CS4710: Artificial Intelligence Search

Many AI problems boil down to graph search. What are the primary techniques for searching graphs? What do we do when the graph is very large?

Quick Review of Graphs

Yes, I realize several of you have seen this many times!

Definition: Directed graph

- Directed Graph
  - A directed graph, or digraph, is a pair
  - G = (V, E)
  - where V is a set whose elements are called vertices, and
  - E is a set of ordered pairs of elements of V.

    - Vertices are often also called nodes.
    - Elements of E are called edges, or directed edges, or arcs.
    - For directed edge (v, w) in E, v is its tail and w its head;
    - (v, w) is represented in the diagrams as the arrow, v -> w.
    - In text we simple write vw.

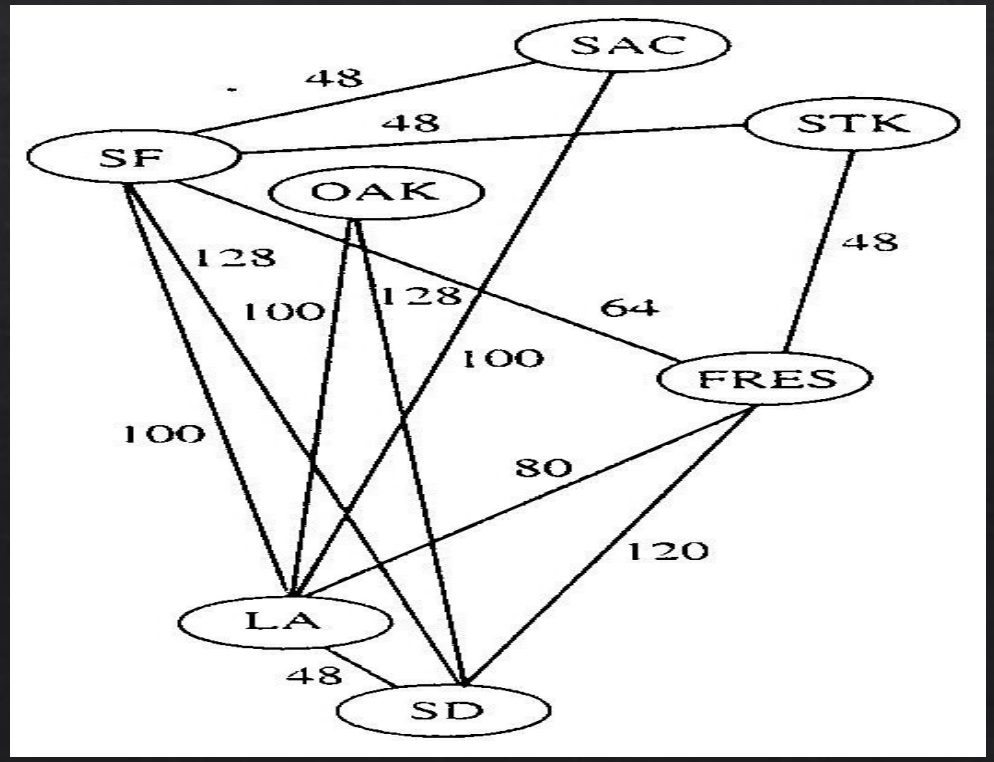
Definition: Undirected graph

- Undirected Graph
  - A undirected graph is a pair
  - G = (V, E)
  - where V is a set whose elements are called vertices, and
  - E is a set of *unordered* pairs of *distinct* elements of V.

    - Vertices are often also called nodes.
    - Elements of E are called edges, or undirected edges.
    - Each edge may be considered as a subset of V containing two elements,
    - {v, w} denotes an undirected edge
    - In diagrams this edge is the line v---w.
    - In text we simple write vw, or wv
    - vw is said to be *incident* upon the vertices v and w

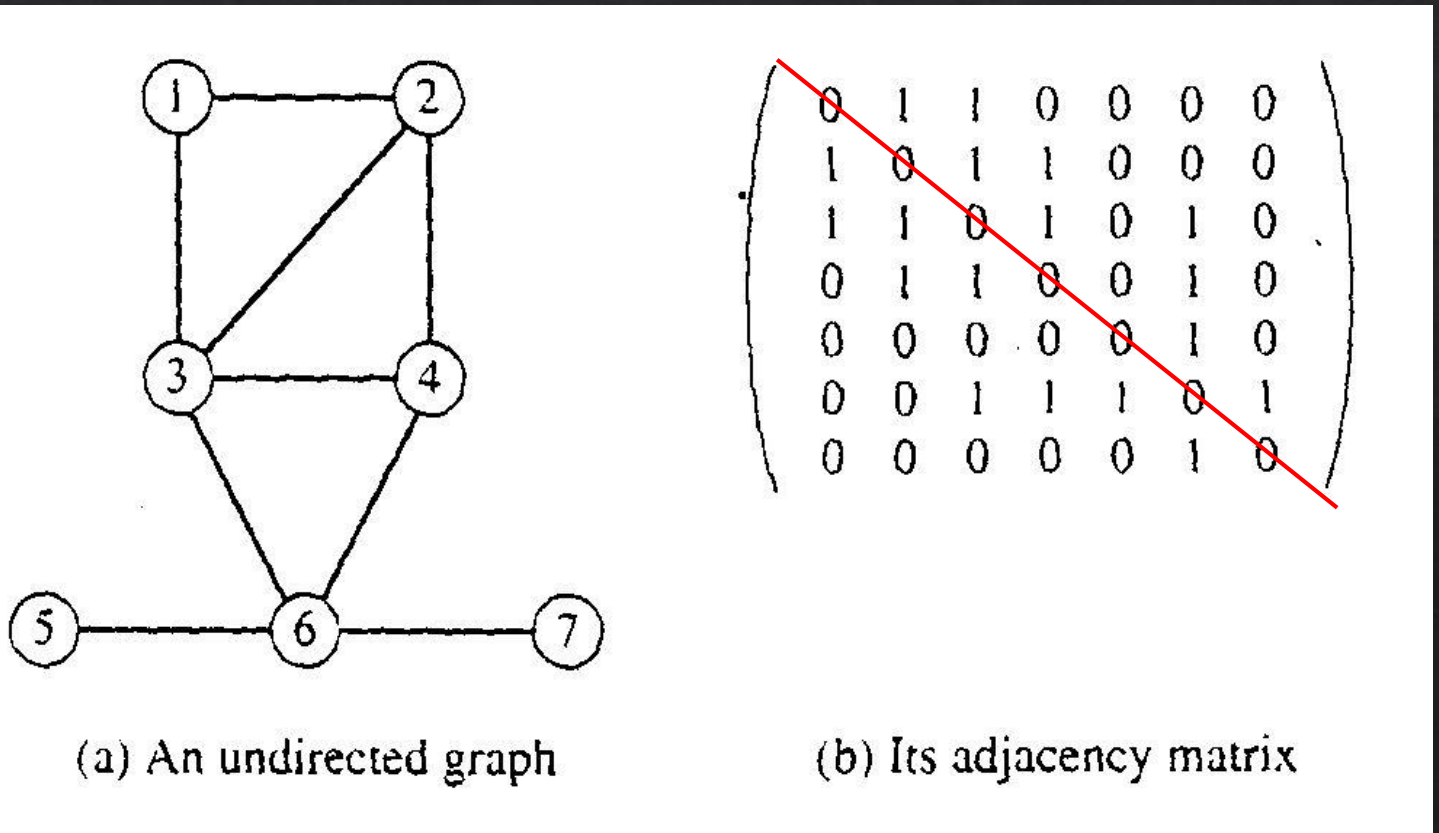
Definitions: Weighted Graph

- A weighted graph is a triple (V, E, W)
  - where (V, E) is a graph (directed or undirected) and
  - W is a function from E into R, the reals (integer or rationals).
  - For an edge e, W(e) is called the weight of e.

