An Algorithm for Finding Two Node-Disjoint Paths in Arbitrary Graphs

Mehmet Hakan Karaata

Department of Computer Engineering, Kuwait University, Safat, Kuwait

Given two distinct vertices (nodes) source $s$ and target $t$ of a graph $G = (V, E)$, the two node-disjoint paths problem is to identify two node-disjoint paths between $s \in V$ and $t \in V$. Two paths are node-disjoint if they have no common intermediate vertices. In this paper, we present an algorithm with $O(m)$-time complexity for finding two node-disjoint paths between $s$ and $t$ in arbitrary graphs where $m$ is the number of edges. The proposed algorithm has a wide range of applications in ensuring reliability and security of sensor, mobile and fixed communication networks.

ACM CCS (2012) Classification: Networks $\rightarrow$ Network algorithms $\rightarrow$ Data path algorithms
Mathematics of computing $\rightarrow$ Discrete mathematics $\rightarrow$ Graph theory $\rightarrow$ Paths and connectivity problems

Keywords: disjoint paths, distributed systems, fault tolerance, network routing, security

1. Introduction

Given two distinct vertices (nodes) $s$ and $t$ of a graph $G = (V, E)$ with set of vertices $V$ and set of edges $E$, paths $P_1$ and $P_2$ from vertex $s$ to vertex $t$ are said to be node-disjoint if paths $P_1$ and $P_2$ do not contain any common vertices except for the endpoints.

The two node-disjoint paths problem is to find two node-disjoint paths from source vertex $s$ to target vertex $t$ [2]. The two node-disjoint paths problem is a fundamental problem with abundant number of applications in diverse areas including VLSI layout [3], [4], [5], reliable network routing [6], secure message transmission [7], [2], and network survivability [8]. For instance, perfectly secure transmission can be implemented using node disjoint paths by breaking up data into several shares and sending them along the disjoint paths. This simple expedient makes it difficult for an adversary with bounded eavesdropping capability to intercept a transmission or tamper with it. In addition, the same crucial message can be sent over multiple disjoint paths in networks that are prone to message losses to avoid omission failures or, in the presence of faults, information on re-routing of traffic along the network can be provided. Recently, [8] introduced a new strategy for using network coding over p-Cycles to provide 1 + N protection against single link failures in optical hypercube networks. Protection paths and cycles are commonly used in optical networks to enhance performance and reliability [9], [10], [11], [12].

The two node-disjoint path problem and its variations are fundamental and extensively studied in graph theory. Algorithms to find edge-disjoint paths were proposed in [13], [14], [15], [16], [17], [3]. Ford and Fulkerson [18] proposed an $O(m)$-time algorithm to find edge-disjoint paths between two nodes. Node-disjoint paths can also be computed in $O(m)$-time using the same method after applying a graph transformation with nodesplitting. Later, Suurballe-Tarjan [19], [13] and Bhandari [20] proposed algorithms that can be used to solve both the edge and the node-disjoint paths problems with $O(m + n \log n)$ time complexity based on the same method (using Dijkstra's algorithm implemented using Fi-
Our proposed algorithm has a very simple basis given as a simple lemma and adopts an entirely different approach. The node disjoint paths algorithms based on Maximum-Flow computation such as Ford-Fulkerson and Suurballe-Tarjan [19], [13] involve a number of phases after the discovery of a shortest path between two endpoints. First, the initial graph is transformed into a new graph where arc weights and directions are recomputed. Second, each node on the shortest path is split and additional arcs are introduced leading to a new graph. Third, another shortest path is constructed between the two endpoints in the newly obtained graph. Fourth, paths common between the two constructed disjoint paths and the cycles that do not contain both the endpoints are removed prior to constructing the two disjoint paths. On the other hand, the proposed algorithm is not Maximum-Flow computation based and the basis of the algorithm is given in the form of a single lemma. It requires only the identification of link paths prior to the construction of the disjoint paths. Therefore, the proposed algorithm is simpler and more understandable than those based on Maximum-Flow computation. In addition, most solutions available in the literature [23], [24], [19], [13] suffer from being overly complex or being unfit for use in distributed applications. These drawbacks primarily stem from the adaptation of solutions to other fundamental problems such as edge-disjoint paths, fundamental cycles (and kernel) and network flow to the solution of the node-disjoint paths problem. In addition, many of these solutions require the discovery of some global properties of the entire graph instead of local properties. These impose severe restrictions to the adaptation of the solutions to distributed applications. Therefore, it is not clear how these solutions could be used to devise a distributed solution to the node-to-node disjoint paths problem.

In this paper, we present a novel $O(m)$-time sequential algorithm for finding two node-disjoint paths between two distinct vertices $s$ and $t$ in arbitrary graphs. The proposed algorithm is based on a new method of identifying link paths, which is a simple property of general
graphs given as a single lemma in the paper. In addition, the proposed algorithm is designed in a way to ease its transformation to a distributed implementation. As a result, the proposed approach is well suited for devising distributed and fault-tolerant solutions to the problem. It is anticipated that this work will initiate further work in the area of distributed and fault-tolerant computing.

The paper is organized as follows. Section 2 presents the basis of the algorithm and some required notations for the formal description of the algorithm. Section 3 presents the two node-disjoint paths algorithm. In Section 4, we provide a correctness proof and the proofs of the time complexity bound of the algorithm. We conclude the paper in Section 5 with some final remarks.

2. Basis of Algorithm

In this section, we present the basis of the proposed solution. Let $G = (V, E)$ be a graph with two distinct vertices $s, t \in V$ such that $G$ contains two node-disjoint paths between $s$ and $t$. We first define link paths to facilitate the description of the basis of the algorithm.

**Definition 1.** Let $P$ be a path between $s$ and $t$ and $d_s(v)$ the distance of vertex $v$ on $P$ from vertex $s$. A link path of path $P$ in $G$ is a path disjoint from $P$ except for its endpoints that extend from a vertex $o$ on $P$ to a vertex $w$ on $P$ such that $w$ is the farthest vertex reachable from $o$, i.e., the distance from $o$ to $w$ is maximal. A vertex is said to be reachable from another vertex if the graph contains a path connecting them.

