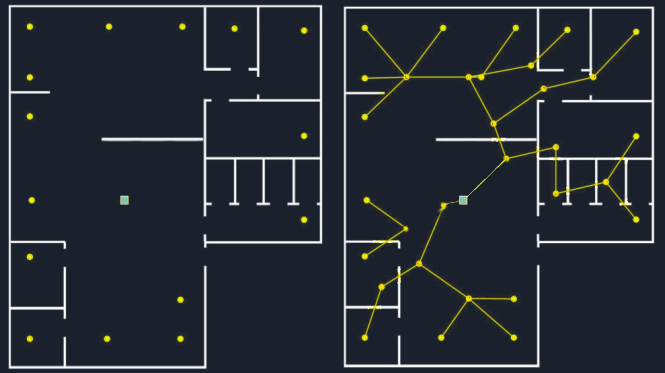
Router Placement Optimization



Given a building plan, a decision needs to be made on:

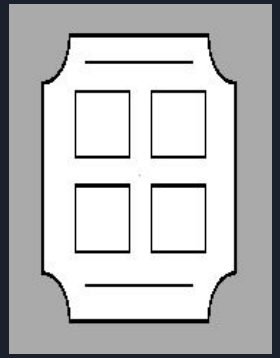
“ Where to put wireless routers and how to connect them to the fiber backbone to maximize coverage and minimize cost. ”

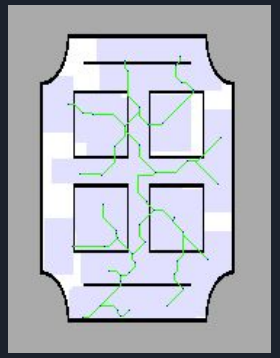
Eduardo Brito up201806271@fe.up.pt | Paulo Ribeiro up201806505@fe.up.pt | Rita Silva up201806527@fe.up.pt

Formulation as an Optimization Problem

- Solution Representation

Given a building blueprint, represented as a character matrix, the solution will consist of the same matrix of the building, with the routers and the backbone on their respective positions.





The cells used to link the backbone and the router will be highlighted, along with the cells inside the router range.

- Constraints

➔ To start, there is only one cell connected to the backbone, which can be of any type - target(.), void(-) or wall(#).

➔ A router can't be placed inside a wall and it has to be connected to the fiber backbone - a cable that delivers Internet to the router itself.

➔ Each router covers a square area of at most $(2 \times R + 1)^2$ cells around it, unless the signal is stopped by a wall cell:

- $|a - rx| \leq R$, and $|b - ry| \leq R$
- there is no wall cell inside the smallest enclosing rectangle of [a, b] and [rx, ry]. That is, there is no wall cells [w, v] where both $\min(a, rx) \leq w \leq \max(a, rx)$ and $\min(b, ry) \leq v \leq \max(b, ry)$.

➔ Placing a single router costs Pr and connecting a single cell to the backbone costs Pb. To ensure the budget constraint, the cost to place the routers and its connections should not be higher than the budget:

$$Nb \times Pb + Nr \times Pr \leq B$$

Formulation as an Optimization Problem

- Neighborhood/Mutation

  One possible new-solution generator may be to start by randomly placing a number N of routers.

  The solution neighborhood is represented by the neighboring cells of each one of the routers' positions.

- Crossover Functions

  The crossover between two distinct solutions will consist on randomly selecting and combining the routers from each solution.

- Evaluation Function

Being c the total number of *target cells* covered, the value of a solution can be computed as follows:

$$value = c + (B - (Nb \times Pb + Nr \times Pr))$$

where:

- ➔ B is the Budget
- ➔ Nb is the number of Backbone cells used
- ➔ Pb is the cost per Backbone cell
- ➔ Nr is the number of Routers used
- ➔ Pr is the cost per Router

A solution found to have an higher value for a given building plan may be a solution that covers more *target cells*, while respecting the budget and having lower implementation costs.

Work already done and Tools in use

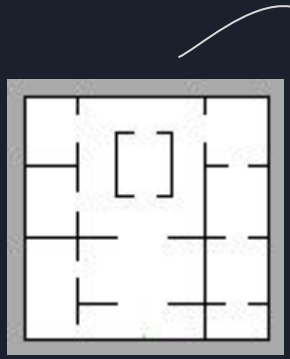
- Tools

➔ **Programming Language:** Python3

➔ **Development Environment:** Visual Studio Code

➔ **File Structure:** The input file may be an image or an ASCII representation of the building, with only the void, target and wall cells. It can also contain the information about the budget and the costs. The solution file will consist on the same schema, but with the router(s) placed, along with the cells within the range and the backbone links. An image of the solution will be generated, with the representative colors.