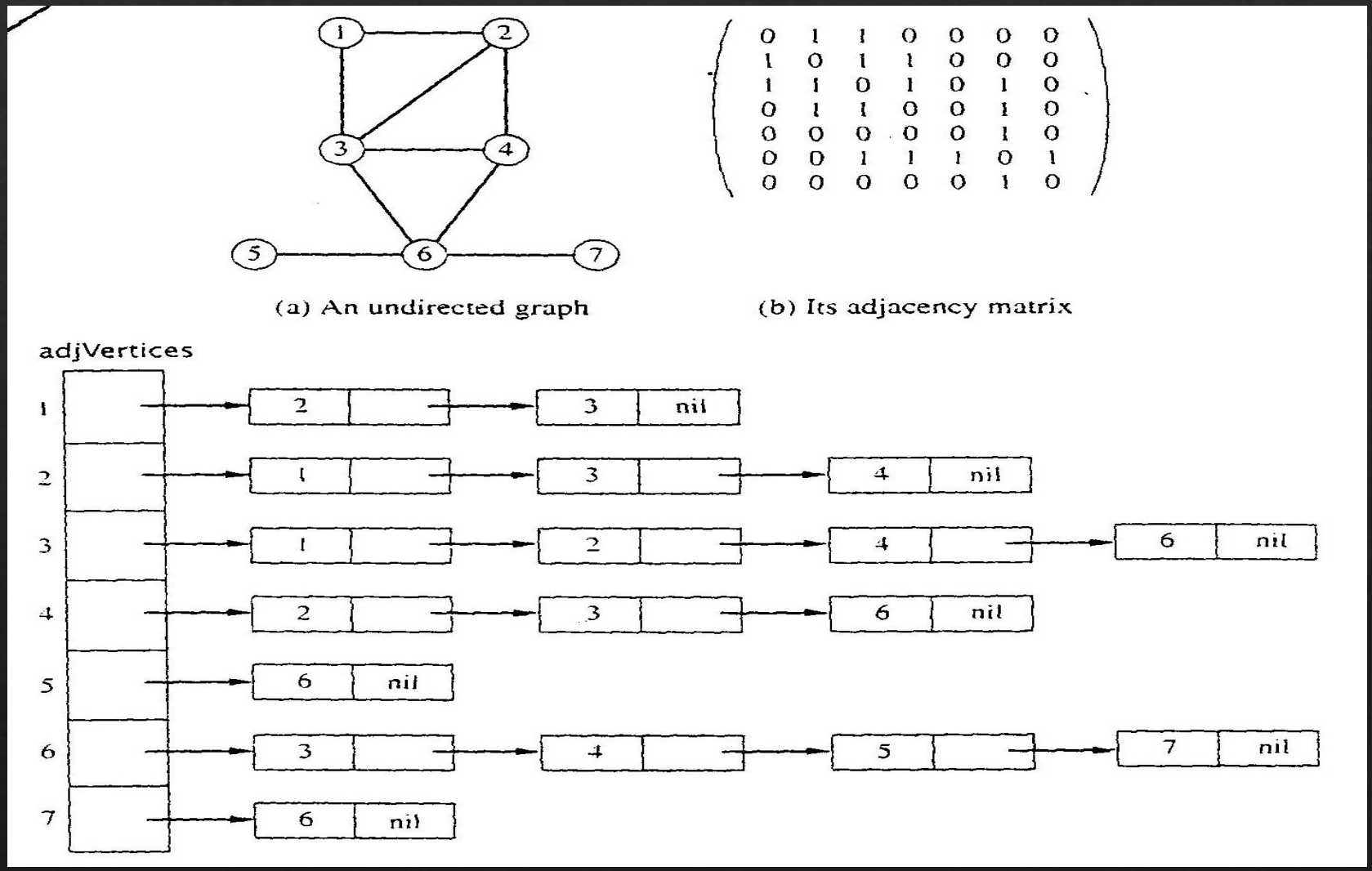


Graph Representations using Data Structures

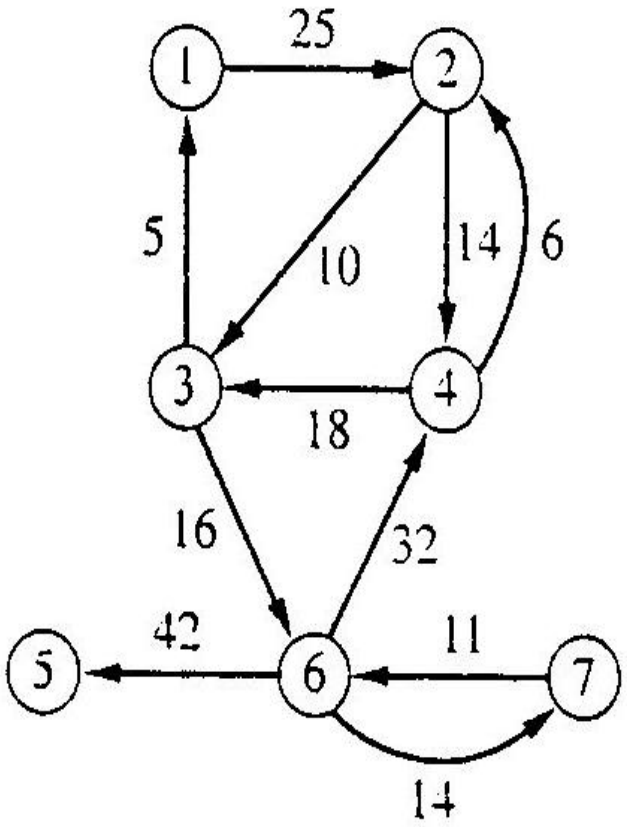
- Adjacency Matrix Representation
  - Let G = (V,E), n = |V|, m = |E|, V = {v1, v2, …, vn)
  - G can be represented by an n × n matrix



