Graph Search

Lecture 17

March 24, 2020

Part I

Graph Basics

Why Graphs?

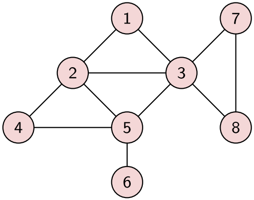
1. Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links), and many problems that don't even look like graph problems.
2. Fundamental objects in Computer Science, Optimization, Combinatorics
3. Many important and useful optimization problems are graph problems
4. Graph theory: elegant, fun and deep mathematics

Graph

Definition

An undirected (simple) graph $G = (V, E)$ is a 2-tuple:

1. V is a set of vertices (also referred to as nodes/points)
2. E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.

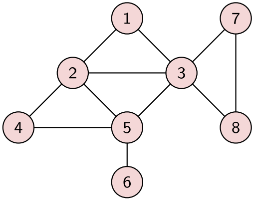


Graph

Definition

An undirected (simple) graph $G = (V, E)$ is a 2-tuple:

1. V is a set of vertices (also referred to as nodes/points)
2. E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.



Example

In figure, $G = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$.

Notation and Convention

Notation

An edge in an undirected graphs is an *unordered* pair of nodes and hence it is a set. Conventionally we use (u, v) for $\{u, v\}$ when it is clear from the context that the graph is undirected.

1. u and v are the end points of an edge $\{u, v\}$
2. **Multi-graphs** allow
   1. *loops*
   2. *multi-edges*

Notation and Convention

Notation

An edge in an undirected graphs is an *unordered* pair of nodes and hence it is a set. Conventionally we use (u, v) for $\{u, v\}$ when it is clear from the context that the graph is undirected.

1. u and v are the end points of an edge $\{u, v\}$
2. **Multi-graphs** allow
   1. *loops*
   2. *multi-edges*
3. In this class we will assume that a graph is a simple graph unless explicitly stated otherwise.

Graph Representation I

Adjacency Matrix

Represent $G = (V, E)$ with n vertices and m edges using a $n \times n$ adjacency matrix A where

1. $A[i, j] = A[j, i] = 1$ if $\{i, j\} \in E$ and $A[i, j] = A[j, i] = 0$ if $\{i, j\} \not\in E$.
2. Advantage: can check if $\{i, j\} \in E$ in $O(1)$ time
3. Disadvantage: needs $\Omega(n^2)$ space even when $m \ll n^2$

Graph Representation II

Adjacency Lists

$G = (V, E)$ with n vertices and m edges:

1. For each $u \in V$ store list $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of u.
   - Sometimes store edges incident to u instead.

Graph Representation II

Adjacency Lists

$G = (V, E)$ with n vertices and m edges:

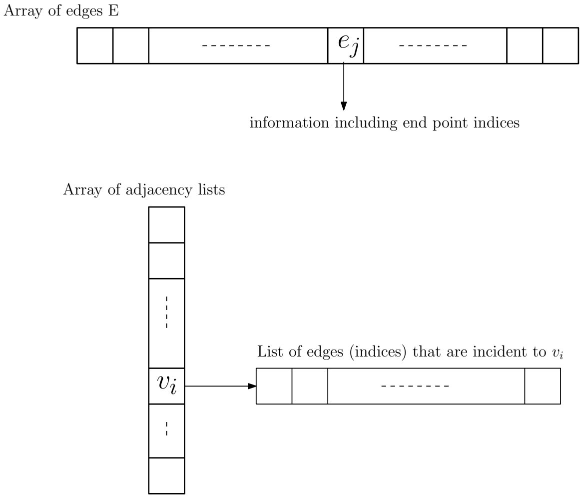
1. For each $u \in V$ store list $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of u.
   - Sometimes store edges incident to u instead.
2. Advantage: space is $O(m + n)$
3. Disadvantage: cannot check in $O(1)$ time if $\{i, j\} \in E$
   1. By sorting each list, one can achieve $O(\log n)$ time
   2. By hashing "appropriately", one can achieve $O(1)$ time

Note: In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.

