COMP 424 - Artificial Intelligence
Lecture 2: Uninformed Search

Instructor: Jackie CK Cheung (jcheung@cs.mcgill.ca)

Class website:

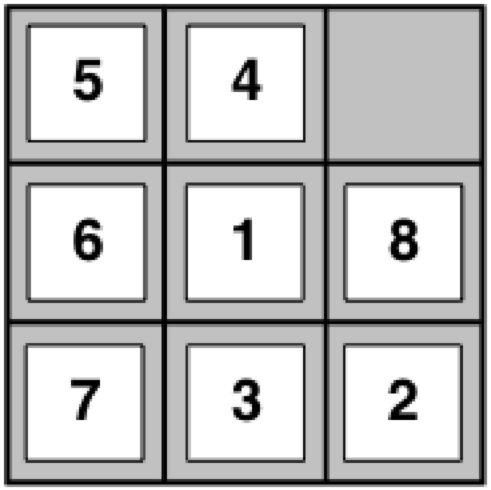
http://cs.mcgill.ca/~jcheung/teaching/winter-2017/comp424/index.html

Based on slides by Joelle Pineau

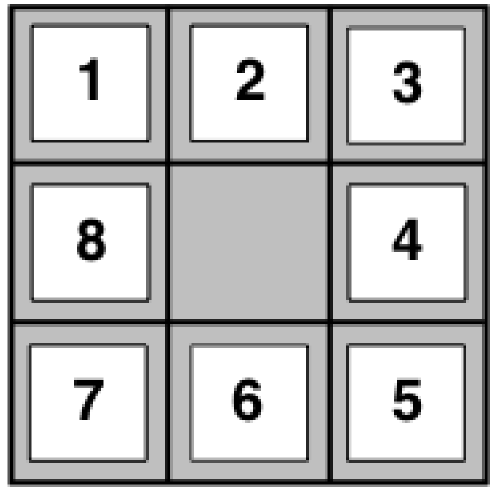
Goals for today's class

- Identify defining elements of generic search problems
- Uninformed search algorithms
  1. Breadth-first search
  2. Uniform-cost search
  3. Depth-first search
  4. Depth-limited search
  5. Iterative deepening
- Define criteria for evaluating search algorithms

Eight-Puzzle



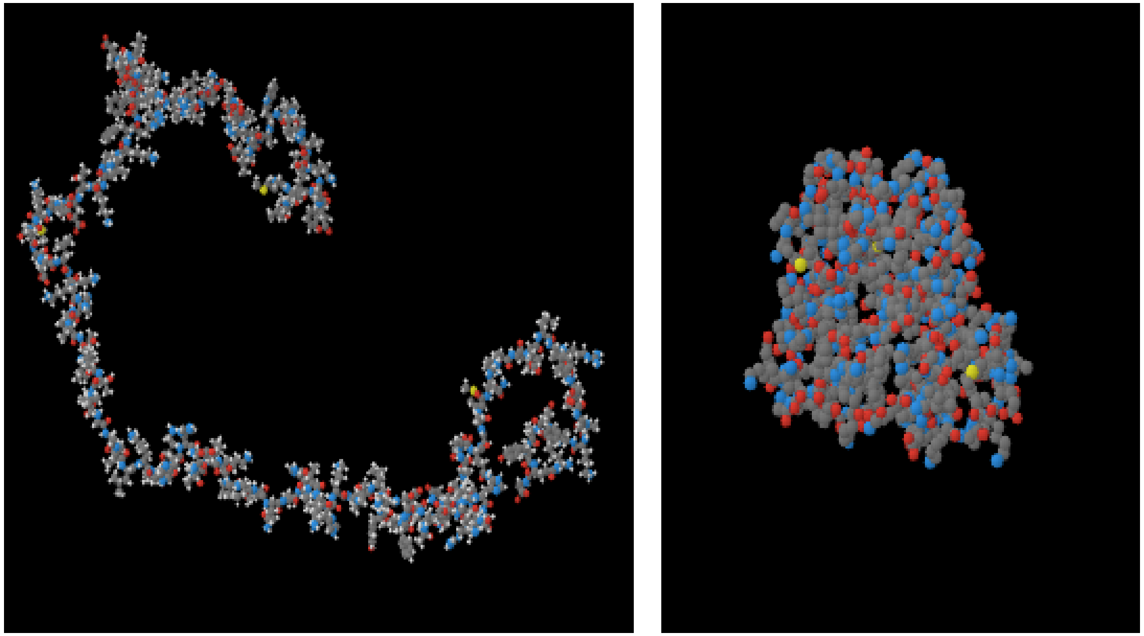
Start State



Goal State

How would you build an AI agent to solve this problem?

Protein design



- The 3D shape of protein depends on its amino acid sequence, determines its function.
- **Search** for a sequence of amino acids that give some desired shape

Search in AI

- One of the first topic studied in AI:

  Newell & Simon (1972). *Human Problem Solving*.

- Central component to many AI systems:
  - Theorem proving
  - Game playing
  - Automated scheduling
  - Robot navigation

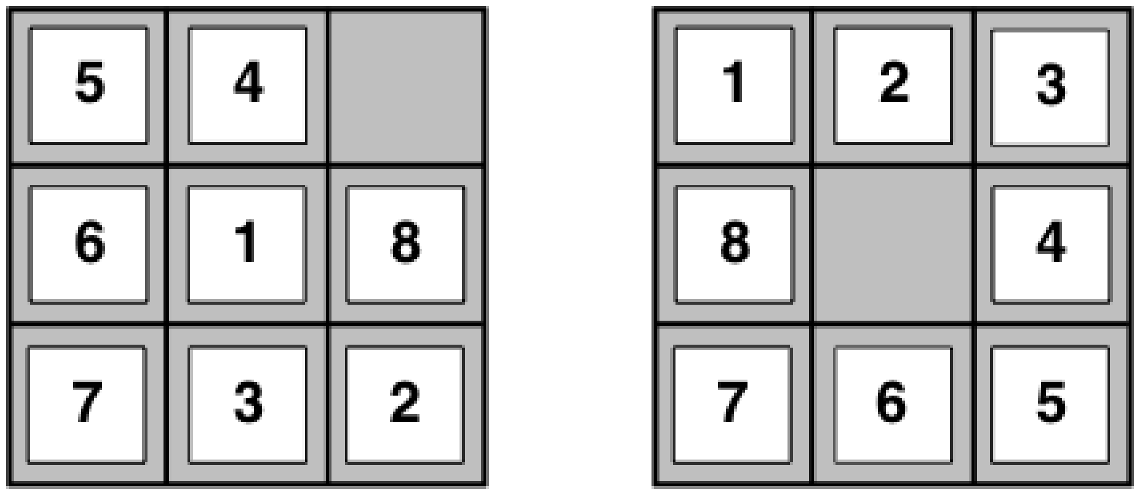
Defining a search problem

- **State space** S: all possible configurations of the domain.
- **Initial state** $s_0 \in S$: the start state
- **Goal states** $G \subset S$: the set of end states
- **Operators** A: the actions available
  - Often defined in terms of mapping from state to successor state.

Defining a search problem (cont'd)

- **Path**: a sequence of states and operators.
- **Path cost**, c: a number associated with any path.
- **Solution** of search problem: a path from s_0 to $s_g \in G$
- **Optimal solution**: a path with minimum cost.

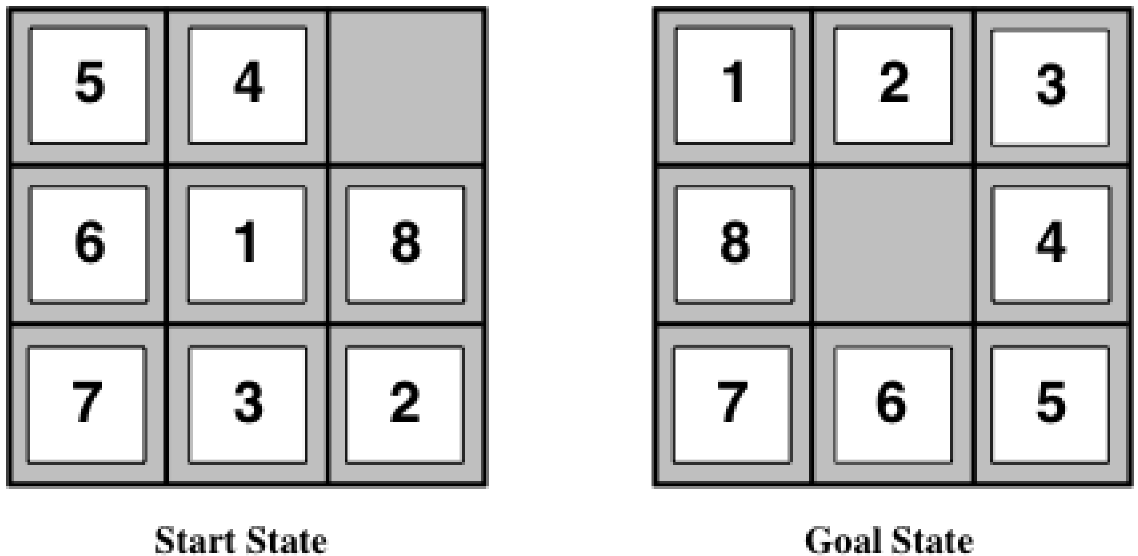
Example: Eight-Puzzle



Start State Goal State

- States?
- Goals?
- Operators?
- Path cost?

