A Survey of Shortest-Path Algorithms Summary

Taxonomy & Breakdown

Project Title	Interactive evaluation of shortest path methods
Client & Advisor	Goce Trajcevski, Mengxuan Zhang
Team	sddec23-14
Team Members	Alex Blomquist, Sam Caldwell, Selma Saric, Yadiel Johnson

1 Table of Contents

There are two broad categories for shortest-path algorithms:

Single-source Shortest-path (SSSP)	All-pairs Shortest-path (APSP)
Finding shortest-path from a single source vertex to all other vertices	Finding the shortest-path between all pairs of vertices in a graph

But they may be classified under additional, different categories due to how much research there exists on the subject. Madkour et al. propose the following taxonomy for shortest-path algorithms.

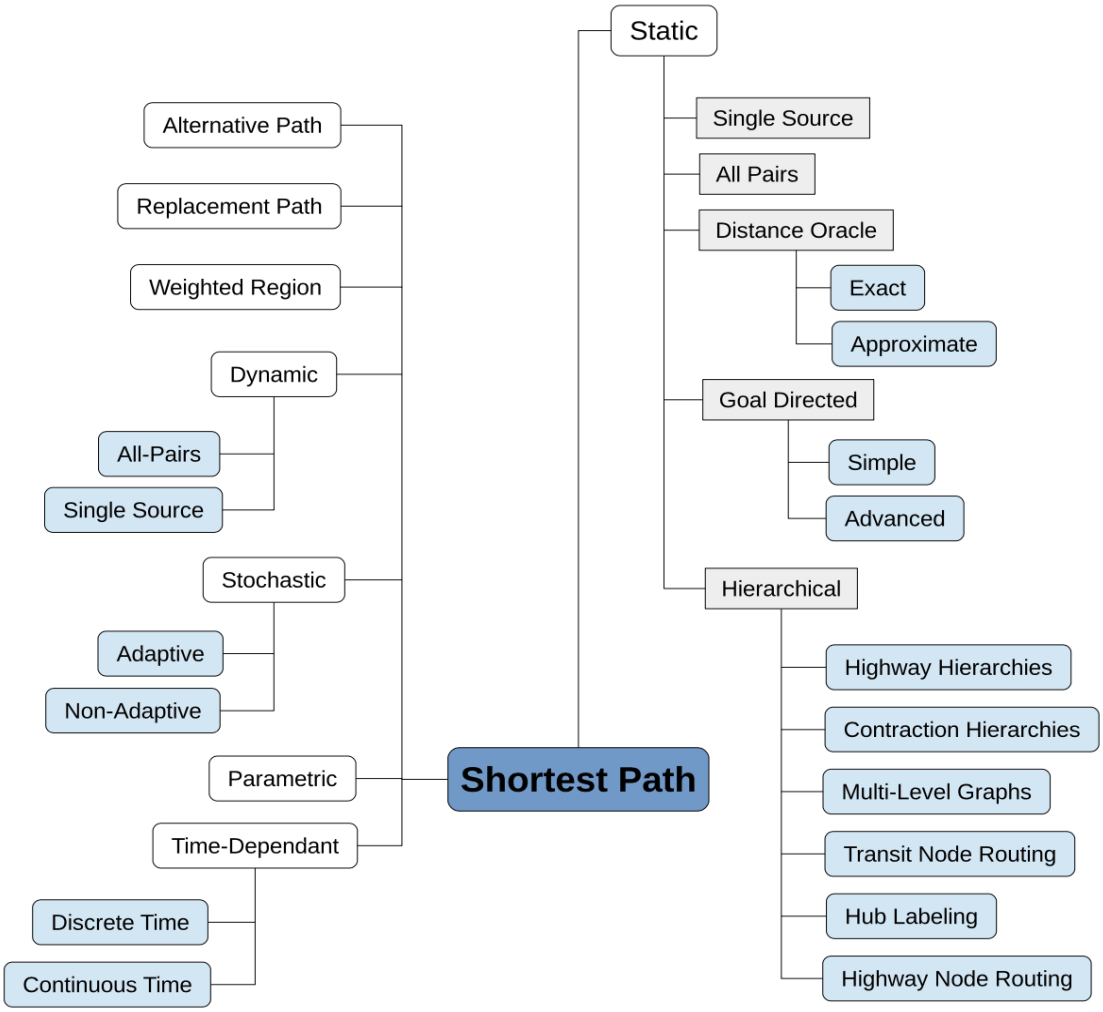

Figure 1: Taxonomy of Shortest-Path Algorithms

2 Taxonomy

The following is a list detailing these classifications.

- **Hierarchical Algorithms:** Breaks shortest-path into a linear complexity problem, can lead to enhanced performance via orders of magnitude
- **Goal-Directed Algorithms:** Optimize in terms of distance or time toward the target solution
- **Distance Oracle Algorithms:** Includes a pre-processing step to speed up the shortest-path query time, which can be either exact or approximate
- **Dynamic Algorithms:** Process updates or query operations on a graph over time. These updates can insert, delete edges, or update from graphs. The query operation computes the distance between source and destination vertices. Includes both APSP and SSSP
- **Time-Dependent Algorithms:** Best for graphs that change over time in a predictable fashion
- **Stochastic Shortest-path Algorithm:** Captures uncertainty associated with edges by modeling them as random variables.
- **Parametric Shortest-path Algorithm:** Computes solutions based on all values of a specific parameter
- **Replacement Path Algorithm:** Computes a solution that avoids a specified edge for every edge between a source vertex and the destination vertex. Replacement paths achieve good performance by reusing the computations of each edge it avoided
- **Alternative Path Algorithm:** Also computes the shortest path between vertices that avoids a specified edge. The big difference is that the replacement paths are not required to indicate a specified vertex or edge, avoiding rather a specified edge on the shortest path.

3 Breakdown

3.1 Static Shortest-Path Algorithms

3.1.1 Single-Source Shortest-Path (SSSP)

Definition

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

Algorithms

