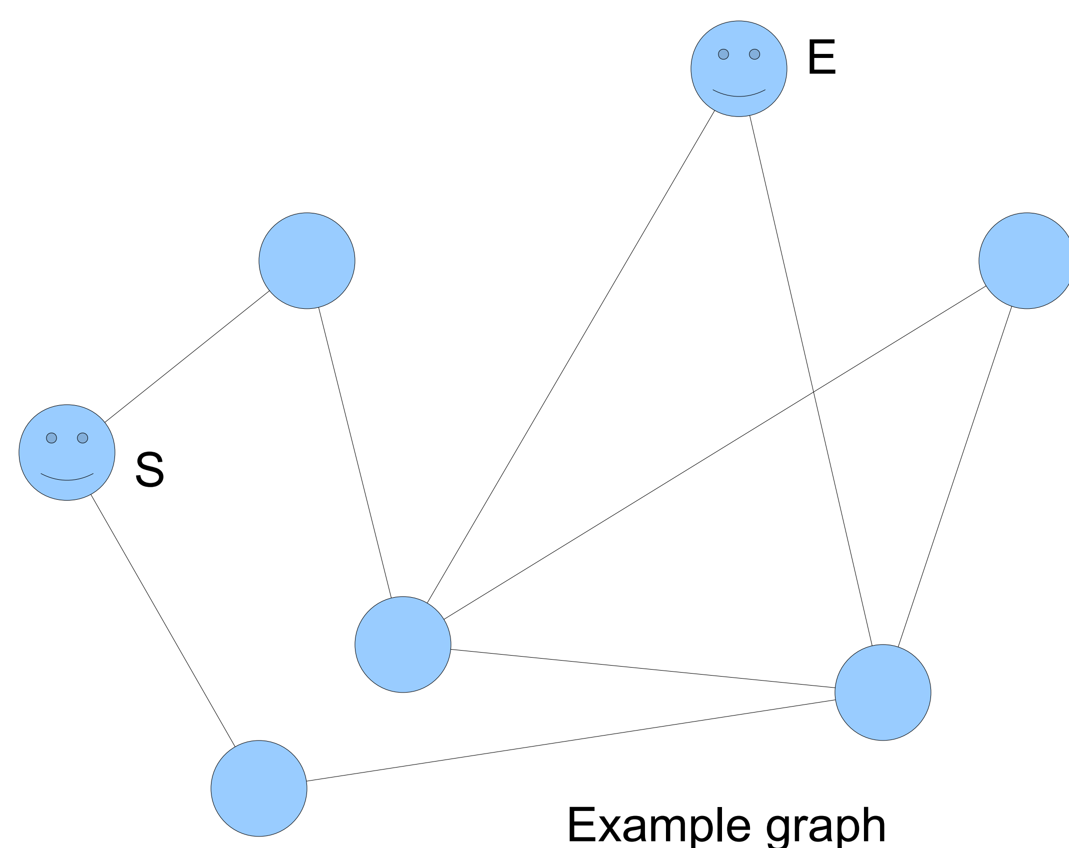
Pathfinding Algorithms for Mutating Graphs

Haitao Mao
Computer Systems Lab 2007-2008

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 the time it will take to travel between landmarks, as well as an archive about how the travel times changed through the past. 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 weights are rapidly changing due to some extreme condition of the place, 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 edge weights mutate randomly to a significant extent. Algorithms may involve probabilistic analysis, dynamic programming, heuristics, genetic programming, and variations of standard shortest-path algorithms such as Dijkstra's algorithm.



Example graph

Algorithms

Define randomized distance as the distance to destination node taking graph structure 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. We use genetic programming to find optimal values for algorithm-specific variables. 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. Complexity will be limited to $O(E^2\log(E)+V^2\log(V))$.

Background

For this problem, the structure of the graph will be static; that is, no vertices or edges will be added or removed. Only the edge weights will be dynamic, and they must change to a significant extent for the algorithm to be effective. If the mutations are negligible, then a standard shortest path algorithm will also serve as a pathfinder. Also, the mutations should form a pattern or probability distribution. The algorithm relies upon observing previous mutation to predict future mutation, so the two must be interdependent.

In this project, several simplifications to the general problem will be made for easier simulation. In any simulation, mutation must be discretely quantified. Here, mutation will be quantified in time steps, and every edge will take one time step unit to traverse. Hence, edge weights will not represent time but instead some generic cost. In a travel analogy where edges represent roads and vertices represent cities, road condition changes due to weather would be a time-based mutation, but if each road section had a toll that changed every hour, and everybody traveled at a constant speed, then it could be accurately modeled with a mutating weight graph. Edge weights must remain positive doubles. If the mutation renders the weight too large, then it will be reverted to the maximum double value, and similarly for weights too small. Also, the algorithm design will be tailored towards mutation which is essentially random, where the edge weight mutation is only dependent on the edge weight of that edge at the previous time step. Specifically, the mutation is assumed to be independent of time, graph structure, and other edge weights.

Results and Conclusions

As of now, the algorithm runs a lot better than an algorithm that doesn't take mutations into account, such as Dijkstra's algorithm, would. For every case tried so far, the proposed algorithm has reached the end with a significantly lower cost than the Dijkstra would have. This difference sometimes got as high as a factor of 5, because the Dijkstra pathfinder would often get stuck in choke points because the path it found earlier has changed and one of the edges no longer exists, sometimes forcing it to go back on itself. When Dijkstra's goes back on itself, it automatically wastes two turns' worth of time and cost and gets nowhere. However, the chance that our algorithm wastes time and cost is much, much lower due to its ability to predict mutations. Sometimes it will count on a edge becoming available in order to progress, but these assumptions are completely reasonable because either the path is far away and will have a lot of time to mutate, or has been seen as drastically changing from its history data. The algorithm can also detect when it may get stuck and avoid paths that may cause it to be stuck for long periods of time. Note that these are not really results, just the current progress. Add results next quarter.