# Sim City

Kevin Huang, Christine Hwang, Alex Kahng, Joseph Song

Documentation video: `https://www.youtube.com/watch?v=waUQ4_mCwb0`
Original/Revised Spec: See report folder

## 1 Overview

Consider the following problem: Given a town $G$ consisting of $n$ buildings and the distances between each pair of them, we want to construct some number of roads between the buildings that will minimize the average travel time between buildings, weighted by the importance of the buildings (so for example, the time it takes to travel between a fire department and a school may be weighted more than the time it takes to travel between a house and library). We call this value the "optimality" of a road construction, which we define more rigorously below. The road construction must satisfy two restrictions: (1) due to budget constraints, exactly $m$ roads can be constructed (for some $m$), and (2) to avoid too much congestion at any given building, at most $k$ roads can extend out of any building (i.e. a maximum degree restriction of $k$). For this entire project we set $k = 3$, but this value can be changed by altering the value of MAX_K in roads.h. A brute force search for the best road constructions of a given $(n.m)$ pair needs to check up to an order of $O(n^{n \cdot k}) = O(n^{3n})$ such constructions. We tried to design a heuristic that would find a relatively optimal construction, but would take considerably less time, on the order of $O(k \cdot n^{k+1}) = O(n^4)$, by recursively constructing road constructions for towns based on road constructions for smaller towns.

## 2 Calculation of "Optimality"

Given a road construction as a boolean matrix, we assign

$$traffic_{xy} = \begin{cases} 1 + \dfrac{I_x}{d(x)} + \dfrac{I_y}{d(y)} & : roads_{xy} = true \\ \infty & : roads_{xy} = false \end{cases}$$

where $d(x)$ represents the degree of building $x$ (i.e. the number of edges incident to $x$). Intuitively, a road has more traffic if the endpoint buildings are of higher importance, distributed across all the roads connected to these buildings. If no road exists, a car cannot pass through it, so we give it a traffic value of $\infty$. We can then scale this traffic factor by the distance between the buildings, to get a timing for how long it takes to travel between two buildings:

$$traffic\_dist_{xy} = \begin{cases} (1 + \dfrac{I_x}{d(x)} + \dfrac{I_y}{d(y)}) \cdot dist_{xy} & : roads_{xy} = true \\ \infty & : roads_{xy} = false \end{cases}$$

Then for any two buildings $x, y$ we define the travel time between them as

$$time_{xy} = \{\min\left(\sum_{e \in p} traffic\_dist_e\right) \mid p \text{ is a path from } x \text{ to } y\}$$

which we compute using the Floyd-Warshall algorithm. We will want to minimize the average value of this, weighted by the importance of the buildings (call this value "optimality" of a construction):

$$optimality = \sum_{x,y \in G} time_{x,y} \ln(I_x + I_y)$$

The reason we chose to minimize this instead of time weighted by $(I_x + I_y)$ is because we thought that would make the program focus almost entirely on the most important buildings and rather than the entire town.

# 3 Heuristic Design and Implementation

For each $(n, m)$ pair, consider associating an optimal construction on $n$ buildings with $m$ roads. Our heuristic recursively finds an $(n, m)$ construction by taking the optimal constructions associated with $(n-1, n-2), (n-1, n-1), \ldots, (n-1, m-1)$ and "extending" these constructions to include 1 more building. A construction is "extended" by adding in another building, and then considering every possible combination of extending new roads from this building to the original construction. The most optimal construction out of all of these extensions is recorded and associated with $(n, m)$. We call all the constructions associated with $\{(n, n-1), (n, n), \ldots, (n, \left\lfloor \frac{n \cdot k}{2} \right\rfloor)\}$ a platform of value $n$ (thus, a platform contains every value of $m$ possible for given $n$). In other words, our heuristic builds a platform of value $n$ by extending all the constructions in the platform of value $n-1$, and keeping the most optimal ones associated with each $m$ value. The heuristic first builds a platform of value 2, consisting of two buildings (the first two buildings in the town) connected to each other, and extends this single construction to a platform of value 3, then extends all constructions in this platform to obtain a platform of value 4, and so on until we arrive at a platform of value $n$.

Given a town (with given importances of buildings and distances between them), the heuristic will thus finish with the most optimal road constructions it has found associated with each possible $m$ value that allows the town to be connected but not violate the maximum degree restriction of $k$.

# 4 Comparison to Brute Force

For each $(n, m)$ pair, for small $n$, we also ran a brute force search to compare how the heuristic results compared to the most optimal possible constructions. Through the brute

force search, we kept track of the most optimal (i.e. lowest "optimality" value) construction, as well as the number of possible connected constructions and the sum of their optimalities, so we could calculate the average optimality of all possible connected constructions. We could then see where the optimality of heuristic constructions were in comparison to the optimalities of the most optimal construction and the average optimality found by the brute force. Ideally, the value of

$$\chi = \frac{optimality_{heuristic} - optimality_{best}}{optimality_{avg.construction} - optimality_{best}}$$

will be close to 0, meaning the heuristic returns a construction much closer to the optimal one in comparison to a random construction.

# 5 Instructions

Our program will mainly be run in the terminal window. Once it is compiled and executed, the terminal will ask for the number of "buildings" in the town. The user can either choose to manually type in values describing the importances of buildings (e.g. a fire department may be assigned a higher importance than a home) and the distance matrix describing the pairwise distances between buildings, or choose to let the program run a template town, which assigns an equal importance of 1 for every building and a distance of 1 between each pair of buildings. After this, the program will run the heuristic as described in section 3, starting from a platform consisting of the first two buildings. For each possible number of roads that can be placed, the most optimal constructions found by the heuristic are printed into a separate file, "Heuristic Optimal Road Constructions" as boolean matrices representing which buildings to place roads between.

