Artificial Intelligence

CS 165A

May 2, 2023

Instructor: Prof. Yu-Xiang Wang

Today

→ Informed Search and Heuristics

→ Games and minimax search

Notes for Project 1

- Almost everyone has completed
  - Don’t forget to submit the report and the leaderboard

- How to catch up if you missed the deadline?
  - 4 late days --- no question asked
  - 75% credits if submit before this Thursday (for the basic coding)
  - 50% credits if submit after this Thursday

- 9 students were able to beat the TA baseline!
  - Truly awesome! Keep it coming!

- Start Project 2 early!

Recap: Search agent and search algorithms

- Representing states, operators and costs
  - State-space diagram: What are the vertices, edges, edge weights?
  - Examples: Romania, Missionary and Cannibals, Pacman, 8-puzzle (and the MU puzzle from the quiz)
- Search algorithms
  - BFS, DFS, Depth-Limited, IDS, Bidirectional Search

time	b^d	b^m	b^l	b^d	$b^{\frac{d}{2}}$
space	b^d	bd	bl	bd	$b^{\frac{d}{2}}$
Comp	Y	N	N	Y	Y
Opt	Y	N	N	Y	Y

- Four criteria to evaluate the search algorithms:
  - Completeness, Optimality, Space complexity, time complexity

This lecture

- Uniform cost search
- Informed search, aka Heuristic Search
- Admissible and consistent heuristics
- Tree search vs Graph Search
- (if time permits) Intro to games and adversarial search

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

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)
```

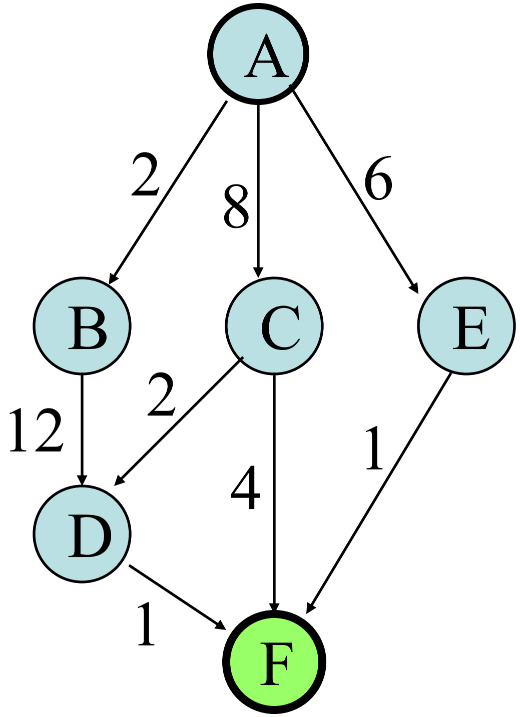
Uniform Cost Search

- 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)
```

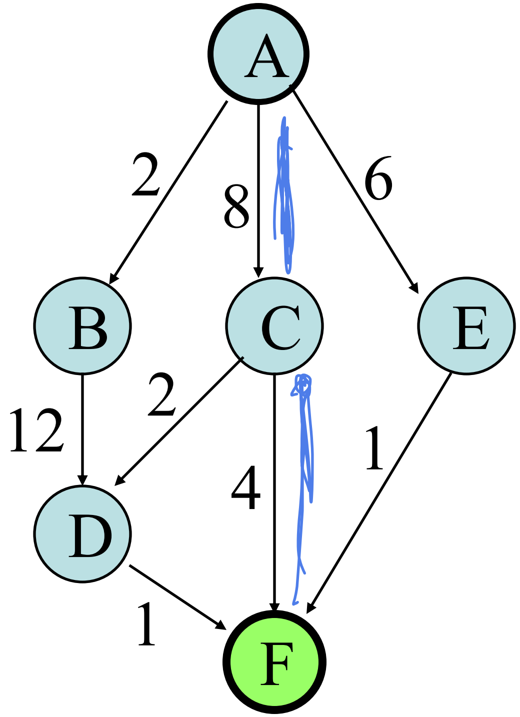
(Dijkstra’s algorithm of an potentially infinite graph)

Example (3 min work)



Try breadth-first and uniform cost

Example (3 min work): Breath-First Search

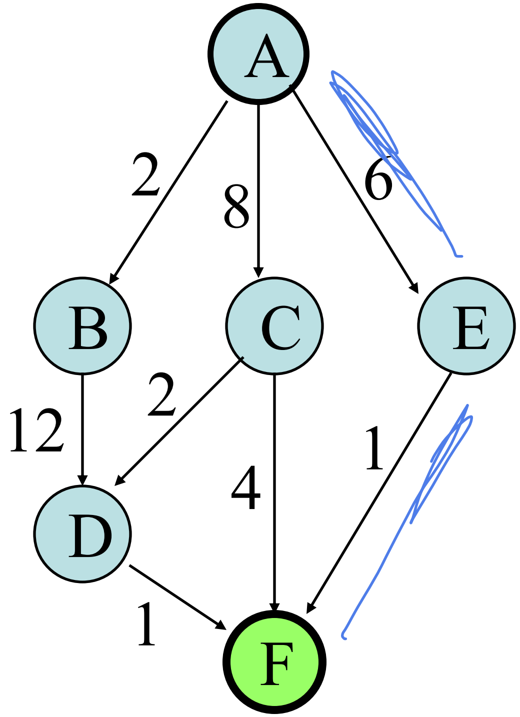


Node to expand:	Frontier:
A	A
B	B C E
C	C E D
E	E D D F
D	D D F F
D	D F F F
F	F F F F

Solution: A -> C -> F

path cost: 12

Example (3 min work): Uniform Cost Search



Node to expand:	Frontier:
A	A: 0
B	B:2, C:8, E:6
E	C:8, E:6, D:14
F	C:8, D:14, F:7

Solutn A → E → F

Cost: 7

Uniform-Cost Search

C = optimal cost
ε = minimum step cost

- Complete? Yes, if $\varepsilon > 0$
- Optimal? Yes
- Time complexity?
- Space complexity?

Uniform-Cost Search

C = optimal cost
ε = minimum step cost

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

Uniform-Cost Search

C = optimal cost
ε = minimum step cost

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

Uniform-Cost Search

C = optimal cost
ε = minimum step cost

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

Same as breadth-first if all edge costs are equal

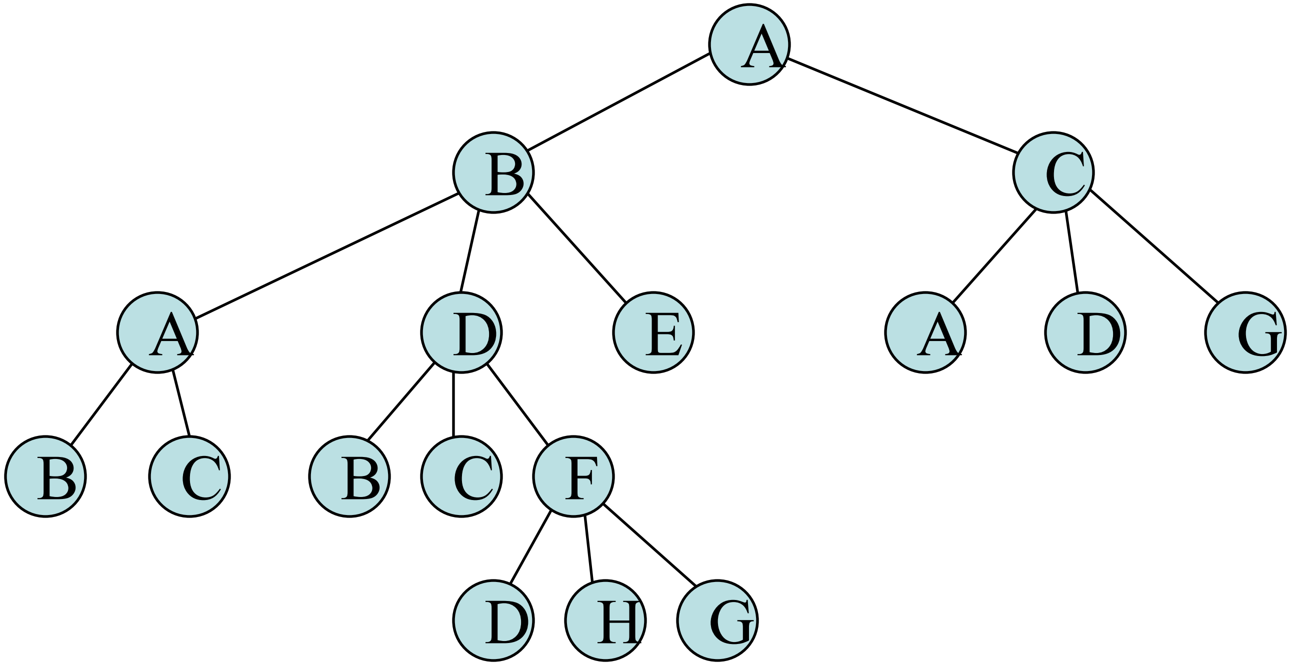
Can we do better than Tree Search?

