Priyanka Sharma al. International Journal of Recent Research Aspects ISSN: 2349-7688,
Vol. 1, Issue 1, June 2014, pp. 30-34

Shortest Path Algorithms Technique for Nearly Acyclic Graphs

Priyanka Sharma[1], Surjeet Dalal[2]

[1]Student, M. Tech, ESEAR, Ambala

[2]Assistant Professor, Dept. of CSE, E-Max group of Institutions, Ambala

***Abstract*—Dijkstra's algorithm solves the single-source shortest path problem on any di- rected graph in O(m+nlogn) worst-case time when a Fibonacci heap is used as the frontier set data structure. Here n is the number of vertices and m is the number of edges in the graph. If the graph is nearly acyclic, then other algo- rithms can achieve a time complexity lower than that of Dijkstra's algorithm. Abuaiadh and Kingston gave a single source shortest path algorithm for nearly acyclic graphs with O(m + nlogt) worst-case time complexity, where the new parameter t is the number of delete-min operations performed in priority queue manipulation. For nearly acyclic graphs, the value of t is expected to be small, allowing the algorithm to outperform Dijkstra's algorithm. Takaoka, using a different definition for acyclicity, gave an algorithm with O(m+nlogk) worst- case time complexity. In this algorithm, the new parameter k is the maximum cardinality of the strongly connected components in the graph. This paper with the refrence of Shane Saunders thesis presents several new shortest path algorithms that define trigger vertices, from which efficient computation of shortest paths through underlying acyclic structures in the graph is possible. If trigger vertices are defined as a set of precomputed feedback vertices, then the all-pairs shortest path prob- lem can be solved in O(mn + nr2) worst-case time. This allows all-pairs to be solved in O(mn) worst-case time when a feedback vertex set smaller than the square root of the number of edges is known.**

***Keywords*—Acyclic, complexity, GSS, planarity, strongly connected components, topological ordering , Trigger vertex.**

I. INTRODUCTION

Computing shortest paths in a graph G = (V,E) is used in many real-world applications like route planning in road networks, timetable information for railways, or scheduling for airplanes, route planning systems for cars, bikes, and hikers , spatial databases [Shekhar et al., 1997], and web searching [Barrett et al., 2000]. In general, Dijkstra's algorithm finds an exact shortest path of length d(s, t) between a given source s and target t. Unfortunately, the algorithm is far too slow to be used on huge datasets. Thus, several speed-up techniques have been developed yielding faster query times for typical instances, e.g., road or railway networks. In, basic speed-up techniques have been combined systematically. One key observation of their work was that it is most promising to combine hierarchical and goal-directed techniques. However, since the publication of , many powerful hierarchical speed-up techniques have been developed, goal-directed techniques have been improved, and huge data sets have been made available to the community. In this work, we revisit the systematic combination of speed-up techniques.

For our study, we exemplarily consider the following four speed-up techniques:

Goal-Directed Search. The given edge weights are modified to favor edges leading towards the target node [Hart et al., 1968, Shekhar et al., 1993]. With graphs from timetable information, a speed-up in running time of a factor of roughly 1.5 is reported in [Schulz et al., 2000].

Bidirectional Search. Start a second search backwards, from the target to the source (see[Ahuja et al., 1993], Section 4.5). Both searches stop when their search horizons meet. Experiments in [Pohl, 1969] showed that search space can be reduced by a factor of 2, and in [Kaindl and Kainz, 1997] it was shown that combinations with goal-directed search can be beneficial.

Multi-Level Approach. This approach takes advantage of hierarchical coarsenings of the given graph, where additional edges have to be computed. These can be regarded as distributed to multiple levels. Depending on the given query, only a small fraction of these edges have to be considered to find a shortest path. Using this technique, speed-up factors of more than 3.5 were observed for road map and public-transport graphs [Holzer, 2003]. Timetable information queries could be improved by a factor of 11 (see [Schulz et al., 2002]), and also in [Jung and Pramanik, 2002] good improvement for road maps is reported.

Shortest-Path Containers. These containers provide a necessary condition for each edge, whether or not it has to be respected during the search. More precisely, the bounding box of all nodes that can be reached on a shortest path using this edge is stored. Speedup factors in the range between 10 and 20 can be achieved [Wagner and Willhalm, 2003].

The combination of the four techniques is very natural, since all of the techniques modify the search space of Dijkstra's algorithm independently of each other: Goal-directed search directs the search space towards the target of the search by modifying the edge lengths; bidirectional search maintains two search spaces; the multi-level graph approach runs common Dijkstra's algorithm on a subgraph of the augmented input graph; and with shortest-path containers, search space can be pruned by ignoring such edges that for sure do not contribute to a shortest path.

