

Router Placement Optimization Problem

IART – Checkpoint 1, Group 44

Afonso Caiado, up201806789

Diogo Nunes, up201808546

João Pinto, up201806667

Router Placement

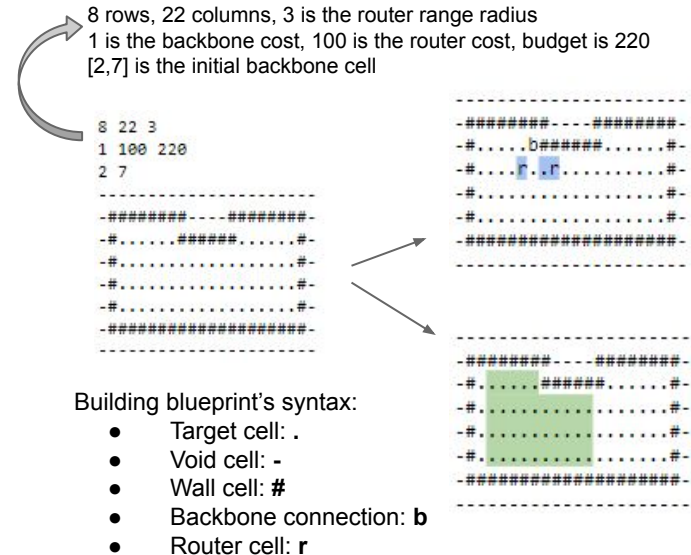
Objective: Given a building plan, decide where to put wireless routers and how to connect them to the fiber backbone to maximize coverage and minimize cost

The program receives a text file as input, that contains the blueprint of the building and the area that should be wirelessly connected to the internet. In the blueprint there is a socket that represents the fiber backbone, which provides the routers with access to the internet.

The input file specifies the building's blueprint, the range of a router, the cost of a tile of a backbone connection, the cost of a router, the budget and the cell where the fiber backbone is connected.

Note that the wireless connections cannot pass through walls. Void cells ("-") don't need wireless internet coverage. A router covers a square of side $2R+1$, being R its radius, centered in the router's position.

The cost of a router's connection to the backbone fiber is calculated by $N \cdot P_b$, being N the number of cells of the connection to the backbone, and P_b the cost of a connection that is provided by the input file. The total cost of a solution is the sum of all the routers' costs and the costs of all cells connected to the backbone. The total cost cannot exceed the budget.



Problem formulation (1)

Solution representation (in Python):

`[(xRouter1, yRouter1), (xRouter2, yRouter2), ..., (xRouterN, yRouterN)]`

Note: Identifiers starting with x and y are integers

Our representation consists on a **list of N tuples**, where N is the number of maximum routers possible to place in the solution, taking into account the budget. Each tuple represents a router coordinate. In case that we don't need the maximum routers possible to find a better solution, then the not needed routers will be replaced by (-1, -1).

Mutation Function: Chooses a number of routers to mutate and replace them with random neighbours.

```
def mutate(sol) :  
    routersToMutate = max(1, maxRouters // 10)  
    for i in range(routersToMutate):  
        if r == 0: break  
        sol = randomNeighbour(sol)  
        r -= 1  
    return sol
```

Crossover Function:

Our chromosomes will be the router's coordinates. In this function, we randomize a number between the minimum of the actual number of routers of the 2 solutions to cross, and then we create a child with each part (separated on the index we randomized) of each solution.

```
def crossover(sol1, sol2) :    # sol1 and sol2 are lists containing routers' coordinates  
    child = []  
    rand = random(1, max(1, minRouters - 1))  
    for i in range(len(sol1)) # or sol2, same size  
        if i < rand: child.append(sol1[i])  
        else: child.append(sol2[i])
```

Problem formulation (2)

Strong constraints: these constraints can never be broken, as they automatically invalidate a solution.