- **Unweighted:** The simplest SSSP case. It involves a breadth-first search that scans all neighboring vertices from the root vertex. For each neighboring vertex, it searches non-visited vertices until the path with the minimum number of edges is found.
- **Dijkstra's Algorithm:** Used to find the shortest path in directed graphs with non-negative weights by identifying vertices as "solved" or "unsolved". Initially, it sets the source vertex as a solved one and proceeds to check all other edges (with unsolved vertices) connected to the source vertex and the destination. It's a brute-force algorithm achieving $O(n^2)$ time complexity and has the advantage of not needing to search all edges, which is especially useful when some weights are particularly expensive. However, it cannot be done on non-static graphs and cannot deal with negative weights.
    - It can be altered to solve the dynamic programming equation through a method called "reaching". Reminder: dynamic programming avoids brute-force search by opting into tackling subproblems instead.
- **Fredman and Tarjan's Fibonacci Heap:** By modifying Dijkstra's algorithm, this variation achieves $O(n \log n + m)$ time complexity. This is because the incurred time for heap operations is still $O(n \log n + m)$ while other operations only cost $O(n + m)$.
    - Later, an extension that included $O(m + (n \log n)/\log\log n)$ variant of Dijkstra's was introduced (called the "AF-Heap") which provides amortized costs for most heap operations and $O(\log n/\log\log n)$ for deletion.
    - Driscoll and Gabow propose a relaxed heap, which allows the heap order to be violated. It is simply a parallel implementation of Dijkstra's.
- **Stratified Binary Tree (Improved Priority Queue):** An online, manipulatable priority queue with a time-complexity of $O(\log\log n)$ and storage complexity of $O(n \log\log n)$.
    - This can be done with greater efficiency by providing memory and focusing on the fact that you are simply sorting edges and applying a deterministic integer sorting algorithm in a linear space that achieves a time complexity of $O(m \log\log n \log\log\log n)$.
- **Thorup's Deterministic Linear Time and Space Algorithm:** Using a hierarchical bucketing structure that avoids the sorting operation, the algorithm works by traversing a component tree and, with Hagerup's additions, reaches a time complexity of $O(n + m \log w)$ with w being the width of the machine word.
- **Bellman-Ford Algorithm:** It is different from Dijkstra's in that instead of selecting the shortest distance neighbor edges, it selects all the neighbor edges. This allows it to work

