Artificial Intelligence

CS 165A

Apr 27, 2023

Instructor: Prof. Yu-Xiang Wang

Today

→ Problem solving by search (continue)

→ Search algorithms

Coding Project 1 is due midnight today

- Submit your code today!
- Leaderboard and report will be open until next Thursday.
- Declare your collaboration (help you've received)
- Project 2 is released (instruction and code are on the course website)

Recap: Problem Formulation and Search

- Problem formulation
  - State-space description < **{S}, S_0, {S_G}, {O}, {g}** >
    - **S**: Possible states
    - $\mathbf{S_0}$: Initial state of the agent
    - $\mathbf{S_G}$: Goal state(s)
      - Or equivalently, a goal test **G(S)**
    - **O**: Operators O: {S} => {S}
      - Describes the possible actions of the agent
    - **g**: Path cost function, assigns a cost to a path/action
- At any given time, which possible action $\boldsymbol{O_i}$ is best?
  - Depends on the goal, the path cost function, the future sequence of actions….
- Agent's strategy: Formulate, Search, and Execute
  - This is *offline* problem solving

This lecture

- More examples on "Problem Solving by Search"

- Search algorithms
  - BFS / DFS
  - Depth-limited search
  - Iterative Deepening search
  - Bidirectional search
  - Uniform cost search

- Tree search vs Graph search

- Informed Search
  - A*-Search

Example: Missionaries and Cannibals (3 min discussion)

Goal: (0, 0, False)

Problem: Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place

(3, 3, ~~0, 0~~, True) $\xrightarrow{+1}$ (2, 2, False)

(3, 3, ~~0, 0~~, True) $\rightarrow$ (3, 1, False)

(3, 3, ~~0, 0~~, True) $\rightarrow$ (1, 3, False) $\rightarrow$ (2, 3, True)

- States, operators, goal test, path cost?

# of C | # of M

3	3

x y

Boat on left

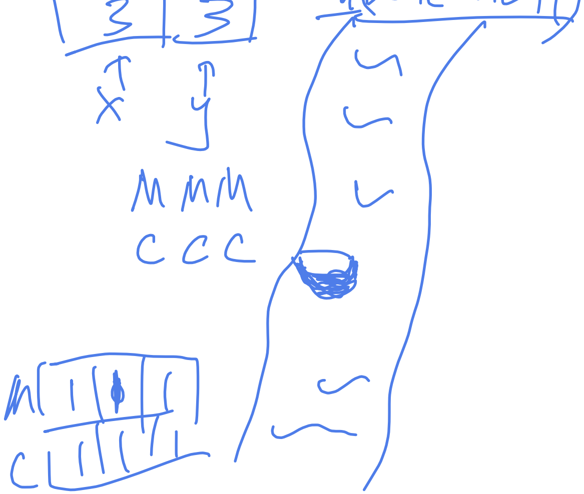
# of C | # of M

0	0

3−x 3−y

M M M

C C C



# of M − # of C on the left

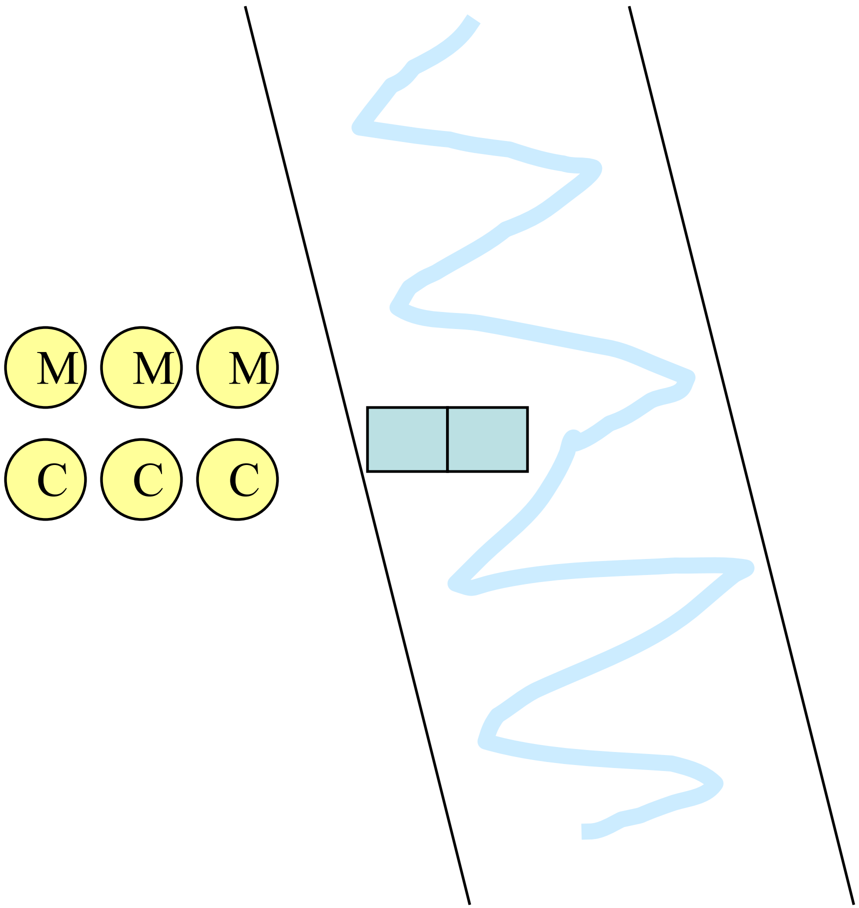
# of M − # of C on the right

$$\begin{cases} x - y = a_1 \\ (3-x) - (3-y) = a_2 \end{cases}$$

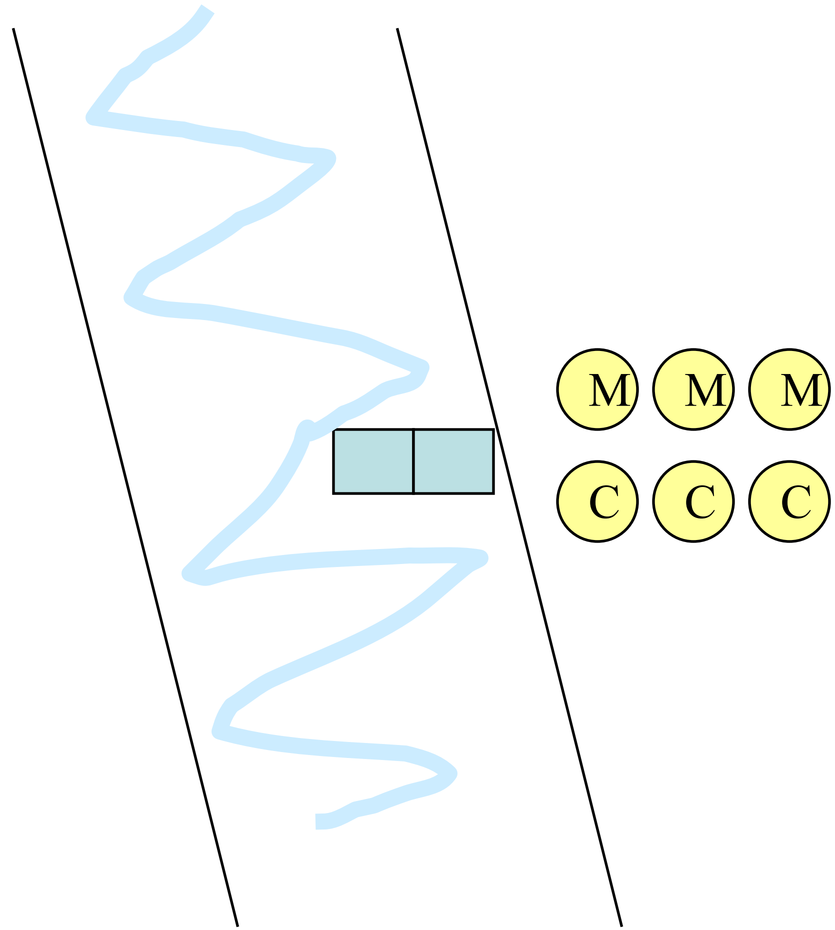
$-x + y = a_2$

M&C (cont.)

- Initial state

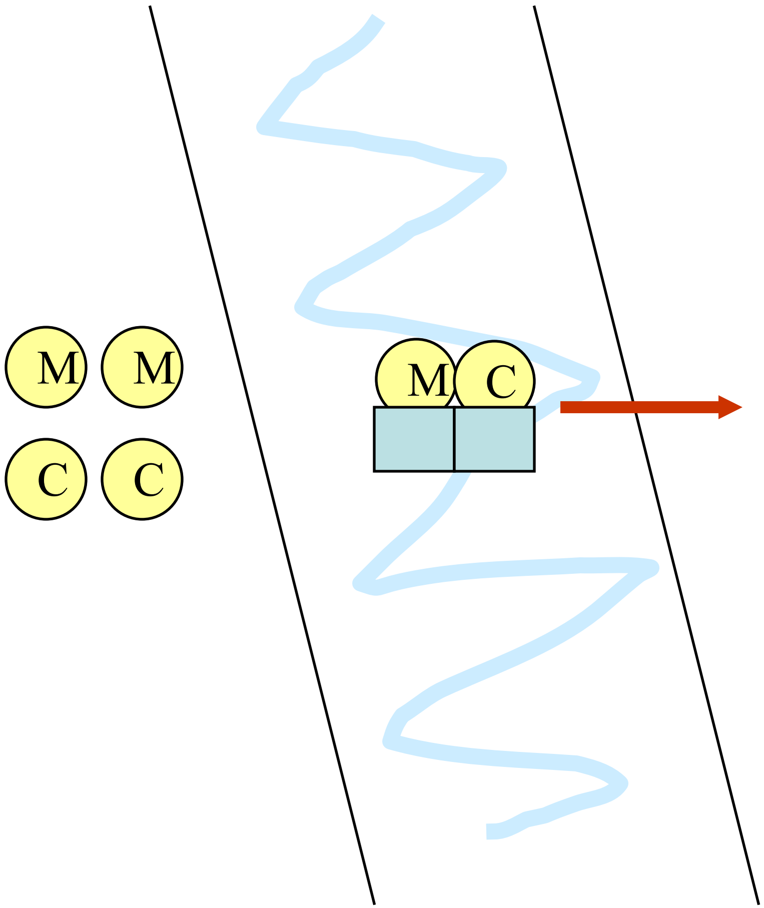


- Goal state



(3 3 1) $(M_L\ C_L\ B_L)$ (0 0 0)

M&C (cont.)



