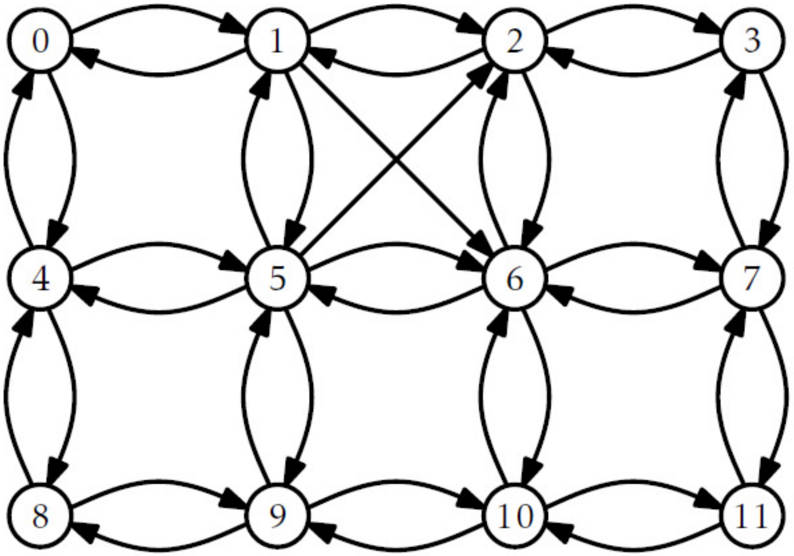
Graph Algorithms

Dr. Said P. Martagón

Outline

Mathematically, a (*directed* graph) is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of ordered pair of vertices called *edges.* An edge (i, j) is *directed* from *itoj*; i is called the *source* of the edge and j is called the *target.* A *path* in G is a sequence of vertices $v_0, \ldots, v_k$ such that, for every $i \in \{1, \ldots, k\}$, the edge (v_{i-1}, v_i) is in E. A path (or cycle) is *simple* if all of its vertices are unique. If there is a path from a some vertex v_i to some vertex v_j then we say that v_j is reachable from v_i.


0
1
2
3
4
5
6
7
8
9
10
11

We will use n to denote the number of vertices of G and m to denote the number of edges of G. That is, $n = |V|$ and $m = |E|$. Furthermore, we will assume that $V = \{0, \ldots, n-1\}$. Any other data that we would like to associate with the elements of V can be stored in an array of length n.

Some typical operations performed on graphs are:

- add_edge(i, j): Add the edge (i, j) to E.
- remove edge(i, j): Remove the edge (i, j) from E.
- has edge(i, j): Check if the edge $(i, j) \in E$.
- out edges(i): Return a List of all integers j such that $(i, j) \in E$.
- in edges(i): Return a List of all integers j such that $(j, i) \in E$.

Outline

An *adjacency matrix* is a way of representing an n vertex graph $G = (V, E)$ by an $n \times n$ matrix, a, whose entries are boolean values.

example

$a \leftarrow new\_boolean_matrix(n, n)$

The matrix entry $a[i][j]$ is defined as

$$a[i][j] = \begin{cases} true & \text{if}(i, j) \in E \\ false & \text{otherwise} \end{cases} \tag{1}$$

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

In this representation, the operations add edge(i, j), remove edge(i, j), and has edge(i, j) just involve setting or reading the matrix entry $a[i][j]$:

add_edge(i,j)

```
$a[i][j] \leftarrow true$
```

remove_edge(i,j)

```
$a[i][j] \leftarrow false$
```

has_edge(i,j)

```
return $a[i][j]$
```

Where the adjacency matrix performs poorly is with the out edges(i) and in_edges(i) operations. To implement these, we must scan all n entries in the corresponding row or column of a and gather up all the indices, j, where $a[i][j]$, respectively $a[j][i]$, is true. These operations clearly take O(n) time per operation.

Another drawback of the adjacency matrix representation is that it is large. It stores an $n \times n$ boolean matrix, so it requires at least n^2 bits of memory. The implementation here uses a matrix of values so it actually uses on the order of n^2 bytes of memory. A more careful implementation, which packs w boolean values into each word of memory, could reduce this space usage to $O(n2/w)$ words of memory.

