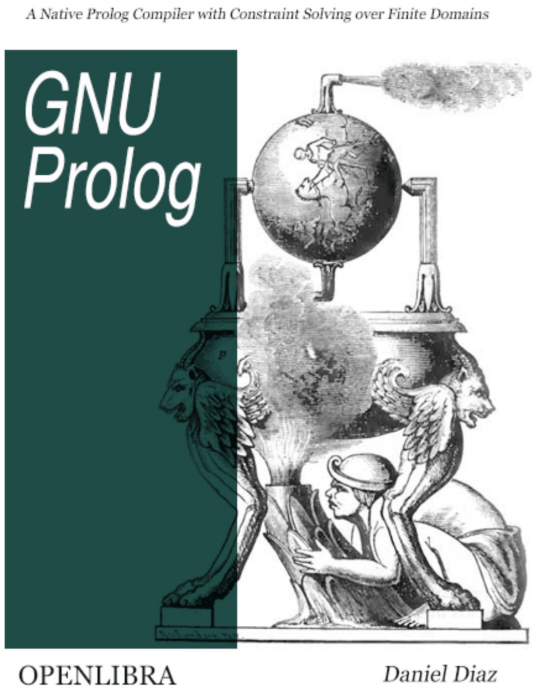
CS 4100: Introduction to AI

Wayne Snyder
Northeastern University

Lecture 7: Problem Solving with Search; Uninformed Search



Problem Solving with Uninformed Search

Today's Lecture on Uninformed Search:

- Review of basic notions: trees, graphs, recursive tree traversals
- Non-recursive tree traversals using stacks and queues; depth-first vs breadth-first vs best-first (a glimpse at heuristic search)
- Iterative deepening
- Graph search (same ideas but with graphs!)

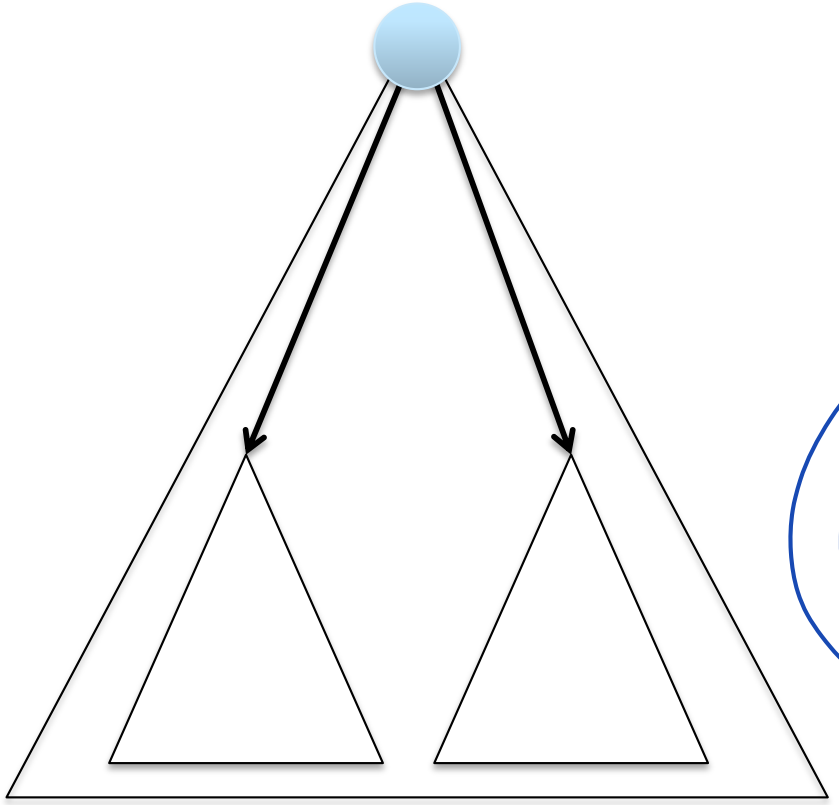
Binary Trees

Binary Trees are an inherently recursive data structure and often manipulated by recursive algorithms:

Recursive Definition:

A Binary Tree is either

- Null (empty tree); or
- A node containing data, with pointers left and right to two Binary Trees

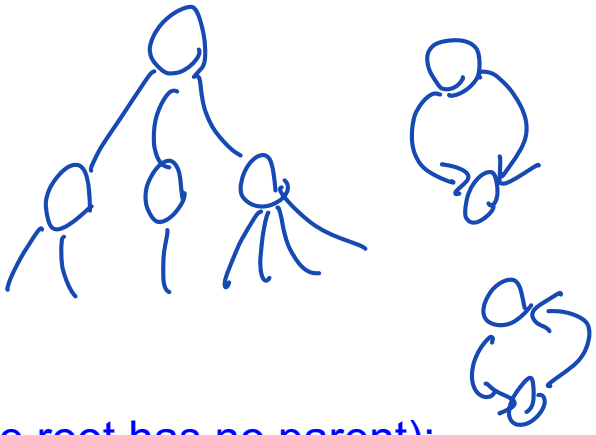


Note: Code in this lecture is in Java!

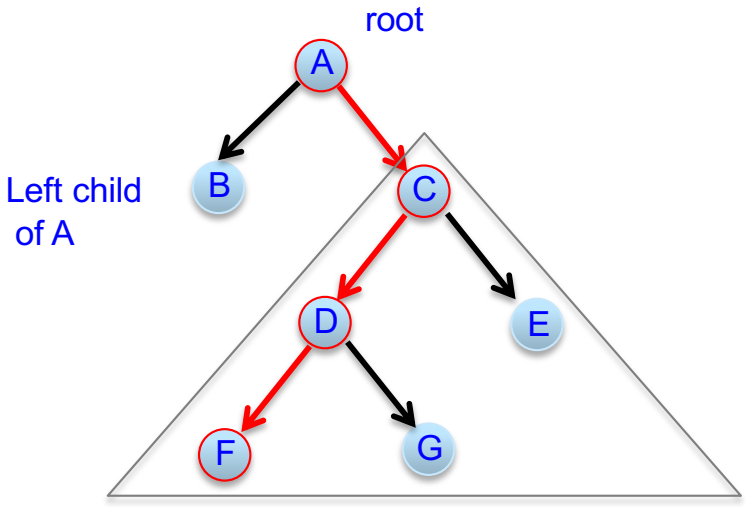
```
class Node {
    int item;
    Node left;
    Node right;
}
```

Binary Trees

Some basic definitions:



- The root is the node at the top of the tree.
- Trees under a node are called subtrees of that node;
- The size of a tree is the number of nodes in it;
- If A points to B, then B is called the child of A;
- The parent of a node is the (unique) node which points to it (the root has no parent);
- A node is a leaf node if it has no children.



Root is A

Size is 7

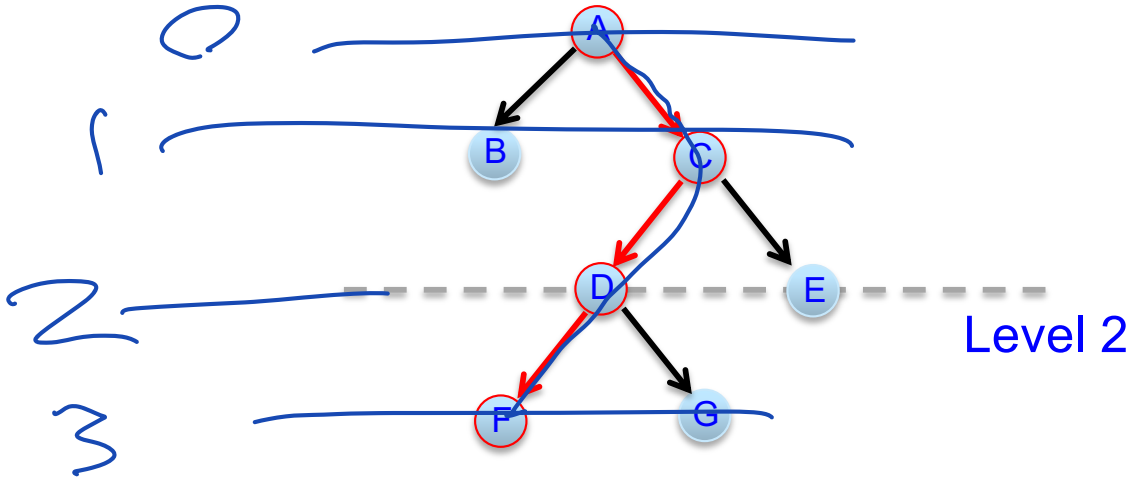
C is parent of D

Leaf nodes: B, F, G, E

Binary Trees

Some basic definitions:

- A path is a sequence of nodes connected by pointers (from parent to child);
- The length of the path is the number of links;
- If there is a path from A to B of length at least one, then A is an ancestor of B and B is a descendant of A.
- The depth of a node in a tree is the number of links on the path from the root;
- The height of a tree is the maximum depth among any of its nodes (or: the length of the longest path).
- Level K in a tree is all the nodes of depth K.



