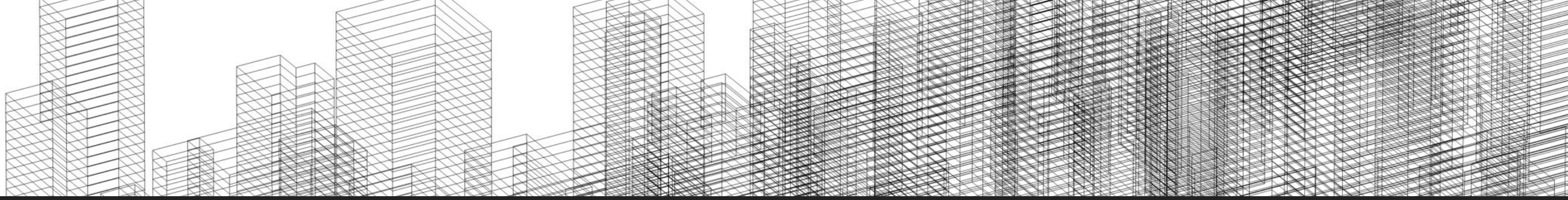


Router Placement

IART - Checkpoint 1

Ana Inês Oliveira de Barros- up201806593@fe.up.pt
João de Jesus Costa - up201806560@fe.up.pt
João Lucas SIlva Martins - up201806436@fe.up.pt

Problem Specification

Task: Given a building plan, decide where to put wireless routers and how to connect them to the fiber backbone to maximize coverage and minimize cost. More Info at: https://storage.googleapis.com/coding-competitions.appspot.com/HC/2017/hashcode2017_final_task.pdf

Building:
. **H** rows and **W** columns
. Coords **[r, c],** starting at 0
. **[0, 0]** is the upper left corner of the grid
. wall is **'#'**
. target is **'.'** - these cells need **wireless coverage**
. void is **'-'** - these cells **don't** need wireless coverage

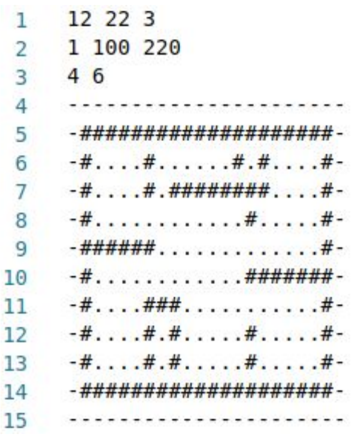
Routers:
. Each router covers at most **(2 * R + 1)^2** cells around itself, where **R** is the router's **range**.
. Signals are stopped by walls:
  . |a - x| <= R,
  . |b - y| <= R,
  . there is no wall [w, v] where **min(a, x) <= w <= max(a, x) && min(b, y) <= v <= max(b, y)**
. described as: there are no walls in the smallest enclosing rectangle of [a, b] and [x, y]

Backbone:
. Routers can only be placed in cells connected to the backbone.
. In the beginning, only 1 cell is connected to the backbone.
. Cells of any type can be connected to the backbone (one of its eight neighboring cells must already be connected to the backbone).

Budget:
. Placing a router costs **Pr**
. Connecting a cell to the backbone costs **Pb**
. The maximum budget is **B**

Input:
.1st line - **H W R**
.2nd line - **Pb Pr B**
.3rd Line - **br bc**
.Rest of lines - Cells

(1 <= H <= 1000) - number of rows on the grid
(1 <= W <= 1000) - number of columns on the grid
(1 <= Pb <= 5) - price of connecting one cell to the backbone
(5 <= Pr <= 100) - price of one wireless router
(1 <= B <= 10^9) - maximum budget
(0 <= br < H) - row of the initial cell connected to the backbone
(0 <= bc < W) - column of the initial cell connected to the backbone



```
12 22 3
1 100 220
4 6
----------------------
-####################-
-#....#......#.#....#-
-#....#.########....#-
-#............#.....#-
-######.............#-
-#...........########-
-#....###...........#-
-#....#.#.....#.....#-
-#....#.#.....#.....#-
-####################-
----------------------
```

Related work

Problem solved using simple hill climbing techniques:

https://github.com/tasosxak/Router-Placement

Problem solved using a genetic/memetic algorithm:

https://github.com/admirkadriu/router_placment_ga

Formulation of the problem as an optimization problem

Rigid Constraints:

- We can't go over the max budget.
- Routers have to be placed next to or on top of a backbone.
- Routers can't be placed on top of walls, other routers, or void cells.