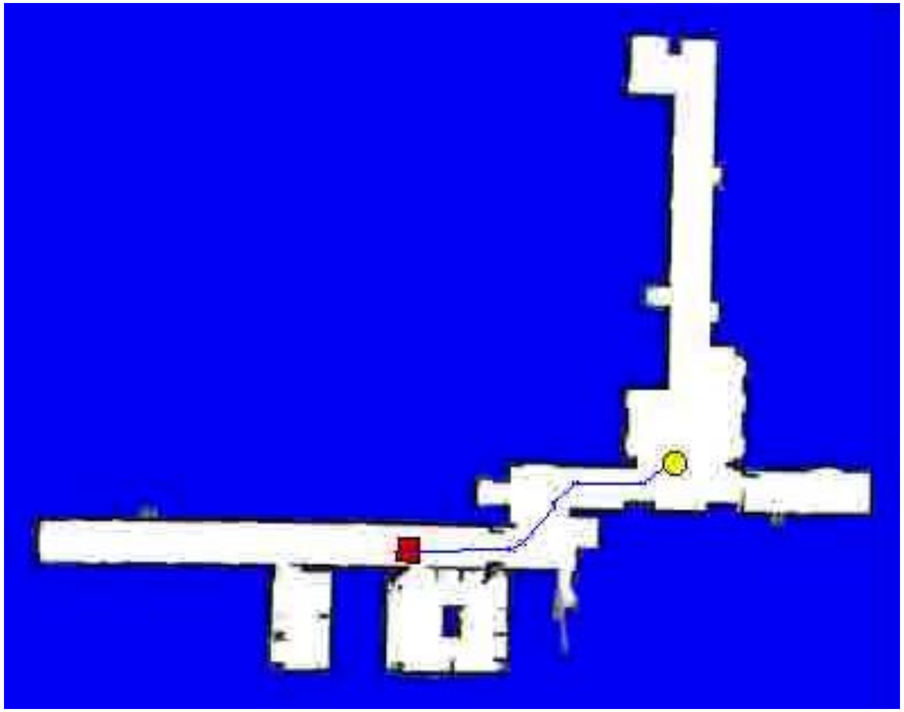
Example: Eight-Puzzle



- States? Configurations of the puzzle.
- Goals? Target configuration.
- Operators? Swap the blank with an adjacent tile.
- Path cost? Number of moves.

Example: Robot path planning

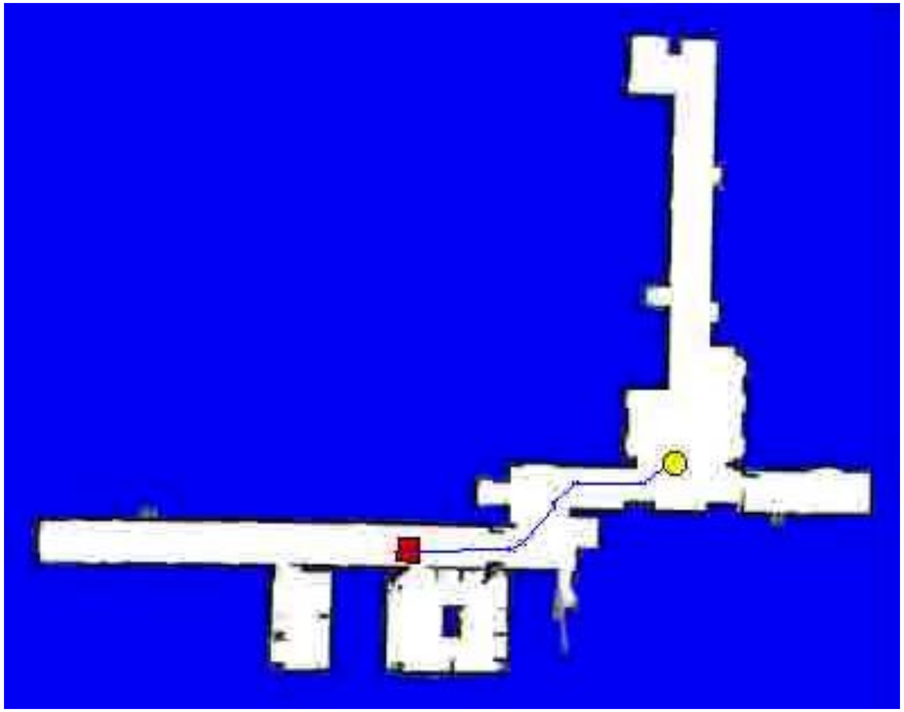
Get from red square to yellow dot.



- States?
- Goals?
- Operators?
- Path cost?

Example: Robot path planning

Get from red square to yellow dot.



- States? Position, velocity, map, obstacles.
- Goals? Get to target position without crashing.
- Operators? Small steps in several directions.
- Path cost? Length of path, energy consumption, time to goal, ...

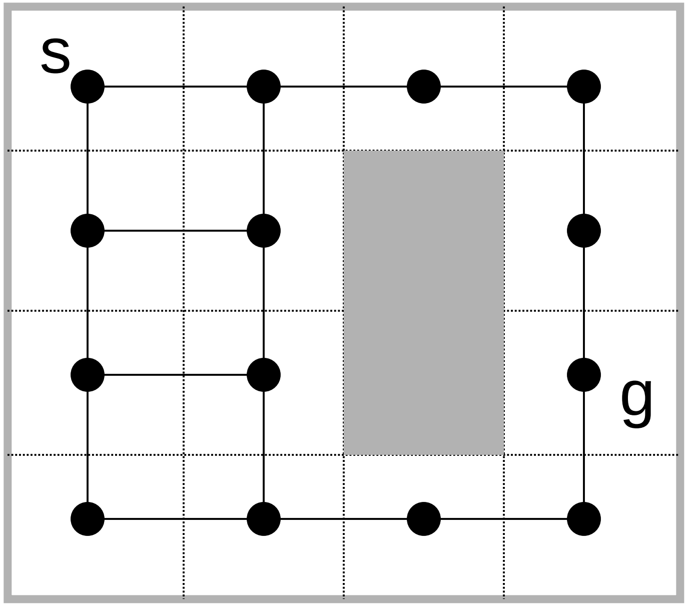
Basic assumptions (for next few lectures)

- Static (c.f. dynamic) environment
- Observable (c.f. unobservable) environment
- Discrete (c.f. continuous) states
- Deterministic (c.f. stochastic) environment

The general search problem does not make these assumptions, but most of the search algorithms discussed today require them.

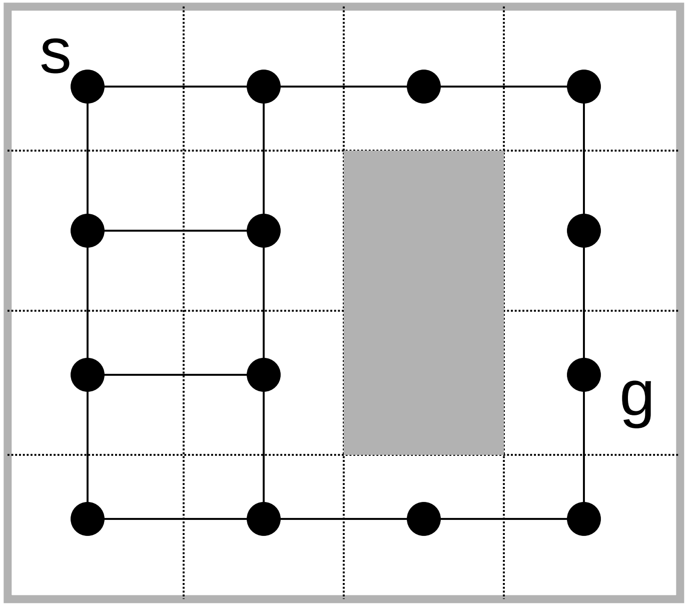
Representing search: Graphs and Trees

- Visualize the state space search in terms of a graph.
- Graph defined by a set of vertices and a set of edges connecting the vertices.
  - Vertices correspond to states.
  - Edges correspond to operators.

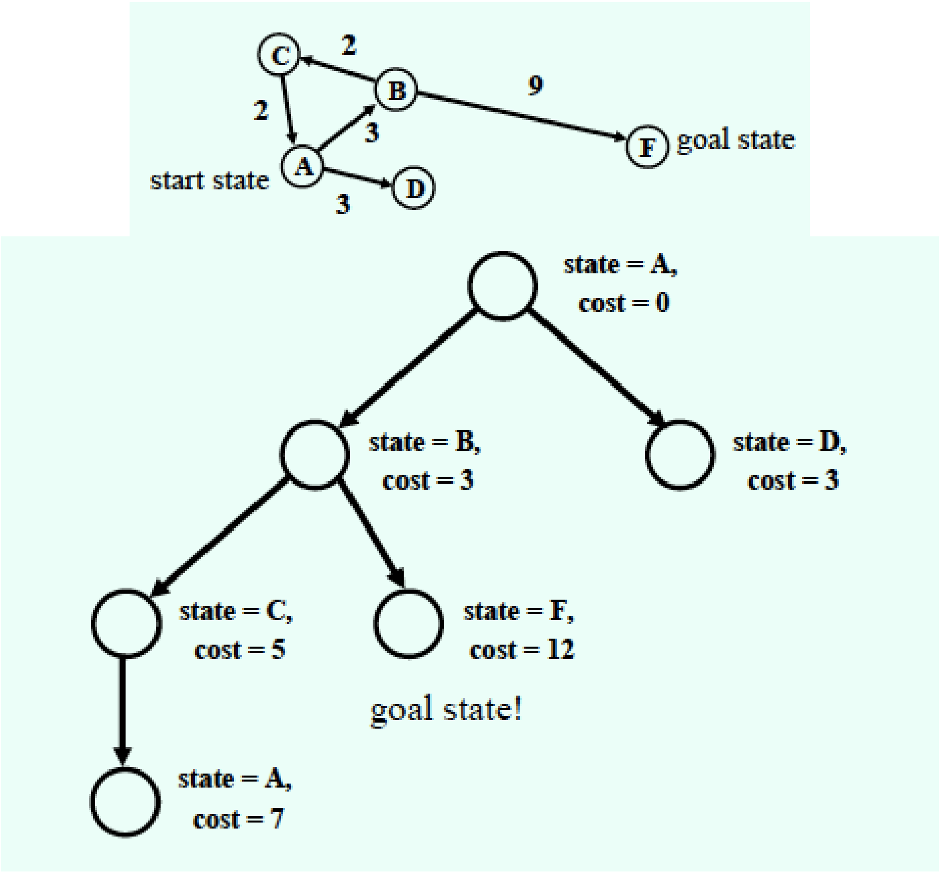


Representing search: Graphs and Trees

- Visualize the state space search in terms of a graph.
- Graph defined by a set of vertices and a set of edges connecting the vertices.
  - Vertices correspond to states.
  - Edges correspond to operators.
- We search for a solution by building a **search tree** and traversing it to find a goal state.