(2 2 0)

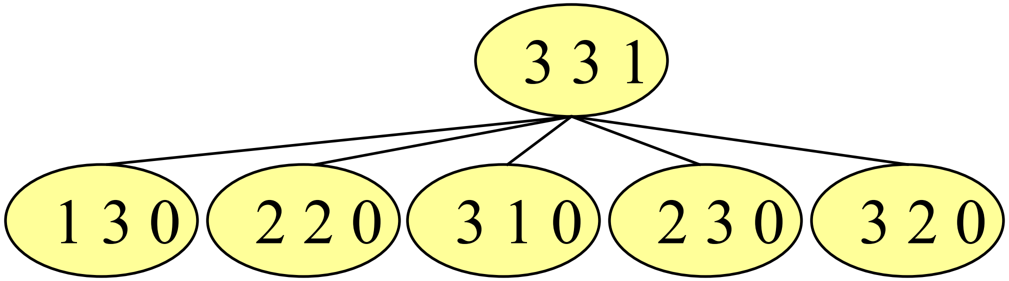
M&C (cont.)

- Problem description $<\{\mathbf{S}\}, \mathbf{S_0}, \{\mathbf{S_{G_j}}\}, \{\mathbf{O_i}\}, \{\mathbf{g_i}\}>$
- $\{\mathbf{S}\}$: { ({0,1,2,3} {0,1,2,3} {0,1}) }
- $\mathbf{S_0}$: (3 3 1)
- $\mathbf{S_G}$: (0 0 0)
- $\mathbf{g} = 1$
- $\{\mathbf{O}\}$: { (x y b) → (x' y' b') }
- Safe state: (x y b) is safe iff
  - $x > 0$ implies $x \geq y$ and
    $x < 3$ implies $y \geq x$
  - Can be restated as
    $(x = 1 \text{ or } x = 2)$ implies $(x = y)$

Operators:

(x y 1) → (x-2 y 0)
(x y 1) → (x-1 y-1 0)
(x y 1) → (x y-2 0)
(x y 1) → (x-1 y 0)
(x y 1) → (x y-1 0)
(x y 0) → (x+2 y 1)
(x y 0) → (x+1 y+1 1)
(x y 0) → (x y+2 1)
(x y 0) → (x+1 y 1)
(x y 0) → (x y+1 1)

M&C (cont.)



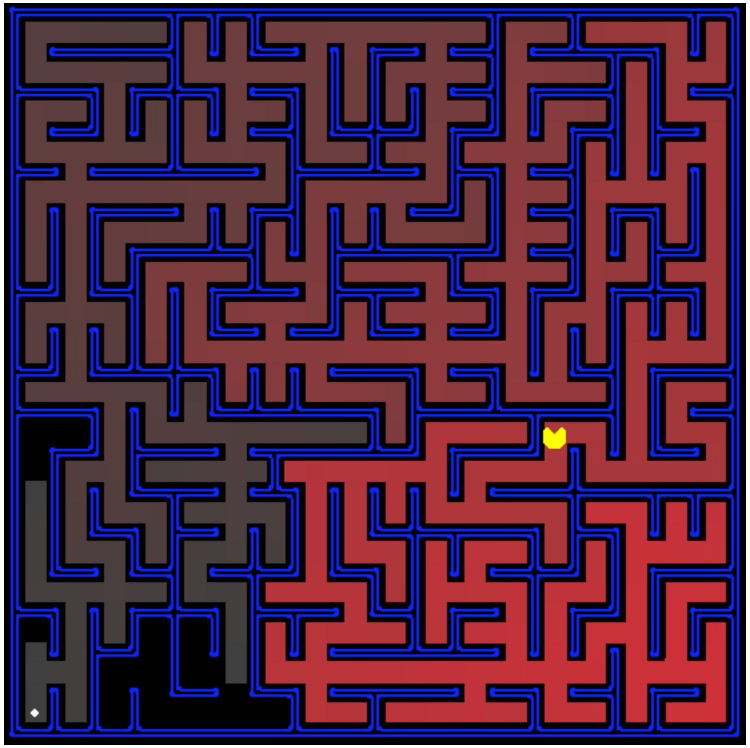
- 11 steps
- 5^{11} = 48 million states to explore

One solution path:

(3 3 1)
(2 2 0)
(3 2 1)
(3 0 0)
(3 1 1)
(1 1 0)
(2 2 1)
(0 2 0)
(0 3 1)
(0 1 0)
(0 2 1)
(0 0 0)

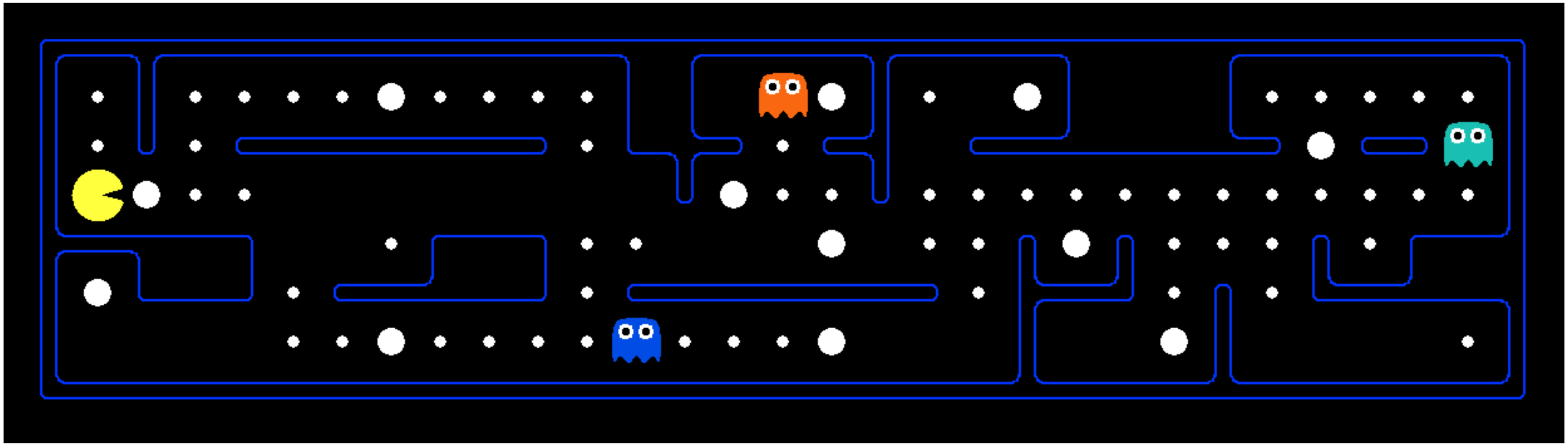
Example: PACMAN

- The goal of a simplified PACMAN is to get to the pellet as quick as possible.
  - For a grid of size 30*30. Everything static.
  - What is a reasonable representation of the State, Operators, Goal test and Path cost?



Example: PACMAN with static ghosts

- The goal is to eat all pellets as quickly as possible while staying alive. Eating the “Power pellet” will allow the pacman to eat the ghost.



- Think about how to formulate this problem. We will revisit it in the next lecture.

Quick summary on problem formulation

- Formulate problems as a search problem
  - Decide your level of abstraction.   State, Action, Goal, Cost.
  - Represented by a state-diagram
  - Required solution: A sequence of actions
  - Optimal solution: A sequence of actions with minimum cost.

- Caveats:
  - Might not be a finite graph
  - Might not have a solution
  - Often takes exponential time to find the optimal solution

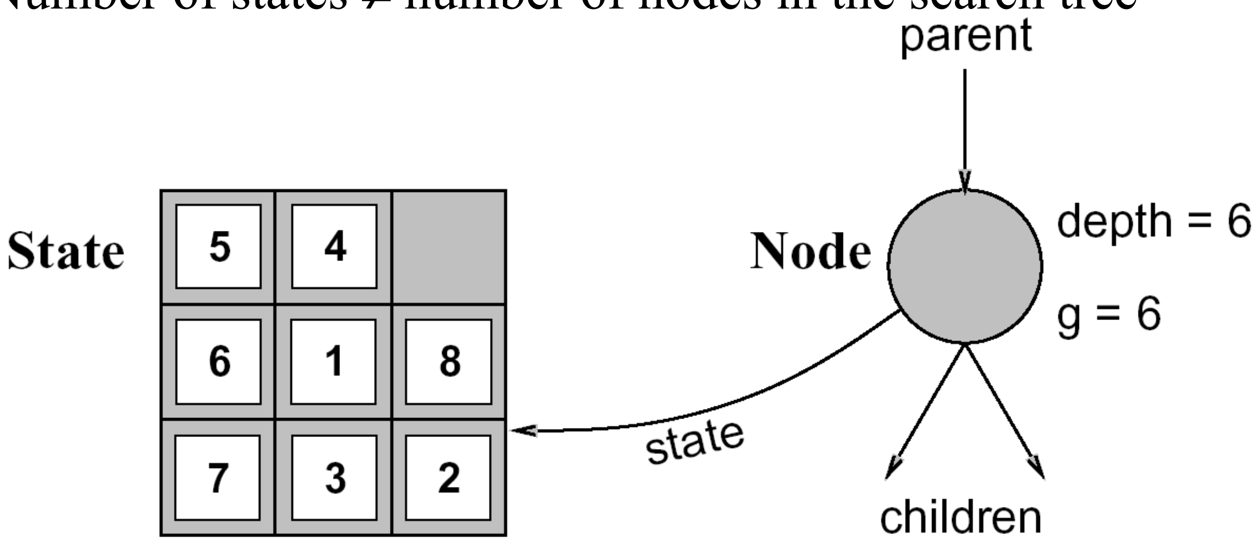
Let's try solving it anyways!
- Do we need an exact optimal solution?
- Are problems in practice worst case?

