Lecture Notes on Search in Graphs

15-122: Principles of Imperative Computation
Frank Pfenning, André Platzer, Rob Simmons, Penny Anderson

Lecture 24
April 22, 2014

1 Introduction

In this lecture we introduce *graphs*. Graphs provide a uniform model for many structures, for example, maps with distances or Facebook relationships. Algorithms on graphs are therefore important to many applications. They will be a central subject in the algorithms courses later in the curriculum; here we only provide a very basic foundation for graph algorithms.

With respect to our learning goals we will look at the following notions.

Computational Thinking: Implicit and explicit graphs

Algorithms and Data Structures: Adjacency matrices and adjacency lists; depth-first and breadth-first search

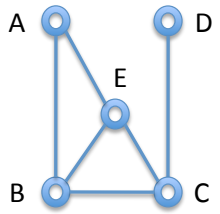
Programming: Flexible array members of structs; recursion versus iteration using an explicit stack.

2 Paths in Graphs

We start with *undirected graphs* which consist of a set V of *vertices* (also called *nodes*) and a set E of *edges*, each connecting two different vertices. In particular, these graphs have no edges from a node back to itself. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction. In a *directed graph* edges provide a connection from one node to another, but not necessarily in the opposite direction. More mathematically, we say that the *edge relation* between vertices is

symmetric for undirected graphs. In this lecture we only discuss undirected graphs, although directed graphs also play an important role in many applications.

The following is a simple example of a connected, undirected graph with 5 vertices (A, B, C, D, E) and 6 edges (AB, BC, CD, AE, BE, CE).



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 Implicit Graphs

There are many, many different ways to represent graphs. In some applications they are never explicitly constructed but remain implicit in the way the problem was solved. One such example was *peg solitaire*. The vertices of the graph implicit in this problem are *board configurations*. There is an edge from A to B if we can make a move in configuration A to reach configuration B. Note that this implicit graph is actually a *directed graph* since the game does not allow us to undo a move we just made. The classical reachability question here would be if from some initial configuration we can reach a given final configuration. We actually solved a related question, namely if we can reach any of a number of alternative configurations (those with exactly one peg) from a given initial configuration. We win the game if we can reach any of those configurations with a single peg.

The reason why we did not explicitly construct the full graph is that for standard boards it is unreasonably large – there are too many reachable configurations. Instead, we incrementally construct nodes in the implicit graph as we search for a solution in the hope we can find a solution without ever generating all nodes. In some examples (like the standard English board), this hope was justified if we were lucky enough to pick a good move strategy. To discover that a board had no solution, however, we still had to visit every reachable configuration. Just because we have 3 pegs remaining with one attempt of trying to solve the board does not mean we could not have been more successful if we had moved the pegs around in a different way.

4 Explicit Graphs and a Graph Interface

Sometimes we *do* want to represent a graph as an explicit set of edges and vertices and in that case we need a graph datatype. In the C code that follows, we'll refer to our vertices with unsigned integers. A minimal interface for graphs would allow us to create and free graphs, check whether an edge exists in the graph, and add a new edge to the graph.

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

graph graph_new(unsigned int numvert);
void graph_free(graph G);
unsigned int graph_size(graph G);
  // number of vertices in the graph
bool graph_hasedge(graph G, vertex v, vertex w);
  //@requires v < graph_size(G) && w < graph_size(w);
void graph_addedge(graph G, vertex v, vertex w);
  //@requires v < graph_size(G) && w < graph_size(w);
  //@requires !graph_hasedge(G, v, w);
```

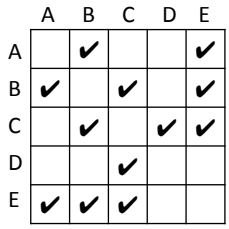
We use the C0 notation for contracts on the interface functions here. Even though C compilers do not recognize the `//@requires` contract and will simply discard it as a comment, the contract still serves an important role for the programmer reading the program. For the graph interface, we decide that it does not make sense to add an edge into a graph when that edge is already there, hence the second precondition.

With this minimal interface, we can create a graph for our running example (letting A = 0, B = 1, and so on).

```
  graph G = graph_new(5);
  graph_addedge(G, 0, 1);
  graph_addedge(G, 1, 2);
  graph_addedge(G, 2, 3);
  graph_addedge(G, 0, 4);
  graph_addedge(G, 1, 4);
  graph_addedge(G, 2, 4);
```

5 Adjacency Matrices

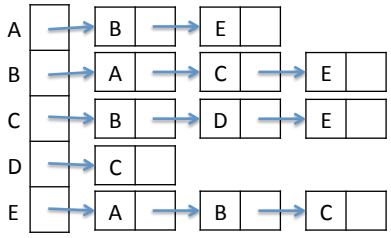
There are two simple ways to implement the graph interface. One way is to represent the graph as a two-dimensional array that represents its edge relation. We can check if there is an edge from B (= 1) to D (= 3) by looking for a checkmark in row 1, column 3. In an undirected graph, the top-right half of this two-dimensional array will be a mirror image of the bottom-left, because the edge relation is symmetric.



This representation of a graph is called an *adjacency matrix*, because it is a matrix that stores which nodes are neighbors.

6 Adjacency Lists

The other classic representation of a graph is as an adjacency *list*. In an adjacency list representation, we have a one-dimensional array that looks much like a hash table. Each vertex has a spot in the array, and each spot in the array contains a linked list of all the other vertices connected to that vertex. Our running example would look like this as an adjacency list:



Adjacenct lists and adjacency matrices have different tradeoffs in the time and space it takes to perform operations. If the matrix would be *sparse*, where there are many vertices and few edges, it usually makes more sense to use an adjacency list. If it would be *dense*, where there are many edges, it may be more efficient to use an adjacency matrix.

If we do use an adjacency list representation, it will often make sense to extend the interface to graphs to give the client access to this linked list of adjacent edges, to represent the neighbors of a given node.