Defination- Trigger vertices:- the definition of trigger vertices depends on the specific algorithm. The simplest such algorithm defines trigger vertices as the roots of trees that result when the graph is decomposed into tree structures.

Motivation:- The motivation of this thesis is to design specialised shortest path algo- rithms for use on nearly acyclic graphs. A nearly acyclic graph is a graph that contains relatively few cycles for its size. One kind of nearly acyclic graph can be visualised by extending the strictly downhill example, described earlier, to allow some uphill paths. In this nearly downhill analogy, most, but not all, paths in the graph are downhill. Since such graphs are not strictly downhill, an efficient strictly downhill shortest path algorithm cannot be used. Therefore a standard shortest path algorithm would normally be used to solve shortest paths in such graphs. However, given that most of the graph is downhill, there should be some more efficient way to solve shortest paths. This requires a new specialised algorithm for nearly downhill graphs to be invented; that is, an algorithm for nearly acyclic graphs. By designing new shortest path algorithms for nearly acyclic graphs these kinds of problems may be solved almost as efficiently as problems on acyclic graphs.

II. Graph Terminology

This section reviews some basic graph theory terms that are important to understanding some of the shortest path algorithms described later. One of the most basic graph theoretic definitions related to shortest paths is that of a path. Firstly, the notation $u \rightarrow v$ denotes the existence of a directed edge from vertex u to vertex v. Under this notation, $v0 \rightarrow v1 \rightarrow \ldots \rightarrow vl$ represents a directed path of length l, where each vi for $0 \leqslant i \leqslant l$ is a vertex on the path. Here v0 is the first vertex on the path, and vl is the last vertex on the path. A path can alternatively be denoted as an ordering of vertices (v0,v1,v2,...,vl) such that there exists an edge $vi \rightarrow vi+1$ for all $0 \leqslant i \leqslant l-1$. A path whose first and last vertices are the same is called a cycle; that is, a path of the form (v,w1,w2,...,wl,v), where $l \geqslant 0$. One of the simplest graph properties is that of acyclicity. The concept of acyclicity is used throughout this thesis. A graph is acyclic if it does not contain any cycles. A topological ordering (v1,v2,...vk) of k vertices satisfies the property $i < j$ wherever there exists an edge $vi \rightarrow vj$ for any $1 \leqslant i \leqslant k$ and $1 \leqslant j \leqslant k$. As will be seen, a topological ordering of vertices can be used to compute shortest paths more easily. It is possible to compute a topological ordering of the vertices in a directed acyclic graph in linear time. One method is to take the reverse of the postorder of vertices produced by a depth first-search of a nearly acyclic graph. Another kind of graph property is that of planarity. A graph is planar if it can be drawn in a plane without any edges crossing. It has been proved that any planar undirected graph satisfies the inequality $m \leqslant 3n - 6$ for $n \geqslant 3$. Consequently, planar directed graphs satisfy $m < 6n - 12$. Therefore, the number of edges m in a planar graph is O(n). The property of planarity is analogous to that of acyclicity in that shortest paths become easier to compute. A further structural property of graphs is connectivity. A graph is strongly connected if there exists a path from u to v for all pairs of vertices u and v in the graph. A graph that is not strongly connected can be partitioned into a set of maximal strongly connected subgraphs, called strongly connected components (or SC components for short). As will be seen, the property of strong connectivity has also been used to speed up shortest path computations.

III. Algorithm

Computing Shortest Paths by Tree Decomposition:-

This section presents a GSS algorithm which decomposes a graph into trees in order to improve the time complexity required when solving the shortest path problem on nearly acyclic directed graphs. This serves as an introduction to the new algorithm which uses a more general acyclic decomposition. For certain kinds of graphs, the algorithm in this section improves on Abuaiadh and Kingston's algorithm(when used for solving GSS problems), and introduces improvement to Takaoka's algorithm.

- Define IN(v) as the set of vertices u such that there is an edge (u,v) in the graph. Then tree structures in a graph can be identified as follows:
- A root vertex v in a tree structure has $|IN(v)| > 1$ or $|IN(v)| = 0$.
- A non-root vertex v in a tree structure has $|IN(v)| = 1$.

Example of a graph viewed as linked tree strucrure

Such a tree structure is denoted using the notation tree(v) where v is the root vertex of the tree. If there is a directed edge from a vertex in a tree T to a root vertex w of some other tree, then T is a neighbouring tree of w. In special cases, where there exists a ring of vertices in the graph, with each vertex v on the ring having IN(v) = 1, any arbitrary vertex can be chosen as the root vertex of the associated tree. Figure 4.1 illustrates a graph viewed as a set of tree structures. In the simplified view, edges with the same source tree and destination root vertex are represented using a single pseudo-edge. From the simplified view, it is easily seen that in general only one delete min operation per tree structure is necessary. The first step of the new algorithm is to scan each vertex v in the graph to determine root and non-root vertices, according to the value of |IN(v)|. 1 In this description, a root vertex is called a trigger vertex. A trigger vertex triggers shortest path distance updates into other vertices in the tree. The rest of the algorithm consists of two updating passes through the graph. Algorithm 1 gives the first updating pass of the algorithm. This calcu- lates first-tentative shortest path distances d1[v] for vertices in each tree. No delete min operations are performed during this first updating pass. At the beginning of the algorithm, each vertex v has an associated GSS initial distance