Let us now define $LP = P_1, P_2, ..., P_k$ to be a maximal sequence of link paths of path $P$ in $G$, each of which has its endpoints on $P$ such that the following four conditions are satisfied by $LP$.

(i) $P_1$ is the first link path with origin $o_1(=s)$ and terminus $w_1$.

(ii) Each link path $P_{i-1}$ where $1 < i \leq k$ has a successor link path $P_i$ that extends from its origin $o_i$ to its terminus $w_i$ such that

$$
\begin{align*}
\begin{cases}
    d_s(o_i) < d_s(o_{i+1}) < d_s(w_i) & (i = 2) \\
    d_s(w_{i-2}) \leq d_s(o_i) < d_s(w_{i-1}) & \text{if } 2 < i \leq k
\end{cases}
\end{align*}
$$

holds.

(iii) For each $i$, $1 < i \leq k$, vertex $o_i$ on $P$ is selected to maximize $d_s(w_i)$.

(iv) The terminus of the last link path $P_k$ is target $w_k = t$.

$P(v, w)$ denotes the subpath of $P$ with origin $v$ and terminus $w$. $P(v, w)$, $P[v, w)$, and $[P[v, w]$ denote the same path excluding the terminus, origin, and both the origin and the terminus of the subpath $P(v, w)$, respectively. Let $P_1, P_2, ..., P_k$ be a sequence of link paths in $G$ for a shortest path $P$ between $s$ and $t$. Also, let $o_i$ and $w_i$ for $0 < i \leq k$ denote the origin and the terminus of link path $P_i$. Let $P'' = P_1 P[w_1, o_3] P_3 P[w_3, o_5] P_5 ... P_{2l+1}$ where $2l + 1 < k$ and $2(l + 1) + 1 > k$, and $P'' = P(s, o_2) P_2 P[w_2, o_4] P_4 ... P_{2l+1}$ where $2l + 1 > k$ and $2(l + 1) + 1 > k$ are two paths in $G$. Using the above definitions, we define paths $P^1$ and $P^2$ as follows. If $k$ is odd, $P^1 = P''$ and $P^2 = P'' P[w_{k-2}, 1, t)$. Otherwise, $P^1 = P' P[w_{k-2}, 1, t)$ and $P^2 = P''$.

The following lemma establishes the basis of the proposed algorithm.

**Lemma 1.** Let $P$ be an arbitrary path between two arbitrary but distinct vertices $s$ and $t$ in $G = (V, E)$. Graph $G$ contains two node-disjoint paths $P^1$ and $P^2$ between endpoints $s$ and $t$ if and only if there exists a maximal sequence of link paths $P_1, P_2, ..., P_k$ in $G$ for $P$ satisfying the four conditions for being a sequence of link paths.

**Proof.** For the "if" direction, we prove the contrapositive. We assume that the sequence of link paths $LP = P_1, P_2, ..., P_k$ does not exist and we show that two disjoint paths do not exist. Observe that the sequence $LP = P_1, P_2, ..., P_k$ does not exist if at least one of link paths $P_i \leq i \leq k$ does not exist. First, if the link path $P_i$ does not exist, then the successor of $s$ on $P$ is common for all the paths starting from $s$. Now, consider the case where link paths $P_1, P_2, ..., P_k$ do not exist. An analogous to the above, the terminus of $P_i$ is common for all the paths starting from $s$. Thus, in both cases, two disjoint paths between $s$ and $t$ cannot exist, hence, the result. For the "only if" direction, we prove by construction. We assume that if the sequence of link paths $LP = P_1, P_2, ..., P_k$ exists, then two disjoint
four phases executed in sequence, namely the
construction of link-paths. Observe that the
sequence of link paths with odd subscripts $P_1$, $P_3$, ..., $P_{2l-1}$ where $2l + 1 = k$ or $P_1$, $P_3$, ..., $P_{2l}$ where $2l = k$ (depending on whether $k$ is odd or even) are used to construct path $P_1$, whereas the maximal sequence of link paths with even subscripts $P_2$, $P_4$, ..., $P_{2l}$ where $2l = k$ are used to construct $P_2$. In each sequence, for every pair of consecutive paths in the sequence, such as $P_1$ and $P_3$ in the sequence of odd subscripted link paths, the terminus of each link path and the origin of the subsequent link path are connected by a segment of $P$ between the terminus of the first link path of the pair and the origin of the subsequent link path on $P$ to construct a disjoint path.

Figure 1 depicts a graph with source vertex $s$, target vertex $t$ and its link paths $P_1, P_2, P_3$ and $P_4$ for shortest path $P$ (shown by a thick line) from $s$ to $t$ to illustrate the above approach. Note that, although not shown, we assume that each of the link paths $P_1, P_2, P_3$ and $P_4$ contain multiple vertices on them other than their endpoints to make $P$ a shortest path in $G$. Observe that link path $P_2$ is a successor of link path $P_1, P_3$ is a successor link path of $P_2$, and so on. Figure 2 shows the same graph shown in Figure 1 with the node-disjoint paths $P_1$ and $P_2$ identified to illustrate the approach to construct the node-disjoint paths. Observe that the figure depicts node-disjoint path $P_1$ shown with thicker lines and $P_2$ shown with thick lines. Notice that the even subscripted link paths $P_2$ and $P_4$ are used to construct node-disjoint path $P_1$, whereas, the even subscripted link paths $P_2$ and $P_4$ are used to construct node-disjoint path $P_3$. Also notice that subpaths $P[w_1, o_3]$ and $P[w_3, t]$ of $P$ are added to the link paths $P_1$ and $P_3$ to construct $P_1$. Similarly, subpaths $P(s, o_2)$ and $P[w_2, o_4]$ of $P$ are added to the link paths $P_2$ and $P_4$ to construct $P_2$.

3. Algorithm

In this section, we provide a formal description of the algorithm to find two disjoint paths.