Searching for Solutions

- Finding a solution is done by searching through the state space
  - While maintaining a set of partial solution sequences
- The *search strategy* determines which states should be expanded first
  - **Expand** a state = Applying the operators to the current state and thereby generating a new set of successor states
- Conceptually, the search process builds up a *search tree* that is superimposed over the state space
  - Root node of the tree ↔ Initial state
  - Leaves of the tree ↔ States to be expanded (or expanded to null)
  - At each step, the search algorithm chooses a leaf to expand

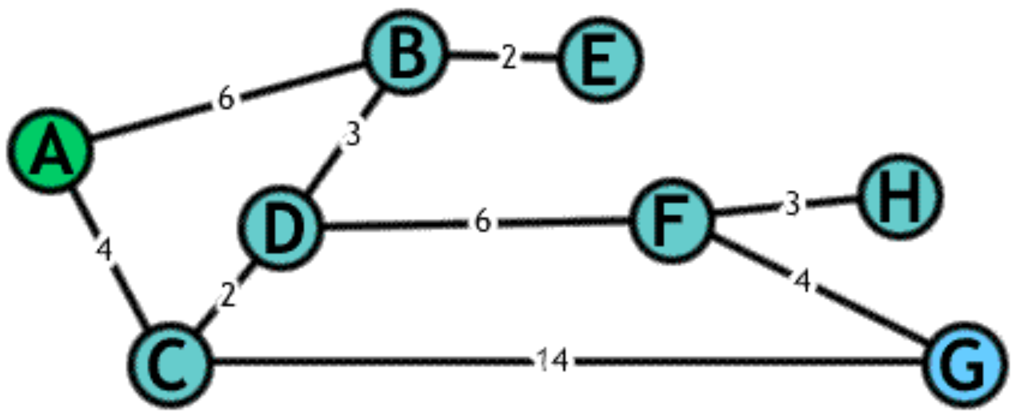
State Space vs. Search Tree

- The **state space** and the **search tree** are not the same thing!
  - A *state* represents a (possibly physical) configuration
  - A *search tree node* is a data structure which includes:
    - { parent, children, depth, path cost }
  - States do not have parents, children, depths, path costs
  - Number of states ≠ number of nodes in the search tree



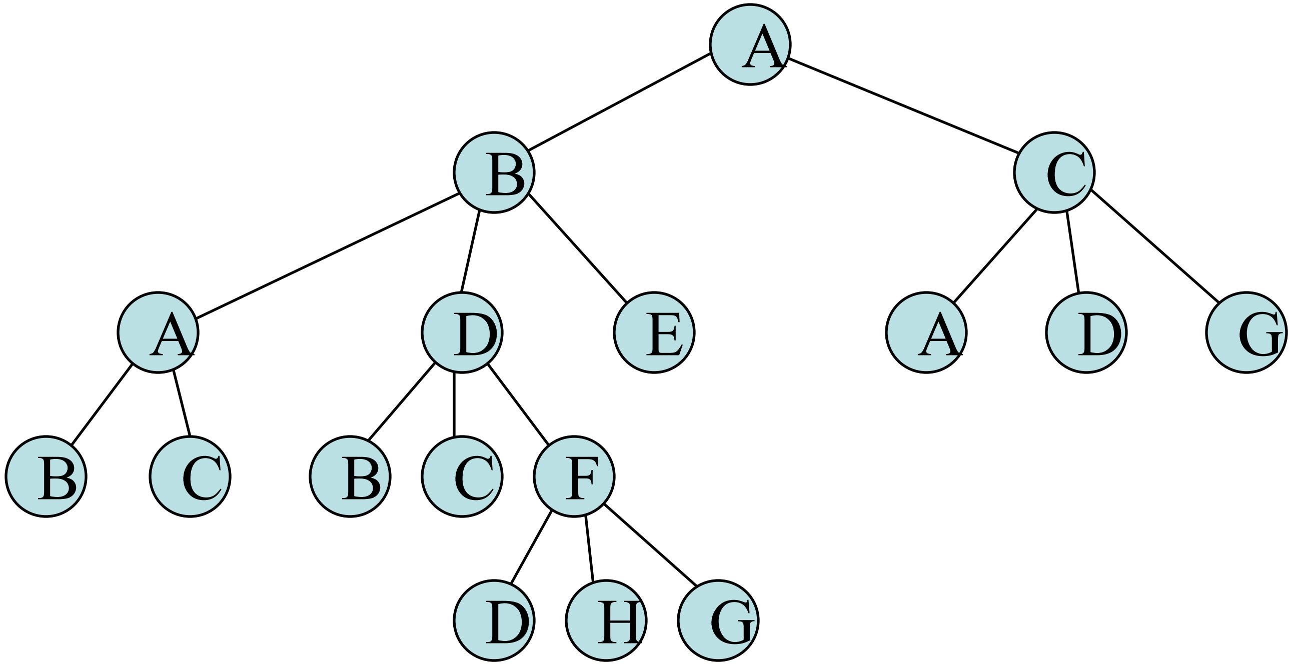
State Space vs. Search Tree (cont.)

State space: 8 states



State Space vs. Search Tree (cont.)

Search tree (partially expanded)



Search Strategies

- Uninformed (blind) search
  - Can only distinguish goal state from non-goal state
- Informed (heuristic) search
  - Can evaluate states

Uninformed ("Blind") Search Strategies

- No information is available other than
  - The current state
    - Its parent (perhaps complete path from initial state)
    - Its operators (to produce successors)
  - The goal test
  - The current path cost (cost from start state to current state)

- Blind search strategies
  - Breadth-first search
  - Uniform cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search

General Search Algorithm (Version 1)

- Various strategies are merely variations of the following function:

```
function GENERAL-SEARCH(problem, strategy) returns a solution or failure

 initialize the search tree using the initial state of problem
 loop do
   if there are no candidates for expansion then return failure
   choose a leaf node for expansion according to strategy
   if the node contains a goal state then return the corresponding solution
   else expand the node and add the resulting nodes to the search tree
 end
```

(Called "Tree-Search" in the textbook)

General Search Algorithm (Version 2)

- Uses a queue (a list) and a **queuing function** to implement a *search strategy*
  - **Queuing-Fn(*queue*, *elements*)** inserts a set of elements into the queue and determines the order of node expansion

