Lecture 2: Uninformed search methods

- Search problems
- Generic search algorithms
- Criteria for evaluating search algorithms
- Uninformed Search
  - Breadth-First Search
  - Depth-First Search
  - Iterative Deepening
- Heuristics

Search in AI

- One of the first and major topics:
  Newell & Simon (1972). *Human Problem Solving*
- Central component to many AI systems:
  - Automated reasoning
  - Theorem proving
  - Game playing
  - Navigation

Example: Eight-Puzzle

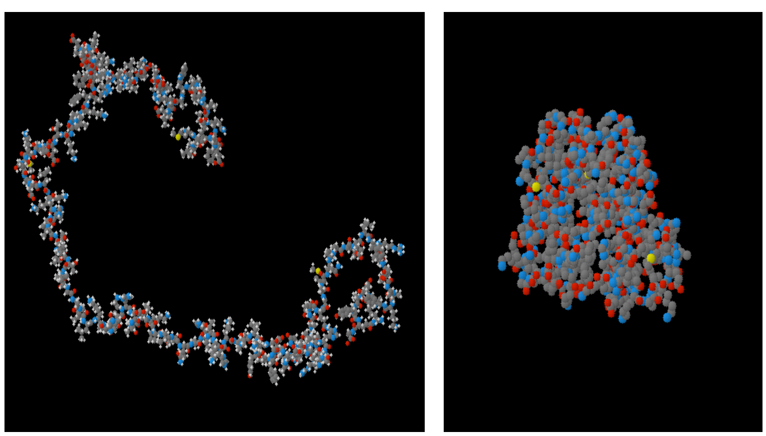
5	4	
6	1	8
7	3	2

Start State

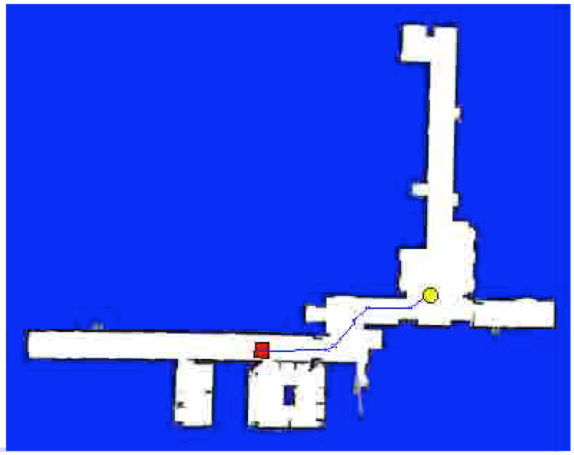
1	2	3
8		4
7	6	5

Goal State

Example: Protein creation



Example: Robot navigation



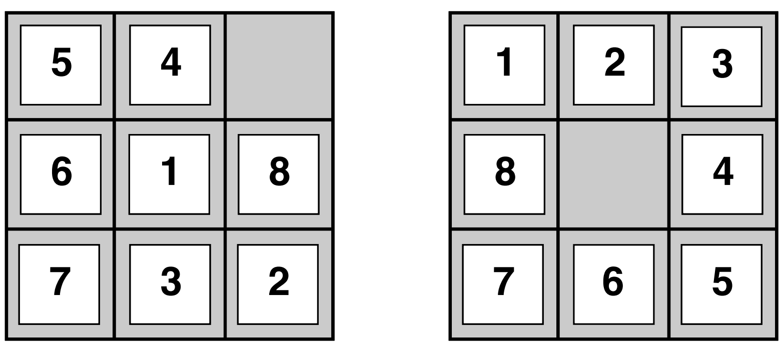
Defining a Search Problem

- *State space* S: all possible configurations of the domain of interest
- *An initial (start) state* $s_0 \in S$
- *Goal states* $G \subset S$: the set of end states
  - Often defined by a *goal test* rather than enumerating a set of states
- *Operators* A: the actions available
  - Often defined in terms of a *mapping from a state to its successor*

Defining a search problem (2)

- *Path*: a sequence of states and operators
- *Path cost*: a number associated with any path
  - Measures the quality of the path
  - Usually the smaller, the better
- *Solution* of a search problem is a path from s_0 to some $s_g \in G$
- *Optimal solution*: any path with minimum cost.

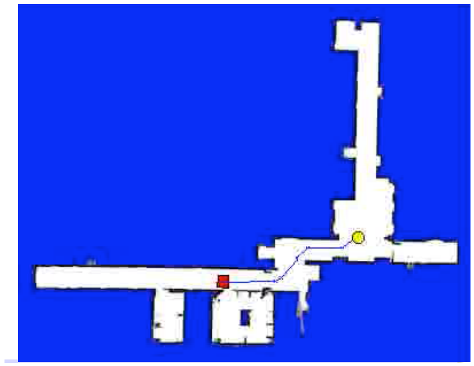
Example: Eight-Puzzle



Start State **Goal State**

- States: configurations of the puzzle
- Goals: target configuration
- Operators: swap the blank with an adjacent tile
- Path cost: number of moves

Example: Robot navigation



- States: position, velocity, map, obstacles, ...
- Goals: get to target position without crashing
- Operators: usually small steps in several directions
- Path cost: length of path, energy consumption, cost, ...

