Simulated-Annealing Cell Placement Tool

CSCE 3304 - Digital Design II

Supervisor: Dr. Mohamed Shalan

THE AMERICAN UNIVERSITY IN CAIRO

Youssif Abuzied (900202802)

Amer Elsheikh (900196010)

Ahmed Eltokhy

Table of Contents

Overview

In this project, we implement a simple cell placement tool that places the digital cells in a grid in order to minimize the total wire length of the nets. The cells and their net connections are given. To do that, we use the simulated annealing algorithm.

Simulated Annealing Algorithm

We mainly follow the simulated annealing algorithm to reach a good minimum and avoid getting stuck in bad local minimas.

The algorithm can be described as follows:

```
- Create an initial random placement
- Calculate the initial cost
- T = T_init
- While (T> T_final)
    - num_moves = 20 * num_cells
    - while(num_moves !=0)
        - num_moves = num_moves - 1
        - Pick 2 random locations in the grid (one may be empty)
        - Swap the two cells and calculate Δcost
        - if (Δcost < 0) accept
        - Else reject with probability $(1-e^{-\Delta cost/T})$
    - T = T * cooling_rate
```

Assumptions

In the project, we assume the following about the algorithm:

- HPWL (half-perimeter of the smallest bounding box containing all pins for a net) is used to estimate the wire length of any net
- The core area is a 2D array of empty squares (sites)
- Each cell is a square and matches the site size; the site size is 1x1.
- No site is assigned to more than one cell.
- The distance between two cells is measured from the center of one cell to the center of the other

And, we make the following assumptions about the input file. It is a txt file with the following format:

- The first line of the file contains 4 values separated by spaces:
    - The first value is the number of cells to be placed.
    - The second is the number of nets.
    - The third is the number of rows of the grid.
    - The fourth is the number of columns of the grid.

- Then a line for each net follows which consist of numbers separated by spaces such as:
    - The first number is the number of cells this specific net connects.
    - Then, the list of these cells (each cell has an ID number from 0 to num cells -1)

Command Line Input and Dependencies

To run the program, you need the following:

- C++ Compiler (we recommend GCC)
- Python 3.x
- Python Libraries: PIL (Python Imaging Library); it is needed for the gif generation.

Then, you can compile the program as follows:

```
g++ -O3 -o program main.cpp
```

We recommend using “O3” for best performance.

Now, to run the program normally on input “d1.txt” with cooling rate “0.95” for example, you do the following:

```
./program d1.txt 0.95
```

And you will get the initial random grid, its cost, then the final grid, its cost and the final temperature.

However, to see the process of swapping through an animation gif, just add the flag -s to the command line. For the above example, we run:

```
./program d1.txt 0.95 -s
```

And a gif will be generated along with the above outputs.

Implementation and Design

In this section, we explore our specific implementation and design for the algorithm explained above.

Data Structures

We mainly used vectors since they provided the best performance as accessing a vector is O(1). We treated the vectors as hash tables. This was possible because each cell was given an ID from 0 to “num_cells - 1” and similarly every net was given an ID from 0 to “num_nets - 1”.

We hence mainly used the following vectors:

```
vector<vector<int>> grid;
vector<vector<int>> net_to_cell, cell_to_nets;
vector<pair<int, int>> placement;
vector<int> net_len;
```

Where:

- "grid" is a 2-D vector that has the specified number of rows and columns as given by the input. For each cell of the grid, it is either empty (and we put -1), or there is a digital component (and we put the ID of the component)
- "placement" is the reverse of "grid". It maps every digital cell to its location on the grid. This location is represented by a pair of integers (its location on the grid).
- "net_to_cell" maps every net to a vector of digital cells that this net connects,
- "cell_to_nets" is the reverse of "net_to_cell", and it maps every digital cell to a vector that includes all the nets that pass through this cell
- "net_len" maps every net to its cost.

We used these data structures through the program stages which we divided into modular functions as follows:

Input Parsing and Vectors Population

We have a function "parse_input_and_populate_nets()" that parses the input and loops over every net in the given data file. Then for every net, it loops over all the cells connected to this net and populates the "cell_to_nets" and "net_to_cell" vectors accordingly.

