G4: Pentocity

Project 1

COMS 4444 Programming and Problem Solving

2016 Fall

Lu Yang

Bruce Wu

Tian Zhou

2016/10/17

Problem Statement

You have bought a large tract of land, and are developing it for residential and industrial building. Traditional methods of land development would require you to subdivide the land into plots in advance, before seeing what kinds of plots people might want. In this project we'll try to see if we can do better by dynamically developing the tract of land in response to demand.

The land is modeled as a 50x50 grid of cells. A cell can contain one of several types of development, and several contiguous cells can be used to place larger structures. The main structural elements are:

- Residences. These contain 5 contiguous cells, and are one of the 18 possible pentomino shapes (reflective symmetry doesn't apply here).
- Factories. These contain up to 25 cells, and are always rectangular, with between 1 and 5 cells on each side.
- Roads. Road cells must be adjacent to the perimeter of the development, or must be connected via a sequence of orthogonally adjacent road cells to the perimeter.
- Water. A contiguous group of four or more water cells is considered a pond.
- Park. A contiguous group of four or more park cells is considered a field.

At the start of the simulation, the development is a blank slate. A sequence of requests to build residences or factories (i.e., buildings) of particular shapes arrives one at a time -- you don't see ahead to future requests. Most of the time you will agree to build the building, and place it on the development somewhere. At all points in time, the development must satisfy the following constraints:

- Every building must have at least one cell orthogonally adjacent to a road. This may mean that to place a building you may additionally need to place a number of additional road cells to meet this constraint. (Remember that road cells must connect with the perimeter.)
- The footprint of the building must be previously empty. You are permitted to rotate the building in order to place it.
- Factories and residences may not be immediately adjacent to one another. But factories can be adjacent to other factories, and residences to other residences.
- A residence that is placed adjacent to a field achieves a bonus (see below) because the purchaser can see that the value of the property is higher. The bonus is achieved only if the field is present at (or before) the time the residence is placed. Adding a field later does not achieve a bonus for existing adjacent residences.

- A residence that is placed adjacent to a pond achieves a bonus in the same way as a field (see below).

You score one point for each cell of a building that is placed. So residences always score 5 points, and factories score a variable number of points. Fields and ponds that are adjacent to a newly placed residence each yield a 2 point bonus for that residence. A residence that is adjacent to two fields (or two ponds) gets just one bonus, while a residence adjacent to a field and a pond gets two bonuses.

You have the option to reject a building request, but after three rejections, the simulation ends. Even if you do a good job of building placement, eventually you will run out of room and be forced to reject requests. Your goal is to have as many points as possible when the simulation ends.

The distribution of shapes in the request sequence is not necessarily uniform. In fact, it may be deliberately non-uniform, to test the robustness of your strategy. We'll provide a variety of sequence generators, and allow you to specify your own for experimentation.

For the tournament at the end of the project, we'll use a variety of generators, some of which you will have seen, and some of which will be new.

Things to think about:

- What kinds of placement give you the most flexibility for later buildings?
- You get to choose the shapes of roads, fields and ponds. What are good shapes?
- What does it take to achieve a “robust” strategy?

Motivation

The problem is an abstract of the dynamic planning problem in the real life. The core question here is “What should we do now with an undetermined future?”

Here are some intuitions that we are thinking about:

- Should we look carefully at all possible ways of future and find some common values among them and design the basic strategy?
- Should we use greedy method without handling the future?
- Is there a trade-off between local optimization and global optimization? How can we take care of that?
- Should our strategy be efficient in just some of the “most possible future” or it should be robust in all possible future including some eccentric ones?
- Should we try to predict the future in some degree in order to get more flexible strategy?

All of our explorations below are the experiments trying to answer the above questions.