Algorithm 1. First Stage of the Tree GSS(generalized single source) Algorithm

/* assume trigger vertices are known */

1. Q =∅;
2. for each vertex v do d1[v] = d0[v];

3. for each trigger vertex u do {
4. add non-trigger vertices in OUT(u) to Q;
5. while there is a vertex v in Q do {
6. remove v from Q;
7. for each vertex w in OUT(v) do {
8. d1[w] = min(d1[w],d1[v] + c(v,w));
9. if w is not a trigger vertex then add w to Q;
10. }
11. }
12. }

d0[v]. The updating of vertices in a tree requires a queue Q to be maintained. The queue can be maintained in either first-in first-out (breadth first search) or last-in first-out (depth first search) order. Alternatively, the algorithm can be implemented as a recursive depth-first search, eliminating the need for the algorithm to maintain a queue. The distance updates in Algorithm 1 are restricted from propagating between trees. This is not strictly necessary for the algorithm to work, but for now it makes the explanation simpler. A first-tentative shortest path distance d1[v] is the shortest distance resulting from the initial distance d0[v] or paths of the form:

(v1,v2,...,vk,v), k ⩾ 1

for which:

d1[v] = d0[v1] + c(v1,v2) + ... + c(vk,v)

With path length defined in terms of the number of edges traversed by the path, this path has length k. The properties of such a path of length k are:

• Each vi, for all 1 ⩽i⩽ k, lies on the same tree T; that is, vi ∈ T for all 1⩽i⩽ k.

• If vertex v is a non-trigger, then it is on the same tree as vertices vi, for all 1 ⩽i⩽ k.

• If vertex v is a trigger vertex, then vertices vi, for all 1⩽ i⩽ k, are on a neighbouring tree of v.

Note that in this restricted algorithm no trigger vertex will be involved in the first-tentative shortest path of another trigger vertex. A trigger vertex can only be updated from as far away as non-trigger vertices in neighbouring trees. At the end of the first updating pass, the following assertions hold:

• For each trigger vertex u, the shortest path to u that can result from non-trigger vertices in neighbouring trees of u has been calculated. This distance is given in d1[u].

Any improvements on d1[u], for any trigger vertex u, must involve a path from another trigger vertex.

Algorithm 2 gives the second updating pass algorithm. For the second up- dating pass, only trigger vertices are involved in the frontier set F and solution set S. At lines 5 and 6, the trigger vertex u that has minimum d[u] is selected and removed from F. Call this the minimum trigger vertex. This vertex is then added to the solution set S. Before the ith iteration at line 5, let the state of the solution set S be:

S = {u1,u2,...,ui−1} (added in this order)

Then, the following theorem applies:

Theorem 1.

1. for trigger vertices uk∈ S, where 1 ⩽ k ⩽i − 1, d[uk] is the shortest distance to vertex uk.

2. for all vertices v ∈tree(uk) and all uk, where 1 ⩽ k ⩽ i−1, d[v] is the shortest distance to vertex v.

Algorithm 2. Second Stage of the Tree GSS Algorithm (Continues from Algorithm1)

1. S = ∅;
2. insert all trigger vertices with nonzero |IN(v)| into F;
3. for each vertex v do d[v] = d1[v];
4. while F is not empty do {
5. select u such that d[u] is the minimum among u in F; /* delete min */
6. remove u from F;
7. add u to S;
8. add u to Q;
9. while there is a vertex v in Q do {
10. remove v from Q;
11. for each vertex w inOUT(v) and not in S do {
12. d[w] = min(d[w],d[v] + c(v,w)); /* If w is a trigger vertex, then a decrease key * operation may occur. */
13. if w is not a trigger vertex then add w to Q;
14. }
15. }
16. }

3. for trigger vertices u ∈ F, d[u] is the distance of the shortest path to u, which consists of an initial path of zero or more non-triggers, followed by zero or more paths through trees tree(v) for trigger vertices v ∈ S, to reach u.

