Independent Sets in Trees and Graph Basics

Lecture 15

How to design DP algorithms

1. Find a "smart" recursion (The hard part)
    1. Formulate the sub-problem
    2. so that the number of distinct subproblems is small; polynomial in the original problem size.

How to design DP algorithms

1. Find a "smart" recursion (The hard part)
    1. Formulate the sub-problem
    2. so that the number of distinct subproblems is small; polynomial in the original problem size.

2. Memoization
    1. Identify distinct subproblems
    2. Choose a memoization data structure
    3. Identify dependencies and find a good evaluation order
    4. An iterative algorithm replacing recursive calls with array lookups

Which data structure?

So far our memoization uses multi-dimensional arrays:

- Fibonacci numbers, 1-D array
- Text segmentation, suffix, 1-D array
- Longest increasing subsequence, suffix+index, 2-D array
- Edit distance, two prefixes, 2-D array

Which data structure?

So far our memoization uses multi-dimensional arrays:

- Fibonacci numbers, 1-D array
- Text segmentation, suffix, 1-D array
- Longest increasing subsequence, suffix+index, 2-D array
- Edit distance, two prefixes, 2-D array

Not always true.

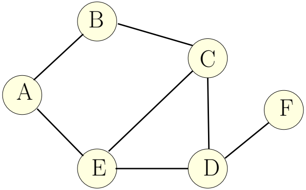
Part I

Maximum Weight Independent Set in Trees

Independent Set in a Graph

Definition

Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is an independent set if there are no edges between nodes in S. That is, if $u, v \in S$ then $(u, v) \notin E$.

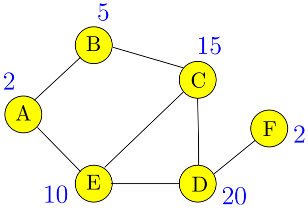


Some independent sets in graph above: $\{D\}, \{A, C\}, \{B, E, F\}$

Maximum Weight Independent Set Problem

Input Graph $G = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

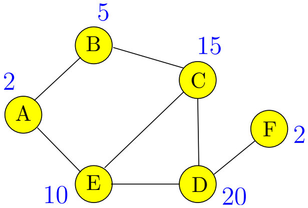
Goal Find maximum weight independent set in G



Maximum Weight Independent Set Problem

Input Graph $G = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal Find maximum weight independent set in G

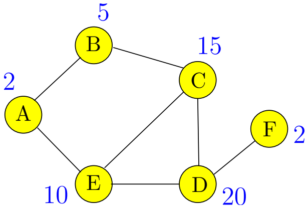


Some independent sets in graph above: $\{D\}, \{A, C\}, \{B, E, F\}$

Maximum Weight Independent Set Problem

Input Graph $G = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal Find maximum weight independent set in G



Some independent sets in graph above: $\{D\}, \{A, C\}, \{B, E, F\}$
Maximum weight independent set in above graph: $\{B, D\}$

Maximum Weight Independent Set Problem

- Finding the largest independent set in an arbitrary graph is extremely hard
- the canonical NP-hard problem

Backtracking

Convert into a sequence of decision problems.

Backtracking

Convert into a sequence of decision problems.

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Decision problem: to include v_n or not

Backtracking

Convert into a sequence of decision problems.

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Decision problem: to include v_n or not
3. Try all possibilities and let the recursion fairy take care of the remaining decisions
4. Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).

Backtracking

Convert into a sequence of decision problems.

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Decision problem: to include v_n or not
3. Try all possibilities and let the recursion fairy take care of the remaining decisions
4. Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).
5. If graph G is arbitrary there is no good ordering that resulted in a small number of subproblems.

