Project 1: Pentocity

Group 1: Chengyu Lin, Kailash Meiyappan, Ying Sheng

October 2016

Contents

1 Introduction

2 Utilities for Analysis

2.1 Necessity for average utilities

Since a single run of Pentocity has too much randomness, both in the sequencer and in the player, we developed some evaluation metrics. We modified the simulator to take as input an integer variable "repeats" that instructs the simulator to run the player "repeats" number of times and give the average of the different metrics we defined. The important metrics we used have been defined below.

2.2 Utility Descriptions

2.2.1 Player Score

This metric is the score at the end of the game and is our primary method of evaluating an idea. It is simply the score returned by the simulator. Clearly, we aim to maximise this.

2.2.2 Number of Empty Cells

This tells us how much whitespace is left unused so we can judge how well our algorithm packs buildings together. We aim to minimize this. This is useful to evaluate our building placement in our function "getBestMove".

2.2.3 Number of Road Cells

The total number of cells used to build roads. We use this to evaluate our road construction approach in the function "findShortestRoad".

2.2.4 Utility of Water/Park Cells

The sum of the total score that the water generates and the total score that the parks generate divided by the total number of cells. This was used to evaluate our water and park building strategy in the function optimizeWaterAndPark.

2.3 Discussion of Utilities

In any future description of scores and utilities, we mean the value of these scores averaged over 50 runs. The reason we picked 50 is as follows. We tried 10, 25 and 50 runs. 10 runs still had too much variance, and could differ from the 50 run score by as much as 40 points. 25 runs worked reasonably well (differing by at most 20 points), but we were not satisfied. 50 runs did not differ by more than 10 points between two separate trials and completed in reasonable time, which is why we chose it.
It is worth noting that each of these metrics is not independent of each other. For example, when we implemented our perimeter strategy, our score went up by around 300 points. This change was in the getBestMove function and greatly reduced the number

of empty cells from our naive strategy, but that was not the only change. The number of road cells also decreased, from around 500 cells to fewer than 400 and the utility of the ponds and parks increased by around 0.2. This led us to realise that we should look at all utilites, even for a change in an unrelated function.

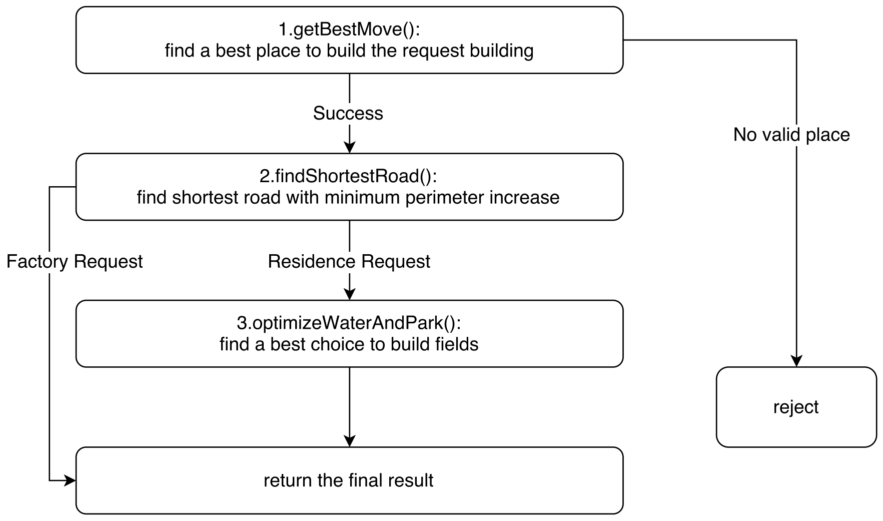
1.getBestMove():
find a best place to build the request building
Success
No valid place
2.findShortestRoad():
find shortest road with minimum perimeter increase
Factory Request
Residence Request
3.optimizeWaterAndPark():
find a best choice to build fields
reject
return the final result

Figure 1: high level algorithm flow chart of our submitted approach

3 Strategies

3.1 Algorithm process

Figure 1 shows the high level process of our algorithm. It first picks the best valid location to build the requested building, which is chosen based on several criteria. Then, build the best road to connect it to an existing road or edge. If the request is a residence, there is an additional step to build an accompanying park and pond. A detailed illustration will be presented in the next subsection.

3.2 Building Placement

The purpose of this module is to find the best place to build the requested building. We look at every possible coordinate that the building could be placed such that it does not overlap with any other buildings, roads, parks or ponds. Then we filter out the coordinates that are in white spaces that are cut off from roads or edges. Then, we try to define the "best" coordinate using different approaches, which will be explained in each of the following sections.
We define a global strategy to be the strategy that decides the overall preference of location for the factories and residences, based solely on the value of its coordinates and not on the position of other buildings.

3.2.1 global: row-by-row

This is a naive strategy that we tried first. The residences are built row by row from the top to bottom, which means that for each residence, iterate over coordinates from the top left to the bottom right each time, finding the first buildable coordinate (which will have the lowest row index). The factories are built row by row from the bottom to the top, which means for each factory, iterate over the rows from the bottom right to top left, finding the first buildable place (which will have the highest row index). The idea behind this approach is to separate the residences and factories. The reason behind this is that factories are rectangular shaped and would fit together well. Similarly, residences being irregular, would not fit with factories. The other reason for separating them is that factories and residences cannot be built adjacent to each other. This means that placing them close to each other would create large white spaces. To avoid this, we build residences at the top and factories at the bottom.

