Lecture 24
Search in Graphs

15-122: Principles of Imperative Computation (Spring 2017)
Frank Pfenning, André Platzer, Rob Simmons,
Penny Anderson, Iliano Cervesato

1 Introduction

In this lecture, we will discuss the question of *graph reachability*: given two vertices v and w, does there exist a path from v to w?

This maps as follows onto the learning goals for this course:

Computational Thinking: We continue learning about graphs, and specifically about paths in a graph. An important question is whether there exists a path between two given nodes. A related problem is to produce this path (if it exists).

Algorithms and Data Structures: We explore two classic approaches to answering these questions: depth-first search and breadth-first search. Both rely on the need to remember what we have done already, and to go back and try something else if we get stuck.

Programming: We give two implementations of depth-first search, one recursive that uses the call stack of C to remember what we have done, and the other iterative that uses an explicit stack for that purpose. We also see that breadth-first search is the variant of the latter where a queue is used instead of a stack.

As a reminder, we are working with the following minimal graph interface, which allows us to query only the size (number of vertices) of the graph and the existence of an edge between two given vertices. We will be implementing our search in terms of this interface.

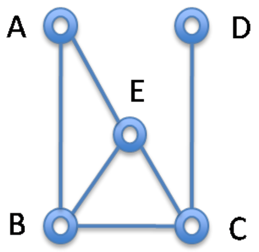
```
typedef unsigned int vertex;
typedef struct graph_header* graph_t;

graph_t graph_new(unsigned int numvert);
void graph_free(graph_t G);
unsigned int graph_size(graph_t G);

bool graph_hasedge(graph_t G, vertex v, vertex w);
  //@requires v < graph_size(G) && w < graph_size(G);

void graph_addedge(graph_t G, vertex v, vertex w);
  //@requires v < graph_size(G) && w < graph_size(G);
  //@requires v != w && !graph_hasedge(G, v, w);
```

2 Paths in Graphs

A *path* in a graph is a sequence of vertices where each vertex is connected to the next by an edge. That is, a path is a sequence

$$v_0, v_1, v_2, v_3, \ldots, v_l$$

of some length $l \geq 0$ such that there is an edge from v_i to v_{i+1} in the graph for each $i < l$.



For example, all of the following are paths in the graph above:

$$\begin{array}{l} A - B - E - C - D \\ A - B - A \\ E - C - D - C - B \\ B \end{array}$$

The last one is a special case: The length of a path is given by the number of edges in it, so a node by itself is a path of length 0 (without following any edges). Paths always have a starting vertex and an ending vertex, which coincide in a path of length 0. We also say that a path connects its endpoints.

The *graph reachability problem* is to determine if there is a path connecting two given vertices in a graph. If we know the graph is connected, this problem is easy since one can reach any node from any other node. But we might refine our specification to request that the algorithm return not just a boolean value (reachable or not), but an actual path. At that point the problem is somewhat interesting even for connected graphs. In complexity theory it is sometimes said that a path from vertex v to vertex w is a *certificate* or *explicit evidence* for the fact that vertex w is reachable from another vertex v. It is easy to check whether the certificate is valid, since it is easy to check if each node in the path is connected to the next one by an edge. It is more difficult to produce such a certificate.

For example, the path

$$A - B - E - C - D$$

is a certificate for the fact that vertex D is reachable from vertex A in the above graph. It is easy to check this certificate by following along the path and checking whether the indicated edges are in the graph.

In most of what follows we are not concerned with finding the path, but only with determining whether one exists. It is not difficult to see how to extend the algorithms we discuss to compute the path as well.

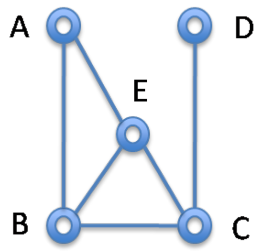
3 Depth-First Search

The first algorithm we consider for determining if one vertex is reachable from another is called *depth-first search*.

Let's try to work our way up to this algorithm. Assume we are trying to find a path from u to w. We start at u. If it is equal to w we are done, because

w is reachable by a path of length 0. If not we pick an arbitrary edge leaving u to get us to some node v. Now we have "reduced" the original problem to the one of finding a path from v to w.

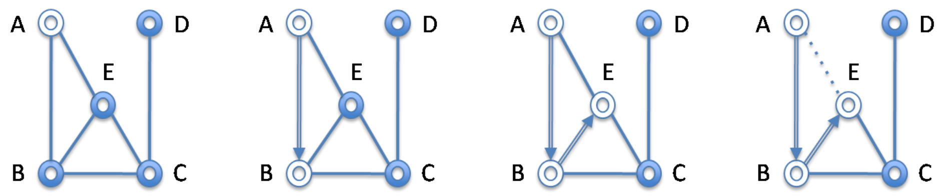
The problem here is of course that we may never arrive at w even if there is a path. For example, say we want to find a path from A to D in our earlier example graph.



