Solving the Generalized Graph Search Problem with Genetic Algorithms

Ben Mowery

P.O. Box 16382
Stanford, CA 94309

(650) 497-6907

bmowery@cs.stanford.edu

Abstract:

This paper describes a genetic algorithm for finding low-cost paths through connectivity graphs. The map search problem does not lend itself to implementation with genetic algorithms because it is difficult to define a physically meaningful crossover operator: combining half of one set of driving directions with half of another set will rarely result in meaningful instructions. A selective crossover operator is used that will cross two fit solutions only if their paths intersect or come within one link of intersecting. The algorithm is demonstrated on a map of the bay area.

The connectivity graph is automatically extracted from snapshots of the Yahoo! Maps database. Thousands of side-by-side snapshots are combined into a single 24000 x 37000 pixel image and automatically converted to a connectivity graph through various image processing techniques.

1. Introduction and Overview

Finding a minimum cost path through a general connectivity graph is a very difficult problem when the graph is large. Its solution is useful in many applications, such as finding driving directions and routing packets through networks. Current algorithms either rely on heuristics to simplify the problem (such as finding a path through the simpler freeway graph first and only then searching on secondary roads), use an exhaustive search, or use a heuristically guided search. An exhaustive search is impractical on large graphs and heuristically guided algorithms will not necessarily find the best solution. The deterministic algorithms repeat the same mistakes time and time again. Although genetic algorithms are computationally very expensive, it is an appealing solution when the graph is large and simple heuristics are either too time consuming or impractical to implement.

Genetic algorithms do not lend themselves to solving graph search problems because most potential crossover and mutation operators are not physically meaningful. Suppose we define the genome to consist of a series right and left turns from the start point. One member of our solution population is a series of turns starting from 4th and King in San Francisco and another is a series of turns starting from downtown Palo Alto. Crossing over the two solutions will not yield a useful result since the graph topology in San Francisco is completely different than that in Palo Alto. This naive approach to applying a genetic algorithm degenerates to random search.

A more physically meaningful approach is to represent solutions as a list of nodes where a link exists between each member. When we attempt to combine two relatively fit solutions, we first check to see if they intersect. If they intersect, we cross them over at the point of intersection. The resulting solution is then a valid path through the graph. There is also the potential to quickly construct valid solutions with this method: if we have one solution that is fit because it comes close to the start point and another that is fit because it comes close to the end point and we cross them over the resulting solution will come close to both points. One interesting point with this approach is that it requires the population size be large and proportional to the complexity of the graph since it may be impossible to find two solutions that intersect in a small population.

2. Previous Work

2.1 Dijkstra's Algorithm

Dijkstra's shortest path algorithm is essentially an exhaustive search. We choose a start node S and our objective is to label every other node on the graph with the length of the minimum cost path to S. We write on each node the current estimated distance from S. Initially the estimate x_S for node S is 0 and x_N for each node N is infinity. Then a worklist algorithm is used to touch each node on the graph and for each node J the estimate written on neighboring node K is set to the minimum of x_K and $x_J + c$ where c is the cost assigned to the arc from J to K. The advantage of this approach is that it can be easily implemented in parallel, making it perfect for finding paths for packets through routers.

2.2 A* Algorithm

The A* algorithm is a heuristically guided exhaustive search. It uses a worklist that is initialized to the start node. Nodes are selected from the worklist based on a heuristically assigned score and expanded. The node is marked as closed and the nodes it links to are added to the worklist.

The heuristic assigns a score to each node on the worklist to determine which nodes are expanded first. One example criteria is the nearness to the destination which would make A* a greedy algorithm. Another heuristic might be nearness to the freeway system.

2.3 Hierarchical Search

Road maps can be searched in a hierarchical manner. A path can be found from a starting point on a small residential street starting point to the secondary road system. Once on the secondary road system, we need not consider the small streets anymore and thus have a much simpler graph to search. Then a path can be found to the nearest freeway. Once on the freeway system, the search graph is even simpler and the path to the freeway exit nearest the destination could probably even be found with a simple exhaustive search.

2.4 Comparison

For a path search on a road map, the A* algorithm and hierarchical search are both viable possibilities. With the right hand crafted heuristics these algorithms can be successful in such an environment. However, they will become too expensive when applied to a general graph or a graph where the heuristics are not valid. The genetic algorithm approach has the potential to perform well on more general graph search problems and without the effort of hand coding a different set of heuristics for each problem.