Let $G = (V, E)$ be a simple undirected graph with two distinct vertices $s, t \in V$ such that two node-disjoint paths exist between $s$ and $t$. Let $P = (V_P, E_P)$ be a shortest path in $G$ from $s$ to $t$. For the sake of brevity, we do not present the algorithm to construct $P$. Instead, we assume that path $P$ is constructed using an algorithm
An Algorithm for Finding Two Node-Disjoint Paths in Arbitrary Graphs

available in the literature, such as [26]. We also assume that for each vertex \( v \) on \( P \), the \( d_s \)-value \( d_s(v) \) of vertex \( v \) denoting its distance from source vertex \( s \) is also computed and available to our algorithm. We present our proposed algorithm in the next four subsections, where each subsection contains part of the algorithm implementing a phase of the algorithm.

3.1. Forest Construction

In the first phase of the algorithm, for each vertex \( r \) on \( P \), a maximal BFS tree \( T_r = (V_T, E_T) \) called link tree rooted at \( r \) is constructed in \( G \). The tree \( T_r \) satisfies that each vertex \( w \) in \( G \) is a descendant of \( r \) in \( T_r \) iff \( w \) is reachable from \( r \) through a path in \( G \) not on \( P \) and \( d_s(r) \) is minimum. That is, each vertex \( w \) in \( G \) is a descendant of root \( r \) on \( P \) iff it is reachable from \( r \) via a path in \( G \) not on \( P \) such that \( r \) is the root among all such roots on \( P \) that makes \( d_s(r) \) minimum. Forest \( F \) is defined as the set of all link trees rooted at nodes on \( P \).

Prior to presenting the algorithm to construct forest \( F \), we need the following definitions. Variables \( V_T \) and \( E_T \) denote a set of vertices and edges included in forest \( F \) constructed thus far, respectively. Variable \( Q \) denotes a queue of vertices that have not been visited yet. Variable \( m(v) \) for each vertex \( v \in V \) denotes whether or not vertex \( v \) has been visited in the process of constructing \( F \). Variable \( p(v) \) for each vertex \( v \in V \) denotes the parent of vertex \( v \). Ordered set \( P = s, v_1, v_2, \ldots, v_k, t \) denotes a shortest path in \( G \) from vertex \( s \) to vertex \( t \). Statement \textbf{for each} \( v \in P \) \textbf{in order} executes its body for each value \( v \) assigned in order from ordered set \( P \). If phrase \textbf{in reverse order} is used instead of \textbf{in order}, the body is executed for each element of the ordered set \( P \) in reverse order. A variation of the statement \textbf{for each} \( v \in P \) \textbf{if} \( \text{predicate} \) \textbf{ensures that the body is executed for those} \( v \) \text{that satisfy} \text{predicate} \text{. Variable} \text{pred}(v) \text{denotes the predecessor of vertex} \( v \) \text{on} \( P \). The algorithm requires the following input parameters: set of vertices \( V \), set of edges \( E \) in \( G \), and an ordered set of vertices of a shortest path \( P \) between \( s \) and \( t \).

The implementation of the first phase of the algorithm to construct a maximal BFS forest \( F \) of \( G \) with the aforementioned property is given with Algorithm 1.

3.2. Discovery of the Farthest Reachable Vertex on the Shortest Path

The objective of the second phase is to allow each vertex \( r \) on \( P \) to discover the \( d_s \)-value of the terminus vertex \( w \) with the largest \( d_s \)-value (if it exists) of potential link paths originating at \( r \). A potential link path is a path in \( G \) disjoint from \( P \) except for its endpoints which are on \( P \).

To implement the objective of the second phase, each vertex \( v \) in \( V_T \) maintains a variable \( l(v) \) that stores the maximal \( d_s \)-value among reachable vertices on \( P \) from \( v \) through a potential link path if it has such a reachable vertex(vertices), and 0 otherwise. This is implemented as fol-
lows. First, for each vertex in $V_F$, the $l$-value of each vertex is assigned 0. After that, in a bottom up manner, starting from leaves, each vertex computes its $l$-value as the maximum of $l$-values of its children in $V_F$ and its neighbors' $d_r$-value on $P$. When the $l$-value is computed for a vertex $v$, it discovers the farthest reachable vertex on $P$ from $v$ through a segment of link path originating at vertex $r$ on $P$ (as $l(v)$ represents the $d_r$-value of such vertex on $P$). The computation of the $l$-values in $T_r$ marks link paths originating at $r$ as maximal paths with origin $r$ on the vertices whose $l$-values are equal to $l(r)$. Algorithm 2 provides an implementation of the second phase of the algorithm.

Thus far, we presented the first two phases of the algorithm where each vertex $r$ on $P$ discovers the $d_r$-value of the farthest vertex on $P$ reachable from $r$ via a link path. In addition, each vertex $r$ on $P$ marks the link path originating at $r$. These link paths (or subset of them) will be used to construct the two disjoint paths in the later phases.

### 3.3. Identification of Link Paths

We now present the third phase of the algorithm to identify origins and terminuses of all link paths in $LP = P_1, P_2, ..., P_k$ using information collected in the previous two phases. For that purpose, first, vertex $s = o_1$ is identified as the origin of the first link path $P_1$. We know that $l(s)$ contains the $d_{s}$-value of the farthest vertex from $s$ on $P$ reachable via a link path. The vertex with $d_{s}$-value equal to $l(s)$ is identified as the terminus $w_1$ of $P_1$. Then, in order to find the origin of $P_2$, the vertex with the largest $l$-value in the interval $[s, w_1]$ is identified as the origin $o_2$ of $P_2$. Observe that $l(o_2)$ contains the $d_{s}$-value of the farthest vertex from $o_2$ reachable via a link path which is the terminus $w_2$ of $P_2$. Then, for each link path $P_i$, $2 < i \leq k$, the origin $o_i$ of link path $P_i$ is identified as the vertex with the largest $l$-value on $P[w_{i-2}, w_{i-1}]$, while the terminus $w_i$ of $P_i$ is identified as the vertex whose $d_{s}$-value is equal to $l(o_i)$. For example, origin $o_3$ of $P_3$ is the vertex that has the largest $l$-value among vertices on $P[w_1, w_2]$ and terminus $w_3$ of $P_3$ is the vertex whose $d_{s}$-value is equal to $l(o_3)$.