Maximum Weight Independent Set Problem

- Finding the largest independent set in an arbitrary graph is extremely hard
- the canonical NP-hard problem

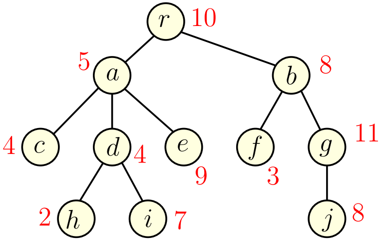
Maximum Weight Independent Set Problem

- Finding the largest independent set in an arbitrary graph is extremely hard
- the canonical NP-hard problem
- But in some special classes of graphs, we can find largest independent sets quickly
- when the input graph is a tree with n vertices, we can compute in O(n) time

Maximum Weight Independent Set in a Tree

Input Tree $T = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal Find maximum weight independent set in T



Backtracking

Convert into a sequence of decision problems.

Backtracking

Convert into a sequence of decision problems.

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Decision problem: to include v_n or not
3. Try all possibilities and let the recursion fairy take care of the remaining decisions
4. Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).
5. If graph G is arbitrary there is no good ordering that resulted in a small number of subproblems.

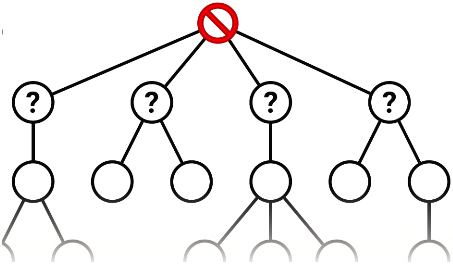
Backtracking

Convert into a sequence of decision problems.

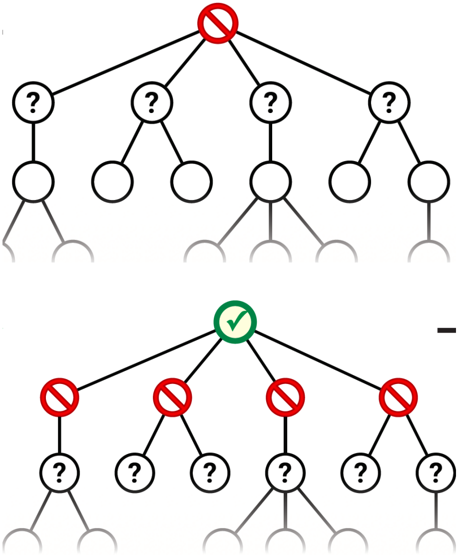
1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Decision problem: to include v_n or not
3. Try all possibilities and let the recursion fairy take care of the remaining decisions
4. Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).
5. If graph G is arbitrary there is no good ordering that resulted in a small number of subproblems.

What is special about a tree?

Optimal substructure



Optimal substructure



Optimal substructure

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) =$$

Optimal substructure

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

Optimal substructure

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

Is it a smart recursion?

Optimal substructure

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

Is it a smart recursion? How many distinct subproblems?

Optimal substructure

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

Is it a smart recursion? How many distinct subproblems? $O(n)$

Optimal substructure

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

Is it a smart recursion? How many distinct subproblems? $O(n)$

Base case: Reach a leaf of the tree

Optimal substructure

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

Is it a smart recursion? How many distinct subproblems? $O(n)$

Base case: Reach a leaf of the tree

What data structure to memoize this recurrence?

Optimal substructure

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

Is it a smart recursion? How many distinct subproblems? $O(n)$

Base case: Reach a leaf of the tree

What data structure to memoize this recurrence? A tree

Order of evaluation

1. Compute $OPT(u)$ bottom up. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u
2. What is an ordering of nodes of a tree T to achieve above?

Order of evaluation

1. Compute $OPT(u)$ bottom up. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u
2. What is an ordering of nodes of a tree T to achieve above? Post-order traversal of a tree.

Iterative Algorithm

MIS-Tree(T):

    Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
    for $i = 1$ to n **do**

$$M[v_i] = \max \begin{pmatrix} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{pmatrix}$$

    return $M[v_n]$ (* Note: v_n is the root of T *)

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

return $M[v_n]$ (* Note: v_n is the root of T *)

Space:

Iterative Algorithm

```
MIS-Tree(T):
    Let v_1, v_2, ..., v_n be a post-order traversal of nodes of T
    for i = 1 to n do
```

$$M[v_i] = \max \begin{pmatrix} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{pmatrix}$$

```
    return M[v_n]  (* Note: v_n is the root of T *)
```

Space: $O(n)$ to store the value at each node of T

Running time:

Iterative Algorithm

MIS-Tree(T):
    Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
    for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

    return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.