- Sometimes.

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

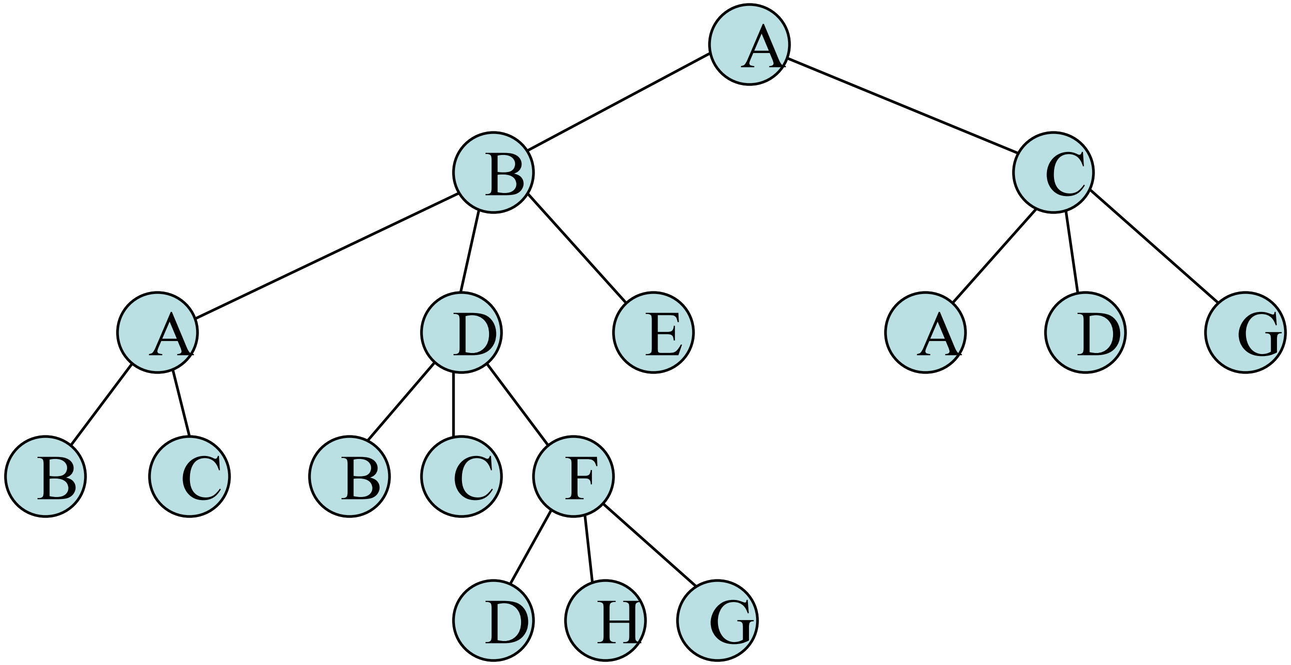
State Space vs. Search Tree (cont.)

Search tree (partially expanded)



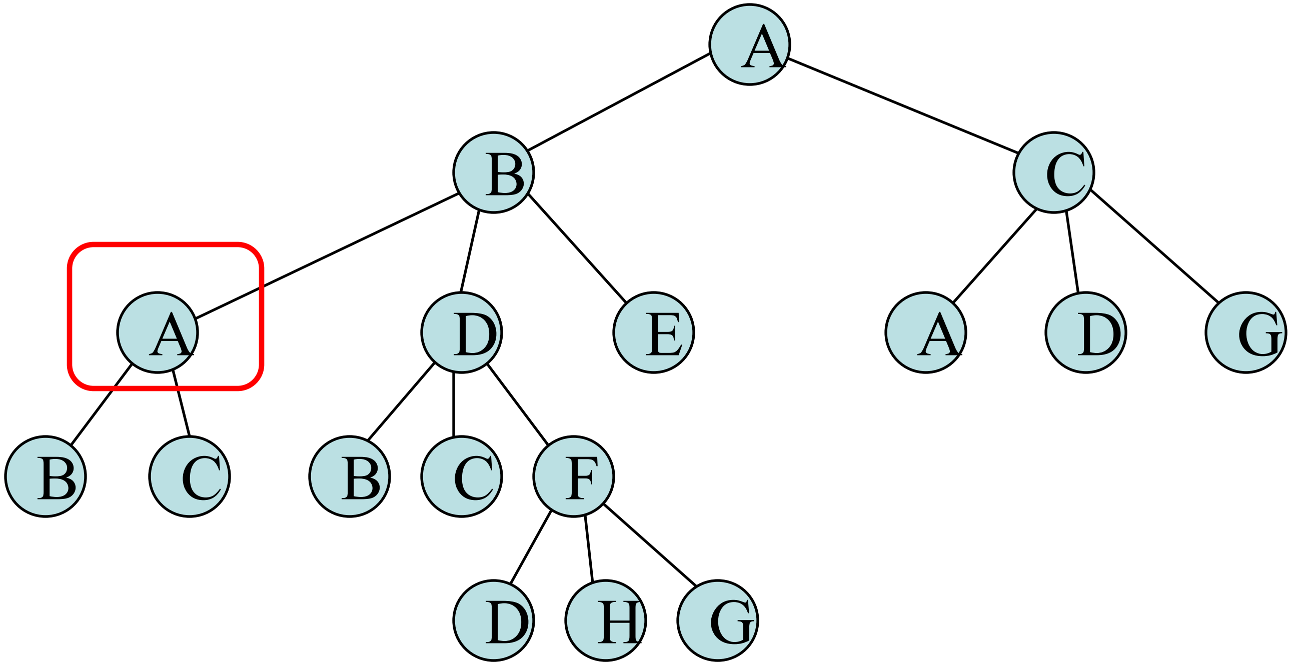
Search Tree => Search Graph

Dynamic programming (with book keeping)



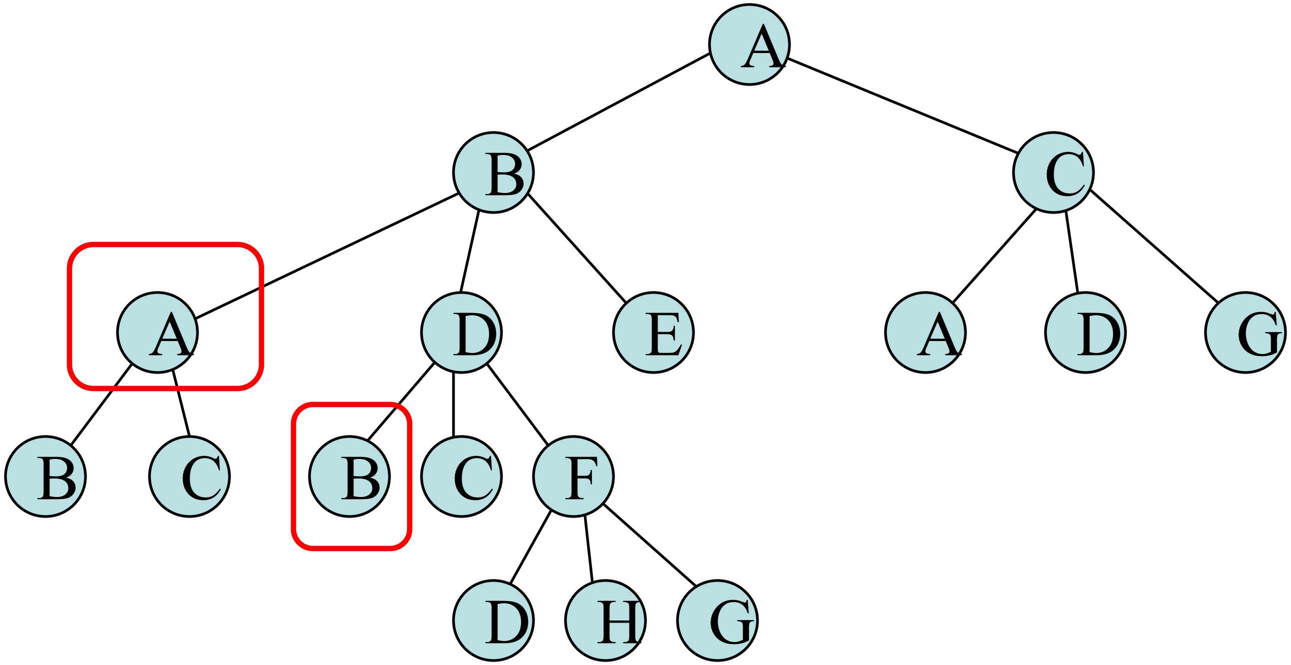
Search Tree => Search Graph

Dynamic programming (with book keeping)



