复旦大学大数据学院
School of Data Science, Fudan University
魏忠钰

Informed Search

March 21st, 2017

AI News

机器之心 原创
2018年3月15日 10:57

中到英新闻翻译媲美人类，

> 今日，微软研究团队表示，微软和微软亚研创造了首
> 翻译系统。黄学东告诉机器之心，他们采用专业人类
> 性，且新系统相比于现存的机器翻译系统有非常大的
> 得了至少和专业翻译人员相媲美的效果。

微软亚洲与美国实验室的研究者称，其中英新闻机器翻
达到了人类水平。该测试集由来自业界和学界的团队共
证结果既准确又能达到人类水平，该团队聘请了外部双
的人类译文。

微软语音、自然语言与机器翻译的技术负责人黄学东称
碑。他对机器之心说：「我们的新系统相比之前的翻译
破，是一个历史性的里程碑。」

「机器翻译达到人类水平是我们所有人的梦想，」黄说

黄学东也领导了最近在对话语音识别任务中达到人类水
碑尤其令人高兴，因为它可以帮助人们更好地理解彼此

8:30 0.00K/s 99%

推文

Sebastian Ruder
@seb_ruder

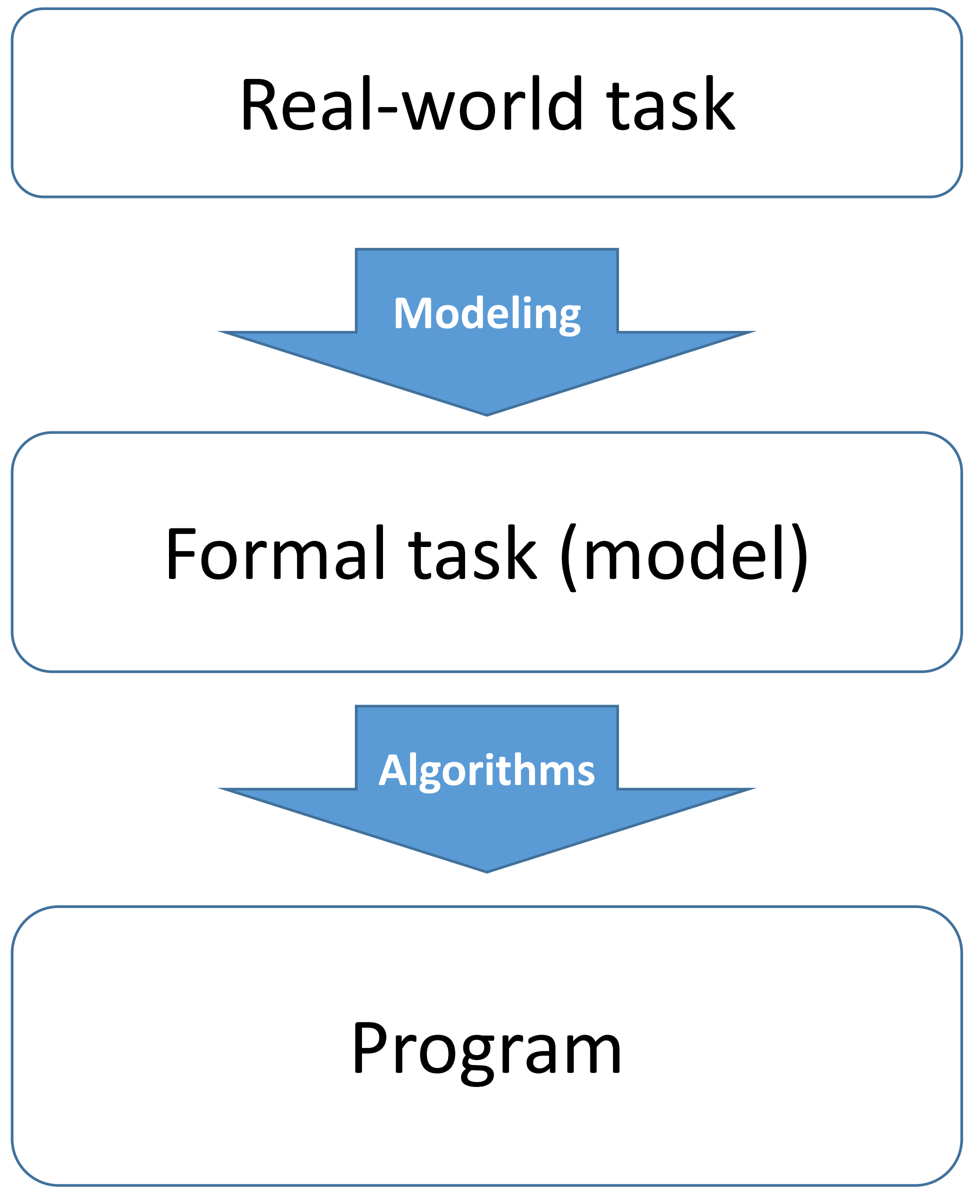
Microsoft reports that they've achieved human parity on Chinese-to-English translation (27.40 BLEU; 1 BLEU better than best result of WMT 2017). Model is a Transformer (NIPS 2017) + Dual Learning (NIPS 2016) + Deliberation Nets (NIPS 2017). microsoft.com/en-us/research...

由 Microsoft 翻译自英文

微软报道说，他们已经实现了汉英翻译中的人的平等 (27.40 蓝，1 蓝比最好的结果 WMT 2017)。模型是一个变压器 (2017) + 双学习 (奶头 2016) + 审议网 (奶头 2017)。microsoft.com/en-us/

发布回复推文

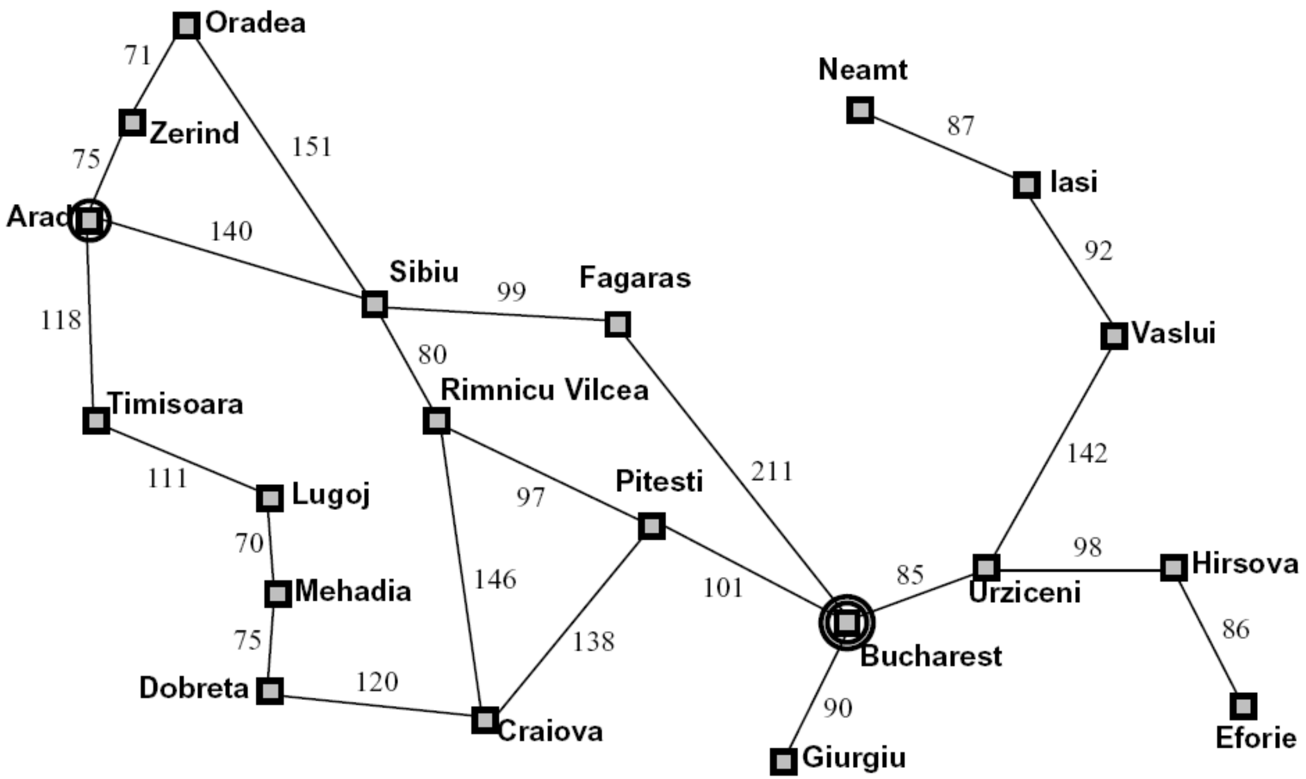
How to tackle these AI tasks?



Search problems

- **A search problem consists of:**
  - **A state space**
  - **A successor function (with actions, costs)**
  - **A start state**
  - **A goal test**

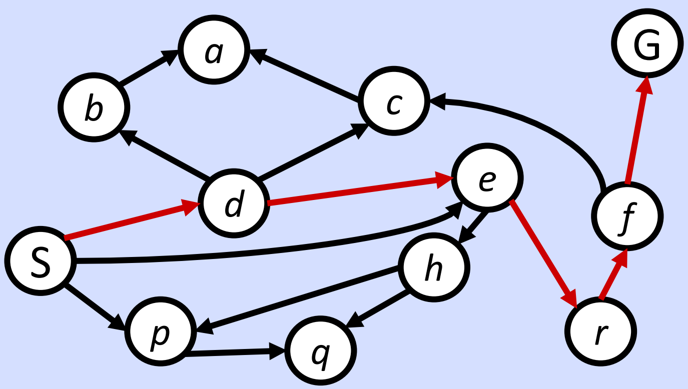
Search problems: Traveling in Romania



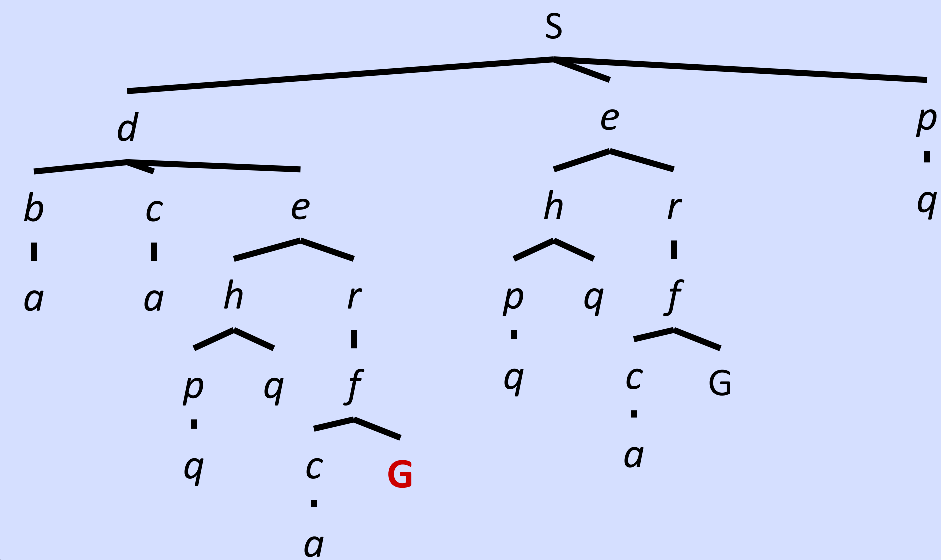
- State space: Cities
- Successor function: Roads: Go to adjacent city with cost = distance
- Start state: Arad
- Goal test: Is state == Bucharest?

State Space Graphs vs. Search Trees

State Space Graph



Search Tree



- *Each NODE in the search tree is an entire PATH in the state space graph.*
- *A search tree might include redundant structures.*

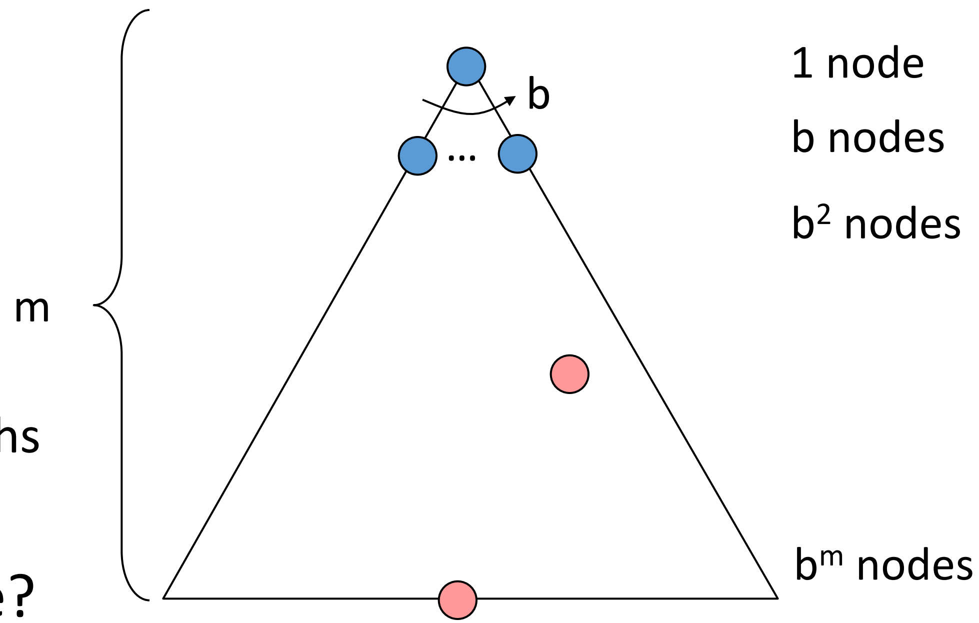
Recap

- **Uninformed Search (无信息搜索策略)**
  - **Depth-First Search**
  - **Breadth-First Search**
  - **Uniform-Cost Search**

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - s is the solutions at various depths

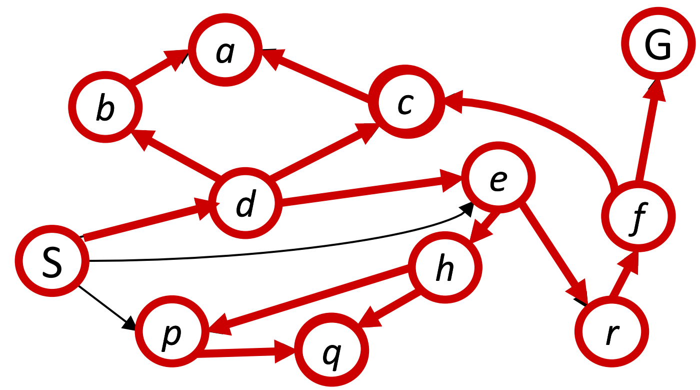


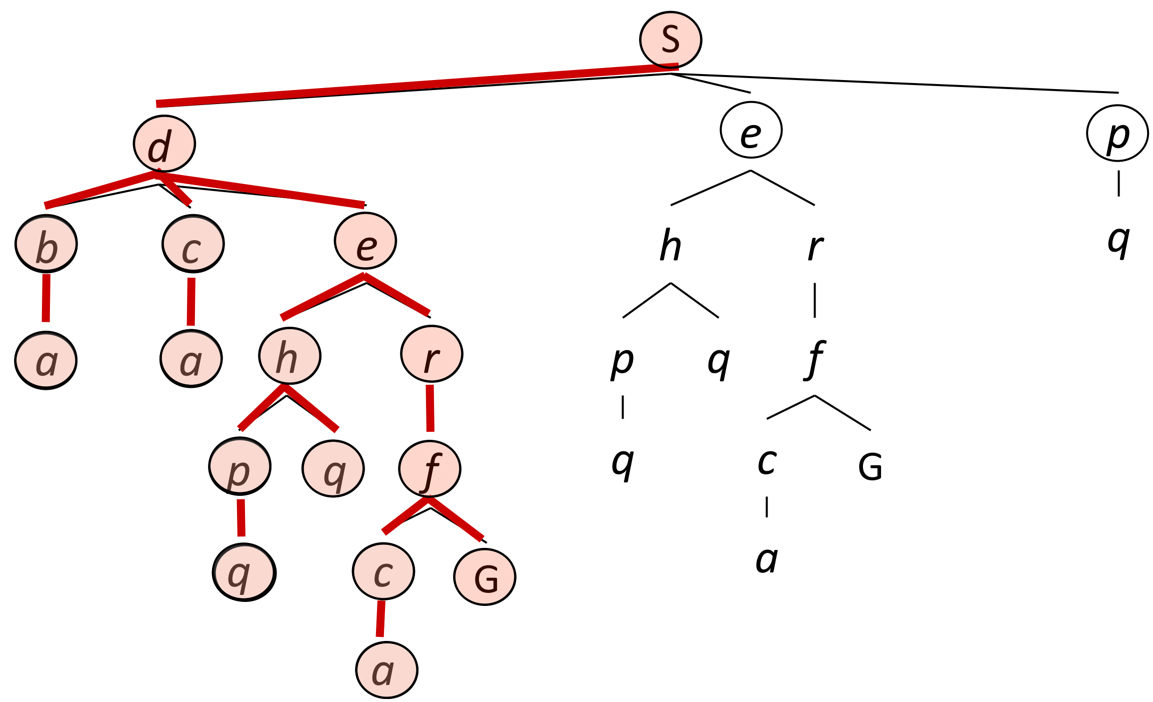
- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots. b^m = O(b^m)$