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution or failure

 nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
 loop do
   if nodes is empty then return failure
   node ← REMOVE-FRONT(nodes)
   if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
   nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
 end
```

"Nodes" is also known as a **"frontier"** --- the set of states we haven't yet explored/expanded.
"EXPAND" is known as the "**successor function**" --- the set of all states that you could expand on.

How do we evaluate a search algorithm?

- Primary criteria to evaluate search strategies
  - **Completeness**
    - Is it guaranteed to find a solution (if one exists)?
  - **Optimality** *Note that this is not saying it's space/time complexity is optimal.
    - Does it find the "best" solution (if there are more than one)?
  - **Time complexity**
    - Number of nodes generated/expanded
    - (How long does it take to find a solution?)
  - **Space complexity**
    - How much memory does it require?

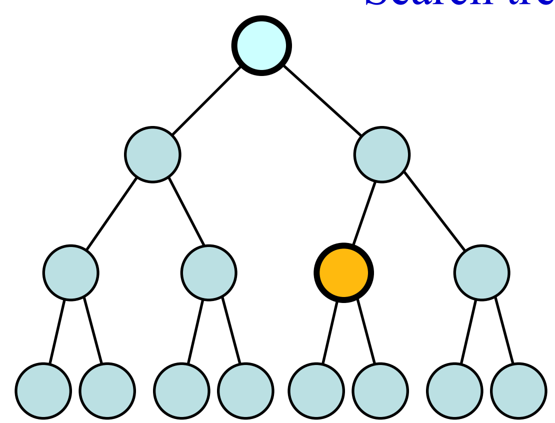
- Some performance measures
  - Best case
  - Worst case
  - Average case
  - Real-world case

How do we evaluate a search algorithm?

- Complexity analysis and O( ) notation (see Appendix A)
  - b = Maximum branching factor of the search tree
  - d = Depth of an optimal solution (may be more than one)
  - m = maximum depth of the search tree (may be infinite)

- Examples
  - O($b^3 d^2$) – polynomial time
  - O(b^d) – exponential time

Search tree



For chess, b_{ave} = 35

$b = 2, \quad d = 2, \quad m = 3$

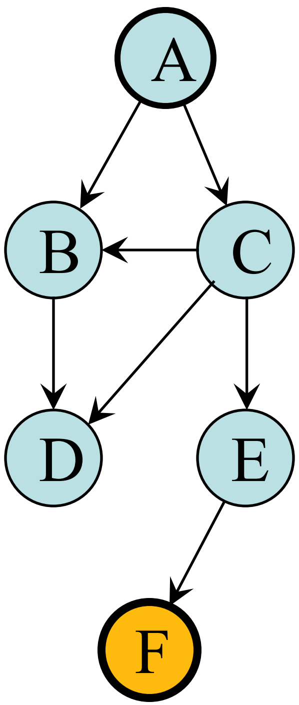
Breadth-First Search

- All nodes at depth *d* in the search tree are expanded before any nodes at depth *d+1*
  - First consider all paths of length N, then all paths of length N+1, etc.
- Doesn't consider path cost – finds the solution with the shortest path
- Uses FIFO queue

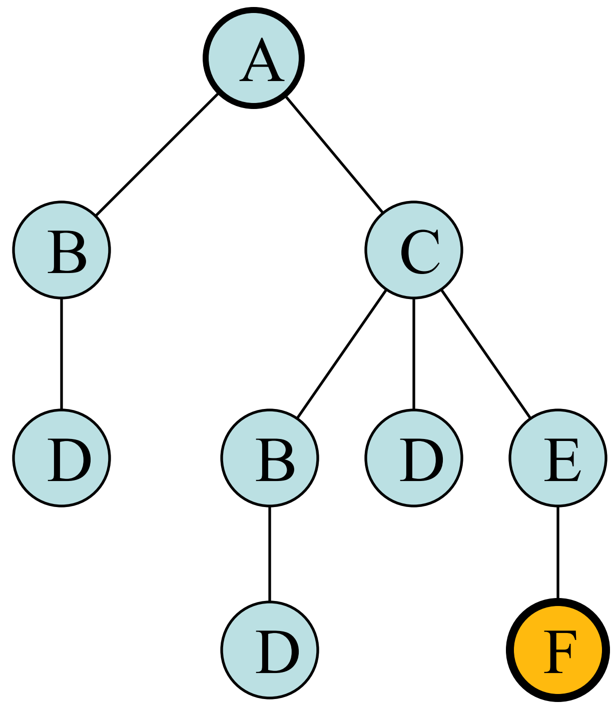
function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution or failure
 return **GENERAL-SEARCH**(*problem,* ENQUEUE-AT-END)

Example

State space graph



Search tree



Queue

(A)

(B C)

(C D)

(D B D E)

(B D E)

(D E D)

(E D)

(D F)

(F)

( )

Breadth-First Search

- Complete? Yes
- Optimal? If shallowest goal is optimal
- Time complexity? Exponential: **O(** b^{d+1} **)**
- Space complexity? Exponential: **O(** b^{d+1} **)**

In practice, the memory requirements are typically worse than the time requirements

b = branching factor (require finite b)
d = depth of shallowest solution

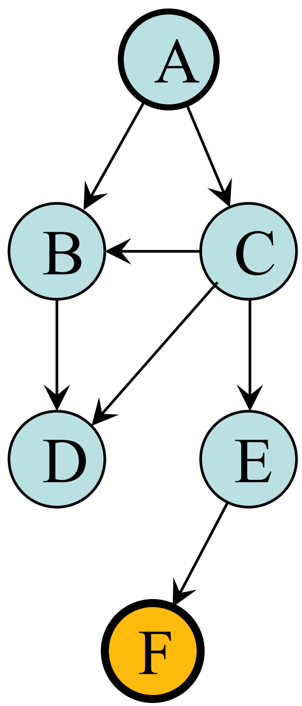
Depth-First Search

- Always expands one of the nodes at the deepest level of the tree
  - Low memory requirements
  - Problem: depth could be infinite
- Uses a stack (LIFO)

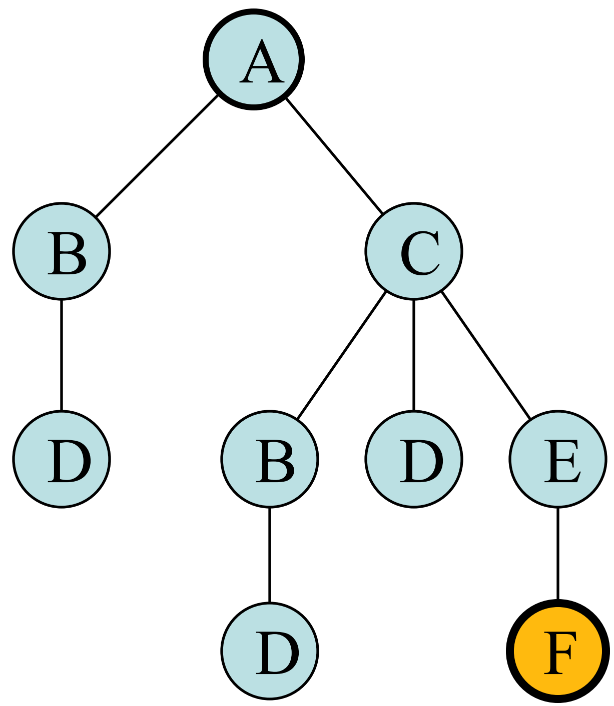
function DEPTH-FIRST-SEARCH(*problem*) **returns** a solution or failure
 return **GENERAL-SEARCH**(*problem,* ENQUEUE-AT-FRONT)

Example

State space graph



Search tree



Queue

(A)

(B C)

(D C)

(C)

(B D E)

(D D E)

(D E)

(E)

(F)

Depth-First Search

- Complete? No
- Optimal? No
- Time complexity? Exponential: **O(b^m)**
- Space complexity? Polynomial: **O(bm)**

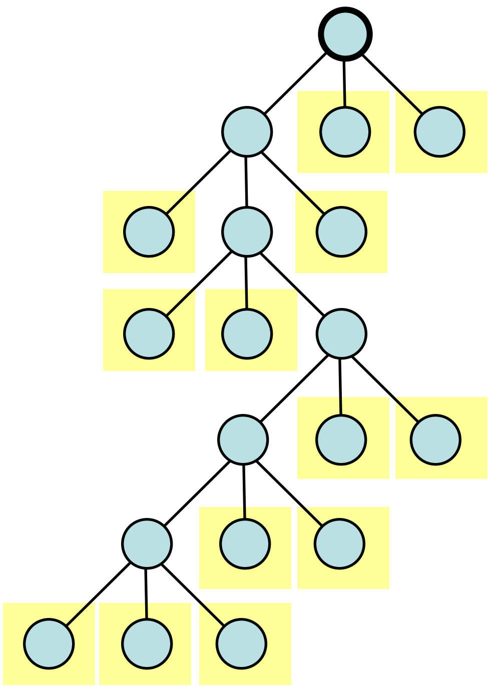
m = maximum depth of the search tree
(may be infinite)

What is the difference between the BFS / DFS that you learned from the algorithm / data structure course?