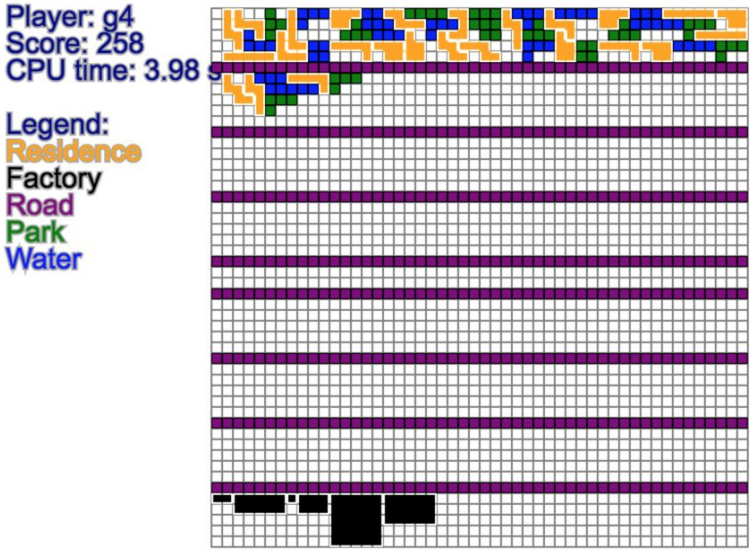
Strategy and Implement: Evolution

1 Pre-allocated road and Left-remaining cells

Intuitions:

Our first attempt to solve the problem is pre-allocate spaces between roads. The algorithm works as follow:

At the beginning of the game, we built some number of straight roads from left to right, with equal amount of empty spaces in between. These roads have length 50 so they seperate the entire board into different parts completely. After that when a request came, we look from left to right to find a placement of the building that would minimize all empty spaces in the area left to the rightmost current placement.



Evaluations:

Pros:

We thought there were two advantages of doing this:

- One is to seperate residence spaces and factory spaces. By allocating sets spaces with 50 as width and arbitrary number as height for residences and factories, and build straight roads in between them, we can seperate residences and factories without wasting the space in between them comparing to other non-preallocating approaches.

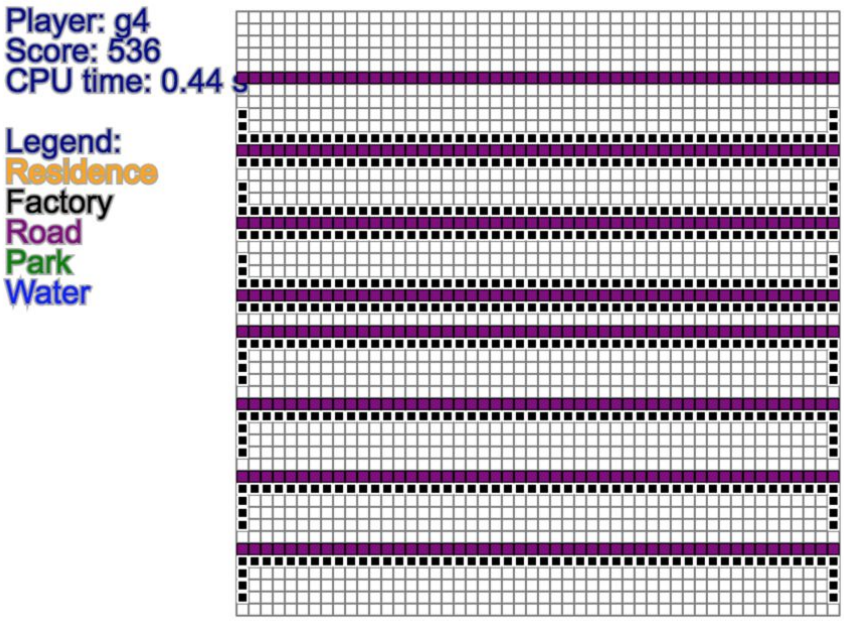
- The second advantage is to utilize spaces more efficiently with respect to roads. If we build residences and factories using the shortest road algorithm, many times the roads ended up being twisted, so it's harder to fit the latter buildings nicely. And since the placement would minimize the empty spaces left in areas to the left of all current placement, it would ideally achieve a good fit that packs buildings together as tight as possible.

cons:

we abandoned this approach after second week. The major issue is that this strategy is not robust against adversary distributions. We build the roads straight at the expense of being forced to place the buildings on top and buttom of the preallocated spaces.

If the distribution is arranged in a way that would cause us to leave large empty areas in between the top and buttom bulidings, our score could drop below accetable levels. For example, an adversarial distribution could give us all factories of size 1x1, and there will be 3 unutilized cells for every 2 cells we place.

