15-122: Principles of Imperative Computation

Recitation 16 — Josh Zimmerman

Representations of graphs

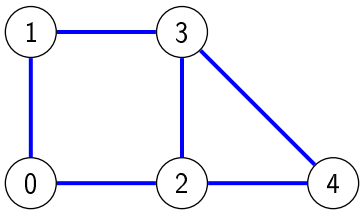
In lecture, we talked about two potential ways to represent graphs: adjacency matrices and adjacency lists.

In all cases we assume that vertices are represented as unsigned ints.

In an adjacency matrix, we keep a two-dimensional array to tell us whether there's an edge between two vertices.

We can implement this as a two-dimensional array of 1s and 0s — a 1 if two vertices are adjacent (connected by a single edge) and a 0 otherwise.

For instance, the adjacency matrix for this graph:



is

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Notice that the matrix is symmetric along the diagonal from top-left to bottom-right. This is true for all undirected graphs, since if there's an edge from u to v, there's also one from v to u in undirected graphs.

We can see if there's an edge between two vertices by looking at the corresponding spot in the matrix. For instance, if we want to know if there's an edge between vertices 1 and 0 we can look at position 1,0 in the matrix — that's row 1, column 0. (The top left is 0,0.)

This representation is great if we have a lot of edges or if we care a lot more about runtime than space usage, since it lets us whether two vertices have an edge between them in constant time. However, we need to store a two dimensional array that has v^2 elements, which is a lot of space.

An adjacency list, on the other hand, saves a lot of space but can be slower to look things up.

For instance, the adjacency list for the above graph is:

0	1,2
1	0,3
2	0,3,4
3	1,2,4
4	2,3

Here, the space is proportional to the number of edges, but looking up whether one vertex is connected to another takes $O(d)$ time, where d is the number of edges out of the first vertex.

Recursive Depth-First Search

Remember the definition of an `struct adjlist`:

```
1 struct adjlist_node {
2   vertex vert;
3   adjlist *next;
4 };
```

With that in mind, let's go through recursive depth first search on the following call. (Assume `G` is the above graph and `mark` is initialized so that only vertex 0 is true and all others are false.)

```
rec_dfsearch(G, mark, 0, 4);
```

```
1  bool rec_dfsearch(graph G, bool *mark, vertex source, vertex target) {
2    REQUIRES(mark != NULL);
3    REQUIRES(source < graph_size(G) && target < graph_size(G));
4    //REQUIRES(graph_size(G) == \length(mark));
5
6    printf("Visiting %u\n", source);
7    if(source == target) return true;
8
9    for(adjlist *L = graph_connections(G, source); L != NULL; L = L->next) {
10     if(!mark[L->vert]) {
11       mark[L->vert] = true;
12       if(rec_dfsearch(G, mark, L->vert, target)) return true;
13     }
14   }
15   return false;
16 }
```

Iterative DFS (with stacks)

Here's the code for iterative DFS. It's very similar to recursive DFS, but it uses a stack that it creates instead of implicitly using the system stack, and avoids the overhead of a recursive call. It also will visit the vertices of the graph in a slightly different order.

Let's go through the same DFS call from above and walk through what happens.

```
1  bool dfsearch(graph G, vertex source, vertex target) {
2    REQUIRES(source < graph_size(G) && target < graph_size(G));
3
4    stack S = stack_new();
5    unsigned int size = graph_size(G);
6    bool mark[size];
7    for(unsigned int i = 0; i < size; i++) {
8      mark[i] = false;
9    }
10
11   push(S, (void*)(uintptr_t)source);
12   mark[source] = true;
13   while(!stack_empty(S)) {
14     vertex v = (vertex)(uintptr_t)pop(S);
15     printf("Visiting %d\n", v);
16     if (v == target) {
```

```
17      stack_free(S, NULL);
18      return true;
19    }
20    for(adjlist *L = graph_connections(G, v); L != NULL; L = L->next) {
21      if(!mark[L->vert]) {
22        push(S, (void*)(uintptr_t)L->vert);
23        mark[L->vert] = true;
24      }
25    }
26  }
27
28  stack_free(S, NULL);
29  return false;
30 }
```

Implementing (LIFO) stacks with priority queues

As we saw on an old theory assignment, we can use priority queues to implement stacks. We do this by choosing appropriate priority values for the elements we insert into the priority queue such that they'll be removed in the order we want them to be to satisfy the LIFO property of a stack.

Specifically, to get a priority queue to behave like a LIFO stack, we insert elements in *decreasing* priority order, so that the thing most recently inserted into the priority queue has lowest priority and will be removed in a delmin operation.

Implementing depth-first search with priority queues

We went over how to implement DFS with stacks above, but we can also use priority queues to implement it, since we can use priority queues to implement stacks.

So, our DFS becomes

1. Pick a vertex to start at. Refer to this as s (for the source).
2. Add s to a priority queue.
3. while the priority queue is nonempty:
   (a) Remove a vertex v from the priority queue.
   (b) If the vertex v is not visited:
      i. Mark v as visited
      ii. If v is the target, exit the search (returning an appropriate value)
      iii. Add all immediate neighbors of v (the vertices connected to v by an edge) to the priority queue, assigning priorities to be smaller than any priority added so far.

We can also assign priorities in different, more interesting ways that let us accomplish different tasks. In fact, we can modify this search algorithm by simply assigning different priorities to implement breadth-first search, best-first search, and even other, more advanced algorithms. (We'd only need to change step 3(b)(iii).)