A Concrete Representation

- Assume vertices are numbered as $\{1, 2, \ldots, n\}$, and edges as $\{1, 2, \ldots, m\}$.
- Edges stored in an array/list of size m. $E[j]$ is j'th edge with info on end points which are integers in range 1 to n.
- Array Adj of size n for adjacency lists. $Adj[i]$ points to adjacency list of vertex i. $Adj[i]$ is a list of edge indices in range 1 to m.

A Concrete Representation

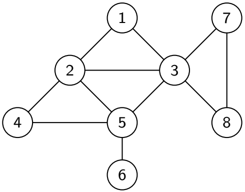


A Concrete Representation: Advantages

- Edges are explicitly represented/numbered. Scanning/processing all edges easy to do.
- Representation easily supports multigraphs including self-loops.
- Explicit numbering of vertices and edges allows use of arrays: $O(1)$-time operations are easy to understand.
- Can also implement via pointer based lists for certain dynamic graph settings.

Connectivity

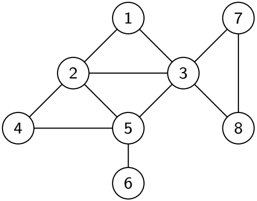
Given a graph $G = (V, E)$:



1. Path: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$.
   - The path is from v_1 to v_k.
   - Length of the path $=$ # edges $= k - 1$
   - A single vertex u is a path of length 0.
2. A vertex u is connected to v if there is a path from u to v.
3. Connected component of u, $\text{con}(u)$: the set of all vertices connected to u. Is $u \in \text{con}(u)$?

Connectivity

Given a graph $G = (V, E)$:

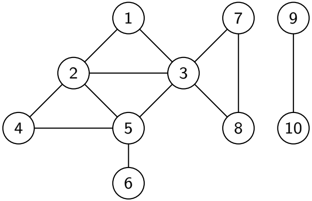


1. Path: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$.
   - The path is from v_1 to v_k.
   - Length of the path $=$ # edges $= k - 1$
   - A single vertex u is a path of length 0.
2. A vertex u is connected to v if there is a path from u to v.
3. Connected component of u, $\text{con}(u)$: the set of all vertices connected to u. Is $u \in \text{con}(u)$?
4. Cycle: in addition, $\{v_1, v_k\} \in E$.
   Note: Single vertex is not a cycle.
   Caveat: Some times people use the term cycle to also allow vertices to be repeated; we will use the term tour.

Connectivity contd

Define a relation C on $V \times V$ as uCv if u is connected to v

1. In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.
2. Graph is connected if only one connected component.



Connectivity Problems

Algorithmic Problems

1. Given graph G and nodes u and v, is u *connected* to v?
2. Given G and node u, find all nodes that are connected to u.
3. Find all connected components of G.

Connectivity Problems

Algorithmic Problems

1. Given graph G and nodes u and v, is u *connected* to v?
2. Given G and node u, find all nodes that are connected to u.
3. Find all connected components of G.

Can be accomplished in $O(m + n)$ time using Breadth-First Search (**BFS**) or Depth-First Search (**DFS**).
BFS and **DFS** are refinements of a basic search procedure which is good to understand on its own.

Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

Explore(G, u):
    array $Visited[1..n]$
    Initialize: Set $Visited[i] = FALSE$ for $1 \leq i \leq n$
    List: $ToExplore$, S

Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

Explore(G, u):
    array $Visited[1..n]$
    Initialize: Set $Visited[i] = FALSE$ for $1 \leq i \leq n$
    List: $ToExplore$, S
    Add u to $ToExplore$ and to S, $Visited[u] = TRUE$
    while ($ToExplore$ is non-empty) do
        Remove node x from $ToExplore$

Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

```
Explore(G,u):
    array Visited[1..n]
    Initialize: Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
```

Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

```
Explore(G, u):
    array Visited[1..n]
    Initialize: Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
```

Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

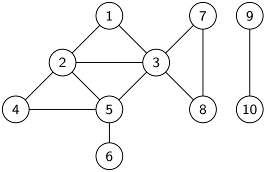
```
Explore(G, u):
    array Visited[1..n]
    Initialize: Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
                Add y to ToExplore
                Add y to S
```

Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

```
Explore(G, u):
    array Visited[1..n]
    Initialize: Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
                Add y to ToExplore
                Add y to S
    Output S
```

Example



Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Termination and runtime:
  Howmany times do we add a node v to ***ToExplore***?