Assumptions

- *Static* (vs dynamic) environment
- *Observable* (vs unobservable) environment
- *Discrete* (vs continuous) state space
- *Deterministic* (vs stochastic) environment

The general search problem formulation does not make these assumptions, but we will make them when discussing search algorithms

Coding a Generic Search Problem in Java

```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

```java
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

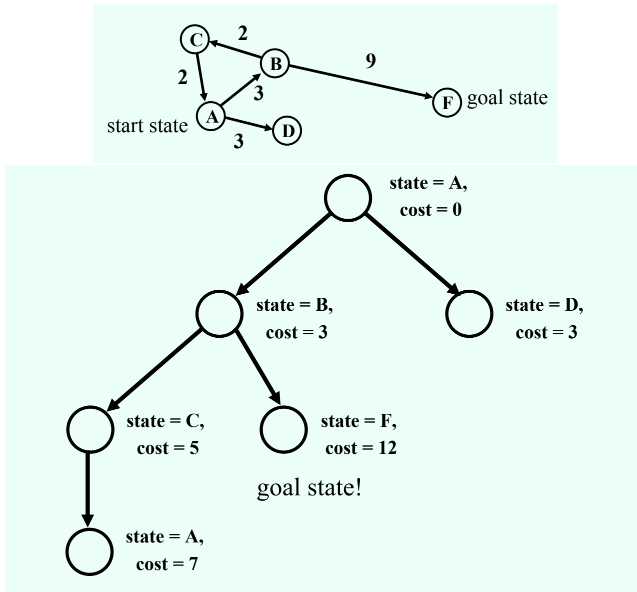
Coding a Generic Search Problem in C

- Write code for the different problems in separate files
- Be disciplined about the way in which functions are called (basically do the checks of an object-oriented parser)
- Write different search algorithms in different files
- Link together files as appropriate.

Representing Search: Graphs and Trees

- Visualize a state space search in terms of a *graph*
  - *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!

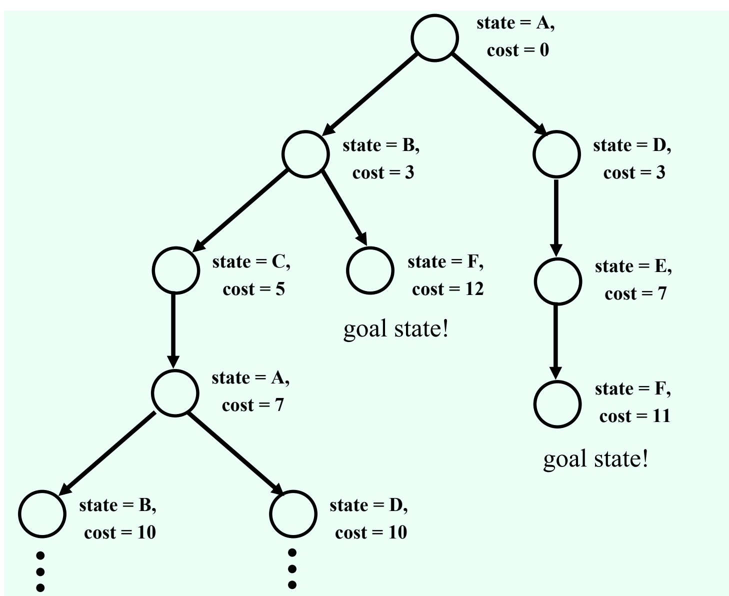
Data Structures for Search

- *Defining a search node*:
  - Each node contains a state
  - Node also contains additional information, e.g.:
    - The parent state and the operator used to generate it
    - Cost of the path so far
    - Depth of the node
- *Expanding a node*:
  - Applying all legal operators to the state contained in the node
  - Generating nodes for all the corresponding successor states.

Generic Search Algorithm

1. Initialize the search tree using the initial state of the problem
2. Repeat
   (a) If no candidate nodes can be expanded, return failure
   (b) Choose a leaf node for expansion, according to some search strategy
   (c) If the node contains a goal state, return the corresponding path
   (d) Otherwise expand the node by:
      - Applying each operator
      - Generating the successor state
      - Adding the resulting nodes to the tree

Problem: Search trees can get very big!



Implementation Details

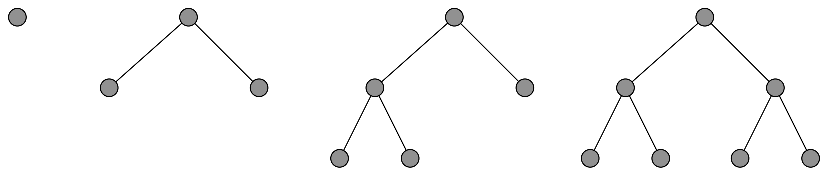
- We need to keep track only of the nodes that need to be expanded - *frontier* or *open list*
- This can be implemented using a (prioritized) *queue*:
  1. Initialize the queue by inserting the node for the initial state
  2. Repeat
     (a) If the queue is empty, return failure
     (b) Dequeue a node
     (c) If the node contains a goal state, return the path
     (d) Otherwise expand the node, inserting the resulting nodes into queue
- *Search algorithms differ in their queuing function!*