Random Placement

We have a function called "random_placemnt". It initializes the "placement" and "grid" vectors randomly as follows:

```
- For each cell i
    - sample x_random less than number of rows
    - sample y_random less than number of columns
    - While (grid[x_random][y_random] is not empty) sample x_random and y_random
    - Assign grid[x_random][y_random] = i
    - Assign placement[i] = (x_random, y_random)
```

Time Complexity:
Best case: O(n) where n is the number of cells.
Average case: O(n * c): where n is the number of cells and c is the number of collisions.
Worst case: O(n^2), where n is the number of cells. This scenario happens when we try to place a cell, we will always collide with all the placed cells already until we find a free location on the grid.

Cost Calculation

We have a function “calc_cost(idx)” that takes the index of any net and calculate its cost according to the populated “placement” vector using HPWL estimate as shown in the next piece of code:

```
int calc_cost(int idx) { // net's index
   int dist = 0;

   auto &cur_n2c = net_to_cell[idx];
   int maxx = 0, maxy = 0, minx = INT_MAX, miny = INT_MAX;

   for (auto &x : cur_n2c) {
       auto u = placement[x];
       maxx = max(maxx, u.first);
       maxy = max(maxy, u.second);
       minx = min(minx, u.first);
       miny = min(miny, u.second);
   }

   return (maxx - minx) + (maxy - miny);
}
```

The function simply loops over all the cells that the given net connects and keeps track of the Max_x, Max_y, Min_x, Min_y of the placement of these cells.

It then calculates the HPWL cost by returning “(maxx - minx) + (maxy - miny)”

This function is used to calculate the total initial_cost and populate the “net_len” vector.
Time complexity: O(n), where n is the number of cells this net connects.

Grid Printing

We have two functions for printing:

- “print_grid” which prints the current state of the grid. If there is an empty cell, it simply prints “--”. Otherwise, it prints the ID of the digital cell. Moreover, it prints the current Cost of the grid at the end.
- “print_binary_grid” which prints whether every cell in the grid is occupied or not. If it is occupied, it puts 1 in it. Otherwise, it puts 0

Time Complexity: O(nm) where n is the number of rows and m is the number of columns.

Main Loop and Change in the total wire length

For the main logic, we initialize the “T_cur” and “T_fin” such as

```
   auto T_cur = 500 * initial_cost;;
   double T_fin = 5e-6 * initial_cost / (double) num_nets;
```

And, we perform “20 * num_cells” trials of swaps per each temperature. Now, for each trial, we do the following:

1. Sample a point (U_x, U_y) such that there is a digital cell placed at (U_x, U_y). We sample from the placement vector to be sure of this.
2. Sample a coordinate (V_x, V_y) from the grid. This coordinate can be empty or have a cell.
3. Swap both coordinates
4. Calculate the new_cost after swap through:
    a. Initialize new_cost = initial_cost
    b. For every net n in cells_tonets[U]:
        i. new_cost -= net_len[n] #subtract old cost
        ii. temp_net_len[n] = calc_cost(n)
        iii. new_cost += temp_net_len[n]
    c. Do the same for cell_to_nets[V] while making sure not to compute the cost any common nets between U and V (since they were already computed in step b)
5. Assign rej_porb = (1-$e^{-\Delta cost/T}$) and sample should_I_reject accordingly from a uniform distribution
6. if (new_cost >= cure_cost && should_I_reject): swap back U and V
7. Otherwise: update the cur_cost and net_len with temp_net_len

After finishing the trials per one temperature, we update the temperature and continue until it is less than the final temperature.