We thought about changing the height parameter to somewhat alleviate this problem, but the same problem exists regardless of height being 10 or 7. We also thought about building new roads from one side to the other as the buildings come along, but then we agreed it would be better off if we just use the shortest road algorithm and abandon this approach.
(Below is a screenshot of such an adversarial distribution.)



2 Find shortest water, i+j and Search space

Intuitions:

After first unsuccessful experiment, we realized the importance of the robustness. So what we want to do is to find a basic strategy firstly and than add more heuristic into it.

We choose the i+j logic (build residents from up-left corner and factories from the down-right corner) because compared to the i logic (up and bottom), it needs less road cells.

And we also want to improve the water built strategy. When we need to build a water, we want that it can be firstly find out if we can extend the water nearby by less than 4 cells so that we can reach the same efficient with less water cells.

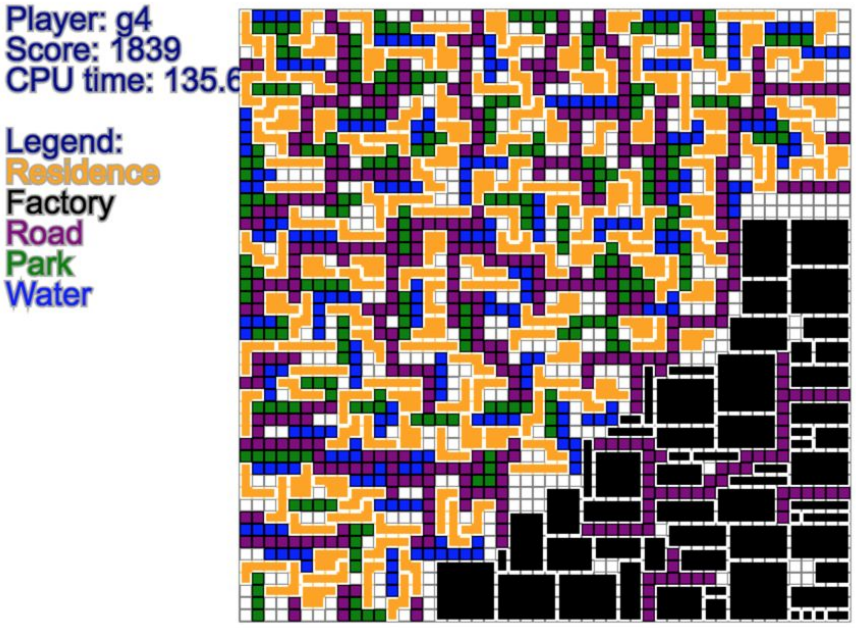
Strategies:

- i+j logic for residents and factories: When we want to place resident, we search all the map to get all the possibility that it can be built. Then we calculate them by add all the cells' i, j together for each possible position. And we choose the position with the smallest sum as our move. Similarly the factories are put in the biggest sum position.
- Roads: using find-shortest-road.
- Waters and Parks: Try to find waters and parks that are less in 4 cells away. If we find nearby water, we just create a water link to that. Otherwise we create a 4-cell water and a 4-cell park.

Implements:

- i+j logic for residents and factories: Use a sum function to calculate the sum of i+j of all the cells of a possible position.
- Waters and Parks: We design a new BFS function which is used for waters parks and roads. For waters and parks, we track the length of BFS, if we find that the length is small that 4 and it link to another water/park, it is done. Else we built the result of 4 step BFS.
- Roads: Use the function above to find shortest road.

Results:



Evaluations:

Pros:

- By using the link-to-nearby-water logic, we made water cells in higher score density.
- Well packed.
- Small amount of road cells.

Cons:

- Because of the BFS algorithm, all of the 4-cell water and park we built is just in the shape of straight line which is the result of BFS. This can potentially make packing looser. We tried to redesign the BFS but found that it is impossible to use BFS or DFS to create all the possible water and park in limited time.
- We want to create a search space that we can search all the possible combination of building, road, water and park. But the result is that the space became so big we can't finish the game in limited time. We tried many ways to limit the search like only searching the position near the current element rather search the whole map. But it still exceed the time limit.

