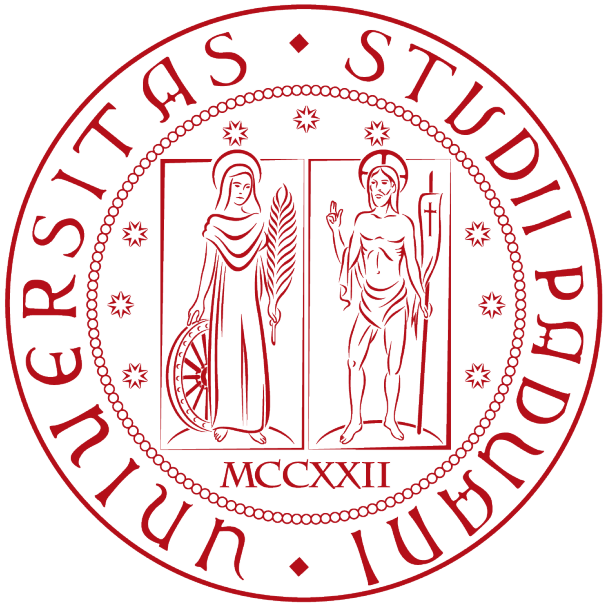

Project Report

Methods and Models for Combinatorial Optimization

Protopapa Francesco

2023/01/15

Index

1. Introduction

Introduction Introduction

2. Problem modelling

2.1. Problem description

A company produces boards with holes used to build electric panels. Boards are positioned over a machine and a drill moves over the board, stops at the desired positions and makes the holes. Once a board is drilled, a new board is positioned and the process is iterated many times. Given the position of the holes on the board, the company asks us to determine the hole sequence that minimizes the total drilling time, taking into account that the time needed for making an hole is the same and constant for all the holes.

2.2. Input representation

An instance of the problem is represented as a list of points on the Cartesian plane. The first line of an input instance contains a single integer n, the number of points. Each of the following n lines contains two doubles separeted by a space.

2.3. Holes generation

The class `InputGenerator` deals with generating an instance of the problem. Since the abstract of the problem is about making holes on a motherboard, input instances are not generated completely at random, instead, a given number of random shapes representing the components of the motherboard is generated. There are four different shapes (line, rectangle, circle, set of points) and each shape has random dimensions.

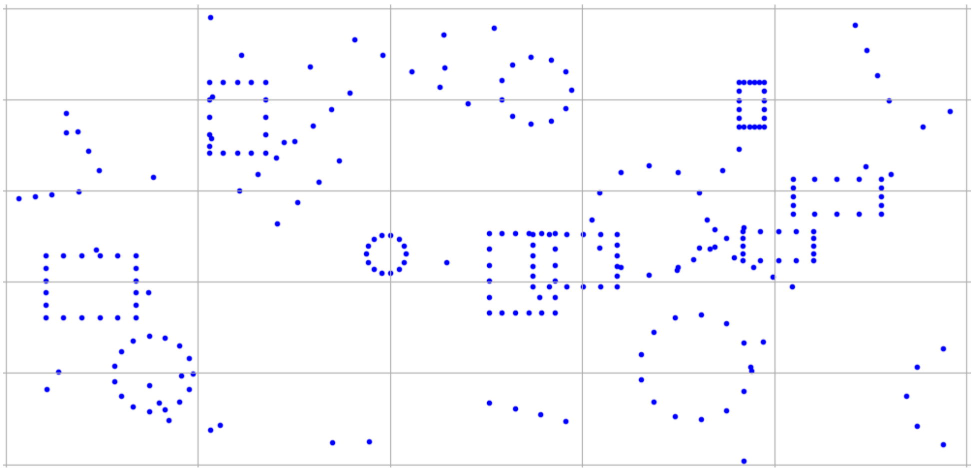

Figure 2: Input instance example

2.4. Problem representation

- A `Graph` is represented as a matrix of dimensions $n \times n$ where n is the number of holes and `graph[i][j]` is the euclidean distance between holes `i` and `j`.
- A `Path` represents a solution to the problem, the path is represented as a list of integer.
- A `Point` represents an hole on the motherboard and is represented as a pair of doubles.