Using this naive approach, we could get a score of around 1600 to 1700, depending on whether or not we used ponds and parks. The implementation of the naive strategy help us more understand the game at the beginning. It also helped us estimate the utility of parks and ponds, since including them increased our score by around 100.
Later, we switched to other strategies.

3.2.2 global: diagonal

This is another strategy of placing residences and factories in a way that separates them from each other. The residences are built from the top left corner to the bottom right corner, which means each time find the buildable coordinate with the lowest sum of row index and column index. The factories are build from the bottom right corner to the top left corner, which means each time find the buildable coordinate with the highest sum of row index and column index. The idea behind this approach is similar to the row-by-row approach in that it tries to separate the residences and factories. However, the diagonal gap is better than the row-by-row strategy because there are a larger number of possible locations with the same $(i + j)$ value to choose from.

In addition to the global diagonal pattern, we add another small features to the selection process. For the candidate spaces that have the same value of $(i + j)$, which denotes the sum of row index and column index, we pick the one that has the lower value of $|(i - j)|$, which means the one closer to the diagonal line. The idea behind this feature is to make the frontier of residences and factories more convex. Since edges are considered roads, building buildings further from the edges first increased the flexibility of how we use the edges later. In addition, placing a factory near the edge always creates very few white spaces, so if we have an equally good option for the factory away from the edge, it would make sense to pick the option away from the edge and save the edge for a future factory.

3.2.3 global: residences at center, factories on edges

This is a different global strategy we have tried. In this strategy, we build factories from the edges to the center and build residences from the center to the edges. The idea behind it is that the factories are more regular in shape so they can fit very well on the edges, while residences have more irregular shapes. But the experiments demonstrated to us that this is not a good strategy, so we gave up on it later. The reason we thought that this is a good strategy is that the factories are easy to fit compactly on the edges. But the problem with this is if we build all the factories near the edges, it makes the whitespace gap between factories and residences longer. This strategy has another problem in that it is vulnerable to adversarial distributions - it increases the difficulty of building roads. If we get an adversarial distribution that requests factories first, this strategy would build factories around the edges and cut off the inner area from the edges. We decided to avoid this strategy for these two reasons.

3.2.4 perimeter

The perimeter strategy is a non-global strategy that looks at the cells around the candidate location for a requested building. For a candidate location, some of its cells are adjacent to already occupied cells in the map and some of its cells are neighbor to vacant cells in the map. The algorithm count the number of edges of the neighbour that it shares with an occupied cell, which is the perimeter of this location. Then we choose the candidate with the largest perimeter. The idea behind it is the larger perimeter means the buildings are more compactly packed. And, it has been shown that perimeter evaluation made a significant contribution to our good performance. Adding the perimeter strategy improved the score of the naive global strategy by around 300 points. Finally, we use the diagonal strategy as our global strategy. However, we only use it to break ties between candidates that have the same perimeter.
We also tried a different way to count the perimeter. That is to count the number of cells surrounding the candidate location, should the building be placed there. This differs from the the ordinary perimeter strategy, since the previous strategy counts edges instead of cells, so it might count certain cells multiple times. In practice, the edge counting strategy worked better, scoring around 30 points more. The reason for this is described in the image.

As a summary, in the submitted version, we use the diagonal global strategy and the perimeter local strategy. When comparing different choices, we use the perimeter as the first key, $(i + j)$ as the second key, and $(i - j)$ as the third key.

3.2.5 number of components

Besides the submitted version, we have tried another feature to choose the best location. In the previous section, we mentioned the number of cells that neighbor to already occupied cell. It generally makes the buildings compact, but sometimes creates small cut off areas that lower the land use efficiency. so we consider another measurement called number of components. It is defined as the number of connected components comprising vacant cells that are created when a building is placed. The idea is to create as few vacant components as possible. Figure 2 shows the idea more intuitively.

In our experiments, we tried using the number of components as either the first or second key, but the score decreased slightly, while slowing down the simulation. So finally we removed this feature.

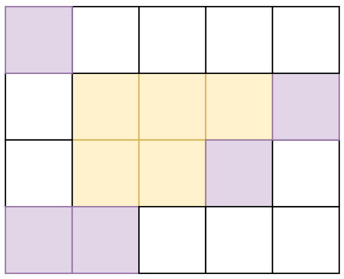

Figure 2: Suppose the yellow grids denote the candidate building place, the purple grids denote already occupied cells, the white grids denote vacant cells. Then the number of components in this example is 3.

3.3 Road Construction

This module is tend to find the best road building method. The strategies we have tried include global ones and local ones, which will be explained in each subsubsection.