3 Largest perimeters (Credit goes to g1)

Intuitions:

By looking at adjacent points of buildings, we know how packed is this placement.

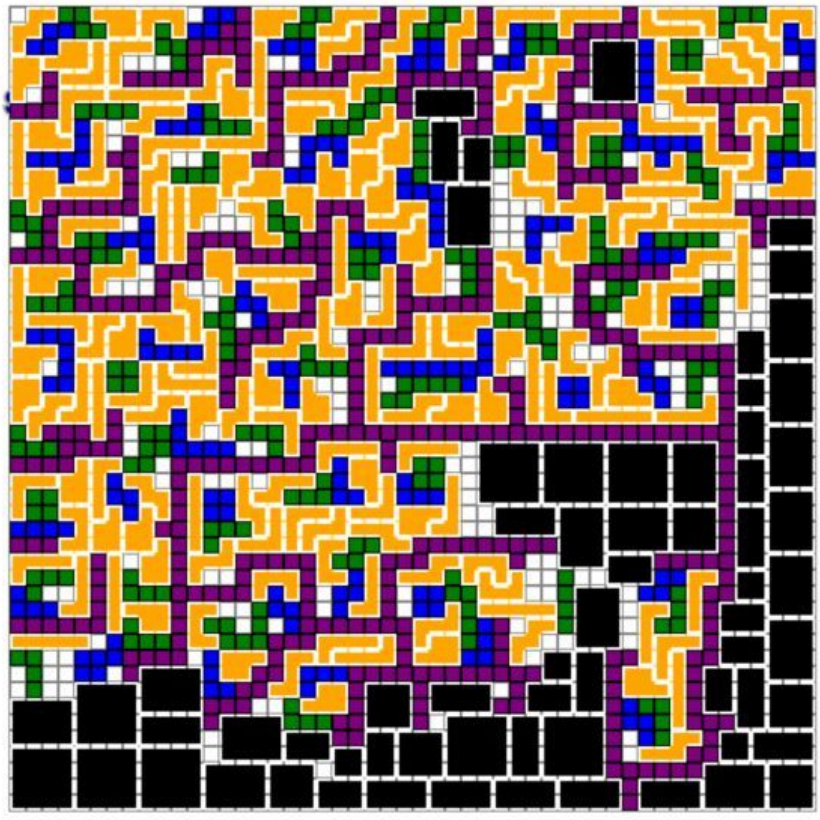
Strategies:

By counting unoccupied cell around building placement, and use the one have smallest unoccupied one.

Implements:

Basically we reused g1 deliverable on 9/26 and fix their bug (where they count all the building cell's neighbor instead of only outer building cell)

Results:



Evaluations:

Pros:

- Board is pretty well-packed and score has a huge increase (from 1800+ to 2000+ for random sequencer)
- Robust to different kinds of sequencers, and not easy to find an adversary one.

Cons:

- Simply use occupied vs unoccupied casues many problems:
    - Factory goes into residence area and vise versa.
    - Resources (road, water and park) is not well shared
- Hard to combine it with other strategies like our existed strategies i+j

4 Sharing vs. Packing

Intuitions:

Previously we use a lot of methods try to pack our placement (like i+j logic, largest parameter), however we actually don't know whether packing is a good idea or not. Because for the resource like road, water and park, it may be better to for residence to take less as you can leave residence coming after you access to the resource. We want to investigate more about sharing strategies.

Strategies:

- For water/park, do not place it to sideline because it is not good for sharing(also because sideline is great place for residence/factory because no road needed)
- For placement of residence/factory, introduce new parameter first_*, which mean only first cell important. (For example, add 1 for perimeter for the first road but do not add more for later road in perimeter)

Implements & result:

See next section for detail

Evaluations:

Pros:

- More road, water/park can be shared

- It gives us a big picture way to think about problem instead of greedily optimize local placement

Cons:

- Not well packed

Overall:

- Obviously, there is a tradeoff between sharing and packing in our model, thus how to find this balance is very important and we solve it by using framework mentioned below

5 Framework to combine strategies

Intuitions:

The rigid if-else logic to combine strategies will give priority to each strategy which will in turn restrict the number input option of each priority. Therefore, our group converts all the strategies mentioned about into score and add them up.