Proof (By induction). Basis i = 1: Assertions 1 and 2 above are automatically true since S is empty. For assertion 3 above, d[u] is correctly computed by Algorithm 4.1 since S is empty. Induction step: Assume the theorem is true for S = {u1,u2,...,ui−1}. If ui is the minimum among trigger vertices in F, then d[ui] is the shortest distance to ui since the distance for a path through any other trigger vertex in F will be longer. In addition, for v ∈tree(ui), the shortest distance d[v] is correctly computed since there is no shorter path to v that goes through other triggers. Finally, for trigger vertices u remaining in F, d[u] will be updated if tree(ui) is a neighbouring tree of u. Therefore, for triggers u remaining in F, the distance of the shortest path that goes through trigger vertices in u1,u2,...,ui is correctly computed since ui and tree(ui) will be the latest possible trigger and tree structure to go through to reach u. Hence, the theorem is true for S ={u1,u2,...,ui}. Let there be a total of n vertices and m edges in the graph. The first updating pass through the graph takes O(m) time. Now assume a Fibonacci heap is used for F. Suppose there are r trigger vertices in the graph, then there will be r delete min operations in the second updating pass, each taking at most O(logr) time, giving a combined worst-case time complexity O(rlogr). The second updating pass also has an O(m) time component, which accounts for each edge traversed, and any decrease key operations. Combining these times, the worst-case time complexity of the entire algorithm is

O(m+rlogr). For the conventional single-source problem, the first updating pass can be simplified to only involve the tree rooted at the source vertex. The GSS algorithm will perform well when a graph is made up of large tree structures; that is, r n. For the same graph, Abuaiadh and Kingston's algorithm could take O(m + nlogn) time to compute GSS since the worst- case value for t is n. The worst-case value of t is not as bad for conventional single-source,2 taking at most O(m + nlogr) time since t is at most r + 1. Applying tree decomposition with Abuaiadh and Kingston's concept of easy vertices produces a hybrid GSS algorithm with a worst-case time complexity of O(m + rlogt), where t is the number of easy trigger vertices resulting from r trigger vertices. This new GSS algorithm can be applied in Takaoka's single source algorithm for acyclic graphs [27] when solving GSS on each SC component. This gives a time complexity of O(m+rlogk), where k is the maximum number of trigger vertices in any single SC component, and r is the total number of trigger vertices in the graph.

IV. Future research

Solving shortest paths on nearly acyclic graphs is still a relatively new research area. There is much potential for further improving on some of the new algorithms that have been presented, and for extending some of the concepts used. There are currently several different measures for acyclicity that allow shortest paths to be solved efficiently. By combining the minimum feedback vertex set and SC decomposition measures, a superior measure is obtained which supersedes all simpler measures. Other ways to measure acyclicity may be discovered in the future. It is hypothesised that there exists a super-measure for acyclicity, which captures all forms of acyclicity contained within a graph. Such a super-measure could provide an efficient shortest path algorithm for any form of nearly acyclic graph. Similar super-measures may even exist for capturing other kinds of graph properties, such as how planar a graph is. Combining such super-measures may lead to a unified framework for solving shortest path efficiently on any kind of graph.

V. References

[1]. Abuaiadh, D. On the complexity of the shortest path problem. PhD thesis, Basser Department of Computer Science, University of Sydney, Australia, July 1995.

[2]. Abuaiadh, D., and Kingston, J. Are Fibonacci heaps optimal? In ISAAC '94 (1994), Lecture Notes in Computer Science, pp. 41–50.

[3]. Abuaiadh, D., and Kingston, J. Efficient shortest path algorithms by graph decomposition. Tech. rep., Basser Department of Computer Science, University of Sydney, Australia, 1994. Technical Report 93-475.

[4]. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[5]. Ahuja, R. K., Mehlhorn, K., Orlin, J., and Tarjan, R. E. Faster algorithms for the shortest path problem. Journal of the ACM 37, 2 (1990), 213–223.

[6]. Cormen, T. H., Leiserson, C. E., and Rivest, R. L. Introduction to Algorithms, 2nd ed. MIT Press, 2001.

[7]. Dantzig, G. B. On the shortest route through a network. Management Science 6 (1960), 187–190.

[8]. Dijkstra, E. W. A note on two problems in connexion with graphs. NumerischeMathematik 1 (1959), 269–271.

[9]. Driscoll, J. R., Gabow, H. N., Shrairman, R., and Tarjan, R. E. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. Communications of the ACM 31, 11 (1988), 1343– 1354.

[10]. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway Hierarchies Star. In: 9th DIMACS Implementation Challenge - Shortest Paths (2006)

[11]. Sanders, P., Schultes, D.: Engineering Highway Hierarchies. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 804–816. Springer, Heidelberg (2006)

[12]. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. In:Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 2008), pp. 13–26. SIAM (2008)

[13]. Schieferdecker, D.: Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries. Master's thesis, Universit¨at Karlsruhe (TH) (2008)

[14]. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Shortest-Path Queries in Road Networks. In: Proceedings of the 9th Workshop onAlgorithm Engineering and Experiments (ALENEX 2007), pp. 46–59. SIAM(2007).