After the heuristic is finished running, the user is then given the option of running the brute force algorithm to compare to the heuristic optimalities. For $n \leq 6$, the brute force algorithm will take 0 seconds, for $n = 7$ it will take 6 seconds, and for $n = 8$, it will take 553 seconds ($\sim$9 minutes). For $n \geq 9$, we do not suggest using the brute force algorithm since it could take more than several hours or days to run.

# 6 Original Plan/New Features

Our original plan for our project worked fairly well in our final project. This was largely attributed to the fact that we spent a majority of our time in the beginning to draft our outline, taking into consideration future obstacles before implementing any code. Because our project dealt largely with the theoretical nature of our heuristic, this proved to be one of the greatest challenges, and once we went through the thought process on the macro scale, deciding the signatures of the functions was easier. As we had planned, we were able to implement a heuristic which ran much faster than the brute force algorithm, in $O(k \cdot n^{k+1})$ time rather than $O(n^{n \cdot k})$ time. Instead of implementing priority queues as discussed in the original draft specification to increase the sample size of road

constructions to be extended (by associating multiple constructions with each $(n, m)$ pair), we decided to pursue a different optimization of the heuristic that we started to notice by looking at patterns of early heuristic outputs. We noticed that given a town, if we made the heuristic consider the two most important buildings as its base platform rather than any other two buildings as the base platform, the heuristic found much more optimal road constructions in the end. Following this discovery, a new feature we implemented was to permute the importance array and distance matrix given by the user so that the heuristic would build the platforms by adding in the next most important building at each iteration of extending platforms, and therefore optimize the resulting road constructions. This reindexing of the buildings would be stored, so that at the end before road constructions found by the heuristic were printed to a separate file, an inverse reindexing could be applied to show how roads should be constructed with respect to the original building indices.

In comparison to the original specification, we implemented features to make the interface more user-friendly: when inputting the distance matrix for a town, the program automatically fills in the entries below the diagonal to mirror the values typed in by the user, so the distance matrix is automatically symmetric. The program also gives the option of using a normalized town, instead of having the user manually input town importances and distances. Finally, the optimal road constructions found by the heuristic are printed into a separate file rather than the command terminal itself.

# 7 Heuristic Results

When executing our code, we found the heuristic ran significantly faster than a brute force algorithm search. TimeResults.jpg (in the reports folder) is a graph for the time it took to execute each code for a given number of buildings. The running time for the brute force algorithm grows extremely quickly, taking 0 seconds for $n = 6$, 6 seconds for $n = 7$, 9 minutes for $n = 8$, and at least a few hours for $n = 9$. On the other hand, the heuristic ran up to $n = 40$ in less than 1 minute. Despite being much faster, the heuristic still returned road constructions that had optimality values much closer to the best constructions found by brute force compared to the average optimality of connected constructions, when ran on template towns with building importances and pairwise distances all set to 1 (see TemplatesResults.xlsx in the reports folder). The average $\chi$ value (defined in section 4) for all $(n, m)$ pairs up to $n = 8$ (the maximum we could find results from the brute force search in a reasonable time) was only 0.360. This low $\chi$ value was even more exaggerated for towns with buildings assigned differing importance values (instead of template values of all 1). For example, the average $\chi$ value found for the town shown in the documentation video was .335, and after we implemented the reindexing optimization discussed in section 6, this $\chi$ value dropped further to a meager .067. In this sense, our heuristic was successful in drastically reducing the time of a brute force search while minimizing the value of $\chi$ obtained for a given $n$.

# 8 Reflections

Our planning was very spaced out and organized throughout the entire project, allowing us to have enough time to implement and meticulously test every function we wrote, as well as allow ample leeway time for unexpected bugs. While switching to C after a semester of OCaml proved to be quite difficult, it stretched our capacity to learn and forced us to consider programming with a broader perspective. We carefully planned out our structures types and modularized our functions, as well as planned for dozens of hours over pencil and paper the flowcharts and pseudocode for functions before implementing them, so they could fit together very smoothly by the end of the project.

A recurrent mistake we ran into was obtaining negative values, "-nan", and "segmentation fault (core dumped)". Our distance matrices or road construction matrices would often become suddenly nonsymmetric after being passed through a function, which we found was due to memory allocation problems. Memory leaks eventually became a very large problem, since our program would run for a long time before being killed. Through this project, we have become much better at exhaustively pinpointing what lines of code are causing bugs and fixing them, as well as using Valgrind to locate memory leaks. When the heuristic is now compiled and executed, we have made sure that all memory leaks are gone.

Our next step would have been to implement a more friendly GUI. Rather than printing our matrix and optimality in the terminal, we would ideally like to make our program return a graph that draws the nodes with distance respective to the distance matrix, to provide a visual representation of the optimal road constructions found by the heuristic. The most important lesson we learned from this project was how important planning extensively is before coding anything. This saved a lot of time in the long run, as we discovered what problems we would have encountered had we started immediately coding an idea, and instead modified our function and structure signatures to bypass these hurdles.

Kevin Huang - platform extensions, memory allocation
Christine Hwang - optimality calculations, code optimization, user interface
Alex Kahng - reindexing/sorting algorithm, connectedness, testing
Joseph Song - brute force implementation, Floyd-Warshall algorithm