Uninformed (blind) search

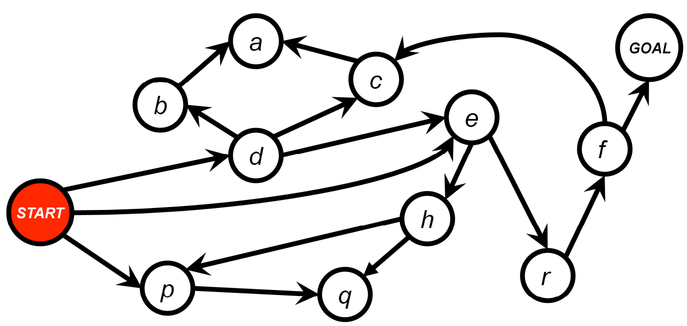
- If a state is not a goal, we cannot tell how close to the goal it might be
- Hence, all we can do is move systematically between states until we stumble on a goal
- In contrast, informed (heuristic) search uses a guess on how close to the goal a state might be

Breadth-First Search (BFS)

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

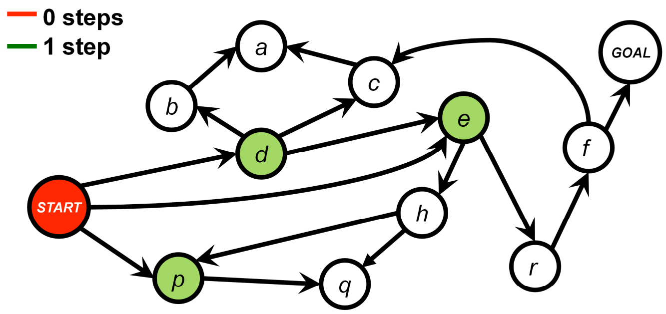


Example



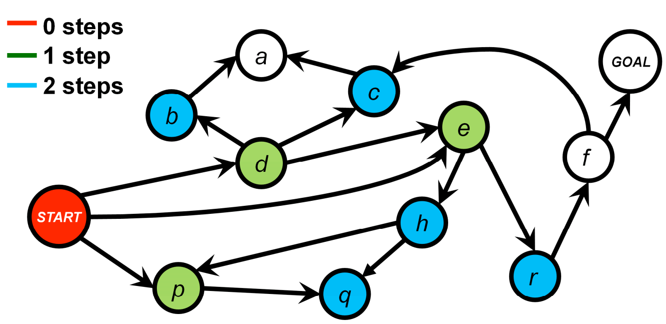
Label all start states as set V_0

Example



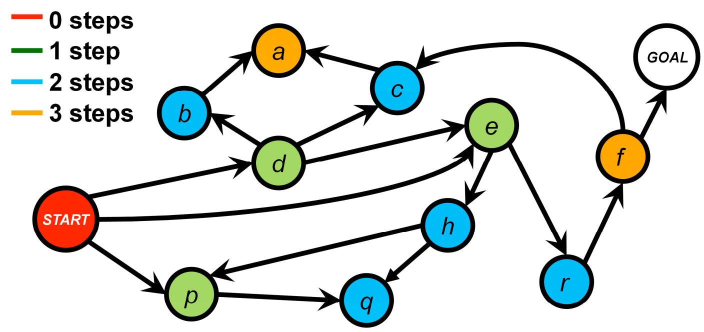
Label all successors of states in V_0 that have not yet been labelled as set V_1

Example



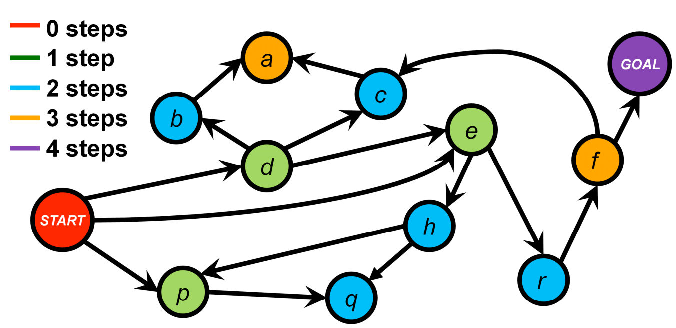
Label all successors of states in V_1 that have not yet been labelled as set V_2

Example



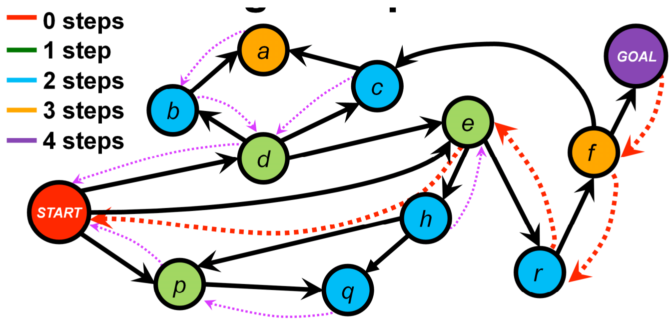
Label all successors of states in V_2 that have not yet been labelled as set V_3

Example



Label all successors of states in V_3 that have not yet been labelled as set V_4

Example: Recovering the path



Follow pointers back to the parent node to find the path

Key Properties of Search Algorithms

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

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 re-used for a different set of start/goal states?

Search Performance

It is evaluated in terms of two characteristics of the problem:

- *Branching factor of the search space (b)*: how many operators (at most) can be applied at any time?

  E.g. For the eight-puzzle problem, the branching factor is considered 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) solution?