Path from A to F in red, of length 3

D is descendant of A

C is ancestor of G

Depth of D = 2

Height of tree = 3

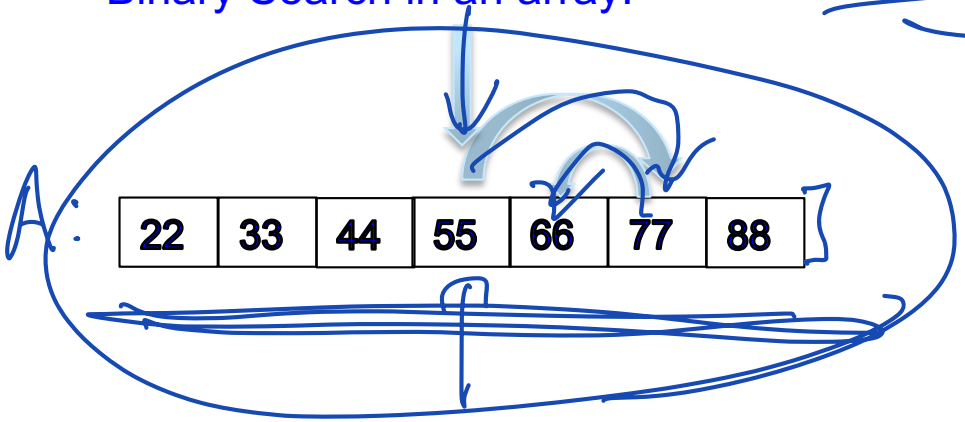
State Space Search

Let's question one of our basic assumptions:

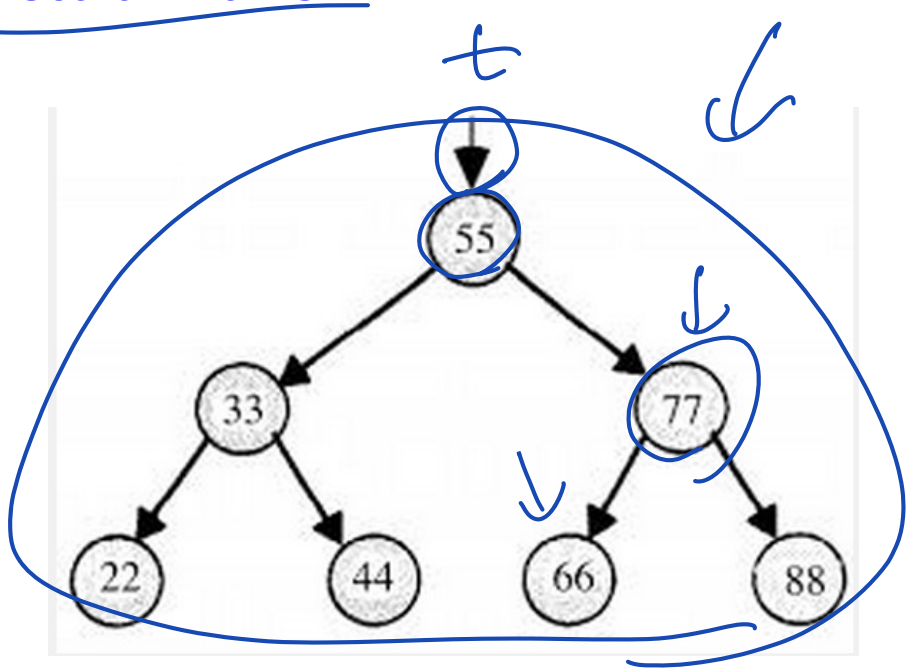
DO NODES In TREES and HAVE TO EXIST?

Well.... not really....many computations follow a "tree structure" without having to actually construct trees of explicit nodes….

Binary Search in an array:



Search in a BST:

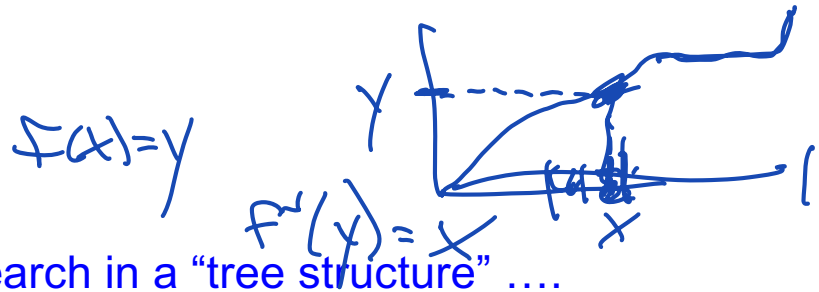


Of course, we DO have a data structure

to search, it just isn't exactly a tree….

Are data structures necessary for search?

State Space Search



N	N*log(N)
5	11.6096405
2.5	3.30482024
3.75	7.15083973
3.125	5.13705059
2.8125	4.19583683
2.96875	4.66050884
3.046875	4.89733455
3.0859375	5.01683588
3.06640625	4.95699548
3.07617188	4.98689332
3.08105469	5.00185902

Here's another example of binary search in a "tree structure" …. without any data structure to search…..

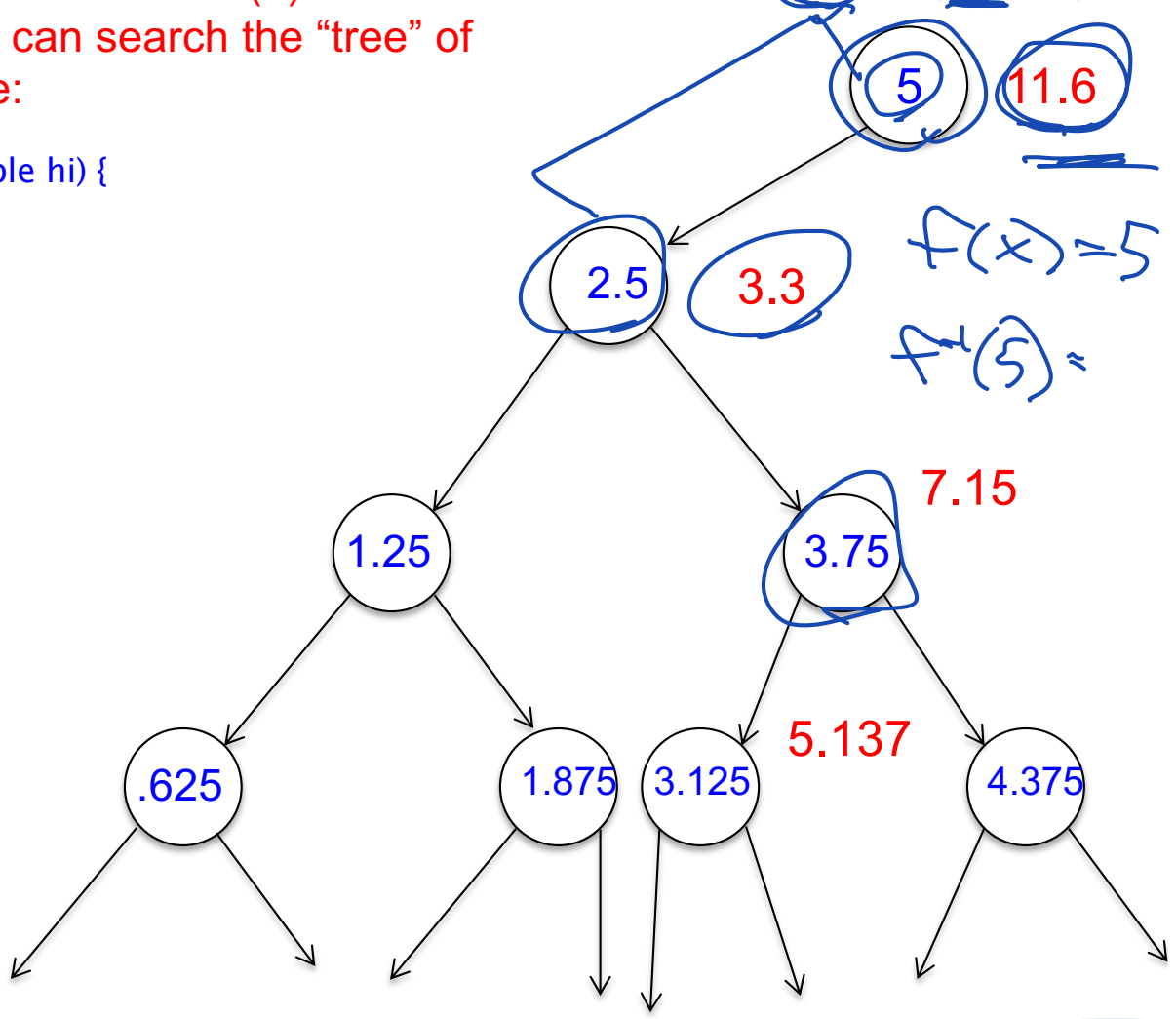
Suppose we want to find the inverse of a function f(n) which we know to be non-decreasing (monotonic); we can search the "tree" of possibilities without constructing a tree:

```
int inverse(double r, double n, double lo, double hi) {
      if( abs(n*Math.log2(n) – r) < 0.01)
           return n;
      double mid = (lo + hi)/2;
      if(f(mid) ≤ r)
          return inverse(r, mid, lo, mid);
      else
          return inverse(r, mid, mid, hi);
}

double f(double x) {
     return x *Math.log2(x);
}


// called like this:

        inverse(5, 5, 0, 5)
```