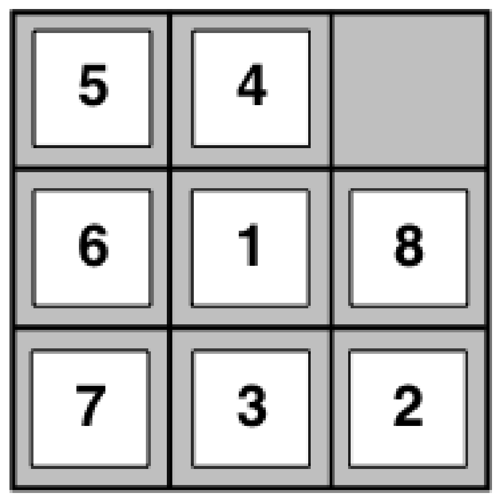
Example



Search tree nodes are NOT the same as the graph nodes!

In-Class Exercise

- Draw the search tree for the eight-puzzle example from before down to a depth of 2:



Start State

Data structures for search tree

- Defining a search tree node:
  - Each node contains a state id (from the states in the graph).
  - Node also contain additional information:
    - The parent state and the operator used to generate it.
    - Cost of the path so far.
    - Depth of the node.

- Expanding a search tree node:
  - Applying all legal operators to the state.
  - Generating nodes for all the corresponding successor states.

Generic search algorithm

- **Initialize** the search tree using the **initial state** of the problem

- **Repeat**
  1. If no candidate nodes can be expanded, return failure.
  2. Choose a node for expansion, according to some search strategy.
  3. If the node contains a **goal state**, then
     - return the corresponding **path**.
  4. Otherwise expand the node, by:
     - applying each applicable **operator**
     - generating the **successor state**, and
     - adding the resulting nodes to the tree.

Coding a Generic Search Problem in Java

```
public abstract class Operator {}

public abstract class State {
    abstract void print(); }

public abstract class Problem{
    State startState;
    abstract boolean isGoal (State crtState);
    abstract boolean isLegal (State s, Operator op);
    abstract Vector getLegalOps (State s);
    abstract State nextState (State crtState, Operator op);
    abstract float cost(State s, Operator op);

    public State getStartState() { return startState; }
}
```

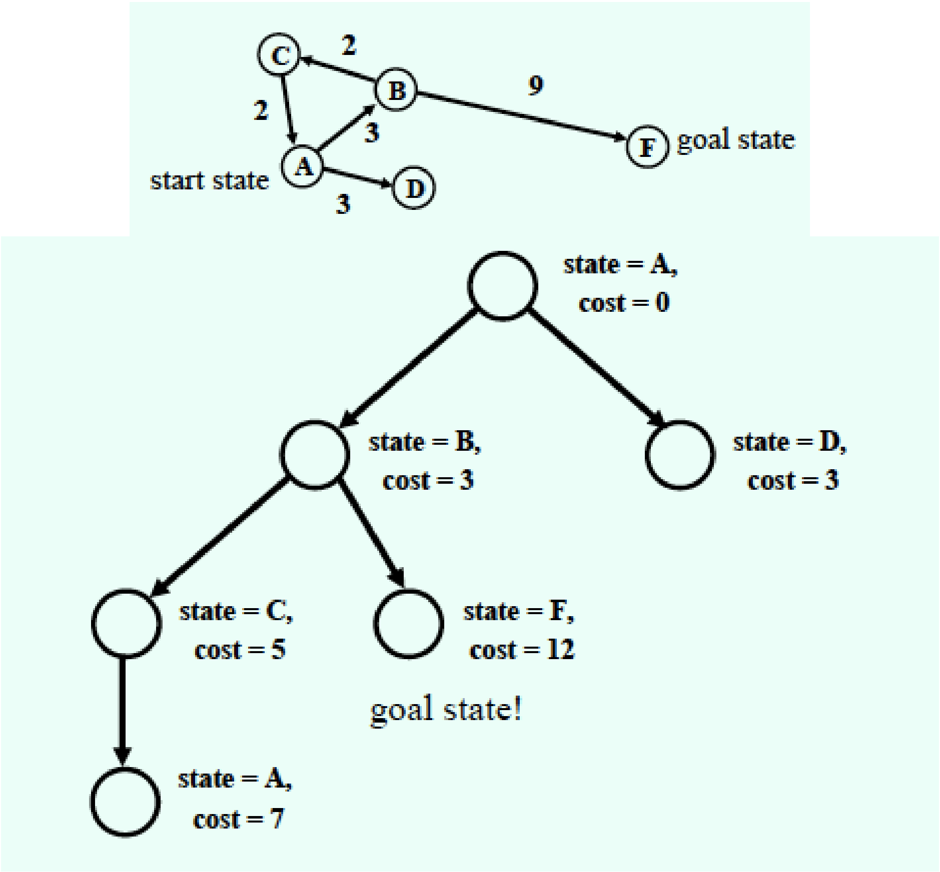
Coding an Actual Search Problem

```
public class EightPuzzleState extends State {
  int tilePosition[9];
  public void print() {//
  }
}

public class EightPuzzleProblem extends Problem{
  boolean isLegal (EightPuzzleState s,
                   EightPuzzleOperator op){
    // check if blank can be moved in the desired direction
  }}
```

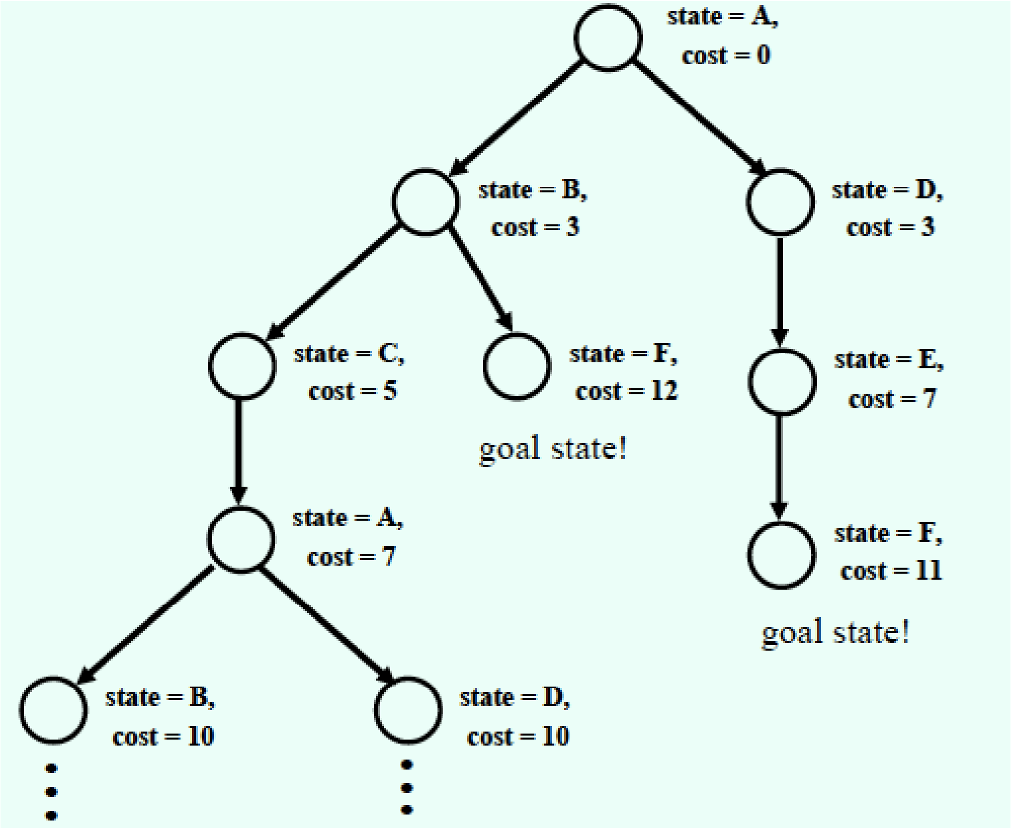
Specialize the abstract classes, and add the code that does the work

Example



Now expand a little further…

Example



Problem: Search trees can get very big!

