On the Generation of Short Paths and Minimal Cutsets of the Hierarchical Web Graph

Beth Hayden, Daniel MacDonald

July 21, 2005

Introduction

In the theory of graphs, there exist several algorithms that will find the shortest path between two vertices. Two of these algorithms are due to Dijkstra and Ford. However, an even more general problem would be to develop an algorithm that finds all shortest paths between two vertices in a graph. Furthermore, we may also want to look at minimal ways to cut the edges in a graph such that all shortest paths between two vertices no longer exist. Before we look into this problem, a few definitions are necessary.

Definition (1): An $s-t$ path in a graph $G$ is a sequence of vertices (starting at $s$ and ending at $t$) and edges such that no edge is repeated.

Definition (2): A short $s-t$ path in a graph $G$ is a path between two vertices $s$ and $t$ of length $\leq L$.

Definition (3): A minimal short-cutset in a graph $G$ is a set of edges such that the removal of those edges cuts all short $s-t$ paths.

Main Objectives

We have two major objectives in our project:

1. Generate all short $s-t$ paths in an efficient manner.
2. Generate all minimal short-cutsets efficiently.
1 The Hierarchical Web Graph (HWG)

We concentrated on a specific type of graph, that is, the HWG.

1.1 Structure of the Hierarchical Web Graph

The HWG has a structure that simplifies the problem of generating the short paths. This is true because each 'side' of the graph is really a tree, and this allows for easy enumeration. One such algorithm would be the following (refer to the above example of an HWG for reference on the notion of “side” in the algorithm):

An Algorithm Generating Short Paths in an HWG:

1. Define a graph $G = (V, E)$ and a central vertex $c$.
2. Let $J \subseteq E$, $J \leftarrow \emptyset$, input value for L.
3. Add to $J$ all edges adjacent to $c$.
4. Determine whether or not there exists an $s - t$ path along the given edges (for the initial case, there should be one: connecting $s - c - t$).
5. List the path and label it with its value $a_i$.
6. Begin now on one 'side' (either the $s$ side or $t$ side). Without loss of generality, assume we begin on the $s$ side. For all $e \in J$ on the $s$ side, add edges to $J$ that are incident to $e$. Some of these will connect to $s$.
7. List all newly created $s - t$ paths and assign them values.
8. Continue the process on the $s$ side until all the edges are used, or the path length $a_i > L$.
9. Now consider the other side. Add all edges incident to the edges added at the third step of the algorithm, but now on the $t$ side. Calculate the length $b_j$ of any $c - t$ path that is created.
10. Find all such $a_i$ such that $a_i - 1 + b_j \leq L$. List these paths.
11. Continue this procedure until all edges are used or the path length $b_j > L + 1 - a_i$. 
2 Independence Systems

We now examine an application to our problem based on the theory of independence sets. The generalized notion of an independence set simply requires the following definitions:

**Definition (2.1):** A family of sets $F$ is an independence system if any subset $I'$ of $I \in F$ is also in $F$.

**Definition (2.2):** An independent set $I$ is maximal if there exists no set $I' \in F$ such that $I' \supset I$.

This leads to our application of the independent set:

**Definition (2.3):** A set $I \subset E$ is independent if no short $s - t$ path exists in $I$. Furthermore, $I$ is maximally independent if the addition of any edge in $E - I$ to $I$ creates a short path.

2.1 Some More Notions of Independence

There are a few problems with our independence definition. First, the general maximal independence problem is known to be NP-Complete. However, in a paper by Lawler, Lenstra and Rinnooy Kan (1980), it was shown that if a particular subproblem of the maximal independence set problem can be applied, then the maximal independence set problem can be solved in polynomial time.

The subproblem is called the ‘$I \cup \{j\}$ problem’. So if you have a set $E = \{1, 2, \ldots, n\}$ for which you are trying to find independent sets, you first find all independent sets $I$ which are maximal within $\{1, 2, \ldots, j - 1\}$, and then consider $I \cup \{j\}$ for each $I$. So for each $I' = I \cup \{j\}$, you generate the independence sets, and repeat until $j = n$.

But if we were to just imagine a general planar graph, we would get stuck at the very first step of this problem: determining a proper labeling.

