# Finding all paths on undirected graph

I have an undirected, unweighted graph, and I'm trying to come up with an algorithm that, given 2 unique nodes on the graph, will find all paths connecting the two nodes, not including cycles. Here's an illustration of what I'd like to do: Graph example

Does this algorithm have a name? Can it be done in polynomial time?

Thanks,

Jesse

graph-theory path-connected

asked Mar 18 '10 at 15:31
JesseStimpson
176 1 1 4

15 It cannot be done in polynomial time: in the complete graph $K_n$ there are approximately $n!$ paths between every pair of vertices. Just outputing all these paths would take longer than a polynomial in n. – Boris Bukh Mar 18 '10 at 15:34

1 @Boris Is it possible, though, to get some nicer estimates on the complexity of enumerating all paths in terms of how many such paths there are? It won't be polynomial in the number of nodes of the graph, but the enumeration might be polynomial in the number of paths it outputs, or something like that? Or is there an obstruction to such an algorithm as well? – Mikael Vejdemo-Johansson Mar 18 '10 at 16:49

3 @mikael that's not a problem. start at the first node, and do a DFS. Each time you reach the second node, spit out the current path, and note which edge on the stack got you there. So overall the time taken is linear in the total length of all the paths, which is at most n times the number of paths. – Suresh Venkat Mar 18 '10 at 17:31

2 @Suresh: Are you sure a DFS will find *all* paths? Asssume two $K_n$ connected by a single bridge edge, with the source node in one $K_n$ and the target node in the other. DFS will traverse the bride edge exactly once, while there is certainly a much larger number of distinct paths from source to target that are crossing this edge. – MRA Mar 19 '10 at 12:50

2 I don't think polynomial in the number of paths is possible. Consider the disjoint union of two large cliques with $u$ in one clique and $v$ in the other. Then there are a constant number of $u-v$ paths (zero of them), but we have to at least look at the entire graph to determine so. – Tony Huynh Apr 19 '12 at 16:42

## 5 Answers

There is an easy way to partition the set of $s$-$t$ paths in a graph $G$. Fix an edge $tt'$ in $G$. Let $P_1$ be the set of paths from $s$ to $t$ which use the edge $tt'$, and let $P_2$ be the set of paths from $s$ to $t$ in $G-tt'$. Then $P_1 \cap P_2 = \emptyset$ and the set of $s$-$t$ paths $P = P_1 \cup P_2$. Moreover, there is a one to one correspondence between the set of paths $P_1$ and the set of $s$-$t'$ paths in the graph $G-t$.

Thus, we get an easy recursive algorithm to find the set of paths $s$-$t$ paths in a graph $G$. Pick an edge $tt'$ incident the vertex $t$ and recursively calculate the sets $P_1$ and $P_2$. With a small amount of pre-processing, we can ensure that the runtime is $O(m(p+1))$ where $m$ is the number of edges and $p$ is the number of $s$-$t$ paths.

To make the recurrence relation on the runtime work, consider the following. We can test in time $O(m)$ if a given graph $G$ and pair of vertices $s$ and $t$ if $G$ has 0, exactly one, or at least two distinct $s$-$t$ paths. To see this, simply find the block decomposition of the graph and, and check if there is any non-trivial block between $s$ and $t$ in the tree.

We can push this slightly farther. Given an instance of the problem $G$ and vertices $s$ and $t$, we can reduce the problem in time $O(m)$ to a graph $\bar{G}$ and vertices $x$ and $y$ such that for all edges $xx'$ incident to $x$, we have that

1. $xx'$ is in some $y$-$x$ path,
2. there exists a $y$-$x'$ path in $\bar{G}-x$.

To see this, again using the block decomposition, we contract any bridge in the graph and delete edges not contained in any $s$-$t$ path. As above, this can be done in $O(m)$ time.

We give an $O(m(p-1))$ time algorithm to find the set of $s$-$t$ paths in a given graph $G$ with at least two $s$-$t$ paths.

1. We may assume, as above, that every edge $tt'$ incident to $t$ is contained in some $s$-$t$ path and that there exists at least one $s$-$t'$ path in $G-t$.
2. Check if the number of $s$-$t$ paths in $G-tt'$ is at least two, and if not, let $P_1$ be the set of the unique $s$-$t$ path in $G-tt'$. If there are at least two such paths, we recursively find the set of all such paths. Let $p_1 = |P_1|$. By choice of $tt'$, $p_1 \geq 1$.

3. Check if the number of $s$-$t'$ paths in $G - t$ is at least two, and if not let $P_2$ be the set of the unique $s$-$t'$ path in $G - t$. Otherwise, we recursively find the set $P_2$ of $s$-$t'$ path in $G - t$. Let $p_2 = |P_2|$, and as above, we again have $p_2 \geq 1$.

Step 1 can be performed in $c'm$ operations for some constant $c'$. The initial check in steps $2$ and $3$ can be performed in $c'(m-1)$ steps. If we must recursively run the algorithm, this will require another $c(m-1)(p_i - 1)$ operations for $i = 1, 2$ in Steps 2 and 3, respectively. As $p_i \geq 1$, we can bound the work in each of Steps 2 and 3 by $c'm + cm(p_i - 1)$. Thus, the total number of operations is at most $3c'm + c(m)(p_1 + p_2 - 2) \leq cm(p-1)$ if we choose $c \geq 3c'$.

answered Apr 26 '12 at 21:33
Paul Wollan
**532** 4 6

---

Suresh suggested DFS, MRA pointed out that it's not clear that works. Here's my attempt at a solution following that thread of comments. If the graph has $m$ edges, $n$ nodes, and $p$ paths from the source $s$ to the target $t$, then the algorithm below prints all paths in time $O((np+1)(m+n))$. (In particular, it takes $O(m+n)$ time to notice that there is no path.)