- Nothing, except:
  - Now you are applying them to solve an AI problem
  - The graph can be infinitely large
  - The graph does not need to be known ahead of time (you only need local information: goal-state checker, successor function)

Space complexity of DFS

- Why is the *space* complexity (memory usage) of depth-first search O(*bm*)?
  - Remove expanded node when all descendents evaluated
  - At each of the *m* levels, you have to keep *b* nodes in memory



Example:

$b = 3$

$m = 6$

Nodes in memory: $bm+1 = 19$

Actually, $(b\text{-}1)m + 1 = 13$ nodes, the way we have been keeping our node list

Depth-Limited Search

- Like depth-first search, but uses a depth cutoff to avoid long (possibly infinite), unfruitful paths
  - Do depth-first search up to depth limit l
  - Depth-first is special case with limit = *inf*
- Problem: How to choose the depth limit ***l*** ?
  - Some problem statements make it obvious (e.g., TSP), but others don't (e.g., MU-puzzle, from the supplementary slide last time)

function DEPTH-LIMITED-SEARCH(*problem, depth-limit*) **returns** a solution or failure
return **GENERAL-SEARCH**(*problem,* ENQUEUE-AT-FRONT-IF-UNDER-DEPTH-LIMIT)

Must explicitly represent node depth

Depth-Limited Search

l = depth limit

- Complete? No, unless $d \le l$
- Optimal? No
- Time complexity? Exponential: $\mathbf{O}(\, \boldsymbol{b^l} \,)$
- Space complexity? Exponential: $\mathbf{O}(\, \boldsymbol{bl} \,)$

Iterative-Deepening Search

- Since the depth limit is difficult to choose in depth-limited search, use depth limits of $l = 0, 1, 2, 3, \ldots$
  - Do depth-limited search at each level

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution or failure
  for *depth* $\leftarrow$ 0 to ∞ **do**
    if DEPTH-LIMITED-SEARCH(*problem, depth*) succeeds **then return** result
  end
  return failure

Iterative-Deepening Search

- IDS has advantages of
  - Breadth-first search – similar optimality and completeness guarantees
  - Depth-first search – Modest memory requirements

- This is the preferred blind search method when the search space is *large* and the solution depth is *unknown*

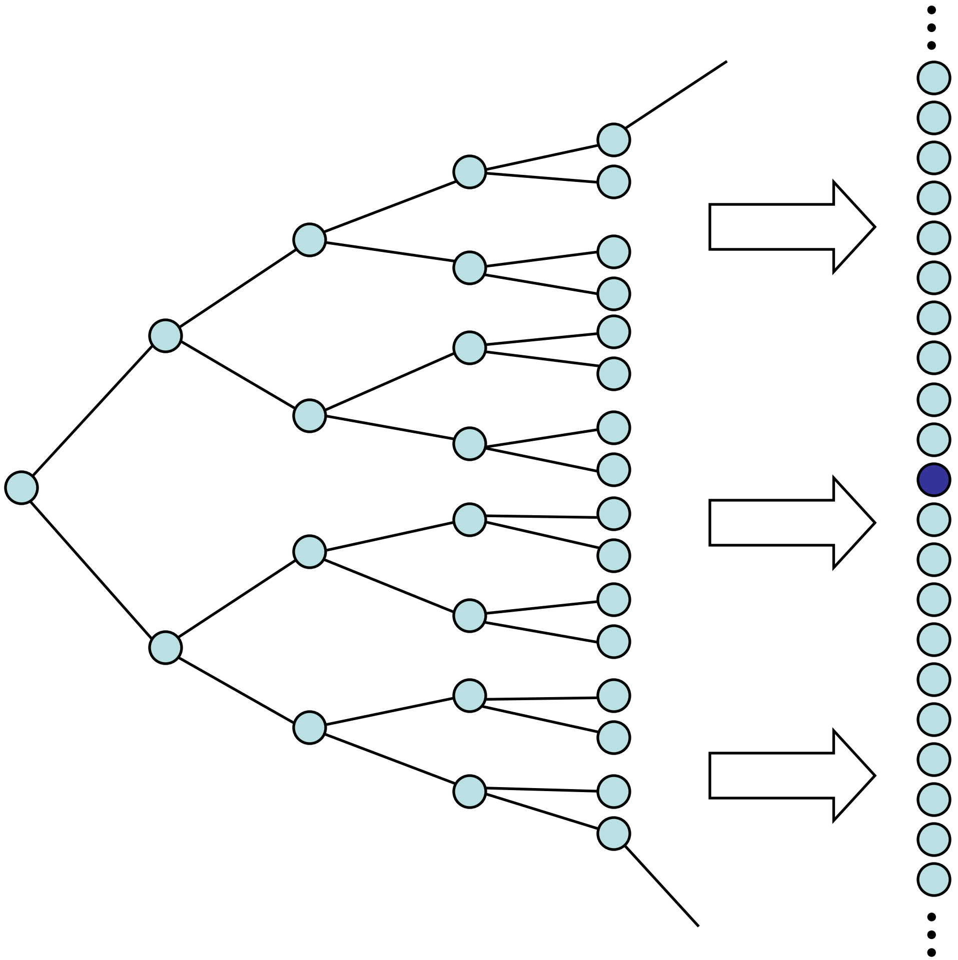
- Many states are expanded multiple times
  - Is this terribly inefficient?
    - No… and it's great for memory (compared with breadth-first)
    - Why is it not particularly inefficient?

Iterative-Deepening Search: Efficiency

- Complete? Yes
- Optimal? Same as BFS
- Time complexity? Exponential: **O(** $\boldsymbol{b^d}$ **)**
- Space complexity? Polynomial: **O(** $\boldsymbol{bd}$ **)**

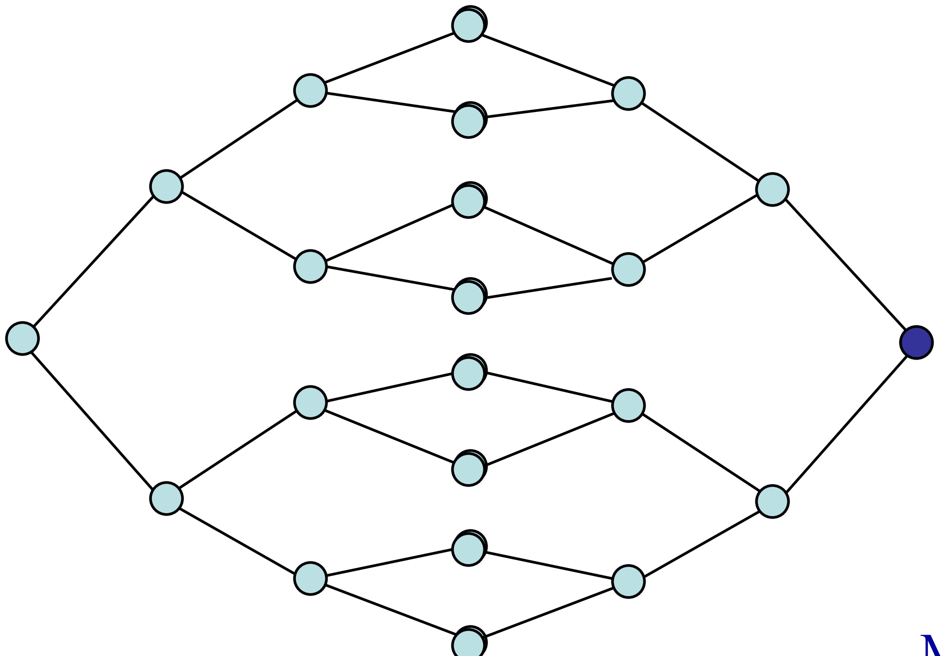
Bidirectional Search

Forward search only:



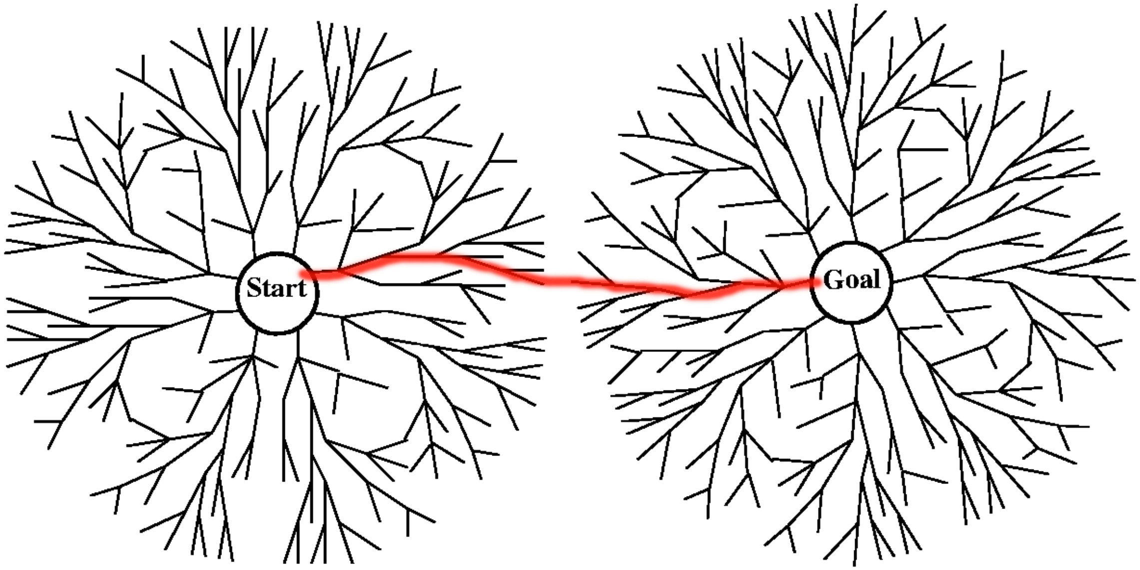
Bidirectional Search

Simultaneously search forward from the initial state and backward from the goal state



Much more efficient!

Bidirectional Search



Example:
$4^{10} \approx 1{,}000{,}000$
$2*4^5 \approx 2{,}000$

- O($b^{d/2}$) rather than O(b^d) – hopefully
- Both actions and predecessors (inverse actions) must be defined
- Must test for intersection between the two searches
  - Constant time for test?
- Really a search strategy, not a specific search method
  - Often not practical….

Bidirectional Search

- Complete? Yes
- Optimal? Same as BFS