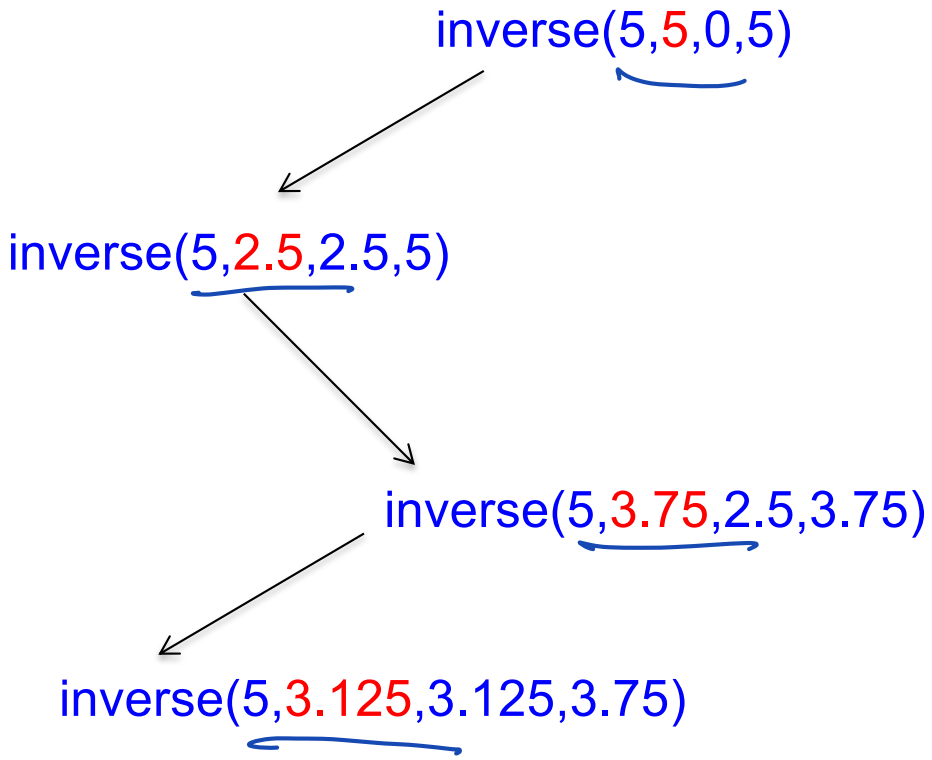


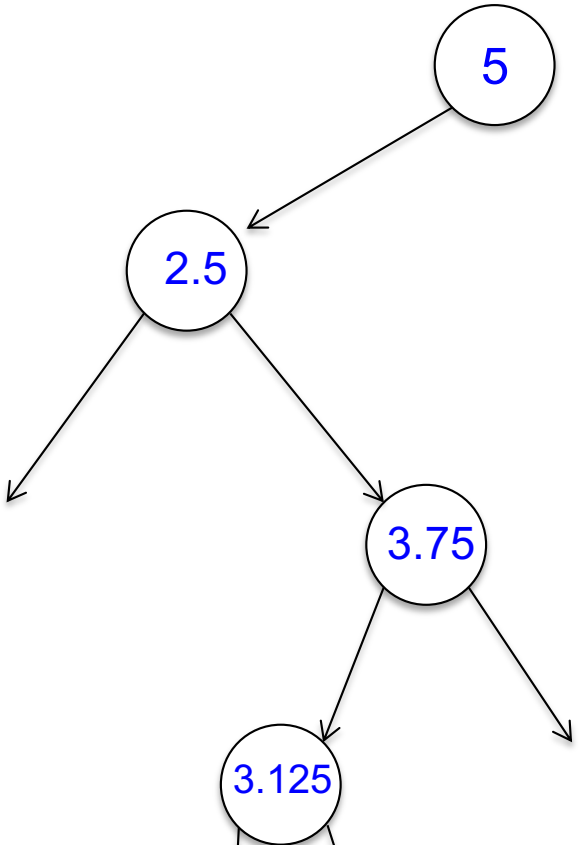
State Space Search

DO NODES In TREES and HAVE TO EXIST? NO!

The parameters to the function call are in effect “nodes” in the computation, and are not explicit nodes linked by explicit pointers; but they are essentially the same. They are created by need as the computation proceeds.

NOTE: The tree is infinite!





State Space Search: Tree Examples

The Eight Queens Problems: Place 8 queens on a chess board so that no one queen can attack another:

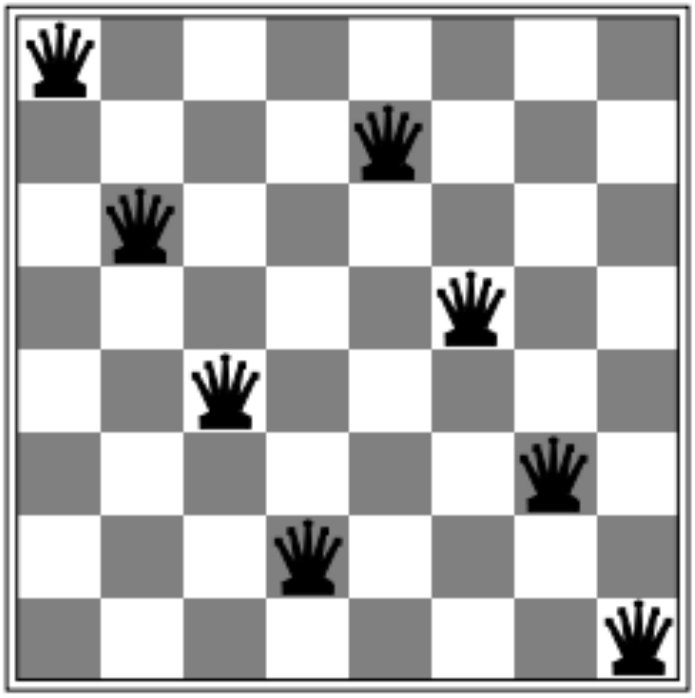
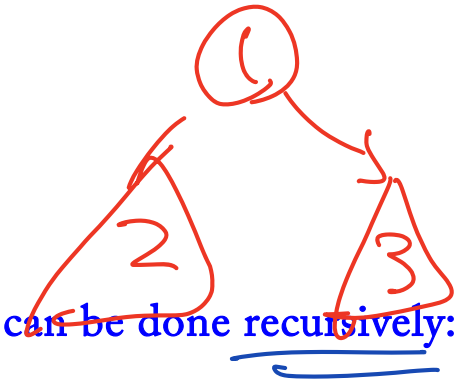


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

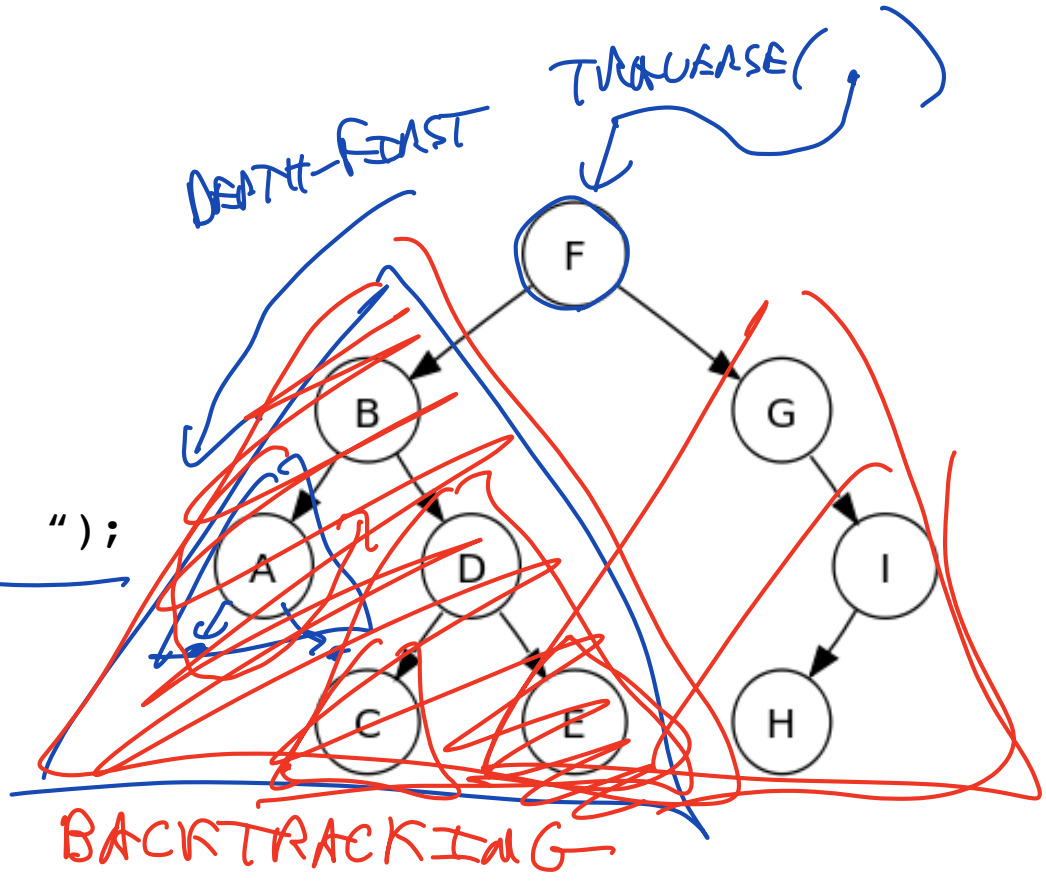
Tree Traversals



Searching a (finite) tree is known as a tree traversal. This can be done recursively:

```
void traverse(Node t) {
    if( t != null ) {          // Base case is implicit
       visit(t);               // V
       traverse(t.left);       // L
       traverse(t.right);      // R
    }
}

void visit(Node t) {
     System.out.print(t.key + "   ");
}
```



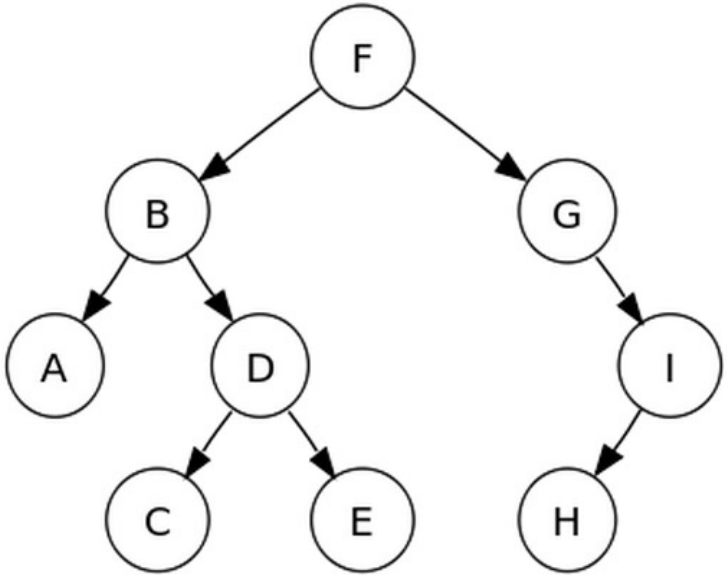
F B A D C E G I H

Non-Recursive Traversals

We can traverse a tree without recursion if we use an auxiliary data structure such as a stack or queue to keep track of the path traversed.