Analyzing BFS

- Good news:
  - Complete
  - Guaranteed to find the *shallowest* path to the goal
    This is not necessarily the best path! But we can "fix" the algorithm to get the best path.
  - Different start-goal combinations can be explored at the same time

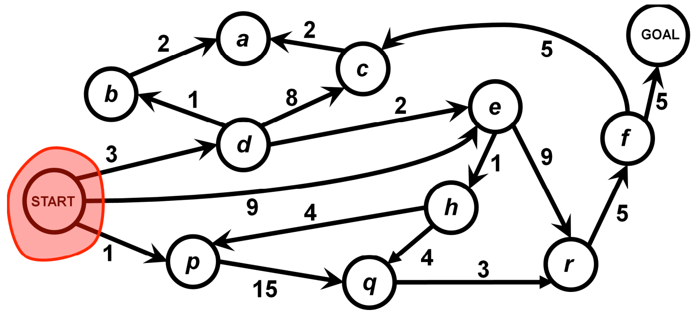
Analyzing BFS

- Good news:
  - Complete
  - Guaranteed to find the *shallowest* path to the goal
    This is not necessarily the best path! But we can "fix" the algorithm to get the best path.
  - Different start-goal combinations can be explored at the same time
- Bad news:
  - Exponential time complexity: $O(b^d)$ (why?)
    This is the same for all uninformed search methods
  - *Exponential memory requirements!* $O(b^d)$ (why?)
    This is not good...

Fixing BFS To Get An Optimal Path

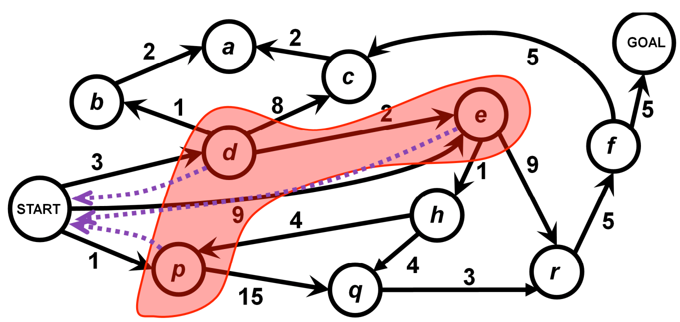
- Use a priority queue instead of a simple queue
- Insert nodes in the increasing order of the cost of the path so far
- Guaranteed to find an optimal solution!
- This algorithm is called *uniform-cost search*