2.4.1. Example

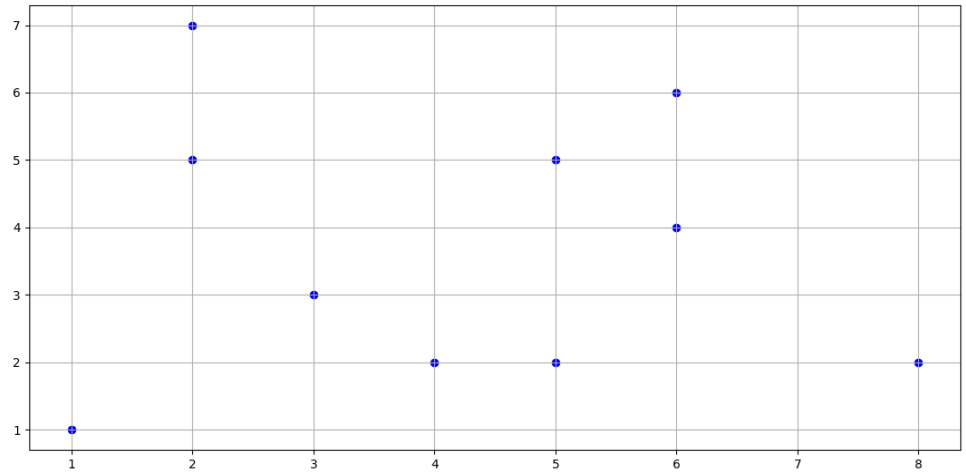

Figure 3: Graph example

The graph in the Figure 3 corresponds to the following points:

$$(1,1);(2,5);(3,3);(6,4);(2,7);(5,5);(6,6);(8,2);(4,2);(5,2)$$

And is represented in this way:

$$[0.00, 4.12, 2.83, 5.83, 6.08, 5.66, 7.07, 7.07, 3.16, 4.12];$$
$$[4.12, 0.00, 2.24, 4.12, 2.00, 3.00, 4.12, 6.71, 3.61, 4.24];$$
$$[2.83, 2.24, 0.00, 3.16, 4.12, 2.83, 4.24, 5.10, 1.41, 2.24];$$
$$[5.83, 4.12, 3.16, 0.00, 5.00, 1.41, 2.00, 2.83, 2.83, 2.24];$$
$$[6.08, 2.00, 4.12, 5.00, 0.00, 3.61, 4.12, 7.81, 5.39, 5.83];$$
$$[5.66, 3.00, 2.83, 1.41, 3.61, 0.00, 1.41, 4.24, 3.16, 3.00];$$
$$[7.07, 4.12, 4.24, 2.00, 4.12, 1.41, 0.00, 4.47, 4.47, 4.12];$$
$$[7.07, 6.71, 5.10, 2.83, 7.81, 4.24, 4.47, 0.00, 4.00, 3.00];$$
$$[3.16, 3.61, 1.41, 2.83, 5.39, 3.16, 4.47, 4.00, 0.00, 1.00];$$
$$[4.12, 4.24, 2.24, 2.24, 5.83, 3.00, 4.12, 3.00, 1.00, 0.00]$$

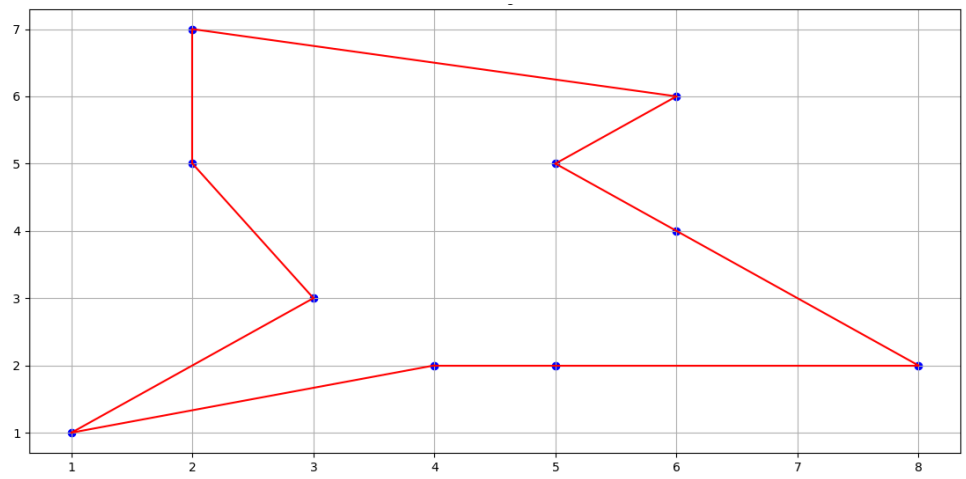

Figure 4: Path example

Finally the path in the Figure 4 is represented as follows:

