A Survey of Shortest-Path Algorithms

Amgad Madkour[1], Walid G. Aref[1], Faizan Ur Rehman[2], Mohamed Abdur Rahman[2], Saleh Basalamah[2]

[1] *Purdue University, West Lafayette, USA*

[2] *Umm Al-Qura University, Makkah, KSA*

May 8, 2017

Abstract

A shortest-path algorithm finds a path containing the minimal cost between two vertices in a graph. A plethora of shortest-path algorithms is studied in the literature that span across multiple disciplines. This paper presents a survey of shortest-path algorithms based on a taxonomy that is introduced in the paper. One dimension of this taxonomy is the various flavors of the shortest-path problem. There is no one general algorithm that is capable of solving all variants of the shortest-path problem due to the space and time complexities associated with each algorithm. Other important dimensions of the taxonomy include whether the shortest-path algorithm operates over a static or a dynamic graph, whether the shortest-path algorithm produces exact or approximate answers, and whether the objective of the shortest-path algorithm is to achieve time-dependence or is to only be goal directed. This survey studies and classifies shortest-path algorithms according to the proposed taxonomy. The survey also presents the challenges and proposed solutions associated with each category in the taxonomy.

1 Introduction

The shortest-path problem is one of the well-studied topics in computer science, specifically in graph theory. An optimal shortest-path is one with the minimum length criteria from a source to a destination. There has been a surge of research in shortest-path algorithms due to the problem's numerous and diverse applications. These applications include network routing protocols, route planning, traffic control, path finding in social networks, computer games, and transportation systems, to count a few.

There are various graph types that shortest-path algorithms consider. A *general graph* is a mathematical object consisting of vertices and edges. An *aspatial graph* contains vertices where their positions are not interpreted as locations in space. On the other hand, a *spatial graph* contains vertices that have locations through the edge's end-points. A *planar graph* is plotted in two dimensions with no edges crossing and with continuous edges that need not be straight.

There are also various settings in which a shortest-path can be identified. For example, the graph can be *static*, where the vertices and the edges do not change over time. In contrast, a graph can be *dynamic*, where vertices and edges can be introduced, updated or deleted over time. The graph contains either *directed* or *undirected* edges. The weights over the edges can either be *negative* or *non-negative* weights. The values can be real or integer numbers. This relies on the type of problem being issued.

The majority of shortest-path algorithms fall into two broad categories. The first category is *single-source shortest-path* (SSSP), where the objective is to find the shortest-paths from a single-source vertex to all other vertices. The second category is *all-pairs shortest-path* (APSP), where the objective is to find the shortest-paths between all pairs of vertices in a graph. The computation of shortest-path can generate either *exact* or *approximate* solutions. The choice of which algorithm to use depends on the characteristics of the graph and the required application. For example, approximate shortest-path algorithms objective is

a time-dependent network. In essence, they investigate which networks can existing techniques be adopted to. Bast [11] illustrates speed-up techniques for fast routing between road networks and transportation networks. Bast's survey argues that the algorithms for both networks are different and require specialized speed-up techniques for each. Also, the survey presents how the speed-up technique performs against Dijkstra's algorithm. Moreover, the survey presents two open questions, namely, (1) how to achieve speed-up despite the lack of a hierarchy in transportation networks, and (2) how to efficiently compute local searches, e.g., as in neighborhoods.

Demetrescu and Italiano [39] survey algorithms that investigate fully dynamic directed graphs with emphasis on dynamic shortest-paths and dynamic transitive closures. The survey focuses on defining the algebraic and combinatorial properties as well as tools for dynamic techniques. The survey tackles two important questions, namely whether dynamic shortest-paths achieve a space complexity of $O(n^2)$, and whether single-source shortest path algorithms in a fully-dynamic setting be solved efficiently over general graphs. Nannicini and Liberti [105] survey techniques for dynamic graph weights and dynamic graph topology. They list classical and recent techniques for finding trees and shortest-paths in large graphs with dynamic weights. They target two versions of the problem, namely, time-dependence, and what they refer to as cost updates of the weights. Dean's survey [35] focuses on time-dependent techniques in a dynamic setting. It surveys one special case, namely, the First-In-First-Out (FIFO) network as it exposes structural properties that allow for the development of efficient polynomial-time algorithms.

This survey presents these aspects that are different from all its predecessors. First, it presents a taxonomy that can aid in identifying the appropriate algorithm to use given a specific setting. Second, for each branch of the taxonomy, the algorithms are presented in chronological order that captures the evolution of the specific ideas and algorithms over time. Moreover, our survey is more comprehensive. We cover more recent algorithms that have been invented after the publication of the other surveys.

4 Problem Definition

Given a set of vertices V, a source vertex s, a destination vertex d, where $s, d \in V$, and a set of weighted edges E, over the set V, find the shortest-path between s and d that has the minimum weight. The input to the shortest-path algorithm is a graph G that consists of a set of vertices V and edges E. The graph is defined as $G = (V, E)$. The edges can be *directed* or *undirected*. The edges have explicit weights, where a weight is defined as $w(e)$, where $e \in E$, or unweighted, where the implicit weight is considered to be 1. When calculating the algorithm complexity, we refer to the size of the set of vertices V as n and the size of the set of edges E as m.

