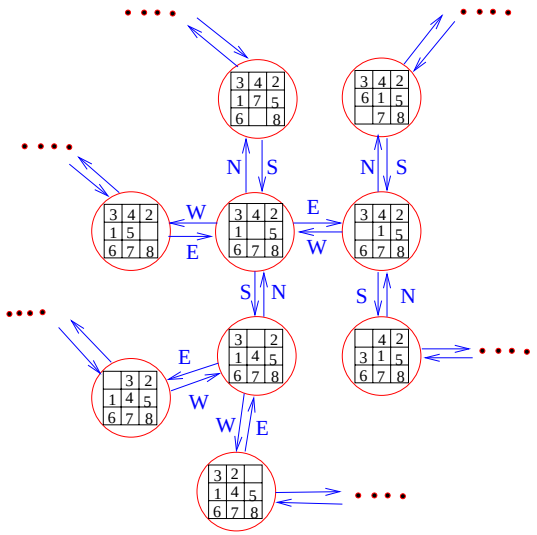
Data Structures

Goal: Understand data structures by solving the puzzle problem

- Elementary Structures
  1. Arrays
  2. Lists
  3. Trees
- Search Structures
  1. Binary search trees
  2. Hash tables
- Sequence Structures
  1. Stacks
  2. Queues
  3. Priority queues
- Graphs

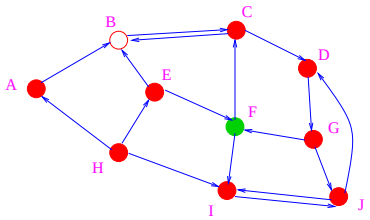
A Motivating Application

State Transition Graph of 8-Puzzle



Graph: a very general data structure

V: set of nodes E: set of edges (pairs of nodes)



V = {A,B,C,D,E,F,G,H,I,J}
E = {(A,B), (B,C),(C,B),.....}

Edge (A,B):
A is source of edge
B is destination of edge

Graphs can represent state transitions, road maps, mazes .....

In some graphs, edges may have additional information.

- puzzle graph: edges annotated with N/S/E/W

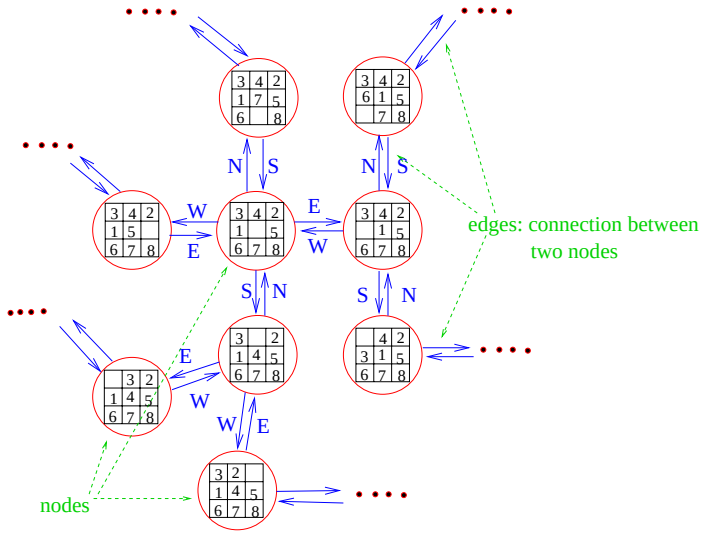
In some graphs, certain nodes may be special

- puzzle graph: initial and final (sorted) nodes are special

Graph in example is a DIRECTED graph

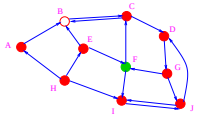
Undirected graph: no arrows on edges
analogy: 1-way street vs 2-way street

Graph: set of Nodes and Edges between nodes



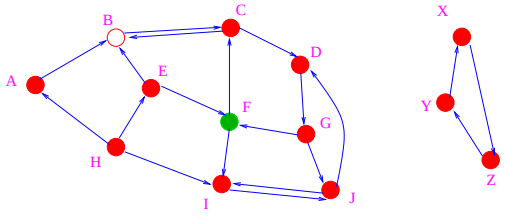
- Path: a sequence of edges in which destination node of an edge in sequence is source node of next edge in sequence
  Examples:(i) (A,B),(B,C),(C,D) (ii) (H,I),(I,J)
- Source of a path: source of first edge on path
  Destination of path: destination of last edge on path
- Reachability: node n is said to be reachable from node m if there is a path from m to n.
- There may be many paths from one node to another.
  Example: (E,F) and (E,B),(B,C),(C,D),(D,G),(G,F)
- Simple path: a path in which every node is the source and destination of at most two edges on the path
  Example: (not a simple path) (C,B),(B,C),(C,D)
- Cycle: a simple path whose source and destination nodes are the same. Example: (i) (C,B),(B,C) (ii) (D,G),(G,J),(J,D)

Some terminology



- Out-edges of a node n: set of edges whose source is node n (eg. out-edges of C are {(C,B),(C,D)})
- Out-degree of a node n: number of out-edges of node n
- In-edges of a node n: set of edges whose destination is node n (eg. in-edges of C are {(B,C),(F,C)})
- In-degree of a node n: number of in-edges of node n
- Degree of a node n in an undirected graph: number of edges attached to n in that graph
- Adjancency: node n is said to be adjacent to node m if (m,n) is an edge in the graph
  Intuitively, we can get from m to n by traversing one edge.