Another disadvantage with A* and hierarchical search is that both algorithms are typically deterministic, depending on the exact heuristics used. Thus, for a given problem they will always return the same answer. This may not be desirable for some applications because the algorithms will make the same mistakes over and over again. Even a good set of heuristics will occasionally make a mistake. Since the genetic algorithm approach is not based on any specific heuristic and is randomized, it will not be as vulnerable to this problem, although a flawed fitness function can sometimes cause similar effects.

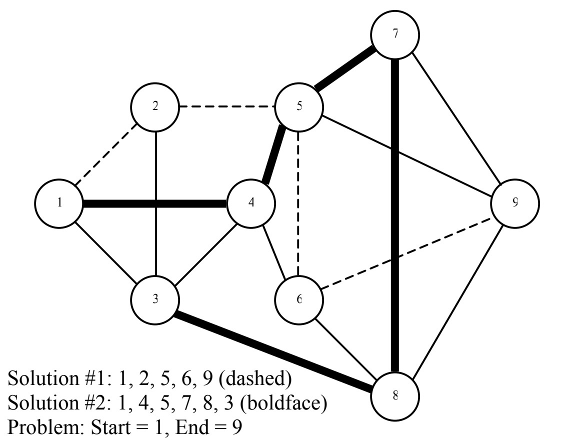

1
2
3
4
5
6
7
8
9
Solution #1: 1, 2, 5, 6, 9 (dashed)
Solution #2: 1, 4, 5, 7, 8, 3 (boldface)
Problem: Start = 1, End = 9

Figure 1: Two example solution structures displayed on a graph and their corresponding numerical representations

3. Description of the Algorithm

3.1 Data Structures

The connectivity graph consists of a numbered array of nodes. Each node contains a list of the nodes it links to (referenced to by position in the array) and the cost associated with each link. Each node is uniquely identified by its position in the connectivity graph array. Longitude and latitude coordinates are included in each node to permit rendering of the graph and a graphical display of the solution.

A solution consists of a list of nodes where each node is connected to its neighbors (figure 1).

The problem statement simply consists of a start node and an end node.

3.2 Creating the Initial Population

The population is initialized to a set of random solutions generated by picking a start node and randomly choosing links. The distribution of the lengths of these solutions is heuristically based on a function of the square root of the size of the graph. It is important to use longer solutions for larger graphs because the average number of nodes between the start node and end node will typically be larger.

One third of the solutions start at the start node, another third start at the end node, and the last third start at random points. This heuristic priming is preferable to using totally random solutions because a larger portion of the initial population will have the potential to be part of a final solution. Using totally random solutions will still result in a good solution, but convergence will take longer.

3.3 Fitness Function

The fitness function works on a pseudo-point system where points are awarded for the solution ending near the target end point, starting near the start point, and being low-cost. The score in the first two areas can be negative if the solution is exceptionally bad.

The fitness is computed with the following algorithm:

1. Compute the distance "as the crow flys" from the start point to the end point:

$$d = \sqrt{\left(startNode.lon - endNode.lon\right)^2 + \left(startNode.lat - endNode.lat\right)^2}$$

2. Compute the distance from the end point to the last point of the solution and normalize

$$d_e = \sqrt{\left(sol.lastNode.lon - endNode.lon\right)^2 + \left(sol.lastNode.lat - endNode.lat\right)^2} / d$$

3. Compute the distance from the start point to the first point of the solution and normalize

$$d_s = \sqrt{\left(sol.firstNode.lon - startNode.lon\right)^2 + \left(sol.firstNode.lat - startNode.lat\right)^2} / d$$

4. Compute the cost of the solution, totalCost, by summing the cost of the traversed links

5. If the solution is close to the goal points, consider the cost of the solution:

```
If $d_s < 0.3$ and $d_e < 0.3$ then,
    Fitness = $1 + (1 - d_e) + (1 - d_s) + 10 / totalCost$
Otherwise,
    Fitness = $1 + (1 - d_e) + (1 - d_s)$
```

6. Make sure the fitness value is never negative. If it is, set it to zero:

```
If Fitness < 0 then,
    Fitness = 0
```