Example



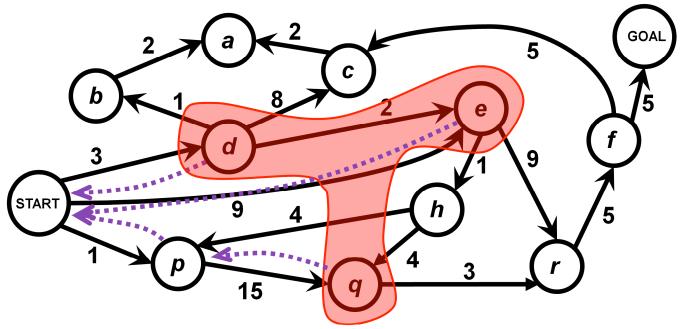
PQ = {(START,0)}

Example



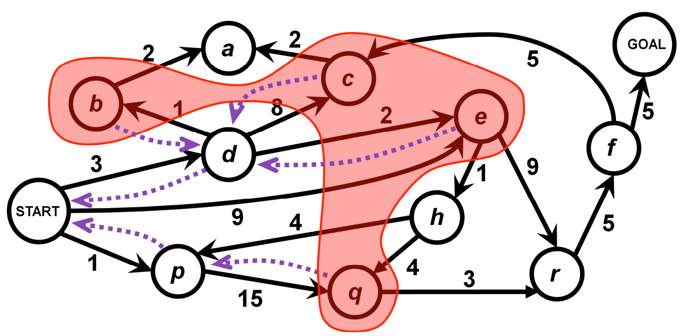
PQ = {(p,1) (d,3) (e,9)}

Example



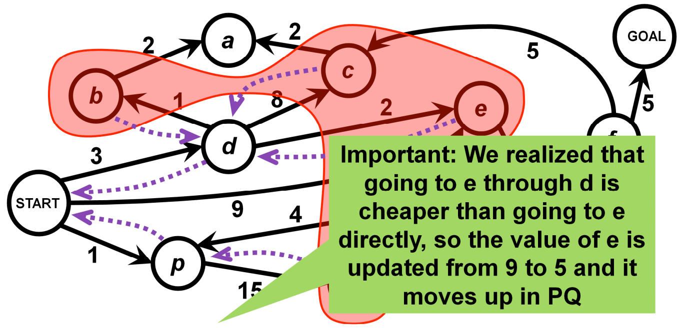
PQ = {(d,3) (e,9) (q,16)}

Example



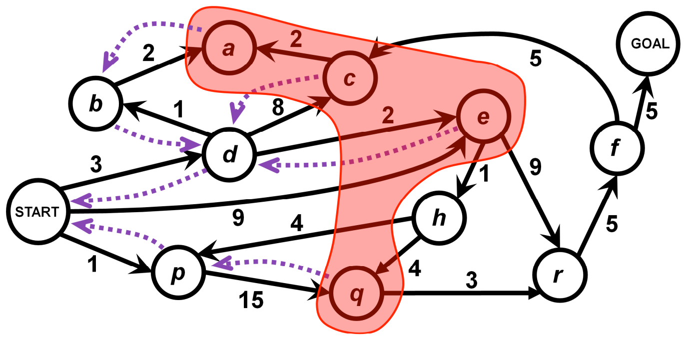
PQ = {(b,4) (e,5) (c,11) (q,16)}

Example



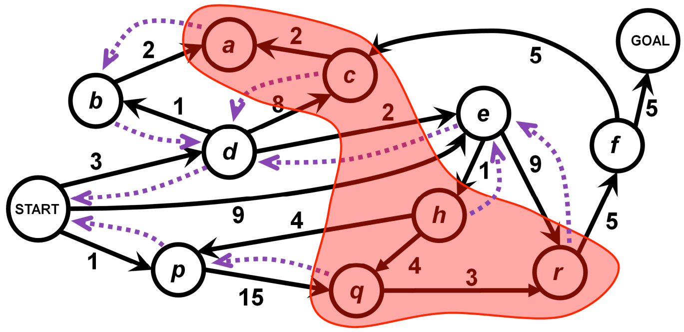
PQ = {(b,4) (e,5) (c,11) (q,16)}

Example



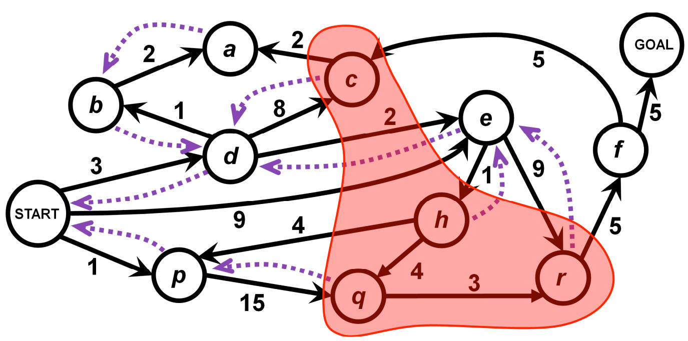
PQ = {(e,5) (a,6) (c,11) (q,16)}

Example



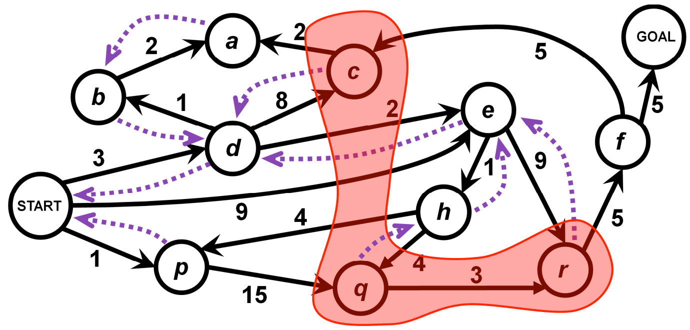
PQ = {(a,6) (h,6) (c,11) (r,14) (q,16)}

Example



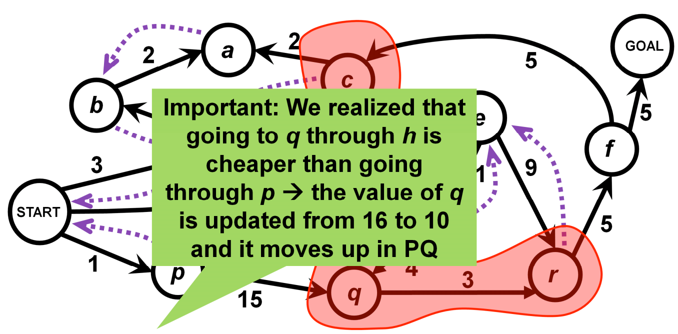