Array of Adjacency Lists Representation



Adjacency Matrix for weight digraph

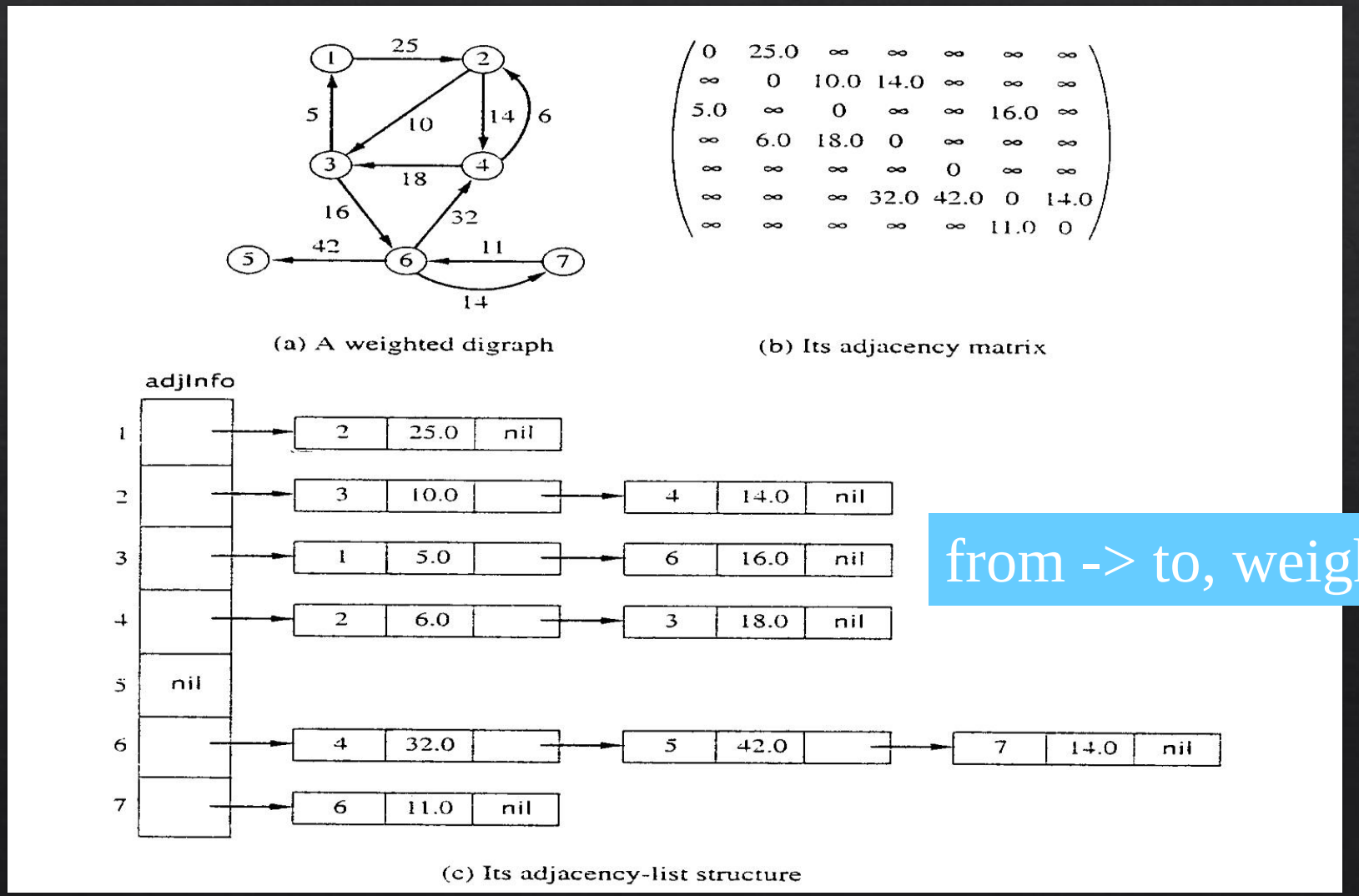


(a) A weighted digraph

$$\begin{pmatrix} 0 & 25.0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10.0 & 14.0 & \infty & \infty & \infty \\ 5.0 & \infty & 0 & \infty & \infty & 16.0 & \infty \\ \infty & 6.0 & 18.0 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32.0 & 42.0 & 0 & 14.0 \\ \infty & \infty & \infty & \infty & \infty & 11.0 & 0 \end{pmatrix}$$

(b) Its adjacency matrix

Array of Adjacency Lists Representation



from -> to, weight

Problems Solved With Search

AI Problems Solved by Search

- Path Planning
  - Robot or some agent is searching a path through an environment
- Game Playing
  - Searching through a set of states of a game to find a winning path for the AI
  - E.g., chess, checkers, etc.
- Puzzle Solving
  - Rubik's Cube, etc.
- Is this really "intelligence"…you decide.

Breadth-First Search

Traversal Strategies

- Note: traversal algorithms start at some vertex
  - Which? Trees have a root, but graphs don't.
  - Might matter, might not.

- Breadth-First and Depth-First are most basic strategies
  - Often aren't fit for AI problems where state space is VERY large
  - We'll review them first, then discuss slightly better approaches

BFS Strategy

- Breadth-first search: Strategy (for digraph)
  - choose a starting vertex, distance d = 0
  - vertices are visited in order of increasing distance from the starting vertex,
  - examine all edges leading from vertices (at distance d) to adjacent vertices (at distance d+1)
  - then, examine all edges leading from vertices at distance d+1 to distance d+2, and so on,
  - until no new vertex is discovered

BFS Strategy: More Details

- Maintain a Queue (Let's call it Q)

- Start at some node 's' (push 's' to Q and mark as visited)
- While Q not empty
  - Pop a node 'n' from queue
  - Process 'n' if necessary (depending on problem you are solving)
  - For each non-visited neighbor of 'n'
    - Mark neighbor as visited
    - Push neighbor onto Q
  - Repeat

Breadth-first search: Analysis

- For a digraph having V vertices and E edges
  - Each edge is processed once in the while loop for a cost of $\theta(E)$
  - Each vertex is put into the queue once and removed from the queue and processed once, for a cost $\theta(V)$
  - Extra space is used for color array and queue, there are $\theta(V)$
- From a *tree* (breadth-first spanning tree)
  - the path in the tree from start vertex to any vertex contains the *minimum* possible number of edges
- Not all vertices are necessarily reachable from a selected starting vertex

Breadth-first search: More Analysis

- In AI, graphs are often very large. So…

- Let ‘d’ be the shallowest depth of any goal node
- Let ‘b’ be the branching factor
  - *Average number of branches coming off of each node in graph*

- Runtime of BFS in terms of these factors is:
  - Time: O(b^d) why?
  - Space: O(b^d) why?

- Notice that b^d is bounded by E Why?

Problem Solving Time!

Depth-First Search

DFS: the Strategy in Words

- Depth-first search: Strategy
  - Go as deep as can visiting un-visited nodes
    - Choose any un-visited vertex when you have a choice
  - When stuck at a dead-end, backtrack as little as possible
    - Back up to where you could go to another unvisited vertex
  - Then continue to go on from that point
  - Eventually you'll return to where you started
    - Reach all vertices? Maybe, maybe not

Observations about the DFS Strategy

- Note: we must keep track of what nodes we've visited
- DFS traverses a subset of E (the set of edges)
  - Creates a tree, rooted at the starting point: the Depth-first Search Tree (DFS tree)
  - Each node in the DFS tree has a distance from the start. (We often don't care about this, but we could.)
- At any point, all nodes are either:
  - Un-discovered
  - Finished (you backed up from it), or
  - Discovered (I.e. visited) but not finished
    - On the path from the current node back to the root
    - We might back up to it

DFS Strategy 1: Use a stack

- Maintain a Stack (Let's call it S)

- Start at some node 's' (push 's' to S and mark as visited)
- While S not empty
  - Pop a node 'n' from S
  - Process 'n' if necessary (depending on problem you are solving)
  - For each non-visited neighbor of 'n'
    - Mark neighbor as visited
    - Push neighbor onto S
  - Repeat

- Sound familiar? Same as BFS but uses stack instead of queue!
- Or we can implement recursively (see next slide)

DFS Strategy 2: Recursion

```
def dfs(graph, start):
    visited = {}
    dfs_recurse(graph, start, visited)

def dfs_recurse(graph, curnode, visited):
    visited[curnode] = True
    alist = graph.get_adjlist(curnode)
    for v in alist:
        if v not in visited:
            print "  dfs traversing edge:", curnode, v
            dfs_recurse(graph, v, visited)

    return
```

Time Complexity of DFS

- For a digraph having V vertices and E edges
  - Each edge is processed once in the while loop of dfs_recurse() for a cost of θ(E)
    - Think about adjacency list data structure.
    - Traverse each list exactly once. (Never back up)
    - There are a total of 2*E nodes in all the lists
  - Must visit every node potentially
  - Thus over all time-complexity is θ(V+E)
    - Remember: this means the larger of the two values
    - Note: This is considered “linear” for graphs since there are two size parameters for graphs.

Time Complexity of DFS

- In terms of d and b
- Time: O(b^d) Why?
- Space: O(d)

Notice ‘d’ here is actually the maximum depth the search reaches. In a moment we will bound this more tightly.

Why? Depends on implementation, but when can we get ‘d’ space?

Pros and Cons

- BFS
  - Finds path with lowest number of edges
  - Uses a lot of memory on large well connected graphs. How much?
  - Same time complexity as DFS
- DFS
  - Memory efficient
  - Will often find ‘some’ path very quickly
  - Doesn’t find path with lowest number of edges