Path problems



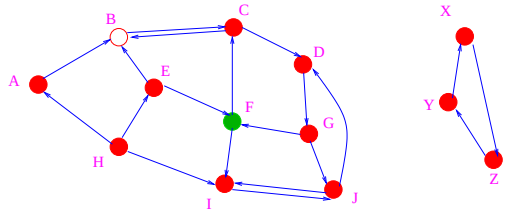
Length of a path: number of edges in path

Minimal path problems:
Find the shortest path from node A to node F.
Find the shortest path from every node to F.

For puzzle problem, this corresponds to path with fewest moves.

Sometimes, edges have distances.
Length of path = sum of lengths of distances on path.
This is more appropriate for path problems in graphs representing maps.

Path problems



Many interesting problems can be phrased as path problems in graphs.

(1) Is there a way to reach the sorted state, starting from any scrambled position of tiles? GRAPH SEARCH (similar to search in array)

Reachability problem:
Is there a path from a given node to the node representing sorted position?

For puzzle problem, answer is no for some nodes!

Sam Loyd: cannot reach sorted state from

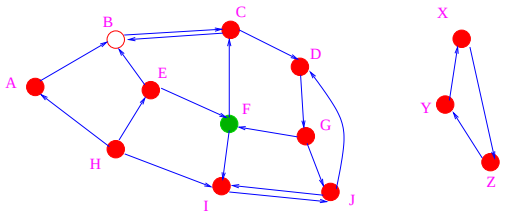
1	2	3
4	5	6
8	7	0

These kinds of problems are studied in *graph theory*.

If you get turned on by this stuff, become a CS major and take CS 410, 381/481 etc.

We will have time only to study some simple problems like reachability, with the goal of understanding modern data structures.

Path problems



Cycle: Path that starts and ends at same node.

Travelling salesman's problem:
Find the smallest length cycle that passes through all nodes.

Easy to come up with slow algorithms for this problem.
No one knows if there is an efficient algorithm for this problem.
(NP/NP-complete problems)

Goal: write a program to determine if sorted state is reachable from scrambled state for puzzle problem

Idea:

- Start in the scrambled state.
- Generate states adjacent to scrambled state.
- Generate states adjacent to those states.
  ....
- Stop if you either generate sorted state or you have generated all states reachable from scrambled state.

Think: *Graph* search is similar to *linear* search except that we are searching for something in a graph rather than in an array.

Requirements:

1. should not get stuck in cycles (correctness)
2. should be exhaustive: if we terminate without reaching sorted state, sorted state must be unreachable from scrambled state (correctness)
3. should not repeatedly examine states adjacent to a state(efficiency)

Key Idea: Keep two sets of nodes

1. toDo : set of nodes whose adjancies might need to be examined
2. done : set of nodes whose adjacencies have been examined

Pseudocode for Graph Search algorithm:

```
initialize toDo set with scrambled configuration;
while (toDo set is not empty)
  {Remove a node  v  from toDo set;
   if (v is in done set) continue;
   //we reach here if we have never explored v before
   for each node w adjacent to v do //there is edge (v -> w)
       {If  w  is the goal node, declare victory;
        Otherwise, add w to toDo set;
       }
   add v to done set;
  }
```


Modification: before adding w to toDo set, check if it is already there in the done set.

- Advantage: we do not put it into toDo set and get it out again if it has already been explored.
- Disadvantage: if node has not been explored, we will look it up in done set twice.

Code: see next slide.

Modification: handling self-loops more efficiently

If (v − > v) is an edge, we would add v to toDO set when exploring v.

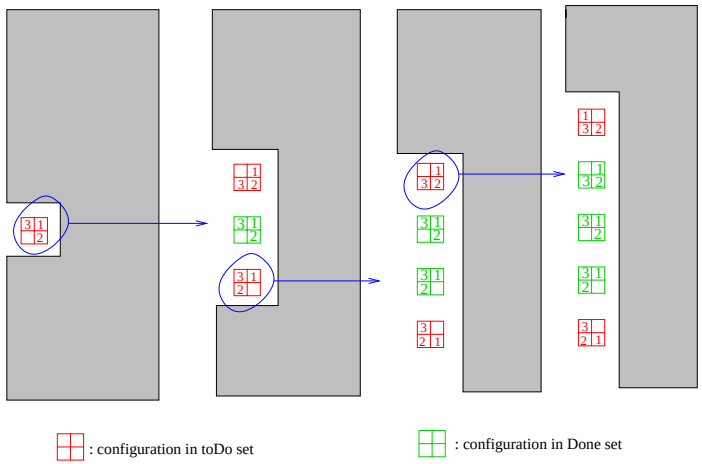
This is not necessary, so let us fix code.

```
initialize toDo set with scrambled configuration;
while (toDo set is not empty)
  {Remove a node  v  from toDo set;
   if (v is in done set) continue;
   //we reach here if we have never explored v before
   add v to done set;//this optimizes self-loops
   for each node w adjacent to v do //there is edge (v -> w)
       If (w is not in done set) {
            If (w  is the goal node) declare victory;
            add w to toDo set;
       }
  }
```