Figure 3 illustrates the approach adopted by the third phase of the algorithm. In the figure, the numbers above the vertices denote the $d_{s}$-values, whereas the numbers below the vertices denote the $l$-values of the corresponding vertices. Since the $l$-value of $s$ is 2, $P_1$ extends from $s$ to $w_1$. The origin of $P_2$ is $o_2$ since $o_2$ has the largest $l$-value of 4 among vertices on $P[0, 2]$. Similarly, since $o_3$ has the largest $l$-value of 6 among the vertices on $P[2, 4]$, the origin of $P_3$ is $o_3$, and so on. We now describe the implementation details of the above approach. For each vertex $v \in V_F$, three integer variables $tc(v), m(v)$, and $or(v)$ are maintained by the algorithm. The two variables $tc(v)$ and $m(v)$ are used to hold the $d_{s}$-value of current and next terminuses, respectively, of link paths as the vertices on $P$ are traversed starting from $s$ towards $t$.

For each $i$, $1 < i \leq k$, $m$-values are used to find the largest $l$-value encountered thus far among vertices on $P[w_{i-1}, w_i]$ (or from the origin of $P_i$ to its terminus for $i = 2$) towards $t$ on $P$. Whereas, $tc$-values are used to propagate the largest $l$-value found between $w_{i-1}$ and $w_i$ on $P$ (using $m$-values), from terminus $w_i$ to terminus $w_{i+1}$ on $P$.

The reason for using two variables to find and propagate the largest $l$-value from the origin of a link path to its terminus is as follows. Observe that for $i$, $1 < i \leq k$, we need to separately propagate the largest $l$-value on $P[w_{i-1}, w_i]$ that was found on $P[w_{i-1}, w_{i+1}]$ and find the largest $l$-value found so far on $P[w_i, w_{i+1}]$. Propagating the first value is necessary for identifying $w_{i+1}$, while finding the second value is required for identifying $w_{i+2}$. Since the propagating and finding the values takes place on the same interval and they depend on each other, the discovery and the propagation on both cannot be implemented using a single integer variable. Therefore, we use two variables, namely $m$ and $tc$ for each vertex to implement the discovery and the propagation of the largest $l$-values.

The $tc$ and $m$-values are computed in the following manner. The $m$-value of node $s$ and upon its discovery, $m$-value of the terminus $w_i$ of each link path $P_i$, $1 \leq i \leq k$, is assigned 0. Then, every other vertex assigns to its $m$-value the largest of its predecessor’s $m$-value and its $l$-value. Upon completion of these assignments, for each $i$, $1 < i \leq k$, $m(pred(w_i))$, where $w_i$ is the terminus of link path $P_i$, contains the largest $l$-value among vertices on $P[w_{i-1}, w_i]$ (or $P[s, w_i]$ for $i = 1$). To discover the terminus of each link path
Algorithm 1. Forest construction.

1. procedure TWODISJOINTPATHALGORITHM($V, E, P$)
2. boolean $m(v) :=$ false for each vertex $v \in V$;
3. vertexId $p(v)$ for each vertex $v \in V$;
4. vertexSet $V_T = \emptyset$;
5. edgeSet $E_T = \emptyset$;
6. queue $Q = \emptyset$;
7. for all $v \in P$ in order do
8. $V_T := V_T \cup \{v\}$;
9. $m(v) :=$ true;
10. addQueue($Q$, $v$);
11. while $\neg$empty($Q$) do
12. $x :=$ remQueue($Q$);
13. if $\{x \in P \rightarrow ds(x) - ds(v) = 1\}$ then
14. pred($x$) = $v$;
15. end if
16. for all $w \in V \setminus P | \{x, w\} \in E \land \{x, w\} \notin E_T \land m(x) \land \neg m(w)$ do
17. addQueue($Q$, $w$);
18. $m(w) :=$ true;
19. $p(w) = x$;
20. $V_T := V_T \cup \{w\}$;
21. $E_T := E_T \cup \{x, w\}$;
22. end for
23. end while
24. end for
25. end procedure

Algorithm 2. Discovery of the farthest reachable vertex on the shortest path.

1. integer $l(v)$ for each vertex $v \in V_T$;
2. $m(v) :=$ false; $l(v) = 0$; for each vertex $v \in V_T$;
3. for all $v \in V_T | \neg m(v) \land \forall \{v, w\} \in E_T \{\{p(w) = v \rightarrow m(w)\}$ do
4. $l(v) := \max \{\max_{\forall \{v, z\} \in E_T \land z \in P} \{v, w \in P - d_{\{w\}} - d_{\{v\}} > 1\} \{d_{\{z\}}, \max_{\forall \{v, z\} \in E_T \land p(z) = v \land z \notin P} \{l(z)\}, 0\};$
5. $m(v) :=$ true;
6. end for

Figure 3. The origins and the terminuses with their $d_{\{\}}$ and $l$-values of its link paths of a graph for source $s$ and target $t$ are shown.
of the terminuses of the link paths), vertices on the link path\( \Pi_i \), \( 1 \leq i \leq k \), assigns \( \mathtt{tn}(\mathtt{pred}(w_i)) \) to \( \mathtt{tc}(w_i) \). Then, each of the consecutive vertices assigns its \( \mathtt{tc} \)-value, the \( \mathtt{tc} \)-value of its predecessor in \( P \). This propagation continues until encountering the (terminus) vertex whose \( d_i \)-value is equal to the propagated value. That is, each terminus \( w_i \), \( 1 < i \leq k \), of a link path is discovered by node \( w_i \) on \( P \) upon discovering that \( d_i(w_i) = \mathtt{tc}(\mathtt{pred}(w_i)) \) holds. When \( m \) and \( \mathtt{tc} \)-values of all the vertices on \( P \) are computed, each vertex whose \( d_i \) and its predecessor's \( \mathtt{tc} \)-value are equal is identified as a terminus vertex of a link path. In this manner, the \( \mathtt{tc} \) and \( \mathtt{tn} \)-values of the vertices on \( P \) are computed in order, and the terminuses of the link paths in \( L_P \) are identified one after the other.