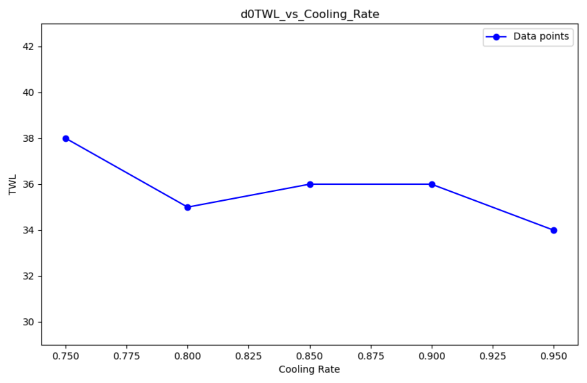
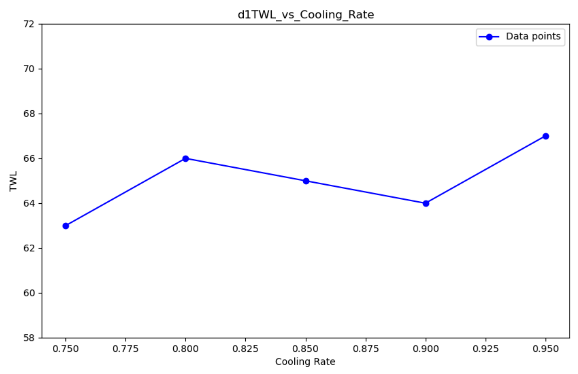
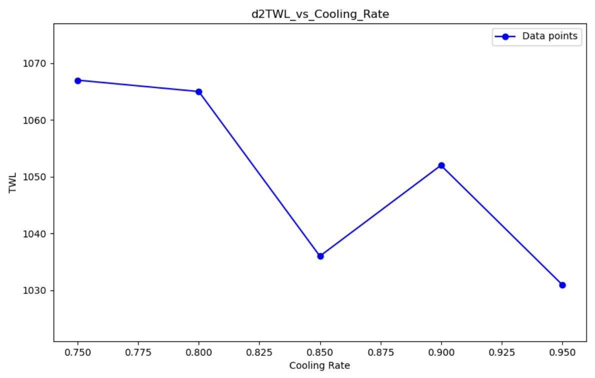
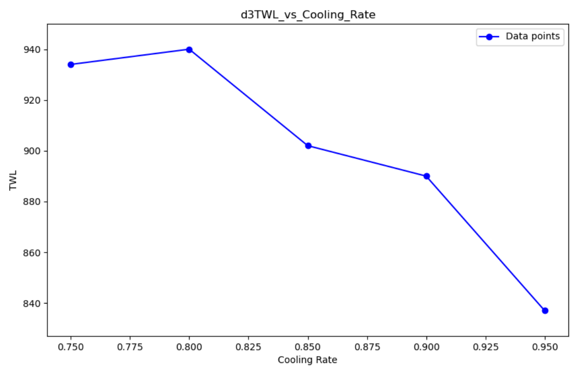
This implementation ensures that we only compute the cost of the affected nets after every potential swap. This gives much better performance than computing the whole cost every time one attempts a swap

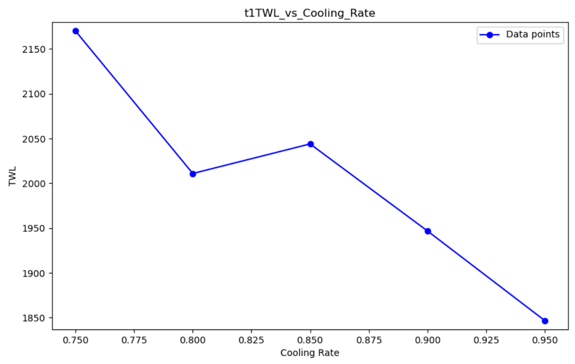
Graphs and Conclusions

To get some insights about what happens during the algorithm and also to judge the correctness of our algorithm, we graphed the relation between the total wire length and the cooling rate and the relation between the total wire length and the temperature.

First, TWL vs Cooling Rate:

Design	Graph

d0		d0TWL_vs_Cooling_Rate Data points TWL Cooling Rate
d1		d1TWL_vs_Cooling_Rate Data points TWL Cooling Rate
d2		d2TWL_vs_Cooling_Rate Data points TWL Cooling Rate
d3		d3TWL_vs_Cooling_Rate Data points TWL Cooling Rate