Depth-First Search

Strategy: expand a deepest node first

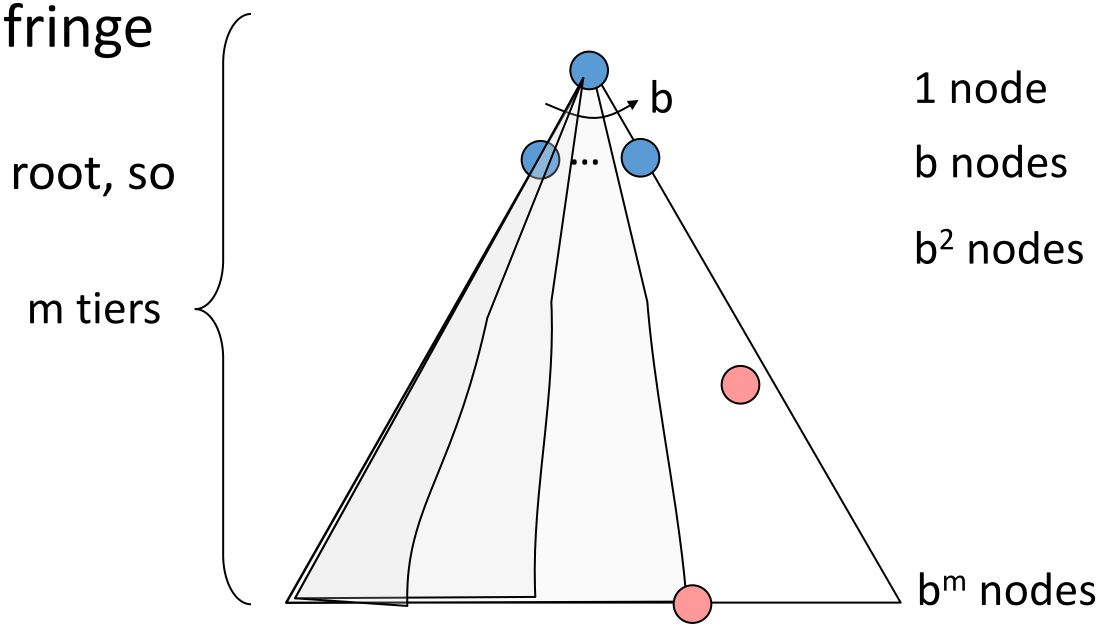
Implementation: Fringe is a LIFO stack





Depth-First Search (DFS) Properties

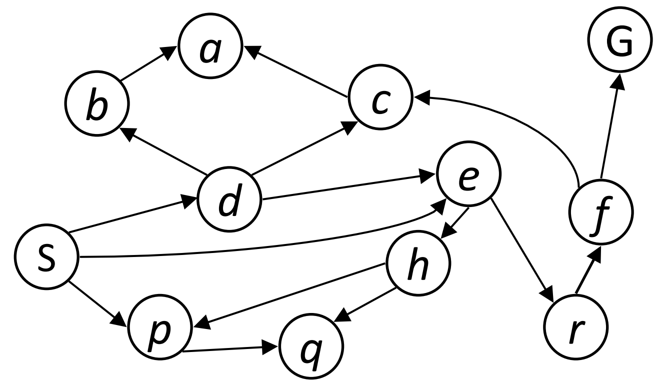
- What nodes DFS expand (Time Complexity)?
  - Some left prefix of the tree.
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take (Space Complexity)?
  - Only has siblings on path to root, so $O(bm)$
- Is it complete?
  - No, m can be infinite
- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost

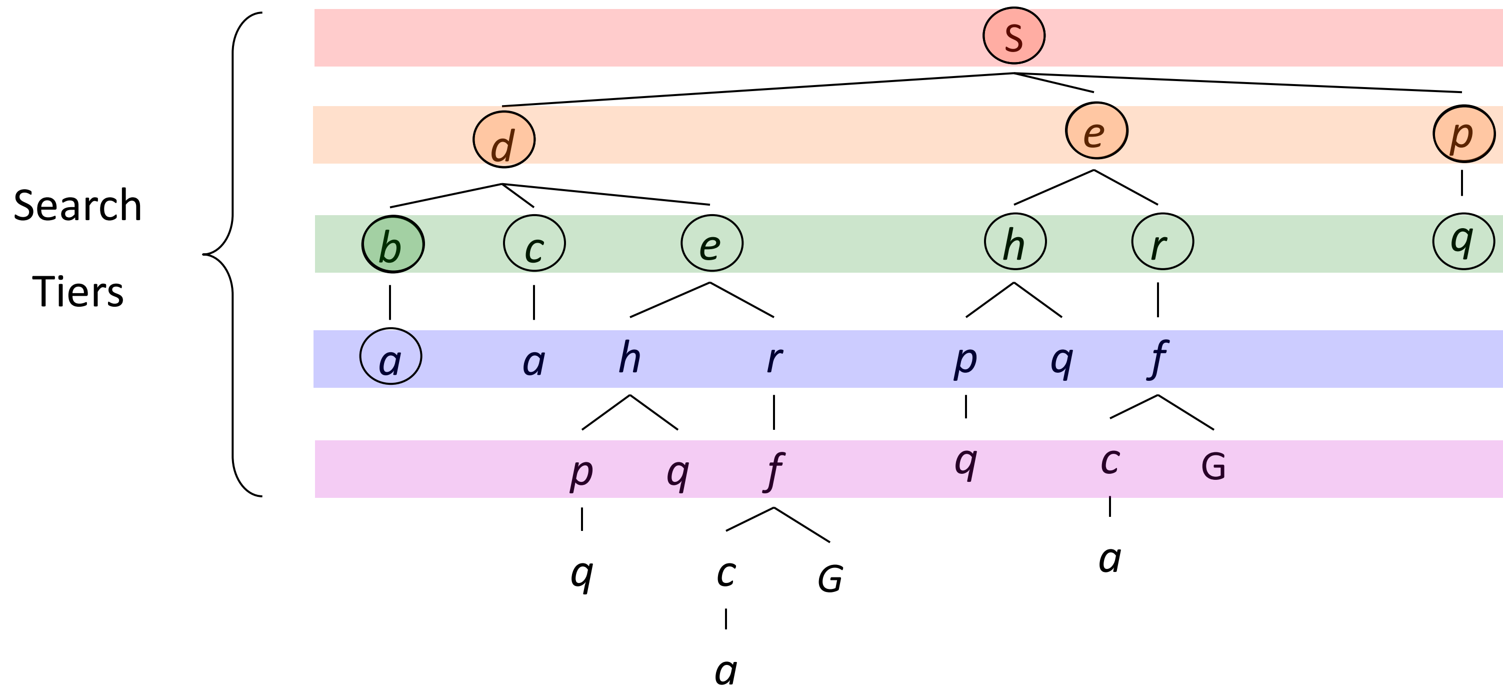


Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue





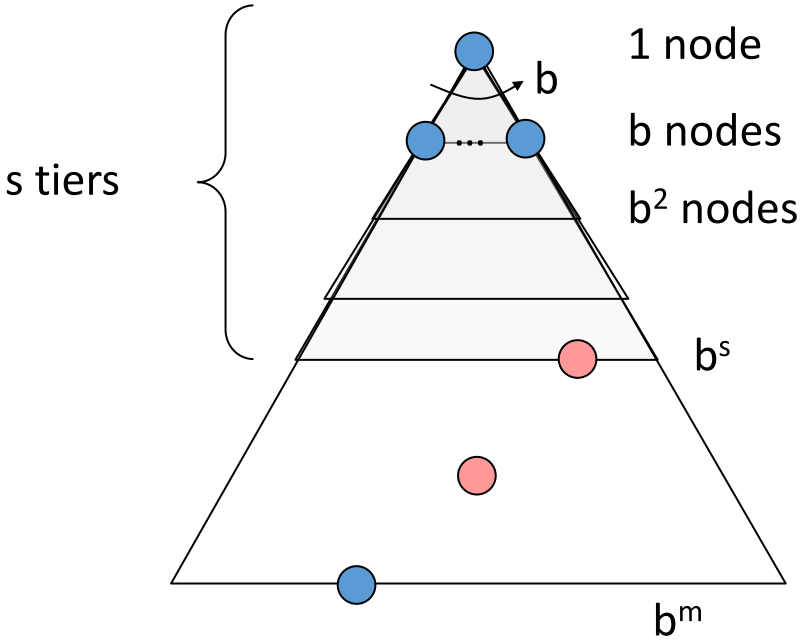
Breadth-First Search (BFS) Properties

- What nodes does BFS expand (Time Complexity)?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$

- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$

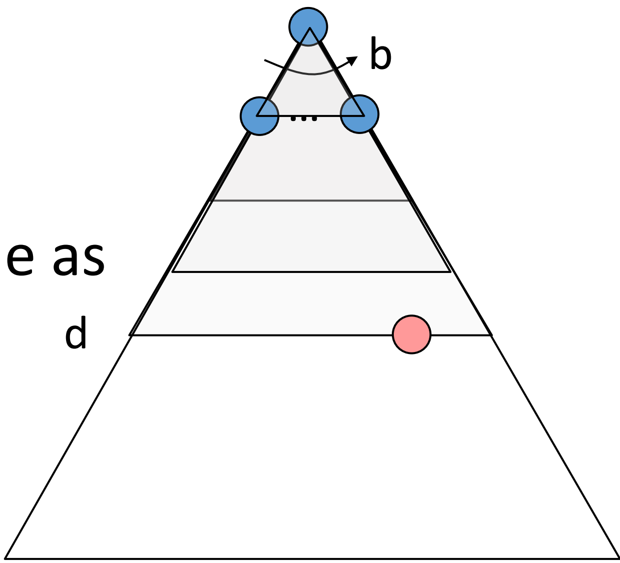
- Is it complete?
  - yes

- Is it optimal?
  - Yes (if the cost is equal per step)

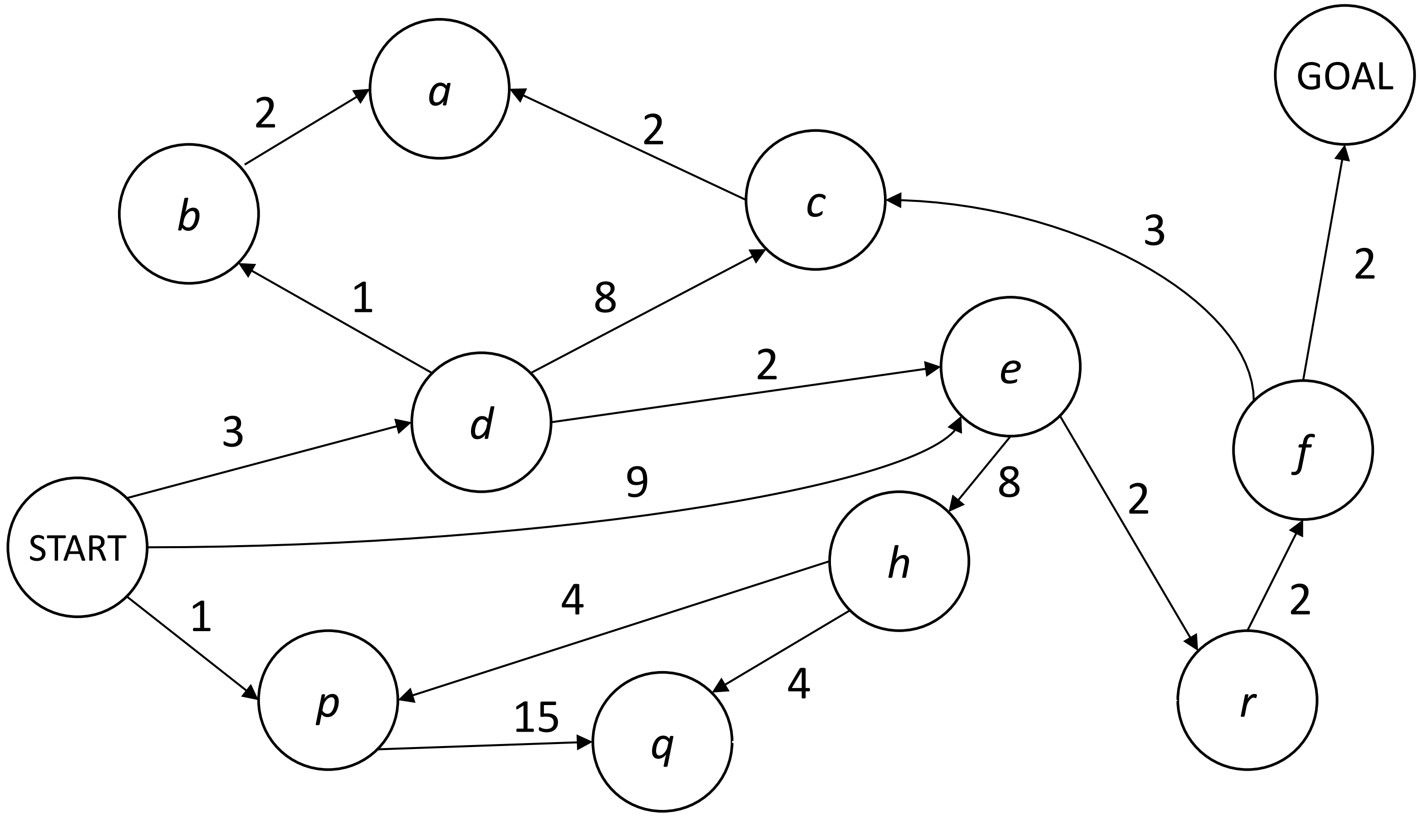


Iterative Deepening Search (IDS)

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..
- How many nodes does BFS expand?
  - $O(b^d)$
- How much space does the fringe take?
  - $O(bd)$
- Although the time complexity of IDS is the same as BFS, it expands many more nodes.
- $b + (b + b^2) + (b + b^2 + b^3)\ldots$
- $= db^1 + (d-1)b^2 + (d-2)b^3 + (d-3)b^4 \ldots + b^d$
- $= O(db^1 + (d-1)b^2 + (d-2)b^3 + (d-3)b^4 \ldots + b^d) = O(b^d)$



Cost-Sensitive Search

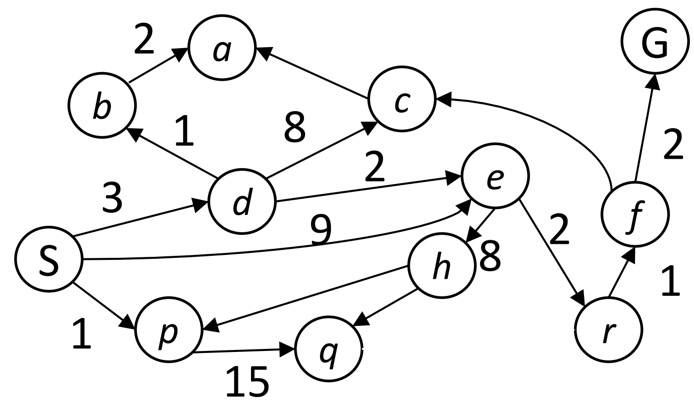


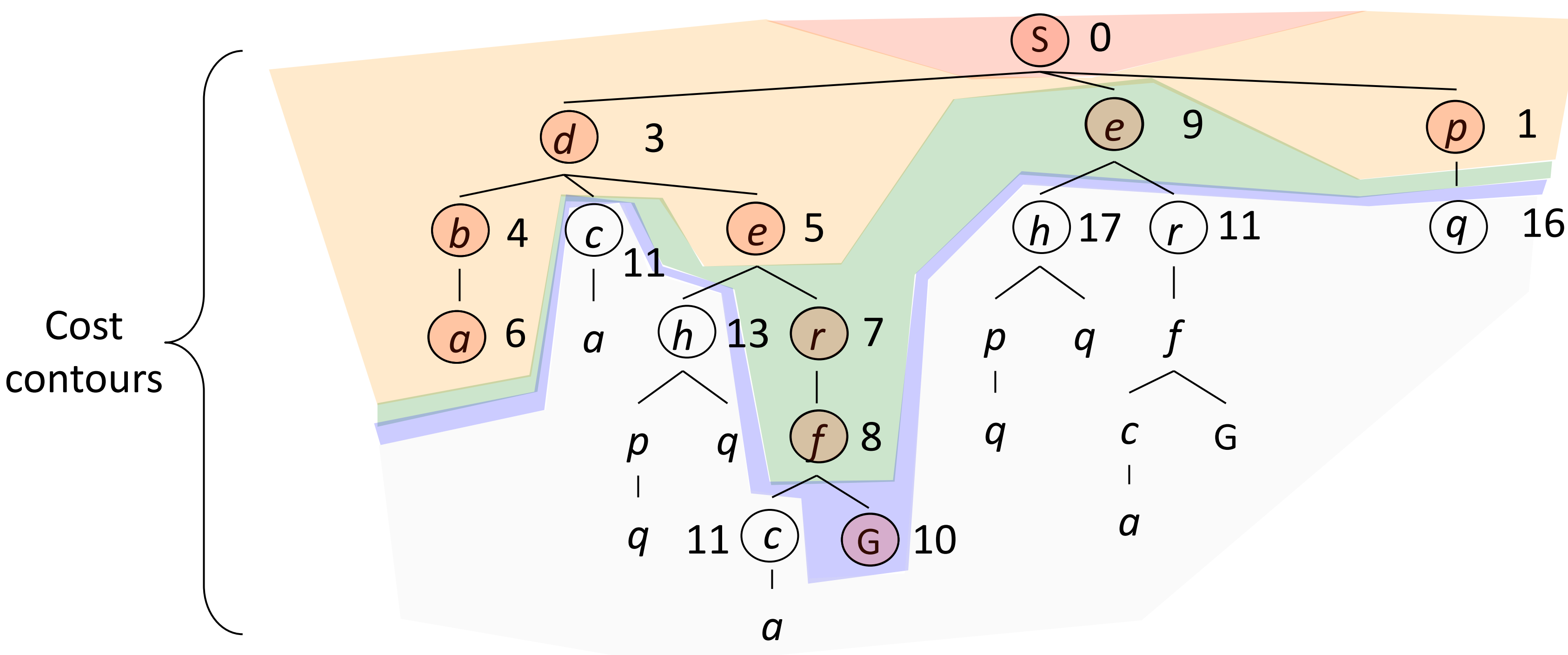
BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.

Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)





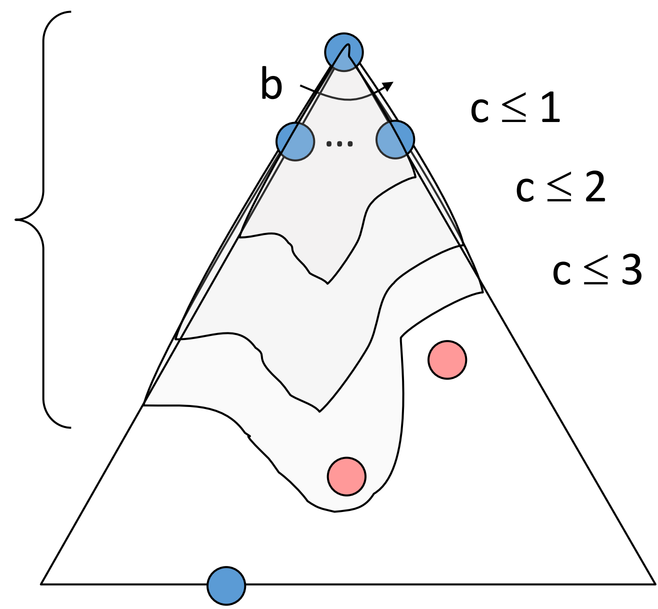
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that **solution costs C^*** and **arcs cost at least ε**, then the "effective depth" is roughly C^*/ε
  - Takes time O($b^{C^*/\varepsilon}$) (exponential in effective depth)

- How much space does the fringe take?
  - Has roughly the last tier, so O($b^{C^*/\varepsilon}$)

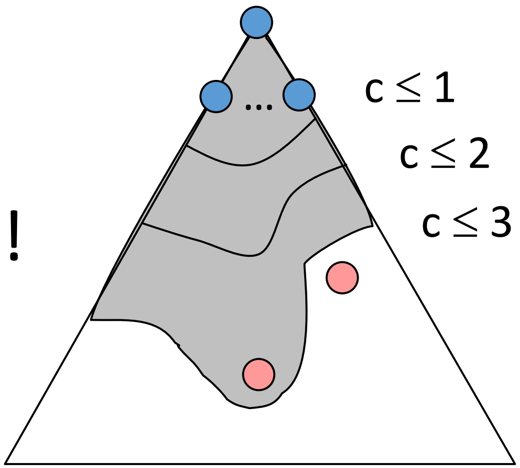
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!

- Is it optimal?
  - Yes!



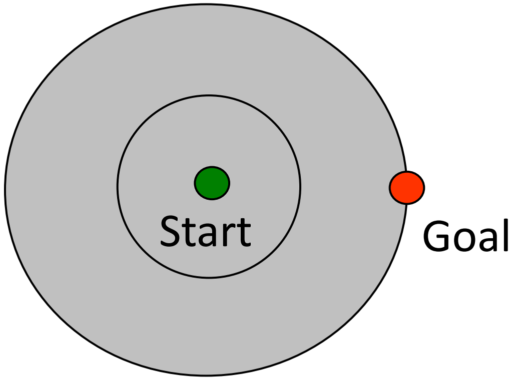
Uniform Cost Issues

- Remember: UCS explores increasing cost contours



- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every "direction"
  - No information about goal location



Comparison

Algorithm	Complete?	Optimal?	Time?	Space?
DFS	N	N	O(b^m)	O(bm)
BFS	Y	Y	O(b^d)	O(b^d)
IDS	Y	Y	O(b^d)	O(bd)
UCS	Y	Y	O($b^{C^*/\varepsilon}$)	O($b^{C^*/\varepsilon}$)

For BFS, Suppose the branching factor *b* is finite and step costs are identical;
d is the depth of the optimal solution;

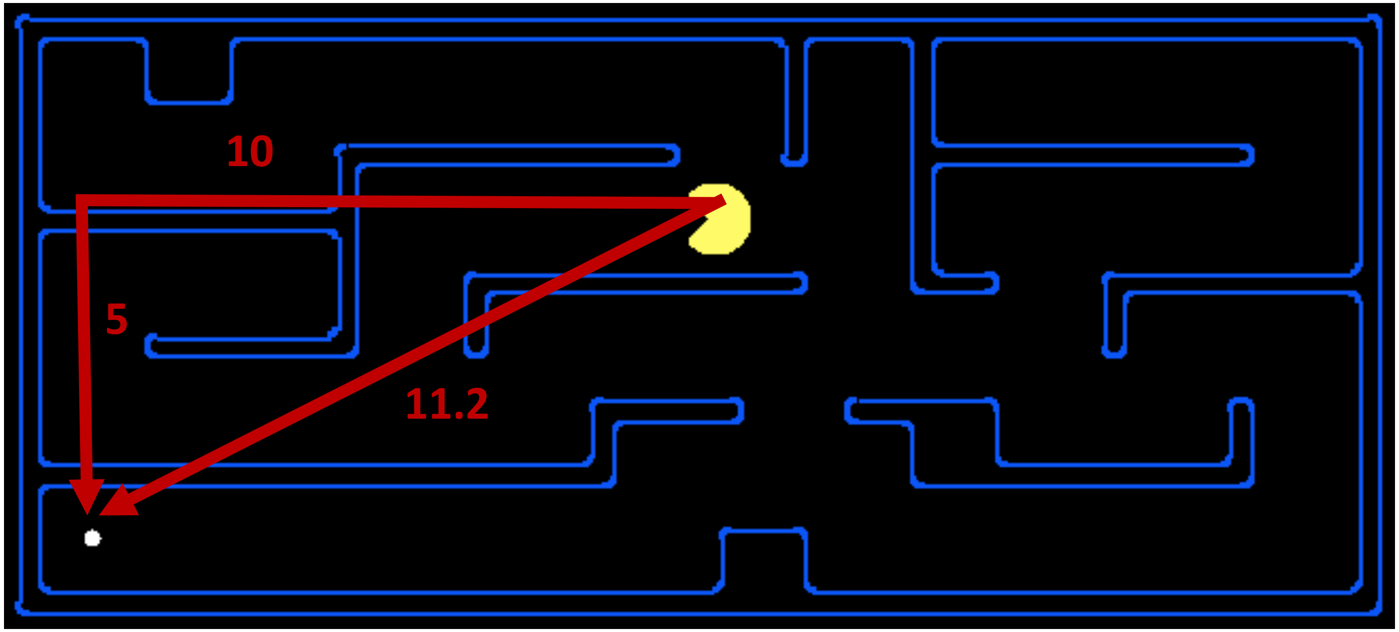
Informed Search

- In uninformed search, we never “look-ahead” to the goal. E.g., We don't consider the cost of getting to the goal from the end of the current path.

- Often we have some other knowledge about the merit of nodes.

Search Heuristics

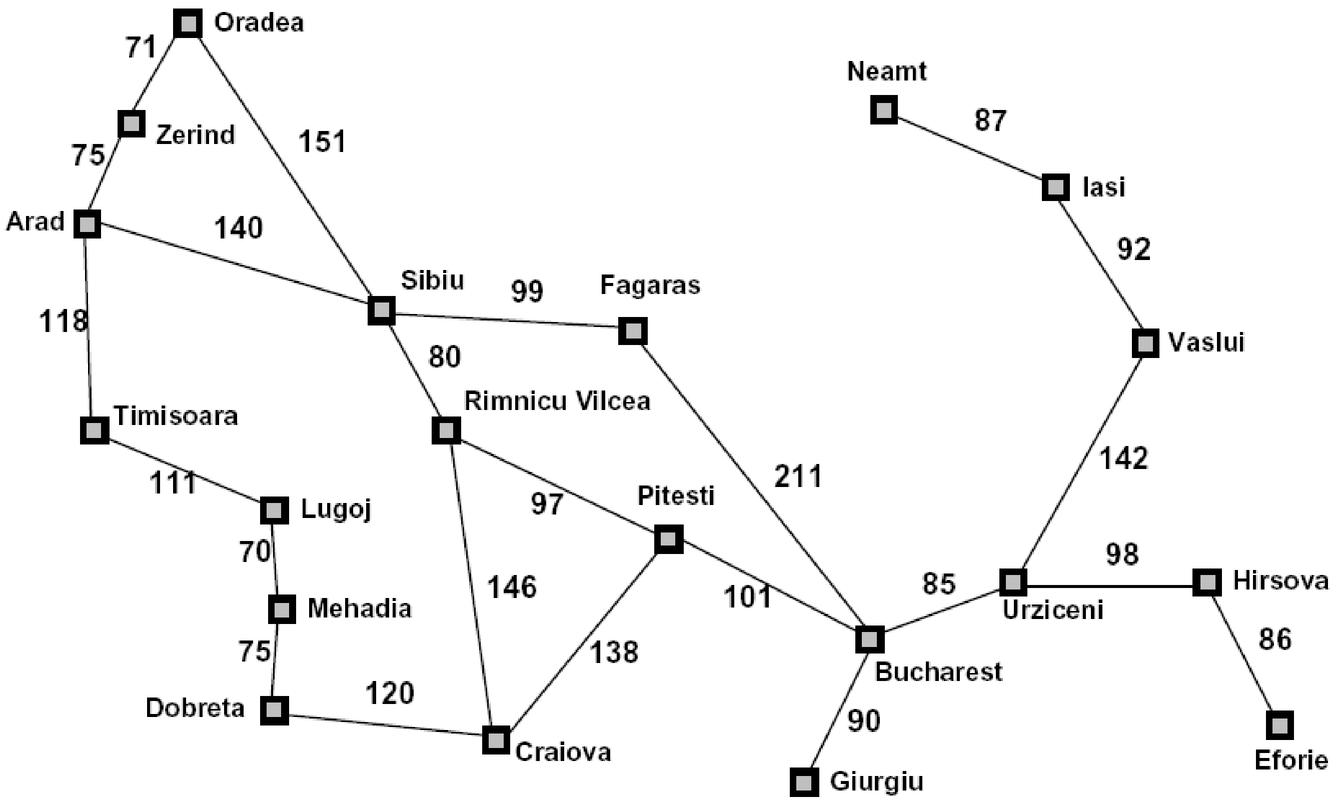
- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing



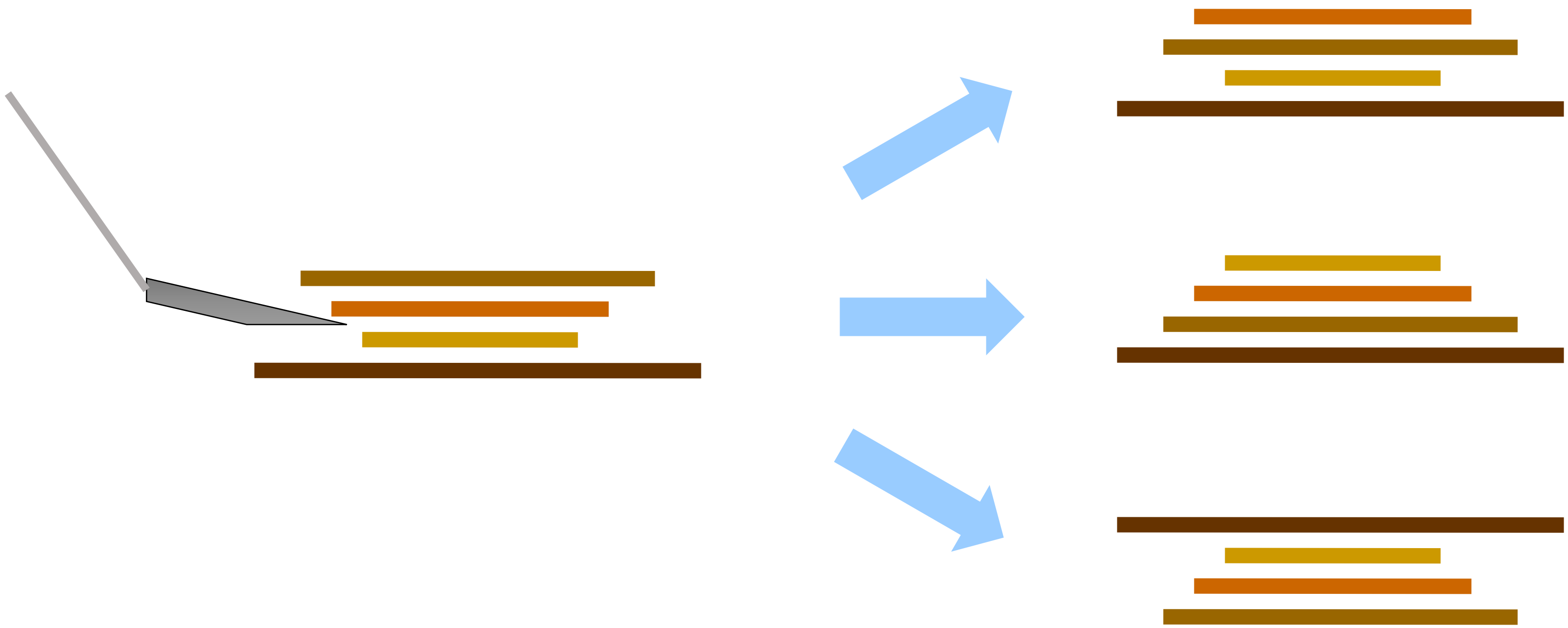
Manhattan distance: $|x_1 - x_2| + |y_1 - y_2|$

Euclidean distance: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Heuristic


Oradea
71
Neamt
Zerind
151
87
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
211
111
Pitesti
Lugoj
97
70
98
85
Hirsova
146
101
Urziceni
Mehadia
86
75
138
Bucharest
Dobreta
120
90
Craiova
Eforie
Giurgiu