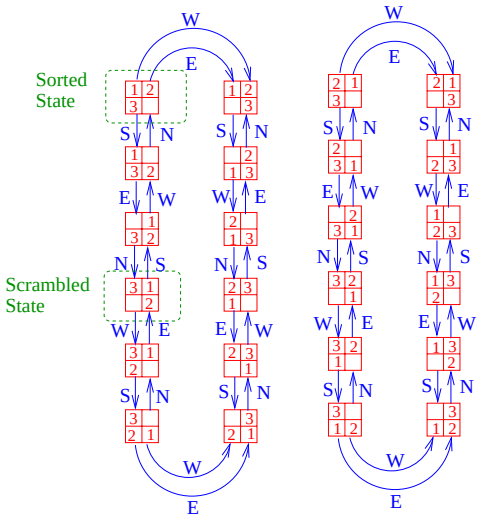
```
initialize toDo set with scrambled configuration;
while (toDo set is not empty)
  {Remove a node  v  from toDo set;
   if (v is in done set) continue;
   //we reach here if we have never explored v before
   for each node w adjacent to v do //there is edge (v -> w)
       If (w is not in done set) {
            If (w  is the goal node) declare victory;
            add w to toDo set;
       }
   add v to done set;
  }
```


Algorithm for Solving Puzzle : some possible initial steps



Note: we are not keeping track of moves that brought us to a given configuration

Let us try to execute a few steps of the algorithm for this problem.



Writing generic code:

- Order in which we explore nodes (order in which they are removed from toDo set) is very important, and can make a big difference in how quickly we find solution.

  How can we write code so that it works for any sequence structure? Answer: use subtyping

- Most time-consuming part: searching if node is in Done set.

  How do we write code so that it works for any search structure? Answer: use subtyping

- Graph search algorithm works for any graph, not just puzzle state transition graph (all we need to some way to determine what nodes are adjacent to a given node).

  How can we write code so that it works for any graph? Use Iterators to return all adjacent vertices of a node.

To make this a program, we need to answer the following questions:

(1) SEQUENCE STRUCTURE: In what order should we get nodes from the *toDo* set? What data structure can we design to give us the nodes in that order? How can we accommodate the fact that the *toDo* set grows and shrinks?

Answer: stacks, queues, priority queues

(2) SEARCH STRUCTURE: How do we organize the *Done* set so that we can search it efficiently?

Answer: binary search trees, hash tables

Two key interfaces:

SeqStructure: all sequence structures implement this interface

SearchStructure: all search structures implement this interface

For search structure, fast search is important!

For sequence structure, fast lookup is not important!

Code for Simulating Puzzle

```
class searcher{

   public static void main(String[] args) {
      IPuzzle p0 = new ArrayPuzzle();
      p0.move('S');
      p0.move('E');
      SearchStructure s = new BST();
      SeqStructure q = new QAsList();
      graphSearch(p0,q,s);
   }
```

```
        if (done.search(p)) continue;
        //if not, let's explore this node
        done.insert(p);
        //determine adjacent nodes and process them
        String Moves = "NSEW";
        for (int i = 0; i < Moves.length(); i++) {
            IPuzzle nP = p.duplicate();
            char dir = Moves.charAt(i);
            //try to make the move
            boolean OK = nP.move(dir);
            if (OK) {
               //move succeeded, so we have a legitimate node
               if (! done.search(nP)) {
                  if (nP.isSorted()) {
                      System.out.println("Hurrah");
                      return;
                     }
                  toDo.put(nP);
```



```
//specialized to puzzle problem
//will work for any search structure and sequence structure
public static void graphSearch(IPuzzle p0,
                               SeqStructure toDo,
                               SearchStructure done){
    if (p0.isSorted()) {
        System.out.println("Already sorted");
        return;
    }
    //initialize work-list
    toDo.put(p0);

    //while there are toDo nodes
    while (! toDo.isEmpty()) {
          //get a toDo node
          IPuzzle p = (IPuzzle)toDo.get();

          //have we explored this node already?
```



```
                  }
                }
             }
          }
          //no more toDo nodes
         System.out.println("Could not reach sorted state");
    }
}
```


The code we have written will work for any search structure and sequence structure that implement the interfaces defined before. Subtyping is wonderful!

The puzzle state transition graph is hardwired into the code. We cannot use it to perform a graph search in a general graph. We will fix this later.

toDo data structure grows and shrinks. How do we implement a good sequence structure?

How do we implement a good search structure?