Solution representation: Solutions are represented by a list of router coordinates (fixed length **chromosome**), *routers*, and an integer, *cutoff*. All these coordinates are distinct. The routers from indice 0 to *cutoff* of the list, are the ones that are counted as part of the solution.

Neighborhood/Mutation: The neighborhood is obtained by applying the following operator to a solution (**mutation**) until there are no more neighbors available.

- **Operator:** Move a router to a cell in one of the cardinal or intercardinal directions (N, NE, E, SE, S, SW, W, NW). If a wall blocks the router in a selected direction, the router attempts to move to the first empty cell in that direction (if any). Routers can't be moved to the top of other routers.

Crossover Function: the crossover function selects routers from each parent by defining a rectangular border with random size and center. The child has the routers from parent 1 that are inside the border and from parent 2 that are outside the border. The order of the two parents (1 and 2) is selected randomly with 50% chance for each. In case there is a need to pad the child's **chromosome**, we add routers from both parents (interleaved) until the **chromosome** has the correct length.

Evaluation function: Routers (coordinates) in the **chromosome** are evaluated to determine if they are part of the solution. Routers from the start to the end of the list are included while there is still budget to add them and adding them increases the value of the solution. When any of these two conditions is violated, the *cutoff* for the solution is set. The value of a solution is given by: **C * 1000 + (B - Cost)**, where **C** is the number of cells covered by at least one router and **B** is the budget.

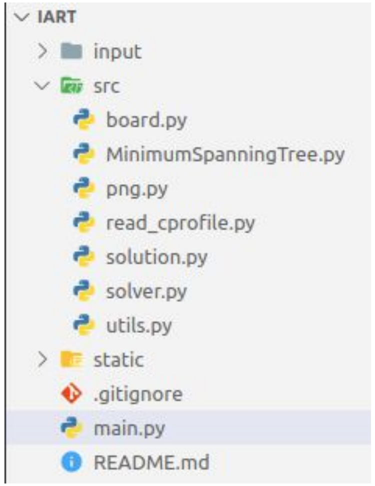
Current work status

Programming Language: Python

Development Environment: NeoVim and VSCode

File structure:

- **input/** - contains the input files.
- **src/** - contains the source code files.
- **static/** - contains images used in **README.md** file
- **main.py** - is the menu interface.
- **README.md** - is the problem and project description



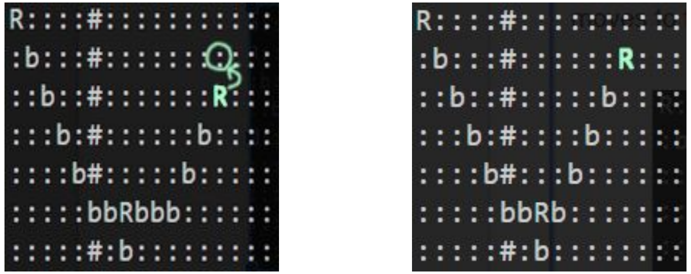
```
IART
  input
  src
    board.py
    MinimumSpanningTree.py
    png.py
    read_cprofile.py
    solution.py
    solver.py
    utils.py
  static
  .gitignore
  main.py
  README.md
```

Data Structures:

- **Classes**
  - **Board** - Holds a problem representation and provides convenient methods to access it.
  - **Solution** - Represents a solution and provides methods to calculate its value and cost, generate mutations, and crossover with another solution.
  - **Graph** - Tree representation of the router's placement which is used to calculate the best backbone path.
  - **Solver** - Finds a solution to a problem using various algorithms.
- **Sets**
  - **CoveredCells** - Keep track of the cells covered by a router/solution
  - **Walls** - Store the coordinates of all walls of a given board
  - **Backbones** - Store the coordinates of all backbones placed for a solution
- **List**
  - **Routers** - Contains routers coordinates of a chromosome.
  - **AvailablePositions** - is a randomly shuffled list of all positions where a router can be placed.