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Termination and runtime:
  Howmany times do we add a node v to ***ToExplore***? **at most Once!** Only if ***Visited***$[v]$ is ***False***. It is then switched to ***TRUE*** and never changed after that.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Termination and runtime:
  Howmany times do we add a node v to ***ToExplore***? **at most Once!** Only if ***Visited***$[v]$ is ***False***. It is then switched to ***TRUE*** and never changed after that. Algorithm terminates in at most n iterations of while loop.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Termination and runtime:
  Howmany times do we add a node v to ***ToExplore***? **at most Once!** Only if ***Visited***$[v]$ is ***False***. It is then switched to ***TRUE*** and never changed after that. Algorithm terminates in at most n iterations of while loop.
- Correctness:
  By induction on iterations, can show $v \in S \Rightarrow v \in \text{con}(u)$

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Termination and runtime:
  Howmany times do we add a node v to ***ToExplore***? **at most Once!** Only if ***Visited***$[v]$ is ***False***. It is then switched to ***TRUE*** and never changed after that. Algorithm terminates in at most n iterations of while loop.
- Correctness:
  By induction on iterations, can show $v \in S \Rightarrow v \in \text{con}(u)$

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Termination and runtime:
  Howmany times do we add a node v to ***ToExplore***? **at most Once!** Only if ***Visited***$[v]$ is ***False***. It is then switched to ***TRUE*** and never changed after that. Algorithm terminates in at most n iterations of while loop.
- Correctness:
  By induction on iterations, can show $v \in S \Rightarrow v \in \text{con}(u)$
  If $v \in S$ then $v \in$ ***ToExplore*** at some point and every edge incident on v was explored $\Rightarrow$ no edges in G leave S.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Termination and runtime:
  Howmany times do we add a node v to ***ToExplore***? **at most Once!** Only if ***Visited***$[v]$ is ***False***. It is then switched to ***TRUE*** and never changed after that. Algorithm terminates in at most n iterations of while loop.
- Correctness:
  By induction on iterations, can show $v \in S \Rightarrow v \in \text{con}(u)$
  If $v \in S$ then $v \in$ ***ToExplore*** at some point and every edge incident on v was explored $\Rightarrow$ no edges in G leave S. Hence no node in $V - S$ is in $\text{con}(u)$.
  Thus $S = \text{con}(u)$ at termination.

□

Properties of Basic Search

Proposition

Explore(G, u) *terminates in* $O(m + n)$ *time.*

Proof: easy exercise

Properties of Basic Search

Proposition

Explore(G, u) *terminates in* $O(m + n)$ *time.*

Proof: easy exercise

BFS and **DFS** are special case of BasicSearch.

1. Breadth First Search (**BFS**): use queue data structure to implementing the list ***ToExplore***
2. Depth First Search (**DFS**): use stack data structure to implement the list ***ToExplore***

Search Tree

One can create a natural search tree T rooted at u during search.

Search Tree

One can create a natural search tree T rooted at u during search.

```
Explore(G, u):
    array Visited[1..n]
    Initialize: Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    Make tree T with root as u
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
                Add y to ToExplore
                Add y to S
                Add y to T with x as its parent
    Output S
```

T is a spanning tree of con(u) rooted at u

Finding all connected components

Exercise: Modify Basic Search to find all connected components of a given graph G in $O(m + n)$ time.

Part II

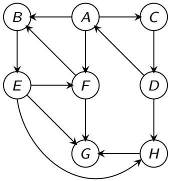
Directed Graphs and Decomposition

Directed Graphs

Definition

A directed graph $G = (V, E)$ consists of

1. set of vertices/nodes V and
2. a set of edges/arcs $E \subseteq V \times V$.