Let's try using a stack first:

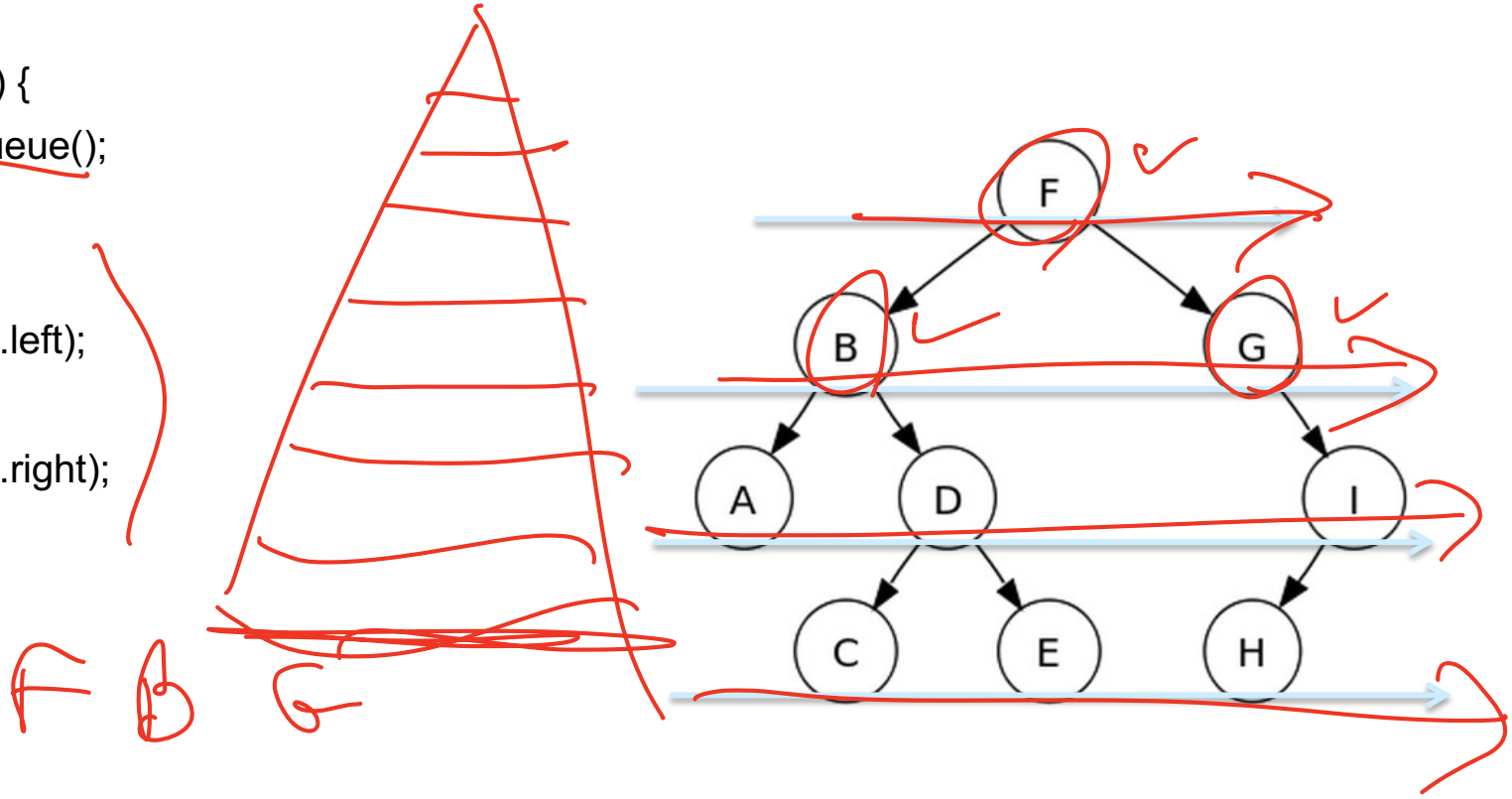
```
void DFS(Node t) {
    Stack<Node> S = new Stack<Node>();
    S.push(t);
    while( !S.isEmpty() ) {
        Node p = S.pop();
        visit( p );
        if( p.right!= null )
            S.push(p.right);
        if( p.left!= null )
            S.push(p.left);
    }
}
```



Non-Recursive Traversals

What happens if we use a **Queue** instead of a stack?

```
void BFS(Node t) {
    Queue<Node> Q = new Queue<Node>();
    Q.enqueue(t);
    while( !Q.isEmpty() ) {
        Node p = Q.dequeue();
        visit( p );
        if( p.left != null )
            Q.enqueue(p.left);
        if( p.right != null )
            Q.enqueue(p.right);
    }
}
```



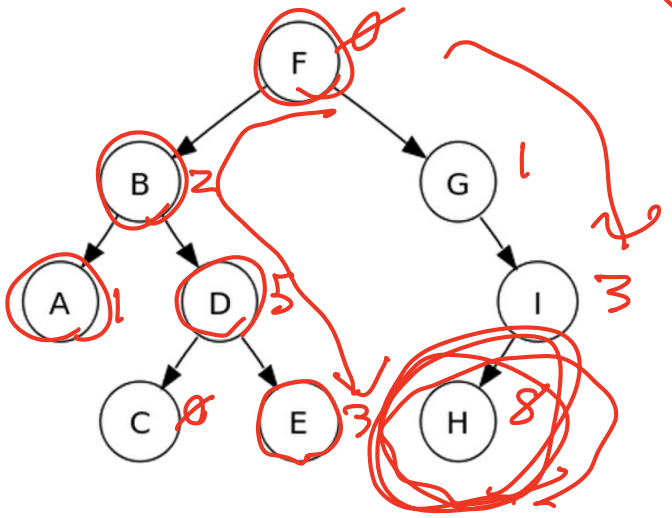
This is called a Breadth-First Search (BSF) or Level-Order Traversal.

Non-Recursive Traversals

In general, we can use any collection that supports adding and removing elements!

Suppose we have an arbitrary collection which stores the nodes in some order:

```
void Search(Node t) {
    Collection<Node> C = new Collection <Node>();
    C.add(t);
    while( !C.isEmpty() ) {
        Node p = C.removeNext();
        visit( p );
        if( p.left  != null )
            C.add(p.left);
        if( p.right != null )
            C.add(p.right);
    }
}
```



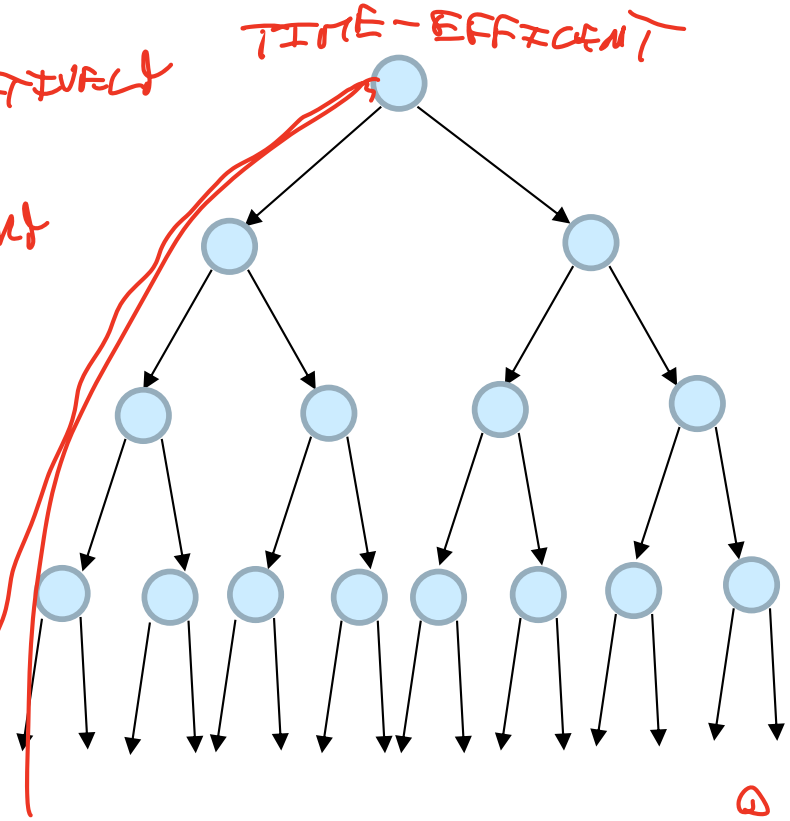
As long as every node eventually pops out of the Collection, this will visit all the nodes; for example, each node could have a priority measure that tells us how it important it is to search it asap. (This will be the basis for heuristic search in the next lecture.)

Searching a large or infinite tree

Traversals can't literally be done on infinite (or even extremely large) trees, but the same algorithms are involved in searching for a particular node (a goal node). Recursive algorithms are possible, but generally not useful, since they get "stuck" on an infinite path that may not have a goal node!

```
void DFS(Node t) {
     if( t != null ) {
        visit(t);
        DFS(t.left);
        DFS(t.right);
     }
}

void visit(Node t) {
      # Check if this is a goal node!
}
```