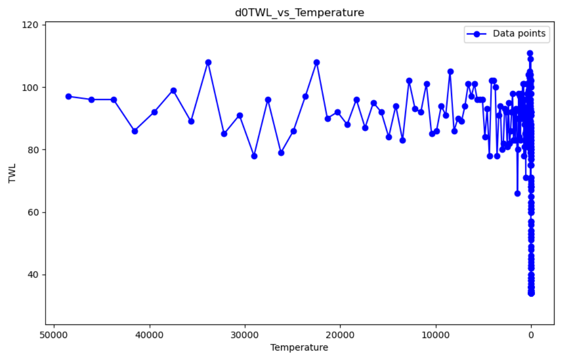
t1	

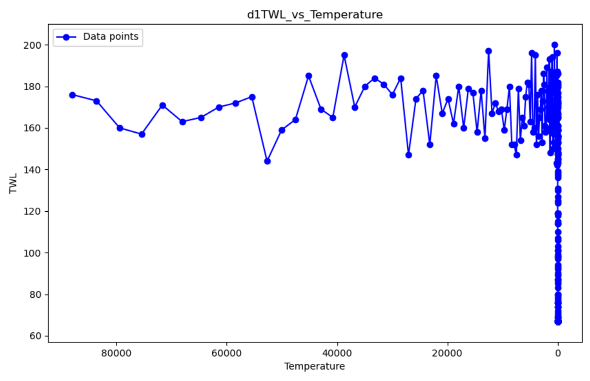
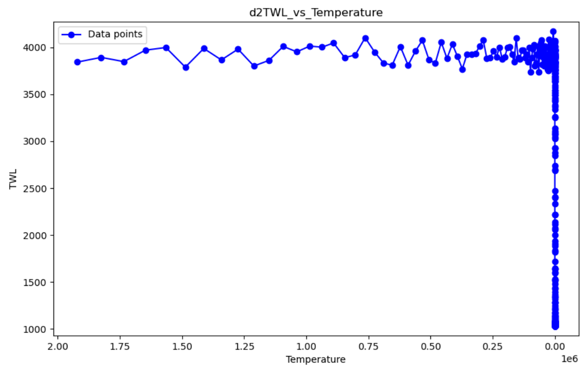
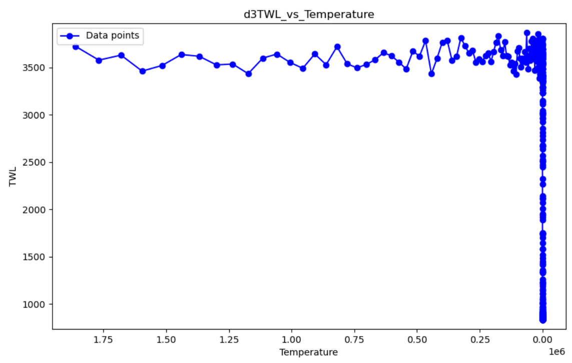
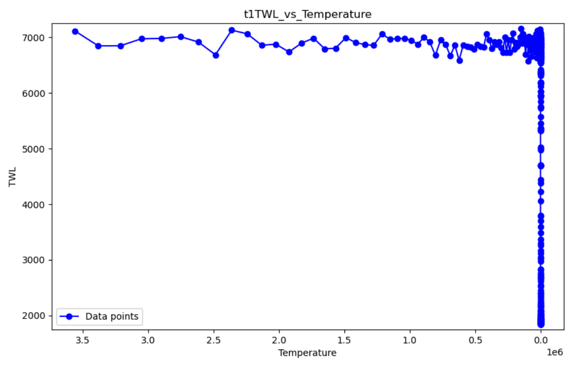
Conclusions:

From the above graphs, we can tell that increasing the cooling rate reduces the final TWL. This can be attributed to the fact that when the cooling rate increases, this allows the placer to accept more bad swaps. As a result, the placer will explore more placements and avoid being stuck at a local optima.

Second, TWL vs temperature.

To get more insight about how the TWL changes while the algorithm is running, we plotted the wire length vs the temperature at a fixed cooling rate (0.95) for the 5 designs we have.