with negative edge weights and detect negative cycles (unlike Dijkstra's) at the cost of slower run time. Its time complexity is $O(nm)$.
    - Yen proposed two performance modifications: one is edge relaxation and the other is dividing edges based on a linear ordering of the vertices.
    - Bannister and Eppstein introduced an improvement over Yen's modifications that use a random ordering instead of an arbitrary linear ordering.
- **Karp's Algorithm:** Designed with the intention of addressing negative weight cycles. Karp defines that a concept dubbed the "minimum cycle mean" and indicates that finding the minimum cycle mean is similar to finding the negative cycle. This algorithm also has a time complexity of $O(nm)$.

3.1.2 All-Pairs Shortest-Path (APSP)

Definition

Given a graph $G = (V, E)$, compute all distances between a source vertex s and a destination v, where $s \in V$ and $v \in V$.

Algorithms

- **Dijkstra's Algorithm**: For graphs with non-negative edge weights, using the normal Dijkstra algorithm over each vertex in the graph results in a time complexity of $O(mn + n^2 logn)$.
- **Floyd-Warshall Algorithm:** Attempts to find all pairs shortest-paths (APSP) in a weighted graph containing positive and negative weighted edges. Their algorithm can detect the existence of negative-weight cycles using a diagonal path matrix but it does not resolve these cycles. The complexity of Floyd-Warshall algorithm is $O(n^3)$, where n is the number of vertices. Note that it cannot find the exact shortest-path pairs because it does not store information about intermediate vertices.
    - By modifying it so that it can also store that information, it can have a space complexity of $O(n^3)$ or $O(n^2)$ depending on the usage of a single displacement array.
    - It may not be better than Dijkstra's if $m < n^2$ (assuming non-negative edge weights).

Variations

There is a list of enhancements for the Floyd-Warshall algorithm in p.7 of the survey.

The best *non-negative edge weight* time complexity is $O(n^2 logn)$, given by an algorithm from Moffat and Takaoka.

- The algorithm sorts all adjacency lists in order of increasing weight. It then performs a SSSP computation *n* times in iterations that feature two phases.

The best *positive integer edge weight* complexity is $O(n^w + c)$ where $w < 2.575$, proposed by Coppersmith and Winograd.

- Their proposed algorithm provides a transition between the *fastest exact* and *approximate* shortest-paths algorithms with a linear error rate. The algorithm focuses on directed graphs with small positive integer weights in order to obtain additive approximations. The approximations are polynomial given the actual distance between pairs of vertices.

3.2 Dynamic Shortest-Path Algorithms

This class of algorithm must be able to process updates to the graph in between query operations. Updates are defined as either insertion or deletion of edges. Algorithms that can only handle the former are *incremental algorithms*, while algorithms that can only do the latter are *decremental algorithms*. These are known as *partially dynamic*, whereas an algorithm that can do both together is known as a *fully dynamic algorithm.*