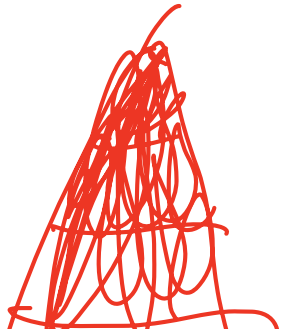


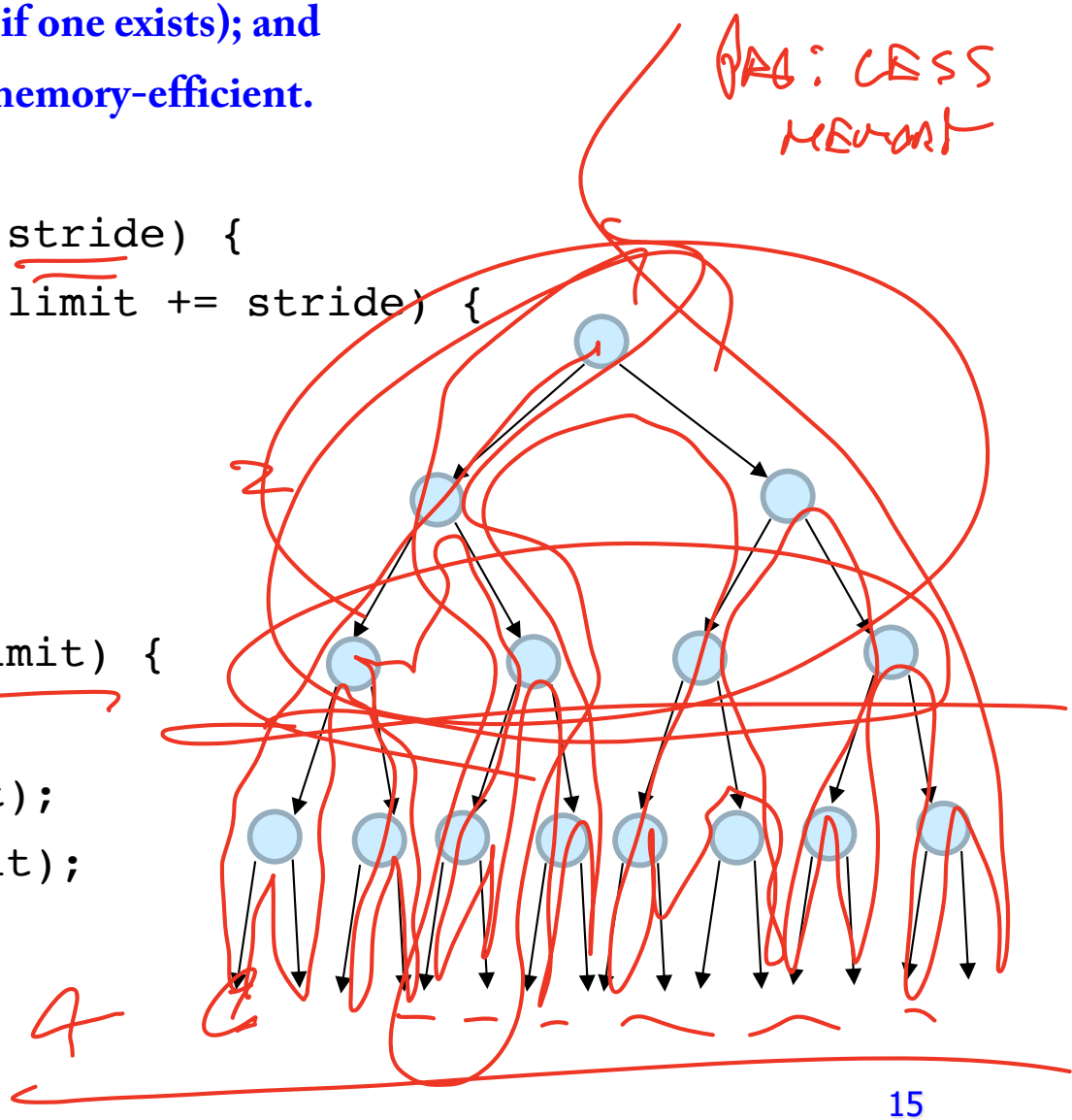
Searching a large or infinite tree

Iterative deepening is a cross between DFS and BFS which has the advantages of both:

- **It is complete (will always find a goal node if one exists); and**
- **Only has to store one path at a time, so is memory-efficient.**

```
void IterativeDeepening(Node t, stride) {
     for( limit = stride; true; limit += stride) {
        ID(t, 0, limit);
     }
}
void ID(Node t, depth, limit) {
    if( t != null and depth < limit) {
       visit(t);
       ID(t.left, depth+1, limit);
       ID(t.right, depth+1, limit);
    }
}
```





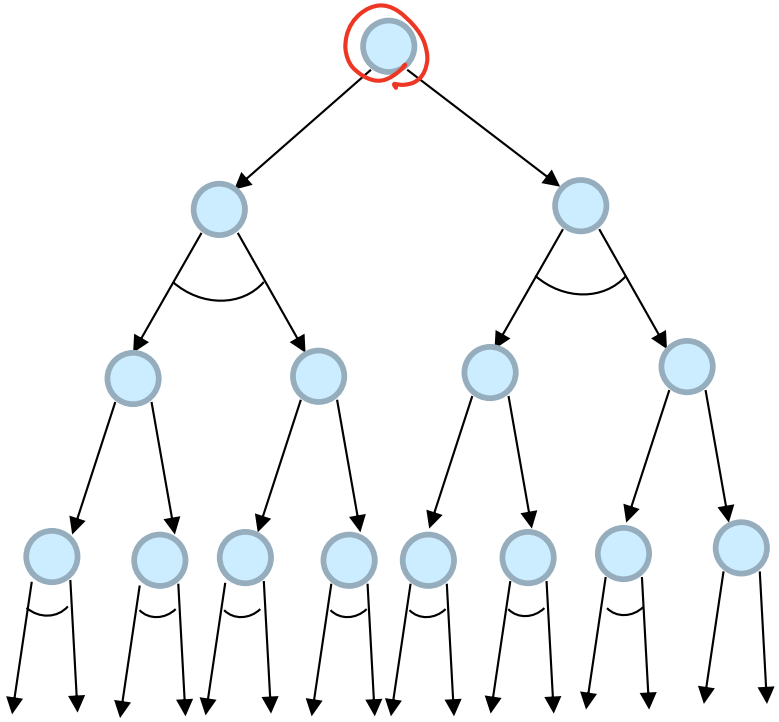
Searching an AND/OR Tree

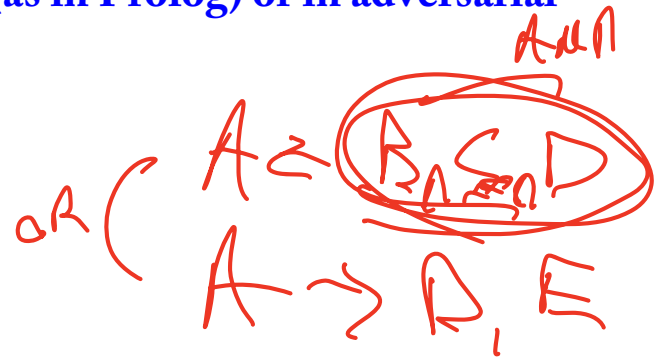
An AND/OR tree is a representation of a search space in which have two kinds of nodes:

- For AND nodes, you must follow the links of ALL children;
- For OR nodes, you only must follow SOME link.

This only makes sense for finite trees, where backtracking occurs: when backtracking, you must follow all links from AND nodes, but only follow OR nodes if no goal is found.

AND/OR trees arise, for example, when there are multiple goals (as in Prolog) or in adversarial search for games.





Graphs: Basic Definitions

Graphs are the most basic model of collections of information, generalizing trees to allow arbitrary links between nodes.

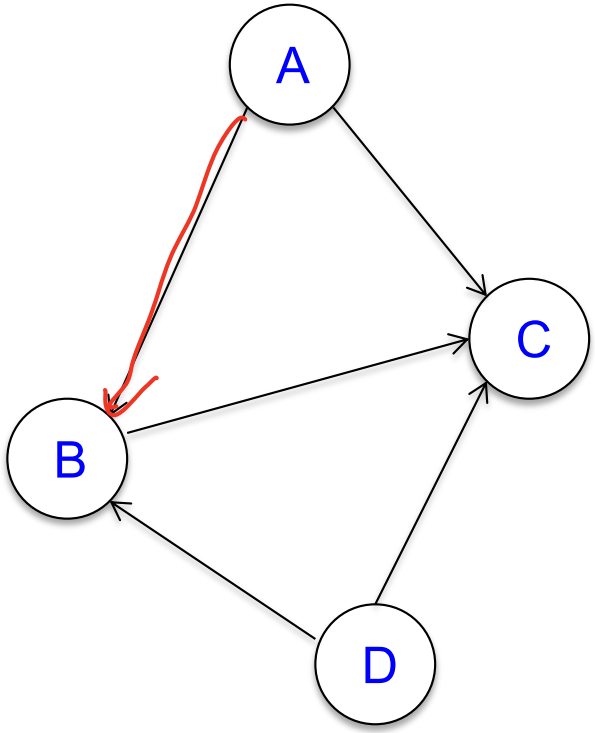
A Directed Graph (or Digraph) is:

- A set V of Vertices (or: Nodes) containing (possibly):
  - A Label (1, 2, … A, B, … etc.)
  - Data fields (boolean flags, counters, etc.)
- A set E of Edges (links) connecting vertices; edges may have labels or data (e.g., weight or cost) associated with them.

E is usually expressed as a relation on V, i.e., E consists of pairs of vertices:

(source, target)

E is a subset of V x V (Cartesian Product of V)



V = { A, B, C, D }

E = { (A,B), (A, C), (B,C), (D,B), (D,C) }

Note: Like a tree, but

- Any vertex can be connected to any other vertex;
- There is no root.

Directed Graphs: Basic Definitions

DAG
DIRECTED
ACYCLIC
GRAPH

Basic Notions of Digraphs:

Vertex (Vertex set V); Edge (Edge set E)

The out-degree of a vertex is the number of edges leaving it; the in-degree is the number arriving at it.

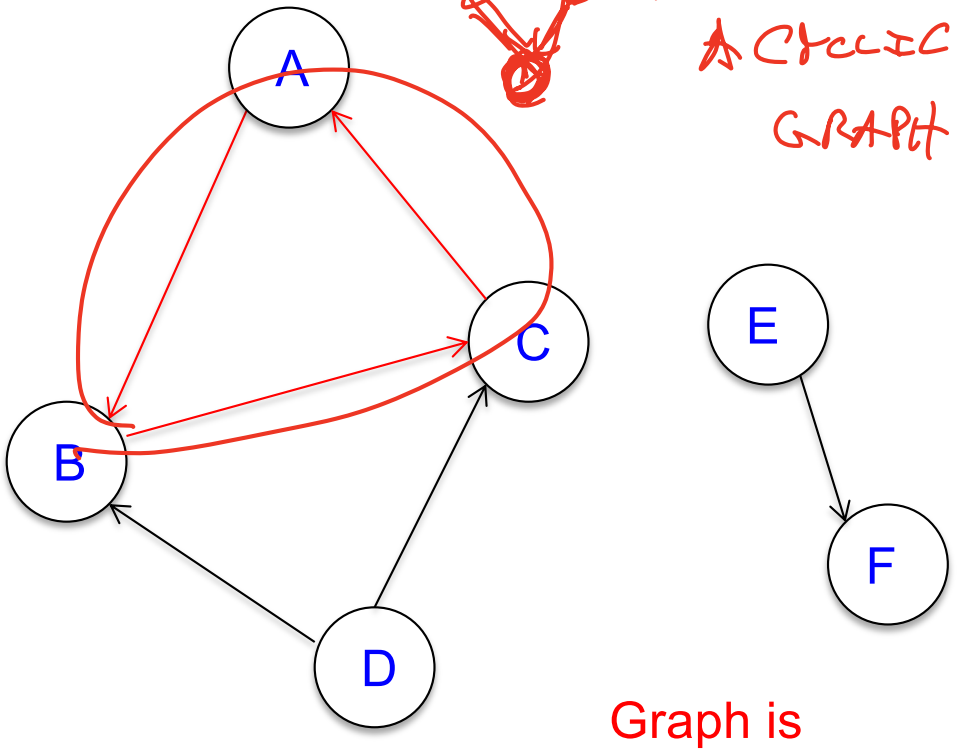
A path is a sequence of vertices

(v_1, v_2, ..., v_n) where 

A set of vertices is reachable from v if there is a path from v to every member of the set. The set of nodes reachable from v is always a tree.