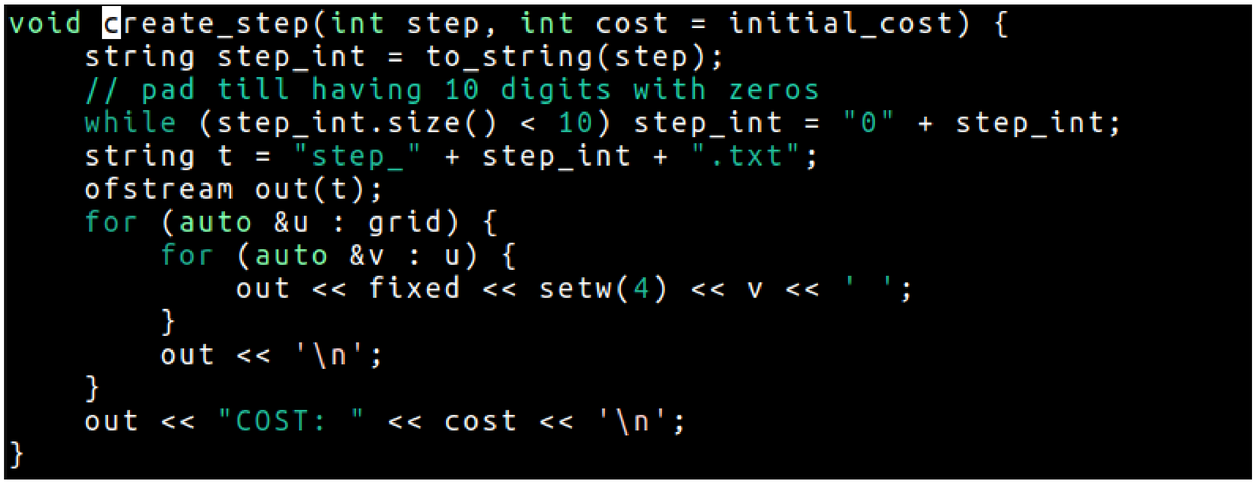
Design	Graph
d0	

d1	 d1TWL_vs_Temperature Data points TWL Temperature 200 180 160 140 120 100 80 60 80000 60000 40000 20000 0
d2	 d2TWL_vs_Temperature Data points TWL Temperature 1e6 4000 3500 3000 2500 2000 1500 1000 2.00 1.75 1.50 1.25 1.00 0.75 0.50 0.25 0.00
d3	 d3TWL_vs_Temperature Data points TWL Temperature 1e6 3500 3000 2500 2000 1500 1000 1.75 1.50 1.25 1.00 0.75 0.50 0.25 0.00
t1	 t1TWL_vs_Temperature Data points TWL Temperature 1e6 7000 6000 5000 4000 3000 2000 3.5 3.0 2.5 2.0 1.5 1.0 0.5 0.0

From the above graphs, we can see that the wire length remains constant for some time but when the temperature decreases to a given value, the wire length starts to decrease drastically. This can be explained by the fact that when the temperature decreases the rejection probability of the bad swaps increases exponentially. As a result, the total wire length decreases greatly for lower temperatures.

Bonus Implementation

To implement the GIF simulation, the implementation had two primary steps that are done with the C++ code and a Python script. The C++ code includes a function to create a step file for each iteration of the Simulated Annealing algorithm, storing the current state of the grid and its cost.

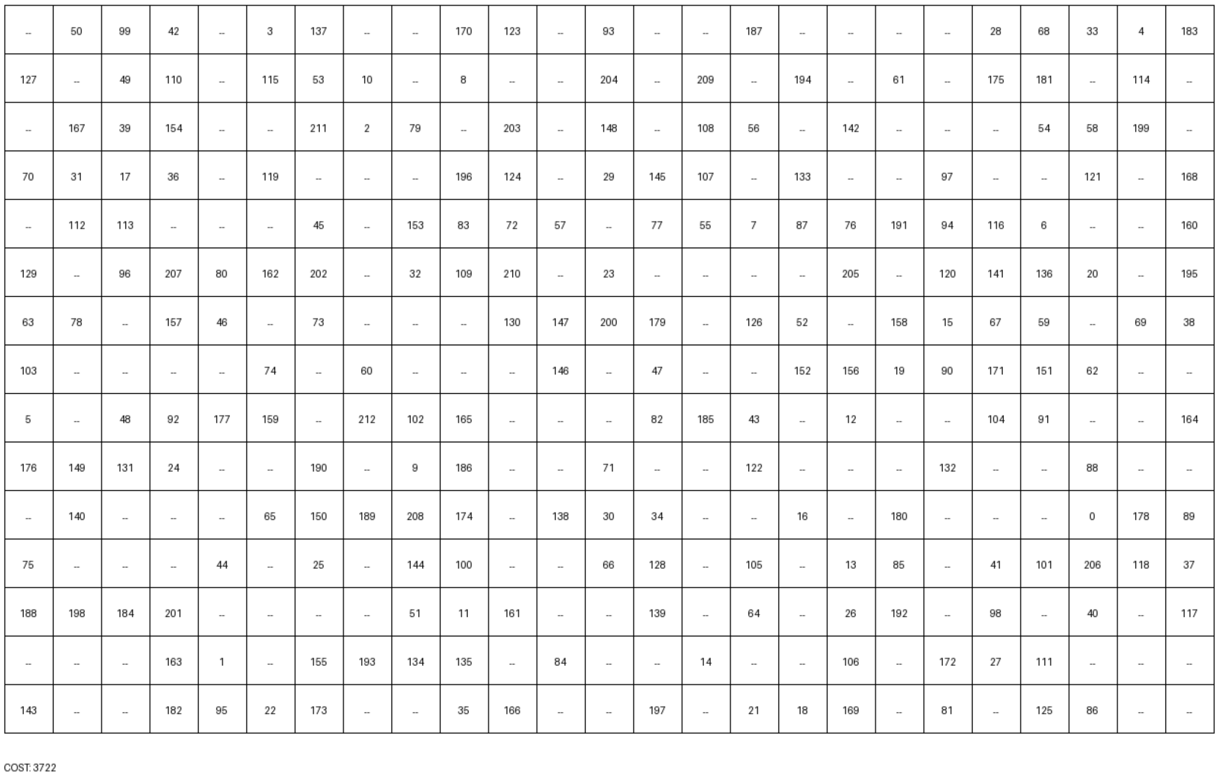


```
void create_step(int step, int cost = initial_cost) {
    string step_int = to_string(step);
    // pad till having 10 digits with zeros
    while (step_int.size() < 10) step_int = "0" + step_int;
    string t = "step_" + step_int + ".txt";
    ofstream out(t);
    for (auto &u : grid) {
        for (auto &v : u) {
            out << fixed << setw(4) << v << ' ';
        }
        out << '\n';
    }
    out << "COST: " << cost << '\n';
}
```