- Resolution Workflow

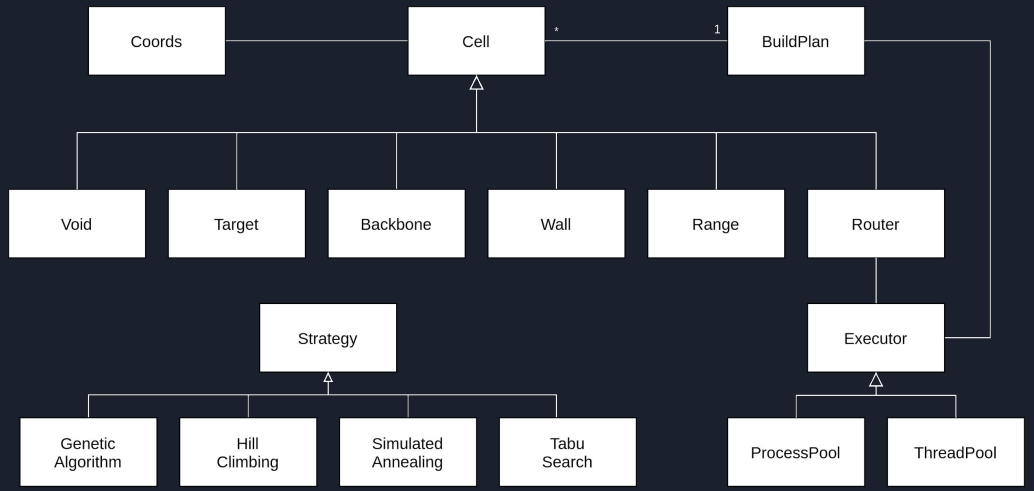


- ASCII Conversion
- Algorithm Application
- Solution Generation
- Image Conversion

ASCII Symbol	Meaning	Color
-	void	grey
.	target cell	white
#	wall cell	black
R	router	blue
r	router range	light blue
b	backbone	green

Work already done and Additional Features

- Classes Diagram



- Additional Features

➔ **Algorithms:**

- Kruskal's algorithm - for generating a minimum spanning tree of backbone links between the routers and the initial backbone cell.
- Parallel flood fill algorithm - custom layered flood fill for generating the set of target cells in a router's range.

➔ **Parallel Computing**:

- ProcessPoolExecutor - for parallel generation of each router's set of target cells.
- ThreadPoolExecutor - for parallel calculation of the targeted cells of each layer in the router's range.

Algorithms/Metaheuristics

Hill Climbing

1. Starts by generating an initial solution, and stores it as the best found so far.
2. For each iteration, generates a neighbour of the current best solution.
3. If the score of that neighbour is greater than the current best, it stores that neighbour as the best solution found so far. Ignores the neighbour otherwise, skipping to the next iteration.

Simulated Annealing

1. Starts by generating an initial solution, and stores it as the current chosen solution.
2. For each iteration, calculates a new factor, the temperature, which decreases each iteration and it generates a neighbour of the current chosen solution.
3. If the score of that neighbour is greater than the current stored solution, or if a random probability based on the temperature is satisfied, it stores the neighbour as the current chosen solution. Ignores the neighbour otherwise, skipping to the next iteration.

Tabu Search

1. Starts by generating an initial solution, and stores it as the current chosen solution.
2. For each iteration, generates a neighbour of the current solution.
3. If the neighbour is considered a Tabu solution (it is similar to previous analyzed solutions), it skips to the next iteration. If the score of that neighbour is greater than the current chosen solution and it's not a Tabu solution, it chooses that neighbour and adds it to the Tabu list.

Genetic Algorithm

1. Starts by generating an initial population, composed by N solutions.
2. For each iteration, the best solution of the current population is selected, along with another random member, and a "child" solution is created based on their characteristics.
3. Then, the child replaces the member of the population with the worst score, and continues to the next iteration.

Local-Search

Common procedures

- Each Local-Search based algorithm starts by generating a random solution, in which the number of routers is calculated based on the number of target cells and the router range, taking into account that the price of those routers is less than the available budget.
- Each of these algorithms also needs to retrieve a neighbour of a given solution. Our neighbourhood function starts by changing a random number of routers, and replacing them with a respective neighbour, that consists of a router placed close to it - the difference between their coordinates must be less or equal to the router range.

