Project 1: Pentocity

Group 1: Chengyu Lin, Kailash Meiyappan, Ying Sheng

October 2016

Contents

1 Introduction

1.1 Problem Description

Pentocity at a high level is a packing problem. We are given an empty 50 x 50 grid of cells. The simulator sends requests to our program for a building to be built on the grid. The building distribution is completely unknown and may even be adversarial. The building is one of two types, a residence or a factory. The residences are in the shape of pentominos, while the factories are rectangles with side length upto five cells. Each building cell generates one point. Placing a residence near an existing park or pond of size four or higher generates two additional points for the residence, but only once per residence per pond/park. The buildings should also be connected to the edges with roads. We return the move, which is the location that the building is placed at, an optional set of cells for parks and ponds, and an optional set of cells, the new roads to be built. If the returned move does not contain a valid position for the building, this move is skipped. The game ends when three moves are skipped.

1.2 Our High-level Approach

Our approach looks at all possible valid locations to place the requested building at. Of these locations, the ones with the highest perimeter length are chosen. The algorithm then picks the location with the lowest (highest for factories) sum of row and column index to break ties. If ties still exist, it picks the one closest to the main diagonal. The idea behind it was to improve the "regularity" of everything that has been placed.
The algorithm then runs a breadth first search from the building to find the shortest path to a road. If there are multiple shortest paths, the one with the highest perimeter is chosen.
If the building is a residence, the algorithm looks for the nearest park and pond within a distance of three cells. If there is one, it extends the park or pond to touch the residence. If there is not park or pond within 3 cells distance from it, the algorithm looks at all vertical and horizontal long parks and ponds and picks the one with the least perimeter. The algorithm then generates 200 random walks and checks if any of these have a lower perimeter. The park and pond with the lowest possible perimeter is chosen finally. The resulting location of the building, set of road cells, park cells and pond cells are returned by the function.

1.3 Report Outline

This report is divided into many sections. This section is the introduction.
The next section describes how we determine whether a certain strategy is better than another. We describe metrics to determine how well an idea performs by looking at the final score, park and ponds and road cells.
The section after that provides a detailed analysis of our algorithm and the different ideas we thought of and why they worked or why they were abandoned. It describes ideas for building placement, road construction, and pond and field generation.
The fourth section describes our distribution for the input buildings and why we chose it.
The last section is an analysis of our approach in this tournament.

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

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.

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".

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".

Utility of Water/Park Cells The sum of the total score that the ponds generates and the total score that the fields generate divided by the total number of water/park 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.

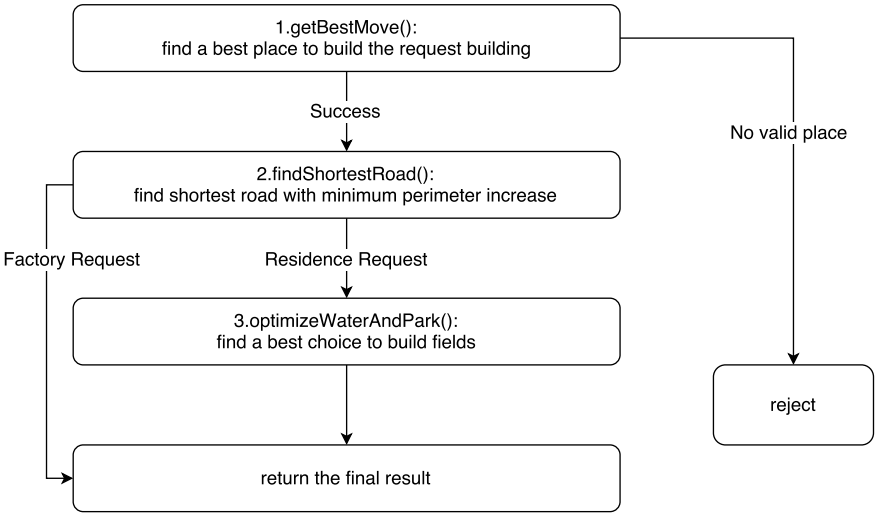

Figure 1: high level algorithm flow chart of our submitted approach

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 Pre-allocation

Due to the constraint that residences and factories cannot be adjacent to each other. It's reasonable to keep them away from each other. One of the simplest approach to this idea is, we "pre-allocate" the residence area and factory area. We give different priorities to the location where the building should be built at for the incoming residence and factory. It doesn't mean that some locations are forbidden for residence/factory construction. Below is a list of strategies we have tried.

Row-by-row construction 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.