$$[0, 2, 1, 4, 6, 5, 3, 7, 9, 8]$$

3. Cplex solver

The model is implemented and solved with the Cplex APIs by the class `SimplexSolver`.

3.1. Row and columns generation

Rows and columns are created one at a time by the method `setupLP`. Model implementation is straightforward and there is only one constraint that need some kind of reasoning.

$$\sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 \qquad \forall k \in N \setminus \{0\}$$

Since the variables x_{kk} appears on both summations, in order to prevent cplex errors, it is necessary to avoid adding its index twice. Since the coefficient of the variable is 1 on the first summation and -1 on the second one, we can just ignore it. This operation is performed by the following code:

```
for (int i = 0; i < size; i++) {
    if (i == k) continue; // avoid adding x_k_k
    coef.push_back(1);
    idx.push_back(tuple_to_index[{'x', i, k}]);
}
```

3.2. Mapping variables

Variable indexes are stored inside an `std::map<std::tuple<char, int, int>, int>` called `tuple_to_index`. Since `std::map` has complexity of $O(\log n)$ for accessing and adding a value, this way of mapping the indexes brings the model generation complexity to $O(n^2 \log n)$ while using an `std::vector` would have kept the complexity to $O(n^2)$. It has been decided to use an `std::map` because it makes the code way more readable allowing to use the following syntax to access an index: `tuple_to_index[{'y', i, j}]`. Moreover model generation is not the bottleneck of the cplex solver since the model solution is way more expensive and so the difference between $O(n^2 \log n)$ and $O(n^2)$ during the model generation is not relevant.

3.3. Result retrieving

In order to retrieve the results it is necessary to make the opposite of what has been done during the model generation, in fact it is needed to map indexes to variables. To do so it is used an `std::vector<std::tuple<char, int, int>>` called `index_to_tuple`. Moreover the result of the problem should be a `Path` as described before, but since cplex output is just an array of values, some reasoning is neede.

3.3.1. Path retrieving algorithm

```
std::vector<double> var_values(index_to_tuple.size()); // vector that will
contain cplex result
std::vector<std::pair<int, int>> arcs; // vector with arcs selected by cplex
CPXgetx(env, lp, &var_values[0], 0, var_values.size() - 1);

for (int i = 0; i < var_values.size(); i++) {
    auto [c, a, b] = index_to_tuple[i];
    if (c == 'y' && equal(var_values[i], 1)) {
```

```
            arcs.emplace_back(a, b); // an arc is added only if the var is y and the
value is 1
        }
    }
    std::sort(arcs.begin(), arcs.end()); // sort arcs for easier access in the next
steps
    std::vector<int> ans; // vector to store the result

    auto [cur, next] = arcs[0];
    // since it is guaranteed that the path is well formed, we can proceed in this
way
    do {
        ans.push_back(cur);
        cur = next;
        next = arcs[cur].second;
    } while (cur != 0);

    return Path(ans);
```

4. Tabu search

4.1. Design choices

In order to be as efficient as possible, the tabu-list is implemented in the following way:

- an `std::deque<Path>` called `tabu_list` is used to store the paths inside the tabu_list;
- an `std::map<Path, bool>` called `tabu_map` is used to know efficiently which paths are inside the tabu-list.

Considering an input with n holes and a tabu-list of maximum legth m the operations performed on the two data structures have the following complexities:

- $O(n)$ for pushing on the front or popping on the back a path in the tabu-list
- $O(n \log m)$ for updating and checking whether a path is in the tabu-list or not.

