CECS 274: Data Structures Graphs

Oscar Morales Ponce

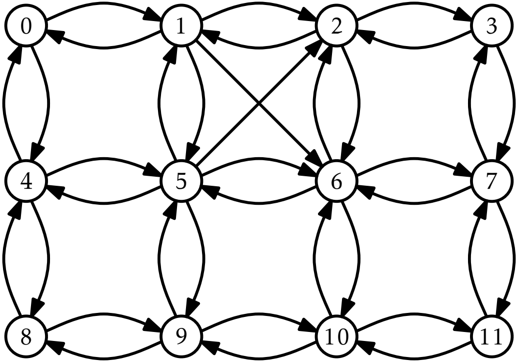
California State University Long Beach

Applications

- In social networks, e.g., Facebook, Instagram, etc,, there is a need of a data structure that supports queries such as
  - List of all friends of a person x?
  - List the friends of the friends of a person x?
  - Given two people x and y, what is the shortest path of friends that connects them?

Applications

Graph Example



Graphs

- a *(directed) graph* is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of ordered pairs of vertices called *edges*.
  - An edge (i, j) is *directed* from i to j; where i is the source and j 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 $v_0, \ldots, v_k$ is a *cycle* if, additionally, the edge (v_k, v_0) is in E. A path (or cycle) is *simple* if all of its vertices are unique.

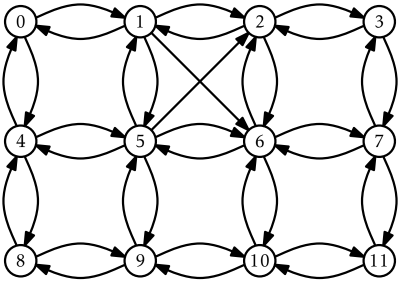
Graph Interface

- $add\_edge(i, j)$: 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 $(i, j) \in E$
- $in\_edges(j)$: Return a List of all integers i such that $(i, j) \in E$

Implementing the Graph Interface using AdjacencyMatrix

- AdjacencyMatrix represents a graph graph $G = (V, E)$ with n vertices
  - State:
    - n: Number of nodes.
    - a: Matrix of dimension 2 of boolean values
  - Invariant: $a[i][j] = \begin{cases} true, \text{ if } (i, j) \in E \\ false, \text{ if } (i, j) \notin E \end{cases}$

AdjacencyMatrix example



	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

AdjacencyMatrix: Initialize

- $initialize()$: allocates memory in a to store $n \times n$ boolean values.

$$
\begin{array}{l}
\text{initialize}() \\
\quad a \leftarrow \text{new\_boolean\_matrix}(n, n)
\end{array}
$$

AdjacencyMatrix: $add\_edge(i, j)$

- $add\_edge(i, j)$: Add the edge (i, j) to E

$$\begin{array}{l} \text{add\_edge}(i, j) \\ \quad a[i][j] \leftarrow true \end{array}$$

AdjacencyMatrix: $remove\_edge(i, j)$:

- $remove\_edge(i, j)$: remove the edge (i, j) from E

$$
\begin{array}{l}
\text{remove\_edge}(i, j) \\
\quad a[i][j] \leftarrow \mathit{false}
\end{array}
$$

AdjacencyMatrix: $has\_edge(i, j)$

- $has\_edge(i, j)$: returns true if the edge (i, j) exists in E

$\mathrm{has\_edge}(i, j)$
    return $a[i][j]$

AdjacencyMatrix: $out\_edges(i)$

- $out\_edge(i)$: returns a list of all integers j such that $(i, j) \in E$

```
out_edges(i)
    l ← ArrayList()
    for j in 0, 1, ..., n − 1 do
        if has_edge(i, j) then
            l.append(j)
    return l
```

AdjacencyMatrix: $in\_edges(j)$

- $in\_edges(j)$: returns a list of all integers i such that $(i, j) \in E$

```
in_edges(j)
    l ← ArrayList()
    for i in 0, 1, ..., n − 1 do
        if has_edge(i, j) then
            l.append(i)
    return l
```

AdjacencyMatrix: Summary

Theorem

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

- $\mathrm{add\_edge}(i, j)$, $\mathrm{remove\_edge}(i, j)$, *and* $\mathrm{has\_edge}(i, j)$ *in constant time per operation; and*
- $\mathrm{in\_edges}(i)$, *and* $\mathrm{out\_edges}(i)$ *in* $O(n)$ *time per operation.*