Strategies:

For each play, we get all possible moves (as large as possible within computational limit). For each possible move, we use scoring function to give each of them a score according to all the strategies mentioned above and finally choose the one with the highest score.

Implements:

- For (i+j) logic, we use (i+j)/divider1 for factory and (100-i-j)/divider2 for residence to add it to scoring method.
- For perimeter, we use 3*10 matrix to represent our scoring strategy. The parameters showing as follows:

	empty	residence	factory	W/P	side	road	first_road	first_factory/resident	i,j divider	first_water/park
Factory	x1	N/A	x3	x4	x5	x6	x7	x8	x9	N/A

Resident	y1	y2	N/A	y4	y5	y6	y7	y8	y9	y10
Water/ Park	z1	z2	z3	z4	z5	z6	N/A	z8	z9	z10

This table means for placement of a factory, every adjacent empty cell contributed to x1 points to the score of this placement, and so on and so forth.

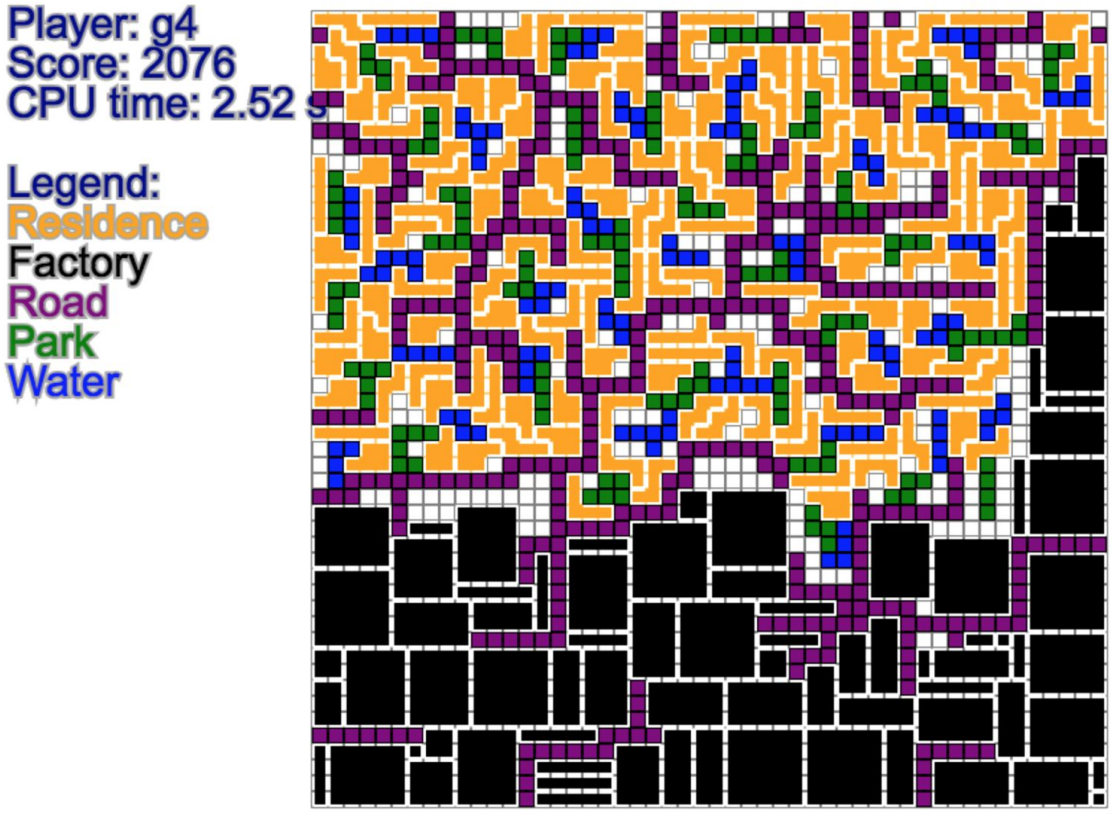
Note that first_* here means only the very first adjacent * cell will contribute corresponding points to score. I add these columns because it potentially benefits to our resource sharing (like road, w/p)

After we insert these parameters to our strategy, we also write a script to train our parameter. Due to the limitation of computing power, we start with a heuristic group of parameters which scores 2000 for random distribution. And each time we try to tune each parameter by range +-3. For each specific parameter we run 3 times to cancel out randomness. Finally, we trained each parameter one by one and do it multiple times to leverage potential dependency between parameters.