The approach

Operator: Move a router a cell in one of the cardinal or intercardinal directions (N, NE, E, SE, S, SW, W, NW). If a wall blocks the router in a selected direction, the router attempts moves to the first empty cell in that direction (if any). Routers can't be moved to the top of other routers.



```
R::::#:::::::::::
:b:::#:::::::::::
::b::#:::::::R:::
:::b:#::::::b::::
::::b#:::::b:::::
:::::bbRbbb::::::
:::::#:b:::::::::
```

```
R::::#::::::::::
:b:::#::::::R:::
::b::#:::::b::::
:::b:#::::b:::::
::::b#:::b::::::
:::::bbRb:::::::
:::::#:b::::::::
```

Mutation of a solution (left to right)

Cost of a solution: given by **(R * Pr + Bb * Pb)**, where:

- **R** is the number of placed routers,
- **Pr** is the price of a router,
- **Bb** is the numbers of placed backbones (besides the initial one),
- **Pb** is the price of each backbone.

Evaluation Function:
Routers in the chromosome (routers list/pseudo-solution) are evaluated to determine if they are part of the solution. Routers from the start to the end of the list are included while there is still budget and adding them increases the value of the solution. When any of these two conditions is violated, the "cut off" (threshold of the routers that are considered part of the solution) for the solution is set.
The **value** of a solution is: **C * 1000 + (B - Cost)**, where **C** is the numbered of cells covered by at least one router.

Heuristics:

- **Backbone Placement -** Connecting routers to the backbone is an instance of the Steiner Tree Problem (NP-Hard). To agilize this calculation, we estimated the paths by solving the problem as a minimum spanning tree problem.

- **Choosing a solution from a chromosome -** Choosing the best combinations of routers that don't surpass the maximum budget from a pseudo-solution has time complexity of O(2 ^ N). As explained in the **Evaluation Solution** section, an heuristic using a "cut off" was considered.

Algorithms implemented

- **Hill climbing: random walk**
  - walks the neighborhood of the current node randomly and accepts the first neighbor that is better.
  - stops when the current Solution doesn't have any neighbors that are better (higher value) than itself.

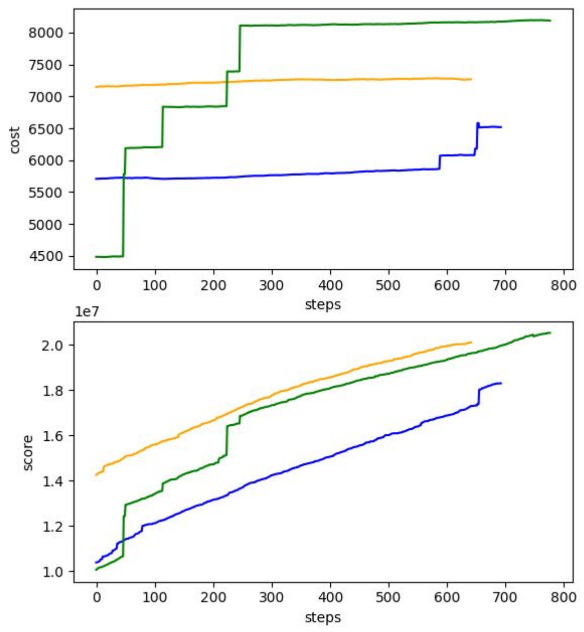
- **Simulated annealing**
  - configurable cooling schedule.
  - configurable minimum temperature (stop condition).
  - returns the best Solution found during the run, because solutions can get worse on this method.
  - dynamic initial temperature: standard deviation of the scores of 400 randomized initial solutions for the given problem.
  - dynamic iterations per temperature: based on the length of the chromosomes.
  - configurable random restart chance: at the end of the iteration for each temperature, there is a chance to go back to the highest rated solution so far (and the temperature at which it was found) if the current solution is worse.

- **Hill climbing: steepest ascent**
  - finds the best neighbor (highest value) of the current node.
  - stops when the current Solution doesn't have any neighbors that are better (higher value) than itself.