Iterative Algorithm

```
MIS-Tree(T):
    Let v_1, v_2, ..., v_n be a post-order traversal of nodes of T
    for i = 1 to n do
        M[v_i] = max( sum_{v_j child of v_i} M[v_j],
                      w(v_i) + sum_{v_j grandchild of v_i} M[v_j] )
    return M[v_n] (* Note: v_n is the root of T *)
```

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Part II

Graph Basics

Why Graphs?

1. Many important and useful optimization problems are graph problems

Why Graphs?

1. Many important and useful optimization problems are graph problems
2. Two levels of resolution:

Why Graphs?

1. Many important and useful optimization problems are graph problems
2. Two levels of resolution:
   1. Classic graph algorithms
   2. How to model a problem as a graph problem and solve it using the classic algorithms

Example: Medieval road network

Example: Modeling Problems as Search

State Space Search

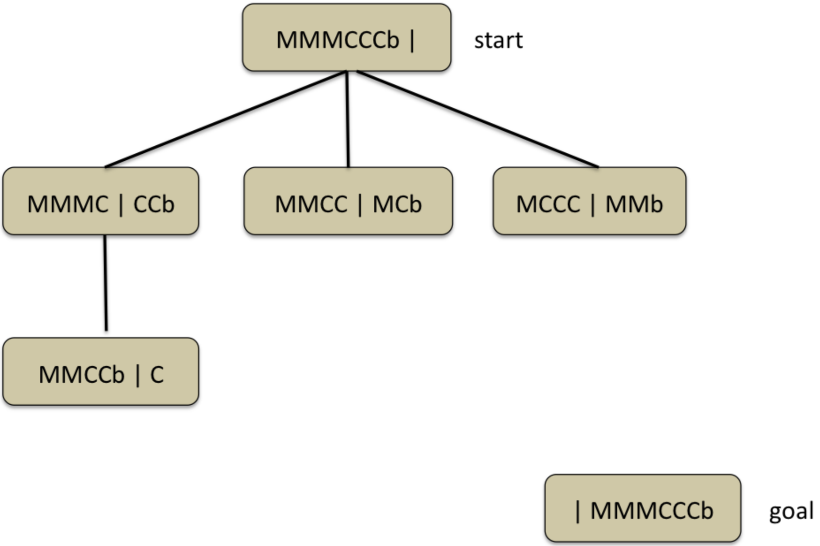
Many search problems can be modeled as search on a graph.
The trick is figuring out what the vertices and edges are.

Missionaries and Cannibals

- Three missionaries, three cannibals, one boat, one river
- Boat carries two people, must have at least one person
- Must all get across
- At no time can cannibals outnumber missionaries

How is this a graph search problem?
What are the vertices?
What are the edges?

Example: Missionaries and Cannibals Graph

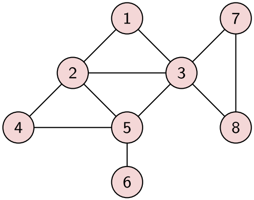


Graph

Definition

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

1. V is a set of vertices (also referred to as nodes)
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

Graph is just a way of encoding pairwise relationships.

Notation and Convention

Graph is just a way of encoding pairwise relationships.

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 $\{u, v\}$

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 $\{i, j\} \notin E$.