Implementation details

- Need to keep track of the nodes to be expanded: the **frontier**.
- Implement this using a **queue**:
  1. Initialize queue by inserting a node for the initial state.
  2. Repeat
     a) If the queue is empty, return failure
     b) Dequeue a node.
     c) If the node contains a goal state, return path.
     d) Otherwise expand the node by applying all legal operators to the state.
     e) Insert the resulting nodes in the queue.

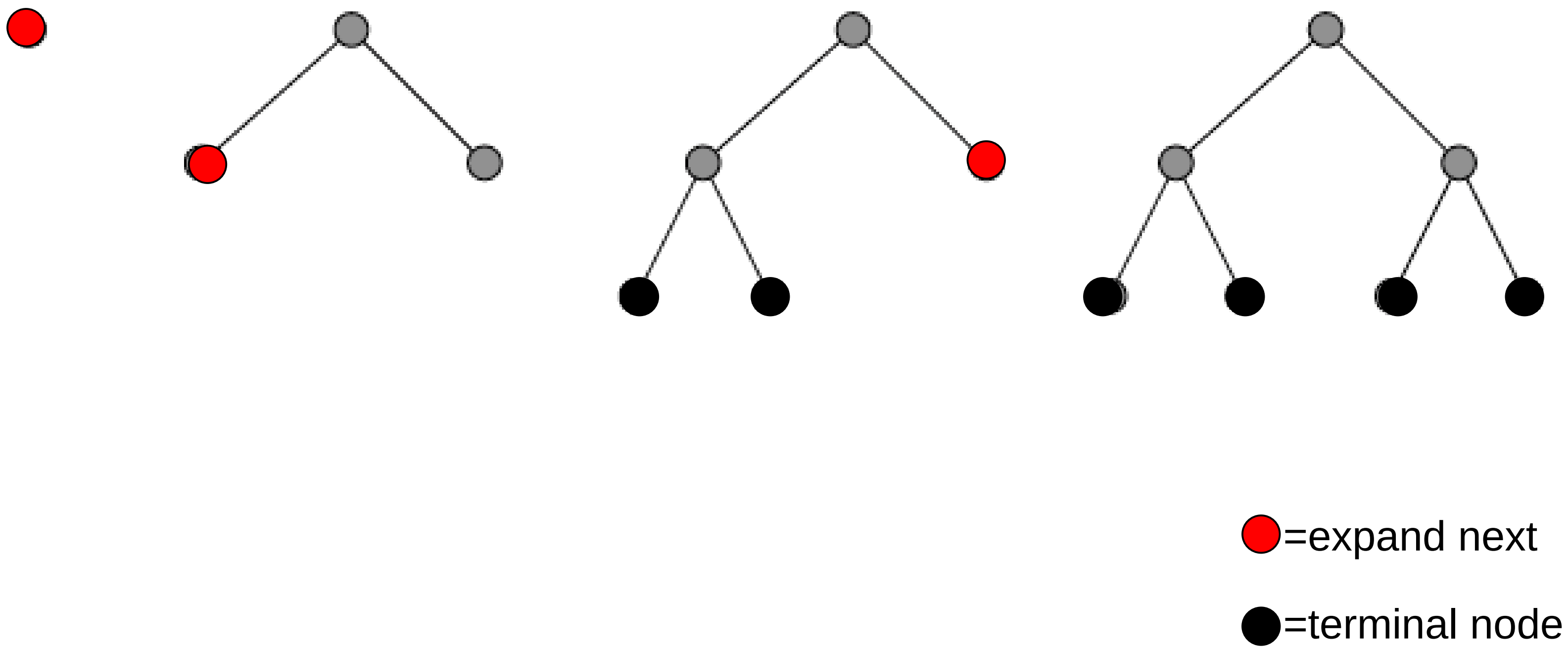
Search algorithms differ in their queuing function.

Uninformed (blind) search

- If a state is not a goal, you cannot tell how close to the goal it might be.
- Hence, all you can do is move systematically between states until you stumble on a goal.
- In contrast, informed (heuristic) search uses a guess on how close to the goal a state might be. (*More on this next class*.)
- Let's first look at several basic uninformed search algorithms:
  - **Breadth-first search**
  - **Uniform-cost search**
  - **Depth-first search**

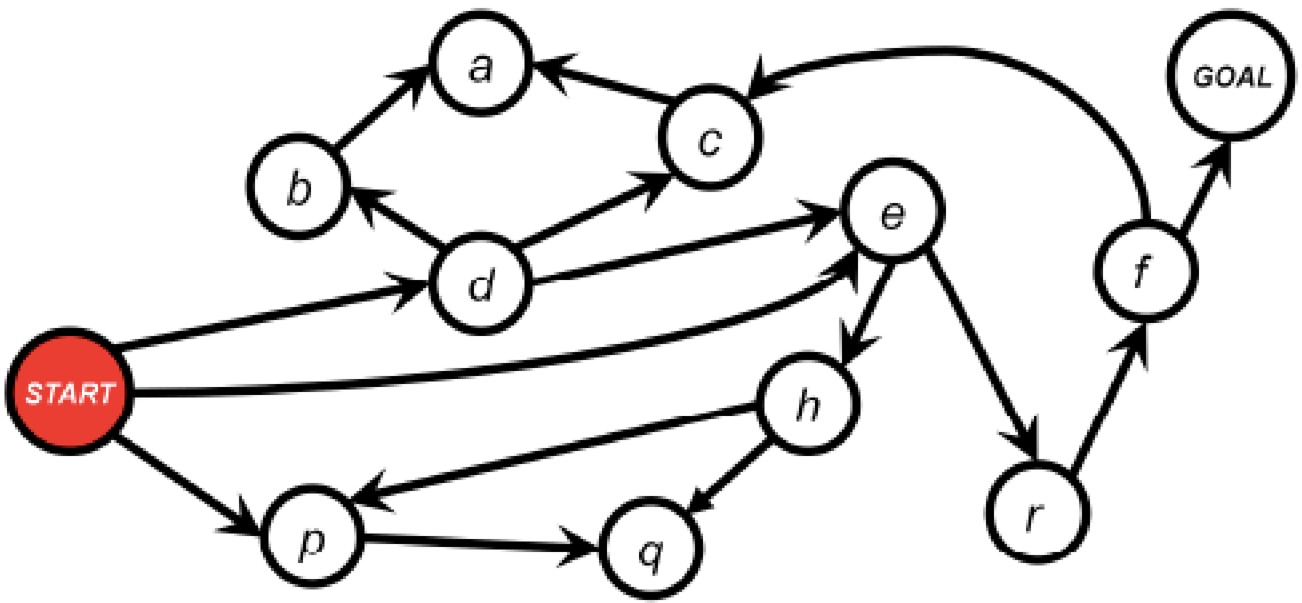
Breadth-First Search (BFS)

- Enqueue nodes at the end of queue.
- All nodes at level i get expanded before all nodes at level i+1.



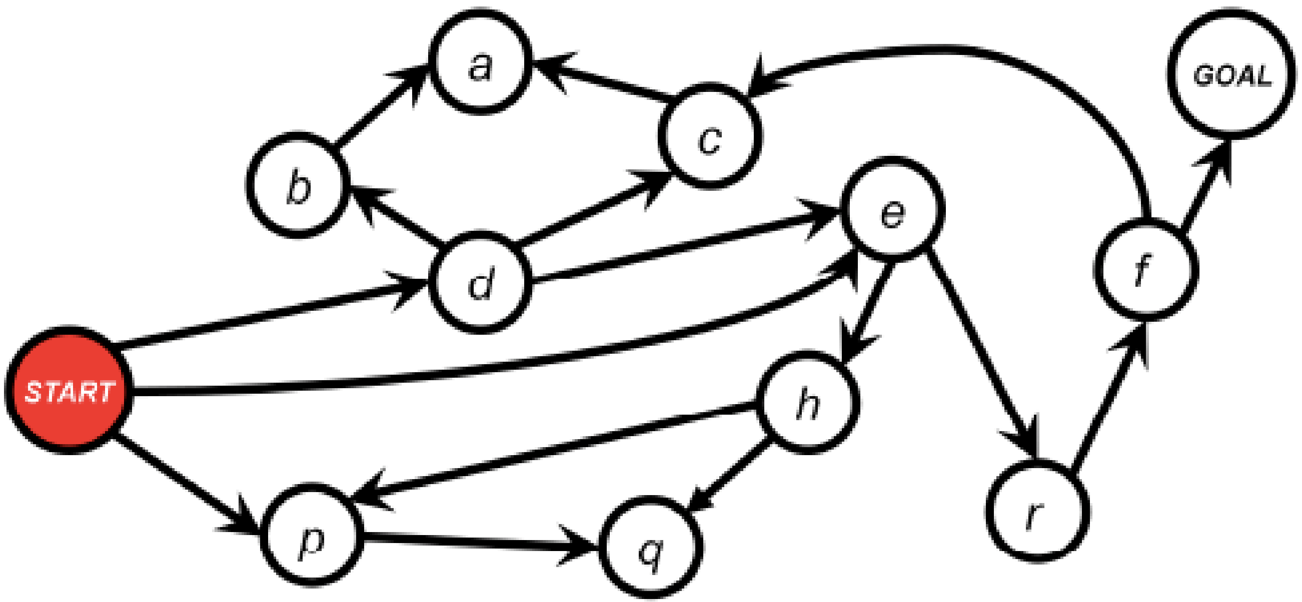
Example

- In what order are nodes expanded using Breadth-first search?