5 Static Shortest-Path Algorithms

In this section, we review algorithms for both the single-source shortest-path (SSSP) and all-pairs shortest-path (APSP) problems.

5.1 Single-Source Shortest-Path (SSSP)

Definition: Given a Graph $G = (V, E)$ and Source $s \in V$, compute all distances $\delta(s, v)$, where $v \in V$.

The simplest case for SSSP is when the graph is unweighted. Cormen et al. [34] suggest that breadth-first search can be simply employed by starting a scan from a root vertex and inspecting all the neighboring vertices. For each neighboring vertex, it probes the non-visited vertices until the path with the minimum number of edges from the source to the destination vertex is identified.

Dijkstra's algorithm [42] solves the single source shortest-path (SSSP) problem from a given vertex to all other vertices in a graph. Dijkstra's algorithm is used over directed graphs with non-negative weights. The algorithm identifies two types of vertices: (1) Solved and (2) Unsolved vertices. It initially sets the source vertex as a solved vertex and checks all the other edges (through unsolved vertices) connected to the source vertex for shortest-paths to the destination. Once the algorithm identifies the shortest edge, it adds the corresponding vertex to the list of solved vertices. The algorithm iterates until all vertices are solved. Dijkstra's algorithm achieves a time complexity of $O(n^2)$. One advantage of the algorithm is that it does not need to investigate all edges. This is particularly useful when the weights on some of the edges are expensive. The disadvantage is that the algorithm deals only with non-negative weighted edges. Also, it applies only to static graphs.

Dijkstra's algorithm performs a brute-force search in order to find the optimum shortest-path and as such is known to be a greedy algorithm. Dijkstra's algorithm follows a successive approximation procedure based on Bellman Ford's optimality principle [17]. This implies that Dijkstra's algorithm can solve the dynamic programming equation through a method called the reaching method [41, 123, 124]. The advantage of dynamic programming is that it avoids the brute-force search process by tackling the sub-problems. Dynamic programming algorithms probe an exponentially large set of solutions but avoids examining explicitly all possible solutions. The greedy and the dynamic programming versions of Dijkstra's algorithm are the same in terms of finding the optimal solution. However, the difference is that both may get different paths to the optimal solutions.

Fredman and Tarjan [56] improve over Dijkstra's algorithm by using a Fibonnaci heap (F-heap). This implementation achieves $O(nlogn + m)$ running time because the total incurred time for the heap operations is $O(n\ log\ n + m)$ and the other operations cost $O(n + m)$. Fredman and Willard [57–59] introduce an extension that includes an $O(m+n\ log\ n/loglog\ n)$ variant of Dijkstra's algorithm through a structure termed the AF-Heap. The AF-Heap provides constant amortized costs for most heap operations and $O(log\ n/loglog\ n)$ amortized cost for deletion. Driscoll and Gabow [47] propose a heap termed the relaxed Fibonacci heap. A relaxed heap is a binomial queue that allows heap order to be violated. The algorithm provides a parallel implementation of Dijkstra's algorithm.

Another line of optimization is through improved priority queue implementations. Boas [23] and Boas et al. [24] implementations are based on a stratified binary tree. The proposed algorithm enables online manipulation of a priority queue. The algorithm has a processing time complexity of $O(loglog\ n)$ and storage complexity of $O(n\ loglog\ n)$. A study by Thorup [128] indicates the presence of an analogy between sorting and the SSSP problem, where SSSP is no harder than sorting edge weights. Thorup [128] describes a priority queue giving a complexity of $O(loglog\ n)$ per operation and $O(m\ loglog\ n)$ complexity for the SSSP problem. The study examines the complexity of using a priority queue given memory with arbitrary word size. Following the same analogy, Han [70] proposes a deterministic integer sorting algorithm in linear space that achieves a time complexity of $O(m\ loglog\ n\ logloglog\ n)$ for the SSSP problem. The approach by Han [70] illustrates that sorting arbitrarily large numbers can be performed by sorting on very small integers.

Thorup [129] proposes a deterministic linear space and time algorithm by building a hierarchical bucketing structure that avoids the sorting operation. A bucketing structure is a dynamic set into which an element can be inserted or deleted. The elements from the buckets can be picked in an unspecified manner as in a doubly-linked list. The algorithm by Thorup [129] works by traversing a component tree. Hagerup [69] improves over the algorithm of Thorup, achieving a time complexity of $O(n + m\ log\ w)$, where w is the width of the machine word. This is done through a deterministic linear time and space algorithm.

Bellman, Ford, and Moore [18, 53, 100] develop an SSSP algorithm that is capable of handling negative weights unlike Dijkstra's algorithm. It operates in a similar manner to Dijkstra's, where it attempts to compute the shortest-path but instead of selecting the shortest distance neighbor edges with shortest distance, it selects all the neighbor edges. Then, it proceeds in $n - 1$ cycles in order to guarantee that all changes have been propagated through the graph. While it provides a faster solution than Bellman-Ford's algorithm, Dijkstra's algorithm is unable to detect negative cycles or operate with negative weights.