DFS w/ Iterative Deepening

Motivation

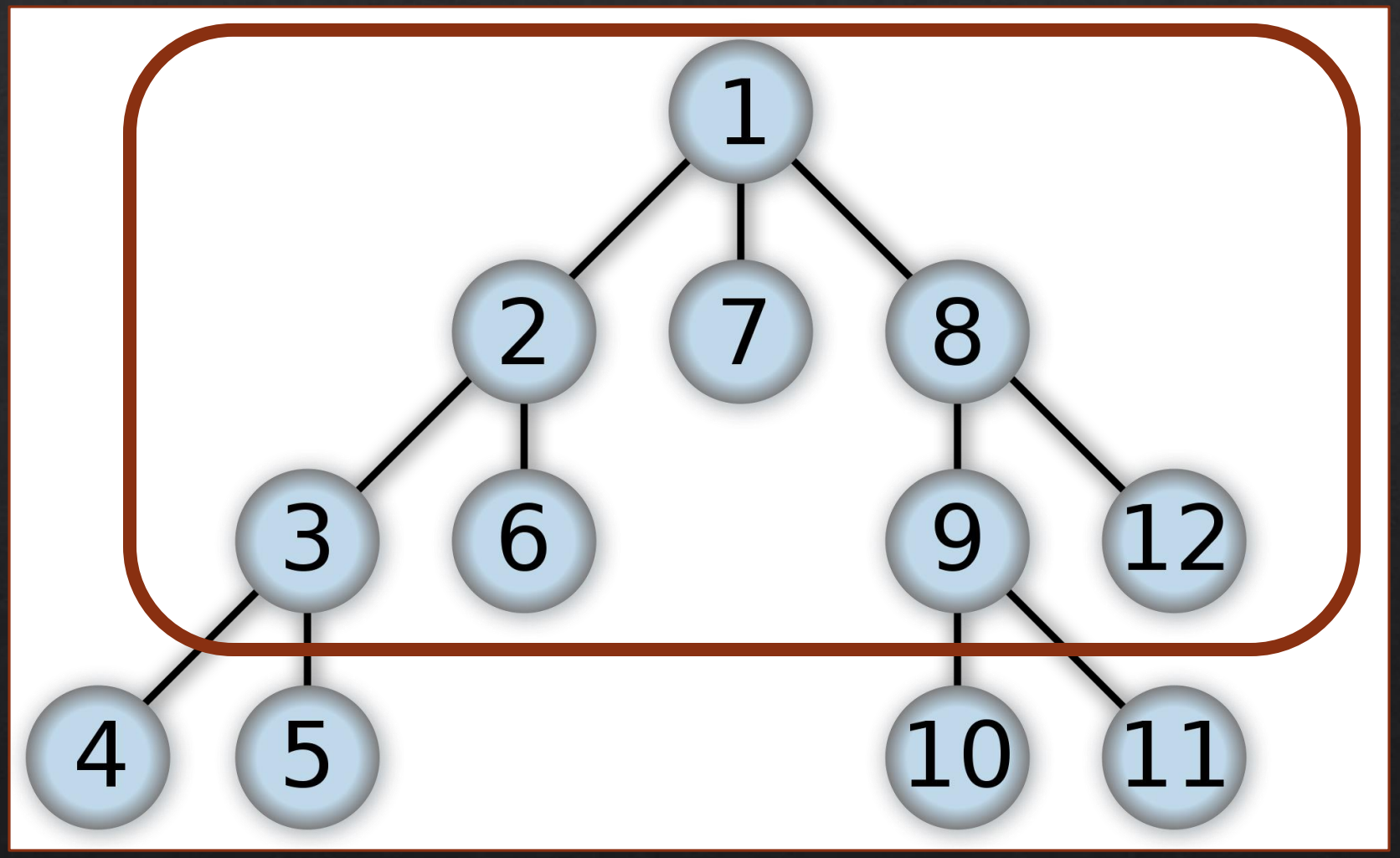
- When search spaces are HUGE, we want:
  - The memory efficiency of DFS
  - The optimality of BFS

- Can we combine the two approaches somehow?

Depth-Limited Search

- IDEA!
  - Run a depth-first search with extra parameter ‘d’
  - ‘d’ is the maximum depth the DFS is willing to search
    - Depth is distance from the start node
  - Once you see a node of depth ‘d’, ignore it’s neighbors and back up as you would normally

Depth-Limited Search



- D = 2

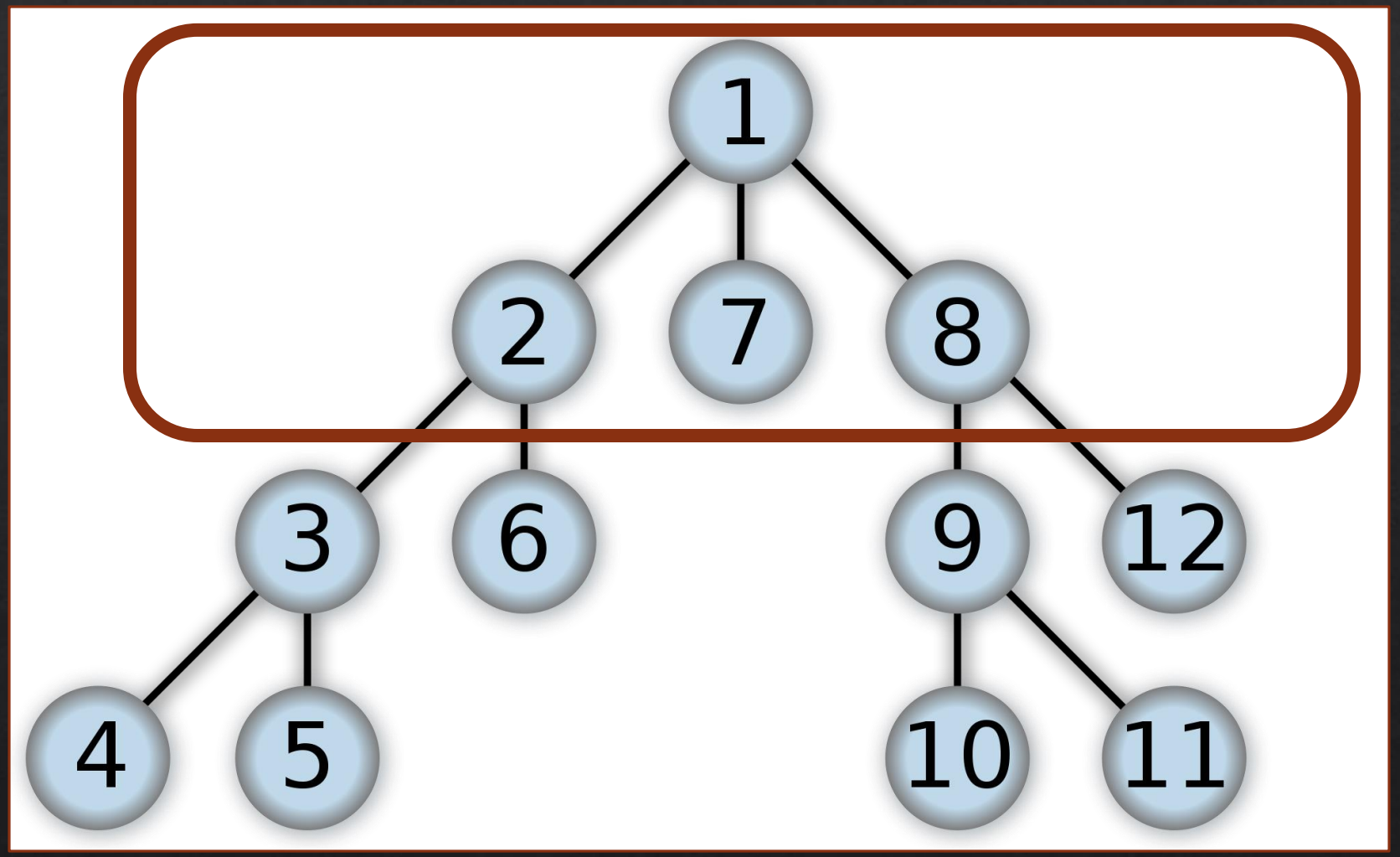
Depth-Limited Search

- Problems with this?
- Will this ever find a goal node if that node is beyond depth ‘d’?
- How do we deal with these problems?

What is Iterative Deepening?

