Find all paths between two graph nodes

Ask Question

I am working on an implemtation of Dijkstras Algorithm to retrieve the shortest path between interconnected nodes on a network of routes. I have the implentation working. It returns all the shortest paths to all the nodes when I pass the start node into the algorithm.

My Question: How does one go about retrieving all possible paths from Node A to say Node G or even all possible paths from Node A and back to Node A

algorithm

graph-theory

edited May 1 '15 at 4:19

chouaib
2,485 5 14 31

asked Mar 2 '12 at 15:25

Paul
400 1 7 16

Well, if your graph has cycles, that could be an extremely long list. – zmccord Mar 2 '12 at 15:27

9 I've taken the liberty to rename your question, since it isn't about Dijkstra's algorithm, but about generating paths between graph nodes – Zruty Mar 2 '12 at 15:30

Do you want paths that don't repeat vertices/edges? – HexTree Mar 2 '12 at 15:31

@ HexTree I'm not too sure what you mean. Each vertice is unique. I'm basically looking for each path the weight of that path and the number of nodes that were touched via each path – Paul Mar 2 '12 at 15:52

1 Hi, Paul, Have you solved this question? Here is a link may be helpful for you: geeksforgeeks.org/find-paths-given-source-destination – GoingMyWay May 15 at 13:02

13 Answers

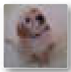
Finding **all** possible paths is a hard problem, since there are exponential number of simple paths. Even finding the kth shortest path [or longest path] are NP-Hard.

One possible solution to find all paths [or all paths up to a certain length] from `s` to `t` is BFS, without keeping a `visited` set, or for the weighted version - you might want to use uniform cost search

Note that also in every graph which has cycles [it is not a DAG] there might be infinite number of paths between `s` to `t`.

edited May 31 at 0:30

answered Mar 2 '12 at 15:29



amit
133k 17 163 269

Thanks amit I will try looking at BFS or the uniform cost search – Paul Mar 2 '12 at 16:01

1 @Paul: You are welcome. just make sure in both of them you don't use a `visited` set [like the original algorithm suggests] or you will get only part of the paths. Also, you should limit paths to a certain length to avoid infinite loops [if the graph have cycles...]. Good Luck! – amit Mar 2 '12 at 16:06

@amit can it be done in DFS as well? – william007 Feb 1 '13 at 2:03

@william007: Sure you can, but beware that you might get stuck in a cycle and stop yielding answers after a while. However - to get all *simple* paths from A to G - DFS is the way to go, and your `visited` set is per path (i.e. when you come back from the recursion, remove the element from the set before you continue to next node). – amit Feb 1 '13 at 9:09

1 @VShreyas That's kinda old thread, the answer specifically says "all paths up to certain length", and that can be done with BFS without visited set. If you want to all *simple* paths between two nodes, you can do it

with DFS with "local" visited set (that deletes a node from the visited set when it tracks back). – amit Aug 17 '15 at 16:42

|

I've implemented a version where it basically finds all possible paths from one node to the other, but it doesn't count any possible 'cycles' (the graph I'm using is cyclical). So basically, no one node will appear twice within the same path. And if the graph were acyclical, then I suppose you could say it seems to find all the possible paths between the two nodes. It seems to be working just fine, and for my graph size of ~150, it runs almost instantly on my machine, though I'm sure the running time must be something like exponential and so it'll start to get slow quickly as the graph gets bigger.

Here is some Java code that demonstrates what I'd implemented. I'm sure there must be more efficient or elegant ways to do it as well.

```
Stack connectionPath = new S
List<Stack> connectionPaths
// Push to connectionsPath t
the method below
void findAllPaths(Object nod
    for (Object nextNode : n
       if (nextNode.equals(t
           Stack temp = new
           for (Object node1
               temp.add(node
           connectionPaths.a
       } else if (!connectio
           connectionPath.pu
           findAllPaths(next
```

```
            connectionPath.po
        }
    }
}
```

edited Mar 18 '14 at 13:17

answered Mar 17 '14 at 20:19

Omer Hassan
111 1 4

is there non-recursive version of this ? – arslan Feb 3 '16 at 9:25

I don't have one, no, but I think in theory, any recursive program can be converted into a non-recursive one, I think by using something like a stack object, the point being to emulate what a recursive program is actually doing using the program's stack space, I believe. You can look up the principle of converting recursive programs to non-recursive. – Omer Hassan Feb 3 '16 at 10:23

I'm gonna give you a (somewhat small) version (although comprehensible, I think) of a scientific proof that you cannot do this under a feasible amount of time.

