Lecture 23 Notes
Search in Graphs

Frank Pfenning, André Platzer, Rob Simmons, Penny Anderson

1 Introduction

In this lecture, we will use the minimal graph interface we developed in the last lecture:

```
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);
```
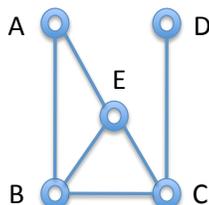
and we will discuss the question of *graph reachability*: given two verticies v and w, does there exist a path from v to w?

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:

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

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.

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 arive 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.
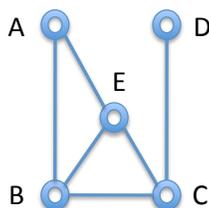

We can go $A - B - E - A - B - E - \cdots$ without ever reaching D (or we can go just $A - B - A - B - \cdots$), even though there exists a path.

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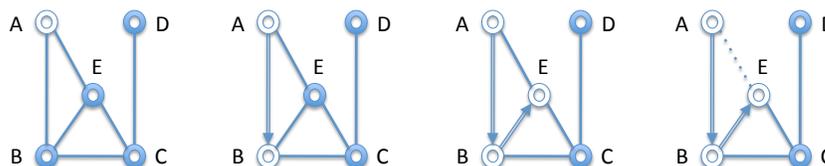
When we are at B we mark B and have three choices for the next step.

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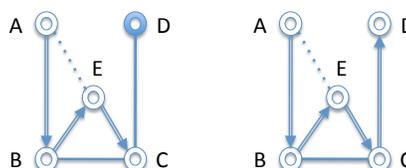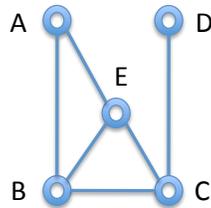
From C we consider the link to D (before considering the link to B) and we arrive at D, declaring success with the path

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

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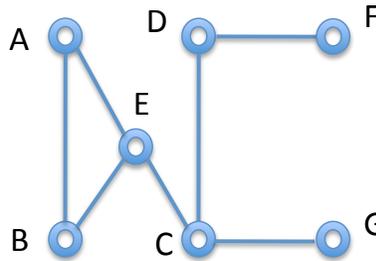
We write the current node we are visiting on the left and on the right a *stack* of nodes we have to return to when we backtrack. For each of these we also remember which choices remain (in parentheses). We annotate marked nodes with an asterisk, which means that we never pick them as the next node to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

For example, at 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 (B, A^*) etc.

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

We can easily write this code recursively, letting the *call stack* keep track of everything we need for backtracking; each `dfs` function body has its own

linked list for the adjacency list. This is the way we wrote the solver for Peg Solitaire; the list of possible moves corresponds to the adjacency list.

```
bool dfs(graph_t G, bool *mark, vertex start, vertex target) {
  REQUIRES(G != NULL && mark != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));
  if(start == target) return true;
  mark[start] = true;
  for(adjlist *L = graph_connections(G, start); L != NULL; L = L->next) {
    if(!mark[L->vert]) {
      mark[L->vert] = true;
      if(dfs(G, mark, L->vert, target)) return true;
  }

  return false;
}
```

4 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. It's not necessary to keep track of the neighbor relationships between them.

   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.

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 dfsearch(graph_t G, vertex source, vertex target) {
  REQUIRES(source < graph_size(G) && target < graph_size(G));
```

```
  stack S = stack_new();
  unsigned int size = graph_size(G);
  bool mark[size];
  for(unsigned int i = 0; i < size; i++)
    mark[i] = false;

  push(S, (void*)(uintptr_t)source);
  mark[source] = true;
  while(!stack_empty(S)) {
    vertex v = (vertex)(uintptr_t)pop(S);
    printf("Visiting %d\n", v);
    if (v == target) {
      stack_free(S, NULL);
      return true;
    }
    for(adjlist *L = graph_connections(G, v); L != NULL; L = L->next) {
      if(!mark[L->vert]) {
        push(S, (void*)(uintptr_t)L->vert);
        mark[L->vert] = true;
      }
    }
  }

  stack_free(S, NULL);
  return false;
}
```

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.

5 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 web page.

Here's an illustration using our running example of search for a path from A to B in the graph
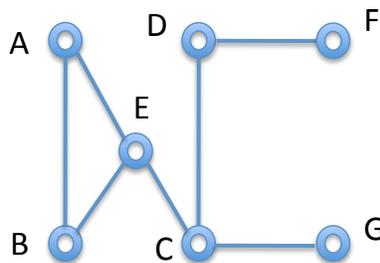
Step	Current	Neighbors	New queue
0			(A^*)
1	A^*	(E, B)	(E^*, B^*)
2	E^*	(B^*, A^*)	$B^*)$
3	B^*	(E^*, A^*)	$()$

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.

6   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.