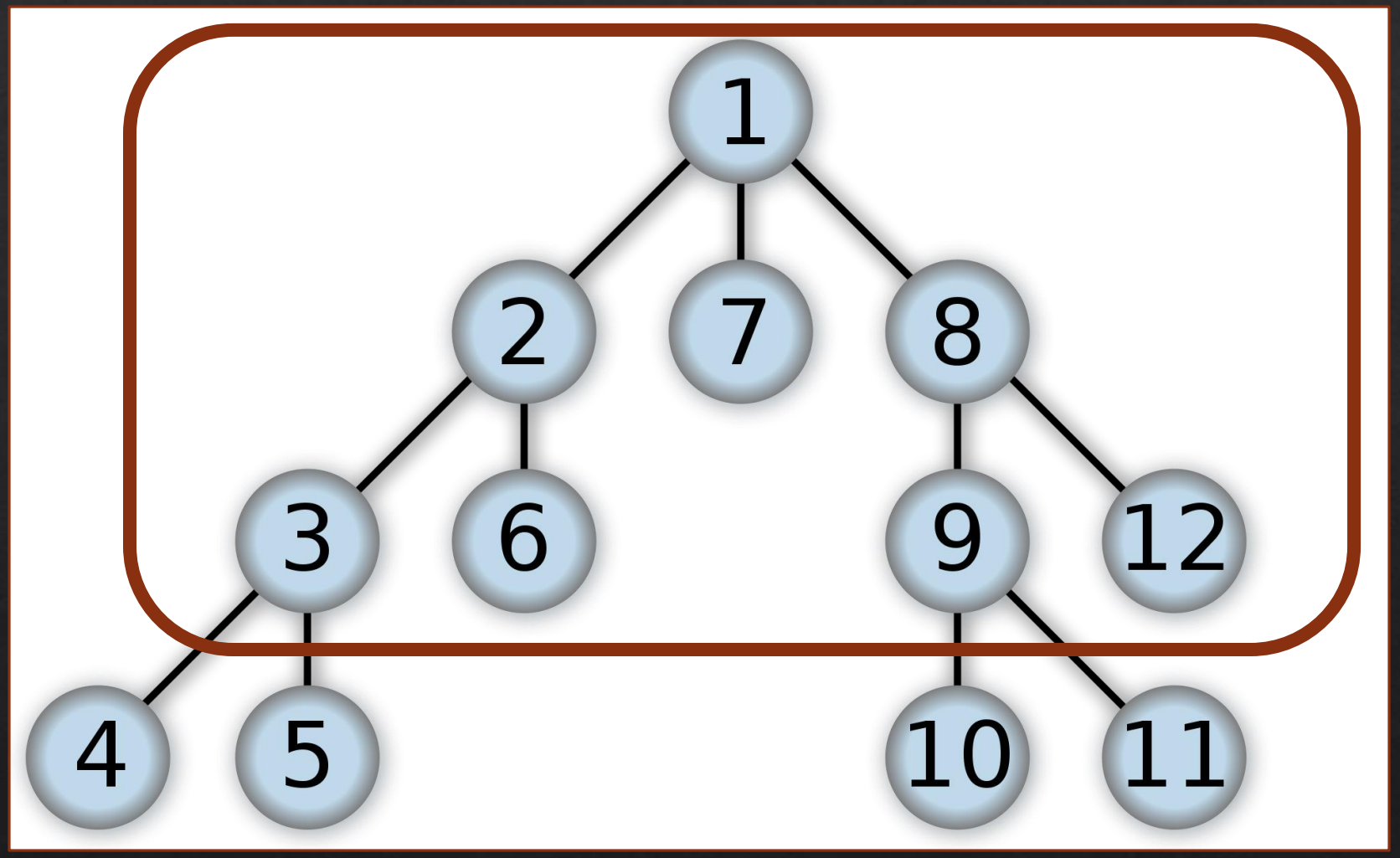
- IDEA!
  - Run Depth-Limited search over and over with increasing ‘d’ value
  - Analysis? What are the pros and cons?

Iterative Deepening Example



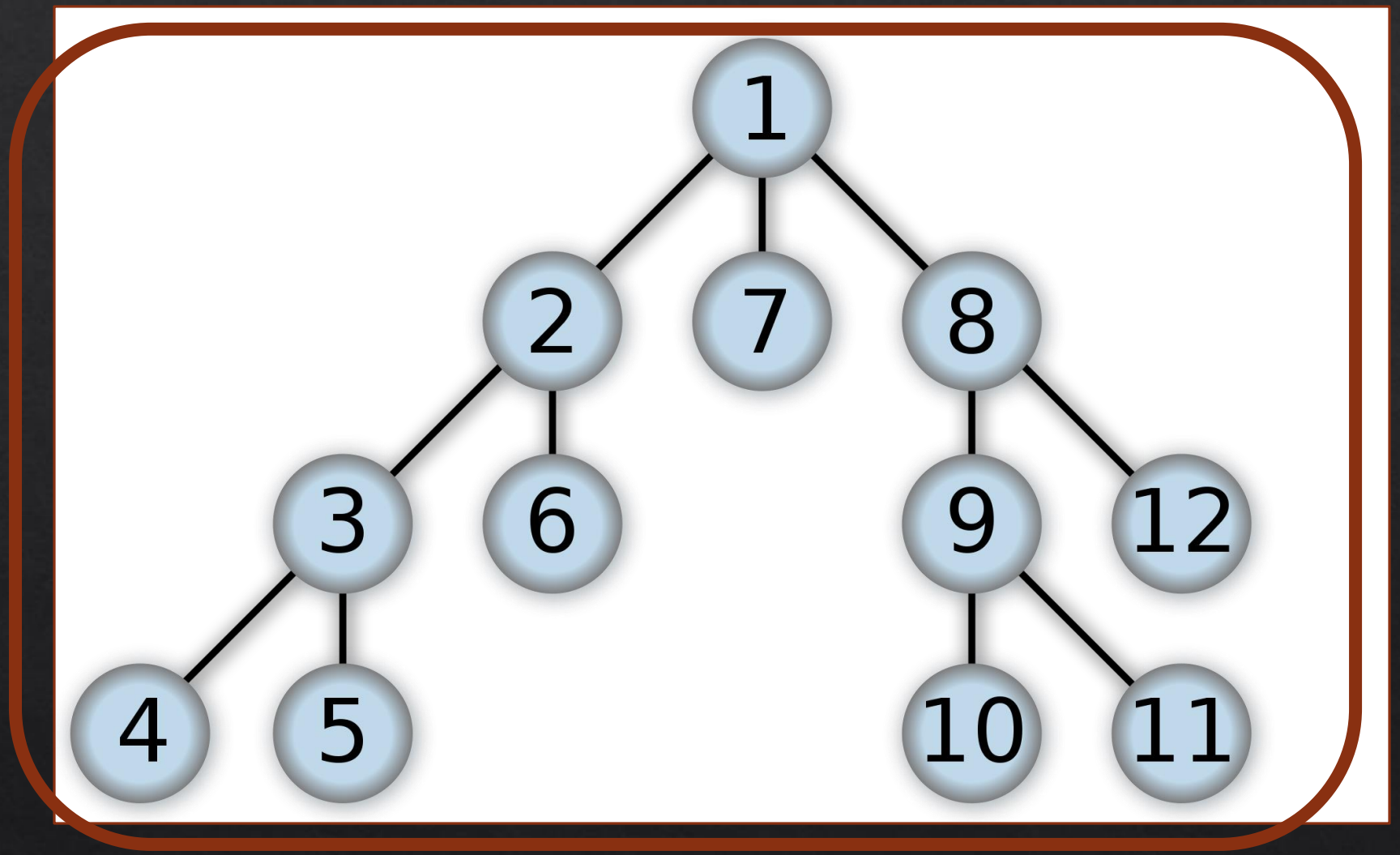
- D = 1

Iterative Deepening Example



- D = 2

Iterative Deepening Example



- D = 3

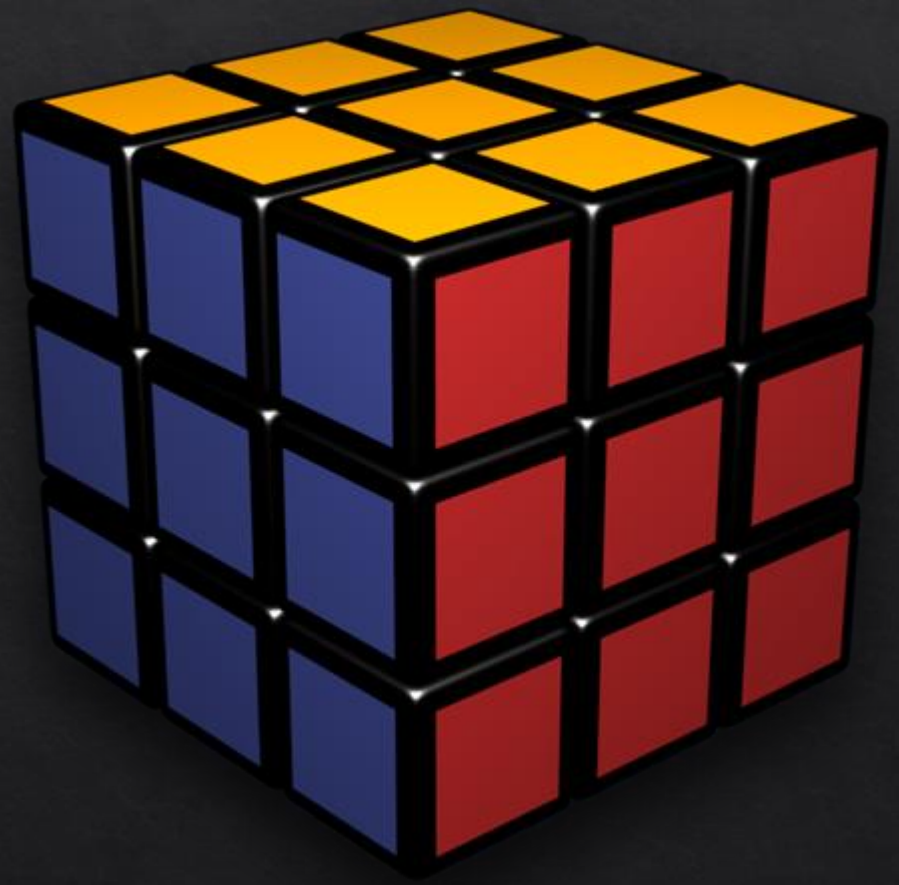
Iterative Deepening Pseudo-Code

```
IDDFS(root, goal)
{
  for(i=1, i<10, i++)
  {
    DLS(root, goal, limit=i)
  }
}
```