We now describe how variable \( \mathtt{or}(v) \) is used to identify origins of link paths. Upon discovering all terminuses of link paths, observe that predecessor of \( w_i \) stores in \( \mathtt{tn}(\mathtt{pred}(w_i)) \) the largest \( l \)-value found in the interval \( [w_i-1, w_i] \), for \( 2 < i < k \). In order to find origin \( o_i \), \( \mathtt{tn}(\mathtt{pred}(w_i)) \) is assigned to \( \mathtt{or}(\mathtt{pred}(w_i)) \) and the \( \mathtt{or} \)-value is propagated toward \( s \) until meeting a vertex on \( P \) with \( l \)-value equal to the propagated \( \mathtt{or} \)-value. This vertex is identified as the origin \( o_i \) of \( P \). This procedure is applied to all vertices on \( P \) starting from \( t \) until reaching \( s \) and identifying the origins of link paths.

Next, we describe the details of identifying the origins of link paths. After all \( \mathtt{tc} \) and \( \mathtt{tn} \)-values of vertices on \( P \) are computed (and the identification of the terminuses of the link paths), \( \mathtt{or} \)-values of vertices on \( P \) are computed in reverse order of vertices in \( P \) as follows. First, \( \mathtt{or}(t) \) is assigned \( 0 \). In reverse order of vertices in \( P[w_k-1, w_k] \), each vertex assigns \( 0 \) to its \( \mathtt{or} \)-value when it copies its successor's \( \mathtt{or} \)-value. In this process, upon identifying itself as a terminus vertex by discovering that \( \mathtt{tn}(w_{k-1}) = 0 \), vertex \( w_{k-1} \) assigns \( \mathtt{tn}(\mathtt{pred}(w_{k-1})) \) to \( \mathtt{or}(w_{k-1}) \). Then, the value in \( \mathtt{or}(w_{k-1}) \) is propagated towards \( s \) in reverse order of vertices as vertices in ordered set \( P[w_{k-2}, w_{k-1}] \) copy the \( \mathtt{or} \)-value of their successors to their \( \mathtt{or} \)-values. The propagation of the value continues until vertex \( o_{k-1} \) on \( P[w_{k-2}, w_{k-1}] \) such that \( \mathtt{lk}(o_{k-1}) = \mathtt{or}(o_{k-1}) \) holds. Observe that the vertex on subpath \( P[w_{i-2}, w_{i-1}] \) with the largest \( l \)-value is the origin \( o_i \) of link path \( P \). In order for vertex \( o_i \), \( 1 \leq i < k \), to discover that it is the origin of \( P \), the \( \mathtt{tn} \)-value of \( \mathtt{pred}(w_{i-1}) \) containing the largest \( l \)-value on \( P[w_{i-2}, w_{i-1}] \) needs to be propagated towards \( s \) until the vertex whose \( l \)-value is equal to \( \mathtt{tc}(\mathtt{pred}(w_{i-1})) \) is encountered using the \( \mathtt{or} \)-values. For that purpose, first, the largest \( l \)-value on subpath \( P[w_{i-1}, w_i] \) stored in \( \mathtt{tn}(\mathtt{pred}(w_i)) \) is assigned to \( \mathtt{or}(w_i) \). Subsequently, this value is propagated using the \( \mathtt{or} \)-values of vertices on subpath \( O(o_i, w_{i-1}) \) towards \( s \) until the vertex \( o_i \) with \( l \)-value is equal to the propagated value. Then, vertex \( o_i \) whose \( \mathtt{or} \)-value is equal to its \( l \)-value is identified as the origin of link path \( P \). The above approach is implemented through the following actions. For each vertex \( v \) on \( P \) in reverse order, if \( v = \mathtt{pred}(w_i) \), i.e., \( v \) is the predecessor of terminus \( w_i \) of a link path \( P \), for \( 1 < i < k \), the value \( \mathtt{tn}(\mathtt{pred}(w_i)) \) is assigned to \( \mathtt{or}(w_i) \), if \( v = \mathtt{pred}(o_i) \), i.e., \( v \) is the predecessor of origin \( o_i \) of link path \( P \), zero is assigned to \( \mathtt{or}(v) \), and otherwise, \( \mathtt{or}(\mathtt{suc}(v)) \) is assigned to \( \mathtt{or}(v) \).

Figure 4 illustrates the usage of \( \mathtt{tn} \) and \( \mathtt{tc} \)-values for the propagation of the \( l \)-values and the discovery of terminuses of link paths. In Figure 4, the numbers below the vertices denote the \( d_i \)-values of the vertices, whereas, the numbers above the vertices denote the \( l \)-values of the vertices. Vertices \( s, t \), and those that are terminuses of link paths are denoted by filled circles to indicate the vertices whose \( \mathtt{tn} \)-values are \( 0 \). Each row of arrows in the figure denotes the propagation of values using variables \( \mathtt{tc}, \mathtt{tn} \), or \( \mathtt{or} \)-value direction in which the computation of a variable is carried out. The top row of arrows in the figure illustrates the computation of the \( \mathtt{tn} \)-values, whereas the second and the third rows of arrows illustrate the computation of the \( \mathtt{tc} \) and \( \mathtt{or} \)-values, respectively, of the vertices on \( P \). The number above each arrow denotes the value assigned to \( \mathtt{tc}, \mathtt{tn}, \) or \( \mathtt{or} \)-values of the vertices in the subpath of \( P \) above the arrows. As shown by the arrows, \( l \)-value of \( s \) propagates towards \( t \) until the vertex whose \( d_i \)-value is \( 2 \) is encountered using the \( \mathtt{te} \)-values. Observe that since \( \mathtt{tc} \)-values of vertices are used to propagate \( 2 \) from the vertex with
$d_v$-value 1 to the one with $d_v$-value 2, they cannot be used to propagate 4. Therefore, $tn$-values are used to carry 4 until the vertex with $d_v$-value 4.