An edge is an *ordered pair* of vertices. (u, v) different from (v, u).

Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

1. Road networks with one-way streets.
2. Web-link graph: vertices are web-pages and there is an edge from page p to page p' if p has a link to p'. Web graphs used by Google with PageRank algorithm to rank pages.
3. Dependency graphs in variety of applications: link from x to y if y depends on x. Make files for compiling programs.
4. Program Analysis: functions/procedures are vertices and there is an edge from x to y if x calls y.

Directed Graph Representation

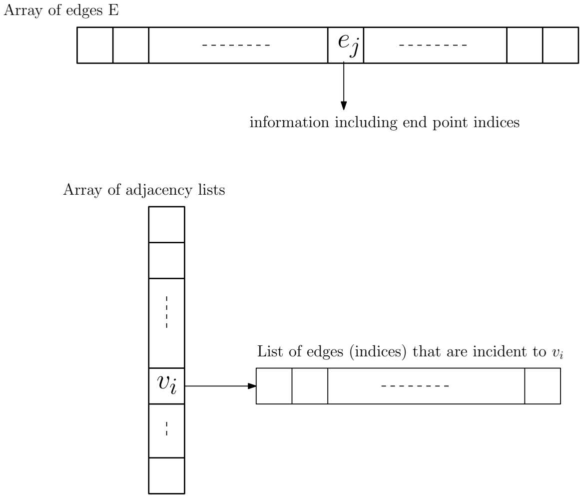
Graph $G = (V, E)$ with n vertices and m edges:

1. **Adjacency Matrix**: $n \times n$ *asymmetric* matrix A. $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ if $(u, v) \notin E$. $A[u, v]$ is not same as $A[v, u]$.
2. **Adjacency Lists**: for each node u, $Out(u)$ (also referred to as $Adj(u)$) and $In(u)$ store out-going edges and in-coming edges from u.

Default representation is adjacency lists.

A Concrete Representation for Directed Graphs

Concrete representation discussed previously for undirected graphs easily extends to directed graphs.



Directed Connectivity

Given a graph $G = (V, E)$:

1. A **(directed) path** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k.
   By convention, a single node u is a path of length 0.
2. A vertex u can reach v if there is a path from u to v.
3. Let $\mathbf{rch}(u)$ be the set of all vertices reachable from u.

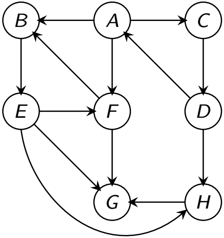
Directed Connectivity

Given a graph $G = (V, E)$:

1. A **(directed) path** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i \le k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k.
   By convention, a single node u is a path of length 0.
2. A vertex u can reach v if there is a path from u to v.
3. Let $\mathbf{rch}(u)$ be the set of all vertices reachable from u.
4. A **cycle** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i \le k - 1$ and $(v_k, v_1) \in E$.
   By convention, a single node u is not a cycle.

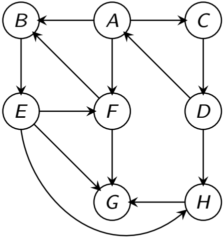
Connectivity contd

Asymmetricity: D can reach B but B cannot reach D



Connectivity contd

Asymmetricity: D can reach B but B cannot reach D



Questions:

1. Is there a notion of connected components?
2. How do we understand connectivity in directed graphs?

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

Define relation C where uCv if u is (strongly) connected to v.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

Define relation C where uCv if u is (strongly) connected to v.

Proposition

C is an equivalence relation, that is reflexive, symmetric and transitive.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

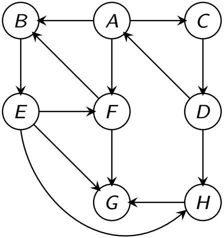
Define relation C where uCv if u is (strongly) connected to v.