Building along the 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, using this strategy we can better utilize the four borders but in row-by-row construction two of them are quickly occupied in the beginning.

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.

Residences at the center, factories to the borders 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 like what we are doing in real worlds.

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.

There are two reasons that we decided to avoid this strategy. The first problem with this is if we build all the factories near the edges, it makes the gap between factories and residences longer. Also there is another problem in that, it is vulnerable to adversarial distributions as it increases the difficulty of building roads. If we get an adversarial distribution that requests factories first, this strategy would build factories around the borders and cut off the inner area from outside.

3.2.2 Jigsaw Puzzle approach

The main idea of our submitted player is that, after placing the requested building, the remain empty region should be "regular". A square-shaped empty region is more favored since it

Figure 2: If only the number of cells were counted, placing the U pentomino on the plus pentomino would have a score of 3, same as placing it against an edge. Counting edges gives a score of 5 versus the same 3 for an edge

has more freedom for the future placement. We don't like the border of empty region to be irregular because it may require some special shape to well fit into it. Another bad example is a long-bar-shaped empty region, it restricts the alignment of the building.

Based on this idea, we propose a robust measurement of "regularity" – the **perimeter** of all the objects that have been placed. 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 this 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 figure 2.

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.

3.2.3 The number of disconnected components

Based on the "regularity" ideas before, we have tried another criterion 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.

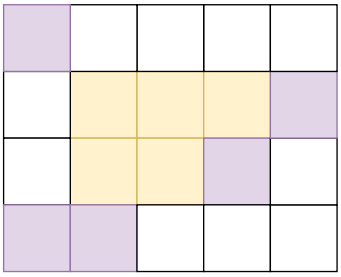

Figure 3: Suppose the yellow grids denote the candidate building place, the purple grids denote already occupied cells, the white grids denote vacant cells. Then the number of components in this example is 3.

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 3 shows the idea more intuitively.

Sadly in our experiments, we tried using the number of components as either the first or second key, but the score decreased slightly, while significantly slowing down the simulation.

3.3 Road Construction

This module describes the different road construction algorithms that we tried. As described in the high-level overview of our approach, the best building location is chosen using the getBestMove function and the road is chosen using the findShortestRoad function after the building is chosen. The techniques used in findShortestRoad will be explained in each section.

3.3.1 Shortest path approach

The most simple but also efficient way to build roads is choose the shortest road that connects the building to existing roads or borders. The implementation of shortest path is simple. It uses a standard breadth-first search from the building cells to reach the existing roads and borders.

This idea also observes the goal to build as few roads as possible, as the road cells generate no score on their own. The shortest path can cause closed vacant areas sometimes, but in most cases, the closed area will be largely filled by future buildings. We also have tried to build roads that cling to existing buildings but this leads to a larger number of unnecessary roads being built. This outweighs the benefits of less white space. This has been confirmed by lower scores in experiments.

3.3.2 Road pre-building

The online road-building algorithm can cause problems if large parts on the border gets covered. This leads to fewer options for new roads being built from the border. To counter this, we also have tried several pre-building approaches. Before the request comes, we pre-build some roads in a tree-like (fractal) fashion. The idea behind this is to let the road reach larger areas efficiently. But this approach got a lower score than the simple shortest path algorithm. After several experiments, we got the sense that pre-building is not a good idea in this game, as it is very dependent on the distribution and adds more constraints to the later planning. Figure 4 demonstrate the tree-like pre-build approach.

3.3.3 Road alongside the buildings

After dozens of experiments, we finally chose the simple but efficient shortest path strategy. However there may be multiple paths of the same length and we do not arbitrarily pick one like

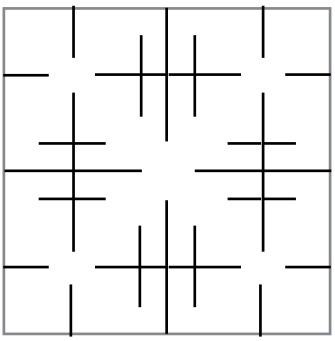

Figure 4: roads grow in fractal manner

in the previous version. We optimize the simple shortest path strategy by adding a second key which is to measure the perimeter of the road. The perimeter is defined in the same way as the building perimeter we mentioned in the building placement section. Then, our submitted version chooses the road with maximum perimeter among those that have the shortest length. This feature makes a slight difference in the performance, not considerably higher. It does come with the cost of lowering the speed of the algorithm.

