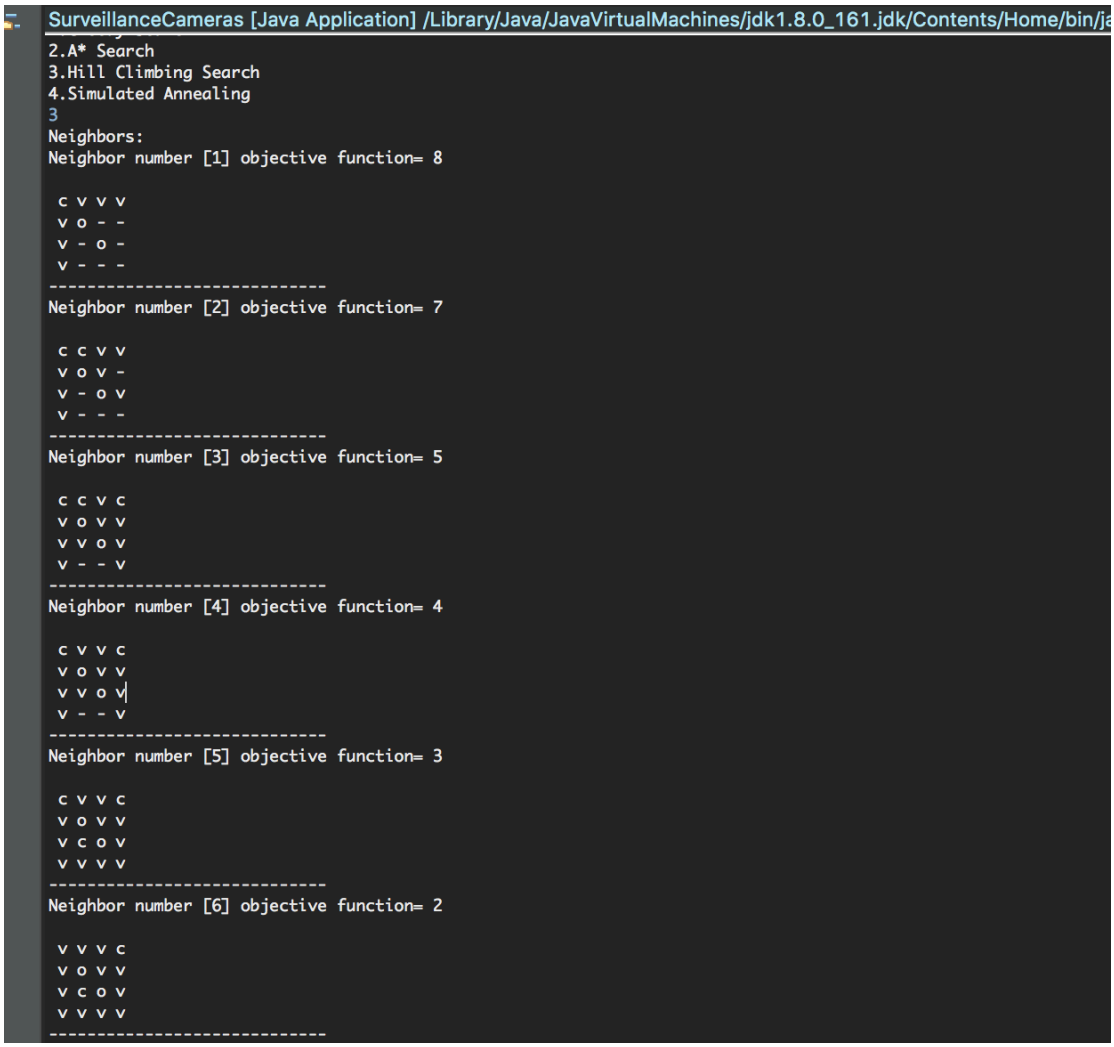
Design

Hill Climbing
• hillClimbing (int maxIteration): LinekedList<int[]> • getBestFisrtNeighbour(int[] cameraPosition): int[]

Implementation

Hill Climbing is done by implementing a method called hillClimbing that takes the number of maxIteration to iterate through the method and then return a list of neighbors that led to the obtained solution. A helping method getBestFisrtNeighbour that takes as an input a list of camera positions and return a list of the camera positions modified after applying the move.

Result



Staring by putting the fist camera in a random position, Hill Climbing improvs the solution quality till it reaches a solution with no improving neighbors. In this case, Hill Climbing was able to find the global optimum solution.

SIMULATED ANNEALING [1]

Initial Solution

Same as in Hill Climbing, we will generate the initial solution by randomly selecting a position for the first camera.

Termination Criterion

1-Predefined number of iterations (maxIteration)

2- The temperature reached zero

Parameters

1- Initial temperature: The temperature plays a crucial role in find an approximation of a global minimum. It should start at a high value and must end at T=0. By testing different temperatures values, we found that choosing the temperature to be the N*N*N , where N is the diminution of the room , gave the best result

2- Equilibrium state: to reach an equilibrium state at each temperature, a number of sufficient transitions (moves) must be applied. The number of iterations must be set according to the size of the problem instance and particularly proportional to the neighborhood size |N(s)|. Thus, we choose it to be N.

Move

Choose randomly either to delete a camera or add a camera to the current solution.

- If we want to delete a camera the number of cameras currently in the solution should be at least one.

-If we want to add a camera at certain location, it must check first if there's no camera or obstacle in that location.

Candidate neighbor selection

After generating a neighbor using one of the moves, the algorithm will accept it directly if it's an improving neighbor. However, if it's a non-improving neighbor it will be accepted with a probability.