Iterative Deepening: Analysis

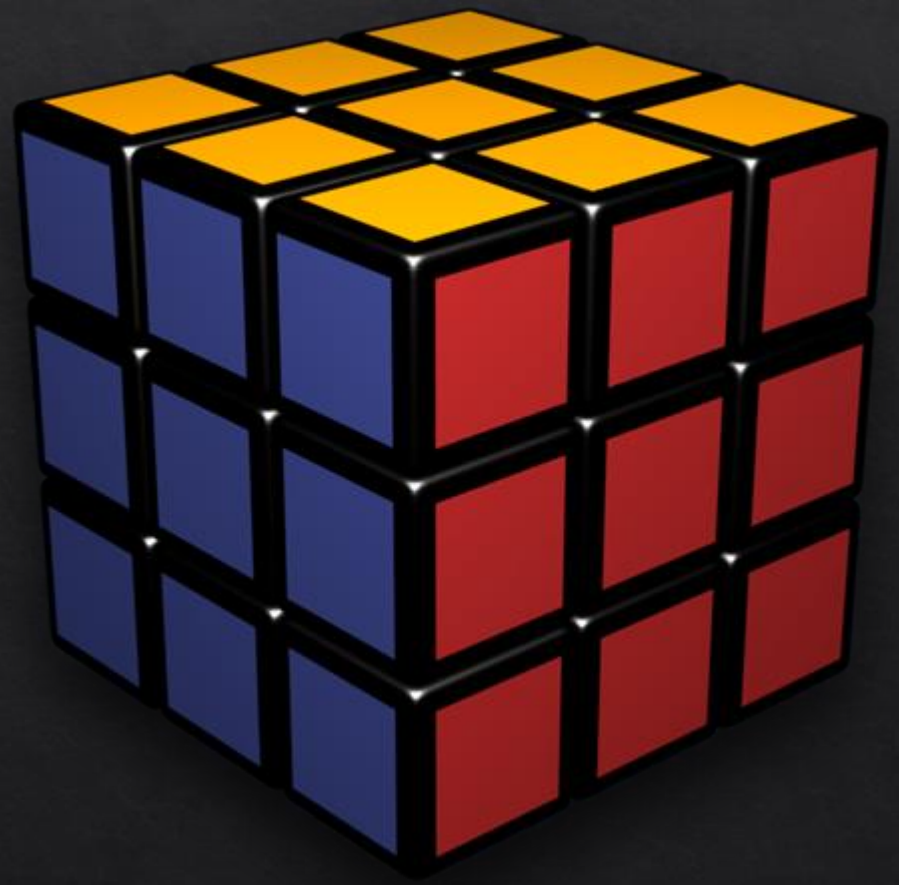
- Time Complexity:
  - DFS multiple times so O(b^d * d) = O(b^d)
    - Same as DFS or BFS
- Space Complexity:
  - Same as DFS, so O(d)
- But this seems so wasteful? Why is this a good approach?
- Some form of Iterative deepening often used when goal depth is unknown and state space is very large

Iterative Deepening: Example



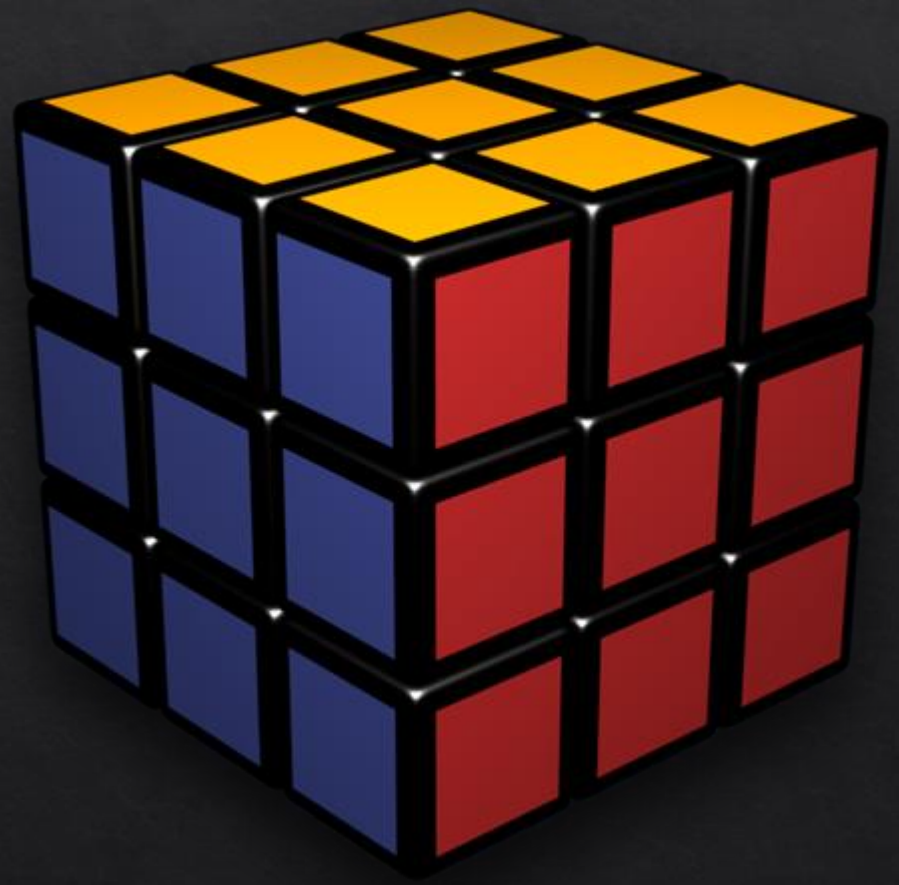
- AI that solves a rubik's cube
- How will we solve this problem?
- Want to find smallest number of moves that lead to the solved state