Using an `std::deque` improves the performance compared to an `std::vector` when pushing on the front because the second has to perform a shift of all the elements with complexity $O(nm)$. Also using an `std::map` to check whether a path is in the tabu-list improves the performance since iterating over the tabu-list costs $O(nm)$ in the worst case. Since `std::map` is implemented as a balanced BST, it was also necessary to implement a comparison operator between paths that has been implemented as lexicographic (this is why the complexity of operations on the tabu-map is $O(n \log m)$ and not $O(\log m)$).

4.2. Initial solution

Since inputs are not completely random, starting from a path in which shapes are grouped together is already a good solution. Grouping parts is obtained for free since the input generation puts them in the right order. This approach is also coherent with the abstract of the problem since a company tha produces motherboards will probably know the parts of the motherboard. In order to test different approaches, also a randomized initial path has been implemented.

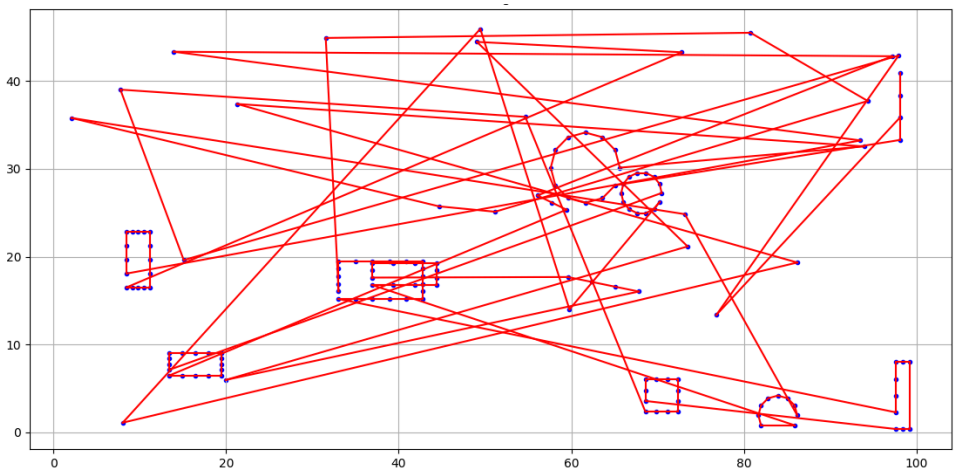

Figure 5: Initial solution with shapes grouped

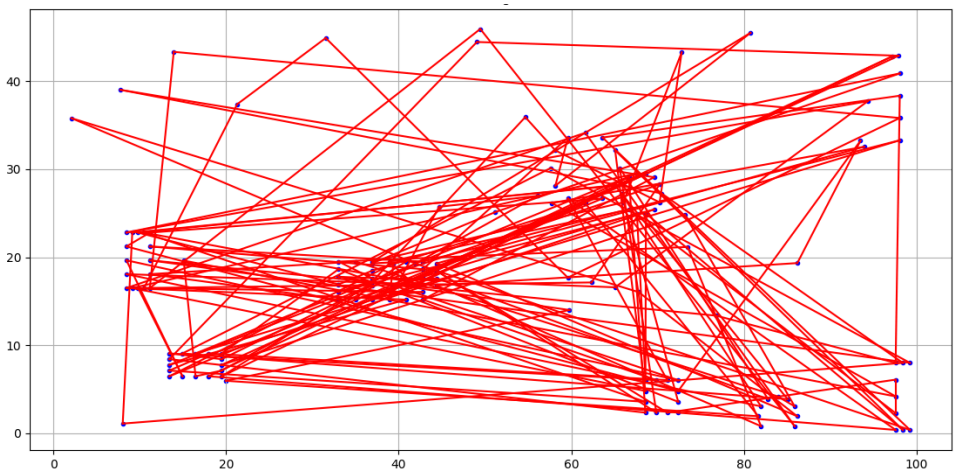

Figure 6: Initial random solution

4.3. Stopping criteria

The following stopping criteria has been implemented:

- max iterations;
- max non increasing iterations;
- time limit.

Their implementation is straightforward and there are not any interesting design choices.

4.4. Parameters

The parameters that can be calibrated are the following:

- max iterations;
- max non increasing iterations;
- time limit;
- tabu-list length.

5. Computational results