- Two routers cannot be placed in the same cell
- A router or a backbone cell cannot be placed in a **wall** cell
- Every router must be connected to the **backbone**
- The cost cannot exceed the available budget: $N \cdot P_b + M \cdot P_r \leq B$, being:
 - N: number of backbone cells
 - M: number of routers
 - P_b : price of connecting one cell to the backbone
 - P_r : price of one wireless router

Weak constraints: these constraints, if broken, will not make the solution invalid, they will probably just make it a weaker solution performance wise.

- A router's provided wireless connection cannot go through **walls**
- A **void** cell does not need to have wireless connection

Work Progress: Project File Structure

Programming language: We are using Python as a programming language.

Development environment: We are using PyCharm or Spyder as our development environment

File Structure:

- **docs** – Relevant document files related to the project, such as the problem statement, the checkpoint assignment report, and the final report
- **inputs** – Text input files containing different buildings information
 - **example.in**
- **out** – Output files produced by our algorithms
 - **example** – Folder containing the output files for the “example” building
 - **hill_climbing_regular.png** – Image of the solution produced by the regular hill climbing algorithm
 - **hill_climbing_regular.txt** – Text file of the solution produced by the regular hill climbing algorithm
- **src** – Source code for our project
 - **aStar.py** – A* Algorithm implementation with necessary methods and classes regarding the algorithm
 - **blueprint.py** – Class containing all methods regarding the building’s blueprint (verification of valid positions, cells covered if a router was to be put in a certain position, etc.)
 - **geneticAlgorithm.py** – Genetic Algorithm implementation with necessary methods regarding the algorithm
 - **kruskal.py** – Kruskal algorithm implementation for minimum spanning tree resolution
 - **main.py** – Main file to be executed when testing the program
 - **simulatedAnnealing.py** – Simulated Annealing implementation with necessary methods regarding the algorithm
 - **tabuSearch.py** – Tabu Search implementation with necessary methods regarding the algorithm
 - **utils.py** – Methods that are used by different algorithms (generate the first solution, calculate the value of the solution, etc.)

The approach

Meta-heuristics: Hill Climbing (regular and steepest ascent), Simulated Annealing, Tabu Search, Genetic Algorithm

Algorithms worth mentioning: Kruskal MST algorithm, A* algorithm

Neighbour function:

We implemented a function that, given any solution, was capable of generating a neighbour to that solution.

- ***randomNeighbour (blueprint,solution)*** - this function returns a random neighbour to the given solution, by moving a router either up, down, left, right or even diagonally.
- ***neighbour(blueprint,solution,routerToChange,coordToChange,upOrDown,numRouters)*** - this function returns a neighbour according to the information used when calling it. *routerToChange* is the number of the router in the solution, *coordToChange* is x or y (0 or 1) and *upOrDown* is 0 or 1, based on if you want said coord to be higher or lower.

Evaluation function (value):

t = number of cells covered by wireless

N = number of connected to the backbone

M = number of placed routers

remainingBudget = budget - (N * backboneCost + M * routerCost)

if remainingBudget < 0:

return None

return 1000 * t + remainingBudget

- 1000 points for each **target** cell covered with Internet access;
- 1 point for each unit of **remaining budget**

Implemented algorithms (1)

Hill Climbing: The Hill Climbing algorithm calculates a local maximum of a problem, given an initial solution. In each iteration, it computes the neighbours of the current solution, and replaces the current solution with a neighbour, if the neighbour is more valuable than the current solution.

Because of its nature, the quality of the output solution relies on the quality of the initial solution. Given a solution, there is a high probability that the HC algorithm cannot reach the global maximum.

The Hill Climbing Steepest Ascent has a better but more expensive approach. It consists in generating every possible neighbour and updating the current solution with the most valuable neighbour. To cheapen the computational load of this algorithm, we used a prediction of the value of each solution, instead of an exact value.