Theorem

The AdjacencyMatrix data structure implements the Graph interface. An AdjacencyMatrix supports the operations

- add edge(i, j), remove_edge(i, j), and has_edge(i, j) in constant time per operation; and
- in edges(i), and out_edges(i) in O(n) time per operation.

The space used by an AdjacencyMatrix is $O(n^2)$.

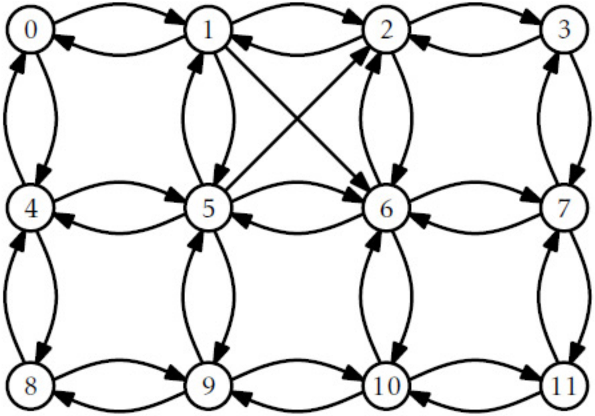
Outline

Adjacency list representations of graphs take a more vertex-centric approach. There are many possible implementations of adjacency lists. In an adjacency list representation, the graph $G = (V, E)$ is represented as an array, *adj*, of lists. The list $adj[i]$ contains a list of all the vertices adjacent to vertex i. That is, it contains every index j such that $(i, j) \in E$.

Algorithm 1 initialize()

```
1: $adj \leftarrow new\_array(n)$
2: for $i = 0$ to $n - 1$ do
3:     $adj[i] \leftarrow ArrayStack()$
4: end for
```

In this particular implementation, we represent each list in *adj* as ArrayStack, because we would like constant time access by position. Other options are also possible. Specifically, we could have implemented *adj* as a DLList.



0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	0	1	5	6	4	8	9	10
4	2	3	7	5	2	2	3	9	5	6	7
	6	6		8	6	7	11		10	11	
	5				9	10					
					4						

The *add_edge(i, j)* operation just appends the value *j* to the list *adj*[*i*]:

Algorithm 2 add_edge(i,j)

```
adj[i].append(j)
```

This takes constant time.
The *remove_edge(i, j)* operation searches through the list *adj*[*i*] until it finds *j* and then removes it:

Algorithm 3 remove_edge(i,j)

```
    for $k = 0$ to $lenth(adj[i]) - 1$ do
       if $adj[i].get(k) = j$ then
3:        $adj[i].remove(k)$
          return
       end if
6:  end for
```

This takes $O(deg(i))$ time, where $deg(i)$ (the degree of i) counts the number of edges in E that have i as their source.

The *has_edge(i, j)* operation is similar; it searches through the list $adj[i]$ until it finds j (and returns true), or reaches the end of the list (and returns false):

Algorithm 4 has_edge(i,j)

```
   for $k \in adj[i]$ do
     if $k = j$ then
       return true
4:   end if
     return false
   end for
```

This also takes $O(deg(i))$ time.

The *out_edges(i)* operation is very simple; it returns the list $adj[i]$:

Algorithm 5 out_edge(i,j)

```
    return $adj[i]$
```

The *in_edges(i)* operation is much more work. It scans over every vertex j checking if the edge (i, j) exists and, if so, adding j to the output list:

Algorithm 6 in_edges(i)

```
    $out \leftarrow ArrayStack()$
    **for** $J = 0$ **to** $n - 1$ **do**
        **if** $has\_edge/i, j)$ **then**
            $out.append(j)$
        **end if**
6:      **return** out
    **end for**
```

This operation is very slow. It scans the adjacency list of every vertex, so it takes $O(n + m)$ time.