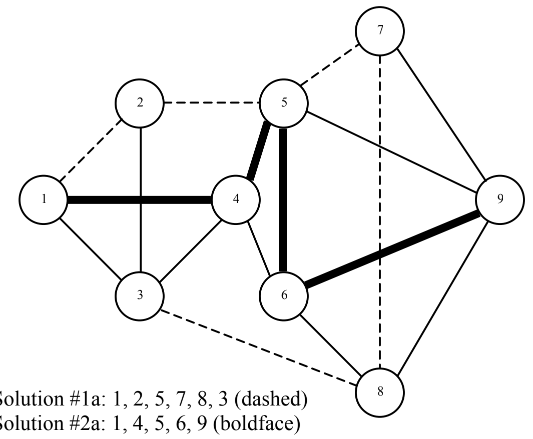
More fit solutions thus have higher scores. We do not consider the cost of the solution until the solution gets near both the start and end points to avoid premature convergence. When cost is considered during the entire process, trivially short solutions take over the population and the algorithm never converges on an acceptable solution. However, if cost is not considered at all we will have solutions that reach both the given start and end points but have strange detours in the middle of the solution or take ridiculously inefficient paths. For example, an infeasible structure might be one that veers in the opposite direction from the start point. It would get a negative $1 - d_e$ term reducing the fitness appropriately. Although occasionally it might be necessary for a solution to take a roundabout path to the goal, these solutions would not dominate the population.

The constant 1 term in the fitness equation is to permit the poorest solutions (those for which the $1 - d_e$ and $1 - d_s$ terms are negative) to have fitness values greater than zero. Negative values complicate the probabilistic selection of fit solutions.

3.4 Crossover

Two solutions are selected randomly based on their fitness. If it is determined that they intersect, the solutions are crossed over at the intersection point to form the new solution. This insures that the result is a valid and allowable structure: since the lists of nodes are crossed over at an intersection it is guaranteed that each node in the resulting list is connected to its neighbor by a link in the graph. Finding a pair that intersects is a very time consuming operation, especially when the graph is large reducing the chance that two solutions intersect.

Crossing over solutions that come within one link of intersecting (by adding the required link) dramatically increases the chances of finding a viable pair. The probability is multiplied by the average number of links from each node. In a representative run of the program, the probability increased from 1.3% to 4.5%.



Solution #1a: 1, 2, 5, 7, 8, 3 (dashed)
Solution #2a: 1, 4, 5, 6, 9 (boldface)

Figure 2: The two solutions that result from crossover of the solutions in Figure 1.

3.5 Mutation

Mutations typically consist of one or two very small changes to a solution. However, the physical meaningfulness requirement requires us to use another approach. A small change, such as simply changing one of the node IDs in the path will not result in a valid path because the nodes will no longer represent a connected path. Moving a node one or two links away from its current position is difficult because it involves a graph search (which is a smaller scale version of the very problem we're trying to solve) to find a path back to the next node in the solution.

When mutating a solution, we randomly choose one of five options:

- Appending additional randomly generated nodes to the end of the solution
- Prepending additional randomly generated nodes to the start of the solution
- Removing nodes from the end of the solution
- Removing nodes from the beginning of the solution
- Replacing the solution with an entirely new one generated in the same way as the population is initialized

These mutations, especially the last, insure that new genetic material will be introduced into the population. This is especially important given our selective crossover operator. If we ever encounter a situation where it is difficult or impossible to find two intersecting solutions that can be crossed over, mutation will eventually create a solution that intersects, allowing the algorithm to continue.

Mutation makes the search space fully accessible. Because the crossover operator can only combine existing solutions, it cannot cause new links on the graph to be explored – solutions can only be constructed from fragments of solutions present in the initial population. However, mutation guarantees that every link on the graph can be touched eventually, although in a much more random and haphazard way than crossover. This also reminds us why it is important to have a large initial population that will represent many possible paths through the search graph.

4. Results and Analysis

4.1 Sample Run

An example demonstrates the effectiveness of this algorithm and illustrates the effects of the various parameters. The parameters used for our first run, a trip from San Francisco to Fremont, are given in Table 1.