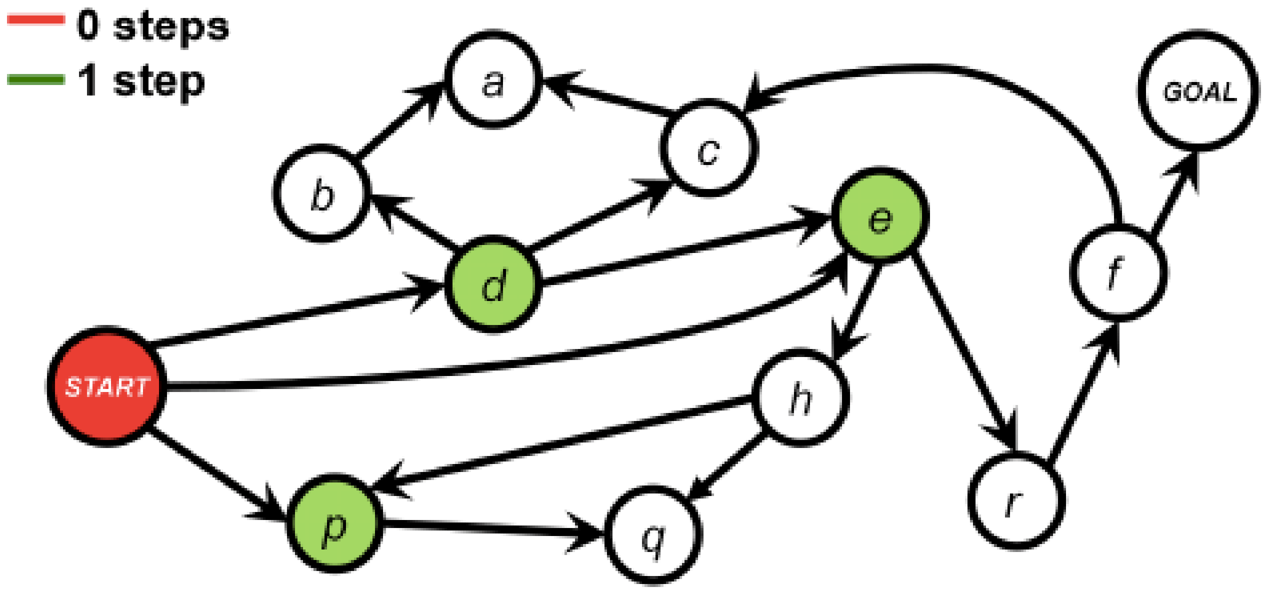
Example

- Label all start states as V_0.



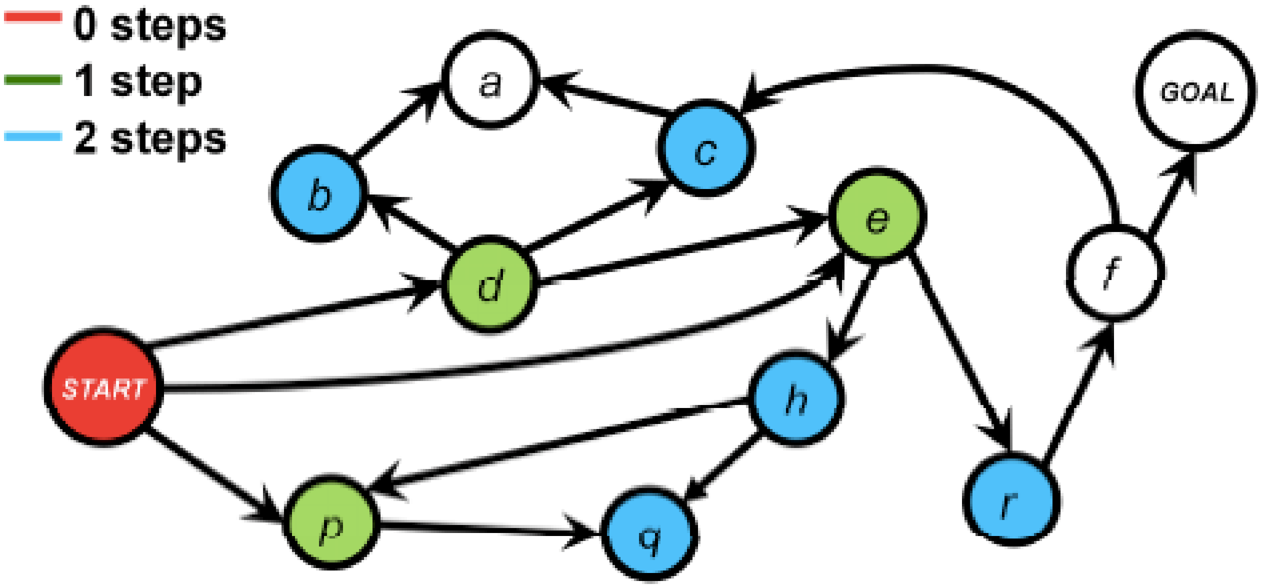
Example

- Label all successors of states in V_0 that have not been labeled as V_1.



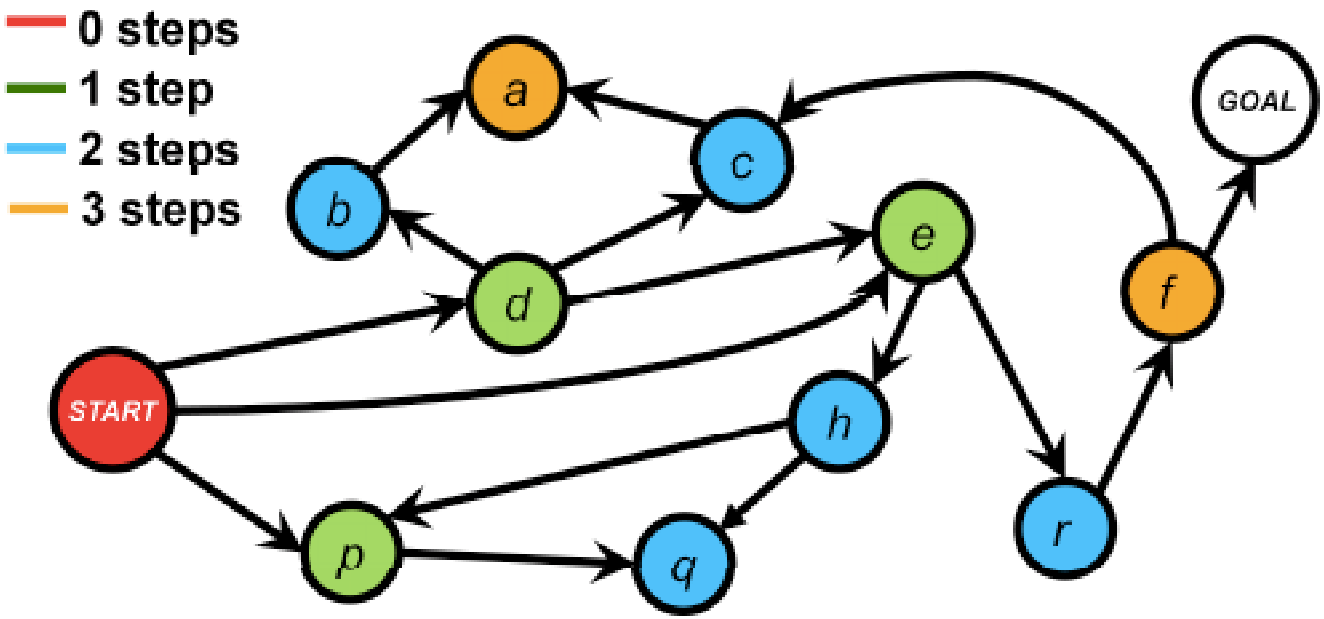
Example

- Label all successors of states in V_1 that have not been labeled as V_2.



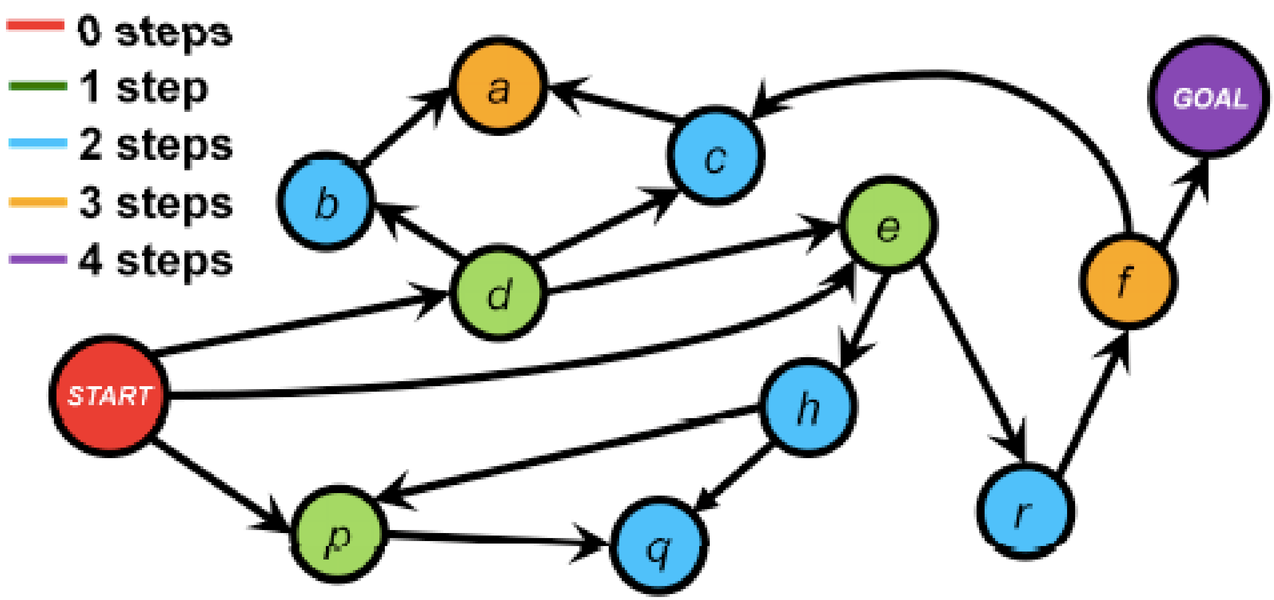
Example

- Label all successors of states in V_2 that have not been labeled as V_3.