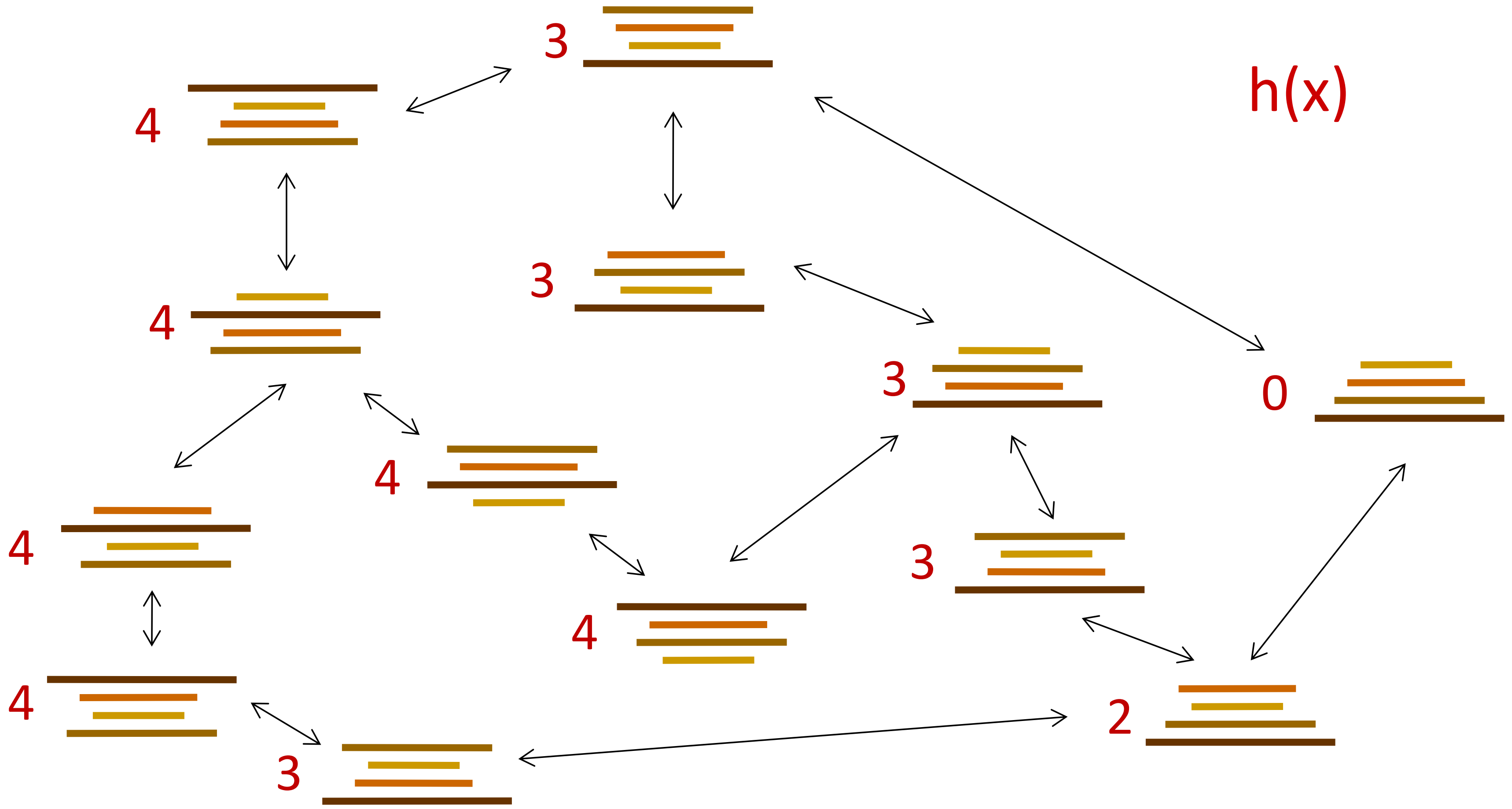
Example: Pancake Problem



Cost: Number of pancakes flipped

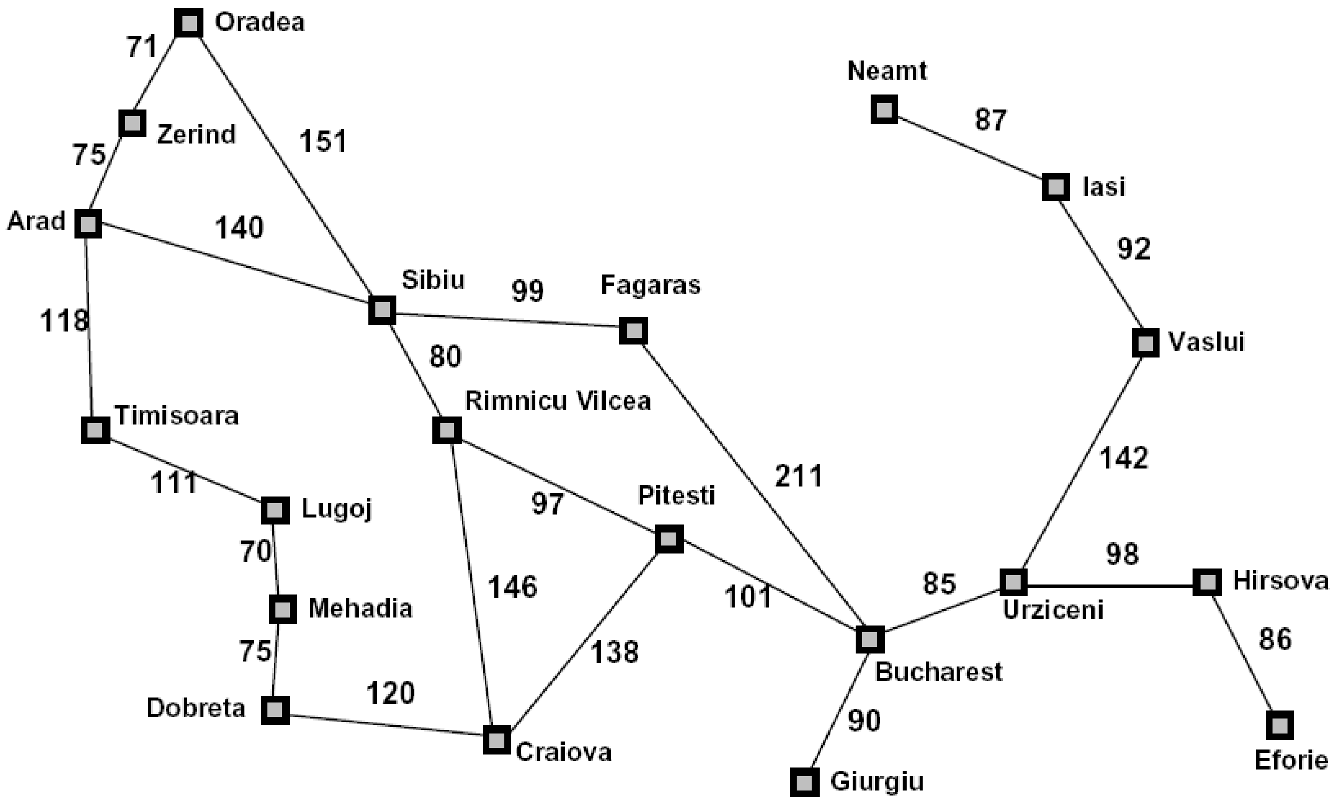
Example: Heuristic Function

Heuristic: the number of the largest pancake that is still out of place



Greedy Search

Example: Heuristic Function

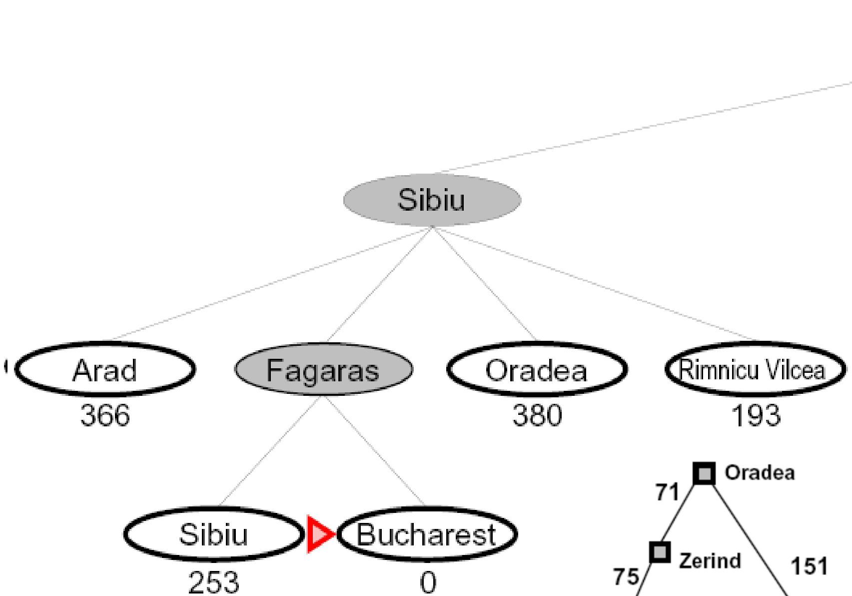


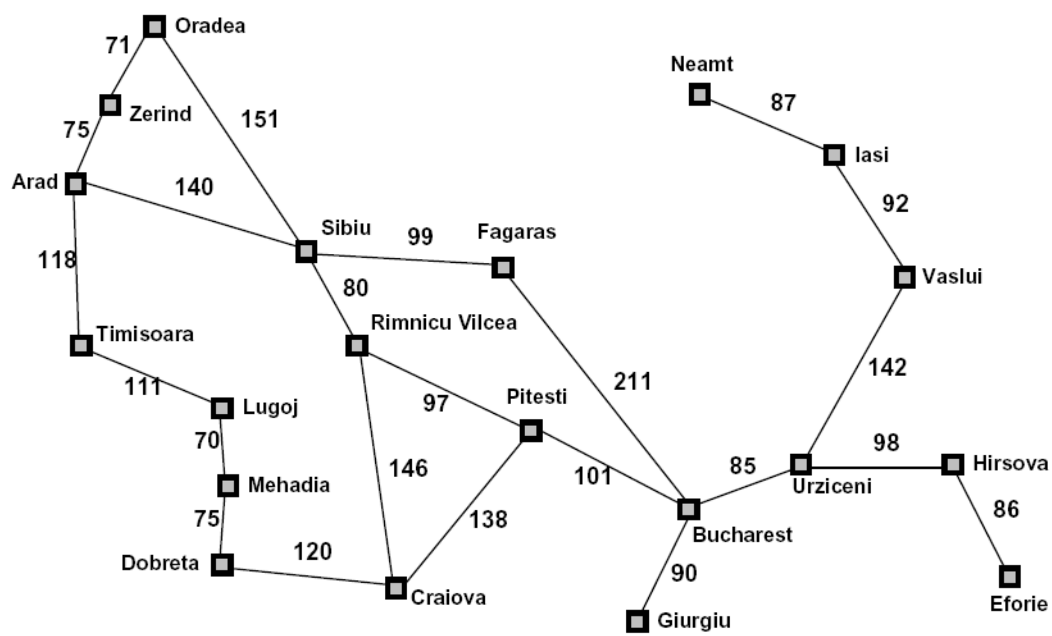
Straight−line distance
to Bucharest

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

Greedy Search

- Expand the node that seems closest…



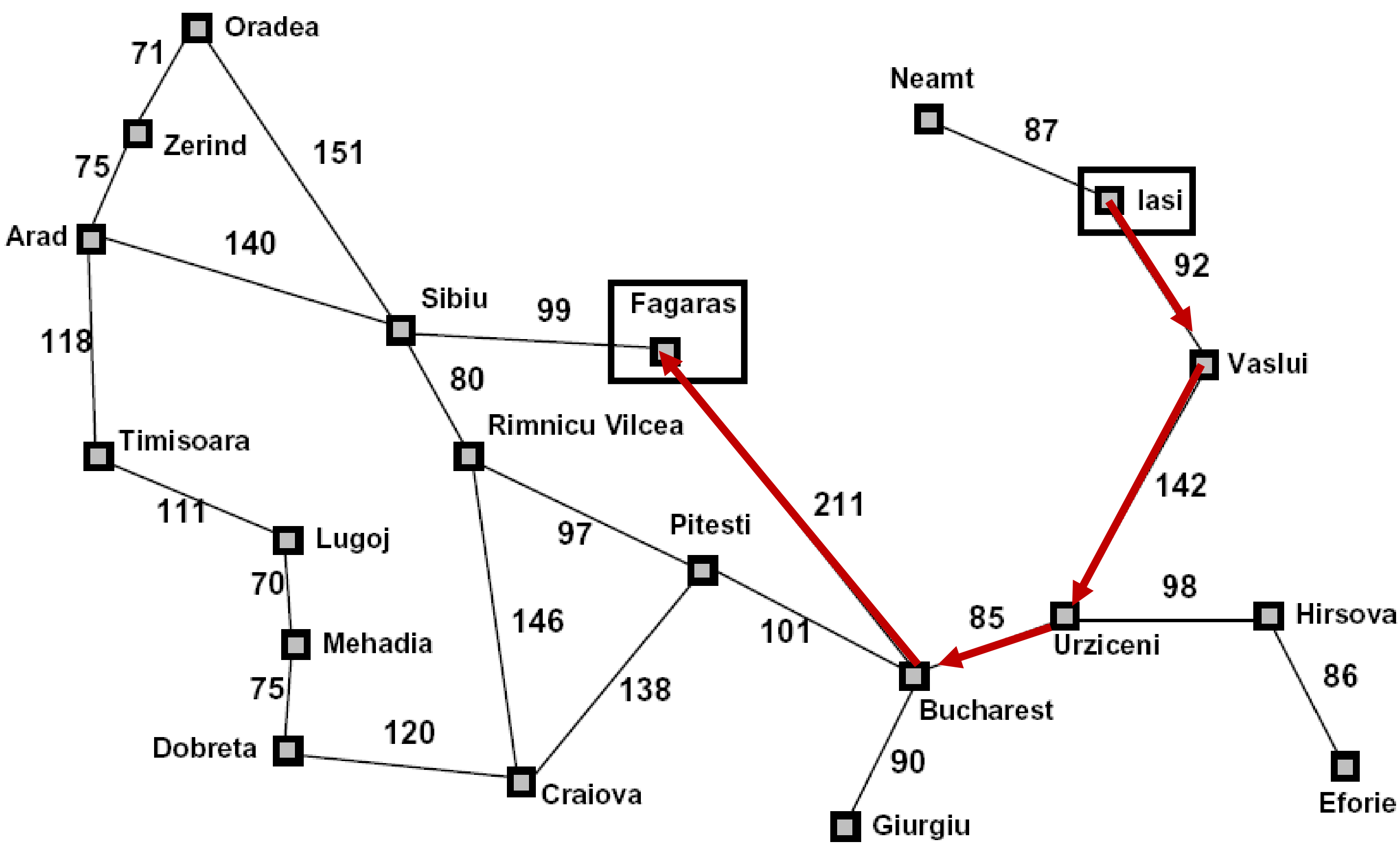


Straight−line distance to Bucharest

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

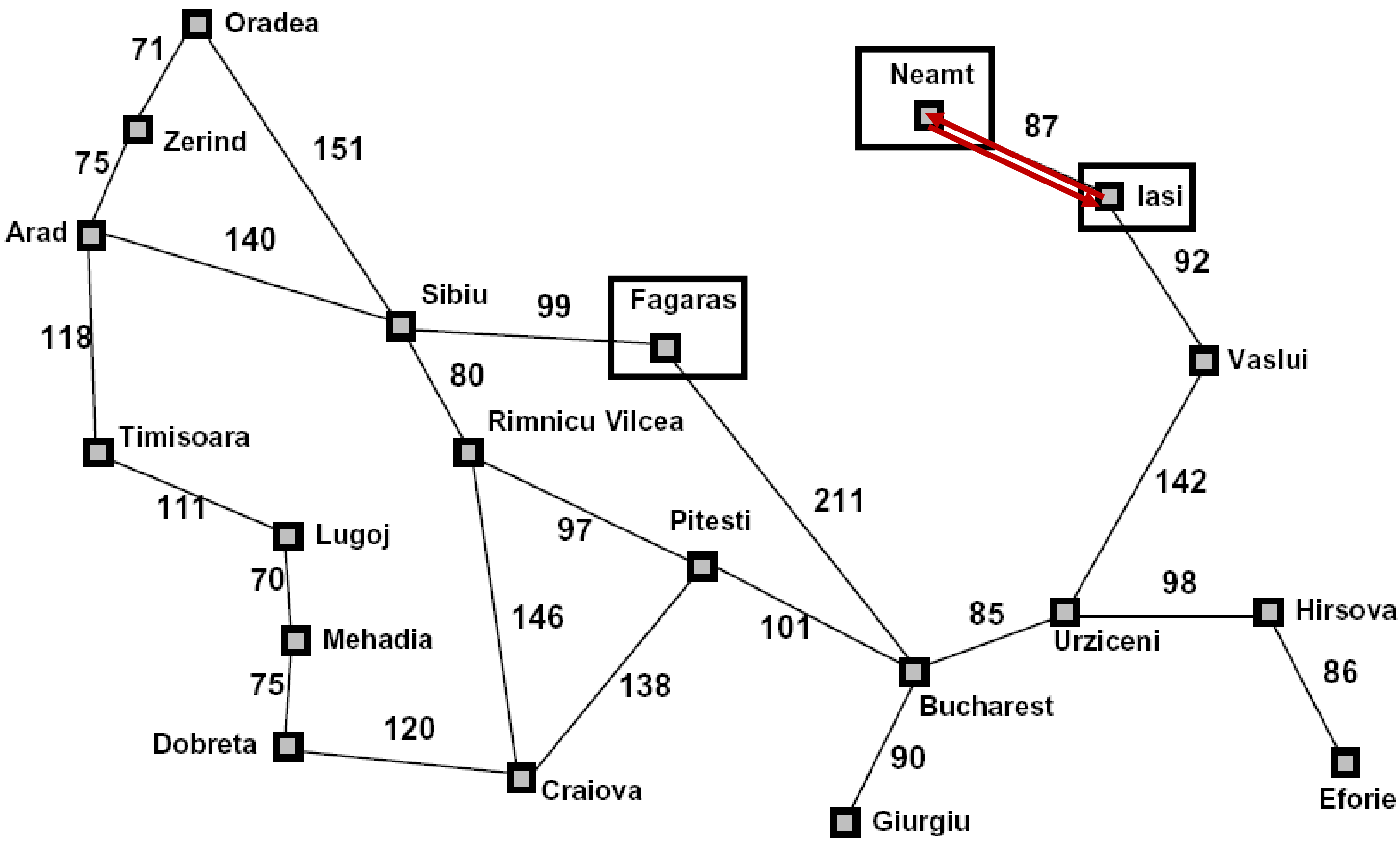
Greedy Search

- What can go wrong?
  - From Lasi to Fagaras



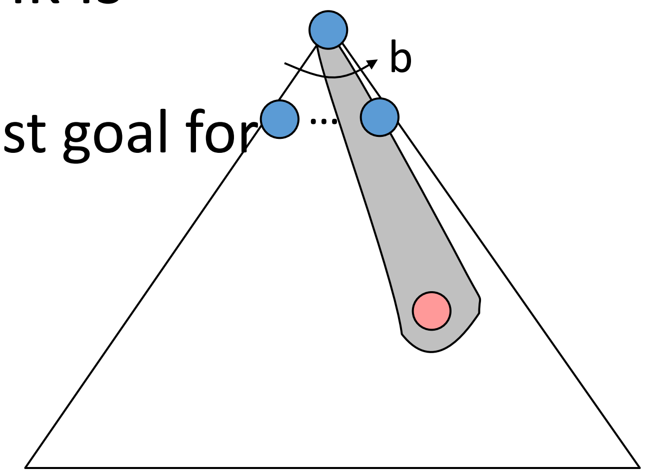
Greedy Search

- What can go wrong?
  - From Lasi to Fagaras

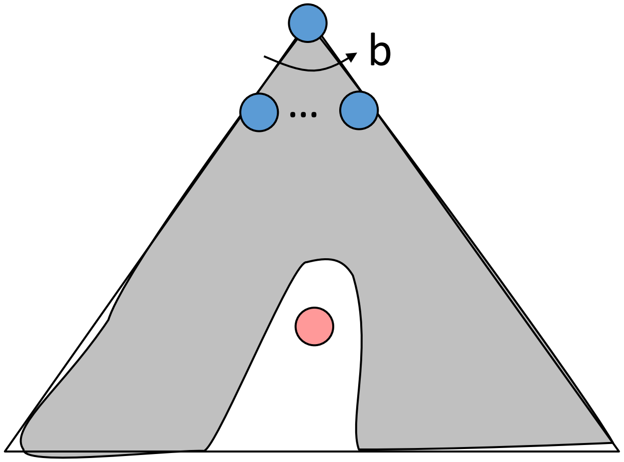


Greedy Search

- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state



- A common case:
  - Best-first takes you straight to the (wrong) goal



- Worst-case: like a badly-guided DFS

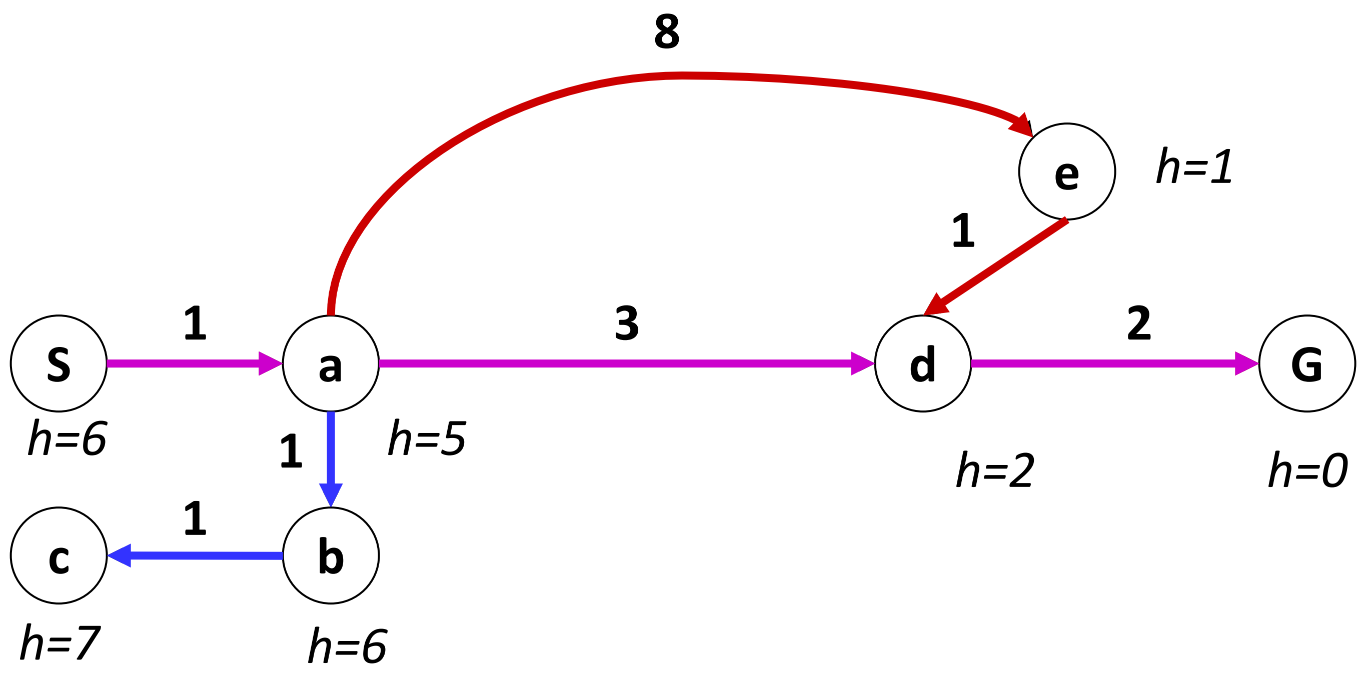
- Not Complete
- Not Optimal

A* Search

- Take into account the cost of getting to the node as well as our estimation of the cost of getting to the goal from the node.
- Evaluation function f(n)
  - $f(n) = g(n) + h(n)$
  - g(n) is the cost of the path represented by node n
  - h(n) is the heuristic estimate of the cost of achieving the goal from n.