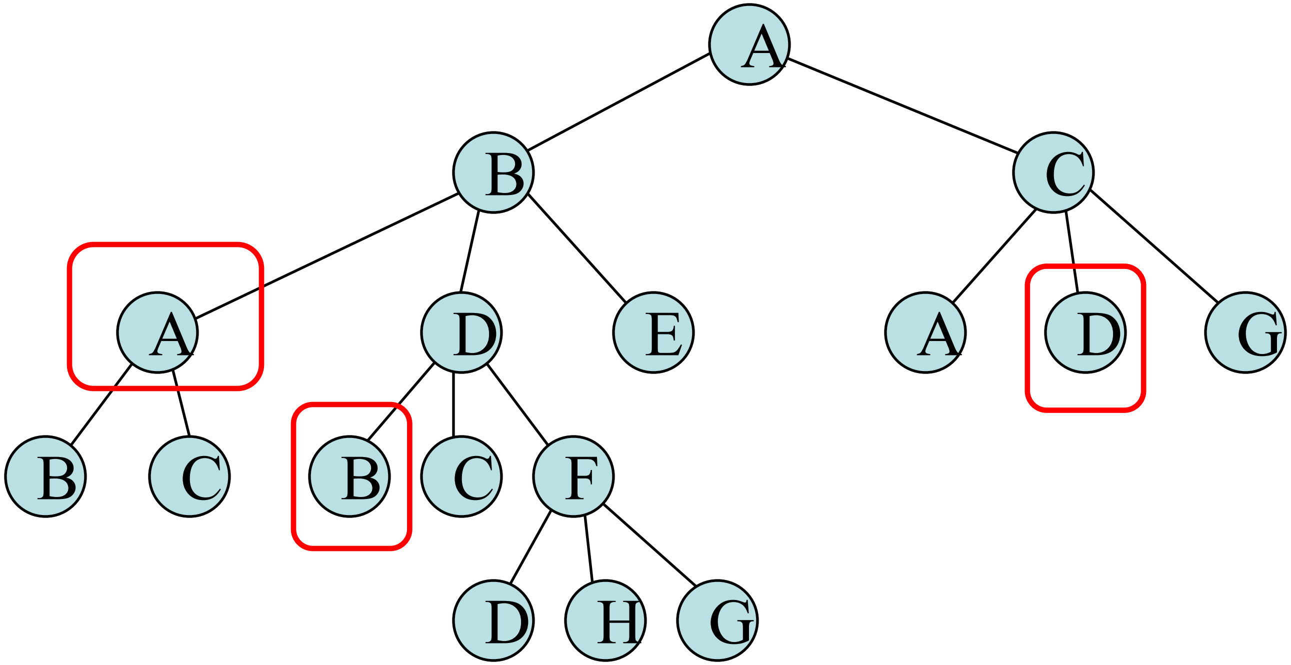
Search Tree => Search Graph

Dynamic programming (with book keeping)



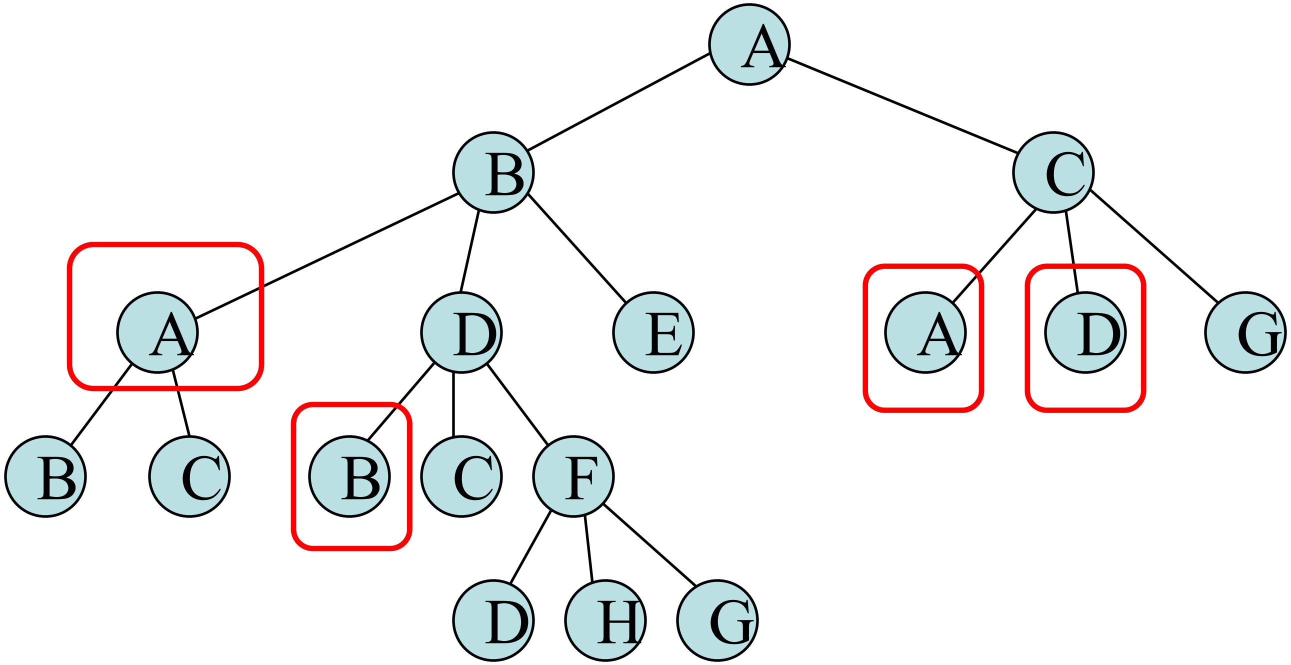
Search Tree => Search Graph

Dynamic programming (with book keeping)