Proposition

C is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of C: *strong connected components* of G.
They *partition* the vertices of G.
$\mathrm{SCC}(u)$: strongly connected component containing u.

Strongly Connected Components: Example



Directed Graph Connectivity Problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\text{rch}(u)$.
3. Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.
4. Find the strongly connected component containing node u, that is $\mathrm{SCC}(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute *all* strongly connected components of G.

Exercise

Prove the following:

Proposition

Let $S = \mathrm{rch}(u)$. There is no edge $(x, y) \in E$ where $x \in S$ and $y \notin S$.

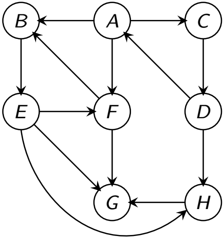
Describe an example where $\mathrm{rch}(u) \neq V$ and there are edges from $V \setminus \mathrm{rch}(u)$ to $\mathrm{rch}(u)$.

Basic Graph Search in Directed Graphs

Given $G = (V, E)$ a directed graph and vertex $u \in V$. Let $n = |V|$.

```
Explore(G,u):
    array Visited[1..n]
    Initialize:  Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List:  ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    Make tree T with root as u
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x,y) in Adj(x) do (edge x → y)
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
                Add y to ToExplore
                Add y to S
                Add y to T with edge (x,y)
    Output S
```

Example



Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = rch(u)$.

Proof Sketch.

Termination and run-time:

A node is added to ***ToExplore*** only if ***Visited***$[v]$ is ***FALSE***. After that ***Visited***$[v]$ is immediately set to ***TRUE*** and is never changed. Hence, it is explored at most once. Thus algorithm terminates in at most n iterations of while loop.

Correctness:

By induction on iterations, can show $v \in S \Rightarrow v \in \text{rch}(u)$

If $v \in S$ then $v \in$ ***ToExplore*** and every outgoing edge from v is explored $\Rightarrow$ no edge leaves S.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = rch(u)$.

Proof Sketch.

Termination and run-time:

A node is added to ***ToExplore*** only if $\textit{Visited}[v]$ is ***FALSE***. After that $\textit{Visited}[v]$ is immediately set to ***TRUE*** and is never changed. Hence, it is explored at most once. Thus algorithm terminates in at most n iterations of while loop.

Correctness:

By induction on iterations, can show $v \in S \Rightarrow v \in \text{rch}(u)$

If $v \in S$ then $v \in$ ***ToExplore*** and every outgoing edge from v is explored $\Rightarrow$ no edge leaves S. Hence no node in $V - S$ is in $\text{rch}(u)$.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = rch(u)$.

Proof Sketch.

Termination and run-time:

A node is added to ***ToExplore*** only if ***Visited***$[v]$ is ***FALSE***. After that ***Visited***$[v]$ is immediately set to ***TRUE*** and is never changed. Hence, it is explored at most once. Thus algorithm terminates in at most n iterations of while loop.

Correctness:

By induction on iterations, can show $v \in S \Rightarrow v \in \text{rch}(u)$

If $v \in S$ then $v \in$ ***ToExplore*** and every outgoing edge from v is explored $\Rightarrow$ no edge leaves S. Hence no node in $V - S$ is in $\text{rch}(u)$.

Caveat: In directed graphs edges can enter S.

Thus $S = \text{rch}(u)$ at termination. □

Properties of Basic Search

Proposition

Explore(G, u) *terminates in* $O(m + n)$ *time.*

Proposition

T *is a search tree rooted at* u *containing* S *with edges directed away from root to leaves.*

Proof: easy exercises

BFS and **DFS** are special case of Basic Search.

1. Breadth First Search (**BFS**): use queue data structure to implementing the list ***ToExplore***
2. Depth First Search (**DFS**): use stack data structure to implement the list ***ToExplore***

Directed Graph Connectivity Problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\text{rch}(u)$.
3. Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.
4. Find the strongly connected component containing node u, that is $\text{SCC}(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute *all* strongly connected components of G.