There is no way of systematically labeling the edges so that the algorithm can run properly. This is why we turned to the study of the $HWG$'s. We were able to determine a labeling strategy that is the first step towards solving our problem.
2.2 A Labeling Procedure for the $I \cup \{j\}$ Problem

To make it clear: we are concerned only with the labeling of the edges adjacent to $s$ and $t$. The other edges must be labeled, but can be done in any arbitrary way.

The $HWG$ basically consists of two trees that share a common root. As such, we can order the nodes in each tree based on how far they are from the central vertex. These different 'levels' will determine how we label our edges.

**The Labeling Procedure**

1. Sort the tree so that all nodes in a certain level (i.e., the same distance from $c$) are seen as in the same group.
2. Sort the group into two subgroups based on what side the nodes are on (the $s$ or $t$ side).
3. The first group one labels is the group of nodes at the greatest distance from the central vertex. Each subsequent group moves to a lower level (i.e., closer to $c$).
4. Label arbitrarily in the subgroup’s space, then build onto the other subgroup, and then follow to the next group’s subgroup.
5. Label the corresponding edge that is adjacent to the labeled vertex and also $s$ or $t$ with the number chosen from (3).

**Proof that the Labeling Procedure Works**

Assume the labeling algorithm goes up to all edges in a specific group (and all lower groups) except one. Let us call that the $j^{th}$ edge. When we then compute all shortest paths along $j$, all other short paths will have already been covered (i.e., no additional short paths exist) in that hierarchy. Thus, the procedure is consistent in labeling according to hierarchy.

Assume now that we have completed some number of labels on one of the two subgroups. If we were to start labeling the other subgroup before completing the first, we would be introducing a smaller path length too soon. For example, if edge $e$ in our graph belongs to a subgroup that we have labeled with the length 5, then we introduce a vertex $f$ from the other subgroup, the minimum length would decrease to 4 before we were done with the subgroup that are all elements of paths length 5 or more. It must be the case then that we complete the labeling of all elements of one subgroup before moving to the next one. Thus, the procedure is consistent in labeling according to side ($s$ or $t$).
3 The Next Step In the \( I \cup \{j\} \) Problem

Now that we have determined a labeling procedure, we need to apply it to the problem. The following algorithm brings us to the next step:

1. Given a graph \( G = (V, E) \), \( E = \{1, 2, \ldots, n\} \).
2. Let \( j \leftarrow \emptyset \), user defines \( L \).
3. For \( j \leftarrow j + 1 \), find all paths that travel along \( j \) of length \( \leq L \).
4. List all such paths.
5. Do Function: Boolean Test.
6. Repeat until \( j = n \).

The function mentioned in (5) is actually the bulk of the program, and will be explained below.

3.1 The Boolean Test

The Boolean test occurs at each value of \( j \), after generating each short path of length \( \leq L \) that lies on \( j \). What it entails is a conversion from a simple list of short paths to a tree, where each node represents an edge of the short path, and the root vertex is \( j \).

Further, this tree and the short paths can be converted to a boolean function, or more specifically, a Conjunctive Normal Form (CNF). For example, if the set of short paths is \( \{\{1, 3, 4, 5\}, \{1, 3, 4, 6\}, \{1, 3, 7\}\} \), this can be converted to a boolean expression: \((1 \lor 3 \lor 4 \lor 5) \land (1 \lor 3 \lor 4 \lor 6) \land (1 \lor 3 \lor 7)\). This expression, when simplified, will give you exactly the minimal cutsets, each separated by a disjunction. For example, the above CNF will be converted to the Disjunctive Normal Form (DNF) of:

\[
1 \lor 3 \lor (4 \land 7) \lor (5 \land 6 \land 7)
\]

Obviously the CNF has some repeats in it, and thus computing it could be made more efficient if we were to cut those out, and this is where the tree comes in. We will show that a specific method of computing the conjunction,
according to each node’s placement on the tree, will reduce the complexity of the Boolean test.

The Method

The idea behind our method is to reduce the possible terms that may arise as we convert the CNF to the DNF. We will use the fact that many of the short paths generated for a certain $j$ share some common edges to help us do this.