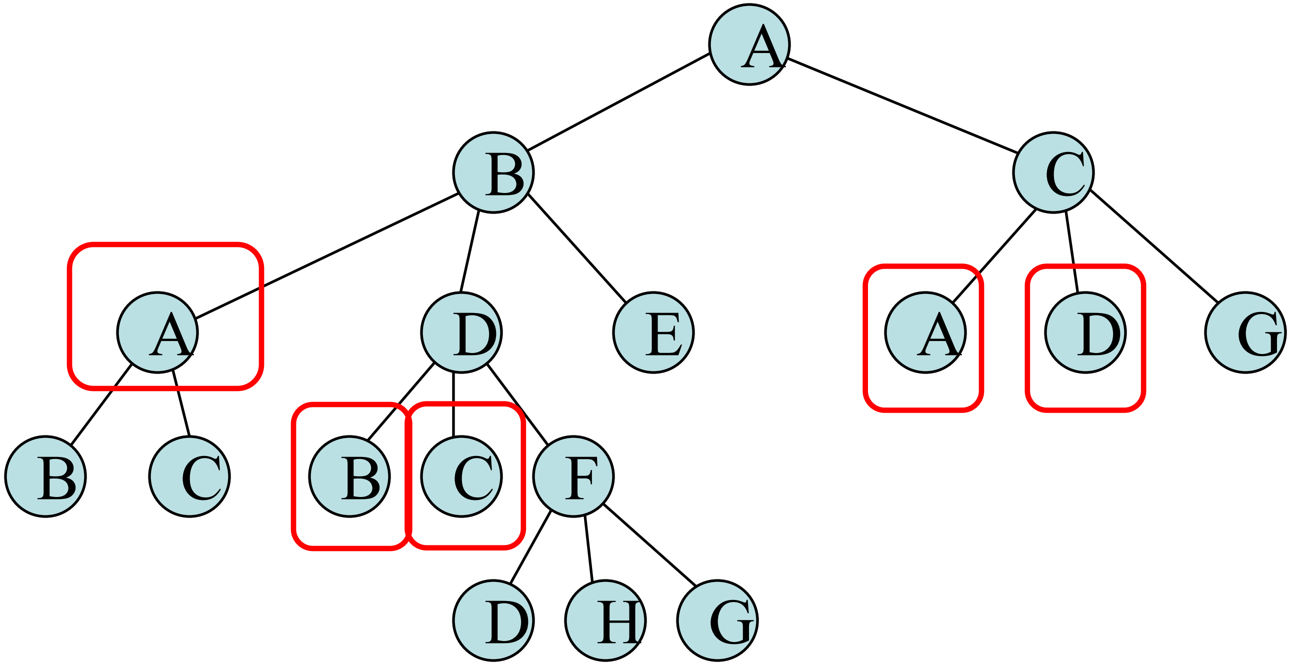
Search Tree => Search Graph

Dynamic programming (with book keeping)



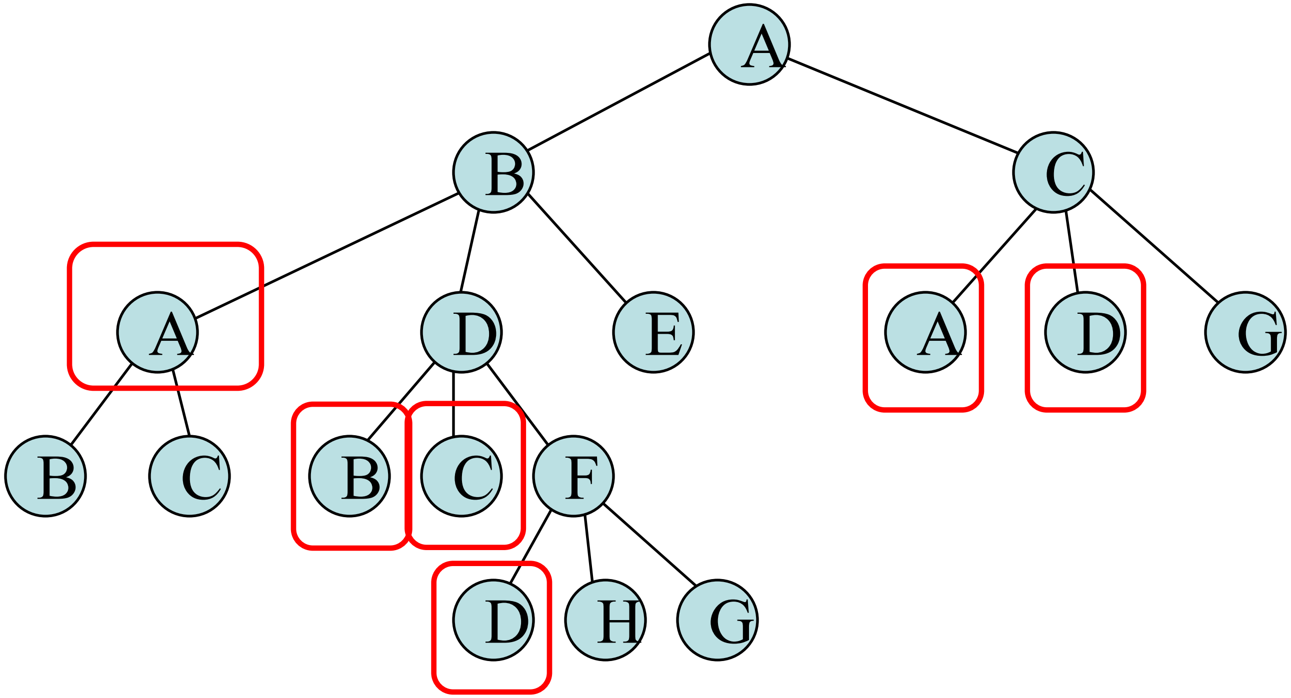
Search Tree => Search Graph

Dynamic programming (with book keeping)



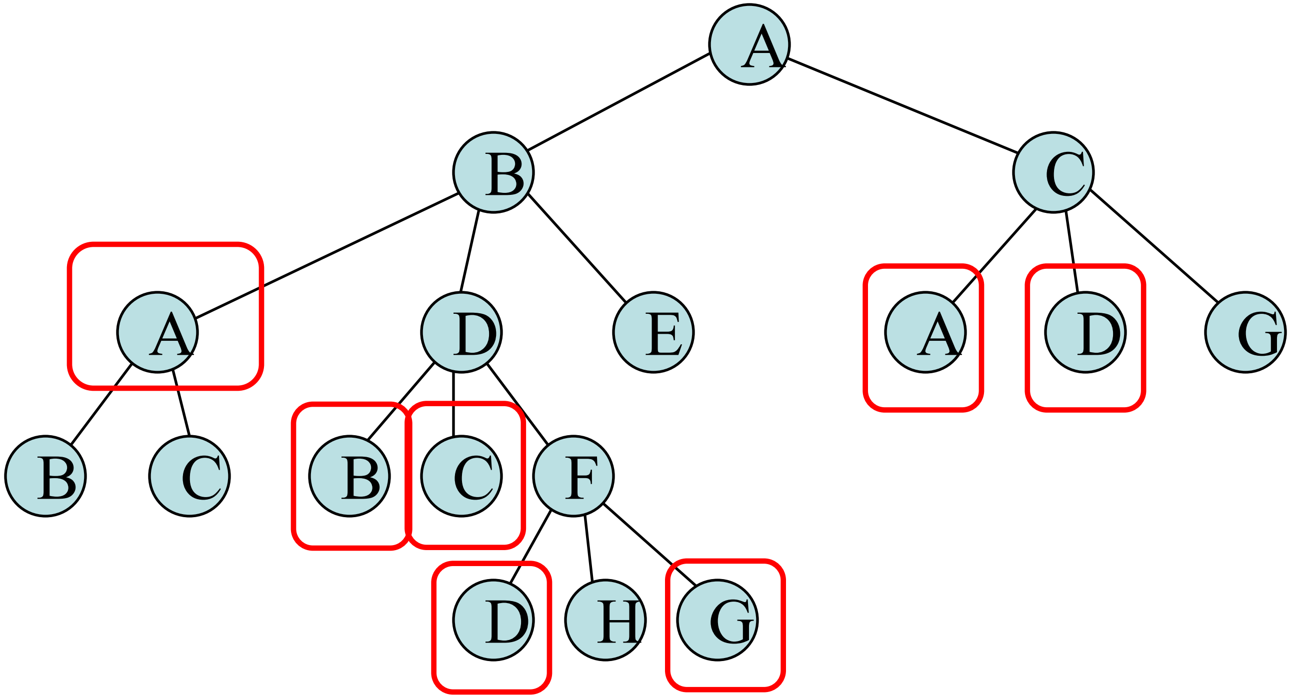
Search Tree => Search Graph

Dynamic programming (with book keeping)