Results:

Final parameter matrix is:

	empty	residence	factory	W/P	side	road	first_road	first_factory/resident	i,j divider	first_water/park
Factory	0	0	1	0	1	1	1	3	30	N/A
Resident	0	4	0	4	2	3	0	7	50	5
Water/ Park	-3	0	3	-1	-1	-1	N/A	N/A	N/A	N/A



Evaluations:

Pros:

- Very flexible in term of logic, and very easy to add new strategy into current framework. As later mentioned in future work, we should add blob detection, and heavily penalize over blob size. With a blob detection function, revising code should be very easy and clean.
- Easy to leverage importance of each strategy by modifying parameter

Cons:

- Not easy to clearly explain the strategies because they are highly mingled with the parameters associated with it.
- With limit computation power, it is easy to overfit each parameter.

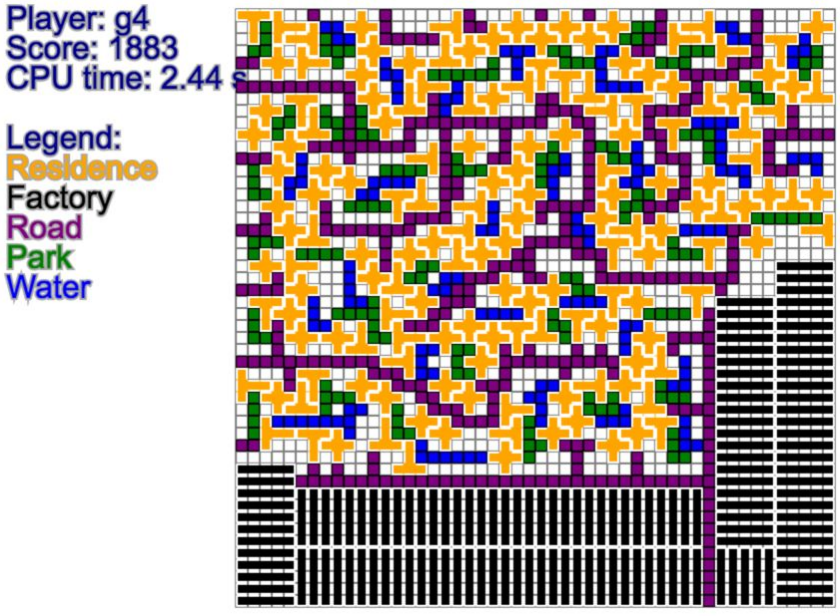
Tournament Performance Analysis

Our placement strategy has combined the characteristics from a few commonly known strategies, including best-fit criteria counting the placement's unoccupied neighbor, increase the exposed area of parks and ponds so they have more potential future values by residences, and find the shortest roads to connect to for current placement.

According to our experiments with different distributions, the usage of taking into account many different parameters sort of averaged out the peaks and buttoms of our score. Which is why although our team did not end up in first place for number of games won or highest median scores, **our team was ranked No.1 overall in the misfits distribution, had the lowest number of games under a score of 1800 (12 times, compare to 28 times on average), and become the second-best team in top 3 ranks (76 times out of 140 games) and buttom 3 ranks (27 times out of 140 games)**.

	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	top 3	bottom 3	less than 1800
g1	34	37	22	16	8	10	10	3	3	93	16	18
g2	0	3	11	21	33	24	24	21	5	14	50	19
g3	14	30	27	18	16	10	10	4	12	71	26	32
g4	9	27	40	25	10	13	13	7	0	76	20	12
g5	37	25	9	15	15	12	12	11	1	71	24	15
g6	34	17	5	13	13	12	12	20	7	56	39	19
g8	11	2	6	11	18	24	24	21	28	19	73	53
g9	1	3	19	20	18	25	25	25	8	23	58	42
g10	0	0	0	4	8	9	9	30	74	0	113	43

Below, a screenshot of our team running the misfits distribution



A particularly interesting case is our code performs poorly (second-to-last) on g8's distirbution. Although there were nothing too speical with the distribution itself (it had 105 random residences over all possible residence shapes, then gives random factories to the end), the fact that our training data set has never seen a distribution over all possible residence shapes could explain its poor performance. It could also be due to the fact that we are only running the distribution 10 times and the difference between teams are not too significant (7 teams average around 2020 points, with our average score being 2018).