- **Demetrescu and Italiano's algorithm:** A fully dynamic algorithm over directed graphs for APSP with real-valued edge weights, where every edge can have a predefined number of values. It has an amortized time complexity of $O(Sn^{2.5} log^3 n)$ for update operations while achieving an optimal worst-case for query processing time.
    - It does this by inserting or deleting a vertex and all of its possible edges whenever it is inserted or deleted.
    - It also maintains a complete distance matrix between updates.
- **Thorup's algorithm:** An improvement over Demetrescu and Italiano's algorithm. It reduces the fully dynamic graph problem into a smaller set of decremental problems using a fully dynamic minimum spanning tree for a more efficient solution.
- **Bernstein's algorithm:** A $(2 + \varepsilon)$-approximation algorithm for APSP over an undirected graph with positive edge weights. Its update time is almost linear and its query time is $O(loglogn)$. Due to it using a randomized update procedure, it relies on guessing several different values for $d(x, y)$.
    - He also proposes an $(1 + \varepsilon)$-approximate algorithm that improves over existing studies with respect to the delete operation and edge weight increase. The algorithm computes the decremental all-pairs shortest-paths on weighted graphs. The approach achieves an update time of $O(mn^2)$ using a randomized algorithm
- **Roddity and Zwick's algorithm:** A fully dynamic APSP algorithm for unweighted directed graphs, where the algorithm is randomized. It utilizes ideas from decremental algorithms.
- **Hezinger et al. algorithm:** Enhances over the fastest deterministic algorithm by achieving an update time of $O(n^{2/5})$. Also achieves a constant query time. They also proposed a deterministic algorithm with an update time of $O(mn)$ and a query time of $O(loglogn)$. The

proposed approach maintains a shortest-paths tree that is bounded by distance with an Even-Shiloach tree based de-randomization technique.

3.3 Time-Dependent Shortest-Path Algorithms

A time-dependent shortest path algorithm processes graphs that have edges associated with a function, known as an *edge-delay* function. The edge-delay function indicates how much time is needed to travel from one vertex to another vertex. The query operation probes for the minimum-travel-time path from the source to the destination vertex over graph. The returned result represents the best departure time found in a given time interval.

3.3.1 Continuous-Time Algorithms

- **Kanoulas et al.:** Proposed an algorithm that finds a set of all fastest paths from source to destination given a specified time interval. This interval is defined by the user and represents the departure/arrival time. The query algorithm finds a partitioning scheme for the time interval and creates a set of sub-intervals where each sub-interval is assigned to a set of fastest paths. Probes the graph only once instead of multiple times.
- **Ding et al.:** Proposed an algorithm that finds the departure time that minimizes the travel time over a road network. Also, traffic conditions are dynamically changing in the road network. The algorithm is capable of operating on a variety of time-dependent graphs.
    - Also proposed an algorithm for the shortest path problem over a large time-dependent graph GT. Each edge has a delay function that denotes the time taken from the source vertex to the destination vertex at a given time. The user queries the least travel time (LTT).
    - Has space complexity of $O((n + m) \propto (T))$ and a time complexity of $O((nlogn + m) \propto (T))$.
- **George et al.:** Proposed a Time-Aggregated Graph (TAG) that changes its topology with time. In TAG, vertices and edges are modeled as time series. Also responsible for managing the edges and vertices that are absent during any instance in time. They proposed two algorithms to compute shortest-path using time-aggregated network (SP-TAG) and best start-time shortest path (BEST).
    - SP-TAG finds the shortest path at the time of given query using a greedy algorithm. Time complexity: $O(e(logT + logn))$
    - BEST finds out the best start-time (i.e., earliest travel time) over the entire period using TAG. Time complexity: $O(n^2eT)$
        - e represents edges, n represents vertices, and T represents the time instance

3.3.2 Discrete-Time Algorithms

- **Nannicini et al.:** Proposed a bidirectional A* algorithm that restricts the A* search to a set of vertices that are defined by a time-independent algorithm. Operates in two modes:
    - The first mode, namely the *forward search* algorithm, is run on the graph weighted by a specific cost function.
    - The second mode, namely the *backward search,* is run on the graph weighted by a lower bound function.