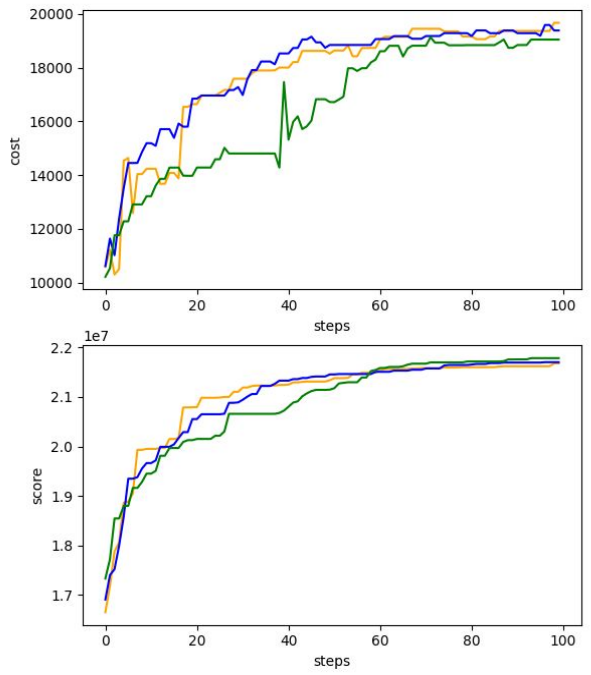
- **Genetic algorithm**
  - the fitness function is the same as the evaluation function described in the previous slide.
  - the mutations are the same as the neighbor selection/generation function (with a configurable chance of happening).
  - the best solution for each population is directly passed to the next (elitism).
  - the parents are selected randomly from the previous population with a weight equal to their value.

Experimental results*

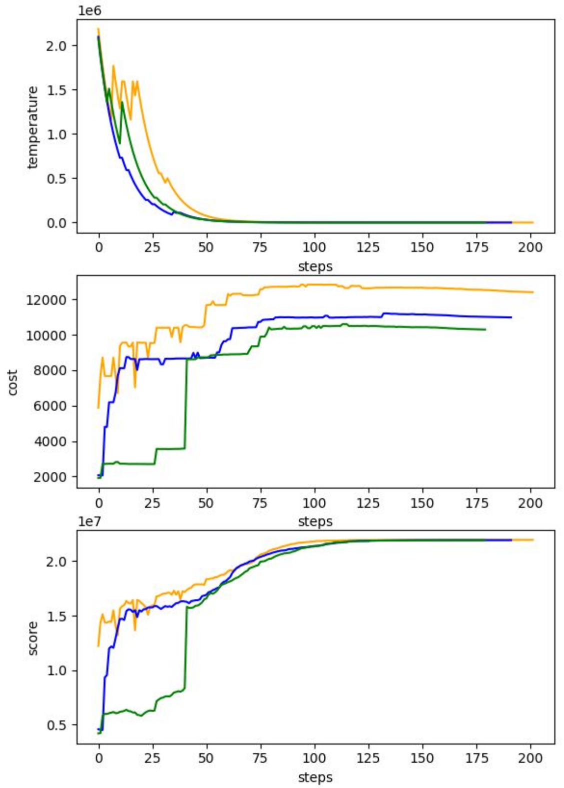
Stochastic Hill-Climbing



Genetic Algorithm
(pop. size: 100 - iterations: 100 - mutate prob.: 10%)



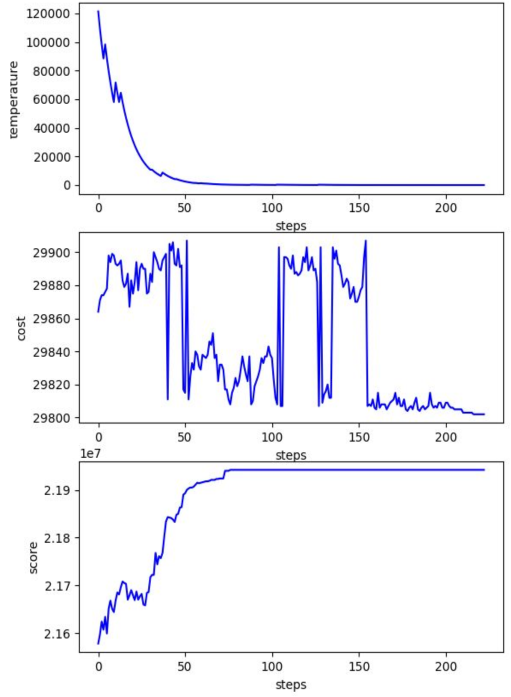
Simulated Annealing (cooling schedule: **t * 0.9**)
(min. temp.: 0.1 - rdm. restart chance: 10%)