3.3.1 Shortest path approach

The most simple but also efficient way to build road is choose the shortest one from existing road an edges to the picked best build place. The implementation of shortest path is simple. it also observe the idea to build road as less as possible, as the road cells make no profits. The shortest path cause closed vacant areas sometimes, but in most of the cases, the closed area will be largely filled by later coming buildings. We also have tried to build the roads cling to existing buildings, but it got lower scores.

3.3.2 Road pre-building

Because the online build road algorithm can disturb the building order, we also have tried several pre-build approaches. Before the request coming, we pre-build some road in a tree-like(fractal) fashion. The idea behind is to let the road touch larger areas efficiently. But this approach also lower the score gained. After several experiments, we got the sense that pre-build is not a good idea in this game, as it add more constraints to the later planning. Figure 3 demonstrate the tree-like pre-build approach.

3.3.3 close to the buildings

After dozens of experiments, we finally choose the simple but efficient shortest path strategy. But add an second key which is measure the perimeter of the road. The perimeter is defined in the same way with the building perimeter we mentioned in the last subsection. Then, our submitted version choose the road with maximum perimeter among those has the shortest length. This feature makes little difference in the performance, not considerably higher but also not considerably lower.

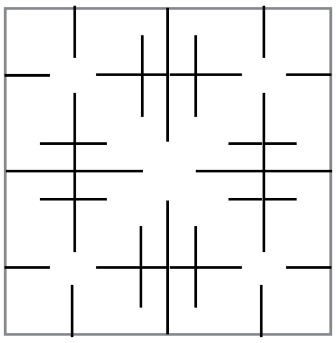

Figure 3: roads grow in fractal manner

3.4 Ponds and Fields

3.5 Adjusting Placement for Each Move

Our Jigsaw puzzle approach turns out to be a big success. We started to seek for an improvement over that.

We thought a possible "problem" of that approach is, we only consider the perimeter change for the residence or factory that we are building. It may be a good idea that, for every possible placement of buildings, we first construct the road, ponds and parks if necessary or applicable. Then we choose the best one by considering the perimeter change of all objects, and the number of disconnected components it creates.

Surprisingly this attempt failed. The average score of 50 runs over random sequencer decreases by almost 200. Also statistics shows a decrease on score per water/park cell. We began to realize that it is no good to pursue the regularity above some certain point. The reason behind may be, the roads, ponds and fields should be more "exposed" so that we don't have to build additional ones for future buildings. Another interesting fact is, if we only consider the perimeter change of buildings and roads, the average score almost gets back to the original one. We guess it's more crucial to expose the ponds and fields.

4 Adversarial Distribution

Our idea for the adversarial distribution is to test the "early space management".

At each request we generate a residence with probability 80% and a factory with probability 20%.

For the residence, with half probability it's going to be a star-shaped, otherwise it's just a straight line. The reason that we chose these two shaped is, we expected them to create a lot of empty "holes" on the map, as they couldn't fit each other nicely.

The factory part consists of two phases. During the first phase we generate 1x1 factories or 1x2 factories. Then at certain point (after 140 rounds in our submitted version), it switches to second phase which produce 5x5 factories only.

We expect that the player should use small factories to fill the empty "holes" that residences created instead of place them into some "empty areas". Otherwise during the second phase the player can hardly find enough empty areas for giant 5x5 factories.

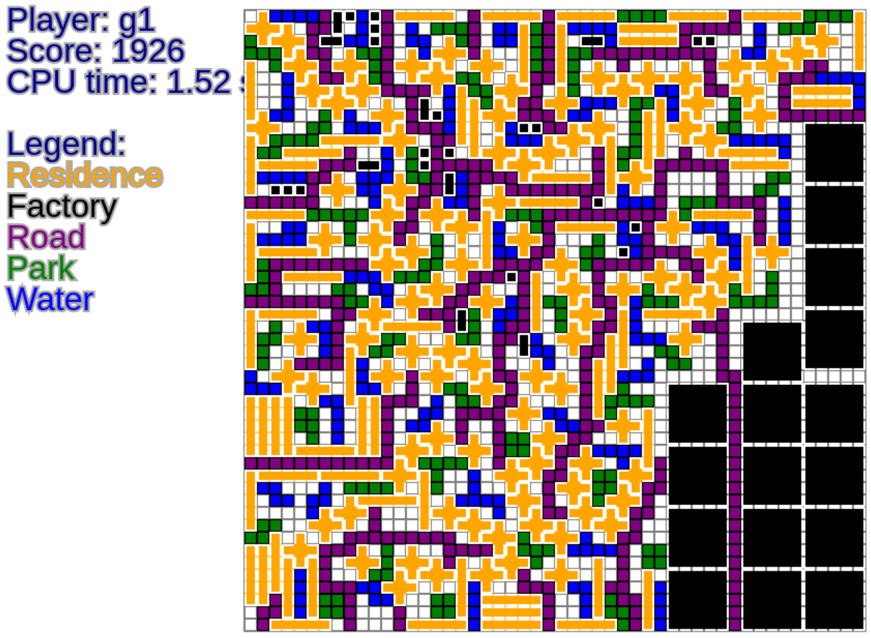

Figure 4: An 'ideal' solution for this adversarial: let small factories sneak into the residences area, which gives more freedom to the placement of large factories

5 Tournament Analysis