- Time complexity? Exponential: **O(** $b^{d/2}$ **)**
- Space complexity? Exponential: **O(** $b^{d/2}$ **)**

* Assuming breadth-first search used from both ends

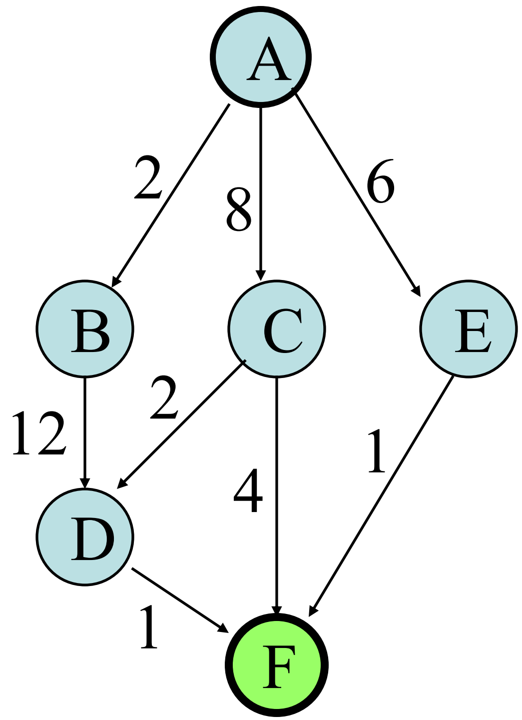
Uniform Cost Search

- Similar to breadth-first search, but always expands the lowest-cost node, as measured by the path cost function, *g(n)*
  - *g(n)* is (actual) cost of getting to node n
  - Breadth-first search is actually a special case of uniform cost search, where *g(n)* = DEPTH(*n*)
  - If the path cost is monotonically increasing, uniform cost search will find the optimal solution

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution or failure
return GENERAL-SEARCH(*problem*, ENQUEUE-IN-COST-ORDER)

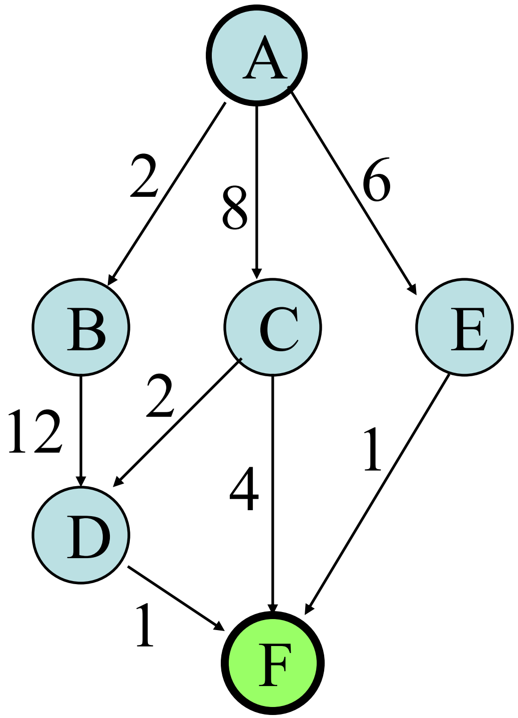
(Dijkstra's algorithm of an potentially infinite graph)

Example (3 min work)



Try breadth-first and uniform cost

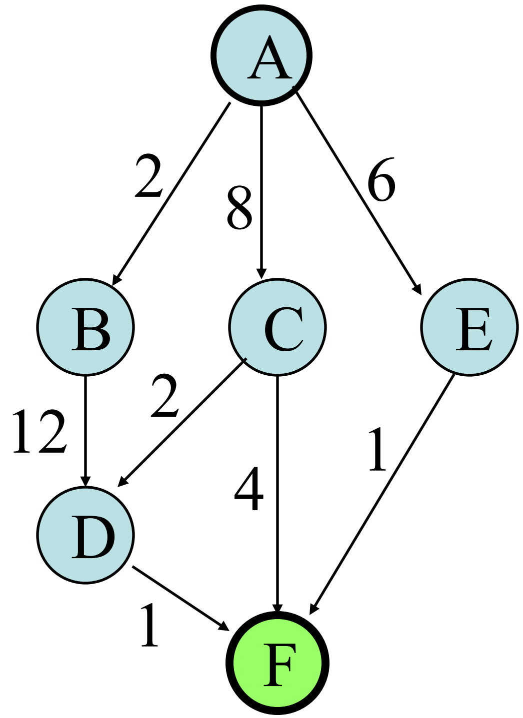
Example (3 min work): Breath-First Search



Node to expand:

Frontier:

Example (3 min work): Uniform Cost Search



Node to expand: Frontier:

Uniform-Cost Search

C = optimal cost
ε = minimum step cost

- Complete? Yes, if $\varepsilon > 0$
- Optimal? Yes
- Time complexity? Exponential: **O($b^{\lfloor C/\varepsilon \rfloor}$)**
- Space complexity? Exponential: **O($b^{\lfloor C/\varepsilon \rfloor}$)**

Same as breadth-first if all edge costs are equal

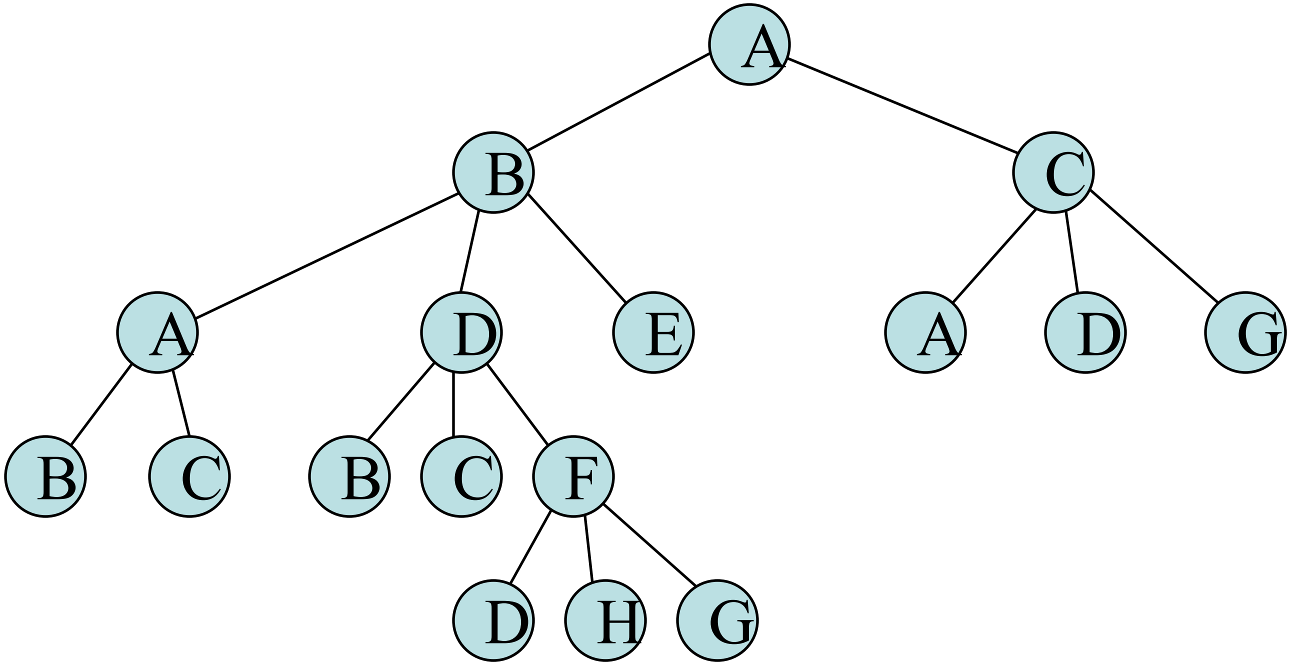
Can we do better than Tree Search?

- Sometimes.

- When the number of states are small
  - Dynamic programming (smart way of doing exhaustive search)

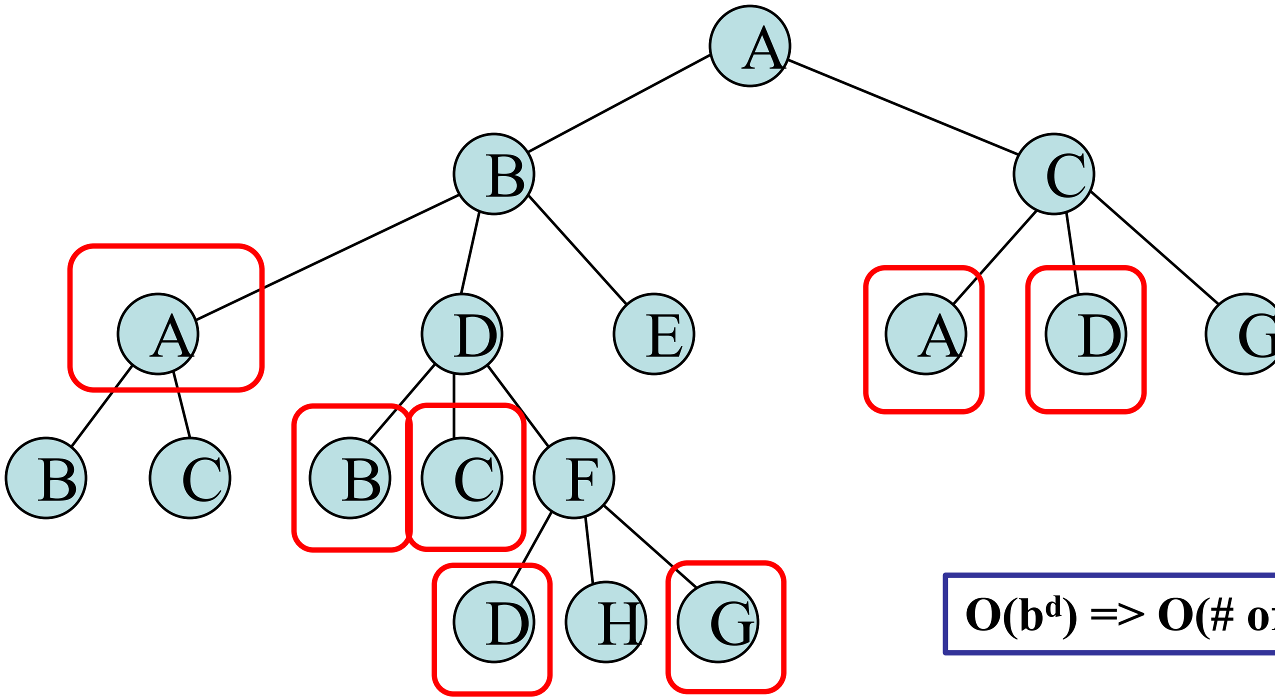
State Space vs. Search Tree (cont.)

Search tree (partially expanded)



Search Tree => Search Graph

Dynamic programming (with book keeping)



O(b^d) => O(# of states)

Graph Search vs Tree Search

- Tree Search
  - We might repeat some states
  - But we do not need to remember states

- Graph Search
  - We remember all the states that have been explored
  - But we do not repeat some states

Summary table of uninformed search

Criteria	BFS	Uniform-cost	DFS	Depth-limited	IDS	Bidirectional
Complete?	Yes#	Yes#&	No	No	Yes#	Yes#+
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes$	Yes	No	No	Yes$	Yes$+

b: Branching factor
d: Depth of the shallowest goal
l: Depth limit
m: Maximum depth of search tree
e: The lower bound of the step cost

(Section 3.4.6 in the AIMA book.)

#: Complete if *b* is finite
&: Complete if step cost >= *e*
$: Optimal if all step costs are identical
+: If both direction use BFS

Practical note about search algorithms

- The computer can't "see" the search graph like we can
  - No "bird's eye view" – make relevant information explicit!
- What information should you keep for a node in the search tree?
  - State
    - (1 2 0)
  - Parent node (or perhaps complete ancestry)
    - Node #3 (or, nodes 0, 2, 5, 11, 14)
  - Depth of the node
    - $d = 4$
  - Path cost up to (and including) the node
    - g(*node*) = 12
  - Operator that produced this node
    - Operator #1

Remainder of the lecture

- Informed search

- Some questions / desiderata
    1. Can we do better with some side information?
    2. We do not wish to make strong assumptions on the side information.
    3. If the side information is good, we hope to do better.
    4. If the side information is useless, we perform as well as an uninformed search method.