Tabu Search

- The criteria to decide if a given solution is Tabu consists of comparing its score with the scores of each solution in our Tabu list, in order to avoid similar solutions.
- The size of the Tabu list, which we designate "*tabu tenure*", is dynamic, since it varies depending on the consecutive tabu solutions found (never leaving the interval between the minimum and maximum size, chosen by the user):
  - if tabu, the size of the list increases by 1,
  - otherwise, the size of the list decreases by 1.

```
def isTabu(self, score, tabuList):
    router_range = int(self.buildPlan.configs["router-range"])
    value_range = math.pow(10,
                           math.floor(
                           math.log10(
                           max(router_range, 1))))
    for tabu in tabuList:
        if (score >= tabu - value_range and score <= tabu):
            return True
    return False
```

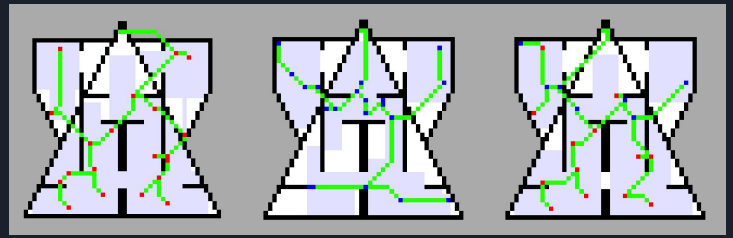
Simulated Annealing

- The factor used to calculate the probability of accepting a solution worse than our current solution, which we designate "*temperature*", is calculated according to the current iteration and decreases as time passes.
- If that probability is greater than a random generated percentage, then the current solution is accepted, even if it has a smaller score.

```
score_diff = (new_score - self.score)
fraction = i / float(iterations)
temperature = max(0.01, min(1, 1 - fraction)) *
      math.pow(10,
      math.floor(
      math.log10(max(abs(score_diff), 1))))
percentage = math.exp(-abs(score_diff) / temperature)
random_percentage = random.uniform(0, 0.5)

if score_diff > 0 or percentage > random_percentage:
      # Accept the solution
```

Genetic Algorithm



parent 1 parent 2 child

- **Selection:** For the parents, each generation, we choose the best member of the current population and another random member.
- **Crossover:** A new "child" solution is generated by merging the characteristics of these two parents. This is done by choosing a random number of routers from the first parent, and the rest of the routers are inherited from the other parent.
- **Mutation:** For each router of the second parent, there's a 20% chance of occurring a mutation, by choosing a neighbour of that router instead of itself.
- **Elitism:** This child replaces the member of the population with the lowest score, being part of the next generation of solutions.
- **Final Score:** The final solution will be the best out of the members of the last generated population.

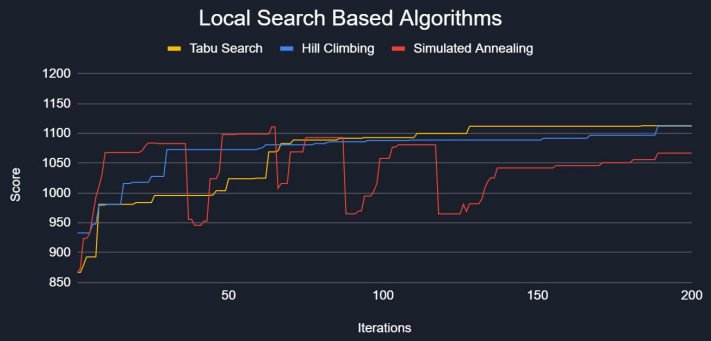
Experimental results

To compare the results of the different algorithms, we ran the program for maps with distinct dimensions:

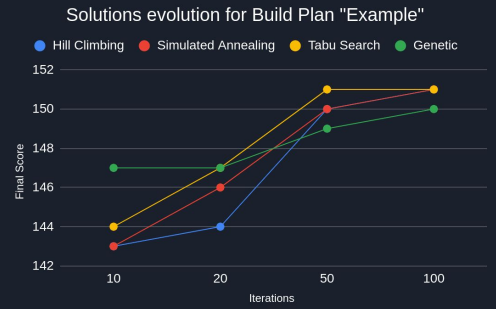
Example (22x8), Space-Ship (50x50), Office (100x100)

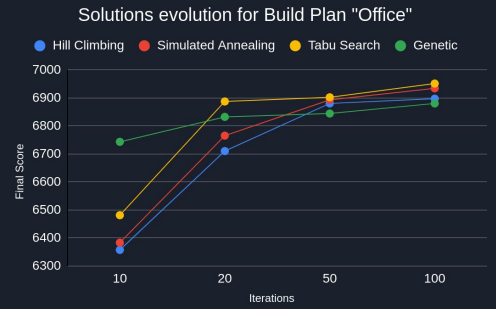
For each map, we tested our code for a different number of iterations (10, 20, 50, 100), and the score obtained for each one of them consists of the average of the final scores of 10 executions.