Design

Simulated Annealing:
• SimulatedAnnealing(int maxIteration): LinekedList<int[]> • probabilityFunction(int delta e, double current temprature): boolean • randomChange(int[] cameraPosition): int[]

Implementation

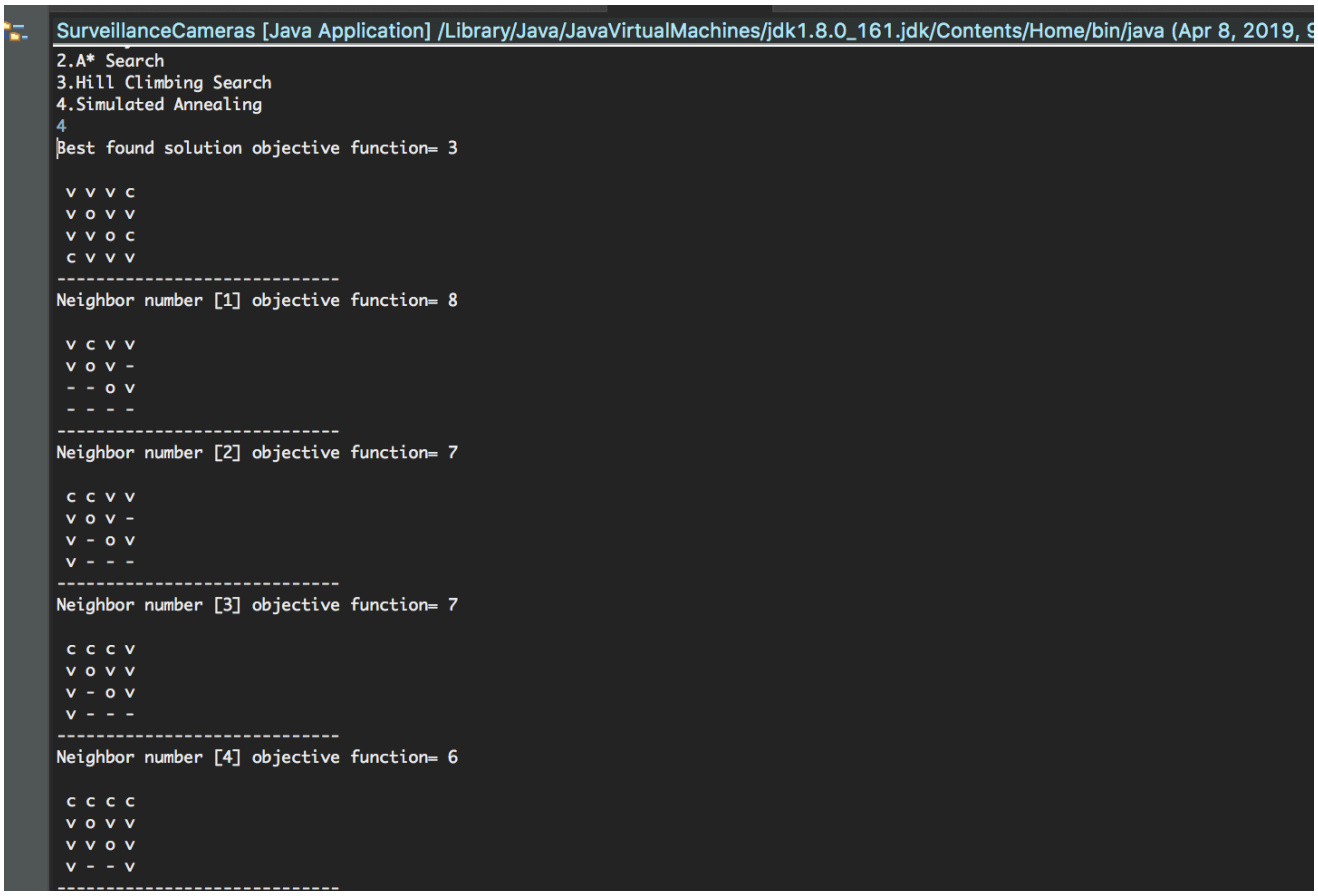
Simulated Annealing is done by implementing a method called SimulatedAnnealing that takes the number of maxIteration to iterate through the method and then return a list of neighbors that led to the obtained solution. A helping method probabilityFunction that takes as an input a delta e and the current temperature and return if we should accept the non improving solution or not. Another helping method is randomChange that takes a list of camera positions and return a list of the camera positions modified after applying the move.

We implemented SA using the algorithm showed below .

Algorithm 2.3 Template of simulated annealing algorithm.

Input: Cooling schedule.
$s = s_0$; /* Generation of the initial solution */
$T = T_{max}$; /* Starting temperature */
Repeat
  Repeat /* At a fixed temperature */
    Generate a random neighbor s' ;
    $\Delta E = f(s') - f(s)$;
    If $\Delta E \leq 0$ **Then** $s = s'$ /* Accept the neighbor solution */
    Else Accept s' with a probability $e^{\frac{-\Delta E}{T}}$;
  Until Equilibrium condition
  /* e.g. a given number of iterations executed at each temperature T */
  $T = g(T)$; /* Temperature update */
Until Stopping criteria satisfied /* e.g. $T < T_{min}$ */
Output: Best solution found.

Result



Since SA accepts non-improving neighbors, we kept the best neighbor we found during the search and consider it as SA result.

Conclusion

Overall, Our program displays the map of the room after applying the algorithms mentioned above. In addition for the Best First Search it displays the solution path and it's cost. For thee Local Search it displays the selected neighbor of the current and the value of the objective function at each iteration.