Directed Graph Connectivity Problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\text{rch}(u)$.
3. Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.
4. Find the strongly connected component containing node u, that is $\mathrm{SCC}(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute *all* strongly connected components of G.

First five problems can be solved in $O(n + m)$ time by via Basic Search (or **BFS**/**DFS**). The last one can also be done in linear time but requires a rather clever **DFS** based algorithm.

Algorithms via Basic Search - I

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\text{rch}(u)$.

Use $\textit{Explore}(G, u)$ to compute $\text{rch}(u)$ in $O(n + m)$ time.

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$.

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.

   Naive: $O(n(n+m))$

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.

   Naive: $O(n(n + m))$

Definition (Reverse graph.)

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$.

   Naive: $O(n(n + m))$

Definition (Reverse graph.)

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute $\mathrm{rch}(u)$ in G^{rev}!

1. **Correctness:** exercise
2. **Running time:** $O(n + m)$ to obtain G^{rev} from G and $O(n + m)$ time to compute $\mathrm{rch}(u)$ via Basic Search.

If both $Out(v)$ and $In(v)$ are available at each v then no need for G^{rev}. Instead of $Adj(v) = Out(v)$, just use $In(v)$.

Algorithms via Basic Search - III

$\text{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

$\mathrm{SCC}(G, u) = \mathrm{rch}(G, u) \cap \mathrm{rch}(G^{rev}, u)$

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

$\mathrm{SCC}(G, u) = \mathrm{rch}(G, u) \cap \mathrm{rch}(G^{rev}, u)$

Hence, $\mathrm{SCC}(G, u)$ can be computed with $\mathit{Explore}(G, u)$ and $\mathit{Explore}(G^{rev}, u)$. Total $O(n + m)$ time.

Why can $\mathrm{rch}(G, u) \cap \mathrm{rch}(G^{rev}, u)$ be done in $O(n)$ time?

Algorithms via Basic Search - IV

1. Is G strongly connected?

Algorithms via Basic Search - IV

1. Is G strongly connected?

Pick arbitrary vertex u. Check if $\mathrm{SCC}(G, u) = V$.

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

Question: Why doesn't removing one strong connected components affect the other strong connected components?

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

Question: Why doesn't removing one strong connected components affect the other strong connected components?

Running time: $O(n(n+m))$.

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

Question: Why doesn't removing one strong connected components affect the other strong connected components?

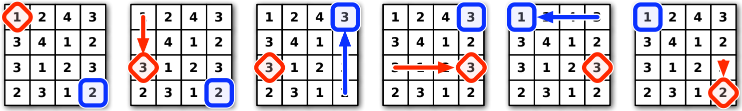
Running time: $O(n(n+m))$.

Question: Can we do it in $O(n+m)$ time?

Modeling Problems as Search

The following puzzle was invented by the infamous Mongolian puzzle-warrior Vidrach Itky Leda in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

In a single turn, you may move either token up, right, down, or left *by a distance determined by the **other** token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move a token off the grid or to the same square as the other token.



A five-move solution for a 4×4 Vidrach Itky Leda puzzle.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle above, your algorithm would return the number 5.

Undirected vs Directed Connectivity

Consider following problem.

- Given *undirected* graph $G = (V, E)$.
- Two subsets of nodes $R \subset V$ (red nodes) and $B \subset V$ (blue nodes). R and B non-empty.
- Describe linear-time algorithm to decide whether *every* red node can reach *every* blue node.

Undirected vs Directed Connectivity

Consider following problem.

- Given *undirected* graph $G = (V, E)$.
- Two subsets of nodes $R \subset V$ (red nodes) and $B \subset V$ (blue nodes). R and B non-empty.
- Describe linear-time algorithm to decide whether *every* red node can reach *every* blue node.

How does the problem differ in directed graphs?

Undirected vs Directed Connectivity

Consider following problem.

- Given *directed* graph $G = (V, E)$.
- Two subsets of nodes $R \subset V$ (red nodes) and $B \subset V$ (blue nodes).
- Describe linear-time algorithm to decide whether *every* red node can be reached by *some* blue node.