Pathfinding Algorithms for Mutating Graphs

Haitao Mao

Computer Systems Lab 2007-2008

1 Abstract

Say you are an avid traveler in today's rapidly changing world. 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 quickly 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 implemented involve probabilistic theory, dynamic programming, heuristics, genetic programming, and variations of standard shortest-path algorithms.

2 Introduction

Consider an unweighted graph whose edges change over time. Think of each edge as a binary switch with states off and on. Some edges will always be off. Then consider a dynamic shortest-path problem on this graph from a given start vertex to a given end vertex. We want an algorithm which will give the optimal first edge for the path, aka a pathfinder. The algorithm will perform in the following scenario: after each move the algorithm makes(with waiting on the same vertex being an option), one unit of time will pass and the edges will mutate accordingly. It will be scored based on the average number of moves it takes the pathfinder to go from start to end, while not using too much memory or computation time in the process. We must assume that the edges mutate based on some pattern that incorporates only a random variable. We may also assume that the graph is sparse; that is, the number is edges is on the order of the number of vertices. 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.

2.1 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.

2.2 Problem Statement

An input file will contain the following information: number of vertices, number of edges, which two vertices each edge connects, the start vertex, the destination vertex, and a mutation history. The mutation history is a list of past mutations for each edge that stores the state of each edge (on or off) at each timestep from a certain starting point up to current time.

3 Theory and Algorithms

3.1 Starting Out

First we look at some simplistic solutions to get a feel for this problem. What is one solution that can be coded in just a few minutes? How about a random walk? A random walk is a pathfinder that chooses the edge to traverse next randomly and uniformly. This algorithm will eventually reach the destination without doubt and takes almost no runtime, but it clearly fails on the grand scale. Whenever we have a dense cluster of vertices that is far away from our destination, our random walk will most likely get stuck for long periods of time. How can we improve on this algorithm without getting rid of its random nature? One helpful optimization is to keep track of the number of times each vertex has been visited and weight less frequently visited vertices higher. This will allow the random walk to get itself unstuck very easily and explore the graph without backtracking as much. In fact, this greatly boosts the efficiency of the algorithm. Consider a linear graph where edges do not mutate very rapidly. The unoptimized version will take exponential time to reach the destination, while the optimized algorithm takes linear time.

Another simplistic idea would be to take a greedy approach. This is done by using a breadth-first search(BFS) to calculate the distance of each vertex from the destination using all available edges, then to move to the closest vertex that is connected by an on edge. However, we run into the problem of an edge that stays off almost all of the time, thus stalling the pathfinder for a very long time. We can amend the floodfill to only consider edges that are on in the calculation of the distance. It turns out that this algorithm is rather effective for its simplicity; it is probably the algorithm that a human mind would use to solve this problem. The main problem with a floodfill approach is that it assumes that the path will remain viable in the future. In other words, it incorrectly assumes that mutations are negligible. The gives us the insight to add an additional layer to make a BFS viable. This will lead to observations later in the paper.

3.2 Probabilistic Analysis

Let's use a small graph as an example. Consider the graph with 6 vertices A, B, C, D, E, F and 8 edges $AB, BC, CD, DE, EF, FA, BD, CE$. We want to get from A to E. Initially, all edges are on. What seems like the intuitive edge to take? Of course, we want to get from A to E via the shortest route, so we compute the distance between each point and E. We see that C, D, F are distance 1 away, and A, B are distance 2 away. Since B, F are the only vertices connected to our starting vertex and F is closer to E than B, our greedy intuition tells us to go to F. However, what if the mutation rate is very high? Our planned path, AFE, might be disconnected before we can finish, and we will be stuck at F. If EF shuts down, then the distance between E and F will be at least 4. Now our move to F doesn't look as rosy anymore. If we move to B first, then we are guaranteed to take at least 3 moves, but we have a much higher chance that we will be able to reach E in 3 or 4 moves. Because the subgraph containing vertices B, C, D, E is much more densely connected. So do we go for the shorter, riskier route or the route with multiple backup routes? It's hard to decide, and our decision could change based on what we already know from the mutation history, so we're going to need some tool to help us decide.

We introduce a concept of randomized distance(RD). This distance encapsulates the mutations into a metric, as well as satisfying the condition that vertices which are more likely to be connected have a lower distance. As such, $RD(A, A) = 0$, $RD(A, B) = RD(B, A)$, and $RD(A, C) \leq RD(A, B) + RD(B, C)$. We have another intuitive observation that adding an edge cannot increase the randomized distance between two edges. This leads us to a possible way to compute randomized distances: by adding in on edge at a time and decreasing the distance continually. However, then the order we consider the edges will matter, but it really shouldn't for an accurate computation. We approach this from another angle. What if we had a bunch of conditions we want the distances to satisfy. We can set up a series of linear equations representing relationships between multiple vertices which are closely connected. For example, let us consider the complete graph with three vertices, A, B, C. Let's assume that each edge has a 50% chance of being on at any given time. The distance between the start and end vertex should be 2, because on average it takes 2 moves to reach the destination. In actuality, an exact computation for just that edge would yield a distance of $\sum_{i=1}^{\infty} i/2^i = 2$. In general, the actual RD would be lower because there are alternative paths, but in this case, it is always beneficial to stay in place because all the edges have the same chance of mutation, and it is also independent of the previous state of the edge. We need a way to conglomerate multiple paths into one distance value, but first, we must deal with an important concept.

There needs to be a way to approximate the state of an edge in future timesteps. All we are given in the input is a history of mutations, so we must

use this as well as possible. We create a history class which stores a count of the number of times it mutates from on or off and the number of times it stays on or off. The history structure captures all the relevant information for each edge in four variables. It then can predict the state of an edge any turns into a future, giving the probability that it'll be on. This data structure also updates dynamically with the pathfinder, which will help in case the history data is insufficient for the algorithm to make good decisions.

We take a moment to digress about a generalization of our history data structure. What if the mutation is just based just on a binary variable, but also on other factors? Four variables would not be sufficient for more complicated mutations. If, say, the graph was weighted, then we can expand the history class to contain a hashmap for each edge, which maps a previous state to its post-mutation state. Then, to compute the predicted mutation of some state, we look at the closest elements in the keyset, and take a weighted average of the mappings, where closer elements are weighted much higher. This generalization will also allow us to deal with an edge being affected by the states of other edges or big picture factors.

So we have an algorithm which makes direct use of the history class. It 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 data to predict the graph structure at that timestep. 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 BFS- based probabilistic 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];
```

3.3 Heuristics

A way to deal with the inherent randomness of the problem is to approximate every random variable as what the history class predicts it to be. For the steady-state convergence process to work on the RD equations, we need a way to weight the edges that come into play. One way is to consider the degree of each vertex and the degrees of the vertices adjacent to one vertex. Another heuristic is to consider the number of predicted mutations occuring around one edge, with higher weights for edges farther away. This heuristic would use the history structure effectively, but it requires a lot of tweaking to get correct, and genetic programming can be used to stablize the weight constants.

4 Conclusions

Throughout the paper we have developed a lot of approaches and ideas for the pathfinding problem on mutating graphs, but a direct comparison is hard to achieve due to the numerous factors that a perfect algorithm must have. Needless to say, different algorithms perform better in different circumstances. Where the enhanced random walk may work better in situations where we have insufficient information, the advanced heuristical algorithm will surely do better in randomly generated graphs with a distinct pattern of mutation. If you have read through the paper, you should be able to decide for yourself which algorithm works best under your circumstances, and

maybe the discussion has fueled your intuition and whetted your appetite for further research.

5 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.