3.4 Ponds and Fields

In this section, we describe the different techniques to choose ponds and fields and discuss their utility. We treat ponds and parks as the same (with respect to placement near a building) in our algorithm and analysis, since they generate the same increase in score. In every algorithm, we first check if the residence is already connected to an existing park or pond. If it is, we skip this step entirely.

3.4.1 Utility

We had doubts about whether ponds and parks were necessary at all in the beginning. In a previous section about analysing the runs, we added a metric, the utility of parks and ponds, that calculates the total points generated by ponds and parks, divided by the number of pond and park cells. This gives us the utility of a single pond or park cell. We discovered that in every single run, even a random walk generated pond or park had a utility of over 1.2. The increase in score between using no parks and ponds versus using parks and ponds was around 100 points, irrespective of the strategy we use, as long as there are a good number of residence buildings. The only other source of points is from buildings, which have a utility of only one point per cell, if you do not consider the roads that must be built to reach them. We concluded that it is always better to build parks and ponds.

3.4.2 Number of Configurations

First let us find a worst case of the number of possible ways to place a park or pond adjacent to a residence. The park or pond is at most four cells large. We found that using larger parks and ponds did not make sense, since we only need four to generate the score, and an extension to it could always be built later. These four cells are in the shape of a tetromino. There are a total of seven possible one sided (counting mirror reflections separately) tetrominos, each having a maximum perimeter of 10. The residence is a pentomino having a perimeter of size upto 12. The total number of ways to place the tetromino adjacent to the pentomino is 840. This is too large to run for each candidate building so we concluded that we had to run the pond and

park generation algorithm after choosing a candidate location, rather than including it in our algorithm to choose the best move.

3.4.3 Evaluating a Park or Pond

We developed a metric to choose between two different park candidates. At first, we thought that like buildings, choosing the candidate with the largest perimeter would increase packing efficiency. However, the random walk discussion below showed that this metric actually decreased the score. We instead went with choosing the path that had the least perimeter instead, since this led to a larger score than maximising the perimeter. The reason for this is that placing a park or pond near buildings does not generate any points for them, since they have already been placed, thus wasting a large part of their perimeter.

3.4.4 Random Walks

The first approach to build parks and ponds was to simply generate a random walk of size four. However, this is clearly not a good strategy. We decided to instead generate a large number of random walks and choose the best one from them. The metrics we use to compare candidates for parks and ponds is given above. We tried generating 50, 100, 200 and 400 random walks. When we tried comparing them using the largest perimeter, the score actually decreased as we increased the number of candidates, leading us to believe that this metric was the opposite of what we needed. So we reversed it and chose the park or pond with the least perimeter. For 50, 100, 200 and 400 random walks, our score increased, with diminishing increases each time as expected. At 400 walks, our pond function started slowing down the program, while improving our score by only around five points (which is so low, it could even be experimental error), so we decided to stop at 200.

3.4.5 Connecting to Previous Parks and Ponds

We already checked if the building was adjacent to a park or pond prior to constructing a new one. We added this feature to check if there was a park or pond within a distance of three cells away, so that we could instead extend it rather than build a new one. This is clearly better than building a new park or pond, because it uses fewer cells and generates the same score increase. We always choose to extend a previous park or pond if it is a possibility.

3.4.6 Straight Parks and Ponds

We hypothesised that since long flat parks and ponds have a large perimeter, they would generate the highest score. Since we use a random walk, some of these might get missed, so we added a feature to include all possible configurations of straight parks and ponds for consideration by our metric, apart from the 200 randomly generated parks and ponds. This improved our score, but only slightly.

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.