The $tn$-value of $s$, the origin of $P_1$, is set to 0 and $tn$-values are computed towards $t$ incrementally to collect the largest $l$-value on $P$ encountered so far until the terminus $w_1$ of $P_1$, the vertex whose $d_v$-value is 2 in the figure. Upon completion of this, $tm(w_1)$ is set to 0 and the $tn$-value of the predecessor of $w_1$ ($pred(w_1)$) contains the largest $l$-value on $P(s, w_1)$. Then, the $tn$-value of $pred(w_1)$ is assigned to the $tc$-value of $w_1$ and this value is propagated using $tc$-values over the subpath $P(w_1, w_2)$ until encountering vertex ($w_2$) whose $ds$-value is equal to propagated value to identify the terminus $w_2$ of $P_2$ in the aforementioned manner. While this propagation takes place, the largest $l$-value on $P(w_1, w_2)$ is found using the $tn$-values. In this manner, the $tn$ and $tc$-values are computed and the terminuses of link paths are identified. However, the origins of link paths are not yet explicitly identified when the aforementioned computation is carried out starting at $s$ and continuing towards $t$. For that purpose, the $or$-values are computed as follows. First $or(t)$ is assigned 0. Notice that in Figure 4, the vertex with $d_v$-value 6 starts the propagation of value 7, and it continues until reaching the vertex with $l$-value of 7. Then, this vertex is identified as the origin of a link path. Similarly, the vertex with $d_v$-value of 4, starts the propagation of value 6, and it continues until reaching the vertex with $l$-value of 6. Then, this vertex is identified as the origin of a link path.

Algorithm 3 shows the third phase of the algorithm implementing the above strategy. In the description of the algorithm $suc(v)$ refers to the successor of vertex $v$ on $P$.

When the third phase of the algorithm terminates, the following propositions hold. As a result, the origins and the terminuses of link paths are identified.

**Proposition 1.** Vertex $v \in P$ is the terminus of a link path iff $tc(pred(v)) = d_v(v)$ holds.

**Proposition 2.** Vertex $v \in P$ is the origin of a link path iff $l(v) = or(v)$ holds.

Algorithm 3. Construction of link-paths.

1. **integer** $tc(v); tn(v), or(v)$ **for each vertex** $v \in V$;
2. $tc(s) := l(s)$;
3. $tn(s) := 0$;
4. **for all** $v \in P\backslash\{s\}$ **in order do**
5. 
6. **if** $d_v(v) \neq tc(pred(v))$ **then**
7. $tc(v) := tc(pred(v))$;
8. $tn(v) := \max\{tn(pred(v)); l(v)\}$;
9. **else**
10. $tc(v) := tn(pred(v))$);
11. $tn(v) := 0$;
12. **end if**
13. **end for**
14. $or(t) := 0$;
15. **for all** $v \in P\backslash t$ **in reverse order do**
16. 
17. **if** $tn(v) \neq 0$ **then**
18. 
19. $or(v) := 0$;
20. **end if**
21. **else**
22. $or(v) := tn(pred(v))$;
23. **end if**
24. **end for**

Figure 4. The figure illustrates the computation of $tn$, $tc$ and $or$-values of vertices on a shortest path between two endpoints $s$ and $t$ in the third phase of the algorithm.
3.4. Construction of Disjoint Paths

We now present the fourth phase of the algorithm that constructs the node-disjoint paths \( P^1 \) and \( P^2 \) based on the previous three phases.

Using the link paths and the shortest path \( P \), node-disjoint paths \( P^1 \) and \( P^2 \) are constructed as follows. First, the first vertex on each is determined to be \( s \). Second, the second vertex on \( P^1 \) is determined to be the neighbor of \( s \) on \( P^1 \), i.e., the second vertex on \( P^1 \) is assumed to be the neighbor of \( s \) with the largest \( l \)-value. In addition, the second vertex on \( P^2 \) is determined to be the neighbor of vertex \( s \) on \( P \). After determining the first two vertices on \( P^1 \) and \( P^2 \), disjoint paths \( P^1 \) and \( P^2 \) are extended in the same manner, as follows. Let \( v \) be the last vertex on the disjoint path, either \( P^1 \) or \( P^2 \), constructed thus far. Notice that vertex \( v \) can either be a vertex on \( P \) or on a link path. We first consider the case where \( v \) is on path \( P \). If \( l(v) = o_r(v) \) holds for \( v \) on path \( P \), then the next vertex is determined to be the neighbor of \( v \) with the largest \( l \)-value, i.e., the next vertex is the second vertex on the link path whose origin is \( v \). Otherwise, the next vertex is the successor of \( v \) on \( P \). We now consider the case where vertex \( v \) is on a link path. In this case, the next vertex is determined to be the next vertex on the link path. Recall that the successor of vertex \( v \) on a link path is a child of \( v \) in \( T \) with the largest \( l \)-value. The construction of each disjoint-path ends after the target vertex \( t \) is added to the path.

We need the following definitions to facilitate the description of the fourth phase of the algorithm. Function \( \text{app}(P, v) \) appends vertex \( v \) at the end of path \( P \). Function \( \text{succ}_P(v) \) returns the successor of vertex \( v \) on path \( P \). Function \( \text{last}(P) \) returns the last vertex on path \( P \). \( N_v \) denotes the neighboring vertices of vertex \( v \) in \( G \).

4. Correctness

In this section, we present a number of lemmas to establish the correctness of the proposed algorithm.

**Lemma 2.** After the completion of the first phase of the algorithm, a tree \( T = (V_T, E_T) \) rooted at vertex \( s \) is constructed in \( G \) such that for each vertex \( v \) and \( w \) on \( P \), if \( d_r(v) = d_r(w) - 1 \), then vertex \( v \) is the parent of vertex \( w \), and for each vertex \( v \) on \( P \), each vertex \( w \) in \( G' = (V \setminus P, E \setminus P_k) \) where \( P_k \) denotes the set of edges connecting consecutive vertices in \( P \), reachable via a path in \( G' \) from vertex \( v \) on \( P \) such that \( d_r(v) \) is minimal, is a descendant of \( v \).