PQ = {(h,6) (c,11) (r,14) (q,16)}

Example



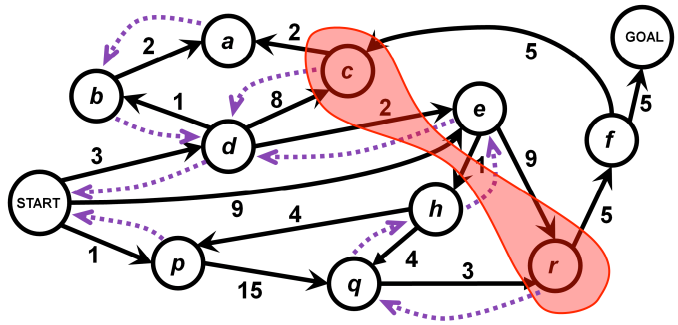
PQ = {(q,10) (c,11) (r,14)}

Example



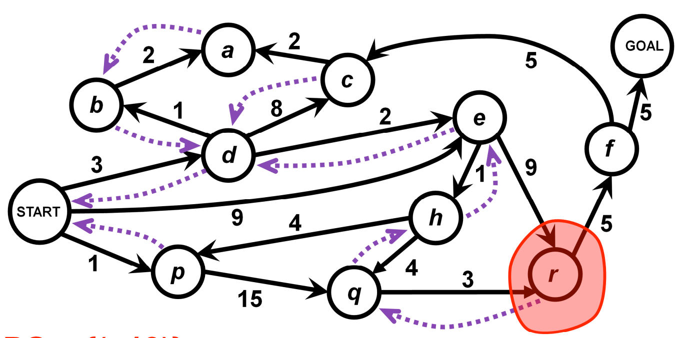
PQ = {(q,10) (c,11) (r,14)}

Example



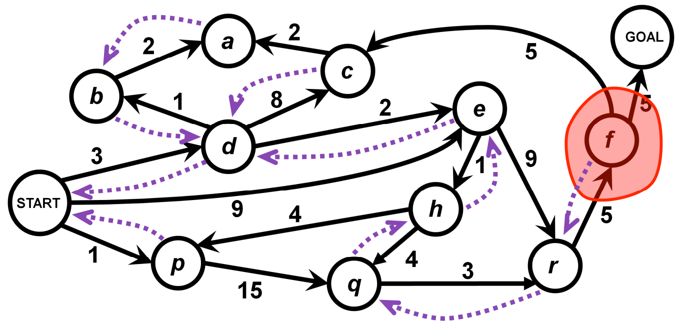
PQ = {(c,11) (r,13)}

Example



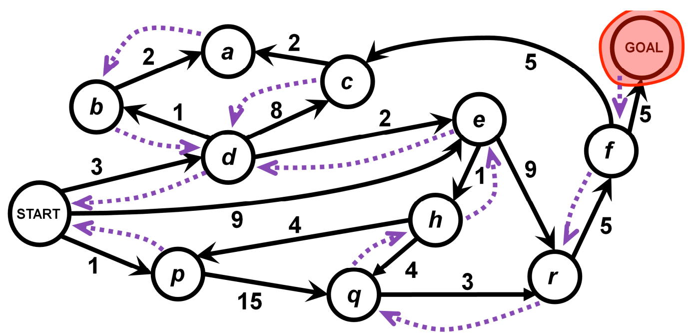
PQ = {(r,13)}

Example



PQ = {(f,18)}

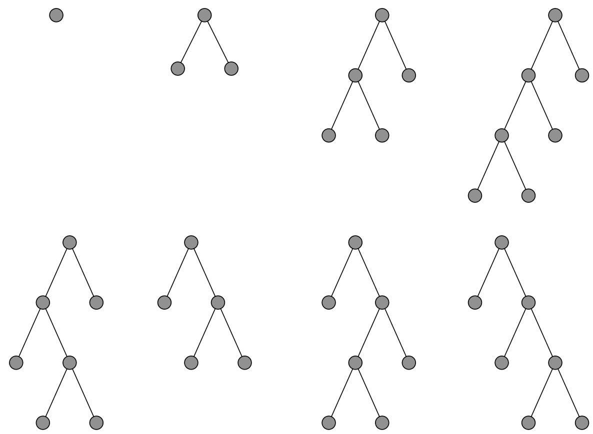
Example



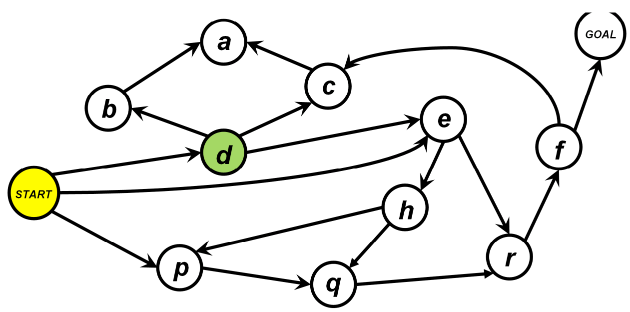
PQ = {(GOAL,23)}

Depth-First Search (DFS)

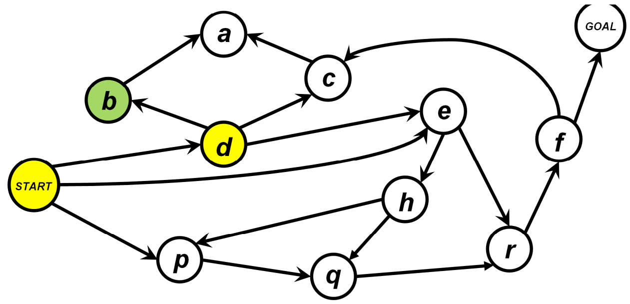
- Enqueues nodes *at the front of the queue*.
- Nodes at the deepest levels get expanded before shallower ones.