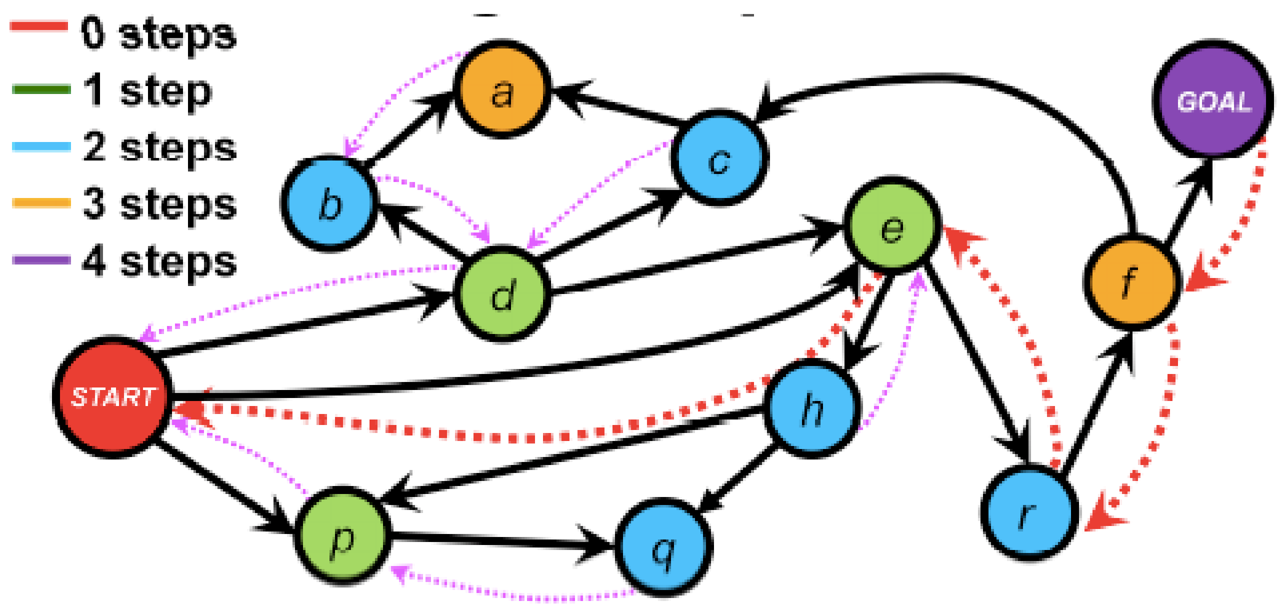
Example

- Label all successors of states in V_3 that have not been labeled as V_4.



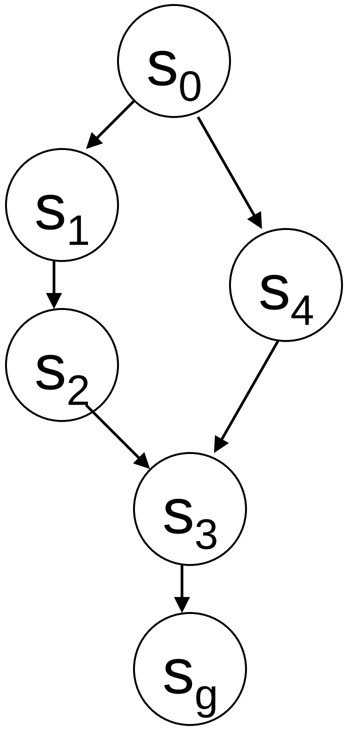
Example

- To recover the path, follow pointers back to the parent node.



Revisiting states

- What if we revisit a state that was already expanded?
- What if we visit a state that was already in the queue?
- Example:



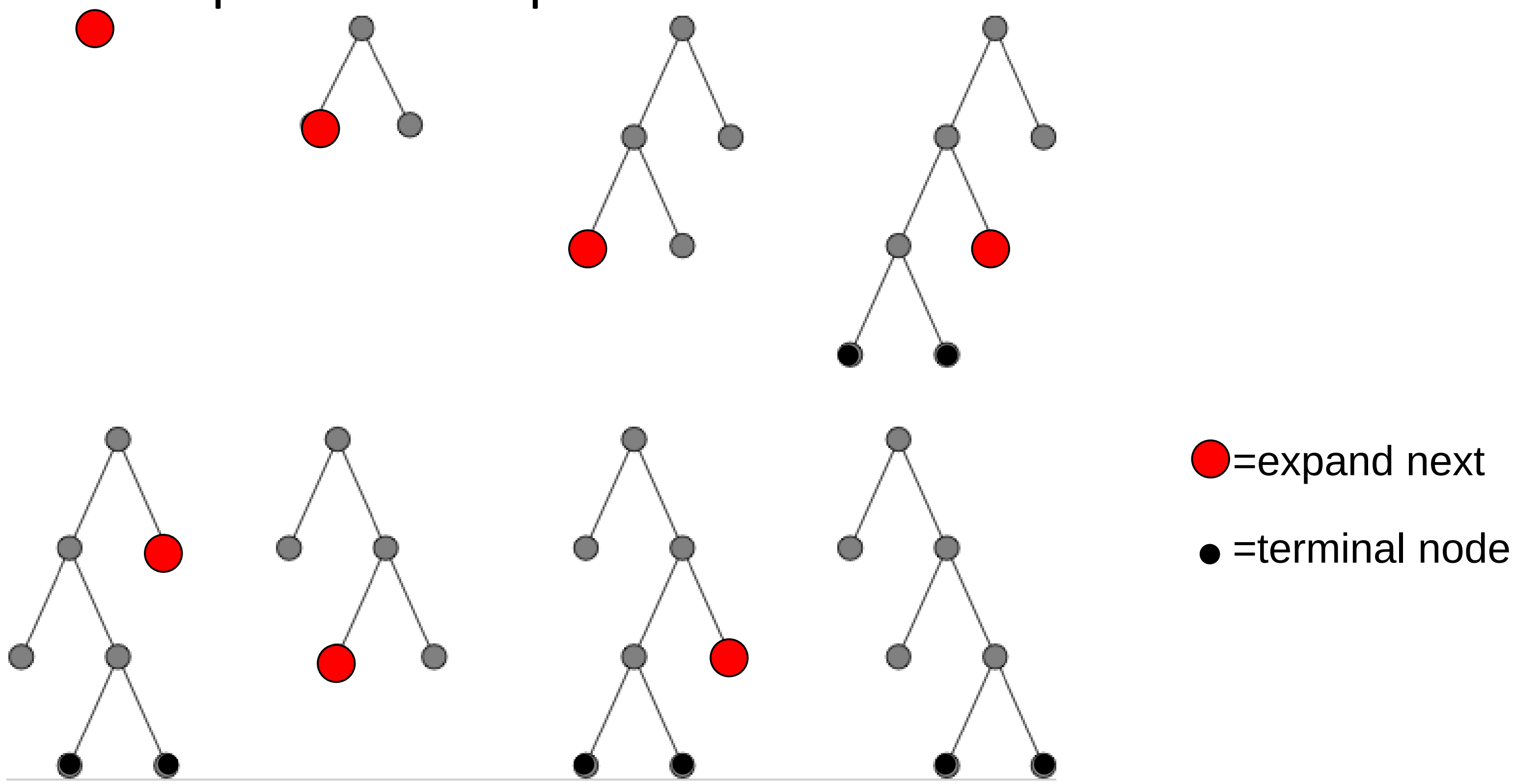
Revisiting states

- Maintain a closed list to store every expanded node.
  - More efficient on problems with many repeated states.
  - Worst-case time and space requirements are $O(|S|)$ ($|S|$ = #states)

- In some cases, allowing states to be re-expanded could produce a better solution.
  - When repeated state is detected, compare old and new path to find lowest cost path.
  - In large domains, may not be able to store all states.

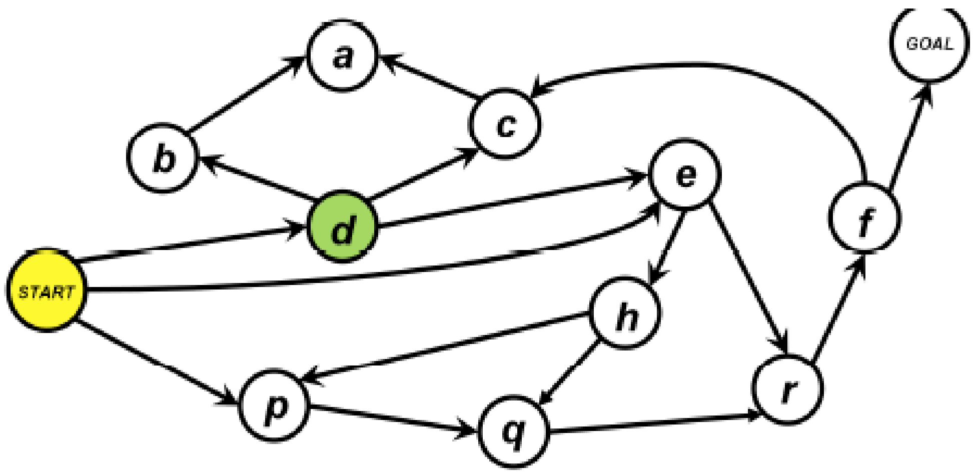
Depth-First Search (DFS)

- Enqueue nodes at the front of queue
  - In other words, use a stack in stead of a queue.
- Nodes at deepest level expanded before shallower ones.