We will first consider the part of the tree that splits off from the branching node farthest from the source node $j$. (EXAMPLE?) We will compute the conjunction for this part of the tree. The resulting disjunction gives us the minimal cuts for this section, and we can rewrite this section of the tree as one single path, with each edge corresponding to one minimal cutset.

We continue the procedure by moving to the new branching node farthest from $j$ (remember, we turned the previous branch into a single path, so that node is no longer a branching node). We do the appropriate conjunction, and then convert that branch into a single path, with each edge corresponding to one minimal cut. The procedure continues until all branching nodes have been considered, and our final result is one path. The final step is to form a disjunction of each edge in this path, which gives us the set of cuts for this $I \cup \{j\}$.

Proof of Complexity

We claim that this method reduces the complexity of the Boolean test. In the worst case, the complexity will be $O(n \cdot M)$, where $n$ is the number of nodes of the tree and $M$ is the number of minimal cuts. We will prove by induction that our method keeps the complexity of the Boolean test at or below $O(n \cdot M)$.

- For 0 branching nodes: The tree is a single path. Thus we can simply form a disjunction of each node, with total work = $O(n) \leq O(n \cdot M)$.

- We assume that for $k$ branching nodes, total work $\leq O(n \cdot M)$. Now we consider $k + 1$ branching nodes. We will call the graph with $k + 1$ branching nodes $G$. We will say that $b$ paths branch out from this $k+1^{th}$ node, and that these $b$ paths have lengths $l_1, \ldots, l_b$, respectively. Now we can define $X = \{1, 2, \ldots, l_1\} \times \ldots \times \{1, 2, \ldots, l_b\}$. There are $|X|$ ways to cut the paths along the $k + 1^{th}$ node. Thus, we can rewrite the $k + 1^{th}$ branch as the single path, with $|X|$ edges. We will call this revised graph $G'$. Note that the minimal cuts of $G$ and the minimal cuts of $G'$
have a one-to-one correspondence, so the number of cuts is maintained. Note also that $|G'| = |G| + |X| \leq |G| + M$, because $|X| \leq M$. We say that $|X| \leq M$ because every cut in $M$ includes exactly one element of $X$ (making the cut minimal), and every element of $X$ is included in some cut in $M$ (because $M$ includes all possible cuts). But, if $|X| > M$, then there is some element of $X$ that is not included in a cut in $M$, which contradicts our previous statement. Thus, in our $G'$ graph, we again have $k$ branching nodes, and the overall size is only greater than that of $G$ by a constant, $M$. Thus the overall work is still $\leq O(n \cdot M)$, based on our inductive hypothesis that for $k$ branching nodes, total work $\leq O(n \cdot M)$.

3.1.1 Proof of Maximality Within Boolean Test

We now want to prove that with this boolean test, maximality through $j$ is obtained for each $j$. As a reminder: an independent set $I$ is maximal if there exists no $I'$ such that $I' \supset I$. In terms of our problem, that means that for a given set of edges that contain no short $s-t$ path (there are many for each $j$), there is no additional edge that we can add to that $I$ such that still no short paths exist.

The CNF contains a boolean operator on each short path that goes through $j$. Assume to the contrary, i.e. there exists an $e \in (E - I)$ such that $I \cup \{e\}$ is maximal. Then $e$ would be an element of our boolean function for that $j$, and so would already be included in our DNF, and would hence be a part of some $I_j$.

4 The complete $I \cup \{j\}$ Algorithm

We have now defined and proved many subsections of the main algorithm discussed at the beginning of the paper (at 2), and we now summarize by combining the different properties to present the complete version of the $I \cup \{j\}$ algorithm when applied to our problem.

1. Given: A Hierarchical Web Graph $G = (V, E), E = \{1, 2, \ldots, n\}, J \subset E, J \leftarrow \emptyset$, a user-defined $L$.

2. Perform short path algorithm defined in 1.1 to acquire a list of all short paths under our threshold $L$.

3. Perform the labeling procedure outlined in (2.2) on $G$. 


4. For($j = 0 \ldots n$), Do:
   - $j \leftarrow j + 1$
   - Search list created in (2) to find short paths.
   - Perform the Boolean Test outlined in 3.1

5. Output: each separate conjunction in the DNF that is produced by the boolean test.