We can conclude that for a low number of iterations, the **Genetic Algorithm** results on a much higher score than the other three. This is due to the fact that it starts with a generation of 10 solutions, and the others only have one random solution to start with. Over time, this advantage goes away, probably because of the genes propagation, that restricts the characteristics of the next generations to the parents characteristics.



Overall, the **Tabu-Search** algorithm results in the best scores, especially as the number of iterations increases. On the other hand, the **Hill Climbing** algorithm doesn't achieve the best results, since he might get near to local maximums. This is avoided in the **Simulated Annealing** algorithm, which is a more reliable approach, due to the jumps made to escape local maximums, although it takes more time to achieve the best solutions.

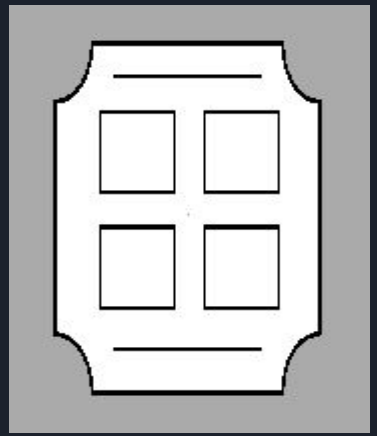


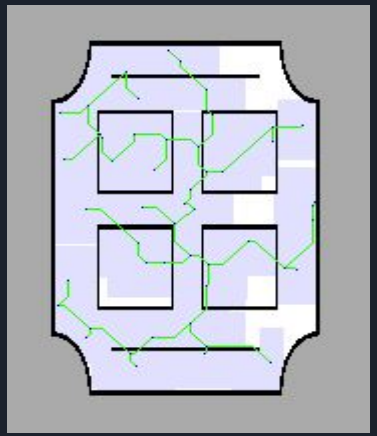


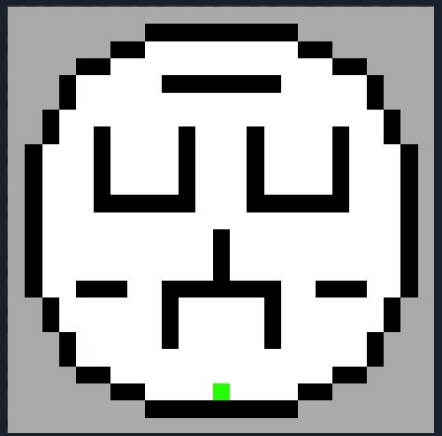
Conclusion

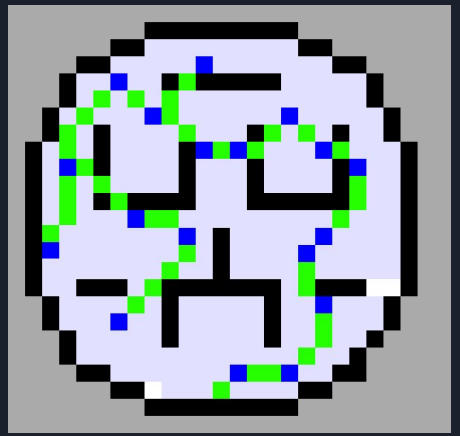
We followed the project enunciate strictly, achieving every requirement and goal that we established when we first discussed our approach. That said, the project has been successfully completed, and during its development we learned new concepts and algorithms from an area over which we had little experience.

The only difficulty we encountered was related to the execution time taken by larger maps, due to the problem typology and consequent generation and evaluation of random solutions each iteration.









References and Related Work

- Mohammed A. Alanezi , Houssem R. E. H. Bouchekara, and Muhammad S. Javaid. (2020). Optimizing Router Placement of Indoor Wireless Sensor Networks in Smart Buildings for IoT Applications.
- Google #Hash Code 2017 Challenge "Router placement" - Team Gyrating Flibbittygibbitts.
- Introduction to Mutation, Crossover Operators, Survivor Selection - Genetic Algorithms, TutorialsPoint
- Google #Hash Code 2018 Challenge “City Plan - Optimization Problem for Public Projects Implementation”