Iterative Deepening: Example



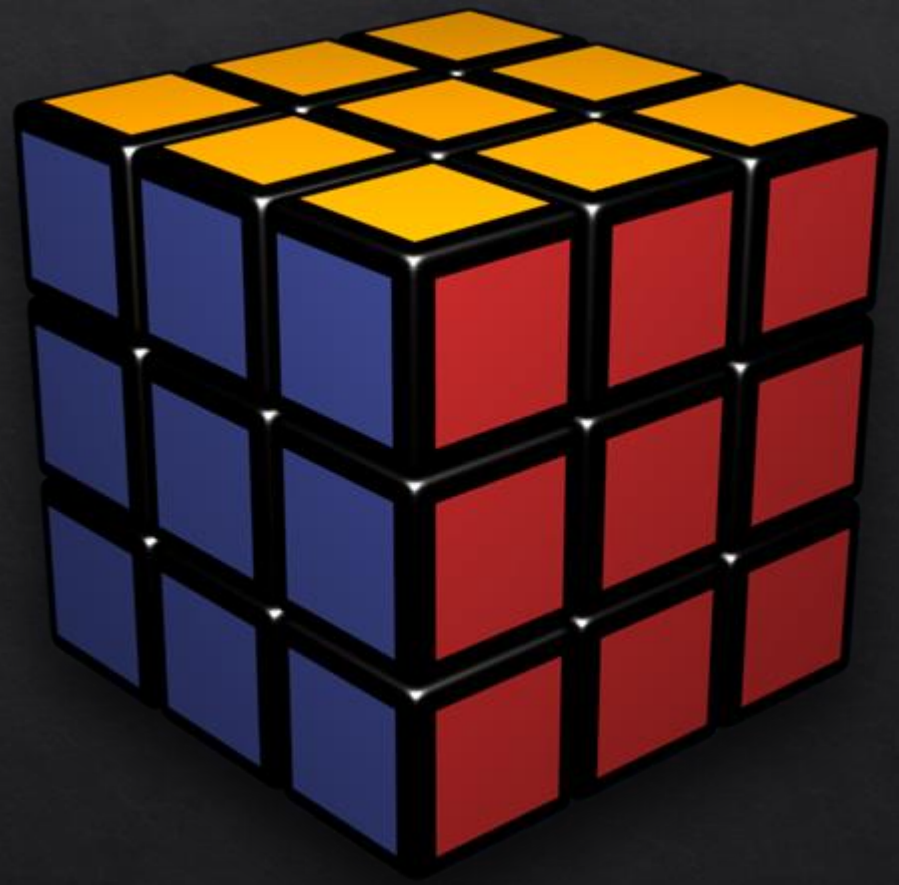
- About how many states does the cube have? Let's estimate
- 8 corner blocks, 3 orientations each
  - 8! * 3^8 permutations
  - Some are not possible, but that is ok
- 12 edge pieces, 2 orientations each
  - 12! * 2^12 permutations
  - Again, some of these not possible but ok
- Total: ~ 4.325*10^19

Iterative Deepening: Example



- Breadth-First Search!
- Analysis:
  - b >= 12
  - d = ? LB Guess: 20
- Space / Time
  - 12^20 = 3.83*10^21
- Wait what? Many of these repeated, but means we could have whole search space in memory at once!!
- Too much memory at once!

Iterative Deepening: Example



- Iterative Deepening!
  - Will still find shortest solution
- Analysis:
  - b >= 12
  - d = ? LB Guess: 20
- Time: Still might search whole space, no way around this
- Space: d = 20
- Probably larger in practice but still very feasible

Heuristic Search

Understanding Heuristic Searches

- All of the following algorithms use two lists
  - The ***open*** list
  - The ***closed*** list
- Open list keeps track of promising nodes
- When a node is examined from open list
  - Taken off open list and checked to see whether it has reached the goal
- If it has not reached the goal
  - Used to create additional nodes
  - Then placed on the closed list
- You already know some of these algorithms
  - BFS, Dijkstra, etc.

Overall Structure of the Algorithms

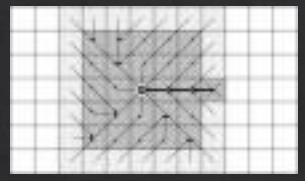
1. Create start point node – push onto open list
2. While open list is not empty
   A. Pop node from open list (call it currentNode)

   B. If currentNode corresponds to goal, break from step 2

   C. Create new nodes (successors nodes) for cells around currentNode and push them onto open list

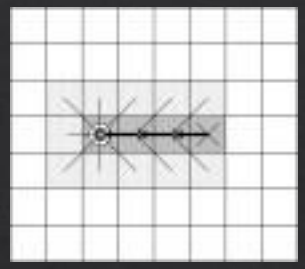
   D. Put currentNode onto closed list

Breadth-First | Characteristics of



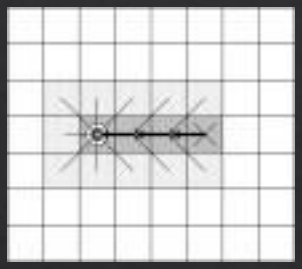
- Finds a path from the start to the goal by examining the search space ply-by-ply

- Exhaustive search
  - Systematic, but not clever
- Consumes substantial amount of CPU and memory
- Guarantees to find paths that have fewest number of nodes in them
  - Not necessarily the shortest distance!
- ***Complete*** algorithm



Best-First

- Uses problem specific knowledge to speed up the search process
- Head straight for the goal
- Computes the distance of every node to the goal
  - Uses the distance (or heuristic cost) as a priority value to determine the next node that should be brought out of the open list
  - …but what is a heuristic? See next slide!

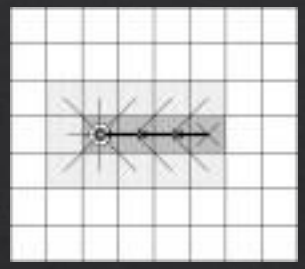


Heuristics

- A *Heuristic* is a guess. For search problems, a *heuristic* is an *estimate* of how much the cost to the goal node will be.
  - Notice that each node in the graph will have its own heuristic value
  - *E.g., h(n3) returns a guess as to the cost of the path from node n3 to the goal node*

- If your heuristic guess is perfect, then you don't need to search, because you know the answer!

- So…as an engineer you can try using different methods for guessing this value.
  - In general, the closer your heuristic guess, the more efficient your search can be.
  - So worth it to work hard to get the heuristic as close to actual cost as possible.



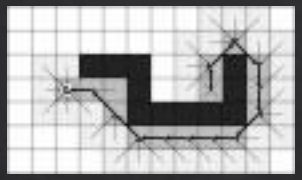
Heuristics

- A *Heuristic* is a guess. For search problems, a *heuristic* is an *estimate* of how much the cost to the goal node will be.
- Can you answer these questions:
  - What happens if your heuristic guess is exactly right every time?
  - What happens if your heuristic is the same for every node? (e.g., h(x) = 5 for all x)
  - What happens if your heuristic is erroneous? (e.g., h(x) > h(y) but x actually closer to goal than y)

About Heuristics

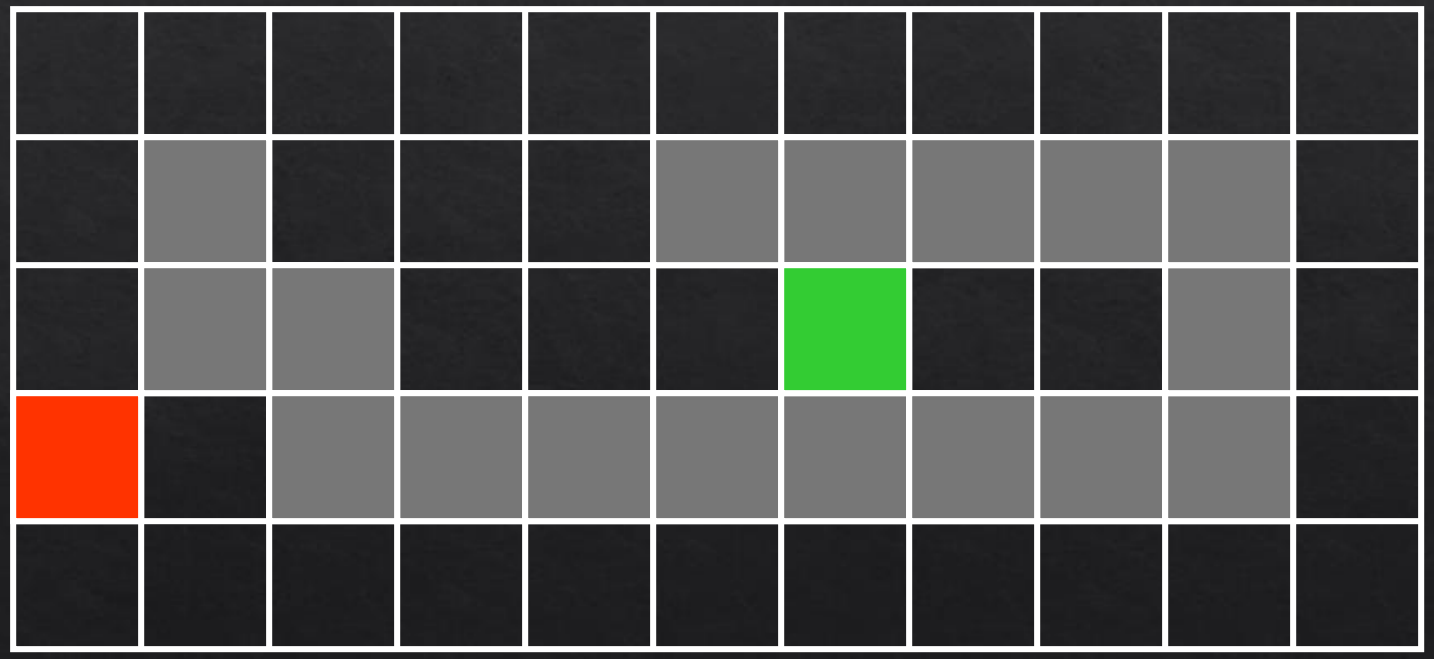
- Heuristics are intended to orient the search along promising paths
- The time spent computing heuristics must be recovered by a better search
- After all, a heuristic function could consist of solving the problem; then it would perfectly guide the search
- Deciding which node to expand is sometimes called meta-reasoning
- Heuristics may not always look like numbers and may involve large amount of knowledge