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 $\{i,j\} \notin E$.
2. Advantage: can check if $\{i,j\} \in E$ in $O(1)$ time

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

Represent $G = (V, E)$ with n vertices and m edges using adjacency lists:

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

Graph Representation II

Adjacency Lists

Represent $G = (V, E)$ with n vertices and m edges using adjacency lists:

1. For each $u \in V$, $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of u. Sometimes $\text{Adj}(u)$ is the list of edges incident to u.
2. Advantage: space is $O(m + n)$

Graph Representation II

Adjacency Lists

Represent $G = (V, E)$ with n vertices and m edges using adjacency lists:

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

Graph Representation II

Adjacency Lists

Represent $G = (V, E)$ with n vertices and m edges using adjacency lists:

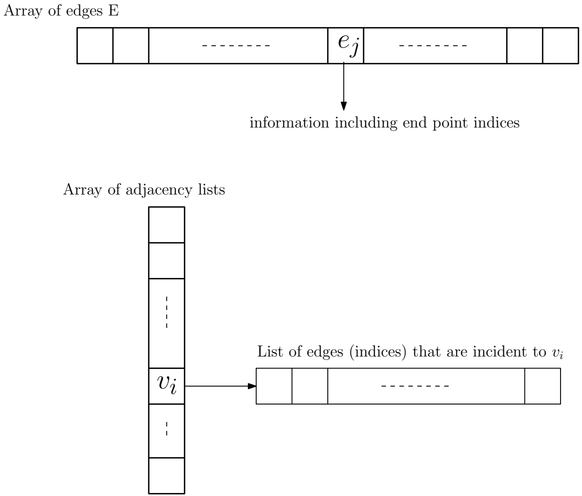
1. For each $u \in V$, $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of u. Sometimes $\text{Adj}(u)$ is the list of edges incident to u.
2. Advantage: space is $O(m + n)$
3. Disadvantage: cannot "easily" determine in $O(1)$ time whether $\{i, j\} \in E$
   1. By sorting each list, one 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 arbitrarily as $\{1, 2, \ldots, n\}$.
- Edges are numbered arbitrarily 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



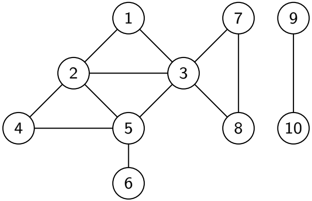
Connectivity Problems

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 on Undirected Graphs

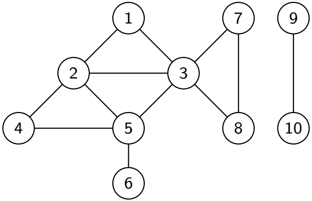
Given a graph $G = (V, E)$:



A 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 $k - 1$ (the number of edges in the path) and the path is from v_1 to v_k.
Note: a single vertex u is a path of length 0.

Connectivity on Undirected Graphs

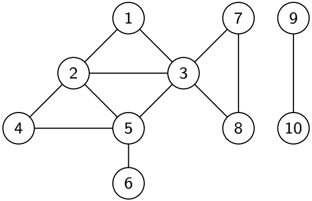
Given a graph $G = (V, E)$:



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 $1 \leq i \leq k - 1$ and $\{v_1, v_k\} \in E$. Single vertex not a cycle according to this definition.

Connectivity on Undirected Graphs

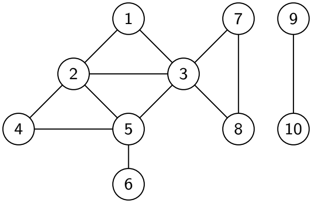
Given a graph $G = (V, E)$:



A vertex u is connected to v if there is a path from u to v.

Connectivity on Undirected Graphs

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



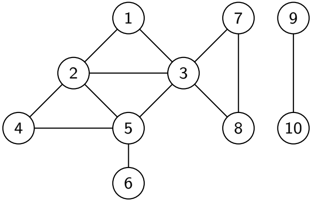
A vertex u is connected to v if there is a path from u to v.

The connected component of u, $\text{con}(u)$, is the set of all vertices connected to u.

Connectivity on Undirected Graphs

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 on Undirected Graphs

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 on Undirected Graphs

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 **BFS** or **DFS**.
BFS and **DFS** are refinements of a basic search procedure which is good to understand on its own.