What I'm gonna prove is that the time complexity to enumerate all simple paths between two selected and distinct nodes (say, `s` and `t`) in an arbitrary graph `G` is not polynomial. Notice that, as we only care about the amount of paths between these nodes, the edge costs are unimportant.

Sure that, if the graph has some well selected properties, this can be easy. I'm considering the general case though.

Suppose that we have a polynomial algorithm that lists all simple paths between `s` and `t`.

If `G` is connected, the list is nonempty. If `G` is not and `s` and `t` are in different components, it's really easy to list all paths between them, because there are none! If they are in the same component, we can pretend that the whole graph consists only of that component. So let's assume `G` is indeed connected.

The number of listed paths must then be polynomial, otherwise the algorithm couldn't return me them all. If it enumerates all of them, it must give me the longest one, so it is in there. Having the list of paths, a simple procedure may be applied to point me which is this longest path.

We can show (although I can't think of a cohesive way to say it) that this longest path has to traverse all vertices of `G`. Thus, we have just found a [Hamiltonian Path](#) with a polynomial procedure! But this is a well known NP-hard problem.

We can then conclude that this polynomial algorithm we thought we had is *very unlikely* to exist, unless P = NP.

answered Jan 31 '13 at 8:23

araruna
96 2 9

If I understand correctly, then that proof only works for undirected graphs, since in a directed graph the assertion that "this longest path has to traverse all vertices of `G` " does not necessarily hold. Is that right? – boycy Jun 6 '14 at 21:31

Well, yes, but you could use your algorithm to answer whether there is a directed hamiltonian path in a similar manner, which is also NP-complete. If your answer is `n-1` , then there is. If it is not, then there couldn't be such a path, or else it would be longer than your known longest. – araruna Jun 7 '14 at 12:08

1 Just to be clear. If the directed version could be solved in poly time, it's answer would give the answer to the Directed Hamiltonian Path. Moreover, if we had weighted edges, one can show that by a polynomial process we could answer the Traveling Salesman Problem. – araruna Jun 7 '14 at 12:18

Here is an algorithm finding and printing all paths from s to t using modification of DFS. Also dynamic programming can be used to find the count of all possible paths. The pseudo code will look like this:

```
AllPaths(G(V,E),s,t)
 C[1...n]    //array of inte
 TopologicallySort(G(V,E))

  for i<-0 to n
      if i<i0
          C[i]<-0  //there i
topological sort
      if i==i0
         C[i]<-1
      for j<-0 to Adj(i)
          C[i]<- C[i]+C[j]

 return C[i1]
```

answered Feb 4 '16 at 16:24

yanis
170 11

You usually don't want to, because there is an exponential number of them in nontrivial graphs; if you really want to get all (simple) paths, or all (simple) cycles, you just find one (by walking the graph), then backtrack to another.

answered Mar 2 '12 at 15:31

jpalecek
39.8k 5 78 126

It's simple, efficient and doable for any DAG. You are misleading @Paul. – Diego Mar 2 '12 at 15:32

find_paths[s, t, d, k]

This question is now a bit old... but I'll throw my hat into the ring.

I personally find an algorithm of the form `find_paths[s, t, d, k]` useful, where:

- s is the starting node
- t is the target node
- d is the maximum depth to search
- k is the number of paths to find

Using your programming language's form of infinity for `d` and `k` will give you all paths§.