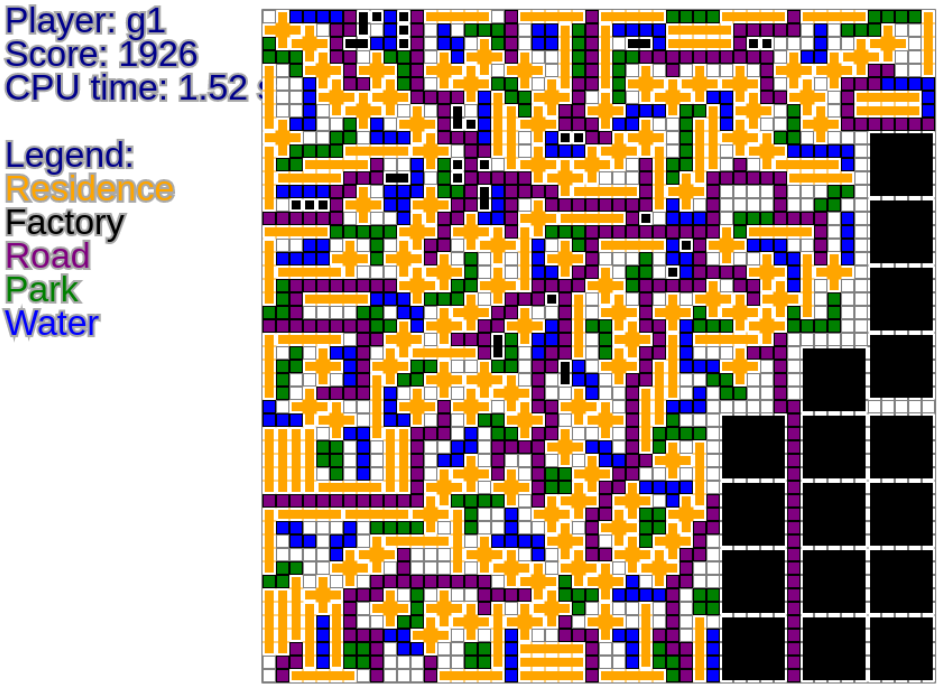

Figure 5: An 'ideal' solution for this adversarial: let small factories sneak into the residences area, which gives more freedom to the placement of large factories

5 Tournament Analysis

5.1 Our expectation

One principle of our approach is, make no assumption on the distribution. All strategies that we use aim to maximize the freedom of future building placement. So we expect that our player would perform consistently well against every possible adversarial distribution.

Because we didn't do any optimization to some specific distributions. It's performance may not be so good if the underline distribution has some simple "optimal solution". But we tends to let the problem itself to decide which placement is "better", instead of giving too specific instructions in some situations.

5.2 Performance

The result of the tournament is close to what we have expected. Despite the fact that our player's performance is quite well: it has the second best winning rounds (34, the first wins 37 rounds), and also the second best average score (1980.55, we'll explain why it's not the highest later). We want to highlight the one property of our player, that is "robustness".

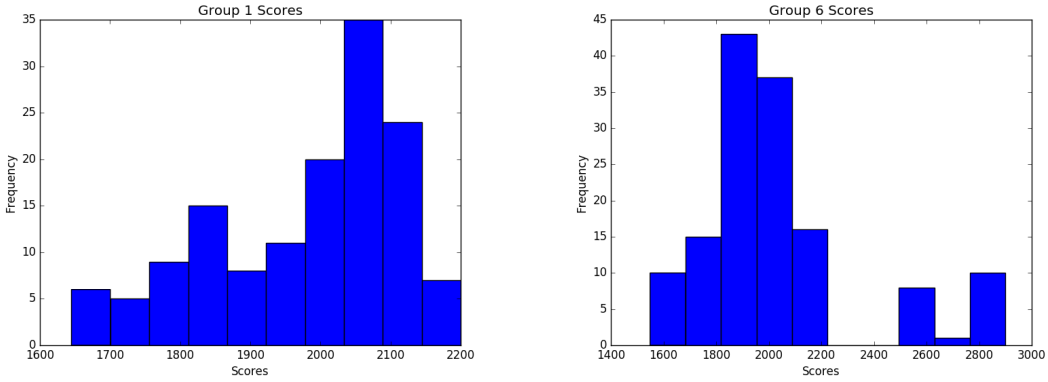

Figure 6: Score distribution of player 1 (left) and the highest average score player (right)

First figure 6 shows the score distribution of our player. Most of our scores are concenterated above 2000. Which makes our player has the highest median score as in figure 7. One interesting fact about the highest average score player is, its score distribution has some isolated "very high" part, where we believe is the key of getting the highest average score.

Figure 8 shows that, on almost every distribution, our performance is significantly higher than the average. It demonstrates the benefits if your strategy is robust and does not depend on the distribution. A more detailed table is listed in figure 1. If we average the scores for each distribution, our player wins 4 distributions and it's a tie with group 6. (While if you look into the distribution presented by group 6 and group 9, you'll find out that they are exactly the target distributions for group 6's strategy).

We also perform an 1-vs-1 match analysis over the tournament result. It's more like the rules of modern football leagues, we perform a round robin for all the players and see the comparison between every pairs of them. The result is shown in table 2. Clearly that our player, which we believe is the most robust one, wins every match of the tournament.