Best-First Search (with an Eval-Fn)

function BEST-FIRST-SEARCH(*problem,* EVAL-FN) **returns** a solution or failure
QUEUING-FN ← a function that orders nodes by EVAL-FN
return GENERAL-SEARCH(*problem,* QUEUING-FN)

- Uses a heuristic function, ***h(n)****,* as the EVAL-FN
- ***h(n)*** estimates the cost of the best path from state *n* to a goal state
  - *h(goal)* = 0

Greedy Best-First Search

- Greedy search – always expand the node that appears to be the closest to the goal (i.e., with the smallest ***h***)
  - Instant gratification, hence "greedy"

function GREEDY-SEARCH(*problem*, ***h***) **returns** a solution or failure
  return BEST-FIRST-SEARCH(*problem*, ***h***)

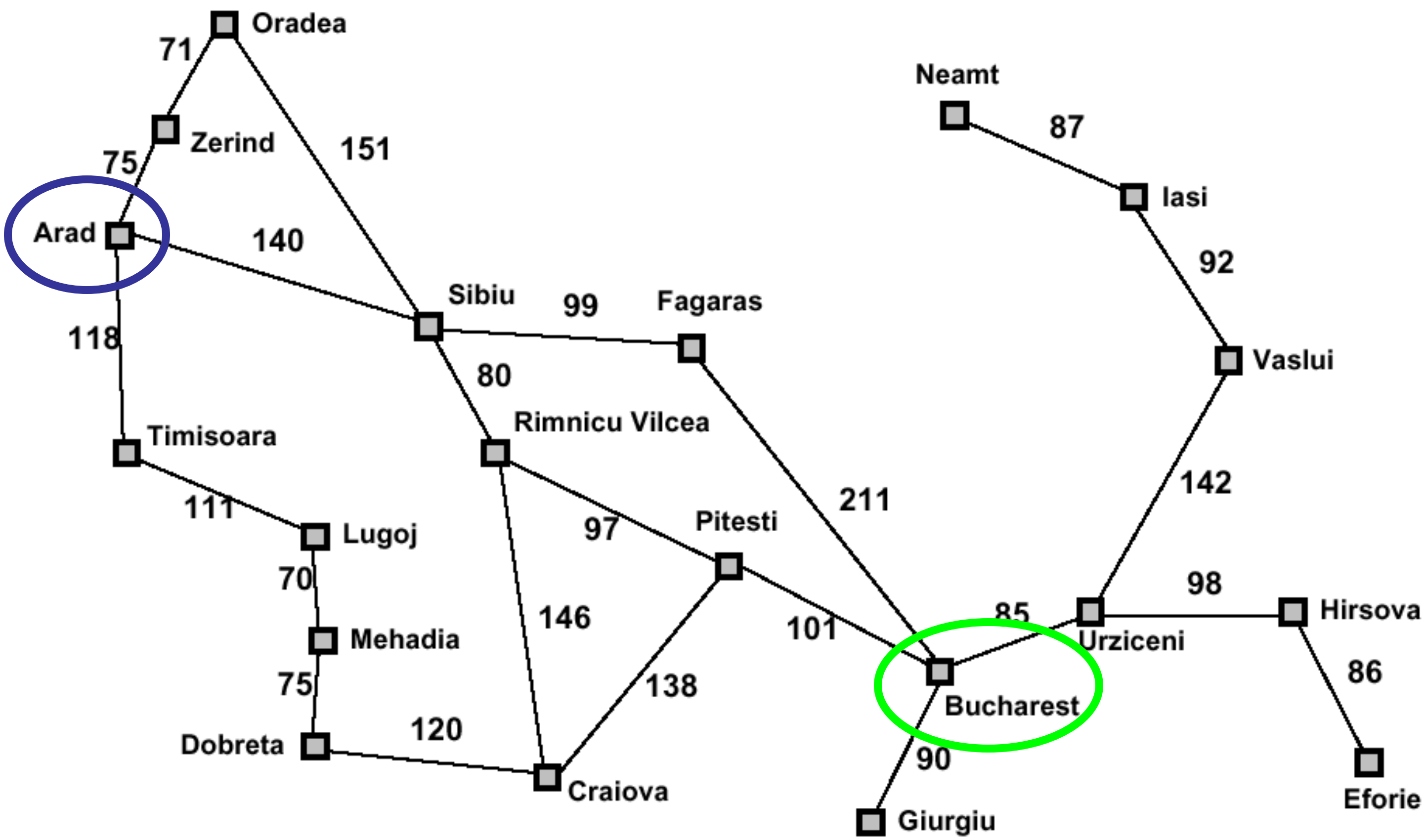
- Greedy search often performs well, but:
  - It doesn't always find the best solution / or any solution
  - It may get stuck
  - It performance completely depends on the particular ***h*** function

A* Search (Pronounced "A-Star")

- Uniform-cost search minimizes ***g(n)*** ("past" cost)
- Greedy search minimizes ***h(n)*** ("expected" or "future" cost)
- **"A*** Search" combines the two:
  - Minimize $f(n) = g(n) + h(n)$
  - Accounts for the "past" and the "future"
  - Estimates the cheapest solution (complete path) through node ***n***

function A*-SEARCH(*problem*, ***h***) **returns** a solution or failure
  return BEST-FIRST-SEARCH(*problem*, ***f***)

A* Example

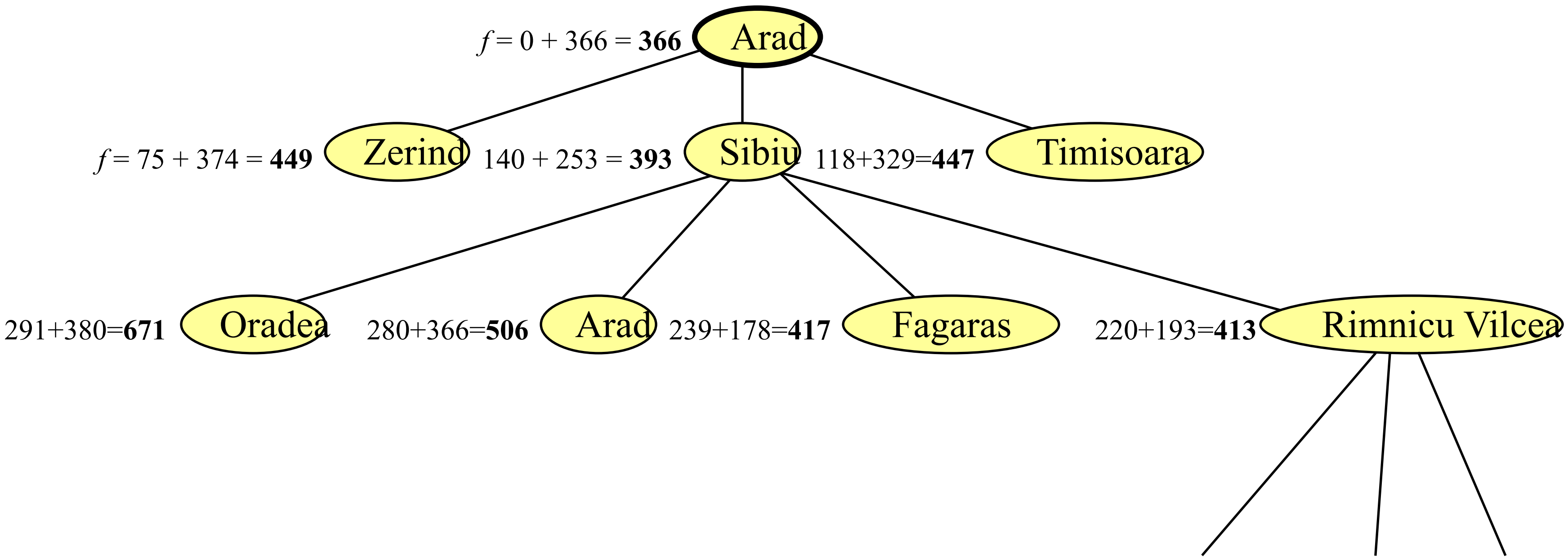


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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

A* Example



f = 0 + 366 = 366
Arad
f = 75 + 374 = 449
Zerind
140 + 253 = 393
Sibiu
118+329=447
Timisoara
291+380=671
Oradea
280+366=506
Arad
239+178=417
Fagaras
220+193=413
Rimnicu Vilcea

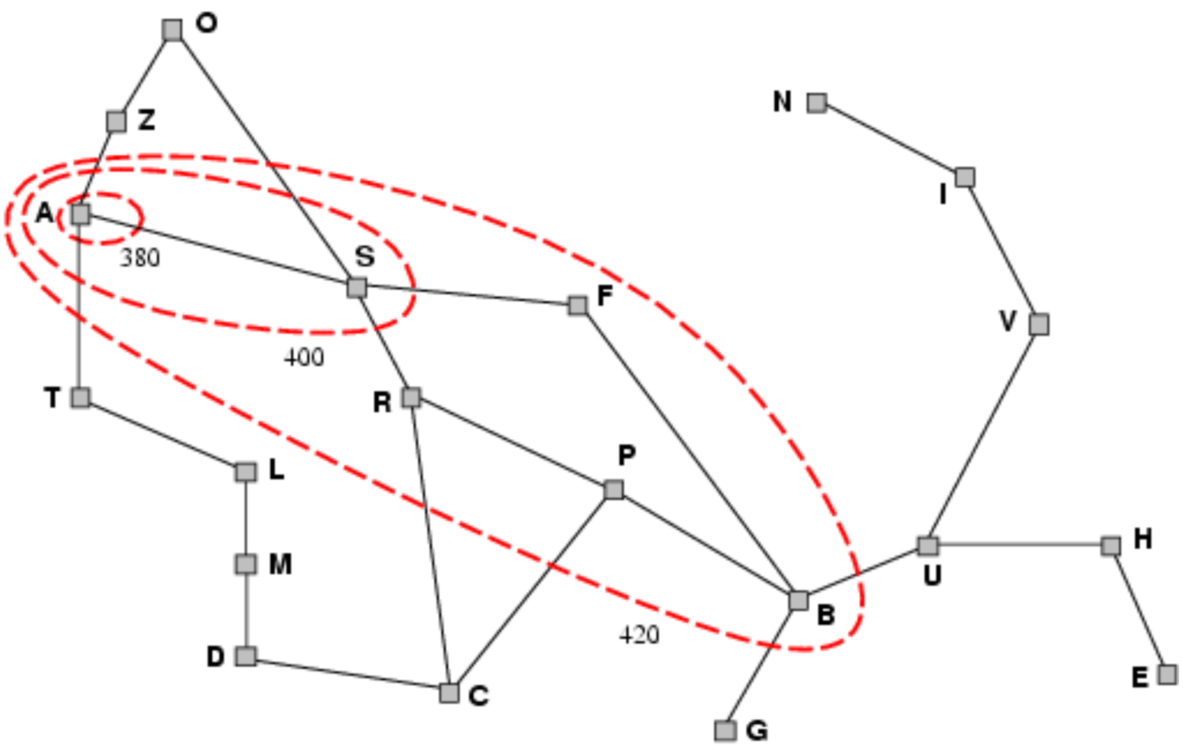
When does A* search "work"?

- Focus on optimality (finding the optimal solution)

- "**A* ** Search" is optimal if *h* is **admissible**
  - ***h*** is optimistic – it never overestimates the cost to the goal
    - $h(n) \leq$ true cost to reach the goal
  - So ***f(n)*** never overestimates the actual cost of <u>the best solution passing through node *n*</u>

Visualizing A* search

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$
-



Optimality of A* with an Admissible h

- Let OPT be the optimal path cost.
  - All non-goal nodes on this path have f $\leq$ OPT.
    - Positive costs on edges
  - The goal node on this path has f = OPT.

- A* search does not stop until an f-value of OPT is reached.
  - All other goal nodes have an f cost higher than OPT.

- All non-goal nodes on the optimal path are eventually expanded.
  - The optimal goal node is eventually placed on the priority queue, and reaches the front of the queue.

Optimal Efficiency of A*

A* is **optimally efficient** for any particular *h(n)*
That is, no other optimal algorithm is guaranteed to expand fewer nodes with the same *h(n)*.

- Need to find a good and efficiently evaluable h(n).

A* Search with an Admissible h

- Optimal? Yes
- Complete? Yes
- Time complexity? Exponential; better under some conditions
- Space complexity? Exponential; keeps all nodes in memory

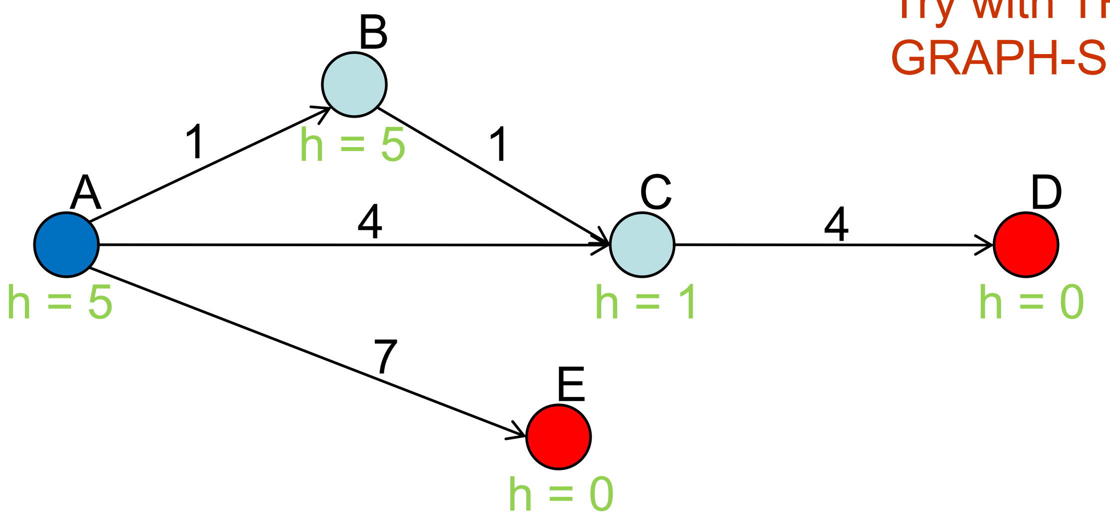
Recall: Graph Search vs Tree Search

- Tree Search
  - We might repeat some states
  - But we do not need to remember states