** Results obtained for the charleston road problem*
Each graph contains 3 runs of the algorithm

Experimental results (pt.2)

Simulated Annealing (cooling schedule: **t * 0.9**)
(min. temp.: 0.1 - rdm. restart chance: 5%)

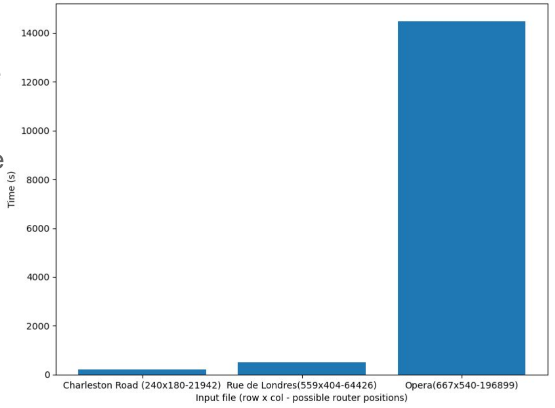


The graph on the left shows us that, by not cutting off the chromosome when adding the next router would not increase the value of the solution, we don't get better scores in the end and the algorithm takes longer to finish.

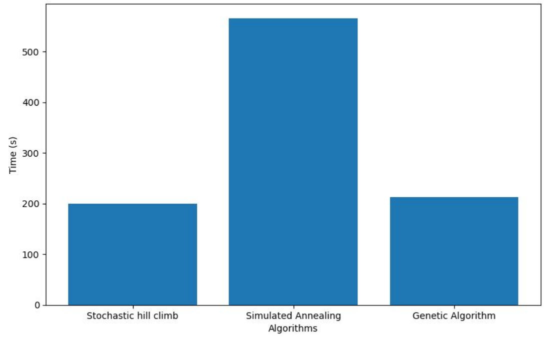
We also get solutions that are a lot more costly (less optimized) this way.

On the right, we can see the time scaling for most of the given problems. 'Let's go higher' is excluded, because the run time is too long (more than 4 hours).

Time scales a greatly with the number of possible positions for routers.



On the right, we can see a comparison of the run times of the different algorithms. The simulated annealing is the one that yields better results, but it also takes longer to end.



** Results obtained for the charleston road problem*

Conclusions

With the development of this project, we learned how to implement a system to solve an optimization problem (using different algorithms). The results varied for each algorithm:

- The hill-climbing algorithm gets slower the closer it is to a (local or global) maximum.
- The steepest ascent version ended up being substantially slower than the stochastic version without achieving better solutions.
- The simulated annealing algorithm proved to be fast at achieving good results. However, it also slows down when close to the global maximum. It is extremely difficult to reach the global maximum so, although this algorithm tends to get close, it still doesn’t reach it. The method used to get the initial temperature yield very good results, but we weren't able to come up with a good formula for the number of iterations for each temperature. We believe an adaptive version of the algorithm would help in this regard.
- The genetic algorithm’s crossover function used is complex but it achieves good results (better than single-point and k-point crossovers). However, it exhausts the genetic pool quickly. We believe that by using a memetic algorithm, we would achieve even better results.

We also concluded that, the evaluation function has a great impact in the results obtained. When we change the way the “cut off” of each solution is set, to not analyze the worth of adding the next router to the solution, we get a much slower analisis, and the results don’t improve much, even in the slightest. In addition, it makes the approach more “brute-force” like.

References

- https://github.com/tasosxak/Router-Placement
- https://github.com/admirkadriu/router_placment_ga
- https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
- https://en.wikipedia.org/wiki/Steiner_tree_problem
- http://www.cs.ucr.edu/~michalis/COURSES/240-08/steiner.html
- https://machinelearningmastery.com/simulated-annealing-from-scratch-in-python/
- https://en.wikipedia.org/wiki/Chebyshev_distance
- https://en.wikipedia.org/wiki/Simulated_annealing
- https://github.com/challengingLuck/youtube/blob/master/sudoku/sudoku.py
- **Genetic Algorithm Applied to the Graph Coloring Problem**, by *Musa M. Hindi and Roman V. Yampolskiy*