We can go $A - B - E - A - B - E - \cdots$ without ever reaching D (or we can go just $A - B - A - B - \cdots$), even though there exists a path.

We need to avoid repeating nodes in the path we are exploring. A *cycle* is a path of length 1 or greater that has the same starting and ending point. So another way to say we need to avoid repeating nodes is to say that we need to avoid cycles in the path. We accomplish this by *marking* the nodes we have already considered so when we see them again we know not to consider them again.

Let's go back to the earlier example and play through this idea while trying to find a path from A to D. We start by marking A (indicated by hollowing the circle) and go to B. We indicate the path we have been following by drawing a double-line along the edges contained in it.



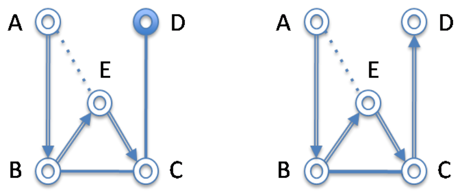
When we are at B we mark B and have three choices for the next step.

1. We could go back to A, but A is already marked and therefore ruled out.

2. We could go to E.

3. We could go to C.

Say we pick E. At this point we have again three choices. We might consider A as a next node on the path, but it is ruled out because A has already been marked. We show this by dashing the edge from A to E to indicate it was considered, but ineligible. The only possibility now is to go to C, because we have been at B as well (we just came from B).

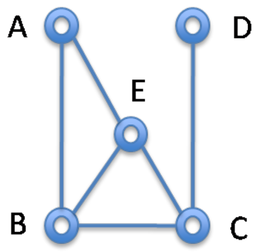


From C we consider the link to D (before considering the link to B) and we arrive at D, declaring success with the path

$$A - B - E - C - D$$

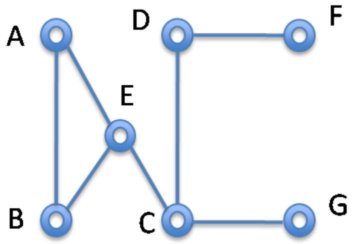
which, by construction, has no cycles.

There is one more consideration to make, namely what we do when we get stuck. Let's reconsider the original graph



and the goal to find a path from E to B. Let's say we start $E - C$ and then $C - D$. At this point, all the vertices we could go to (which is only C) have already been marked! So we have to *backtrack* to the most recent choice point and pursue alternatives. In this case, this could be C, where the only remaining alternative would be B, completing the path $E - C - B$. Notice that when backtracking we have to go back to C even though it is already marked.

Depth-first search is characterized not only by the marking, but also that when we get stuck we always return to our most recent choice and follow a different path. When no other alternatives are available, we backtrack further. Let's consider the following slightly larger graph, where we explore the outgoing edges using the alphabetically last label first. We will trace the search for a path from A to B.



We write the current node we are visiting on the left and on the right a *stack* of nodes we have to return to when we backtrack. For each of these we also remember which choices remain (in parentheses). We annotate marked nodes with an asterisk, which means that we never pick them as the next node to visit.

For example, going from step 4 to step 5 we do not consider E^* but go to D instead. We backtrack when no unmarked neighbors remain for the current node. We are keeping the visited nodes on a stack so we can easily return to the most recent one. The stack elements are separated by | and the lists of neighbors are wrapped in parentheses, e.g., (B, A^*).

Step	**Current**	**Stack**	***Remark***
1	A		
2	E	$A^*\ (B)$	
3	C	$E^*\ (B, A^*) \mid A^*\ (B)$	
4	G	$C^*\ (E^*, D) \mid E^*\ (B, A^*) \mid A^*\ (B)$	*Backtrack*
5	D	$C^*\ () \mid E^*\ (B, A^*) \mid A^*\ (B)$	
6	F	$D^*\ (C^*) \mid C^*\ () \mid E^*\ (B, A^*) \mid A^*\ (B)$	*Backtrack*
7	B	$E^*\ (A^*) \mid A^*\ (B)$	*Goal Reached*

When we think about implementing this using an adjacency list representation, it is apparent that we need some way of retrieving the neighbors of a given vertex v (those vertices that are directly connected to v by edges). Since this isn't directly possible with our minimal interface, we will iterate through all the vertices of the graph, and for each vertex w query whether the edge (v, w) exists in the graph.