Best-First | Characteristics of



- Same as BFS, Dijkstra, etc. except the priority value on the priority queue is different.
- Use h(x) as the priority value
- Situation where Best-First finds a suboptimal path
- Heuristic search
- Uses fewer resources than Breadth-First
- Tends to find good paths
  - No guarantee to find most optimal path
  - Why?
- ***Complete*** algorithm

Example: Robot Navigation



*Red is starting point, green the goal node

Robot Navigation: Best-First Search

f(N) = h(N), with h(N) = Manhattan distance to the goal

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = h(N), with h(N) = Manhattan distance to the goal

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

What happened???

Greedy Search

- f(N) = h(N) → greedy best-first
- Is it complete?

  If we eliminate endless loops, yes
- Is it optimal?
  - No! Need something better

A* Pathfinding

More informed search

- We kept looking at nodes closer and closer to the goal, but were accumulating costs as we got further from the initial state
- Our goal is not to minimize the distance from the current head of our path to the goal, we want to minimize the *overall* length of the path to the goal!
- Let g(N) be the cost of the best path found so far between the initial node and N
- f(N) = g(N) + h(N)

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1+10	0+11	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

A* Search

- Evaluation function:

  $$f(N) = g(N) + h(N)$$

  where:

  - $g(N)$ is the cost of the best path found so far to N
  - $h(N)$ is an admissible heuristic

- Then, best-first search with this evaluation function is called A* search

- Important AI algorithm developed by Fikes and Nilsson in early 70s. Originally used in Shakey robot.

Admissible heuristic

- Let $h^*(N)$ be the **true** cost of the optimal path from N to a goal node
- Heuristic $h(N)$ is admissible if:

$$0 \leq h(N) \leq h^*(N)$$

- An admissible heuristic is always *optimistic*

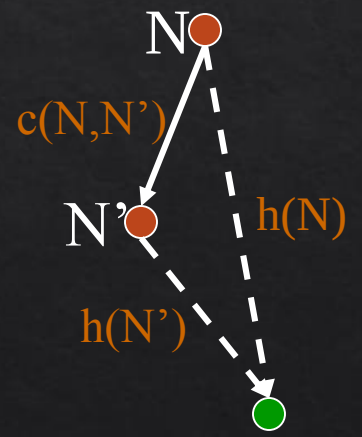
Consistent Heuristic

- The admissible heuristic h is consistent (or satisfies the monotone restriction) if for every node N and every successor N' of N:

$h(N) \leq c(N,N') + h(N')$

$h(Goal) = 0$

(triangular inequality)

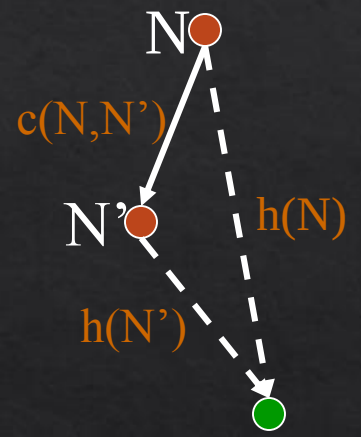


- *h() is consistent -> h() is admissible*
  - *Converse not necessarily true. Can you prove this?*

Claims

- If h is consistent, then the function f along any path is non-decreasing:

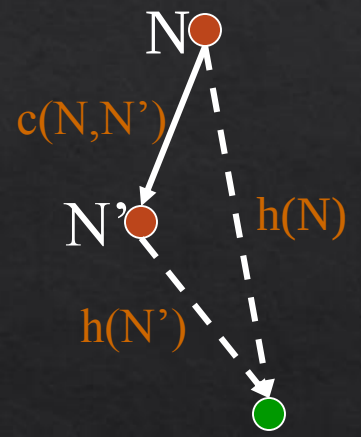
$$f(N) = g(N) + h(N)$$
$$f(N') = g(N) + c(N,N') + h(N')$$



Claims

- If h is consistent, then the function f along any path is non-decreasing:

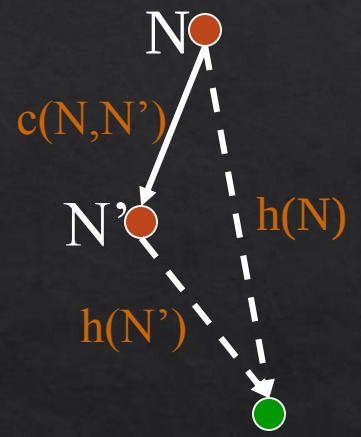
$$f(N) = g(N) + h(N)$$
$$f(N') = g(N) + c(N,N') + h(N')$$
$$h(N) \leq c(N,N') + h(N')$$
$$f(N) \leq f(N')$$



Claims

- If h is consistent, then the function f along any path is non-decreasing:

$$f(N) = g(N) + h(N)$$
$$f(N') = g(N) + c(N,N') + h(N')$$
$$h(N) \leq c(N,N') + h(N')$$
$$f(N) \leq f(N')$$



- If h is consistent, then whenever A* expands a node it has already found an optimal path to the state associated with this node

Back to the Puzzle!

Optimality of A*

- Theorem: **If** h(n) is admissible, then A* is optimal.

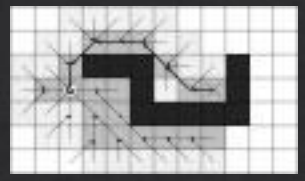
- Corollary: If h(n) is not admissible, then A* is not necessarily optimal
  - *Can you find a counter-example to prove this claim?*

- *Work on this in pairs. Give me:*
  - *1. A counter-example to the corollary above*
  - *2. A intuitive proof of the theorem above*

Heuristic Accuracy

- $h(N) = 0$ for all nodes is admissible and consistent. Hence, breadth-first and uniform-cost are particular A* !!!
- Let h_1 and h_2 be two admissible and consistent heuristics such that for all nodes N: $h_1(N) \leq h_2(N)$.
- Then, every node expanded by A* using h_2 is also expanded by A* using h_1.
- h_2 is more informed than h_1

A* | Summary



- Uses both heuristic cost and given cost to order the open list
- Final Cost = Given Cost + (Heuristic Cost * Heuristic Weight)
- Avoids Best-First trap!

- Heuristic search
- On average, uses fewer resources than Dijkstra and Breadth-First
- *Admissible* heuristic
  - Can never overestimate the cost
- guarantees it will find the most optimal path
- ***Complete*** algorithm

Thinking about these algorithms

- Each algorithm uses the following function to determine the "priority" of each node:

- P(x) = w1*f(x) + w2*g(x)
  - w1 & w2 are binary weights (1 or 0)
  - f(x) is distance traveled so far to node x
  - g(x) is heuristic guess as to distance from this node to goal.

- Dijkstra's Algo:
  - w1 = 1, w2 = 0
  - f(x) is distance to node x
  - No Heuristic included

- Best-first search:
  - w1 = 0, w2 = 1
  - g(x) is heuristic guess of distance to goal

- A*
  - w1 =1, w2 = 1
  - f(x) distance to node x
  - g(x) heuristic guess of distance to goal
  - g(x) <= actual distance to goal