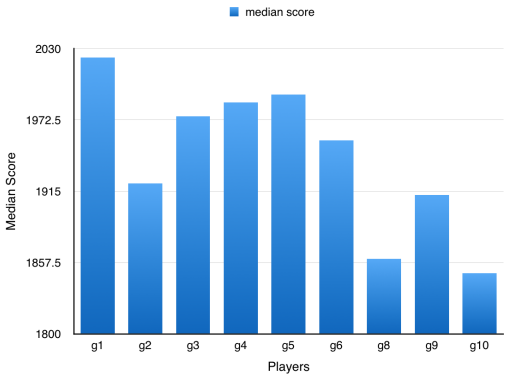

Figure 7: Median scores of each player

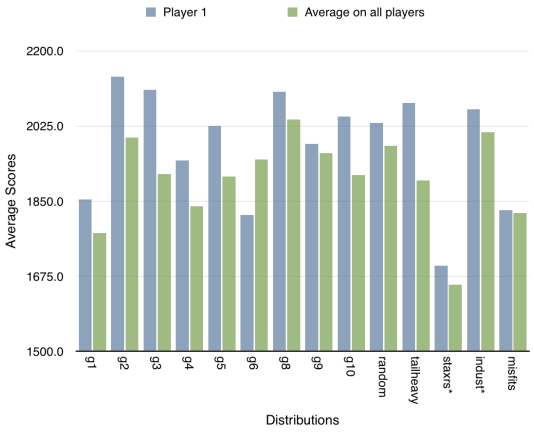

Figure 8: Averge scores of player 1 on each distribution

Sequencer	g1	g2	g3	g4	g5	g6	g8	g9	g10	average
g1	1853.6	1784.1	1776.5	1823.4	**1865.8**	1796	1692.5	1762.7	1621.8	1775.2
g2	**2140.2**	2005	2055.2	2043.2	1942	1948.2	1999.8	2022	1830	1998.4
g3	**2109.2**	1947.7	1410.5	2006.2	2094.4	1941.8	1722	2023.7	1961.9	1913.0
g4	**1944.3**	1872.7	1714.7	1890	1940.9	1826.6	1781.4	1823.4	1748.7	1838.1
g5	2025.6	1900.3	1661.2	1963.2	**2120.2**	1905.6	1755.2	1957.2	1872.8	1906.8
g6	1817.6	1893.4	1494.1	2004.4	1943	**2900**	1830.7	1788	1849	1946.7
g8	2104.5	2030.6	2065.1	2018.5	2033.4	2046.3	**2140.9**	2022.9	1894.5	2039.6
g9	1983.6	1917.3	1713.1	1987.1	1952.3	**2532.3**	1962.5	1809.6	1801.6	1962.2
g10	**2046.9**	1905.7	1729.1	1923.7	2042.8	1863	1885.2	1889.4	1912	1910.9
random	2031.7	1983	2067.2	2046.3	2020.1	**2069.9**	1615.3	2029.8	1939.7	1978.1
tailheavy	2078.5	2010.1	**2119.9**	2083.9	2067	2097.3	615.7	2057.8	1950.3	1897.8
stars*	1699	1672.4	1681.2	1697	**1807.2**	1584.1	1656.1	1588.1	1513.2	1655.4
indust*	2064.4	2001	2089.4	2029	2054	**2093.2**	1811.6	2032.1	1915.4	2010.0
misfits	1828.6	1822.8	1830.1	**1899.8**	1778.9	1881.8	1816.2	1839.9	1703.1	1822.4

Table 1: Average scores of each player running on each distribution.
The **bold red** ones are the distribution winner.

Score Table

	g1	g2	g3	g4	g5	g6	g8	g9	g10	win-lose-draw
g1	--	117:23	89:51	86:54	76:64	87:53	112:28	116:24	133:7	8-0-0
g2	22:118	--	43:97	25:115	36:104	52:88	89:51	69:71	117:23	2-6-0
g3	50:90	97:43	--	71:69	69:71	74:66	99:41	100:40	125:15	6-2-0
g4	52:88	112:28	67:73	--	59:81	79:61	112:28	103:37	137:3	5-3-0
g5	63:77	104:36	70:70	81:59	--	73:67	101:39	104:36	138:2	6-1-1
g6	53:87	87:53	66:74	61:79	67:73	--	87:53	85:55	122:18	4-4-0
g8	27:113	50:90	41:99	28:112	39:101	51:89	--	58:82	97:43	1-7-0
g9	24:116	70:70	39:101	35:105	35:105	54:86	81:59	--	112:28	2-5-1
g10	7:133	22:118	15:125	3:137	2:138	18:122	43:97	28:112	--	0-8-0

Table 2: Score table of pairwise comparison

6 Acknowledgement

We would like to thank Professor Kenneth Ross for this fun course and the discussion classes. We would also like to thank Kevin Shi for making the Simulator.