Simulated Annealing: Given a solution, the Simulated Annealing algorithm tries to obtain the global maximum of a problem. It works in a very similar way to the Hill Climbing algorithm. However, there are some differences.

In the SA algorithm, worse solutions can also be accepted, with a certain amount of probability. This probability depends on the temperature of the system, which starts at a high value and decreases with as the algorithm is processed. There are also various configurations to be made: the neighbourhood (8 neighbours per router), the number of iterations per temperature (10), the temperature's cooling schedule (exponential), starting temperature (10000) and final temperature (10).

Implemented algorithms (2)

Genetic Algorithm: Given 10 initial solutions, they reproduce each over (crossover) for 20 generations, making a child that have a 10% chance of being mutated. For each crossover, the parents are chosen from the best half of the population.

This algorithm depends on the initial population generation, because it will influence all the childs generated. For that reason, it's hard to get a solution where there are no unnecessary routers to get the perfect one. It's a matter of making some tries to find the best one.

Tabu Search: The Tabu Search algorithms main goal is to find the best admissible solution in the neighborhood of the current solution in each iteration, considering recent solutions as “Tabu” to prevent cycling.

Regarding our specific problem and the way we implemented it, it was not the most appropriate algorithm to implement for larger scale buildings, as it has to calculate the value of a certain solution way to many times, taking too much time. The “moves” our algorithm was to perform, were difficult to represent in the tabu list, as the same move (ex: the first router moves up) does not mean that we are visiting an already visited solution. However, it worked extremely well for the smaller building plants.

After generating the initial solution, all possible moves values are calculated for the current solution (move a certain router up, down, left, right or delete it). The move with the highest value is established, and after verified if said move is in the tabu list, we either make that move or not (by comparing it to the current solution's value). Repeat the process until you meet the termination criteria (try for 50 consecutive times to make a better move, without success)

Implemented Algorithms (3)

Kruskal Algorithm: The Kruskal algorithm calculates the minimum spanning tree (MST) of a graph. It is used to determine the cheapest way of connecting all routers to the fiber backbone.

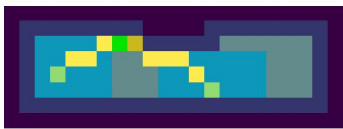
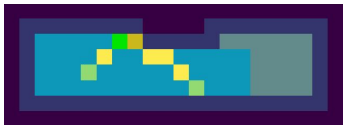


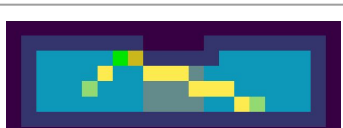
A* Algorithm: The A* Algorithm computes the shortest path between two points of a graph. It is combined with the Kruskal algorithm to translate an MST into a list of grid cells.

Experimental results

Purple: Void cell
Dark blue: Wall cell
Soft blue: Covered cell
“Blue” (almost grey): Uncovered cell

Green: Backbone cell
Soft green: Router cell
Yellow: Path cell
Dark yellow: Path cell + Wall Cell

Example input file (slide 2):

| | | |
|---------------------------------|--|--|
| Simulated Annealing | Value: 45 007 Solution: [(5, 14), (4, 3)] Time: 0.353s |  |
| Hill Climbing (Regular) | Value: 48 011 Solution: [(5, 12), (4, 5)] Time: 0.008s |  |
| Hill Climbing (Steepest Ascent) | Value: 54 007 Solution: [(5, 16), (4, 5)] Time: 0.027s |  |
| Genetic Algorithm | Value: 48 009 Solution: [(3, 13), (5, 5)] Time: 0.011s |  |
| Tabu Search | Value: 54 007 Solution: [(5, 16), (4, 5)] Time: 0.033s |  |