- Graph Search
  - We remember all the states that have been explored
  - But we do not repeat some states

Avoiding Repeated States using A* Search

- Is GRAPH-SEARCH optimal with A*?



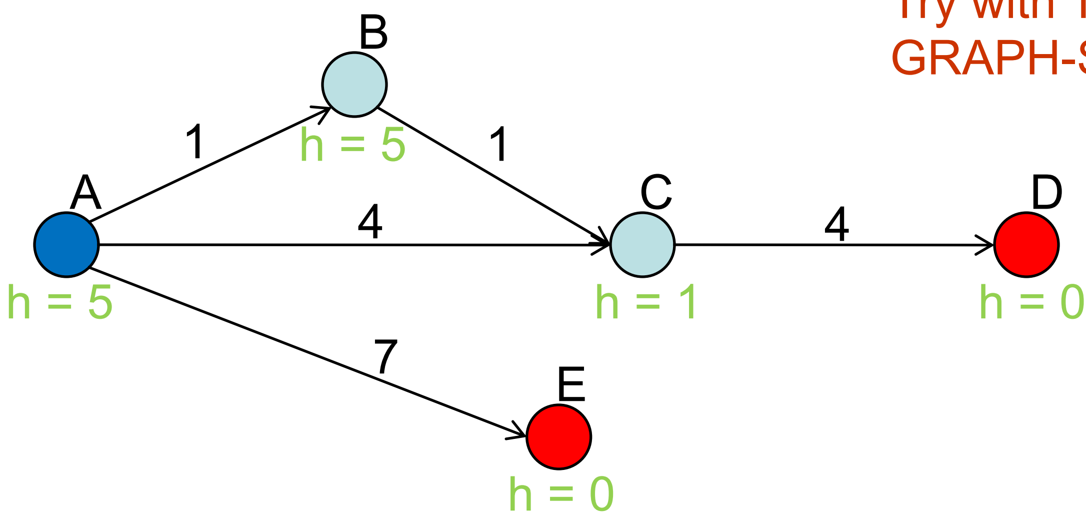
Try with TREE-SEARCH and GRAPH-SEARCH

Graph Search
Step 1: Among B, C, E, Choose C
Step 2: Among B, E, D, Choose B
Step 3: Among D, E, Choose E.   (you are not going to select C again)

Avoiding Repeated States using A* Search

- Is GRAPH-SEARCH optimal with A*?

Try with TREE-SEARCH and GRAPH-SEARCH

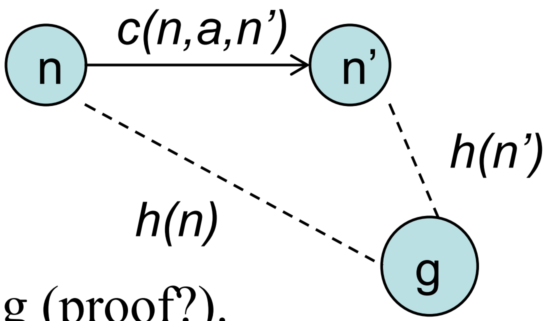


Solution 1: Remember all paths: Need extra bookkeeping

Solution 2: Ensure that the first path to a node is the best!

Consistency (Monotonicity) of heuristic h

- A heuristic is consistent (or monotonic) provided
  - for any node n, for any successor n' generated by action a with cost c(n,a,n')
    - $h(n) \leq c(n,a,n') + h(n')$
  - akin to triangle inequality.
  - guarantees admissibility (proof?).
  - values of f(n) along any path are non-decreasing (proof?).
    - Contours of constant f in the state space
- GRAPH-SEARCH using consistent f(n) is optimal.
- Note that h(n) = 0 is consistent and admissible.



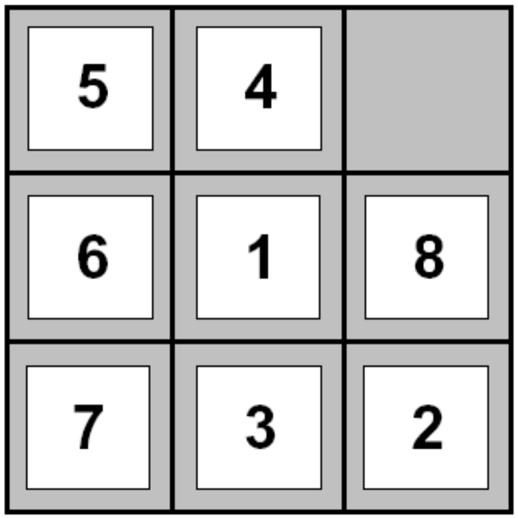
Next lecture

- Examples
- Choosing heuristics
- Games and Minimax Search

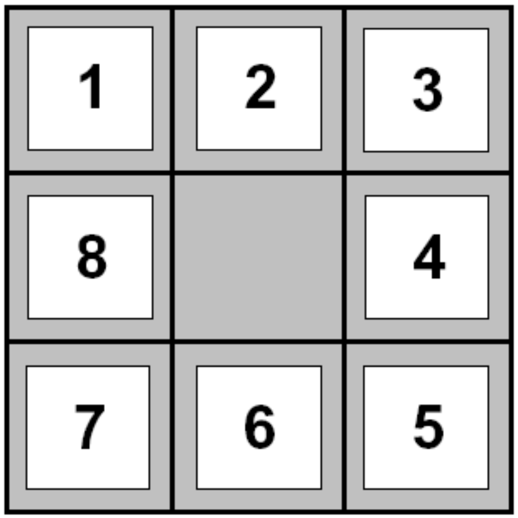
Heuristics

- What's a heuristic for
  - Driving distance (or time) from city A to city B ?
  - 8-puzzle problem ?
  - M&C ?
  - Robot navigation ?
  - Reaching the summit ?
- **Admissible** heuristic
  - Does not overestimate the cost to reach the goal
  - "Optimistic"
- Are the above heuristics admissible? Consistent?

Example: 8-Puzzle



Start State



Goal State

Comparing and combining heuristics

- Heuristics generated by considering relaxed versions of a problem.
- Heuristic h_1 for 8-puzzle
  - Number of out-of-order tiles
- Heuristic h_2 for 8-puzzle
  - Sum of Manhattan distances of each tile
- h_2 dominates h_1 provided $h_2(n) \geq h_1(n)$.
  - h_2 will likely prune more than h_1.
- $\max(h_1, h_2, .., h_n)$ is
  - admissible if each h_i is
  - consistent if each h_i is
- Cost of sub-problems and pattern databases
  - Cost for 4 specific tiles
  - Can these be added for disjoint sets of tiles?

Effective Branching Factor

- Though informed search methods may have poor *worst-case* performance, they often do quite well if the heuristic is good
  - Even if there is a huge branching factor

- One way to quantify the effectiveness of the heuristic: the **effective branching factor, *b****
  - N: total number of nodes expanded
  - d: solution depth
  - $N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$

- For a good heuristic, b^* is close to 1

Example: 8-puzzle problem

Averaged over 100 trials each at different solution lengths

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Ave. # of nodes expanded

Solution length

Summary of informed search

- How to use a heuristic function to improve search
  - Greedy Best-first search + Uniform-cost search = A* Search
- When is A* search optimal?
  - h is Admissible (optimistic) for Tree Search
  - h is Consistent for Graph Search
- Choosing heuristic functions
  - A good heuristic function can reduce time/space cost of search by orders of magnitude.
  - Good heuristic function may take longer to evaluate.