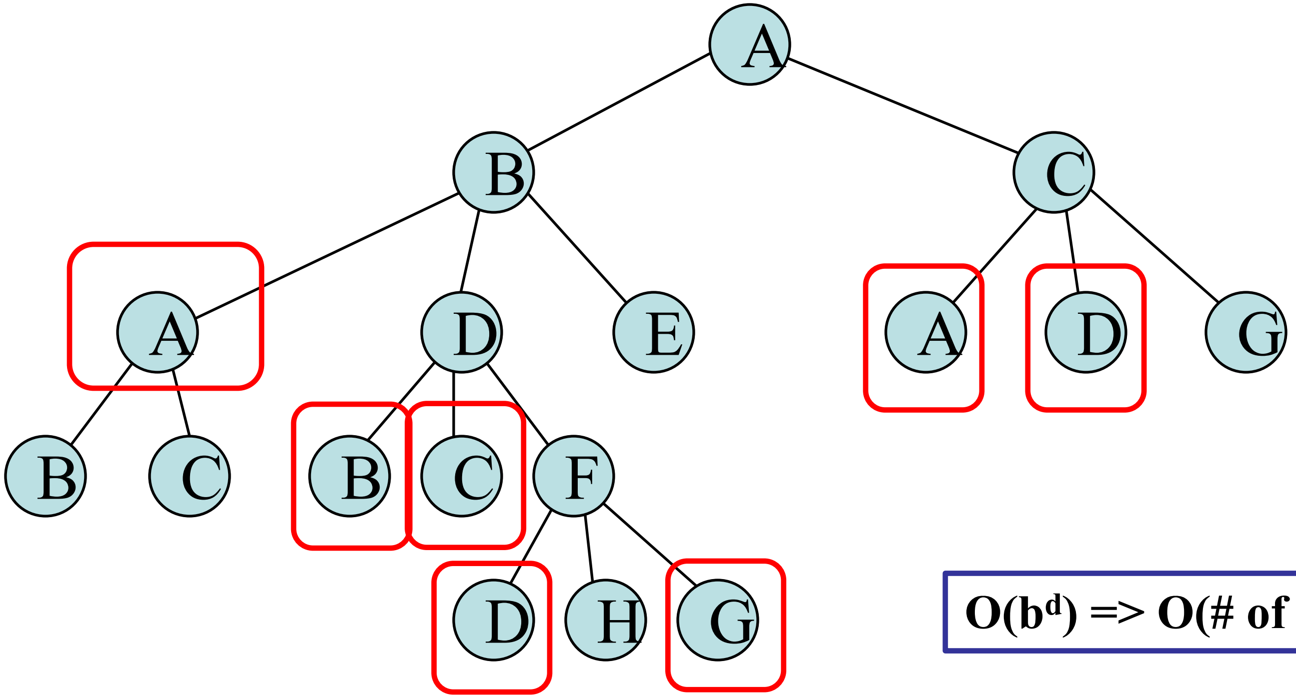
Search Tree => Search Graph

Dynamic programming (with book keeping)



Search Tree => Search Graph

Dynamic programming (with book keeping)



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

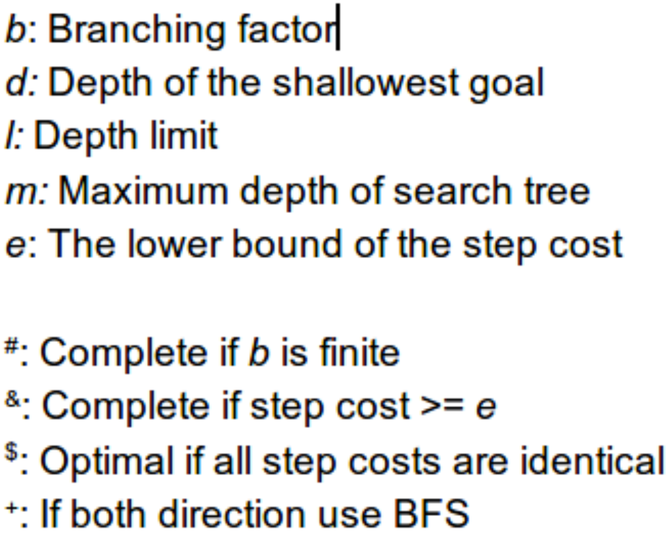
Graph Search vs Tree Search

- 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($b^{1+[C^*/e]}$)	O(b^m)	O(b^l)	O(b^d)	O($b^{d/2}$)
Space	O(b^d)	O($b^{1+[C^*/e]}$)	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

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

(Section 3.4.6 in the AIMA book.)

Practical note about search algorithms

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!

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?

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
    - $g(\mathit{node}) = 12$
  - 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"

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 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)

A* Search (Pronounced “A-Star”)

- Uniform-cost search minimizes ***g(n)*** (“past” cost)
- Greedy search minimizes ***h(n)*** (“expected” or “future” cost)

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 $f(n) = \underline{g(n)} + \underline{h(n)}$
  - Accounts for the "past" and the "future"
  - Estimates the cheapest solution (complete path) through node ***n***

A* Search (Pronounced "A-Star")

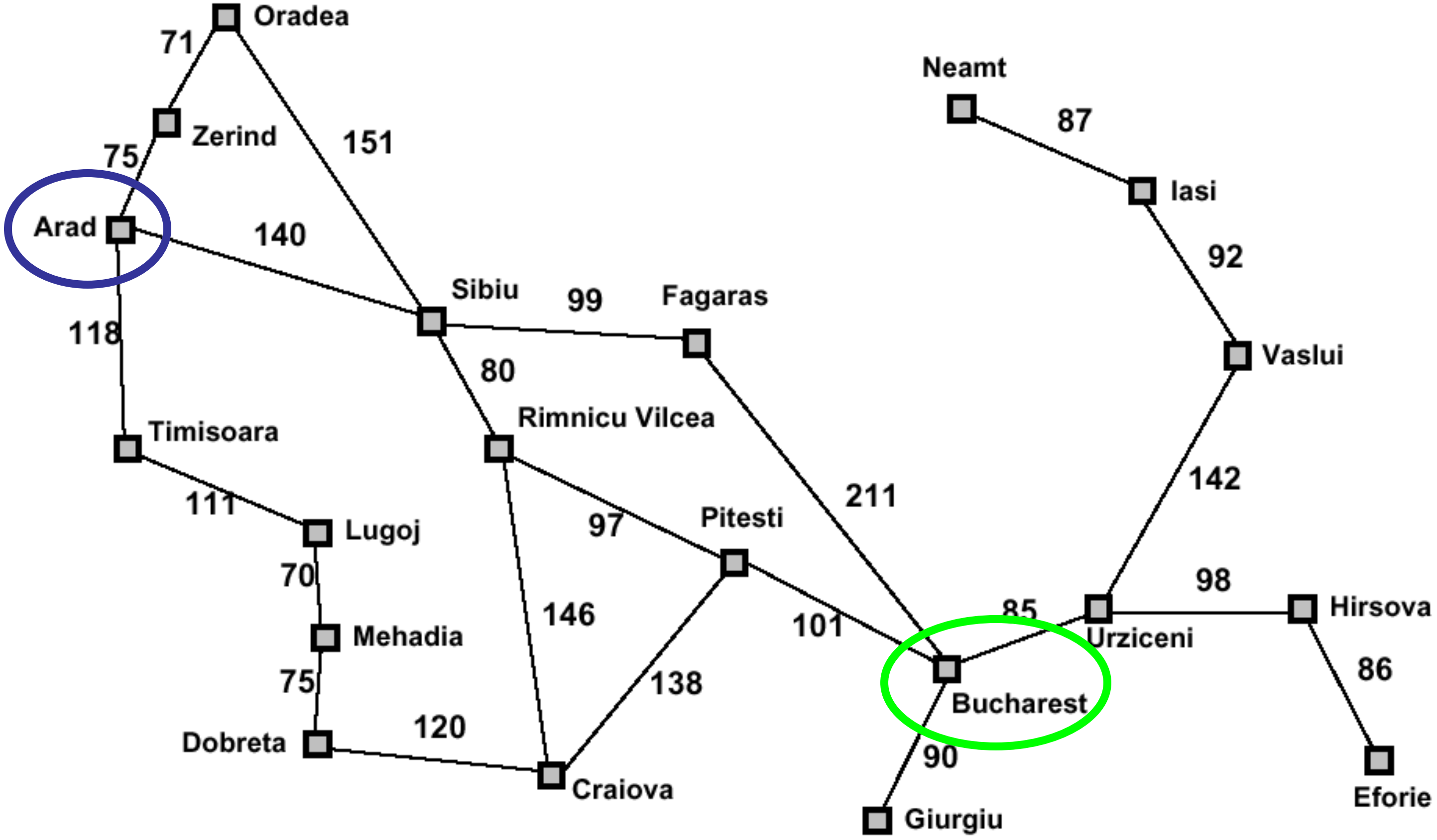
- Uniform-cost search minimizes ***g(n)*** ("past" cost)