Example

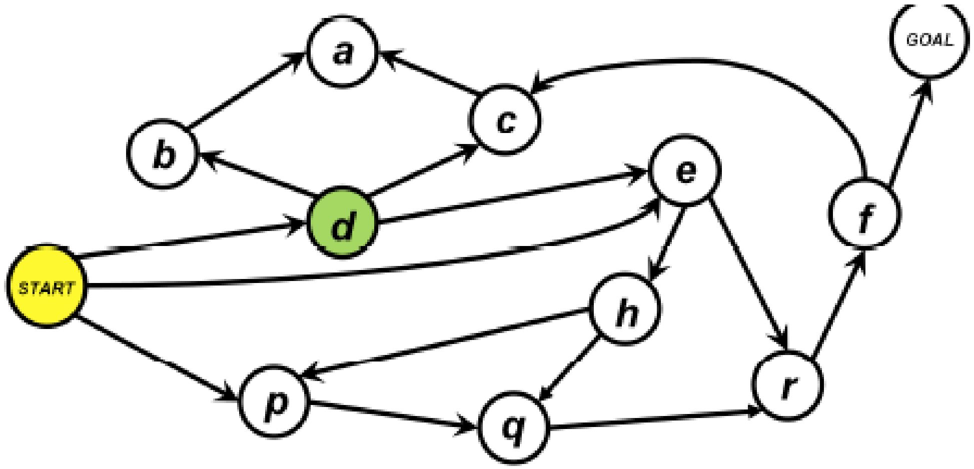
In what order are nodes expanded using Depth-first search?



Example

In what order are nodes expanded using Depth-first search?

Order of operators matter!

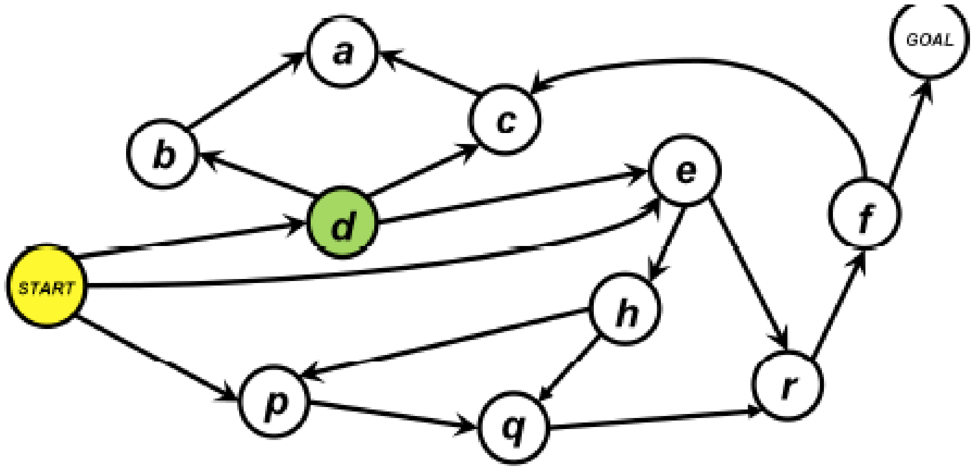


Solution (*if you expand nodes in clockwise order, staring at 9 o'clock*): {Start, d, b, a, c, e, r, f, Goal}

Example

In what order are nodes expanded using Depth-first search?

Order of operators matter!



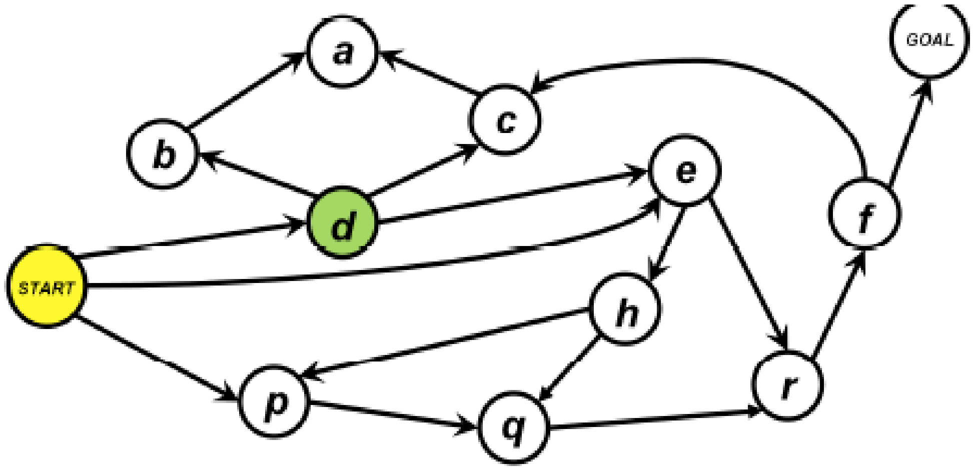
Solution (*if you expand nodes in clockwise order, staring at 9 o'clock*): {Start, d, b, a, c, e, r, f, Goal}

What if we expand nodes **counter-clockwise**, from 9 o'clock?

Example

In what order are nodes expanded using Depth-first search?

Order of operators matter!



Solution (*if you expand nodes in clockwise order, staring at 9 o'clock*): {Start, d, b, a, c, e, r, f, Goal}

Solution (*if we expand nodes counter-clockwise*): {Start, p, q, r, f, Goal}

Key properties of search algorithms

- **Completeness**: Are we assured to find a solution, if one exists?
- **Optimality**: How good is the solution?
- **Space complexity**: How much storage is needed?
- **Time complexity**: How many operations are needed?

Key properties of search algorithms

- **Completeness**: Are we assured to find a solution, if one exists?
- **Optimality**: How good is the solution?
- **Space complexity**: How much storage is needed?
- **Time complexity**: How many operations are needed?

- Other desirable properties:
  - Can the algorithm provide an intermediate solution?
  - Can an inadequate solution be refined or improved?
  - Can the work done on one search be reused for a different set of start/goal states?

Search complexity

- Evaluated in terms of two characteristics:

  - **Branching factor** of the state space ("b"): how many operators (upper limit) can be applied at any time?

    E.g. for the eight-puzzle problem: branching factor is 4, although most of the time we can apply only 2 or 3 operators.

  - **Solution depth** ("d"): how long is the path to the closest (shallowest) goal state?

Analyzing BFS

- **Good news**:
  - Complete.
  - Paths to different goals can be explored at the same time.
  - Guaranteed to find shallowest path to the goal if unit cost per step.

    Will not necessarily find optimal path if **cost per step is non-uniform**.

Analyzing BFS

- **Good news**:
  - Complete.
  - Paths to different goals can be explored at the same time.
  - Guaranteed to find shallowest path to the goal if unit cost per step.

    Will not necessarily find optimal path if **cost per step is non-uniform**.

- **More bad news**:
  - Exponential time complexity: $O(b^d)$ [This is same for all uninformed search algorithms.]
  - Exponential space complexity: $O(b^d)$ [This is not good!]

Uniform Cost Search

- Goal: Fix BFS to ensure an optimal path with general step costs.

Important distinction:

- **Unit cost** = Problem where each action has the same cost.
- **General cost** = Actions can have different costs.

Uniform Cost Search

- Goal: Fix BFS to ensure an optimal path with general step costs.

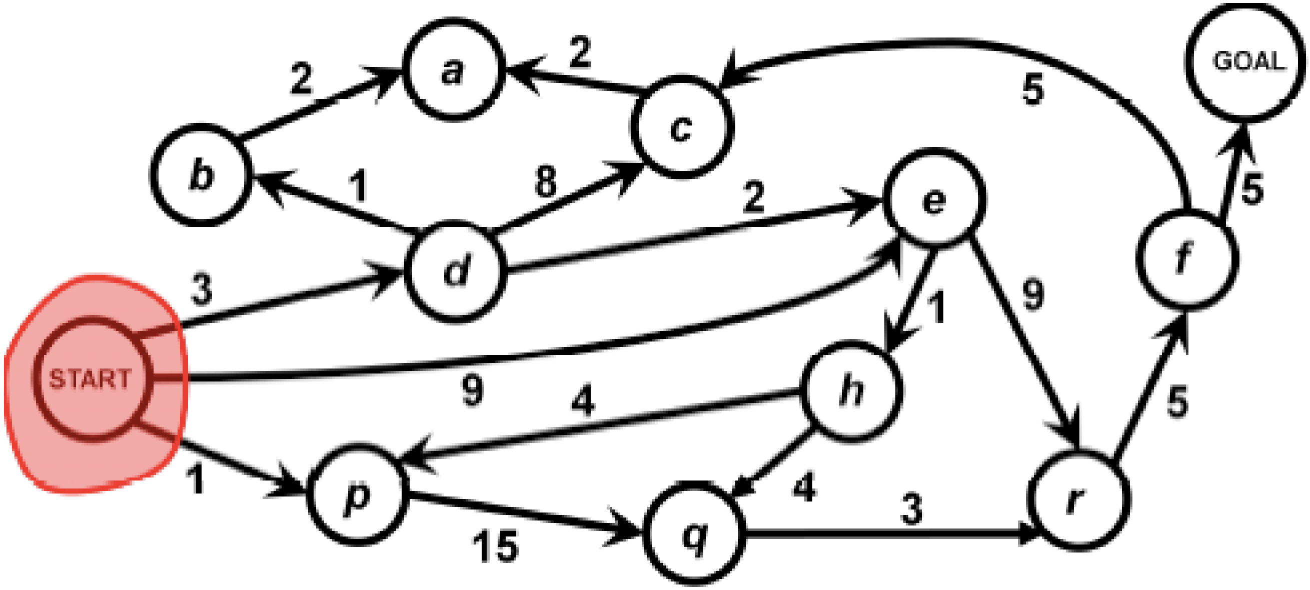
Important distinction:

- **Unit cost** = Problem where each action has the same cost.
- **General cost** = Actions can have different costs.