3.1 Recursive Depth-First Search

Now we can easily write the depth-first search code recursively, letting the *call stack* keep track of everything we need for backtracking.

```
1  bool dfs_helper(graph_t G, bool *mark, vertex start, vertex target) {
2    REQUIRES(G != NULL && mark != NULL);
3    REQUIRES(start < graph_size(G) && target < graph_size(G));
4    REQUIRES(!mark[start]);
5
6    mark[start] = true;     // mark start as seen
7
8    if (start == target) return true;
9
10   for (vertex v = 0; v < graph_size(G); v++) {
11     if (graph_hasedge(G, start, v)) {
12       // This v is one of start's neighbors
13       if (!mark[v] && dfs_helper(G, mark, v, target))
14         return true;
15   }
16   return false;
17 }
```

We've named the function `dfs_helper` because the user of the search should not have to worry about supplying the array of marks. Instead the user calls the function `dfs`, below, which creates the marks and passes them to the recursive helper function.

```
19 bool dfs(graph_t G, vertex start, vertex target) {
20   REQUIRES(G != NULL);
21   REQUIRES(start < graph_size(G) && target < graph_size(G));
22
23   bool mark[graph_size(G)];
24   for (vertex i = 0; i < graph_size(G); i++)
25     mark[i] = false;
26   return dfs_helper(G, mark, start, target);
27 }
```

What is the cost of recursive DFS for a graph with v vertices and e edges? The function `dfs` has an initialization cost of $O(v)$ in lines 23–25, plus the cost of `dfs_helper`. Line 4, and the fact that marks are never reset, ensures that this function will be called at most v times. The loop starting at line 10 runs exactly v times. Using an adjacency matrix implementation, the call to `graph_hasedge` would take constant time. Thus, the run time of `dfs_helper`, and therefore of `dfs`, is $O(v^2)$.

It should be noted that an interface that provides a function that returns the neighbors of a node, coupled with an adjacency list based library, which would implement it as a constant-time function, would give us an $O(e)$ worst case complexity, which is comparable to our current $O(v^2)$ for dense graphs and better for sparse graphs. Specifically, assume the adjacency list library provides the function `get_neighbors(G,w)` which returns a linked list of the neighbors of vertex `w` in `G`. Then we would replace lines 10–11 above with

```
10  for (p = get_neighbors(G, start); p != NULL; p = p->next) {
11    v = p->vert;
```

Altogether, the body of this loop would be called at most $2e$ times — for each edge (v_1, v_2) in `G`, once when `start` is v_1 and once when `start` is v_2 — thereby giving us an $O(e)$ cost for `dfs`.

3.2 Depth-First Search with an explicit stack

When scrutinizing the above example, we notice that the sophisticated data structure of a stack of nodes with their adjacency lists was really quite unnecessary for DFS. The recursive implementation is simple and elegant, but its effect is to make the data management more complex than necessary: all we really need for backtracking is a stack of nodes that have been seen but not yet considered.

This can all be simplified by making the stack explicit. In that case there is a single stack with all the nodes on it that we still need to look at. (In the sample code, we use a stack specialized to hold things of type `vertex` just to keep the code simple.)

Step	**Current**	**Neighbors**	**New stack**
0			(A^*)
1	A^*	(E, B)	(E^*, B^*)
2	E^*	(C, B^*, A^*)	(C^*, B^*)
3	C^*	(G, E^*, D)	(G^*, D^*, B^*)
4	G^*	(C^*)	(D^*, B^*)
5	D^*	(F, C^*)	(F^*, B^*)
6	F^*	(D^*)	(B^*)
7	B^*	(E^*, A^*)	$()$

```
bool dfs_explicit_stack(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // Mark array initially containing only start
  bool mark[graph_size(G)];
  for (vertex i = 0; i < graph_size(G); i++)
    mark[i] = false;
  mark[start] = true;

  // Work list initially containing only start
  istack_t S = stack_new();
  push(S, start);

  while(!stack_empty(S)) {
  // Loop invariants to prove correctness go here
    vertex v = pop(S);               // v is the current node
    for (vertex w = 0;  w < graph_size(G); w++) {
      if (graph_hasedge(G, v, w)) { // w is a neighbor of v
        if (w == target) {
          stack_free(S);
          return true;               // Success!
        }
        if (!mark[w]) {              // w was not seen before
          mark[w] = true;            // Mark it as known
          push(S, w);                // Add to work list
        }
      }
    }
  }
  stack_free(S);
  return false;                      // Failure
}
```

We mark the starting node and push it on the stack. Then we iteratively pop the stack and examine each neighbor of the node we popped. If the neighbor is not already marked, we push it on the stack to make sure we look at it eventually. If the stack is empty then we've explored all possibilities without finding the target, so we return false.

While convincing, this explanation comes short of a proof that our implementation is correct, i.e., that it returns `true` when there is a path between `start` and `target` and returns `false` otherwise. We will now develop a more solid argument, although we will stop short of a formal proof. The function `dfs_explict_stack` returns in exactly three places. The first is on line 5, when `start` is equal to `target`. By definition, there is a degenerate path between these two nodes in this case.