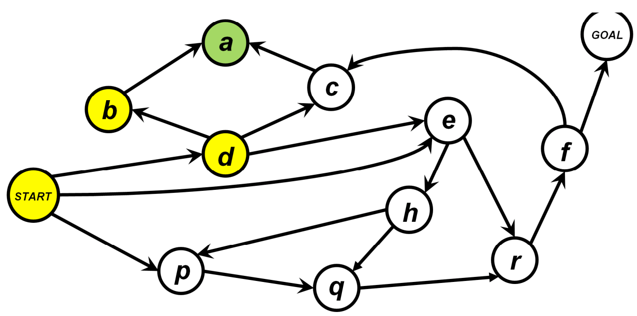
Example



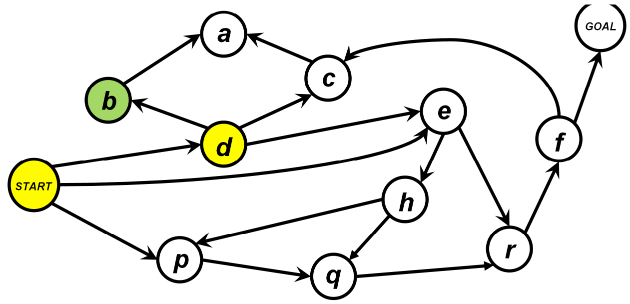
Example



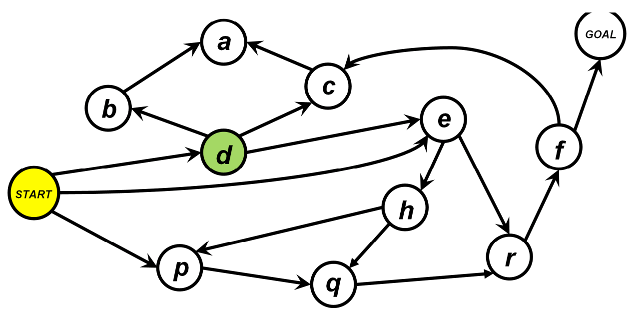
Example



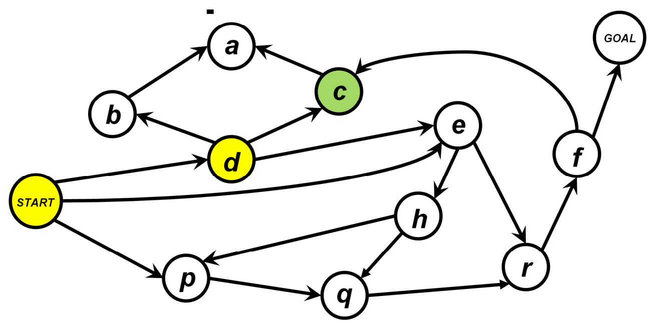
Example



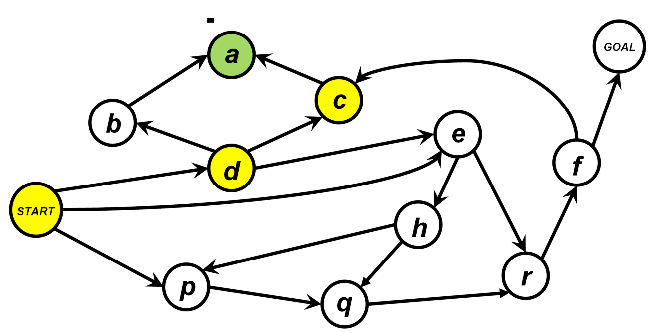
Example



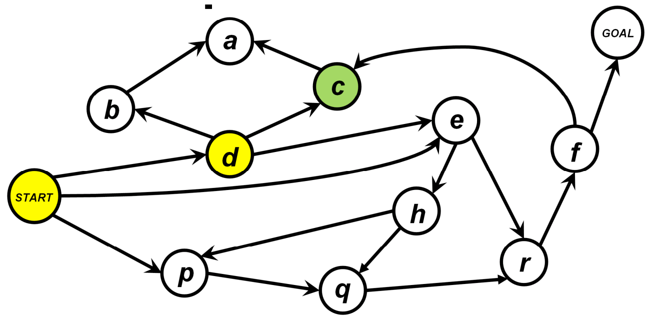
Example



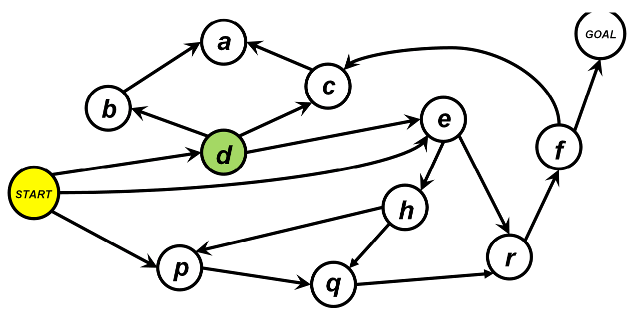
Example



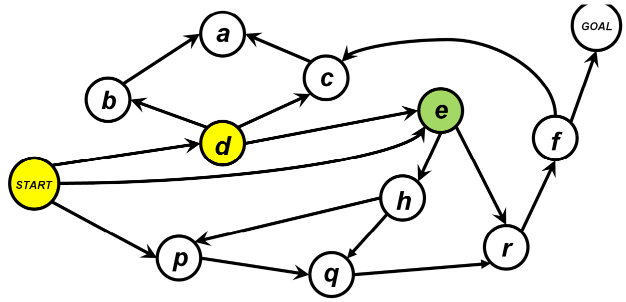
Example



Example



Example



Analyzing DFS

- Good news:
  - Space complexity $O(bd)$ (why?)
  - It is easy to implement recursively (do not even need a queue data structure)
  - More efficient than BFS if there are many paths leading to a solution.