§ obviously if you are using a directed graph and you want all *undirected* paths between `s` and `t` you will have to run this both ways:

```
find_paths[s, t, d, k] <join
```

Helper Function

I personally like recursion, although it can difficult some times, anyway first lets define our helper function:

```
def find_paths_recursion(gra
current_path, paths_found)
  current_path.append(curren

  if current_depth > max_dep
    return

  if current == goal:
    if len(paths_found) <= n
      paths_found.append(cop

    current_path.pop()
    return

  else:
    for successor in graph[c
    self.find_paths_recursio
num_paths, current_path, pat
```

```
  current_path.pop()
```

Main Function

With that out of the way, the core function is trivial:

```
def find_paths[s, t, d, k]:
  paths_found = [] # PASSING
  find_paths_recursion(s, t,
```

First, lets notice a few thing:

- the above pseudo-code is a mash-up of languages - but most strongly resembling python (since I was just coding in it). A strict copy-paste will not work.
- `[]` is an uninitialized list, replace this with the equivalent for your programming language of choice
- `paths_found` is passed by **reference**. It is clear that the recursion function doesn't return anything. Handle this appropriately.
- here `graph` is assuming some form of `hashed` structure. There are a plethora of ways to implement a graph. Either way, `graph[vertex]` gets you a list of adjacent vertices in a *directed* graph - adjust accordingly.
- this assumes you have pre-processed to remove "buckles" (self-loops), cycles and multi-edges

answered Mar 4 '17 at 19:23

SumNeuron

825 7 22

If you actually care about ordering your paths from shortest path to longest path then it would be far better to use a modified A* or Dijkstra Algorithm. With a slight modification the algorithm will return as many of the possible paths as you want in order of shortest path first. So if what you really want are all possible paths ordered from shortest to longest then this is the way to go.

If you want an A* based implementation capable of returning all paths ordered from the shortest to the longest, the following will accomplish that. It has several advantages. First off it is efficient at sorting from shortest to longest. Also it computes each additional path only when needed, so if you stop early because you dont need every single path you save some processing time. It also reuses data for subsequent paths each time it calculates the next path so it is more efficient. Finally if you find some desired path you can abort early saving some computation time. Overall this should be the most efficient algorithm if you care about sorting by path length.

```
import java.util.*;
```

```
public class AstarSearch {
    private final Map<Intege
    private final int destin

    private final NavigableS

    public AstarSearch(Map<I
destination) {
        this.adjacency = adj
        this.destination = d

        this.pending.add(new
    }

    public List<Integer> nex
        Step current = this.
        while( current != nu
            if( current.getI
                return curre
            for (Neighbor ne
                if(!current.
                    final St
current.cost + neighbor.cost
                    this.pen
                }
            }
            current = this.p
        }
        return null;
    }

    protected int predictCos
        return 0; //Behaves
    }

    private static class Ste
        final int id;
        final Step parent;
        final int cost;

        public Step(int id,
            this.id = id;
            this.parent = pa
            this.cost = cost
        }

        public int getId() {
            return id;
        }

        public Step getParen
            return parent;
        }

        public int getCost()
            return cost;
        }

        public boolean seen(
            if(this.id == no
                return true;
            else if(parent =
                return false
            else
                return this.
```

```
    }

    public List<Integer>
        final List<Integ
        if(this.parent !
            path = this.
        else
            path = new A
        path.add(this.id
        return path;
    }

    @Override
    public int compareTo
        if(step == null)
            return 1;
        if( this.cost !=
            return Integ
        if( this.id != s
            return Integ
        if( this.parent
            this.parent.
        if(step.parent =
            return 0;
        return -1;
    }

    @Override
    public boolean equal
        if (this == o) r
        if (o == null ||
        Step step = (Ste
        return id == ste
            cost == step
            Objects.equa
    }

    @Override
    public int hashCode(
        return Objects.h
    }
}

/************************
*   Everything below here
*   It will just be helpf
*   It isnt part of the a
************************

private static class Nei
    final int id;
    final int cost;

    public Neighbor(int
        this.id = id;
        this.cost = cost
    }

    public int getId() {
        return id;
    }

    public int getCost()
        return cost;
    }
```

```
    }

    public static void main(
        final Map<Integer, S
        final AstarSearch se
        System.out.println("
        List<Integer> path =
        while(path != null)
            System.out.print
            path = search.ne
        }
    }

    private static Map<Integ
        final Map<Integer, S

        //This sets up the a
but they dont need to.
        addAdjacency(adjacen
        addAdjacency(adjacen
        addAdjacency(adjacen
        addAdjacency(adjacen
        addAdjacency(adjacen

        return Collections.u
    }

    private static void addA
Integer... dests) {
        if( dests.length % 2
            throw new Illega
arguments, each pair is the

        final Set<Neighbor>
        for(int i = 0; i < d
            destinations.add
        adjacency.put(source
    }
}
```