The following theorem summarizes the performance of the above data structure:

Theorem

The AdjacencyLists data structure implements the Graph interface. An AdjacencyLists supports the operations

- *add_edge*(i, j) *in constant time per operation;*
- *remove edge(i, j) and has edge(i, j) in* $O(deg(i))$ *time per operation;*
- *in edges(i) in* $O(n + m)$ *time per operation.*

The space used by a AdjacencyLists is $O(n + m)$*.*

Outline

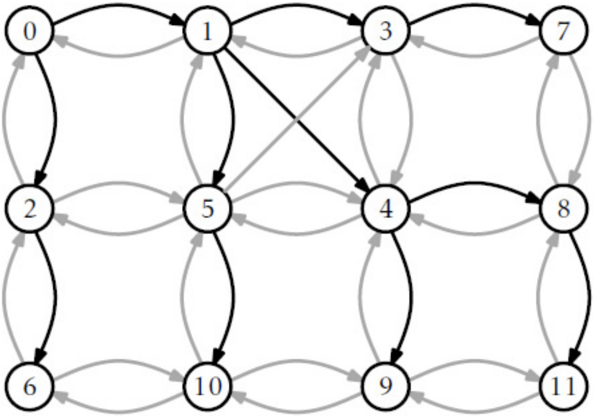
It is presented two algorithms for exploring a graph, starting at one of its vertices, i, and finding all vertices that are reachable from i. Both of these algorithms are best suited to graphs represented using an adjacency list representation. Therefore, when analyzing these algorithms we will assume that the underlying representation is an *AdjacencyLists.*

Outline

This algorithms uses a queue, q, that initially contains only i. It then repeatedly extracts an element from q and adds its neighbours to q, provided that these neighbours have never been in q before.

Algorithm 7 bfs(g,r)

```
$seen \leftarrow new\_boolean\_array(n)$
$q \leftarrow SLList()$
$q.add(r)$
$seen[r] \leftarrow true$
**while** $q.size() > 0$ **do**
   $i \leftarrow q.remove()$
   **for** $j \in g.out\_edges(i)$ **do**
      **if** $seen[j] = false$ **then**
         $q.add(j)$
         $seen[j] \leftarrow true$
      **end if**
   **end for**
**end while**
```



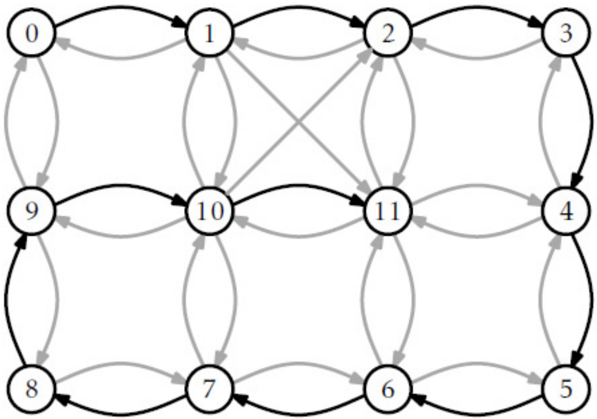
An example of bread-first-search starting at node 0. Nodes are labelled with the order in which they are added to q. Edges that result in nodes being added to q are drawn in black, other edges are drawn in grey.

Outline

During the execution of the depth-first-search algorithm, each vertex, i, is assigned a colour, $c[i]$: white if we have never seen the vertex before, grey if we are currently visiting that vertex, and black if we are done visiting that vertex. The easiest way to think of depth-first-search is as a recursive algorithm. It starts by visiting r. When visiting a vertex i, we first mark i as grey. Next, we scan i's adjacency list and recursively visit any white vertex we find in this list. Finally, we are done processing i, so we colour i black and return.

Algorithm 8 dfs2(g,r)

```
$c \leftarrow new\_array(g.n)$
$s \leftarrow SLList()$
$s.push(r)$
**while** $s.size() > 0$ **do**
    $i \leftarrow s.pop()$
    **if** $c[i] = white$ **then**
        $C[i] \leftarrow grey$
        **for** $j \in g.out\_edges(i)$ **do**
            $s.push(j)$
        **end for**
    **end if**
**end while**
```


0
1
2
3
4
5
6
7
8
9
10
11