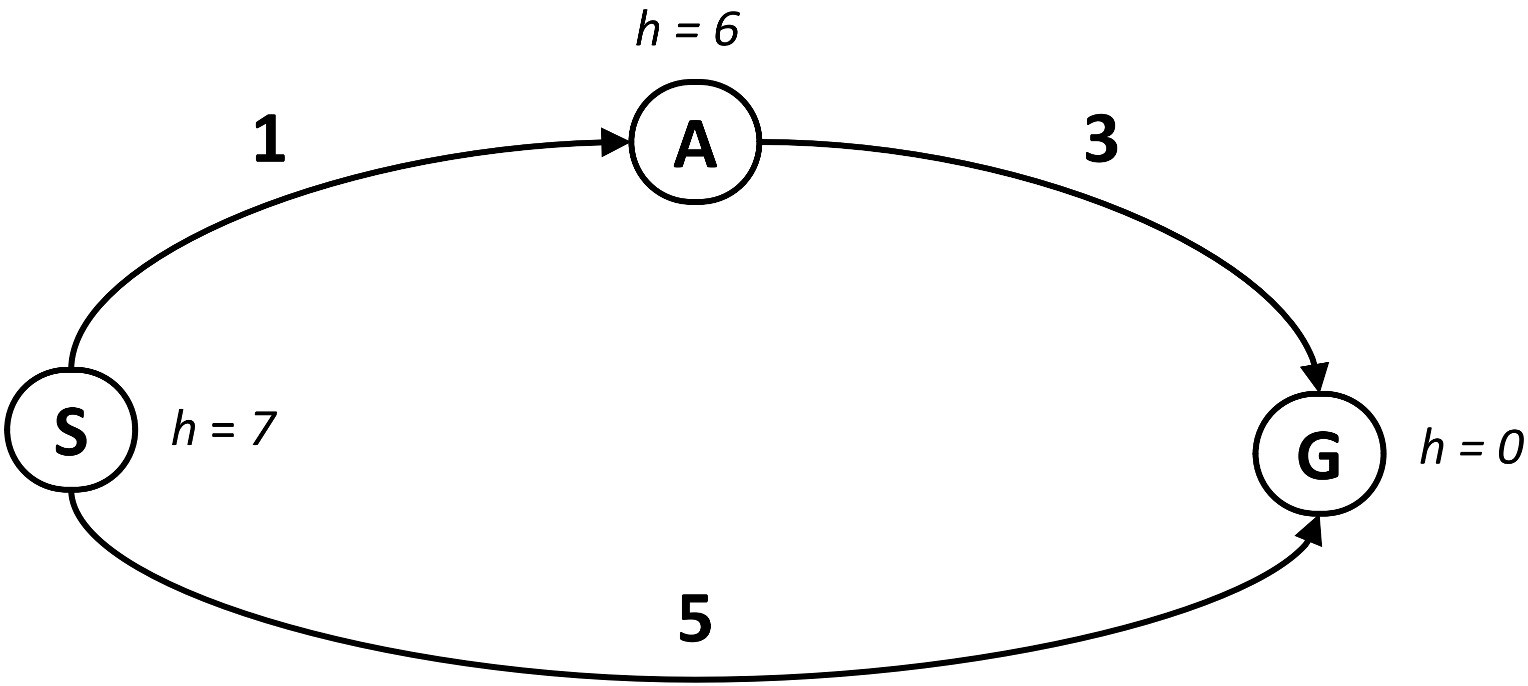
Quiz: Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)



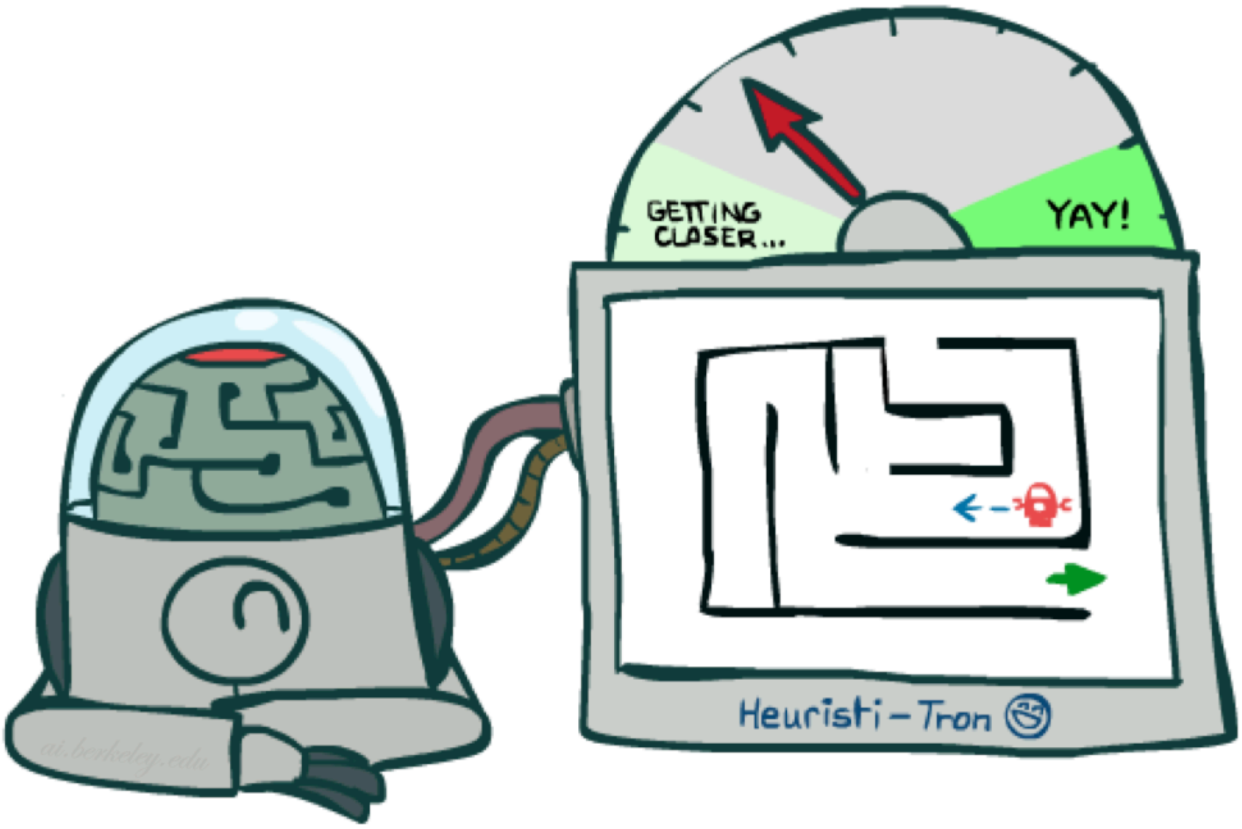
- A* Search orders by the sum: f(n) = g(n) + h(n)

Is A* Optimal?

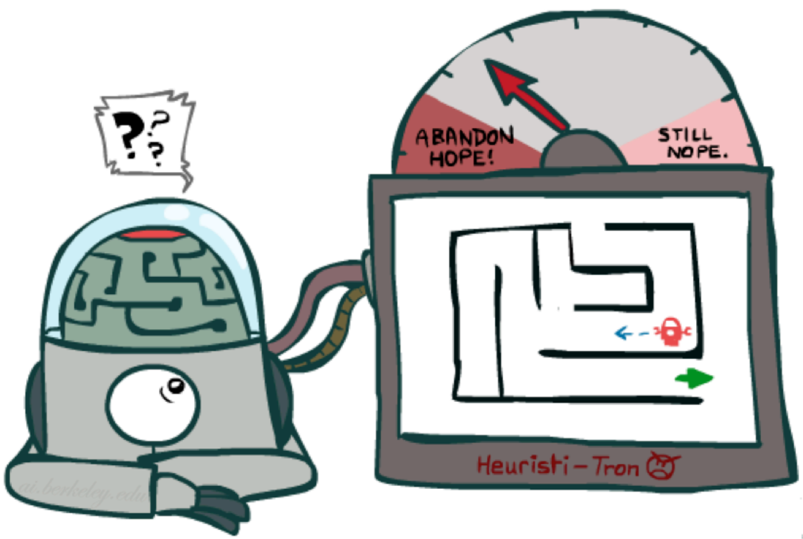


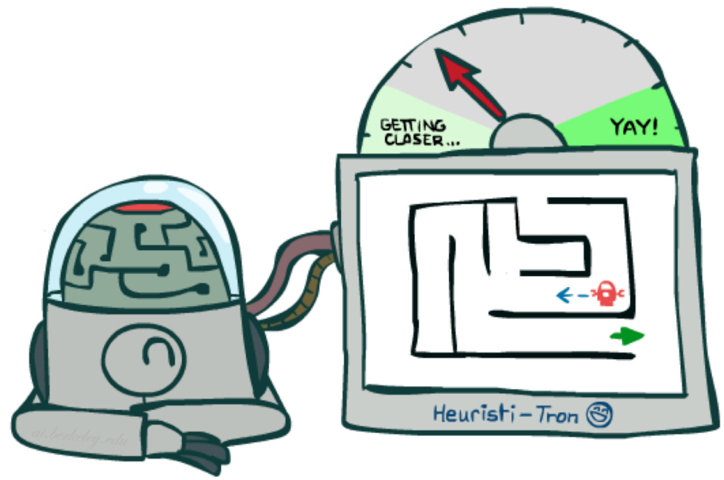
- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

Admissible Heuristics



Idea: Admissibility





Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe

Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs

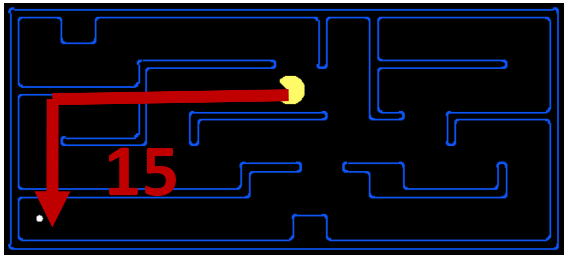
Admissible Heuristics

- A heuristic h is *admissible* (optimistic) if:

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

where $h^*(n)$ is the true cost to a nearest goal

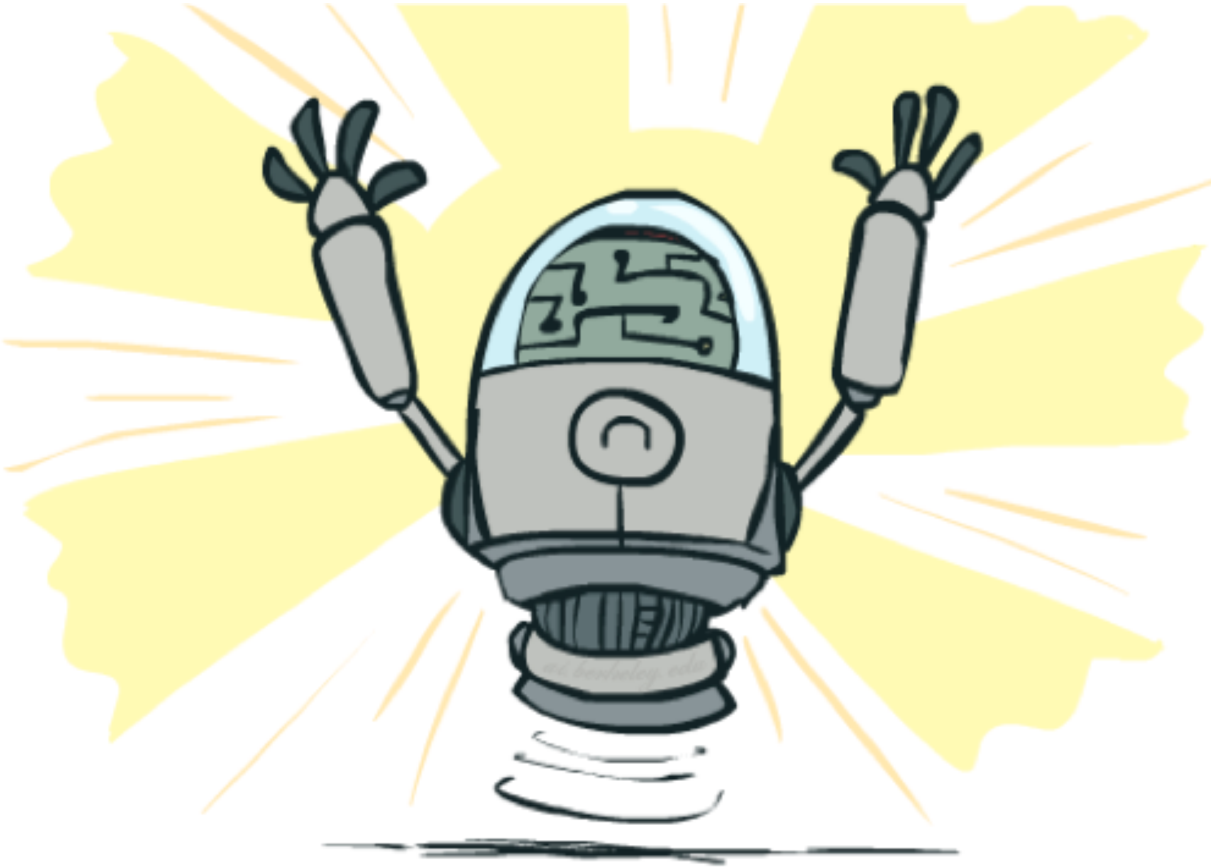
- Examples:



4

- Coming up with admissible heuristics is most of what's involved in using A* in practice.

Optimality of A* Tree Search



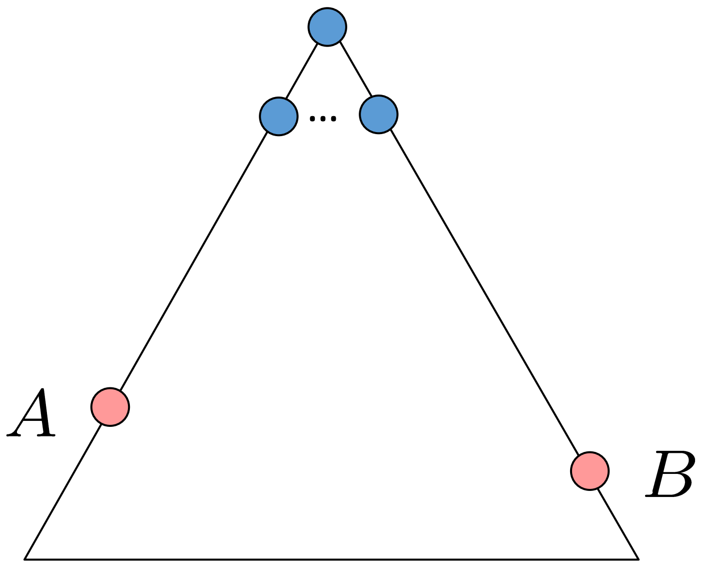
Optimality of A* Tree Search

Assume:

- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:

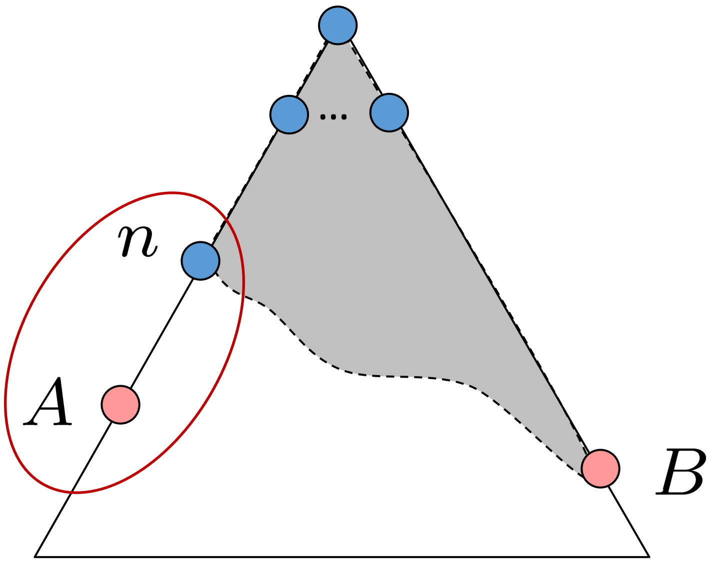
- A will be visited before B



Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too
- Claim: *n* will be expanded before B
  - f(n) is less or equal to f(A)

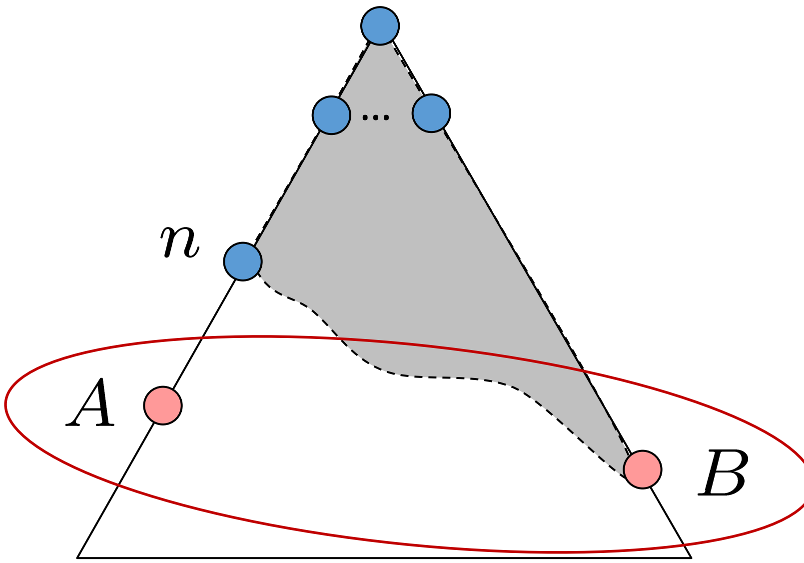


$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too
- Claim: *n* will be expanded before B
  - f(n) is less or equal to f(A)
  - f(A) is less than f(B)