- 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

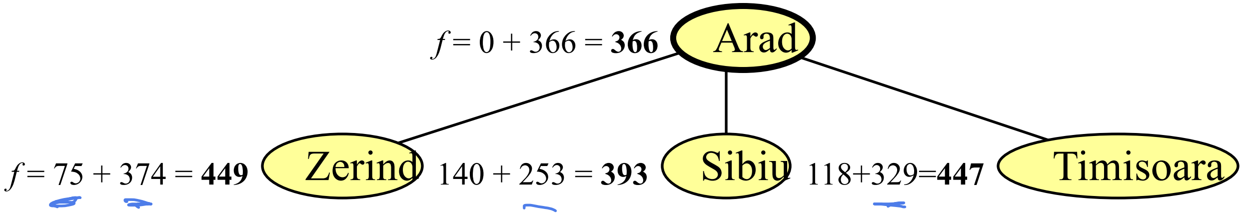
$f(n) = g(n) + h(n)$

A* Example

A* Example

f = 0 + 366 = **366** Arad

A* Example



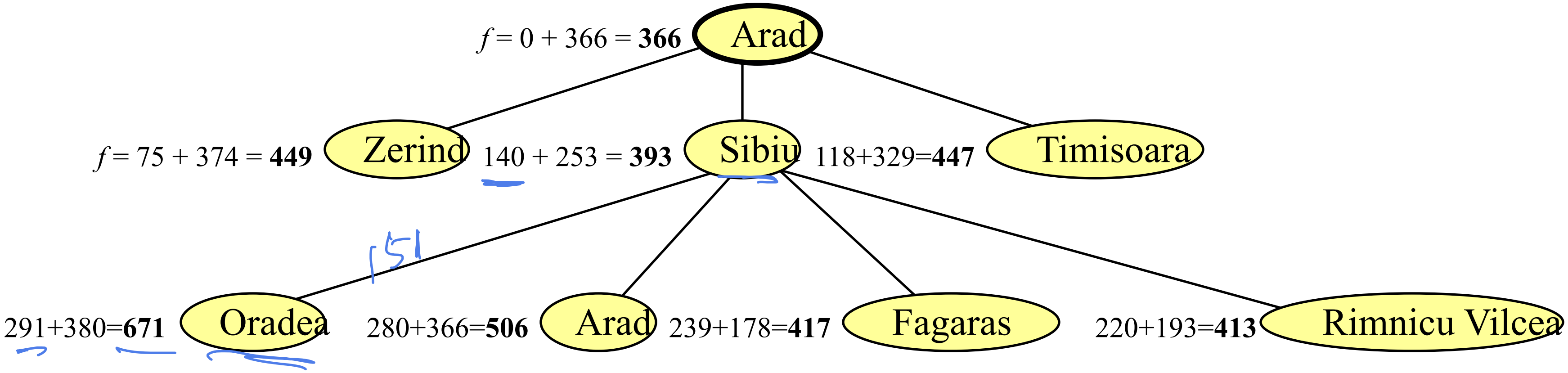
$f = 0 + 366 =$ **366** Arad

$f = 75 + 374 =$ **449** Zerind

$140 + 253 =$ **393** Sibiu

$118+329=$**447** Timisoara

A* Example



$f = 0 + 366 =$ **366** Arad

$f = 75 + 374 =$ **449** Zerind

140 + 253 = **393** Sibiu

118+329=**447** Timisoara

151

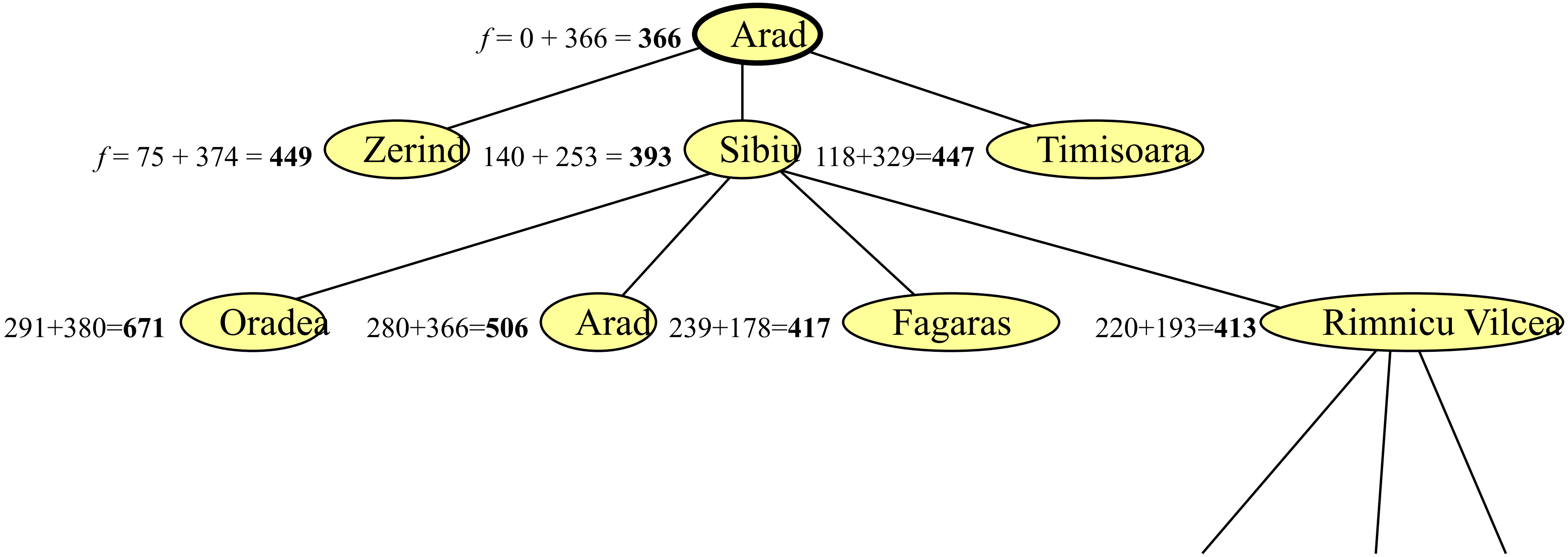
291+380=**671** Oradea

280+366=**506** Arad

239+178=**417** Fagaras

220+193=**413** Rimnicu Vilcea

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**

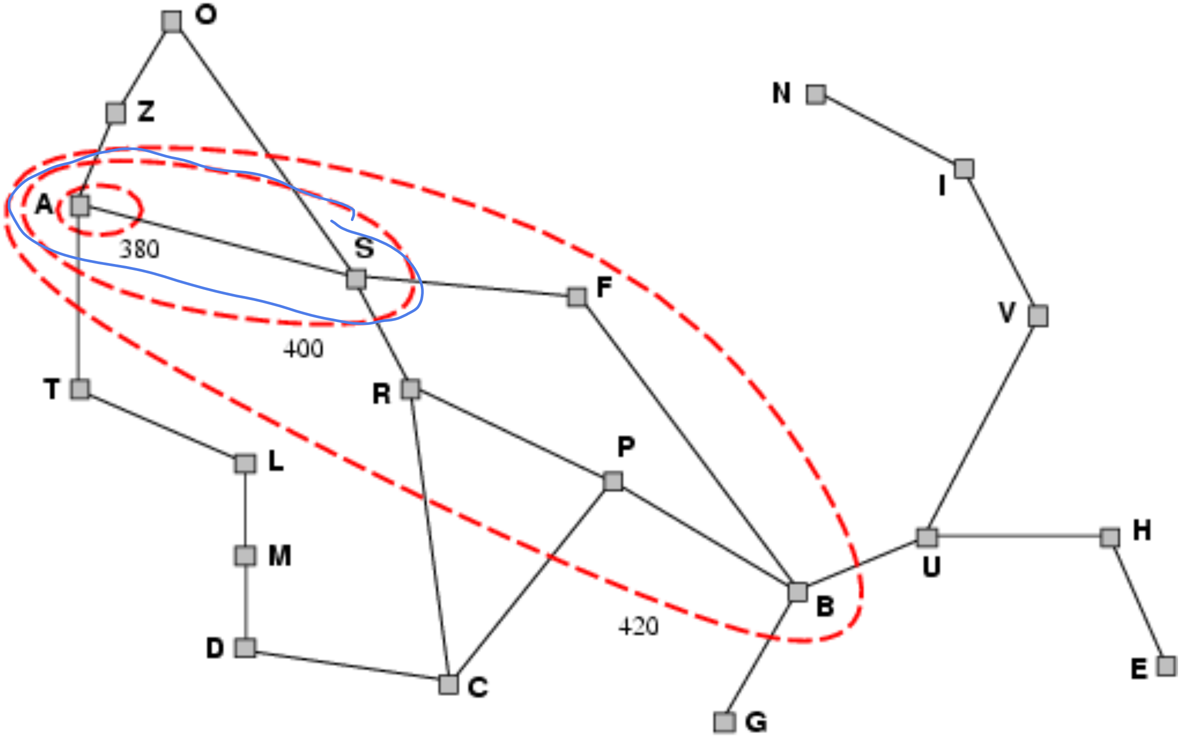
When does A* search "work"?