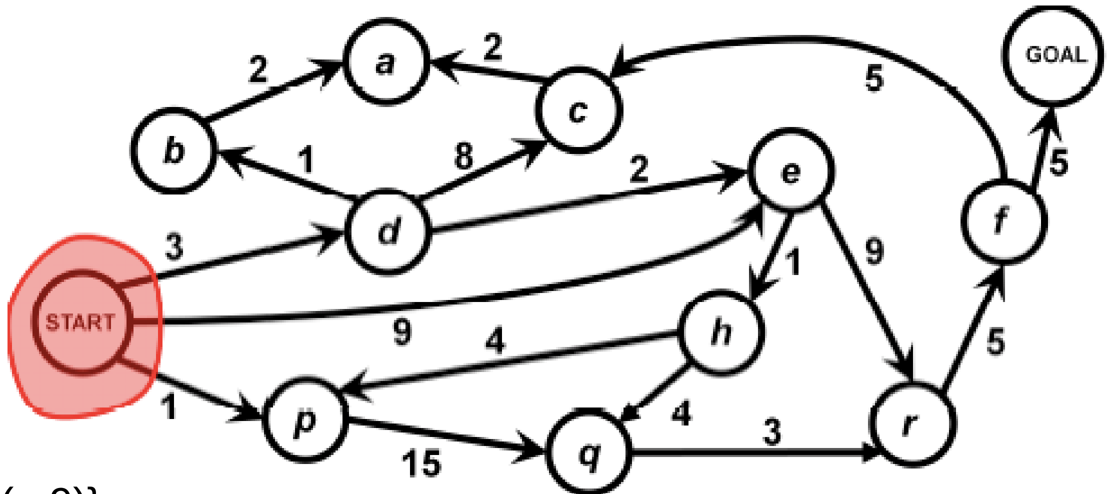
- Approach:
  - Use a **priority queue** instead of a simple queue.
  - Insert nodes in the increasing order of the cost of the path so far.
- Properties:
  - Guaranteed to find **optimal solution** for with general step costs (same as BFS when all operators have the same cost).

Example

- In what order are nodes expanded using Uniform cost search?



Example - solved



Priority queue = {(Start, 0)}
= {~~(Start, 0),~~ (p,1), (d,3), (e,9)}
= {~~(Start, 0), (p,1),~~ (d,3), (e,9), (q,16)}
= {~~(Start, 0), (p,1), (d,3),~~ (b,4), (e,5), (c,11), (q,16)} [Find faster path to e.]
= {~~(Start, 0), (p,1), (d,3), (b,4),~~ (e,5), (a,6), (c,11), (q,16)}
= {~~(Start, 0), (p,1), (d,3), (b,4), (e,5),~~ (a,6), (h,6), (c,11), (r,14), (q,16)}
= {~~(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6),~~ (h,6), (c,11), (r,14), (q,16)}
= {~~(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6),~~ (q,10), (c,11), (r,14)}
= {~~(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10),~~ (c,11), (r,13)}
= {~~(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10), (c,11),~~ (r,13), (f,18)}
= {~~(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10), (c,11), (r,13),~~ (f,18)}
= {~~(Start, 0), (p,1), (d,3), (b,4), (e,5), (a,6), (h,6), (q,10), (c,11), (r,13), (f,18),~~ (goal,23)}

Analyzing DFS

- **Good news:**
  - Linear space complexity: *O(bm)*
    - *m* is the maximum depth of the tree
  - Easy to implement recursively (do not even need queue data structure).
  - More efficient than BFS if there are many paths leading to solution.

Analyzing DFS

- **Good news**:
  - Linear space complexity: *$O(bm)$*
  - Easy to implement recursively (do not even need queue data structure).
  - More efficient than BFS if there are many paths leading to solution.
- **Bad news**:
  - Exponential time complexity: *$O(b^m)$* [This is same as BFS]
  - Not optimal.
  - DFS may not complete!
  - NEVER use DFS if you suspect a big tree depth!

More Search Algorithms

- **Depth-limited search**
- **Iterative deepening search**
- Bidirectional search (read Section 3.4.6 for more details)

Depth-limited search

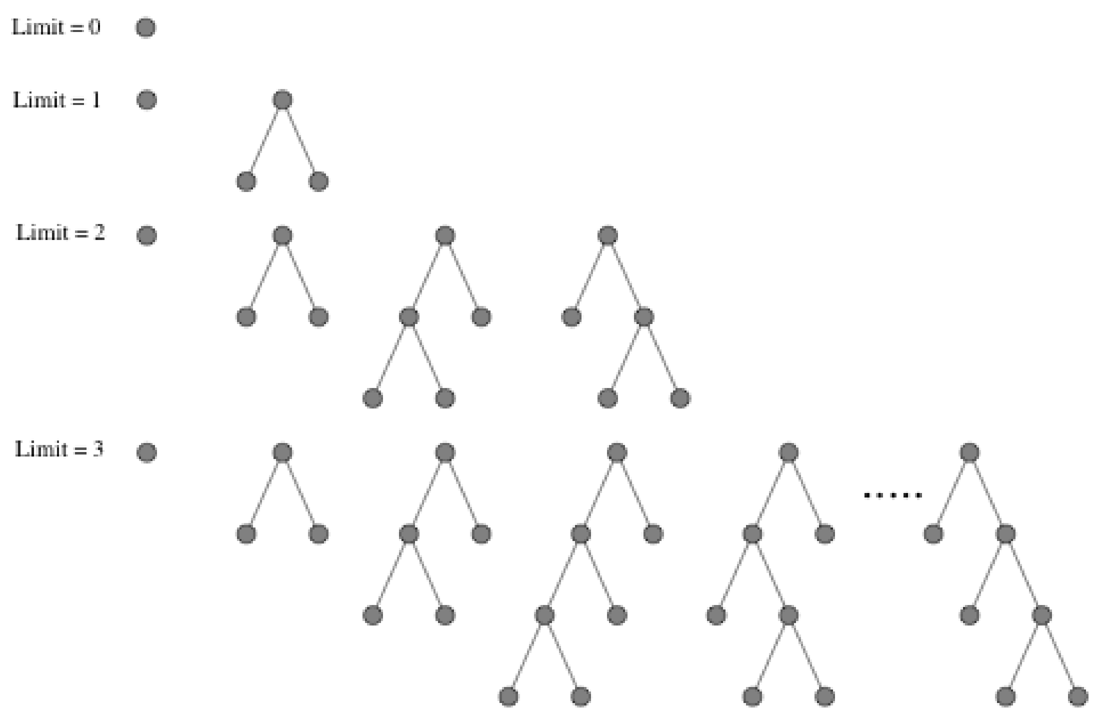
- **Algorithm**: search depth-first, but terminate a path either if a goal state is found, or if the maximum depth allowed is reached.

- Always terminates:
  - Avoids the problem of search never terminating by imposing a hard limit on the depth of any search path.

- However, it is still not complete (the goal depth may be greater than the limit allowed.)

Iterative Deepening Search (IDS)

- **Algorithm**: do depth-limited search, but with increasing depth.
- Expands nodes multiple times, but computation time has same complexity.



Analysis of IDS

- **Complete** (like BFS).
- Has **linear memory** requirements (like DFS).
- Classical time-space tradeoff.
  - **Recall from last lecture**: achieving rationality *subject to resource constraints*
  - Will see similar trade-off often in this course.

Analysis of IDS

- **Complete** (like BFS).
- Has **linear memory** requirements (like DFS).
- Classical time-space tradeoff.
  - **Recall from last lecture**: achieving rationality *subject to resource constraints*
  - Will see similar trade-off often in this course.
- **Optimal** for problems with unit step costs.
- This is the preferred method for large state spaces, where the solution path length is unknown.

Questions

- Which method should you use if….
  - You need to find the optimal solution?
  - The state space is VERY large?
  - You have limited memory?
  - You want to find quickly find the best solution within a cost budget?

Questions

- Which method should you use if….
  - You need to find the optimal solution?
    - BFS, DFS or iterative deepening if unit cost.
    - Uniform-cost search if general cost.
  - The state space is VERY large?
    - Depth-first search if you know the maximum plan length.
    - Iterative deepening search otherwise.
  - You have limited memory?
    - Depth-first search / iterative deepening search.
  - You want to find quickly find the best solution within a cost budget?
    - Depth-limited search if unit cost.
    - Uniform-cost search if general cost.

Summary of uninformed search

- Assumes no knowledge about the problem.
- Main difference between methods is the order in which they consider states.
    - BFS
    - Uniform cost search
    - DFS
    - Fixed-depth DFS
    - Iterative deepening
- Very general, can be applied to any problem.
- Very expensive, since we assume no knowledge about the problem.
- Some algorithms are complete, i.e. they will find a solution if one exists.
- All **uninformed** search methods have **exponential** worst-case complexity.