**Proof.** Observe that, in the first phase of the algorithm, vertices on \( P \) are added to \( T \) in order. Also observe that after each vertex \( v \) on \( P \) is added, all the vertices in \( G \) reachable from \( v \) via a path that does not contain a vertex on \( P \) are added in a BFS manner. Hence, proof follows. □

**Lemma 3.** Upon completion of this phase, \( l(v) \)-value of each vertex on \( P \) denotes the \( d_r \)-value of the farthest vertex from \( s \) on \( P \) reachable via a path disjoint from \( P \) (except for its endpoints.)

**Proof.** Observe that for each vertex \( v \in T \), the second phase of the algorithm computes the largest \( d_r \)-value among vertices on \( P \) that are incident on a non-tree edge connecting these vertices on \( P \) to descendants of \( v \) in \( T \) and assigns it to its \( l \)-value, \( l(v) \) in a bottom-up manner in \( T \). Hence, proof follows. □

**Lemma 4.** Let \( LP = P_1, P_2, \ldots, P_k \) be a sequence of link paths in \( G \) for source \( s \), target \( t \) and shortest path \( P \) between \( s \) and \( t \). \( P_1 \) is a link path with origin \( s \) and terminus \( v(l(s)) \), where \( v(l(s)) \) denotes the vertex on \( P \) with \( d_r \)-value equal to \( l(s) \). \( P_2 \) is a link path with origin \( \alpha_2 = v(\max(s, v(l(s)))) \) and terminus \( v(l(\alpha_2)) \), where \( \max(v_1, v_2) \) denotes the largest \( l \)-value among vertices on the subpath of \( P \) that extends from \( v_1 \) to \( v_2 \) on \( P \). For each link path \( P_i, 2 < i \leq k \), the origin \( \alpha_i \) of link path \( P_i \) is identified as the vertex with the largest \( l \)-value among vertices on \( P \) with the \( d_r \)-value on \( P(d_r(w_{i-2}), d_r(w_{i-1})) \), where \( w_{i-2} \) is the terminus of link path \( P_{i-2} \) and \( w_{i-1} \) is the origin of link path \( P_{i-1} \). Whereas the terminus of each link path \( P_i, 0 < i \leq k \), with origin \( \alpha_i \) is vertex \( v(l(\alpha_i)) \).

**Proof.** Clearly the first link path \( P_1 \) originates at \( s \). Since \( l(s) \) denotes the \( d_r \)-value of the farthest reachable vertex from \( s \) on \( P \) reachable via a path disjoint from \( P \), and by Lemma 3 and the definition of link paths, \( P_1 \) terminates at \( v(l(s)) \). Also by Lemma 3 and the definition of link paths \( P_2 \) is a link path extending from origin \( \alpha_2 = v(\max(s, v(l(s)))) \) to \( v(l(\alpha_2)) \).

1. path $P^1, P^2 := s$;
2. app($P^1, v$), where $v \in N_s | \forall j \in N_v \{l(v) \geq l(j)\}$;
3. app($P^2, \text{succ}(s)$);
4. complete-path($P^3$);
5. complete-path($P^2$);
6. terminate;
7. function complete-path($Q$);
8. while last($Q$) $\neq t$ do
9. \hspace{1em} if (last($Q$) $\in P$) then
10. \hspace{2em} if ($l$(last($Q$)) $\in \text{or}(last(Q))$) then
11. \hspace{3em} app($Q, v$), where $v \in N_{last(Q)} | \forall j \in N_{last(Q)} \{l(v) \geq l(j)\}$;
12. \hspace{2em} else
13. \hspace{3em} app($P^2$, \text{succ}(s));
14. \hspace{1em} end if
15. \hspace{1em} else
16. \hspace{2em} app($Q, v$), where $v \in N_{last(Q)} | \text{last}(Q) = p(v) \land l(\text{last}(Q)) = l(v)$;
17. \hspace{1em} end if
18. end while

Notice that for $1 < i \leq k$, the origin $o_i$ of $P_i$ is the vertex with the largest $l$-value on $P(o_{i-1}, w_{i-1})$. Since the origin $o_{i+1}$ of $P_{i+1}$ needs to be on $P(o_i, w_i)$ but cannot be on $P(o_{i-1}, w_i)$, origin $o_{i+1}$ is on $P(w_{i-1}, w_i)$. Inductively, by Lemma 3 and the above, it is easy to show that for each link path $P_i, 2 < i \leq k$, the origin $o_i$ of link path $P_i$ is identified as the vertex with the largest $l$-value among vertices on $P$ with the $d_i$-value on $P(d_i(w_{i-2}), d_i(w_{i-1}))$, where $w_{i-2}$ is the terminus of link path $P_{i-2}$ and $w_{i-1}$ is the terminus of link path $P_{i-1}$, whereas the terminus of each link path $P_i, 0 < i \leq k$, with origin $o_i$ is vertex $v(l(o_i))$. \hfill $\square$

Lemma 5. Upon completion of the third phase of the algorithm, each vertex of a link path discovers whether or not it is an origin or a terminus of a link path only using the variables of the vertex.

**Proof.** Let $P_i, 0 < i \leq k$, be a link path with origin $o_i$ and terminus $w_j$.

In the third phase of the algorithm, using the $tn$ and $tc$-values, the terminus of each link path $P_i$ starting from link path $P_{i-1}$, one after the other, is identified as follows. On path $P_1$, value $l(s)$ is copied form a vertex to another towards $t$ using $tc$-values, when the copied value is equal to the $d_i$-value of a vertex, this vertex is identified as the terminus of $P_1$. While value $l(s)$ is copied to vertex $w_1$, $tn$-values are used to find and copy the largest $l$-value on $P[s, w_1]$. This largest value is used to identify the terminus of $P_2$ in the same manner by copying the value starting from $w_1$ from a vertex to another towards $t$ using $tc$-values. For the subsequent link paths, $tn$-values are used to find the largest $l$-value between two consecutive terminuses $w_i$ and $w_{i+1}$ and this value is copied form a vertex to another from the latter terminus $w_{i+1}$ towards $t$ using $tc$-values to discover terminus $w_{i+2}$. Upon discovery of each terminus, its $tn$-value is set to zero to start discovering the next largest $l$-value between the terminus and the consecutive terminus. Based on these arguments, it is easy to inductively show that all terminuses of link paths are identified and their $tn$-values are set to zero upon completion of the third phase of the algorithm.