Table 1. Tableau for the Graph Search Problem

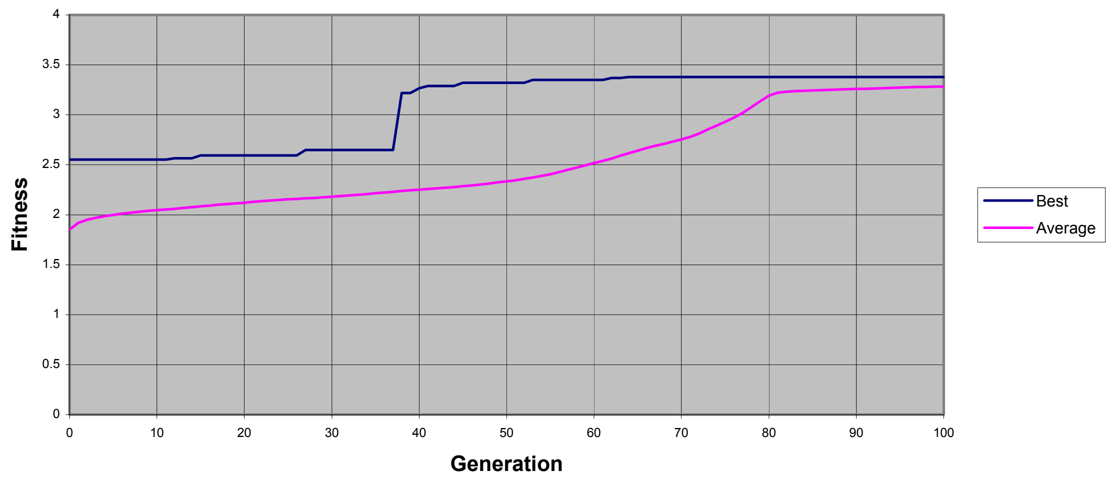
Objective:	Find the lowest cost path between two given points in a weighted graph
Representation Scheme:	Structure: List of node indices for a set of adjacent nodes in the graph K = number of nodes in the graph, N L = number of nodes in each solution can vary: minimum is 1, maximum is N A connectivity graph maps the node numbers onto the topology for a specific problem
Fitness:	Low path cost, nearness of path endpoints to goal points

Parameters:	Population Size M = 10000
	Weight on Cost = 3.0 **Make sure matches equations**
	Crossover = 30%
	Mutation = 1%

Termination Criteria:	Manual intervention
Result Designation:	Best individual is solution

Graphs of the average fitness and best solution fitness for 100 generations are shown in Chart 1.

Chart 1: Best and Average Fitness



The program is instrumented with code that permits us to follow the history of a particular solution. The solution of figure 5 began from humble origins as a route from near the endpoint, Fremont, to well south of San Francisco (figure 3). It only underwent small, incremental improvements (such as mutations and crossovers moving it slightly closer to San Francisco) until generation 38, when it was eclipsed by an alternate route that nearly touches the start and end points and uses the bay bridge (figure 4). However, our original solution evolves and prevails against its East Bay adversary when it crosses over with a fragment that goes towards San Francisco (figure 5). This solution undergoes a few almost unnoticeable refinements and eventually stabilizes at generation 65. In both figures 4 and 5, the solution exactly reaches both goal points.