- 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 the best solution passing through node *n*

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

Optimality of A* with an Admissible h

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

Optimality of A* with an Admissible h

- Let OPT be the optimal path cost.
  - All non-goal nodes on this path have f ≤ 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.

Optimality of A* with an Admissible h

- Let OPT be the optimal path cost.
  - All non-goal nodes on this path have f ≤ 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)*.

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?

- Complete?

- Time complexity?

- Space complexity?

A* Search with an Admissible h

- Optimal? Yes
- Complete?
- Time complexity?
- Space complexity?

A* Search with an Admissible h

• Optimal?	Yes
• Complete?	Yes
• Time complexity?	
• Space complexity?	

A* Search with an Admissible h

Optimal?	Yes
Complete?	Yes
Time complexity?	Exponential; better under some conditions
Space complexity?	

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

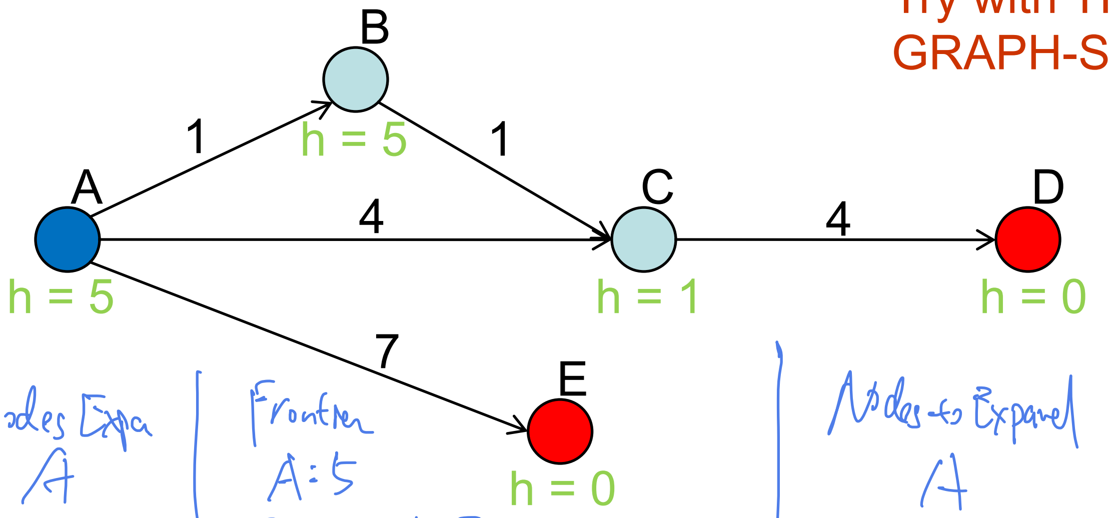
- Tree Search
  - We might repeat some states
  - But we do not need to remember states

- 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



Set	Nodes Expa	Frontier
A	A	A:5
C ←	C	B:6, C:5, E:7
B	B	B:6, E:7, D:8
	E	E:7, D:8

Solution A→E
Cost = 7

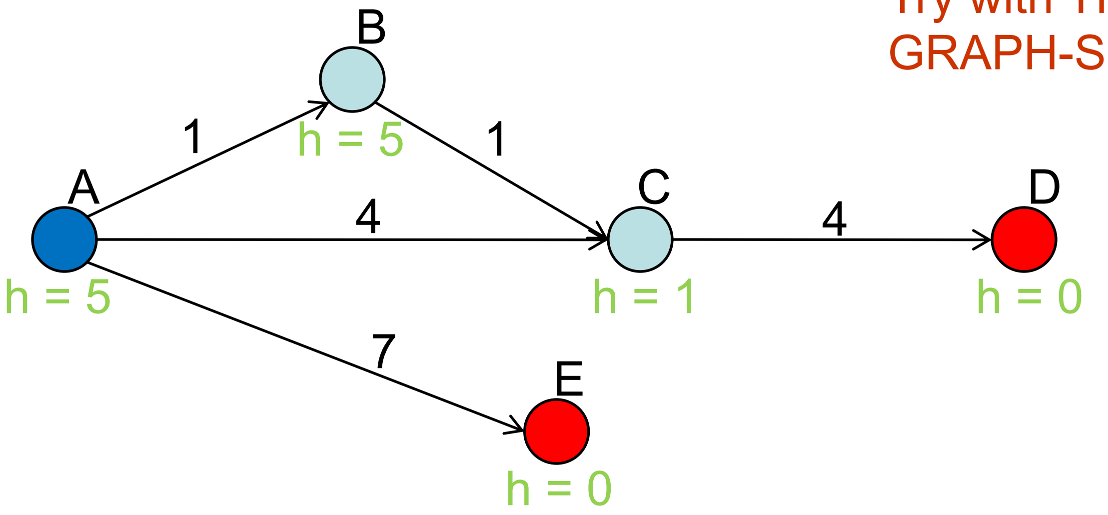
Nodes to Expand	Frontier
A	A:5
C	B:6, C:5, E:7
B	B:6, E:7, D:8
C	E:7, D:8, C:3
D	E:7, D:8, D:6

Solution: A→B→C→D
Cost: 6

Avoiding Repeated States using A* Search

- Is GRAPH-SEARCH optimal with A*?