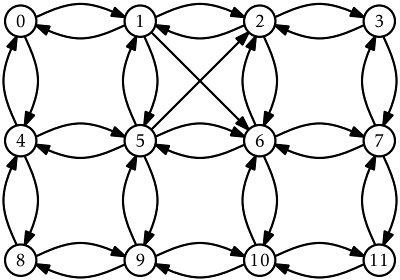
The space used by an AdjacencyMatrix is $O(n^2)$*.*

Implementing the Graph Interface using AdjacencyLists

- AdjacencyLists: represents a graph graph $G = (V, E)$ with n vertices
  - State:
    - n: Number of nodes.
    - adj: Array of lists (ArrayList or Linked-List)
  - Invariant: $adj[i]$ is the list of neighbors, i.,e.,

$$j \text{ is in the list } adj[i] \text{ if and only if } (i, j) \text{ in } E$$

AdjacencyLists example



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						

AdjacencyLists: Initialize

- $initialize()$: allocates n lists adj

```
initialize()
    adj ← new_array(n)
    for i in 0, 1, 2, ..., n − 1 do
        adj[i] ← ArrayList()
```

AdjacencyLists: $add\_edge(i, j)$

- $add\_edge(i, j)$: Add the edge (i, j) to E

$$
\begin{array}{l}
\text{add\_edge}(i, j) \\
\quad adj[i].\text{append}(j)
\end{array}
$$

- How many operations?

AdjacencyLists: $remove\_edge(i, j)$:

- $remove\_edge(i, j)$: remove the edge (i, j) from E

```
remove_edge(i, j)
    for k in 0, 1, 2, ..., length(adj[i]) − 1 do
        if adj[i].get(k) = j then
            adj[i].remove(k)
            return
```

- It takes $O(\deg(i))$ time, where $\deg(i)$ is the degree of i.

AdjacencyLists: $has\_edge(i, j)$

- $has\_edge(i, j)$: returns true if the edge (i, j) exists in E

```
has_edge(i, j)
    for k in 0, 1, 2, ..., adj[i].size() − 1 do
        if k = j then
            return true
    return false
```

- It takes $O(\deg(i))$ time, where $\deg(i)$ is the degree of i.

AdjacencyLists: $out\_edges(i)$

- $out\_edge(i)$: returns a list of all integers j such that $(i, j) \in E$

```
out_edges(i)
    return adj[i]
```

AdjacencyLists: $in\_edges(j)$

- $in\_edges(j)$: returns a list of all integers i such that $(i, j) \in E$

```
in_edges(i)
    out ← ArrayList()
    for j in 0, 1, 2, ..., n − 1 do
        if has_edge(j, i) then out.append(j)
    return out
```

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

AdjacencyLists: Summary

Theorem

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

- $\mathrm{add\_edge}(i, j)$ *in constant time per operation;*
- $\mathrm{remove\_edge}(i, j)$ *and* $\mathrm{has\_edge}(i, j)$ *in* $O(\deg(i))$ *time per operation;*
- $\mathrm{in\_edges}(i)$ *in* $O(n + m)$ *time per operation.*

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

Graph Traversal: Shortest Path from a node

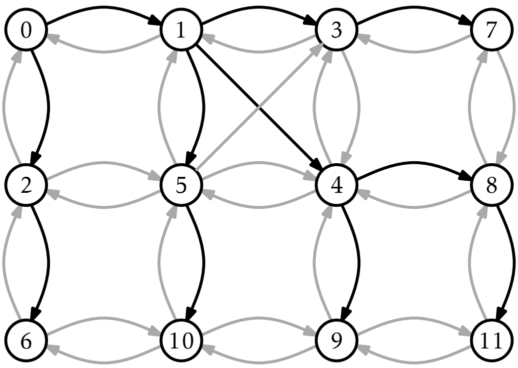
Graph Traversal: Breadth-First Search

- $BFS(i)$: First visit the neighbors of i then the neighbors of the neighbors of i, then the neighbors of the neighbors of the neighbors of i, and so on.
  - State:
    - q: a queue
    - $seen$: an array of boolean values
  - Invariant:
    - $q.peak()$ is the next element to be visited
    - $seen[i] = \begin{cases} true, \text{ if } i \text{ has been visited} \\ false, \text{ otherwise} \end{cases}$
    - For each element i in q, the value of $seen[i]$ is false

Graph Traversal: Breadth-First Search

bfs(g, r)
  $seen \leftarrow \text{new\_array}(n)$
  $q \leftarrow \text{Queue}()$
  $q.\text{add}(r)$
  $seen[r] \leftarrow true$
  while $q.\text{size}() > 0$ **do**
    $i \leftarrow q.\text{remove}()$
    $ngh \leftarrow g.out\_edges(i)$
    for k **in** $0, 1, \cdots, ngh.size() - 1$ **do**
      $j \leftarrow ngh.get(k)$
      if $seen[j] = false$ **then**
        $q.\text{add}(j)$
        $seen[j] \leftarrow true$

Graph Traversal: Breadth-First Search Example



Graph Traversal: Analysis Breadth-First Search

Theorem

When given as input a Graph, g, that is implemented using the AdjacencyLists data structure, the $\mathrm{bfs}(g, r)$ *algorithm runs in* $O(n + m)$ *time.*

Proof.