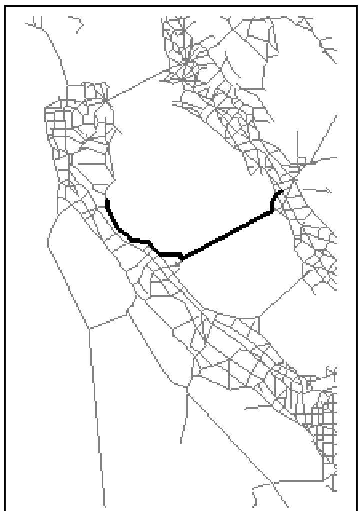


Figure 3: Generation 0

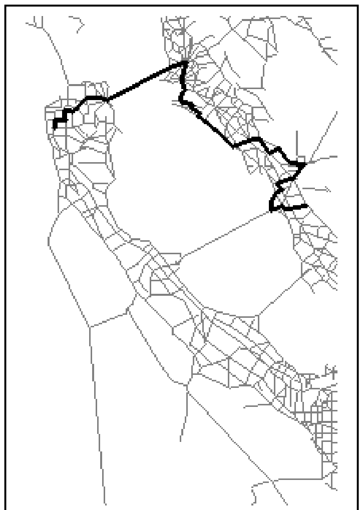


Figure 4: Generation 38

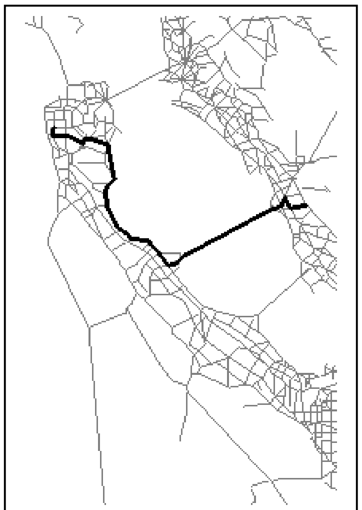


Figure 5: Generation 65

4.2 Effect of Parameters

4.2.1 Population Size and Intersection Method

Due to the selective nature of the crossover operator, a large population size is necessary. Recall that our crossover operator will only combine two solutions if they intersect. If it is called on two nonintersecting solutions, it will do nothing. In the example of section 4.1, 4.5% to 3% of crossovers succeed depending on population diversity. Initially, 4.5% of crossovers succeed, but as population diversity declines only 3% of crossover attempts succeed. It is essential to have a large population to insure that enough crossovers will be successful to evolve a successful solution. With a population of size 1000, the solution quality declines and with a population of 100 or less the algorithm will not converge on a meaningful solution.

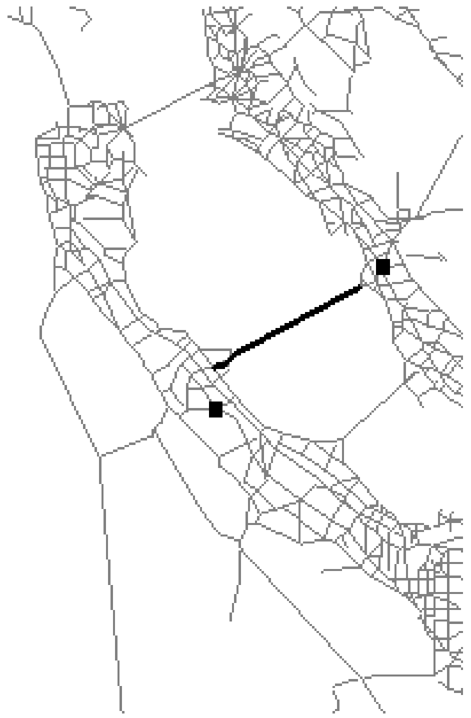


Figure 6: Result of placing excessive weight on cost. Boldface dots represent the goal points.

4.2.2 Cost Weight

The consideration of cost in the fitness function must be done with care to converge on a good solution. If too much emphasis is placed on cost, the fitness function will favor short and inexpensive solutions that do not reach the goal points (figure 6). If too little emphasis is placed on cost, we will obtain a solution that will get us where we're going, but will do so by a very roundabout path. In either case, population diversity can be harmed, prematurely eliminating solutions that could eventually evolve into a good answer.

The constant 3.0 given in Table 1 was picked by trial and error on several different problems. It will work without modification on most other connectivity graphs because each graph's link costs are normalized to values between zero and one. Additionally, most reasonable choices will result in valid solutions because cost is not weighed in the fitness function unless the solution is near both goal points. As long as the population size is large enough to maintain diversity, the most cost effective solution will be chosen from among those that reach the goal points.

4.2.3 Crossover and Mutation Rate

There is a direct correlation between crossover rate and speed of convergence. In the example of section 4.1, a crossover rate of 1% fails to converge on a remotely attractive solution within 100 generations. Increasing crossover causes faster convergence up to about 30% when the computational expense of performing the hundreds of thousands of intersection checks required finding candidate pairs becomes impractical.

5. False Starts

5.1 Image-Based Search

The algorithm we have been discussing involves searches on a traditional connectivity graph structure consisting of nodes with lists of neighbors. As discussed in the appendix, this graph was extracted from a 25000 by 37000 pixel composite map image of the bay area.

Initially, the algorithm attempted to search this image directly without using a connectivity graph. Solutions would consist of a set of (x, y) points in the image connected by straight lines. The starting and ending points would also be described by their pixel location in the image and the initial population would be constructed by randomly "exploring" the map by walking pixel by pixel, identifying which roads are freeways and secondary roads by their color.

Although this approach is unique and interesting, it is impractical because of the large amount of memory and processor power required. Since genetic algorithms generally require large solution populations, it is important that the various operators be inexpensive. Repeatedly walking a 1GB image pixel by pixel is too slow and inefficient to be practical in a genetic algorithm.

The final solution does the image processing once to extract a compact connectivity graph on which to execute the genetic algorithm.