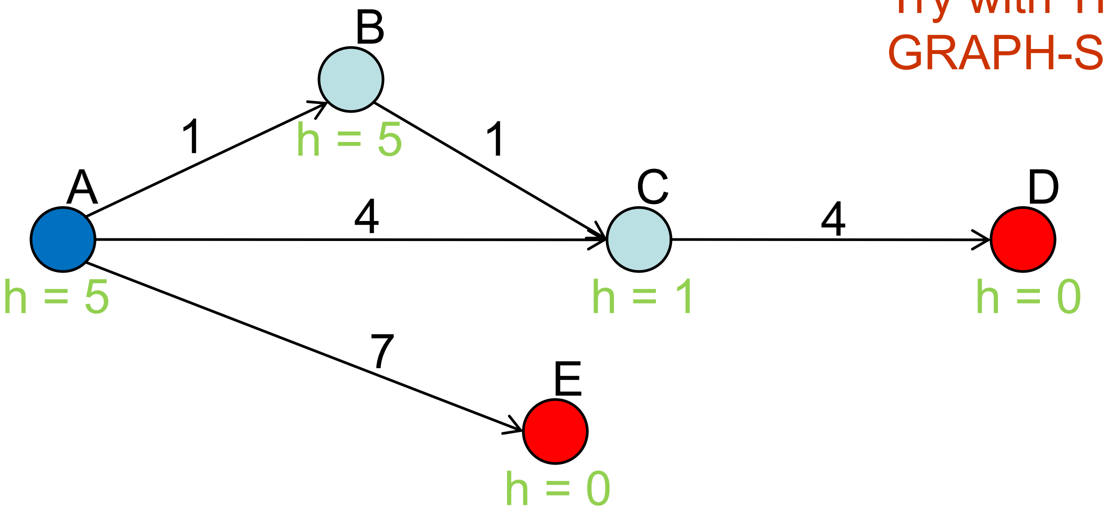
Try with TREE-SEARCH and GRAPH-SEARCH



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

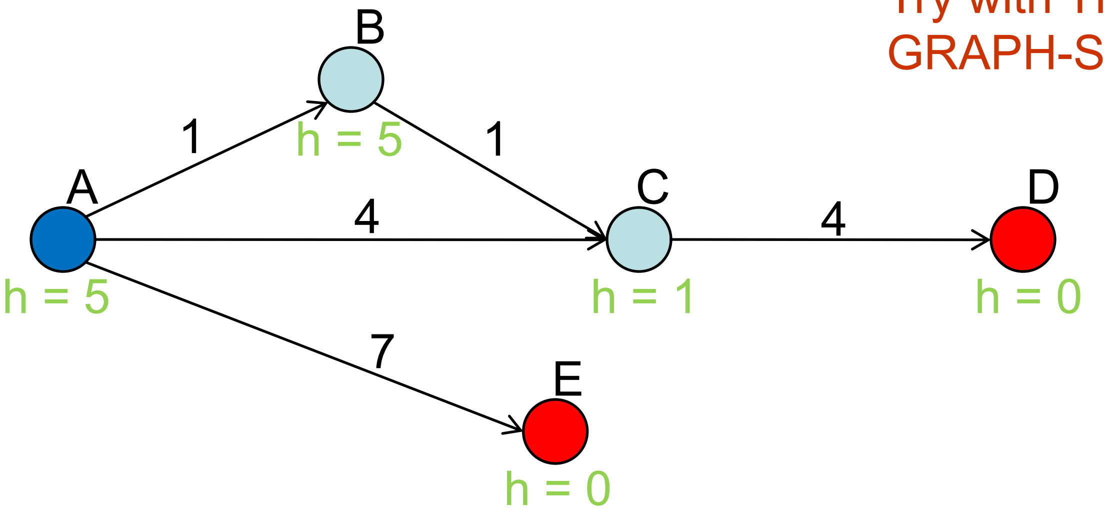
- Is GRAPH-SEARCH optimal with A*?



Try with TREE-SEARCH and GRAPH-SEARCH

Avoiding Repeated States using A* Search

- Is GRAPH-SEARCH optimal with A*?

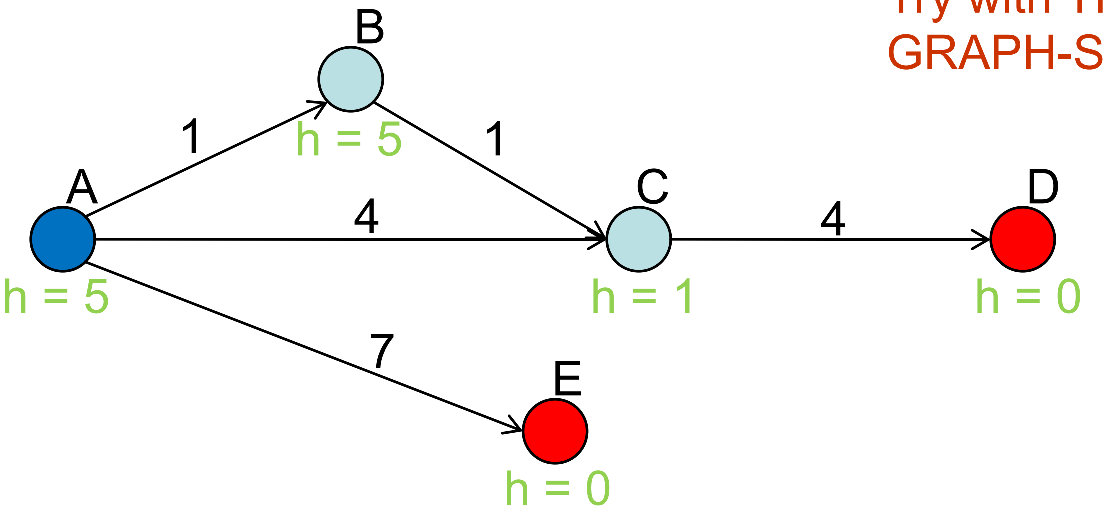


Try with TREE-SEARCH and GRAPH-SEARCH

Solution 1: Remember all paths: Need extra bookkeeping

Avoiding Repeated States using A* Search

- Is GRAPH-SEARCH optimal with A*?



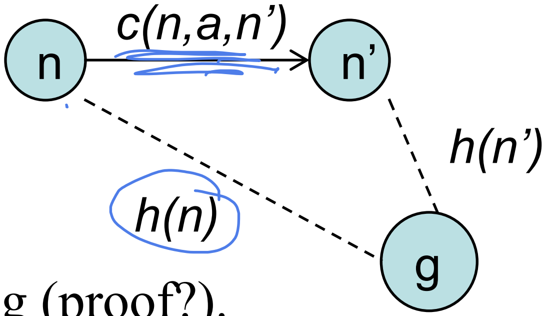
Try with TREE-SEARCH and GRAPH-SEARCH

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?). Exercise
  - 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.



$h(n) \leq h(n') + c(n,a,n')$

Remainder of the lecture

- Examples
- Choosing heuristics
- Games and Minimax Search

Heuristics

- What’s a heuristic for
  - Driving distance (or time) from city A to city B ? $\frac{\text{Straight line dist.}}{\text{speed}}$
  - 8-puzzle problem ? $\sum_t \text{dist}(t, g_t)$ ↑ Manhattan Distance
  - M&C ? # of ppl on the Left bank ≥
  - PACMAN?

Heuristics

- What's a heuristic for
  - Driving distance (or time) from city A to city B ?
  - 8-puzzle problem ?
  - M&C ?
  - PACMAN?

- **Admissible** heuristic
  - Does not overestimate the cost to reach the goal
  - "Optimistic"

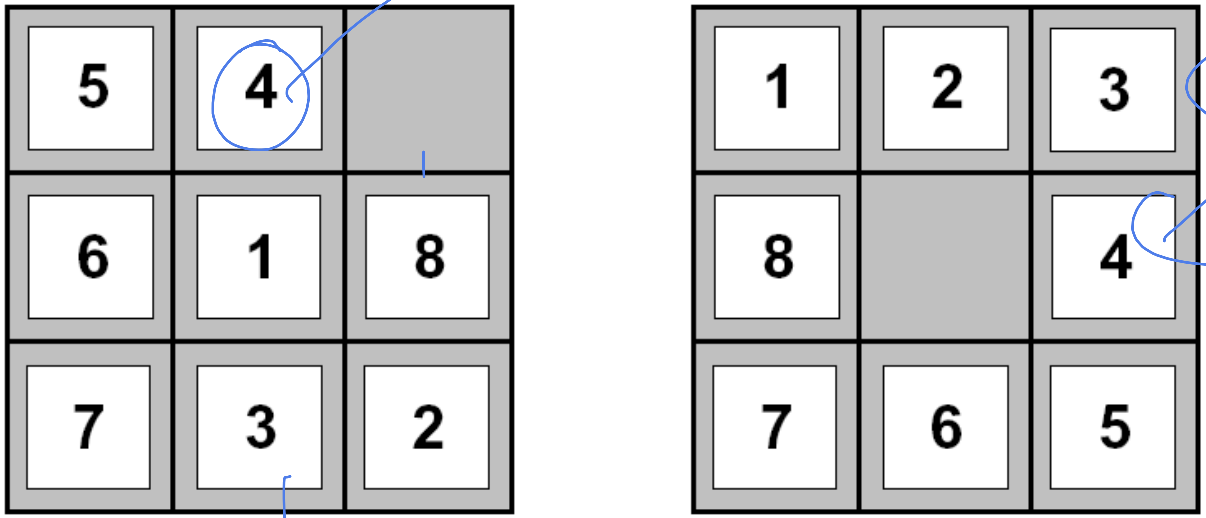
Heuristics

- What’s a heuristic for
  - Driving distance (or time) from city A to city B ?
  - 8-puzzle problem ?
  - M&C ?
  - PACMAN?

- **Admissible** heuristic
  - Does not overestimate the cost to reach the goal
  - “Optimistic”
- Are the above heuristics admissible? Consistent?

Example: 8-Puzzle

2+3+....



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

Effective Branching Factor

- Though <u>informed</u> 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

Effective Branching Factor

- Though <u>informed</u> 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
  - $N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$

$$b^* \simeq O(N^{\frac{1}{d}})$$

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

# of displacement

total Manhattan Dist

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.

Memory Bounded Search

- Memory, not computation, is usually the limiting factor in search problems
  - Certainly true for A* search
- Why? What takes up memory in A* search?

- Solution: Memory-bounded A* search
  - Iterative Deepening A* (IDA*)
  - Simplified Memory-bounded A* (SMA*)
  - (Read the textbook for more details.)

  - Very popular choice: Beam Search (Application in Decoding for Large Language Model!)

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.