A cycle (or loop) is a path that begins and ends on the same vertex.

B C A B



Graph is cyclic!

V = { A, B, C, D, E, F }

E = { (A,B), (A, C), (B,C), (D,B), (D,C), (E,F) }

Graphs: Basic Definitions

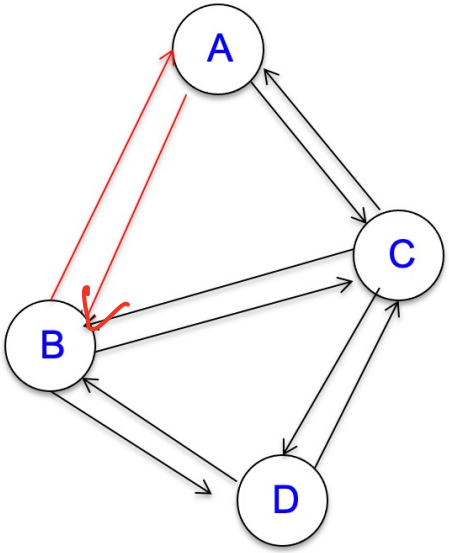
An Undirected Graph has the additional feature that all edges are two way, i.e., the relation E is symmetric:

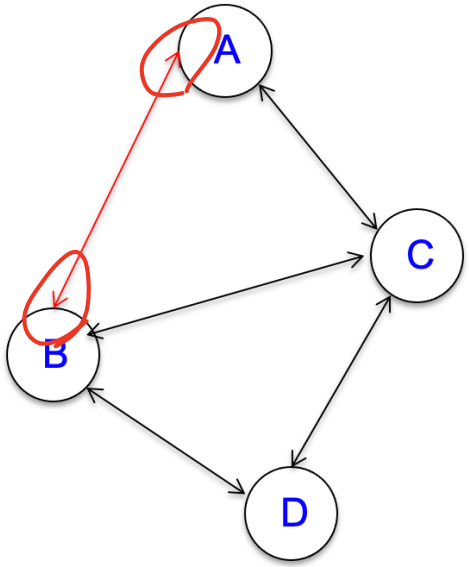
(A, B) is in E iff (B,A) is in E

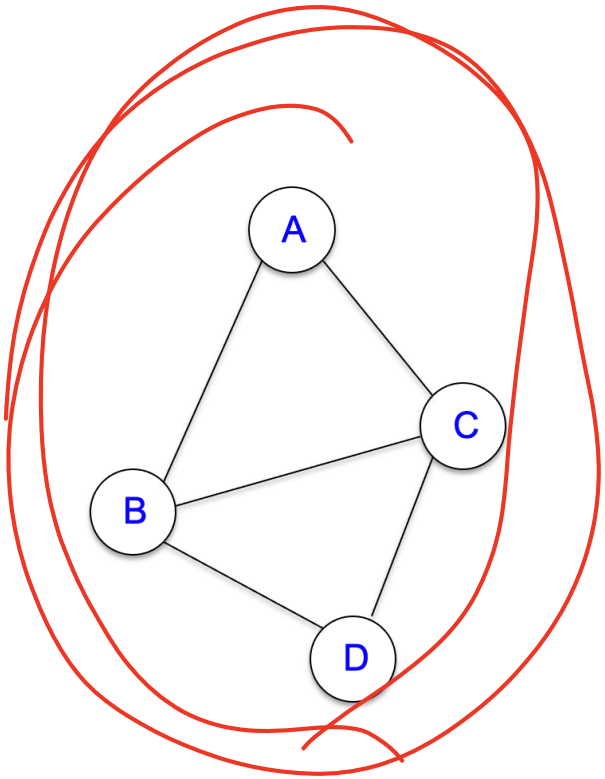
Edges do NOT have a direction (e.g., two-way streets).

"Graph" can mean either, but generally is assumed to be undirected.

There are various ways of drawing an undirected graph:







Digraphs: Search

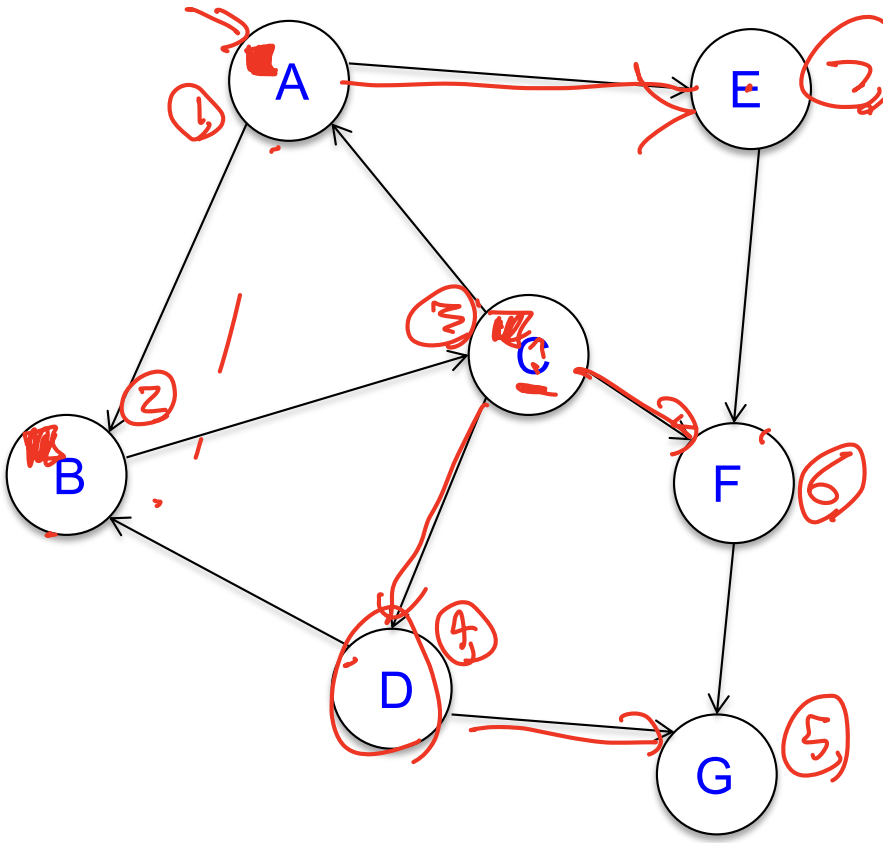
Depth-First Search in Digraph using recursion:

```
boolean visited;

searchGraph(V, E) {
   foreach( v in V )              // initialize all vertices
       v.visited = false;
   foreach( v in V )
       if(!v.visited)
          DFS(v);
}

DFS( Vertex v ) {
   if(!v.visited) {
       visit(v);
       v.visited = true;
       foreach( u in Adjacent(v) ) {
           DFS( u );
       }
   }
}
```

Digraph G



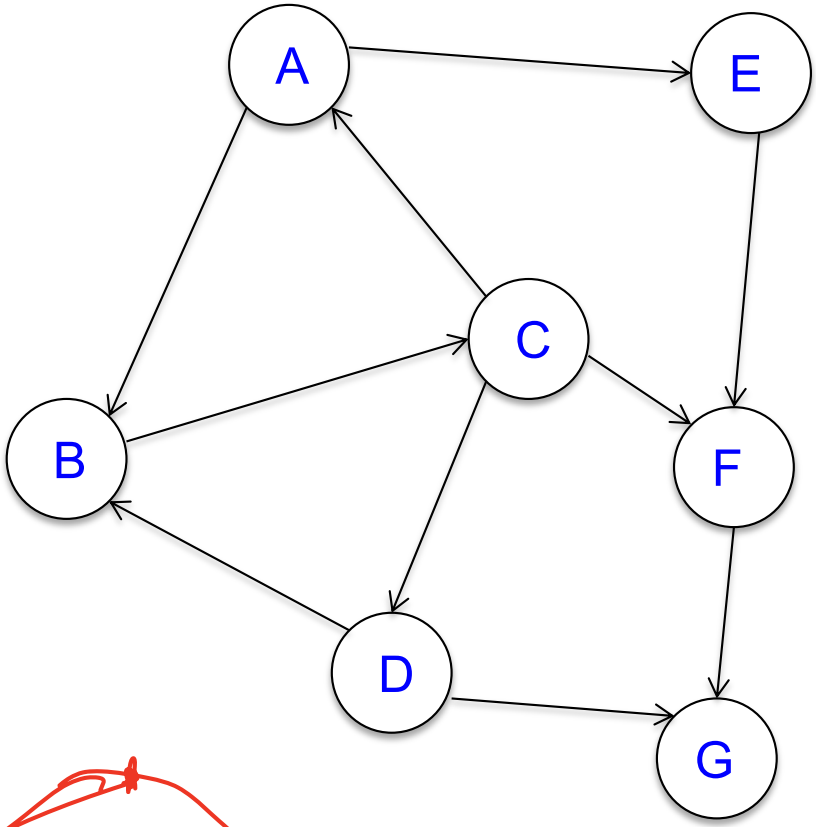
Digraphs: Search

Depth-First Search in Digraph using an explicit stack:

```
boolean visited;
searchDigraph(V, E) {
    foreach( v in V )              // initialize all vertices
        v.visited = false;
    foreach( v in V )
        if(!v.visited)
            DFS(v);
}

DFS( Vertex v )  {
    Stack S = new Stack();
    S.push(v);
    while( ! S.isempty() ) {
        Vertex u = S.pop();
        if(!u.visited) {
            visit(u);
            u.visited = true;
            foreach( w in Adjacent(u) )
                if(!w.visited)
                    S.push(w);
        }
    }
}
```