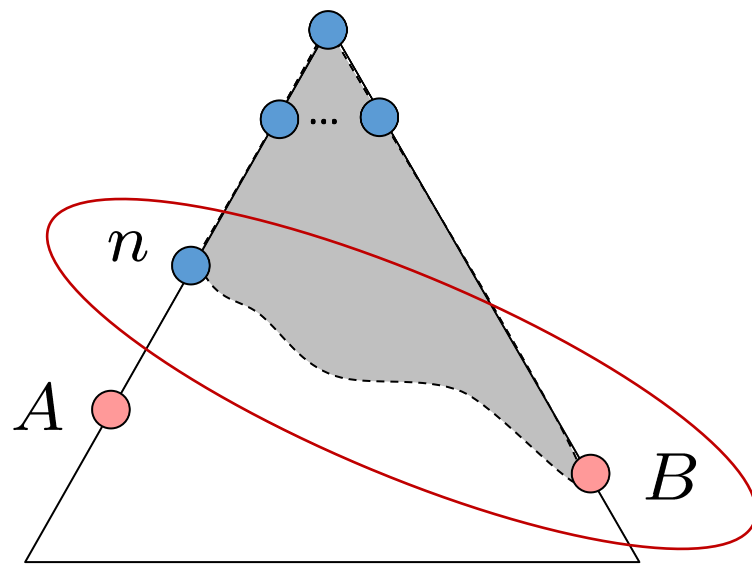


$$g(A) < g(B)$$
$$f(A) < f(B)$$

Optimality of A* Tree Search: Blocking

Proof:

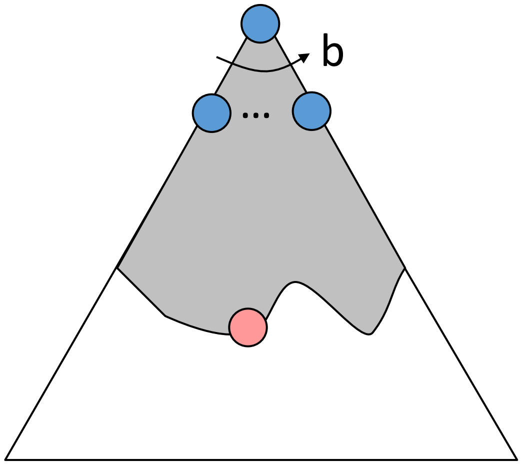
- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  - f(n) is less or equal to f(A)
  - f(A) is less than f(B)
  - *n* expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal



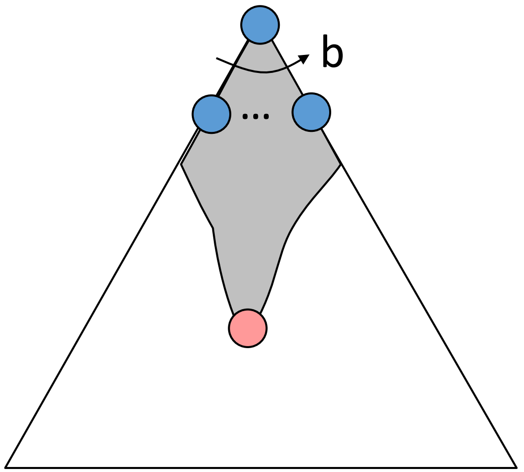
$$f(n) \leq f(A) < f(B)$$

Properties of A*

Uniform-Cost

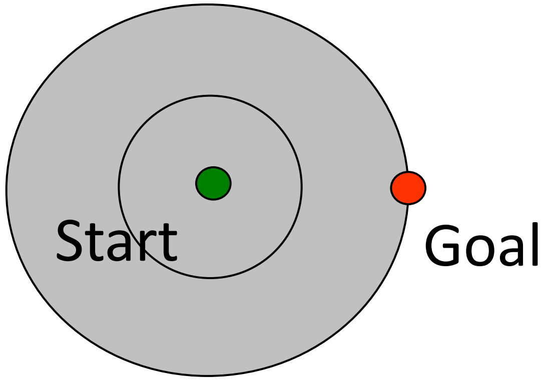


A*

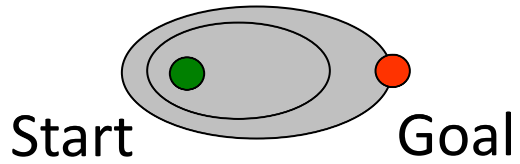


UCS vs A* Contours

- Uniform-cost expands equally in all "directions"



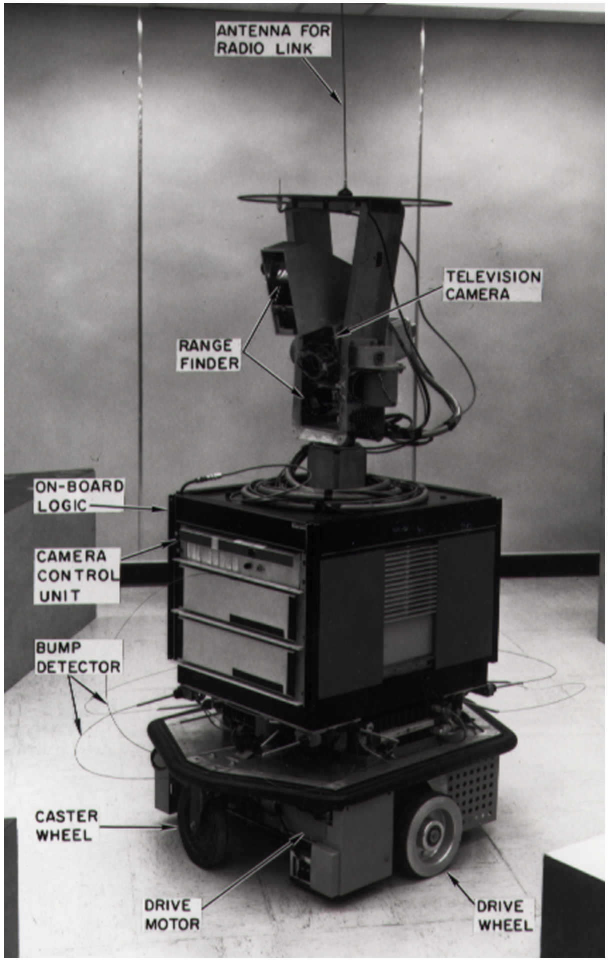
- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



A* History

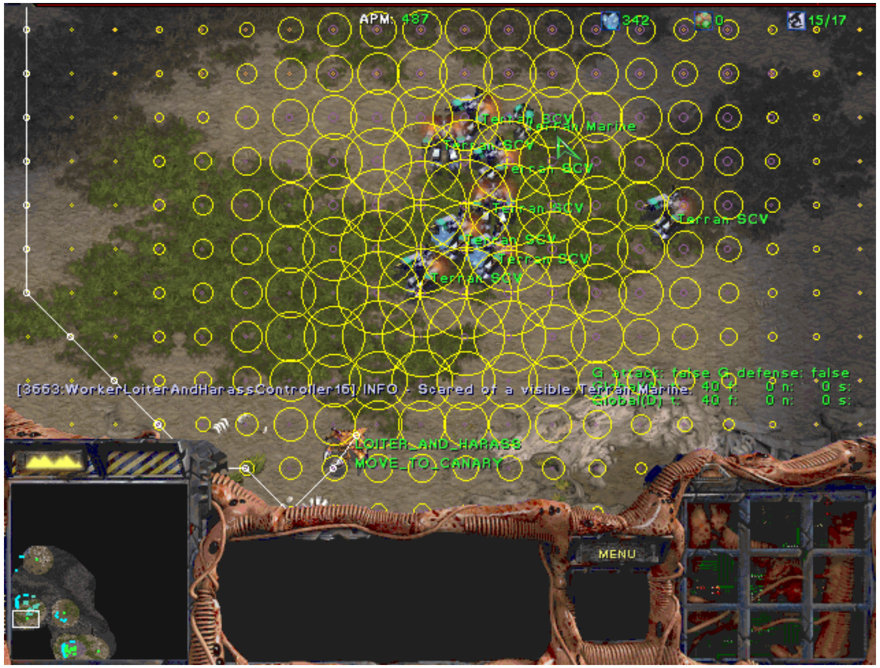
- Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first described the algorithm in 1968.

- A1 -> A2 -> A*(Optimal)

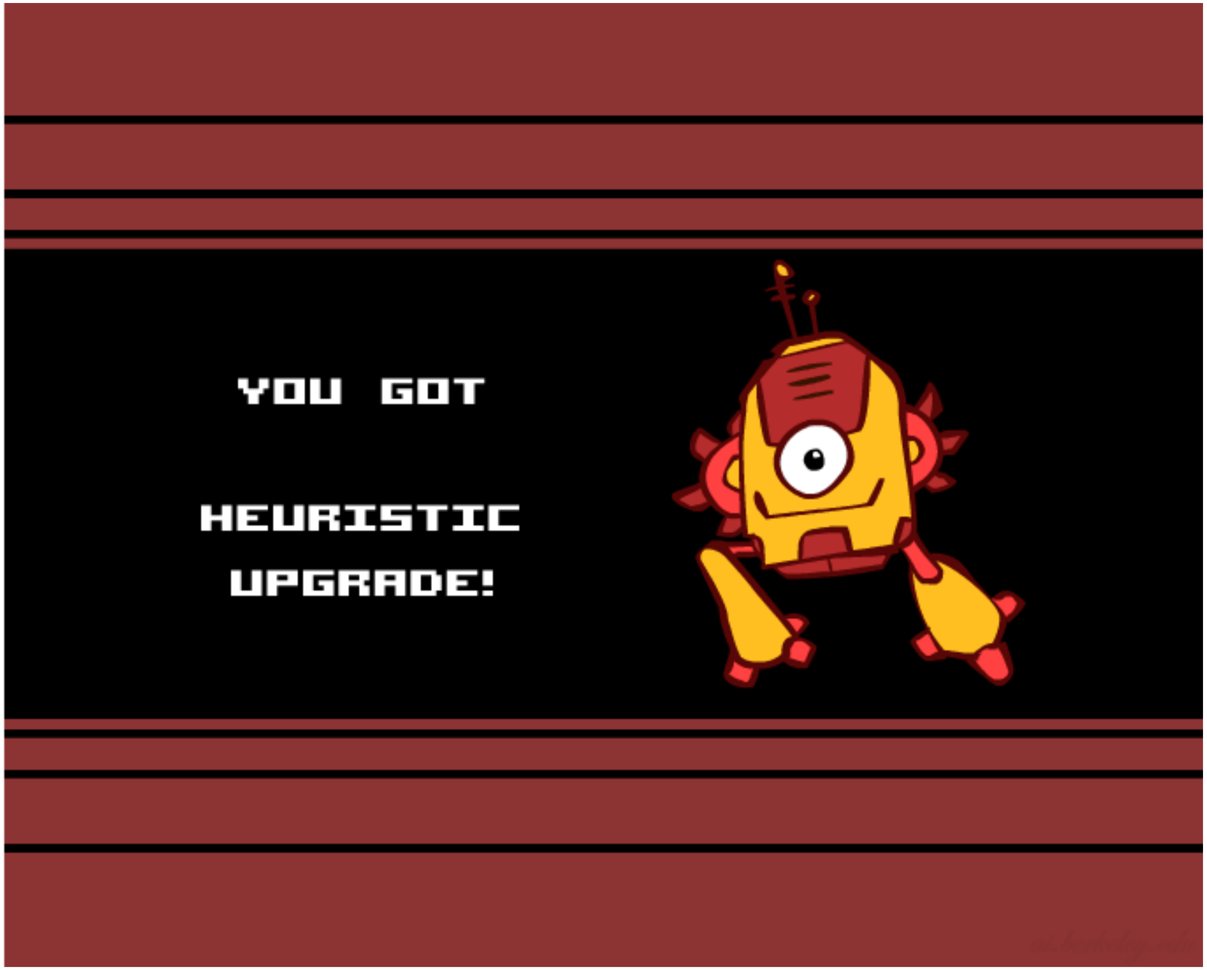


A* Applications

- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …



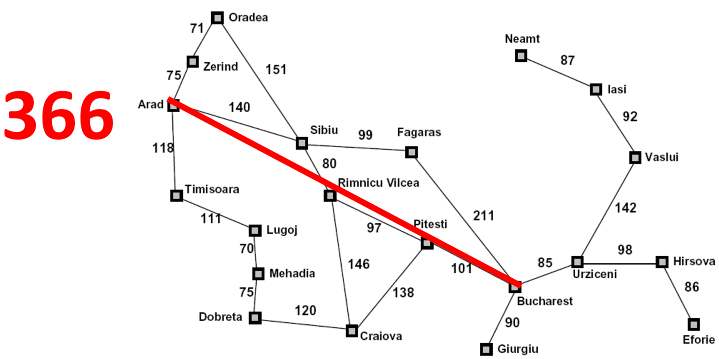
Creating Heuristics

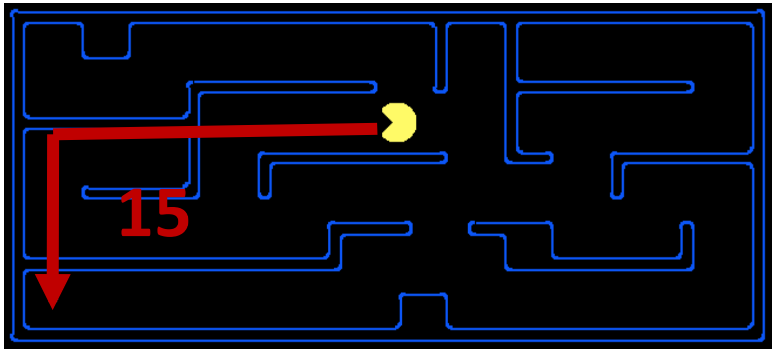


Creating Admissible Heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics

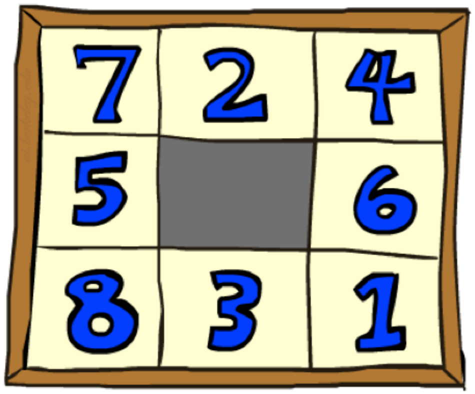
- Often, admissible heuristics are solutions to *relaxed problems,* where new actions are available



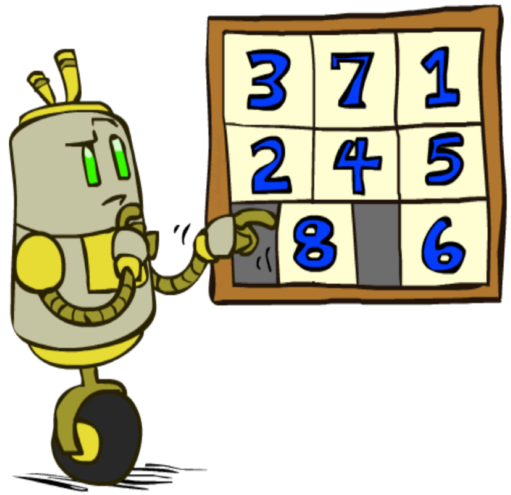


- Inadmissible heuristics are often useful too

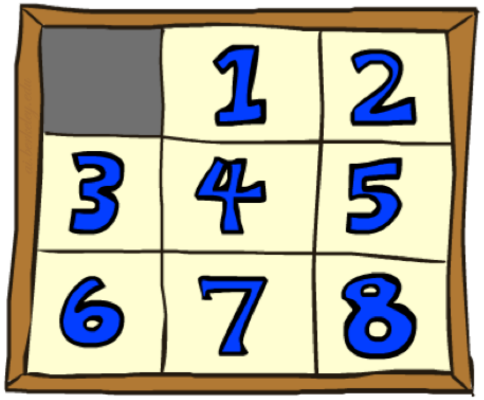
Example: 8 Puzzle



Start State



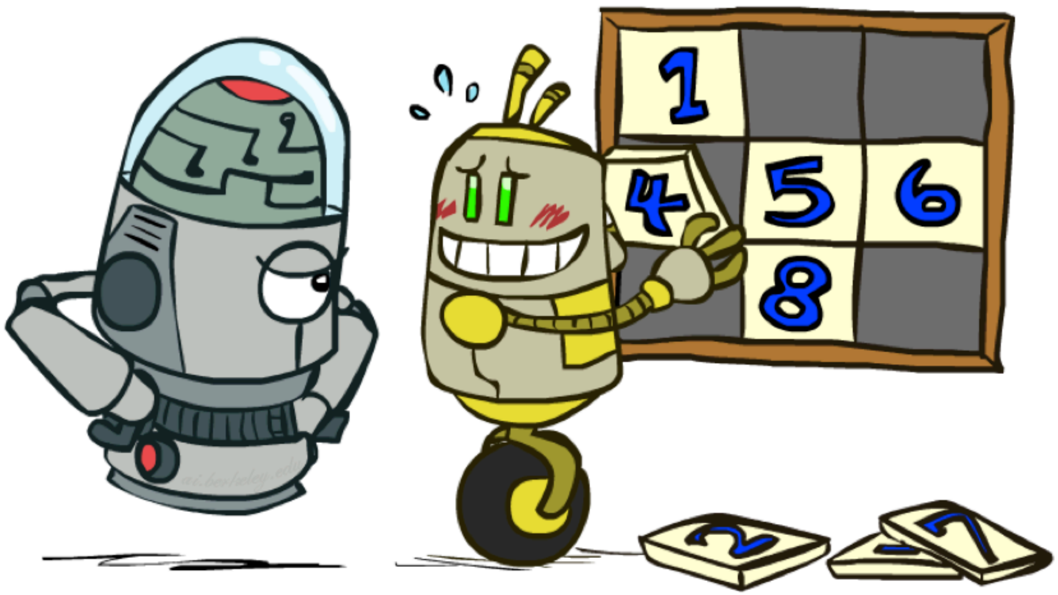
Actions

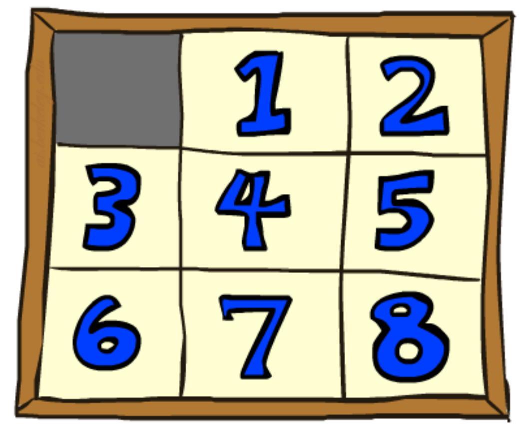


Goal State

8 Puzzle I

- Heuristic: Number of tiles misplaced
- Why is it admissible?
- h(start) = 8



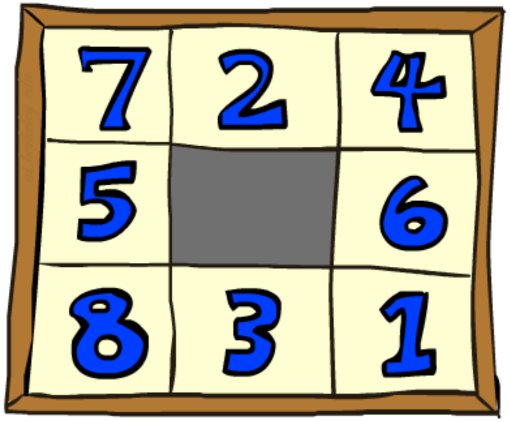


Goal State

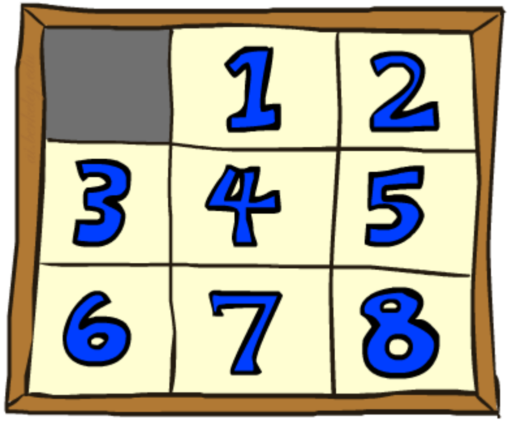
8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total *Manhattan* distance

- Why is it admissible?
- h(start) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18



Start State



Goal State

How to design a admissible heuristic functions ?