The other two places where the function returns, lines 24 and 34, have us go through loops. To reason about loops, we need to develop loop invariants that we will squeeze in the placeholder on line 18. We start with the return statement on line 24: by the conditional on line 21, we know that `w` (which is equal to `target` by line 22) is a neighbor of vertex `v`, but how do we know that there is a path from `start` to `v`? Two invariants will prove helpful here:

1. *Every marked vertex (i.e., a vertex* `u` *such that* `mark[u] == true`*) is connected to* `start`*.*
2. *Every vertex in the stack is marked.*

These two invariants hold initially since `start` is the only marked vertex and the only item in the stack before the loop is run the first time. They are preserved by an arbitrary iteration of the loop since only neighbors of `v` (which is assumed to be reachable from `start`) are marked on line 27 and they are immediately pushed on the stack on line 28. Therefore, if the loop exits at line 24, we know that there is a path from `start` to `target`.

These two invariants are not sufficient to prove that there is no path from `start` to `target` if the function returns `false` on line 34. For this, we need a new concept and a new invariant involving it. The new concept is that of *frontier* of the search. The frontier is a set of vertices that we know are connected to `start` but that we have not explored yet. At any point in the loop on lines 17–32, the frontier is the contents of the stack. The new invariant is the following:

3 *Every path from* `start` *to* `target` *passes through a vertex in the frontier.*

It is clearly true initially when the stack (the frontier) only contains `start`. It is preserved by the loop because intuitively a frontier element is replaced by all of its neighbors (a more formal argument involves reasoning about the fact that some of these neighbors may have already been explored).

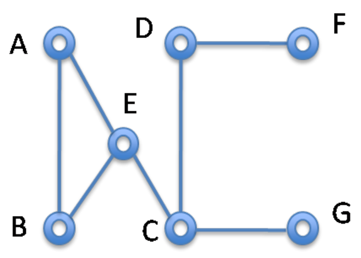
More interesting is why this invariant allows us to prove that there is no path to `target` if the function returns on line 34: for this to happen, we must have exited the loop in lines 17–32, which entails that the negation of its loop guard is true: the stack (our frontier) is empty. By our third invariant (which still holds at this point), every path from `start` to `target` must go through a vertex in the frontier. But the frontier is empty, so it contains no vertex through which such a pass can go: thus, there cannot be any path from `start` to `target`. Only in this way can the invariant be true if the frontier is empty: if all of zero paths from `start` to `target` pass through one of the (zero) vertices in the empty frontier.

The complexity considerations we developed for the recursive version of DFS apply here as well — possibly more explicitly. Assuming an adjacency matrix implementation, the above code has cost $O(v^2)$: each vertex can be pushed on the stack at most once so that the outer loop runs at most v times, and the inner loops always runs v time. Assuming a constant-cost function that returns the neighbors of a node, it can be modified to have cost $O(e)$, exactly as discussed earlier.

4 Breadth-First Search

The iterative DFS algorithm managed its agenda, i.e., the list of nodes it still had to look at, using a stack. But there's no reason to insist on a stack for that purpose. What happens if we replace the stack by a queue? All of a sudden, we will no longer explore the most recently found neighbor first as in depth-first search, but, instead, we will look at the oldest neighbor first. This corresponds to a breadth-first search where you explore the graph layer by layer. So BFS completes a layer of the graph before proceeding to the next layer. The code for that and many other interesting variations of graph search can be found on the course web page.

Here's an illustration using our running example of search for a path from A to B in the graph



Step	Current	Neighbors	New queue
0			(A^*)
1	A^*	(E, B)	(E^*, B^*)
2	E^*	(B^*, A^*, C)	(B^*, C^*)
3	B^*	(E^*, A^*)	(C^*)

We find the path much faster this way. But this is just one example. Try to think of another search in the same graph that would cause breadth-first search to examine more nodes than depth-first search would.

The code looks the same as our iterative depth-first search, except for the use of a queue instead of a stack. Therefore we do not include it here. You could write it yourself, and if you have difficulty, you can find it in the code folder that goes with this lecture. Note that our correctness and complexity analysis for DFS never relied on using a stack. Thus, it remain sound once we swap the stack for a queue. Correctness also hold for any other implementation of work list, but complexity may need to be revisited if these implementation cannot provide constant-time insertion and retrieval operations.

5 Conclusion

Breadth-first and depth-first search are the basis for many interesting algorithms as well as search techniques for artificial intelligence.

One potentially important observation about breadth-first versus depth-first search concerns search when the graph remains implicit, for instance in game search. In this case there might be infinite paths in the graph. Once embarked on such a path depth-first search will never backtrack, but will pursue the path endlessly. Breadth-first search, on the other hand, since it searches layer by layer, is not subject to this weakness (every node in a graph is limited to a finite number of neighbors). In order to get some benefits of both techniques, a technique called *iterative deepening* is sometimes used.