Pathfinding Algorithms for Mutating Graphs

Haitao Mao

Computer Systems Lab 2007-2008

1 Abstract

Consider a map of an unknown place represented as a graph, where vertices represent landmarks and edges represent connections between landmarks. You have current information on whether each edge is traversible, as well past data about the availability of each connection. You have a preset destination that you want to reach as fast as possible. Pathfinding algorithms for static graphs involve computing the whole path from start to destination, but if the graph is rapidly changing, say due to some extreme environmental condition, then calculating the whole path in the beginning will not be feasible. The purpose of this project is to design and compare different pathfinding algorithms for a graph whose structure mutates to a significant extent. Algorithms may involve probabilistic theory, dynamic programming, heuristics, genetic programming, and variations of standard shortest-path algorithms such as Dijkstra's algorithm.

2 Introduction

The problem statement is as follows: given an initial graph structure of a mutating graph, a start vertex, an end vertex, and an edge history for every pair of vertices, develop an algorithm to travel from the start vertex to the end vertex. The mutating graph will be implemented in timesteps. After each move, each edge will either stay the same or be toggled by some random function of the current state of the graph. The plan is to create a sturdy algorithm for the general case of the problem, as well as variations for specific cases where the main algorithm would not be as effective. Algorithms will be compared and analyzed to determine the circumstances for which each one is best. This project will involve both theory and actual programming.

3 Background Literature

There are little to no studies available concerning mutating graphs, so research has been focused on graph theory in general as well as the more specific topic of dynamic graphs, which may change in structure. General shortest path and flow algorithms have been reviewed. Dynamic graph algorithms and query/update algorithms have been reviewed lightly. The results from this project are expected to be completely new and original.

4 Theory and Algorithms

Define randomized distance as the distance to destination node taking the possibility of graph mutation into account. For example, a vertex with two unit length paths leading to the destination will be closer in this sense than a vertex with only one. We use steady-state convergence and methods from numerical analysis to set up a system of equations we want the randomized distances to satisfy, and solve the system. We use dynamic programming to approximate distance to heuristically closer points first, then base calculations for farther vertices on these approximations. We use the previous states of the graph: we can use this data to develop a hashmap to approximate future mutations. The hashmap stores each mutating as a mapping from the original state to the new state, and then calculates the probability of toggling states. Then, that probability is used to calculate the probability that an edge will exist in any number of timesteps. We use genetic programming to find optimal values for algorithm-specific variables, such as probability estimate multipliers and heuristic functions. We focus on sparse graphs, graphs where the number of edges is significantly less than the square of the number of vertices. The edge weights are limited to positive doubles so mutation will be somewhat controlled; edge weights that are too large will never be traversed anyway.

Currently, the first algorithm proceeds chronologically, then for each vertex, it calculates the optimal vertex in the previous time step that could have led to this vertex. It uses the history hash map to predict the graph structure at that timestep, and uses an approximation error to weight lower timesteps. Then, it backtracks to find the best vertex after the first timestep to visit. This is the main body of the working java implementation of this algorithm:

```
for(int v=0; v<vertices; v++) prevvals[v] = inf;
prevvals[curvertex] = 0;
for(int t=0; t<tlimit; t++)
{
for(int v=0; v<vertices; v++)
{
curvals[v] = inf;
for(int e=0; e<adjlist[v].size(); e++)
{
Edge E = (Edge)adjlist[v].get(e);
if(prevvals[E.getVertex(v)]>=inf) continue;
double x = globhist.predictMutations(E.getWeight(),t)
   + prevvals[E.getVertex(v)];
if(x<curvals[v])
{
curvals[v] = x;
bestprev[v][t] = E;
}
}
}
for(int v=0; v<vertices; v++)
prevvals[v] = curvals[v];
if(curvals[vend]<inf)
{
if(curvals[vend]<bestend||bestend<0)
{
bestendtime = t;
bestend = curvals[vend];
}
}
else if(t==tlimit-1) System.out.println("Time limit is
   insufficient for the width of this graph");
}
int btrack = vend;
for(int t=bestendtime; t>0; t--)
{
System.out.println(t + " " + btrack);
btrack = bestprev[btrack][t].getVertex(btrack);
}
return bestprev[btrack][0];
```

By this point, there are several major algorithms working. The preliminary algorithms include taking a greedy at each step, and running a Dijkstra at each step. These algorithms do not take mutation into account, so they run pretty poorly and are used as a measure of how effective other algorithms are.