The output from the above code is the following:

```
[1, 2, 4]
[1, 5, 2, 4]
[1, 5, 3, 2, 4]
```

Notice that each time you call `nextShortestPath()` it generates the next shortest path for you on demand. It only calculates the extra steps needed and doesnt traverse any old paths twice. Moreover if you decide you dont need all the paths and end execution early you've saved yourself considerable computation time. You only compute up to the number of paths you need and no more.

Finally it should be noted that the A* and Dijkstra algorithms do have some minor limitations, though I dont think it would effect you. Namely it will not work right on a graph that has negative weights.

Here is a link to JDoodle where you can run the code yourself in the browser and see it working. You can also change around the graph to show it works on other graphs as well: http://jdoodle.com/a/ukx

edited May 15 at 12:43

GoingMyWay
4,683 10 48 72

answered May 1 at 7:30

Jeffrey Phillips Freeman
637 1 6 22

I think what you want is some form of the Ford–Fulkerson algorithm which is based on BFS. Its used to calculate the max flow of a network, by finding all augmenting paths between two nodes.

http://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm

answered Oct 9 '12 at 19:05

richmb
577 1 8 15

There's a nice article which

may answer your question /only it prints the paths instead of collecting them/. Please note that you can experiment with the C++/Python samples in the online IDE.

http://www.geeksforgeeks.org/find-paths-given-source-destination/

edited Feb 24 '17 at 15:02

answered Feb 24 '17 at 9:39

Attila Karoly
262 1 12

I suppose you want to find 'simple' paths (a path is simple if no node appears in it more than once, except maybe the 1st and the last one).

Since the problem is NP-hard, you might want to do a variant of depth-first search.

Basically, generate all possible paths from A and check whether they end up in G.

answered Mar 2 '12 at 15:36

Zruty
5,457 17 28

In DAGs (directed acyclic graphs), the algorithm is called "Breadth first search"

deleted by owner Mar 2 '12 at 15:42

answered Mar 2 '12 at 15:37

Diego
13.3k 4 44 58

(1) The OP does not mention anything about the graph being a DAG. It is an unlikely assumption, (2) BFS will fail in its standard implementation, since it keeps a `visited` set, and avoids expanding the same vertex twice, which will result in finding only a part of all paths. – amit Mar 2 '12 at 15:41

---

is there non-recursive DFS algorithm for this problem?

deleted by owner Feb 3 '16 at 9:24

answered Feb 3 '16 at 9:11

arslan
472 2 6 31

Here is a blog posting that deals with finding all possible paths between an arbitrarily selected pair of nodes, written in C++.

http://www.technical-recipes.com/2011/a-recursive-algorithm-to-find-all-paths-between-two-given-nodes/

Further down the page there is a section "Example 5: Finding all paths between the same node" that deals with the problem you mentioned of finding "all possible paths from Node A and back to Node A" deleted by josliber ♦ Nov 29 '15 at 2:55

answered Sep 21 '14 at 20:39

AndyUK
2,314 6 31 39

1 While this link may answer the question, it is better to include the essential parts of the answer here and provide the link for reference. Link-only answers can become invalid if the linked page changes. - From Review – amdixon Nov 29 '15 at 1:37