5.2 Waypoints

Another method previously considered was to select a set of waypoints between the start and end points. These waypoints can be placed anywhere on the map and do not have to be connected directly. A conventional graph search algorithm would find paths between the waypoints which would then be concatenated to form the final solution. Since the individual subproblems solved by the conventional graph search algorithm are simple, the complexity and scalability associated with a naive graph search will not be an issue. The genetic algorithm will choose waypoints that guide the conventional algorithm to the most efficient path.

This approach is appealing because the crossover operator can be much simpler since the physical validity requirement is not an issue with waypoint placement. The waypoints need not be directly connected to each other by a single link as with the implemented algorithm. However, picking waypoints to insure that the subproblems are simple requires detailed analysis of the connectivity graph and is more effort than solving the problem directly.

Additionally, generating an initial population that will not dismally fail (even beyond the point where it could potentially evolve into something useful) would be difficult and require either knowledge of or assumptions about the graph topology. For example, suppose we base our initial population on permutations from a straight line between the start and end points that lie on different sides of a mountain.

It may be that the only path is a roundabout path that is radically different than any of the initial guesses would suggest.

Randomly choosing the waypoints to lie anywhere on the map would also make nearly every member of the initial population useless as the subproblems would likely be bigger than the given problem. For example, consider the a case where the first waypoint is the starting point, San Francisco, the second in San Jose, the third back in San Francisco, and the last in Fremont, our destination. Solving the problem directly is easier than solving the first subproblem.

6. Conclusion

A genetic algorithm approach offers a useful alternative to traditional graph search methods. Unlike deterministic heuristic based search algorithms, it is unlikely a genetic algorithm will repeatedly make the same kinds of mistakes. Additionally, this approach is effective even in the absence of simplifying heuristics used by most map search programs making it effective on general graphs. For extremely large graphs, it can find a close to optimal solution where an exhaustive search seeking a perfect solution would fail.

Heuristic search algorithms can be more effective in restricted domains. For example, a genetic algorithm doing address to address routing on a website would be impractical because of the computational expense. A set of heuristics hand coded for road map searches can obtain an answer in a fraction of a second as compared to the several minutes required by the genetic algorithm approach. However, the genetic algorithm approach is an appealing alternative to traditional search methods in problem spaces for which it is difficult to construct effective heuristics.

Appendix 1: Automatic Extraction of Connectivity Graph from Yahoo! Maps

A connectivity graph of real-world data was extracted from about 4300 images automatically downloaded from the Yahoo! Maps website. These images were stitched together to make a single 24000 x 37000 image which was then converted into a connectivity graph by way of various image processing techniques described below.

Step #1: Downloading the Map Images

A distributed perl script downloads many side-by-side map snapshots from the website. The desired longitude and latitude of the snapshot is encoded into the image URL. The conversion ratio between longitude/latitude and image pixels was calculated manually and then used to direct the perl script to systematically download side-by-side snapshots of the entire bay area.

Step #2: Making a Single Virtual Image

Knowing the conversion ratio between longitude/latitude and image pixels permits us to treat the many snapshots as a single virtual image. When a composite image is requested that spans multiple snapshots, each snapshot is loaded and the appropriate portion copied into a bitmap for manipulation by the image processing steps below. To generate the connectivity graph, a huge composite image is extracted that encompasses all the snapshots and is about 840 MB in size.

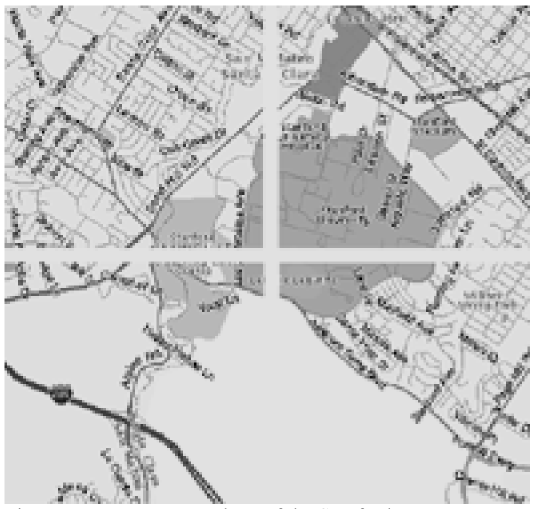


Figure 7: Four map snapshots of the Stanford area

Step #3: Extracting Freeways and Secondary Roads

Dealing with such a large image can be cumbersome since it cannot fit entirely within main memory. To speed up future processing, we construct a lower resolution image that contains only freeways and secondary roads. Freeways and secondary roads can be identified by their color. The 24000 x 37000 pixel composite image is broken into 20x20 blocks. If a block contains part of a freeway, the corresponding pixel in a smaller 1200 by 1850 bitmap is set. The presence of a secondary road will set a different bit in the corresponding image. Since the pixels in the resulting image are treated as bit masks, we can identify where freeways and secondary roads intersect by looking for pixels where both bits are set.