Another case where we didn't do quite (ranked 5th) as well was the industrialization distribution. Although our parameters have some encouragement for factories to be placed next to each other and therefore increase the overall fit between factories, we scarficed some level of fitness for residences our better future utilization of parks and ponds. Therefore when faced with larger factories in later stage of the game, our algorithm struggles to utlize the remaining empty spaces on the board.

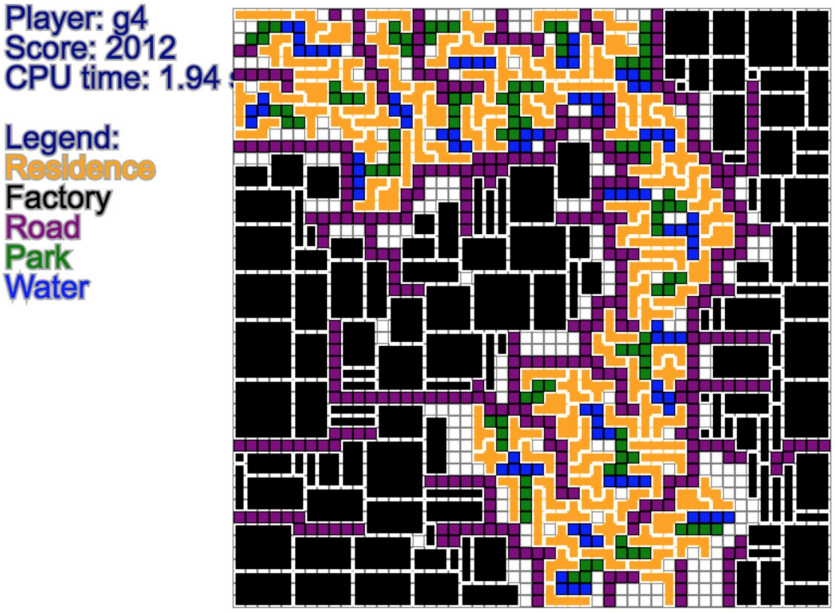
(Below, our code running the industrialization distribution, one can see there are large utilized free spaces near the larger factory placements in late game)

Other than these two cases, and two other 4th place result, in all distribution our scores ranks in overall top 3. We have some confidence in saying that our method is robust against different types of distributions.

Potential Improvements

1 i logic

Though we have many parameters that handle different situation, we can see that most of them is related to the “local optimization”. We tried to evaluate the best local environment every time we are about to build an element. So when look at our results, it is always well packed every single small area. But when we look at the “big picture”, some flaws begin to rise. Like this distribution:



We can see that the contact surface between residents and factories are too big and the space for factories are too “sparse” so that large factories can’t be built in the end. This shows us that though in local optimization we did a good job but we are still limited by the global optimization like “how to divide the area for factories and residents”.

The easiest way to do that is to add the i logic, the most basic logic and earliest adopted logic which put residents in smallest i (top) and factories in largest i (bottom) so that we can get small contact surface and well separation. In that way, factories will have integrated space to handle large square (5x5).

Actually we have done it.
For i logic, we use i/divider1 for factory and (50-i)/divider2 for residence to add it to scoring method.The divider is a parameter which reflect the weight of this logic.

But we met a problem.

2 blob detection

Blob is an empty space that can't be used because it is blocked by elements and can't reach the road. We found that after we add the i logic, there will be a lot of blobs appears so the result is not as well as we image.

Why? We analyzed and guessed the reason. It might because of the conflict of the longest perimeter logic and the i logic. Imagine that when the longest perimeter logic dominates, resident will be more fit to the diagonal direction due to more contact with previous elements. But when the i logic dominates, resident will grow to the up direction. When this to direction happed in turn, there is a high possibility that a blob created. Especially noticing that the diagonal direction is not only for down-right, it can also be up-left. So what we want to do is to use blob detection to avoid such situation.