The first major algorithm consists of many independent parts. In the beginning, the algorithm makes a history of all the mutations and determines the probability of each mutation occurring. The history is its own data structure. Then, at each timestep, a dynamic programming approach takes the shortest path to that step using previous stored results and the predicted distance of the edge, which is just the probability that the edge will exist at that timestep. We are computing distance from our destination here, so that at the end, we can look at all the vertices connected to our starting vertex and see which one has the least distance. Note that here, we must also take into account the starting vertex itself, because the pathfinder can decide to not move anywhere.

The main problem with the first algorithm is that instead of doing any probabilistic calculation, you just approximate everything as the mean. This is suboptimal because it is not just an entirely accurate computation, but it comes pretty close, so it's actually quite difficult to improve upon. In the average case, this algorithm will do relatively well, and should be sufficient if coding time is of any concern.

The next algorithm tries to remedy the flaw of

the first algorithm by introducing a concept of randomized distance. This is basically a distance function of an edge that takes mutation into account. It is not clear how to best define this function, but we must take into account various properties. For one, it should return something at least one. Secondly, disregarding other paths, the more likely it is for a length one path to exist between two vertices, the lower the randomized distance should be. Also, obviously the randomized distance should not be dependent on the timestep. It should be dependent on the structure of the graph and the mutation rates of the edges. Finally, the randomized distance should satisfy the triangle inequality. Our randomized distance should be a metric, but this classification won't really help us much.

As a heuristic for randomized distance, we can either try to make the complexity in terms of number of vertices or number of edges. The second algorithm consdiers each edge, so it is pretty accurate. The third algorithm only considers the vertices in its heuristic, so it it significantly faster than the second algorithm, but it should also run worse than the second algorithm.

5 Testing

The testing interface as well as the algorithms themselves will be written in Java. Since graphs are difficult to develop graphics for, output will be limited to textual lists and charts. Testing will be done by generating graph structures and initial weights and devising a system for the random edge weight mutation. Then, repeated simulations will be run and the algorithms will be scored based on their performance for various types of starting parameters.

First, the algorithms will be tested for functionality and stability by examining its pathfinding in the interface and seeing if it behaves as expected. Then, algorithms will be tested for efficiency through random and user-specified initial states. Algorithms will be compared based on how fast they can find their destination, runtime complexity, and memory usage. If the algorithms take parameters, then genetic algorithms can be used to find optimal values for the parameters.

Second quarter was devoted solely to building algorithms and developing theory for solving the problem. Third quarter, I started actually testing my programs. For preliminary testing, I compared the accuracy of the programs against each other. I ran the Dijkstra which does not implement mutation stuff, and it did not run too well as expected. Then I ran the first algorithm, then the second, then the third. Here is some average data:

```
Dijkstra(no mutation):
small case: size 13
avg 5.75 turns to reach end
large case: size 500
avg 46.34 turns to reach end


first algorithm(most basic):
small case: size 13
avg 4.32 turns to reach end
large case: size 500
avg 28.46 turns to reach end

second algorithm:
small case: size 13
avg 4.25 turns to reach end
large case: size 500
avg 24.09 turns to reach end


third algorithm:
small case: size 13
avg 4.07 turns to reach end
large case: size 500
avg 25.95 turns to reach end
```

6 Expected Results

Results will consist of the efficiency, complexity, and stability of the algorithms tested. Results will be presented in charts, data tables, qualitative statements, and possibly graphics. Applications of the results are undetermined as this point, since this is not a commonly trod subfield of graph theory. Robots may be able to apply the algorithms in natural or man-modeled environments. The graph may be able to simulate a transportation network in order to find paths for pioneers. The randomly mutating edge weights may represent an unknown cause of change in an environment, even if there is a systematic pattern to the change. The factor may simply be too complex to model exactly and would be better approximated by a random variable. The project may be useful for applications further into the future, or may spark further development in the area which will lead to results that may be put into practice.

7 Literature Cited

References

[1] D. Frigoni, M. Ioffreda, U. Nanni, G. Pasqualbne, "Experimental Analysis of Dynamic Algorithms for the Single Source Shortest Paths Problem", 2000. http://www.acm.org/jea/TURING/Vol3Nbr5.pdf.

[2] C. Demetrescu, G. F. Italiano, "Algorithmic Techniques for Maintaining Shortest Routes in Dynamic Networks", 2006.

[3] U. Meyer, "Average-case Complexity of Single-Source Shortest-Paths Algorithms: Lower and Upper Bounds", 2001.