- **Delling and Wagner:** Concluded that most of the techniques that operate over time-dependent graphs guarantee correctness by augmenting the preprocessing and query phases subroutines.
- **Foschini et al.:** Concluded that linear edge-cost functions cause the shortest path to the destination change $n^{\boldsymbol{\theta}(logn)}$ times. They study the complexity of the arrival time by mapping the problem to a parametric shortest-paths problem in order for it to be analyzed correctly.
- **Demiryurek et al.:** Proposed a technique to speed up the fastest path computation over time-dependent spatial graphs. They propose a technique based on the A* bidirectional time-dependent algorithm that operates in two main stages.
    - First stage: Pre-computation - partitions the graph into a set of partitions that do not overlap. Next, they calculate a lower-bound distance label for vertices and borders.
    - Second stage: Online - probes for the fastest path by utilizing a heuristic function based on the computed distance labels.
    - Results indicate that the proposed technique decreases the computational time and reduces the storage complexity significantly.

3.4 Stochastic Shortest-Path Algorithms

A stochastic shortest-path attempts to capture the uncertainty associated with the edges by modeling them as random variables. The objective is to compute the shortest-paths based on the minimum expected costs. There are two types of algorithms for this type of problem, *adaptive* and *non-adaptive algorithms*. Adaptive algorithms determine the next best step based on the current graph at a certain time instance. Non-adaptive algorithms focus on minimizing the length of the path.

3.4.1 Adaptive Algorithms

- **Miller-Hooks and Mahmassani's Algorithm:** Determine the apriori (theoretical deduction) least-expected-time-paths from all source vertices to a single destination vertex. The computations are done for each departure time during busy time of the graph. It also proposes a lower-bound over these apriori least-expected-time- paths.
- **Nikolova's Algorithm:** Maximizes the probability without exceeding a specific threshold for the shortest-paths length. It defines a probabilistic model where the edge weights are drawn from a known probability distribution. The optimal path is one with the maximum probability indicating a path does not pass a specific threshold.

3.4.2 Non-Adaptive Algorithms

- **Loui's Algorithm:** Uses a utility function with the length of the path. The utility function is monotone and non-decreasing. When the utility function exhibits a linear or exponential behavior, it becomes separable into the edge lengths. This allows the utility function to be identified using classical shortest-paths algorithms via paths that maximize the utility function.
- **Nikolova's Algorithm:** Algorithm for optimal route planning under uncertainty. It defines the target as a function of both the path length and the departure time starting from the source. Both the path and start time are jointly optimizable due to the penalizing behavior for late and early arrivals. This joint optimization is reducible to classic shortest-path algorithms.

3.5 Parametric Shortest-Path Algorithms

The objective of these algorithms is to compute the shortest-paths for all vertices based on a specific parameter. It searches for the parameter values, known as *breakpoints*, where the shortest-path tends to change. The edge value varies based on a linear function of the parameter value.

3.5.1 Algorithms

- **Mulmuley and Shah**: This algorithm proposes a model for lower-bound computation. It is a variant of the Parallel Random Access Machine. It starts with a lower-bound definition about the parametric complexity of the shortest-path problem. It plots the weights of the shortest-path as a function, which results in an optimal cost graph that is piecewise-liner and concave. The breakpoints are defined as a fixed set of linear weight functions over a fixed graph.
- **Young:** Uses a modified Karp and Orlin's algorithm to use Fibonacci heaps to improve its performance instead. This allows obtaining shortest-paths in polynomial time by increasing its tractability.
- **Erikson:** Algorithm for computing the maximum flow in planar graphs. It maintains three structures; an edge spanning tree, a predecessor dual vertex set, and the slack value of dual edge set. They compute the initial predecessor pointers and slacks in $O(nlogn)$ time by using Dijkstra's.

3.6 Replacement Shortest-Path Algorithms

For every edge *e* on the shortest-path from the source to the destination, the replacement path algorithm calculates a new shortest-path from the source to the destination that avoids *e.*

3.6.1 Algorithms

- **Emek:** Computes the replacement path in near-linear time. It requires $O(nlog^3(n))$ time during the preprocessing stage and $O(hloglogn)$ time to answer the replacement path query
    - h is the number of hops in a weighted planar directed graph
- **Roditty and Zwick:** Proposes a Monte-Carlo randomized algorithm that computes the replacement path in an unweighted directed graph.