Experimental results

Purple: Void cell

Dark blue: Wall cell

Soft blue: Covered cell

“Blue” (almost grey): Uncovered cell

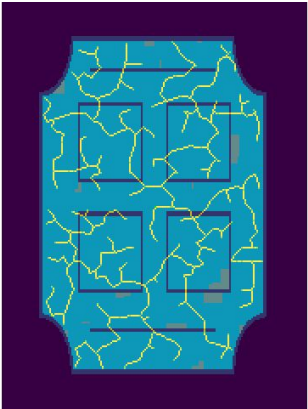
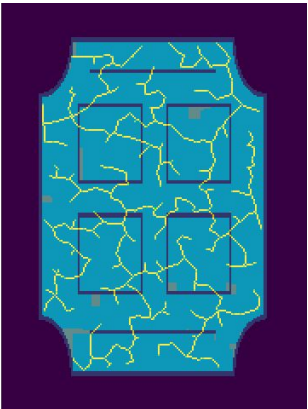
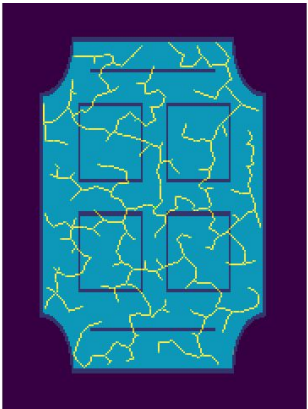
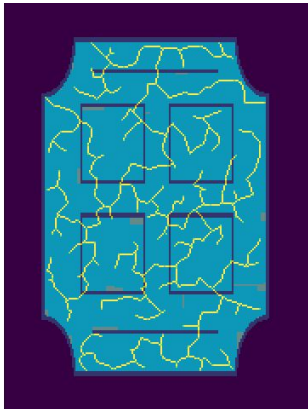
Green: Backbone cell

Soft green: Router cell

Yellow: Path cell

Dark yellow: Path cell + Wall Cell

charleston_road.in:

| Simulated Annealing | Hill Climbing (Regular) | Hill Climbing (Steepest Ascent) | Genetic Algorithm | Tabu Search |
|---|--|---|--|--|
| Score: 21 529 056 Solution: 280 routers Time: 653.4s | Score: 21 614 074 Solution: 280 routers Time: 190.3s | Score: 21 939 038 Solution: 280 routers Time: 705.3s | Score: 21 701 027 Solution: 280 routers Time: 463.6s | Being the only algorithm that takes into account the possibility of removing a router and having to analyse and compare the value of all possible moves for a given solution, this algorithm takes way too much time to complete. We do estimate, though, that it would have the best score of them all. |
|  |  |  |  | |

Related Work & References

<https://core.ac.uk/download/pdf/159237783.pdf>

https://www.cse.ust.hk/~gchan/papers/ICC14_PRACA.pdf

<https://stackoverflow.com/questions/50452893/finding-the-optimal-location-for-router-placement>

<https://github.com/sbrodehl/HashCode>

<https://onlinelibrary.wiley.com/doi/full/10.1002/itl2.35>

[https://en.wikipedia.org/wiki/Crossover_\(genetic_algorithm\)](https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm))

https://storage.googleapis.com/coding-competitions.appspot.com/HC/2017/hashcode2017_final_task.pdf

Conclusions

Project conclusions:

Regarding the implemented algorithms, we conclude that the overall most efficient algorithm was the Hill Climbing Steepest Ascent (HCSA). Although the Tabu Search algorithm was equally successful for the smaller building plants, as it is the only algorithm that considers the removal of a router as a neighbour and possible move, for the larger problems, it takes too much time. On the other hand, the HCSA performs very well for those same problems.

This project was undoubtedly one of the most interesting projects we've done to date. Despite all the encountered difficulties while programming and the amount of work we had to put in to be able to complete it, we have all agreed that the problem in itself was very intriguing and always kept us excited while implementing it.

Considering all the different projects and work we have throughout the semester, we believe that our final product was a success.

Group conclusions:

The work was done both by giving different tasks to each member of the group, but also by utilizing pair programming. We deem that all group members contributed equally and all gave their input and effort into the project.