```
typedef struct adjlist_node adjlist;

struct adjlist_node {
  vertex vert;
  adjlist *next;
};

/* Returns a linked list of the neighbors of vertex v.
 * This adjacency list is owned by the graph and should
 * not be modified by the user.
 * @requires(v < graph_size(G)) */
adjlist *graph_connections(graph G, vertex v);
```

One way to implement the adjacency list version of graphs is as a pointer to a special kind of C struct, a struct with a *flexible array member*, i.e. its last field is an array whose length is only specified at runtime, not at the time of declaring the type of the struct. This is a struct with two fields: the first is an unsigned integer representing the actual size, and the second is an array of adjacency lists.

```
struct graph_header {
  unsigned int size;
  adjlist *adj[]; // Flexible array member!
};
```

The array `adj` of adjacency lists will be contiguous in memory with the `size` field.

Usually, structs aren't allowed to contain arrays directly, only pointers, but C99 allows the last member of a struct to be an array of unknown size. The array will be contiguous in memory with the earlier struct elements. (Another example of the use of flexible array members is the "bare" C0 runtime's implementation of C0 arrays. See `C_IDIOMS.txt` in the Lab 8 starter folder.)

The sizeof operator on a struct with a flexible array member returns the size the struct would have if the array had length 0.

To allocate a struct with a flexible array member, you request space for the struct itself plus however much space you want for the array. For instance:

```
struct graph_header *G =
   xcalloc(1, sizeof(struct graph_header) + 5);
```

allocates enough space for the adjacency-list representation of a graph with five nodes. The fields can be accessed as usual using struct-pointer notation:

```
G->size = 5;
...G->adj[0]...
```

We allocate this adjacency list using `xcalloc` to make sure that it is initialized to an empty array. Behind the scenes `xcalloc` just multiplies its two arguments; because we are allocating a struct with a flexible array member, we pass in 1 for the first argument and explicitly figure out the desired size of the array for the second argument:

```
graph graph_new(unsigned int size) {
  size_t adj_size = sizeof(adjlist*) * size;
  graph G = xcalloc(1, sizeof(struct graph_header) + adj_size);
  G->size = size;
  ENSURES(is_graph(G));
  return G;
}
```

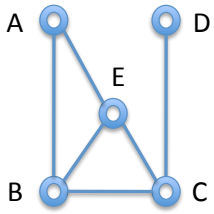
7 Depth-First Search

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

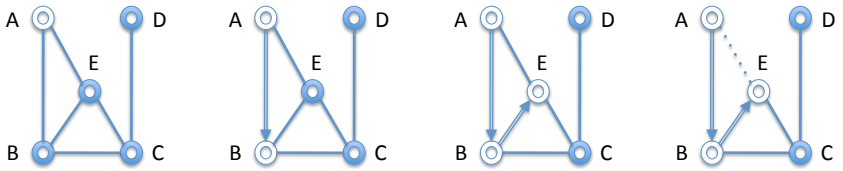
example graph.



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.

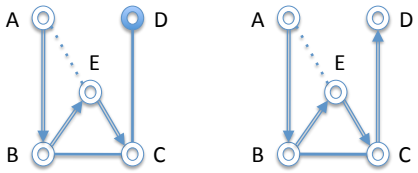
Let's go back to the earlier example and play through this idea while trying to find a path from A to D. We start by marking A (indicated by hollowing the circle) and go to B. We indicate the path we have been following by drawing a double-line along the edges contained in it.



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).

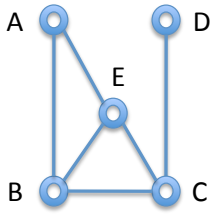


From C we consider the link to D (before considering the link to B) and we arrive at D, declaring success with the path

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

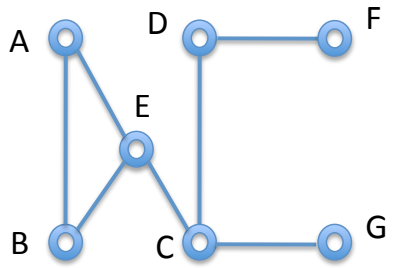
which, by construction, has no cycles.

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)	$E^*\ (B, A^*) \mid A^*\ (B)$	
4	G	(C^*)	$C^*\ (E^*, D) \mid E^*\ (B, A^*) \mid A^*\ (B)$	Backtrack
5	D	(F, C^*)	$C^*\ () \mid E^*\ (B, A^*) \mid A^*\ (B)$	
6	F	(D^*)	$D^*\ (C^*) \mid C^*\ () \mid E^*\ (B, A^*) \mid A^*\ (B)$	Backtrack
7	B	(A^*)	$E^*\ (B, A^*) \mid A^*\ (B)$	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 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;
}
```

8 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 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;
}
```

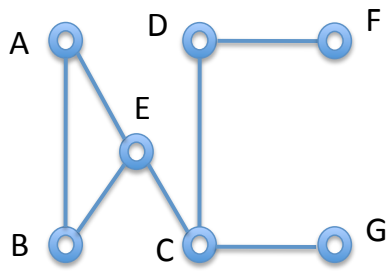
We mark the starting node and push it on the stack. Then we iteratively pop the stack and examine each neighbor of the node we popped. If the neighbor is not already marked, we push it on the stack to make sure we look at it eventually.

9 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.

10 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.