The idea is very simple: Do an exhaustive search, but bail early if you've gotten yourself into a corner.

Without bailing early, MRA's counter-example shows that exhaustive search spends $\Omega(n!)$ time even if $p = 1$: The node $t$ has only one adjacent edge and its neighbor is node $s$, which is part of a complete (sub)graph $K_{n-1}$.

Push s on the path stack and call search(s):

```
path // is a stack (initially empty)
seen // is a set

def stuck(x)
   if x == t
     return False
   for each neighbor y of x
     if y not in seen
       insert y in seen
       if !stuck(y)
         return False
   return True

def search(x)
  if x == t
    print path
  seen = set(path)
  if stuck(x)
    return
  for each neighbor y of x
    push y on the path
    search(y)
    pop y from the path
```

Here *search* does the exhaustive search and *stuck* could be implemented in DFS style (as here) or in BFS style.

edited Mar 19 '10 at 19:21

answered Mar 18 '10 at 18:31
rgrig
**805** 1 12 19

1 This method is correct. Maybe you can reduce the running time per path if you don't do the stuck() test so often. Maybe something like calling stuck() if a path to the destination hasn't appeared in the past $n$ steps can be made to work? The details would be a little hairy. – Brendan McKay Apr 19 '12 at 22:55

Can you elaborate on $O((np+1)(m+n))$ ? How did you come up with it? – hhh Jul 28 '13 at 10:24

I cannot understand the condition in which this algorithm stops. This sequence determines the amount of paths here but how does the algorithm actually select which path to consider and which not? Just cut out paths above certain length? – hhh Aug 2 '13 at 14:53

What exactly is `t` ?? – CppLearner Mar 19 '14 at 15:56

@CppLearner: "the graph has $m$ edges, $n$ nodes, and $p$ paths from the source $s$ to the target $t$" – rgrig Mar 19 '14 at 21:28

|

---

Let $G = (V, E)$ be a graph.

$FindPaths(p, f)$ prints all paths which end in $f$ and can be obtained by adding nodes to path $p$. $p$ is for path, $f$ is for final (node).

```
Def $FindPaths(p, f)$:
Let $x$ be the last node of $p$.
For each edge $xy$ for some $y$ in $E$
  If $y$ is not in $p$
    If $y = f$
      Print $p - y$
    Else
      $FindPaths(p - y, f)$
```

If $s$ is the start node and $t$ is the ending node, run $FindPaths(s, t)$.

You can represent path and edges as strings. To check if a node is in a path $p$ you just have to check whether the string contains the character that represents the node. To get the final node of a path use the function to get the last character of a string.

EDIT: My answer is not math research level, but introduction to programming.

edited Apr 27 '12 at 6:02 answered Apr 27 '12 at 5:53
Ernest
**21** 1 2

---

For shortest paths look at the Wikipedia page of the Floyd–Warshall algorithm

and also:

"An algorithm for computing all paths in a graph" by Lars-Erik Thorelli in BIT Numerical Mathematics, July 1966, Volume 6, Issue 4, pp 347-349

edited Jun 28 '13 at 19:44
user9072
answered Mar 19 '10 at 1:42
Joseph Malkevitch
**1,635** 1 11 11

1 Since the source and the target are fixed and since there are no weights on edges, the shortest path is best computed by BFS, rather than Roy-Floyd-Warshall. – rgrig Mar 19 '10 at 10:36

---

If I'm not mistaken, I think an adaptation of a dynamic programming all-pairs-shortest-path algorithm (like the Floyd-Warshall algorithm, considering edge weights of 1) might find all paths. Consider the following scheme to find the *total number* of all paths leading from *u* to *v* (or, in fact, from any start node to any destination):

A matrix $M_1$ is initialized as the adjacency matrix of the graph. That is, $M_1[u, v]$ containes the number of simple paths of length at most 1 from *u* to *v*. After that, for all $i$ from 2 to the number of nodes the matrix $M_i$ is updated as follows: $M_i[u, v]$ equals the sum of the entries $M_{i-1}[u, w]$ for all nodes *w* adjacent to *v*. Hence, $M_i[u, v]$ containes the number of simple paths of length at most *i* from *u* to *v*. This scheme runs in time polynomial in the input size, and it can also be modified easily to also store the actual paths in time (and space) polynomial in the output size.

answered Mar 19 '10 at 12:44
MRA
**167** 1 5 7

I think this is not an easy problem, and may not be able to compute efficiently. Trying to work out a proof on it. By the way, I think there may be some problems with @MRA's dynamic programming solution, because it seems that in the M_i's, there is no way to memorize whether a path has visited certain vertices, while we are required to find all simple paths. Therefore, the dynamic programming solution might include those paths with cycles, and thus would be incorrect. – Sangxia Huang Mar 19 '10 at 13:10

I agree. Thanks for pointing out the flaw. – MRA Mar 19 '10 at 13:23

---