Digraph G

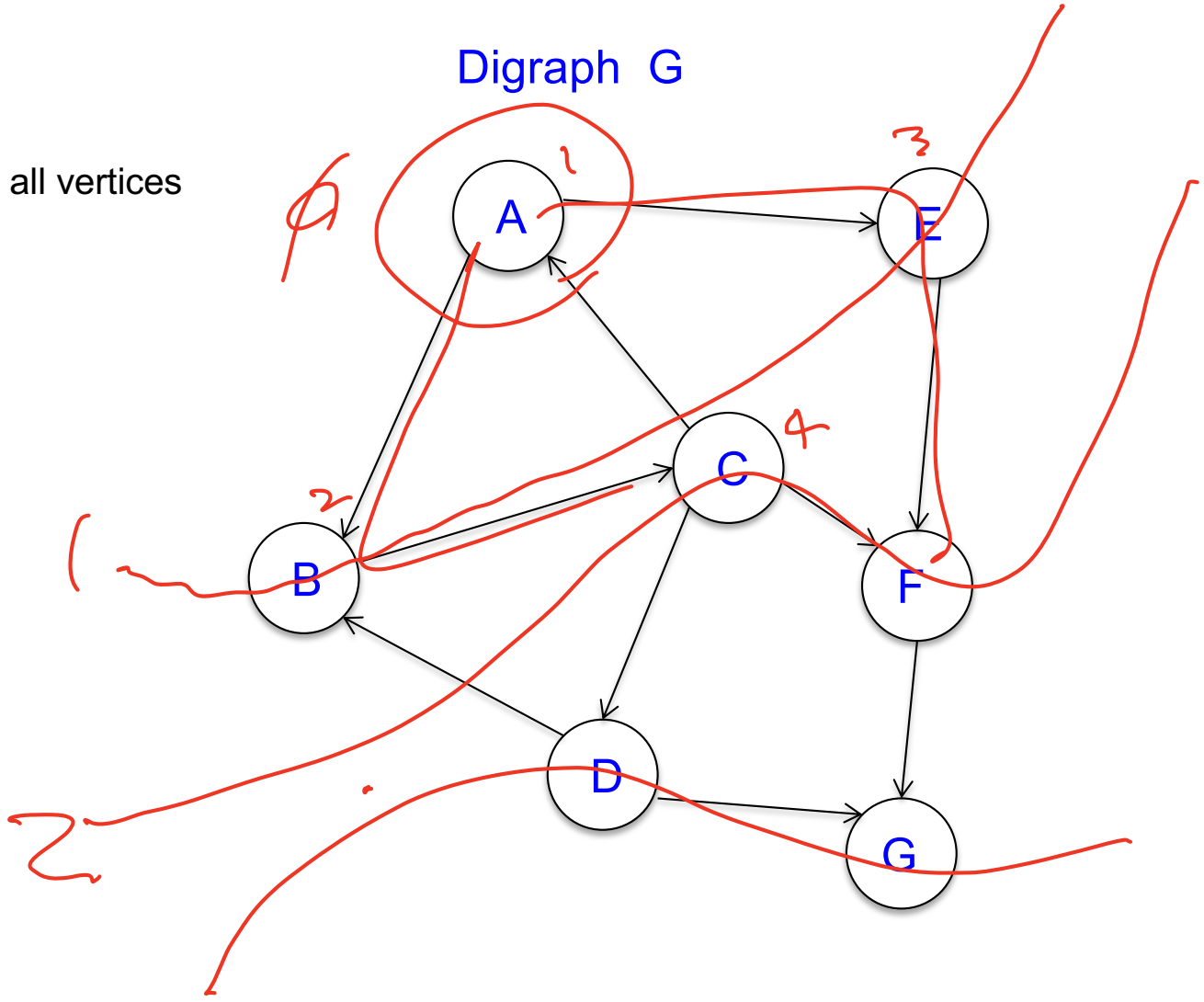


Digraphs: Search

Breadth-First Search in Digraph:

```
boolean visited;
searchDigraph(V, E) {
    foreach( v in V )              // initialize all vertices
        v.visited = false;
    foreach( v in V )
        if(!v.visited)
          BFS(v);
}

BFS( Vertex v )  {
    Queue S = new Queue();
    S.enqueue(v);
    while( ! S.isempty() ) {
        Vertex u = S.dequeue();
        if(!u.visited) {
            visit(u);
            u.visited = true;
            foreach( w in Adjacent(u) )
                if(!w.visited)
                    S.enqueue(w);
        }
    }
}
```



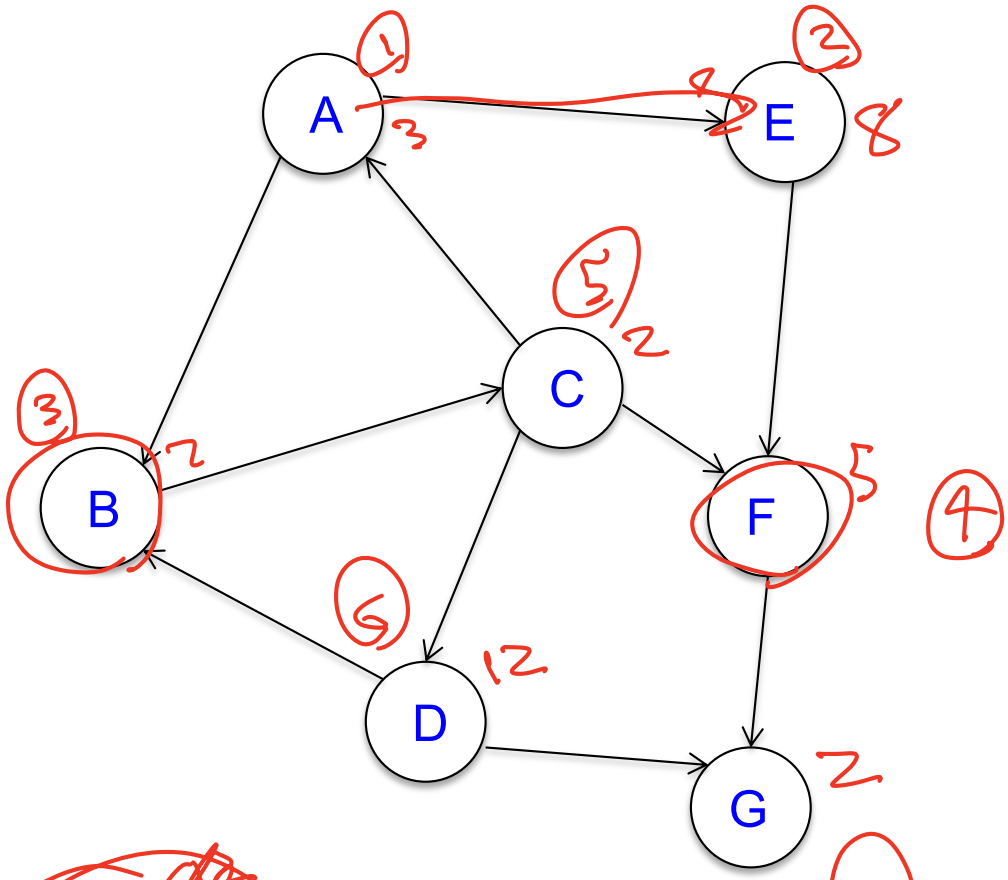
Digraphs: Search

Best-First Search in Digraph:

```
boolean visited;
searchDigraph(V, E) {
    foreach( v in V )              // initialize all vertices
        v.visited = false;
    foreach( v in V )
        if(!v.visited)
          BFS(v);
}

BFS( Vertex v ) {
    PriorityQueue S = new PriorityQueue();
    S.enqueue(v);
    while( ! S.isempty() ) {
        Vertex u = S.dequeue();
        if(!u.visited) {
            visit(u);
            u.visited = true;
            foreach( w in Adjacent(u) )
                if(!w.visited)
                    S.enqueue(w);
        }
    }
}
```

Digraph G



State Space Search: Graph Examples

State Space Search is a situation where

(imaginary) nodes == values of parameters in method calls == states of computation

You are searching among the various states
of the problem, and you “create the children”
of a node “by need” as you call a method
recursively.

Many puzzles and games can be solved this way…..

State Space Search: Graph Examples

Missionaries and Cannibals: Three missionaries and three cannibals are on the left bank of a river and have a boat in which they must cross to the right side, but the boat only holds two people and can not cross the river by itself. Furthermore, if the number of cannibals on either side of the bank is every greater than the number of missionaries, the cannibals will overwhelm and eat the missionaries. How can they all get to the other side without anyone spoiling his dinner?