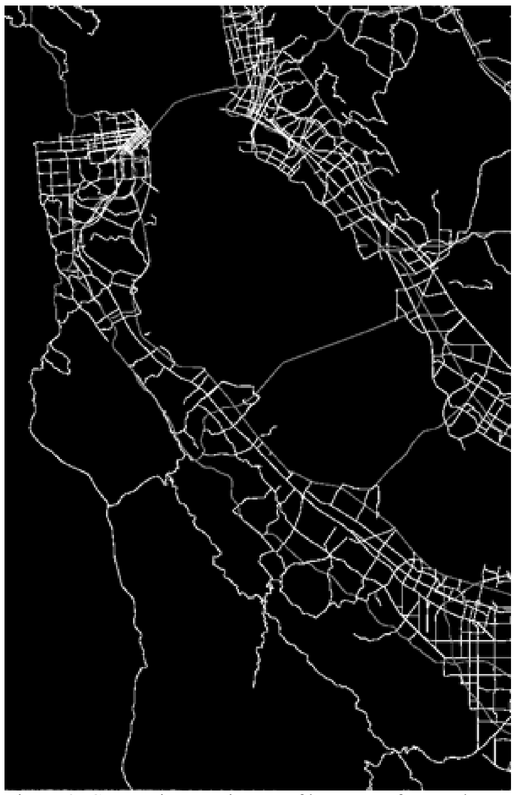


Figure 8: Composite map image of bay area after road extraction

Step #4: Make Roads One Pixel Thick

To simplify extraction of the connectivity graph, it is desirable for all the roads in the image to be only a single pixel wide. A flood fill algorithm is used later on to trace out paths between intersections through the image and could potentially be confused by roads that are wider than one pixel.

A traditional erosion filter is inadequate because it will disrupt the connectivity of the graph. It will destroy features that are already only a single pixel wide. We therefore implement a filter that considers a 3x3 block of pixels and is used to determine whether the center pixel should be erased. The filter asks the following question: if the center pixel were to be removed, would connectivity between the remaining (outer) pixels be disrupted? If not, the pixel will be removed.

We scan the outer pixels as numbered in figure 9 and count the number of state changes. For example, if pixel 5 is set and pixel 6 is not then we have a state change. If there are only two state changes, connectivity will not be disrupted. If there are more than two or no state changes, connectivity will be disrupted and the pixel should be left alone.

Step #5: Scan for and Mark Intersections

We begin constructing the connectivity graph by locating all the street intersections and creating a node for each. This is done with a filter that scans a 3x3 matrix in a manner similar to above, except that it requires more than 4 state changes. Imagine that the filter is centered on the intersection of two roads. In this case the center pixel would be lit, as would #2 and #4, causing 4 state changes. This will highlight all pixels that are near intersections. One problem with using a finite resolution image is that when many roads come close together they will be treated as a single large intersection.

1	2	3
8		4
7	6	5

Figure 9: Numbering of pixels in a 3x3 filter matrix

After highlighting pixels that are near intersections, we scan the image from top to bottom making a list of all nodes. When an intersection pixel is found we flood fill the region to make a list of all connected pixels. The x and y coordinates of these pixels are averaged to determine the node's location.

Step #6: Flood Fill From Intersections to Form Connectivity Graph

To obtain the arcs in the graph we walk the pixels starting from each intersection. Each pixel on the edge of the intersection is used as a seed point for a flood fill algorithm that will trace out the road and stop on the next intersection. Once we hit an intersection we enter the arc and the cost determined by summing the number of pixels walked. Red (freeway) pixels are weighted less than yellow (secondary road) pixels to reflect the lower cost of travel by freeway.

Bibliography:

Nilsson, Nils J. 1998. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach.* Prentice Hall.

Koza, John R. 1992. *Genetic Programming: on the programming of computers by means of natural selection*. The MIT Press.

Zhao, Yilin. 1997. *Vehicle Location and Navigation Systems*. Artech House.