Now, we are to show whether or not a vertex is the origin of a link path using only the variables of the vertex. Notice that after the third phase of the algorithm is completed and the terminuses are identified, $tn(s) = 0$ and $tn(w_j) = 0$ hold for each $P_i, 0 < i \leq k$. Also notice that after the third
phase of the algorithm is completed, all the followings hold.

For every vertex $v$ on $P[a_i, w_i]$, $tn(v)$ denotes the largest $l$-value among vertices on $P[a_i, v]$. For each terminus vertex $w_i, 0 < i < k - 1$, $tn(pred(w_i))$ denotes the largest $l$-value of vertices on $P[w_i, w_{i+1}]$ and the $l$-value of the vertex that is the origin of link path $P_{i+2}$. $tn(pred(t))$ denotes the largest $l$-value (if any) of a potential link path between $w_{k-1}$ and $w_k$. Otherwise, i.e., if no potential link path exists between $w_{k-1}$ and $w_k$, $tn(pred(t))$ denotes 0.

For every $i, 0 < i < k$, $or$-values of all the vertices on path $P[w_i, w_{i+1}]$ contain value $tn(pred(w_i))$ which is the $l$-value of the origin of $P_{i+2}$.

Therefore, for each $i, 0 < i < k$, $or$-values of all vertices on path $P[w_i, w_{i+1}]$ denote the largest $l$-value of vertices on $P[w_i, w_{i+1}]$. Since the origin of each link path $P_{i+2}, 1 < i < k - 1$ is the vertex with the largest $l$-value on path $P[w_i, w_{i+1}]$, then vertex with the largest $l$-value on this path is the origin of a link path iff $l(v) \in or(v)$. Hence, the proof follows. □

Lemma 6. Paths $P^1$ and $P^2$ constructed by the algorithm are disjoint between $s$ and $t$.

Proof. First observe that both $P^1$ and $P^2$ start at $s$ and the second vertex on $P^1$ is the second vertex on $P_1$, whereas the second vertex on $P^2$ is the second vertex on $P$. Notice that these choices of the second vertices on both $P^1$ and $P^2$ are not necessarily unique, however, the choice made leads to the construction of disjoint paths.

Now, we are to show that function call complete-path($P^1$) constructs $P^1$ by including all odd numbered link paths, subpaths of $P$ connecting the terminus of one odd numbered link path to the origin of the consecutive odd numbered link path, and if the terminus of the last odd numbered link path is not $t$, the subpath of $P$ connecting the terminus of the last odd numbered link path and $t$. Clearly, function complete-path($P^1$) in each step adds a new vertex $v$ to the constructed path whose last vertex is $v'$ that satisfies the following. If $v'$ is on $P$ and $v'$ is not an origin of a link path, i.e., $(iv' \notin or(v'))$. If $v$ is the next vertex on $P$, $v'$ is on a link path, $v$ is the next vertex on the link path. Otherwise, if $v'$ is the origin of a link path, then $v$ is the next node on the link path with $v$ as its origin.

Observe that this scheme ensures that after including an odd numbered link path in $P^1$ and a number of vertices towards $t$ are added to $P^1$ until encountering the next origin of a link path which happens to be the origin of the consecutive odd numbered link path or $t$. This is because the origin of the even numbered consecutive link path precedes the odd numbered link path on $P$. It is easy to see that disjoint path $P^2$ is constructed in an analogous manner. It is also easy to see that, since odd numbered and even numbered link paths are node-disjoint, aforementioned subpaths of $P$ connecting odd numbered and even numbered subpaths are disjoint, and $P(s, o_2)$ is included only in $P^2$ and $P(w_{k-1}, t)$ is included in one of the disjoint paths $P^1$ or $P^2$, paths $P^1$ and $P^2$ are disjoint. □

Lemma 7. The proposed algorithm has time complexity of $O(m)$.

Proof. It is easy to see that the first, the second, the third and the fourth phases of the algorithm have the time complexities of $O(m)$, $O(m)$, $O(D)$, and $O(n)$, respectively, where $D$ denotes the diameter of the graph. Hence, the proof follows. □

The following lemma establishes the correctness of the proposed algorithm whose proof follows from Lemmas 6 and 7.

Lemma 8. The proposed algorithm constructs two node-disjoint paths $P^1$ and $P^2$ from $s$ to $t$ in $O(m)$-time.

5. Conclusion

In this paper, we presented a sequential algorithm for finding two disjoint paths in arbitrary graphs. Given two distinct vertices $s$ and $t$ of a graph $G$, the disjoint paths problem is to determine all disjoint paths between $s$ and $t$. It is an open problem to devise an algorithm for finding all disjoint paths algorithm in arbitrary graphs based on the proposed approach. We are currently devising a distributed implementation of the proposed approach.

It is anticipated that the entirely new proposed approach will initiate further research in this area with numerous useful applications.
An Algorithm for Finding Two Node-Disjoint Paths in Arbitrary Graphs

References


Contact address:
Mehmet Hakan Karaata
Department of Computer Engineering
Kuwait University
Safat
Kuwait
e-mail: mehmet.karaata@ku.edu.kw

Mehmet Hakan Karaata received his PhD and MSc degrees in computer science from the University of Iowa, Iowa City, Iowa in 1995 and 1990 respectively, and his BSc degree in computer science from the University of Hacettepe, Ankara, Turkey in 1987. He joined Bilkent University, Ankara, Turkey, as an Assistant Professor in 1995. He is currently working as Professor in the Department of Computer Engineering at Kuwait University. His research interests include mobile computing, distributed systems, fault-tolerant computing, and self stabilization. Prof. Karaata has earned the Distinguished Best Young Researcher Award and Researcher Award in 2001 and 2009 respectively, from Kuwait University, Kuwait.