- Using the array $seen$ ensures that no vertex is added to q more than once.
- Adding (and later removing) each vertex from q takes constant time per vertex for a total of $O(n)$ time.
- Since each vertex is processed at most once, each edge of G is processed at most once. Thus, a total of $O(m)$ time.
- Therefore, the entire algorithm runs in $O(n + m)$ time.

$\square$

Graph Traversal: Shortest path

```
shortestPath(g, r)
    seen ← new_array(n)
    p ← new_array(n)
    q ← Queue()
    q.add(r)
    p[r] ← r
    seen[r] ← true
    while q.size() > 0 do
        i ← q.remove()
        ngh ← g.out_edges(i)
        for k in 0, 1, ··· , ngh.size() − 1 do
            j ← ngh.get(k)
            if seen[j] = false then
                q.add(j)
                seen[j] ← true
                p[j] ← i
    return p
```

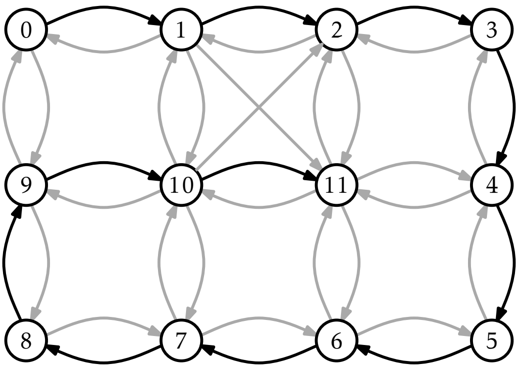
Graph Traversal: Cycles

Graph Traversal: Depth-First Search

- $DFS(i)$: First fully explore one subtree, then a second subtree and so on.
  - State:
    - s: a stack
    - $seen$: an array of boolean values
  - Invariant:
    - $s.top()$ is the next element to be visited
    - $seen[i] = \begin{cases} true, \text{ if } i \text{ has been visited} \\ false, \text{ otherwise} \end{cases}$

```
dfs(g, r)
    seen ← new_array(g.n)
    s ← Stack()
    s.push(r)
    while s.size() > 0 do
        i ← s.pop()
        seen[i] ← true
        ngh ← g.out_edges(i)
        for j in 0, 1, ··· , ngh.size() − 1 do
            if seen[ngh.get(j)] = false then
                s.push(ngh.get(j))
```

Graph Traversal: Depth-First Search Example



Graph Traversal: Analysis Breadth-First Search

Theorem

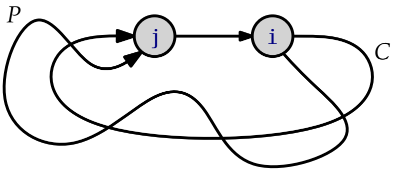
When given as input a Graph, g, that is implemented using the AdjacencyLists data structure, the $\mathrm{dfs}(g, r)$ algorithm runs in $O(n + m)$ time.

Proof.

- Using the array c ensures that no vertex is processed more than once.
- Adding (and later removing) each vertex from s takes constant time per vertex for a total of $O(n)$ time.
- Since each vertex is processed at most once, each edge of G is processed at most once. Thus, a total of $O(m)$ time.
- Therefore, the entire algorithm runs in $O(n + m)$ time.

□

Graph Traversal: Cycle



Graph Traversal: Cycle Detection

$\text{dfs}(g, r)$
  $seen \leftarrow \text{new\_array}(g.n)$
  $s \leftarrow \text{Stack}()$
  $s.\text{push}(r)$
  while $s.\text{size}() > 0$ **do**
    $i \leftarrow s.\text{pop}()$
    $seen[i] \leftarrow true$
    $ngh \leftarrow g.out\_edges(i)$
    for j **in** $0, 1, \cdots, ngh.size() - 1$ **do**
      if $seen[ngh.get(j)] = false$ **then**
        $s.\text{push}(ngh.get(j))$
      else
        $print$(”i and j are in a cycle”)