Analyzing DFS

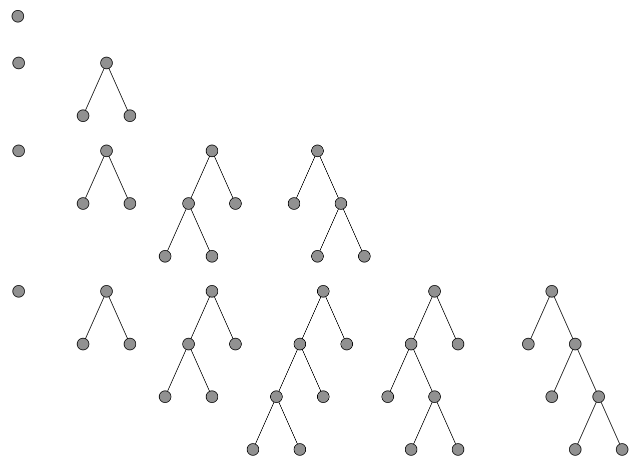
- Good news:
  - Space complexity $O(bd)$ (why?)
  - It is easy to implement recursively (do not even need a queue data structure)
  - More efficient than BFS if there are many paths leading to a solution.
- Bad news:
  - Exponential time complexity: $O(b^d)$
    This is the same for all uninformed search methods
  - Not optimal
  - *DFS may not complete!* (why?)
  - *NEVER* use DFS if you suspect a big tree depth

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.
- Unlike DFS, this algorithm *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

- Algorithm: do depth-limited search, but *with increasing depth*
- Expands nodes multiple times, but time complexity is the same

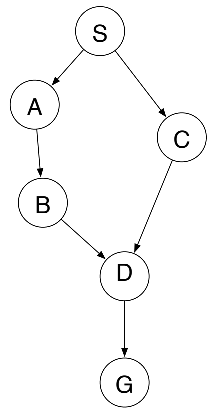


Analysis of Iterative Deepening Search

- *Complete (like BFS)*
- *Has linear memory requirements (like DFS)*
- Classical time-space tradeoff!
- This is the preferred method for large state space, where the maximum depth of a solution path is unknown

Revisiting states

- What if we revisit a state that was already expanded?
- We already saw an example of re-visiting a state that is already in the queue...



Revisiting states (2)

- Maintain a *closed list* to store every expanded node
  - Works best for problems with many repeated states
  - Worst-case time and space requirements are $O(|S|)$ where $|S|$ is the number of states
- Allowing states to be re-expanded could produce a better solution
  - When a repeated state is detected, compare the old and new path and keep best one

Uninformed Search Summary

- Assumes no knowledge about the problem
- Main difference between the methods is in the order in which they consider the states
- Very general, can be applied to any problem but 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

Informed Search

- Uninformed search methods expand nodes based on the *distance from the start node* $d(s_0, s)$

  Obviously, we always know that!
- But what about expanding based on *distance to the goal* $d(s, s_g)$?
- If we knew $d(s, s_g)$ exactly, it would be easy!

  Just expand the nodes needed to find a solution.
- Even if we do not know $d(s, s_g)$ exactly, we often have some *intuition* about this distance!
- We will call this intuition a *heuristic h(s)*.

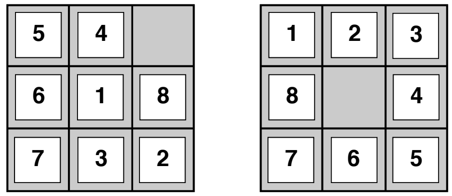
Example Heuristic: Path Planning

- Consider a path along a road system
- What is a reasonable heuristic?

Example Heuristic: Path Planning

- Consider a path along a road system
- What is a reasonable heuristic?
  - The straight-line distance from one place to another
- Is it always right?
  - Certainly not - roads are rarely straight!

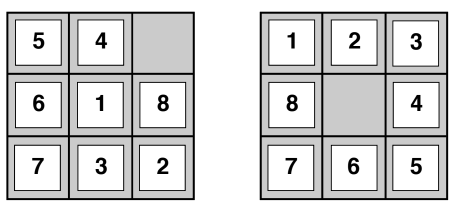
Example Heuristics: 8-puzzle



Start State Goal State

What would be good heuristics for this problem?

Example Heuristics: 8-puzzle

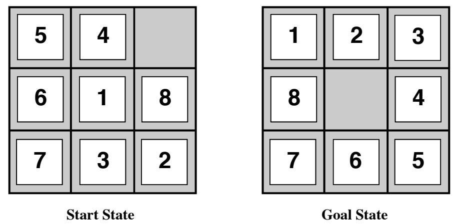


Start State Goal State

Consider the following heuristics:

- h_1 = number of misplaced tiles (=7 in example)
- h_2 = total Manhattan distance (i.e., no. of squares from desired location of each tile) (= 2+3+3+2+4+2+0+2 = 18 in example)
- Which one is better?

Example Heuristics: 8-puzzle



Consider the following heuristics:

- $h_1 =$ number of misplaced tiles (=7 in example)
- $h_2 =$ total Manhattan distance (i.e., no. of squares from desired location of each tile) ($=$ 2+3+3+2+4+2+0+2 $=$ 18 in example)
- Which one is better?
- Intuitively, h_2 seems better: it varies more across the state space, and its estimate is closer to the true cost.

Where Do Heuristics Come From?

- Prior knowledge about the problem
- Exact solution cost of a *relaxed* version of the problem
  - E.g. If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then h_1 gives the shortest solution
  - If the rules are relaxed so that a tile can move to *any adjacent square*, then h_2 gives the shortest solution
- Learning from prior experience - we will study such algorithms later.