State Space Search: Graph Examples

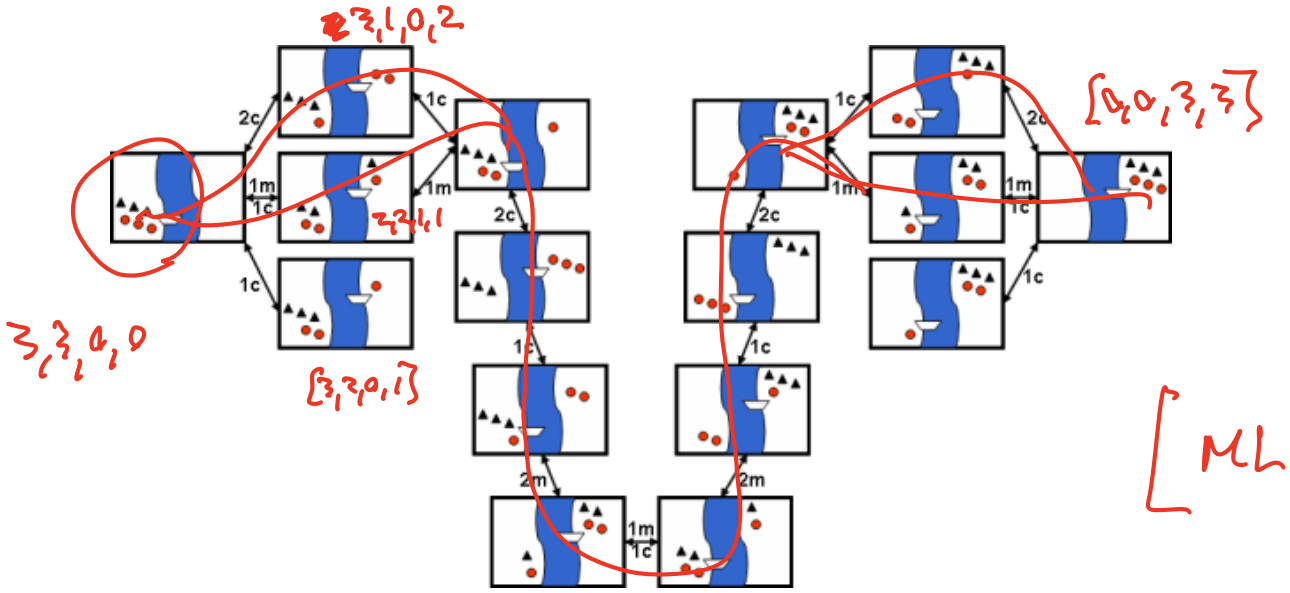
If we collapse the duplicates in the tree down into a graph (nodes and links which can point to anything, even make cycles among links), then:

Each node encodes the current STATE of the problem, and contains the number of missionaries and cannibals on each side:

```
int[] S = {3,3,0,0};    // 3M, 3C on left, 0M, 0C on right
```

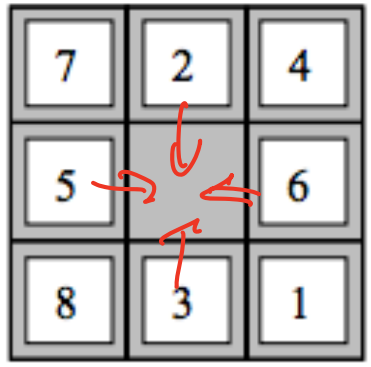
The edges in the graph (it is undirected) are the ways that the boat can go to the other side and change the state of the problem.

You can search for the solution by doing traversal without actually constructing nodes:



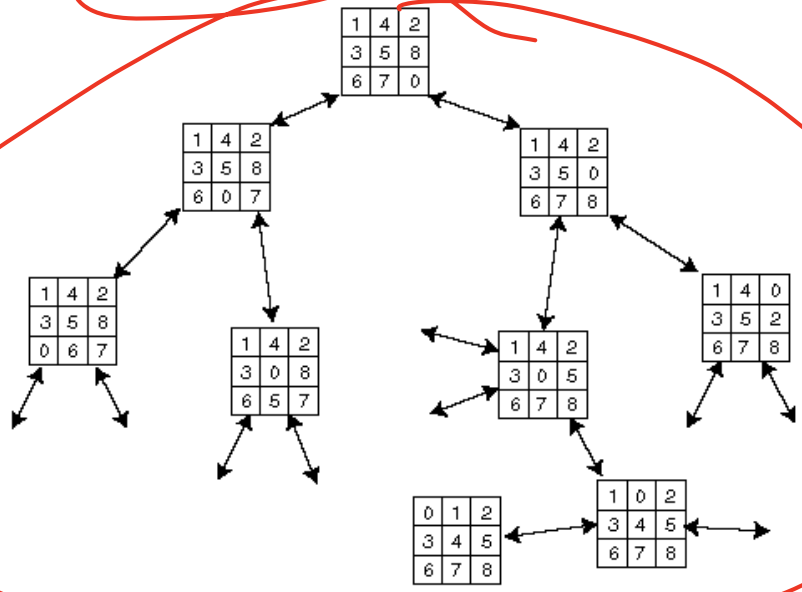
State Space Search: Graph Examples

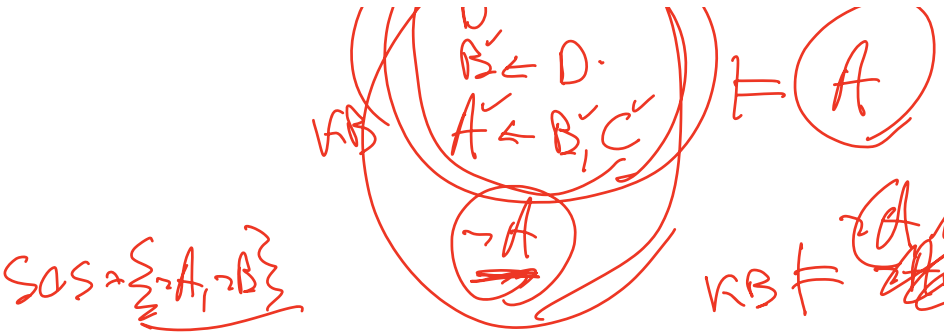
The Eight Puzzle: Slide numbered tiles until they are in order:



Start State

Goal State





$SOS = \{\neg A, \neg B\}$

$KB \models (A \vee B)$

$KB = [\{C\}, \{D\}, \{\neg B, D\}, \{\neg A, B, C\}]$

$SOS = [\{\neg A\}]$

$Q = [\{\neg A\}]$

$C = POP\ Q$

RESOLVER ALL $(KB + Q, C)$ =

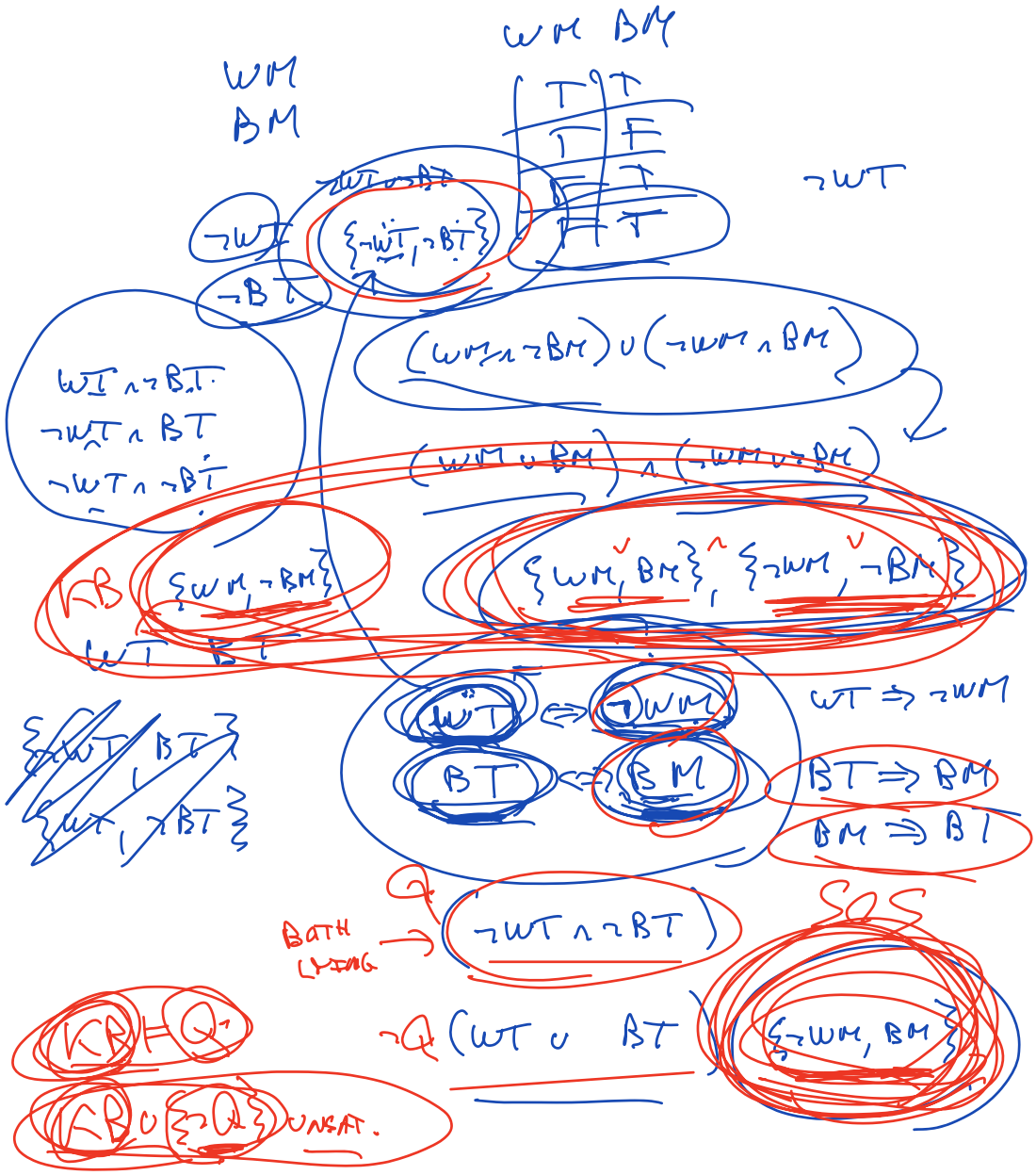
$KB \models A \vee B$ $\quad A \wedge B$

$KB \cup \{\{A\}, \{\neg B\}\}$

$A \to B \qquad \neg A, B$

$B \to C \qquad \neg B, C$

$C \to A \qquad \neg C, A$


WM BM
WM
BM
T T
T F
F T
F F
¬WT
¬WT ∨ ¬BT
{¬WT, ¬BT}
¬BT
(WM ∧ ¬BM) ∨ (¬WM ∧ BM)
WT ∧ ¬BT
¬WT ∧ BT
¬WT ∧ ¬BT
(WM ∨ BM) ∧ (¬WM ∨ ¬BM)
KB
{WM, ¬BM}
WT
BT
{WM, BM}, {¬WM, ¬BM}
WT ⇔ ¬WM
BT ⇔ BM
WT ⇒ ¬WM
BT ⇒ BM
BM ⇒ BT
{WT, BT}
{WT, ¬BT}
BOTH LYING
¬WT ∧ ¬BT
SOS
¬Q (WT ∨ BT)
{¬WM, BM}
KB ∧ Q
KB ∪ {¬Q}
UNSAT.