The basic idea is that every time we want to place an element, we search it's unoccupied neighbor. If there is a neighbor that can't find road to it, we use BFS to figure out what the size of this blob. If the size is bigger than 4, we cancel the placement and find next placement.

We have confident that if there is even a little more time for us to debug the blob detection which is not too hard, we can have a giant improvement. Because we found that this is a bottleneck for both the game and our strategy.

3 parameter training

We haven't finish our training. It should be the most important part for our strategy but we still have very limited time to train it because of the urgent deadline.

What we did this time is that we only trained the parameters for the default distribution. And because we don't have enough to try all the combinations of parameter, we can only train them on by one and for several turns.

We also want to train the parameter more with different distributions to see if the parameters will converge or not. If it will not converge, we may choose a compromised set of parameter.

4 detector

The idea of detector is not only from those questions in the beginning of this report, but also a realistic way to solve the parameter training problem. The basic idea is that if the parameters can't converge in different distributions we create several sets of parameter. Every set of parameter is train by a kind of distribution. And we can use a detector to implement the switch in these sets of parameter.

The basic idea of detector is dividing distribution roughly in to several classes and give every class a pattern. When we find the request is of certain pattern, then we change to that parameter set. For example, if we detected that the most recent 3 residents are all crosses. Than we change to the parameter set which is trained with all cross residents. The disadvantage is that while increasing the robustness in some situation, this idea may perform badly in other situation like 3 crossed 3 lines in turn.

Of cause we can use advanced technique to improve it. We can quantify the patterns or features in the distribution, and use them as several parameter to add in the object function. Then use online learning algorithm like stochastic gradient descent to implement dynamically prediction.

We do not have time to implement it. So I can't say for sure whether it will be good or not.Because in this tournament there are very limited patterns, we might improve in some degree.

But the key question is still there.

Should our strategy be efficient in just some of the "most possible future" or it should be robust in all possible future including some eccentric ones?

5 future simulator

The secret of winning the game is that always make best use of all the resources.
On the second version of our project, we met the lack of computational resource problem. But in the final version, we only use some seconds to complete the game, which also suggests that we waste a lot time resource.

What we plan to do is to look forward several steps so that we can get a better trade-off between local and global.

One simple idea is to use the detector above as a simulation here. Every move we generate our own fake requests for n future steps using the detector and try every combination of building in these n steps and find the best one according to our object function. Then we use the first-step-part of the combination as the move in this request. The number n is determined by the computation power.

Conclusion

I want to go back to the five questions that we are thinking all the time during the project.

1. Should we look carefully at all possible ways of future and find some common values among them and design the basic strategy?
2. Should we use greedy method without handling the future?
3. Is there a trade-off between local optimization and global optimization? How can we take care of that?
4. Should our strategy be efficient in just some of the “most possible future” or it should be robust in all possible future including some eccentric ones?
5. Should we try to predict the future in some degree in order to get more flexible strategy?

And here are our views of these questions after the project:

1. Yes. A basic strategy can handle most of distribution. So we chose the largest perimeter as our basic strategy and done other improvement based on that.
2. No. The first try of our method proved that it may not be a good idea.
3. We’ve done some work to hit a balance of local optimization and global optimization. We haven’t finished that, but we think we are in the right way.
4. Hard to say. There are too many ways to evaluate the result of tournament. What wo did is to try our best to handle all the case.
5. We’ve get some ideas to test that. But we do not have time to implement that. We still think it will at least improve the strategy in some degree.

Actually there are a lot more questions that everyone in this game have met. For instance, when shall we place a water? Will it be enough valuable in the end or it will be a waste of space? Questions like that are too hard to use heuristic strategies to handle. It is because there are too many question that we can’t find directly answers that we come up to use the parameter-training-method to let data and experiments help us to figure out the inner logic behand this game.

Acknowledgements

Prof. Ross
The discussions led by Prof. Ross were very helpful in determining how we could improve our strategy.

Kevin Shi
Jiacheng was very helpful in implementing modifications to the simulator that continued to improve throughout the project.

Other Groups
Lastly, we'd like to thank the other groups. The discussion sections that we participated in have helped us to much better understand the problem and how to improve our own strategy.