That function was used to create step text files with the grid's information during that step. We have assumed that the steps that are needed to be tracked are the initial placement, each T_current adjustment, and the final placement to limit the number of steps to a reasonable and a properly visualizable version.

Then, a command is being run to execute the python script responsible for the GIF creation. For this command to run, a linux system is assumed to be used. If not, the python script will need to be run manually.

The Python script, 'create_gif.py', reads these step files and creates an image for each step, using the binary representation of the grid to visualize the placement of cells. The images are then combined into a GIF using the Python Imaging Library (PIL). The script assumes that the input file is in the correct format, the grid size is large enough, the cooling rate is a positive number less than 1, and the Python script has the necessary permissions and libraries installed.It also assumes that the system has a Python interpreter installed and the necessary permissions to execute the script and create and write to the output GIF file. We have also limited the number of frames to 200 frames (with frame skipping implemented for versions with more frames) to make the visualizations descriptive enough with each frame being run for 200 milliseconds. This step limit can be changed through a command line argument while the duration of each step can be changed through the function parsing. The resulting GIF provides a visual representation of the Simulated Annealing algorithm's progress in finding the optimal placement of cells in the grid with the cost and placement visibly viewed in each step.



--	50	99	42	--	3	137	--	--	170	123	--	93	--	--	187	--	--	--	--	28	68	33	4	183
127	--	49	110	--	115	53	10	--	8	--	--	204	--	209	--	194	--	61	--	175	181	--	114	--
--	167	39	154	--	--	211	2	79	--	203	--	148	--	108	56	--	142	--	--	--	54	58	199	--
70	31	17	36	--	119	--	--	--	196	124	--	29	145	107	--	133	--	--	97	--	--	121	--	168
--	112	113	--	--	--	45	--	153	83	72	57	--	77	55	7	87	76	191	94	116	6	--	--	160
129	--	96	207	80	162	202	--	32	109	210	--	23	--	--	--	--	205	--	120	141	136	20	--	195
63	78	--	157	46	--	73	--	--	--	130	147	200	179	--	126	52	--	158	15	67	59	--	69	38
103	--	--	--	--	74	--	60	--	--	--	146	--	47	--	--	152	156	19	90	171	151	62	--	--
5	--	48	92	177	159	--	212	102	165	--	--	--	82	185	43	--	12	--	--	104	91	--	--	164
176	149	131	24	--	--	190	--	9	186	--	--	71	--	--	122	--	--	--	132	--	--	88	--	--
--	140	--	--	--	65	150	189	208	174	--	138	30	34	--	--	16	--	180	--	--	--	0	178	89
75	--	--	--	44	--	25	--	144	100	--	--	66	128	--	105	--	13	85	--	41	101	206	118	37
188	198	184	201	--	--	--	--	51	11	161	--	--	139	--	64	--	26	192	--	98	--	40	--	117
--	--	--	163	1	--	155	193	134	135	--	84	--	--	14	--	--	106	--	172	27	111	--	--	--
143	--	--	182	95	22	173	--	--	35	166	--	--	197	--	21	18	169	--	81	--	125	86	--	--

COST: 3722