From relaxed problem

A tile can move from square A to square B if A is adjacent to B and B is blank.

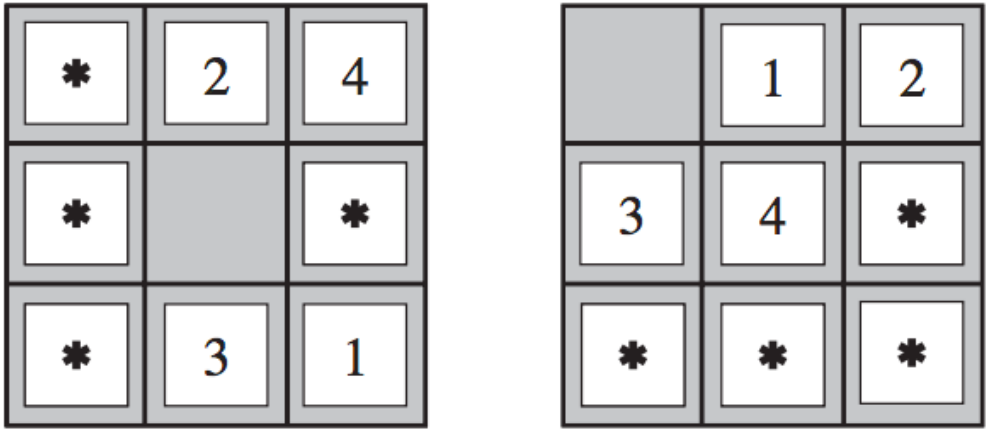
- Constraint 1: A and B is adjacent
- Constraint 2: B is blank

- Problem 1: A tile can move from square A to square B if A is adjacent to B.
- Problem 2: A tile can move from square A to square B if B is blank.
- Problem 3: A tile can move from square A to square B.

Heuristic function can be generated automatically with formal expression of the original question!

From sub-problems

- The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.
- The cost of solving the sub-problem is definitely no more than its original problem.



Start State Goal State

The comparison of different heuristic function

- Effective Branching Factor (b^*) is computed based on the depth and nodes # (N) in the tree.

$$\mathrm{N} + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

	Search Cost (nodes generated)			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	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	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

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

- h1: # of tiles mis-placed
- h2: Total Manhattan distance

Trivial Heuristics, Dominance

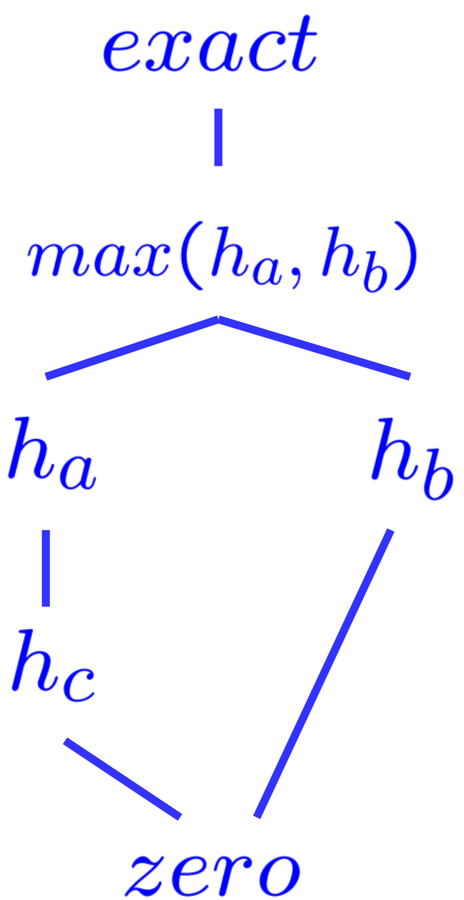
- Dominance: $h_a \geq h_c$ if

$$\forall n : h_a(n) \geq h_c(n)$$

- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible

$$h(n) = max(h_a(n), h_b(n))$$

- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic



More about heuristic function

- How about using the *actual cost* as a heuristic?
  - Would it be admissible?
    - Yes
  - Would we save on nodes expanded?
    - Yes
  - What’s wrong with it?
    - More computational cost.

- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself

Learning for heuristic functions

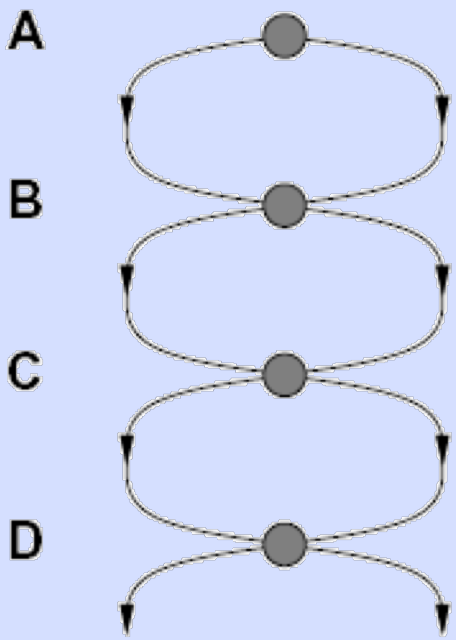
- Learning heuristic functions

$$H(n) = c_1 x_1(n), \ldots, c_m x_m(n)$$

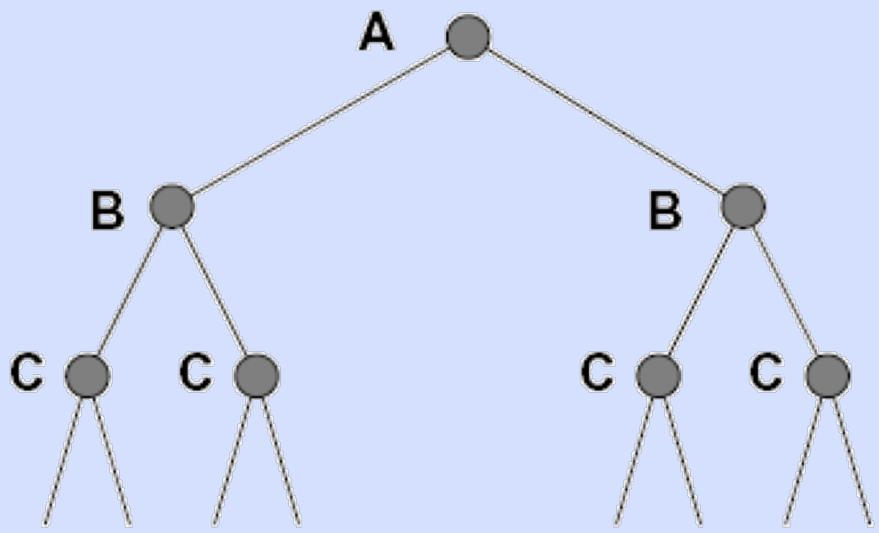
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.

State Graph

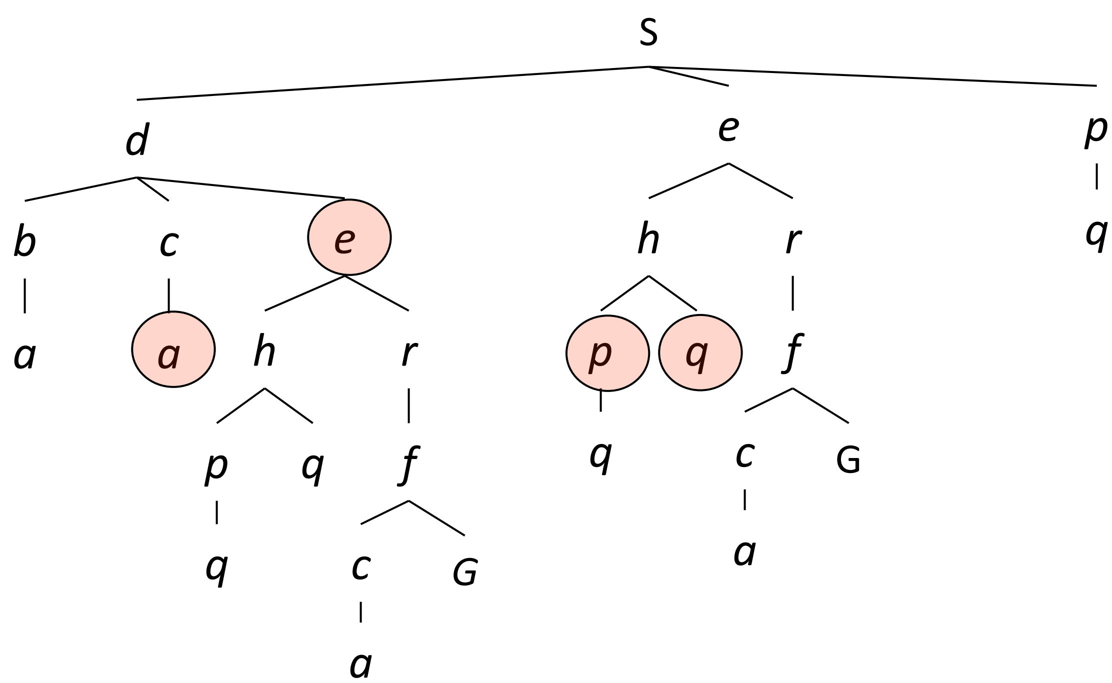


Search Tree



Graph Search

- In BFS, for example, we shouldn't bother expanding the circled nodes (why?)



Graph Search

- Idea: never expand a state twice

- How to implement:
  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set

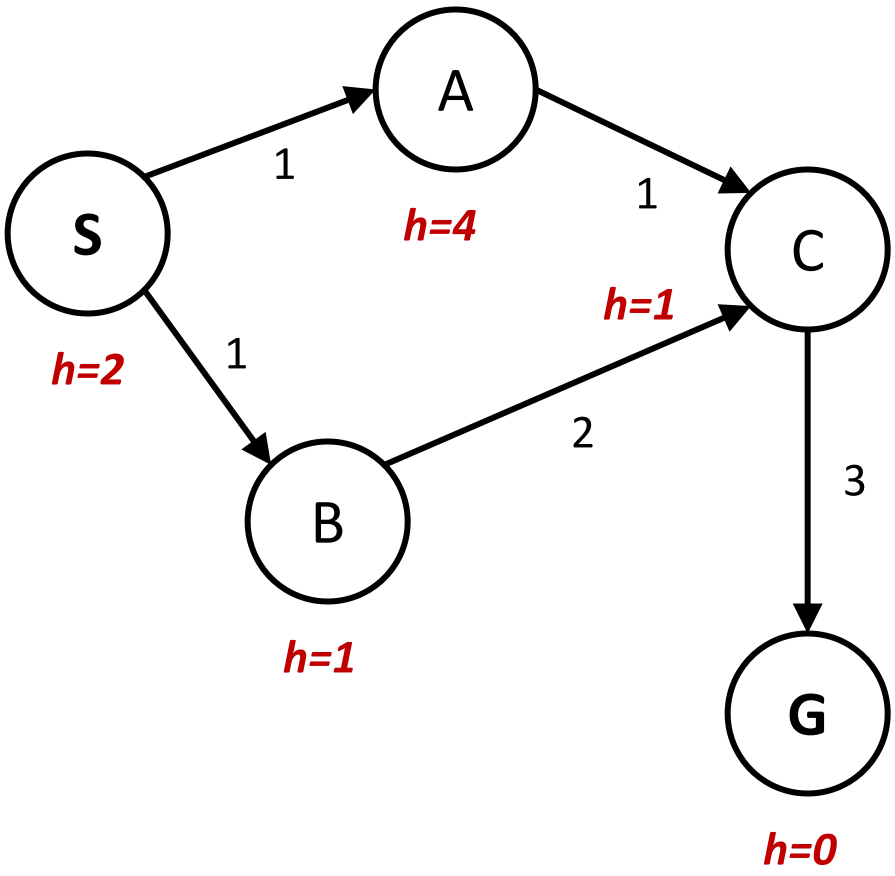
- Important: store the closed set as a set, not a list

- Can graph search wreck completeness? Why/why not?

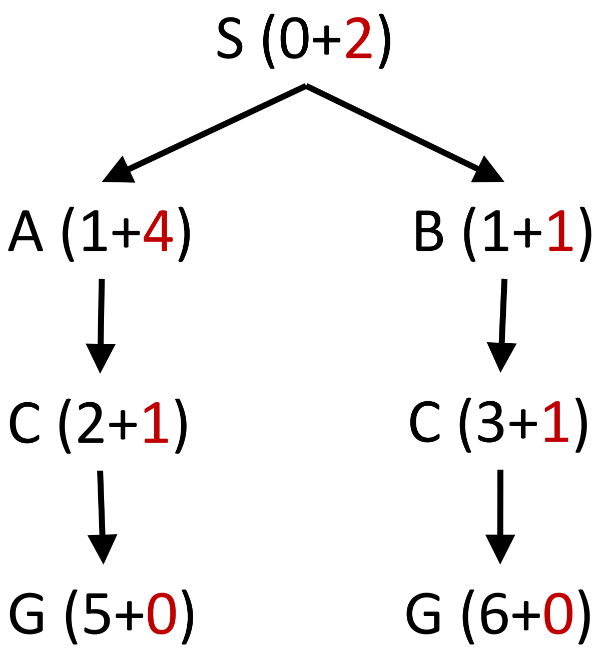
- How about optimality?

A* Gone Wrong with admissible function

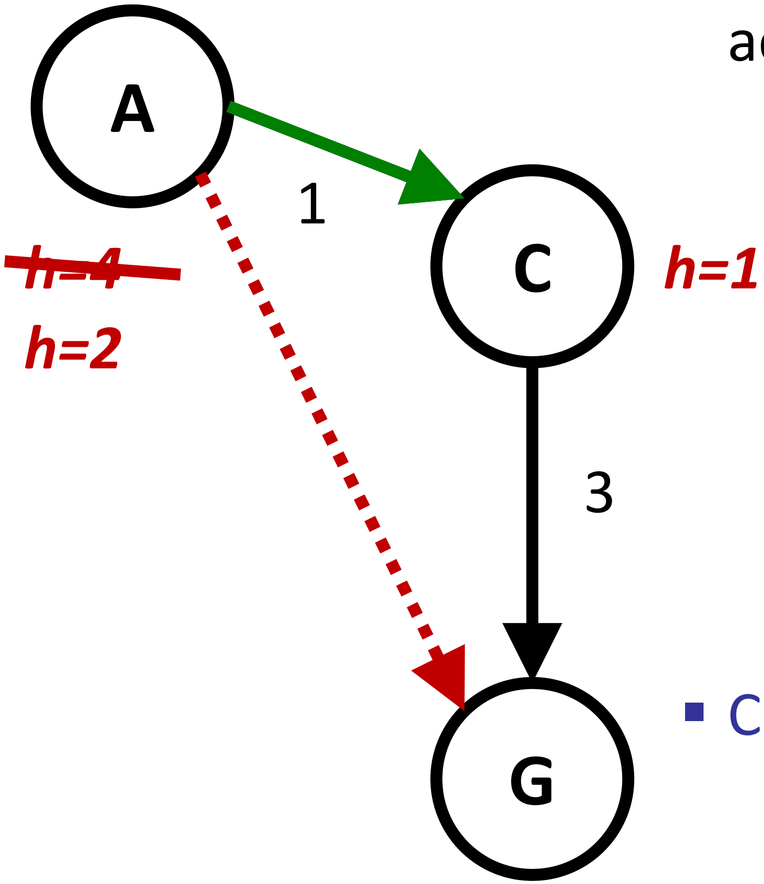
State space graph



Search tree



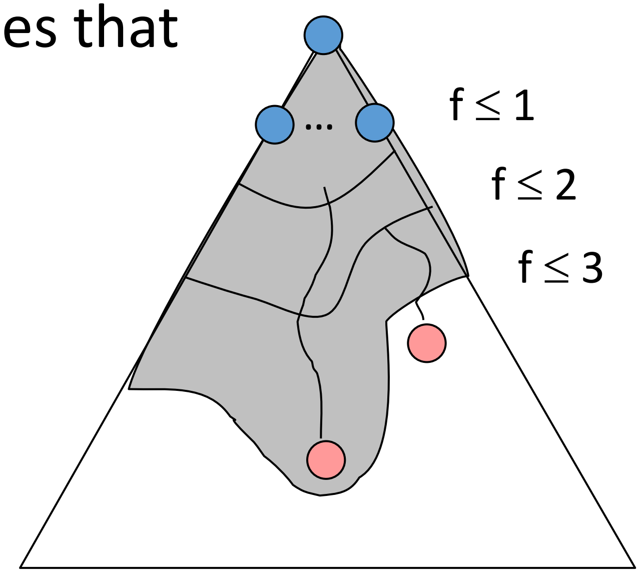
Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
  - h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc
  - h(A) – h(C) ≤ cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases
  - h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal

Optimality of A* Search

- Sketch: consider what A* does with a consistent heuristic:
  - Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
  - Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s sub-optimally
  - Result: A* graph search is optimal

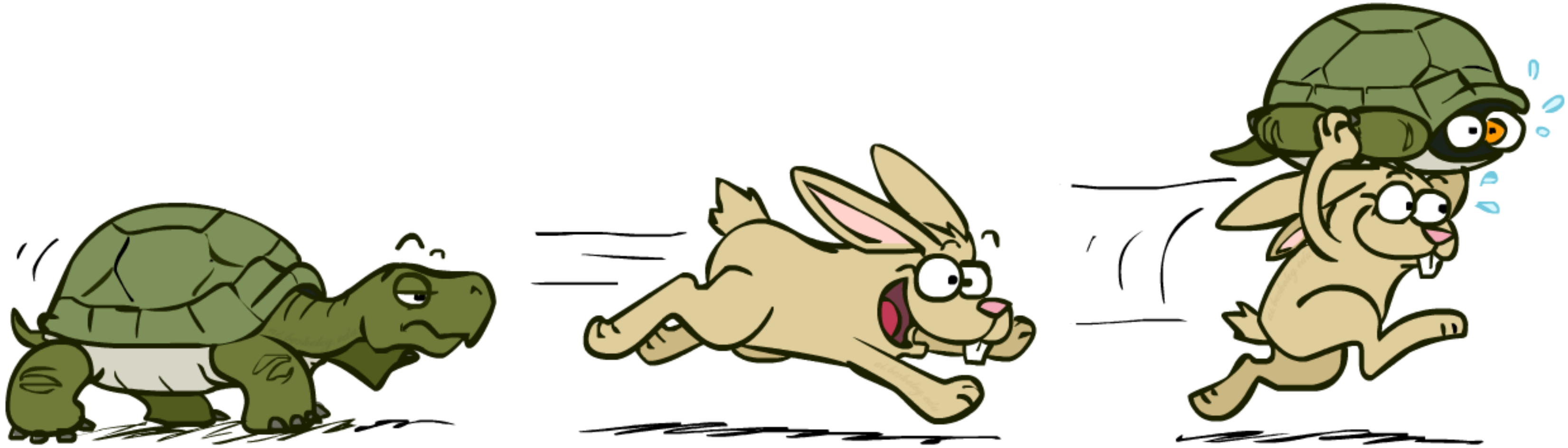


Optimality

- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility

A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



Tree Search Pseudo-Code

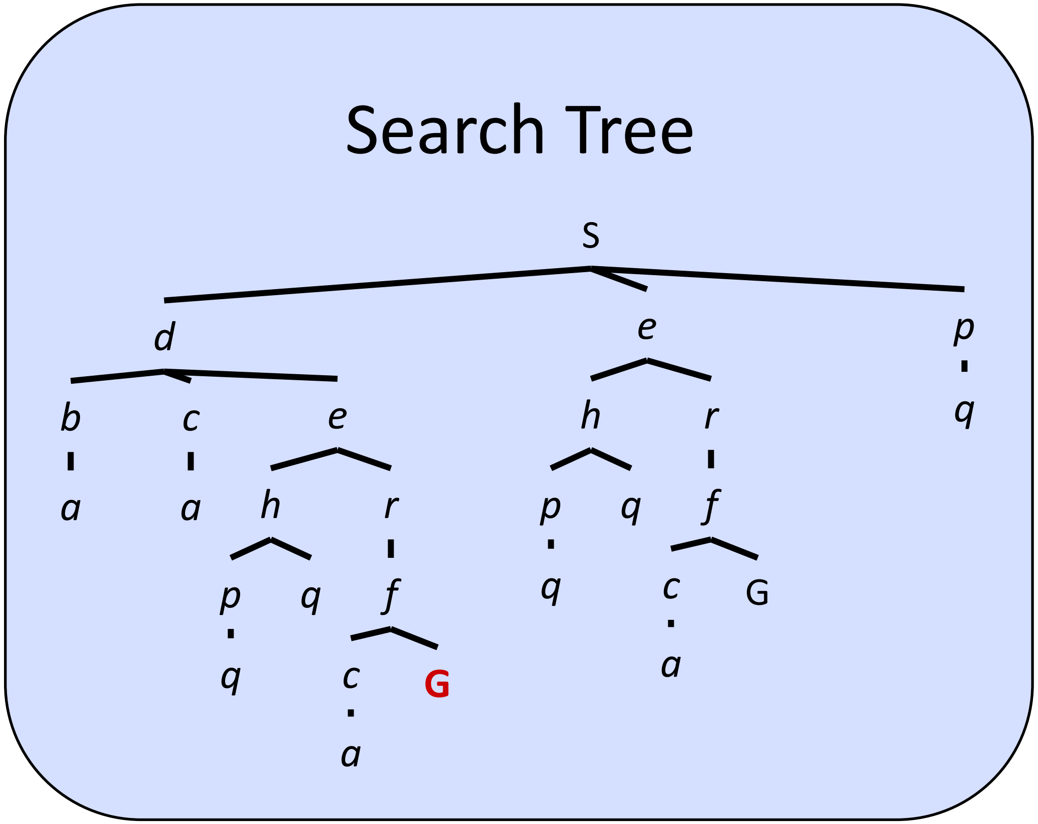
```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Tree Search Pseudo-Code VS Graph Search Pseudo-Code

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

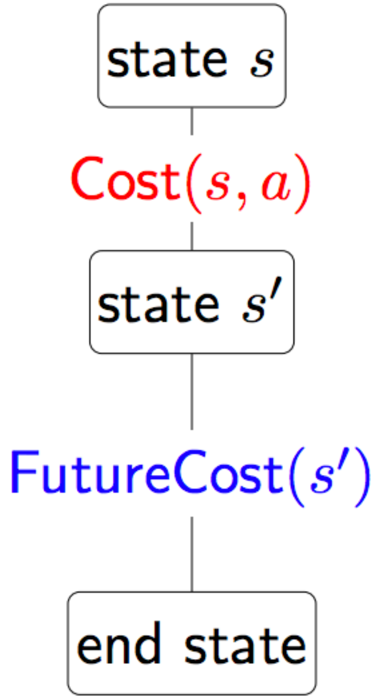
- **Dynamic programming** is a method for solving a complex problem by breaking it down into a collection of simpler sub-problems.



S -> G

1. S -> d; d -> G
2. S -> e; e -> G
3. S -> p; p -> G

Dynamic Programming



S -> G

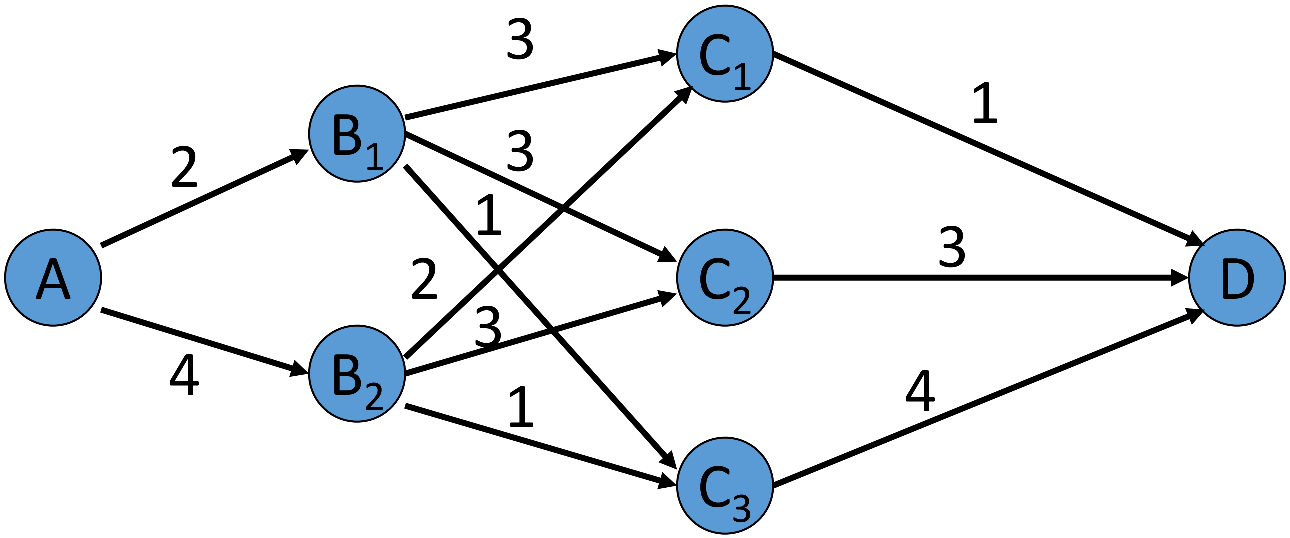
1. S -> d; d -> G
2. S -> e; e -> G
3. S -> p; p -> G

- **s' is an intermediate state**
- **Cost (s, a) is the cost for taking action *a* from state *s***
- **Actions(s) are all actions can be taken from state s**
- **Succ (s, a) is the state from state s by taking action a**

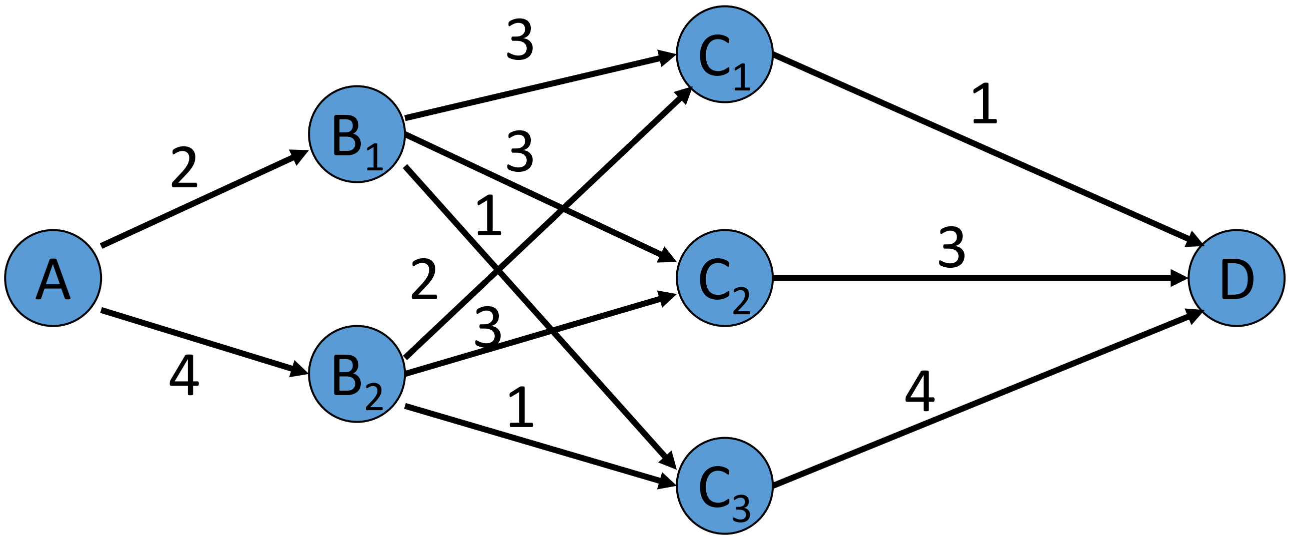
$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

Example: Shortest Path

- Build a gas pipeline from city A to city D.
- Go through two intermediate stations B and C.
- For each station, there are multiple candidate locations.
- Identify the path with shortest cost.

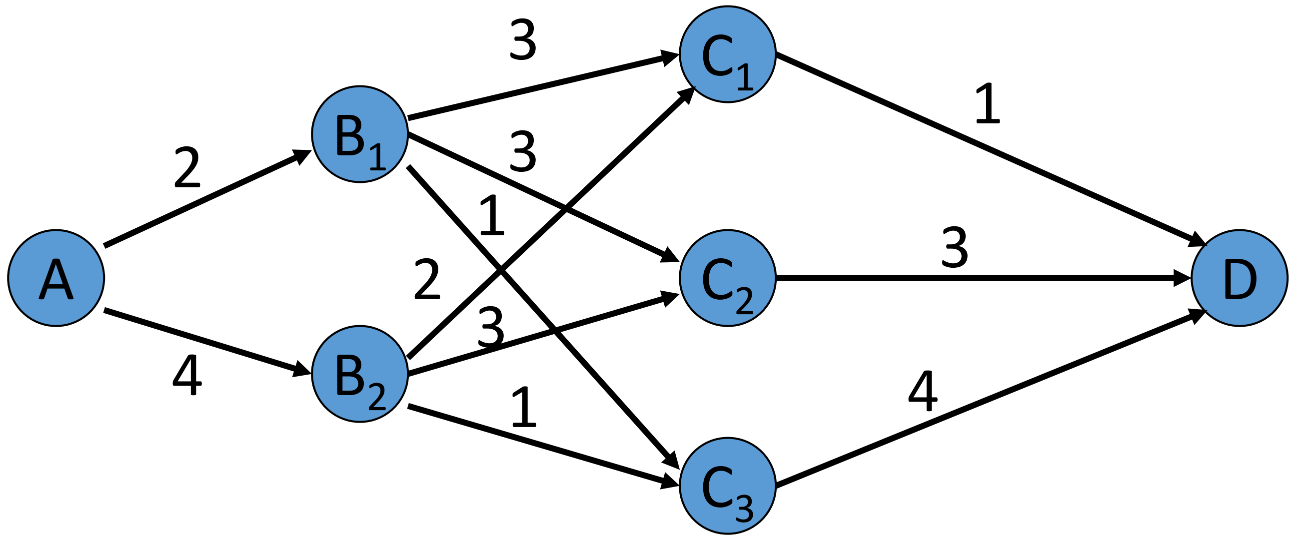


Example: Shortest Path



	A	B_1	B_2	C_1	C_2	C_3	D
A	0	2	4	?	?	?	?
B_1	-	0	-	3	3	1	?
B_2	-	-	0	2	3	1	?
C_1	-	-	-	0	-	-	1
C_2	-	-	-	-	0	-	3
C_3	-	-	-	-	-	0	4
D	-	-	-	-	-	-	0

Example: Shortest Path

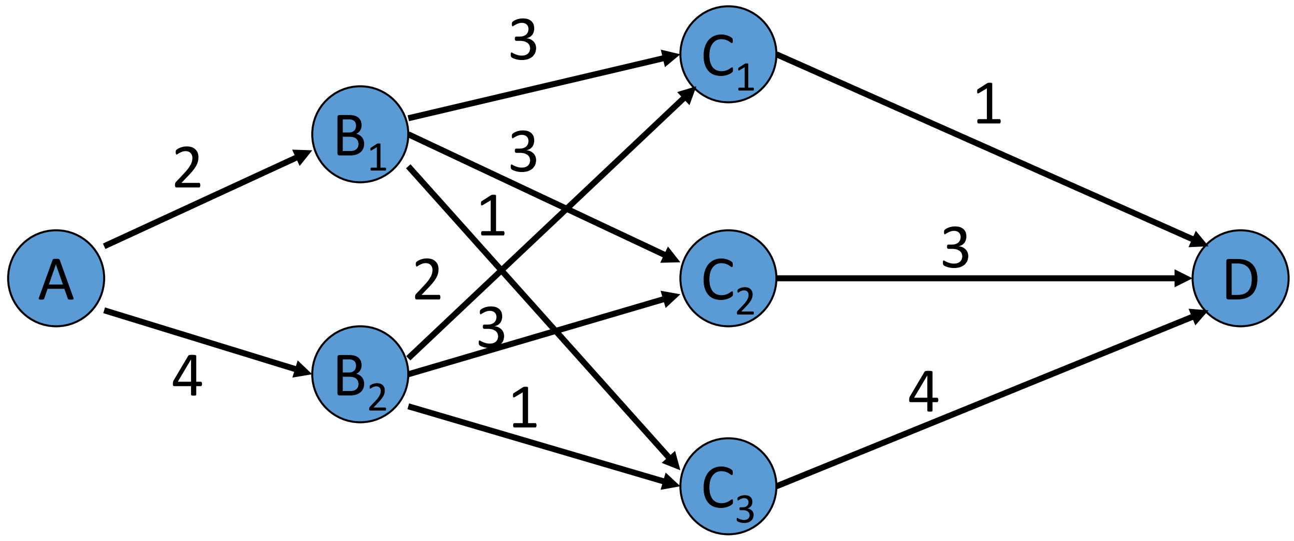


	A	B_1	B_2	C_1	C_2	C_3	D
	0	2	4	?	?	?	?
B_1	-	0	-	3	3	1	$\boxed{4}$
B_2	-	-	0	2	3	1	$\boxed{3}$
C_1	-	-	-	0	-	-	1
C_2	-	-	-	-	0	-	3
C_3	-	-	-	-	-	0	4
D	-	-	-	-	-	-	0

FutureCost(B_1) = Min{
Via C_1: 3 + 1,
Via C_2: 3 + 3,
Via C_3: 1 + 4
}

FutureCost(B_2) = Min{
Via C_1: 2 + 1,
Via C_2: 3 + 3,
Via C_3: 1 + 4
}

Example: Shortest Path



	A	B_1	B_2	C_1	C_2	C_3	D
	0	2	4	?	?	?	$\boxed{6}$
B_1	-	0	-	3	3	1	4
B_2	-	-	0	2	3	1	3
C_1	-	-	-	0	-	-	1
C_2	-	-	-	-	0	-	3
C_3	-	-	-	-	-	0	4
D	-	-	-	-	-	-	0

FutureCost(A)
= Min{
Via B_1: 2 + 4,
Via B_2: 4 + 3,
}

A - > B_1 -> C_1 -> D

Three Components for DP

- The recurrence relation (for defining the value of an optimal solution);
- The tabular computation (for computing the value of an optimal solution);
- The traceback (for delivering an optimal solution).

Example: Fibonacci numbers

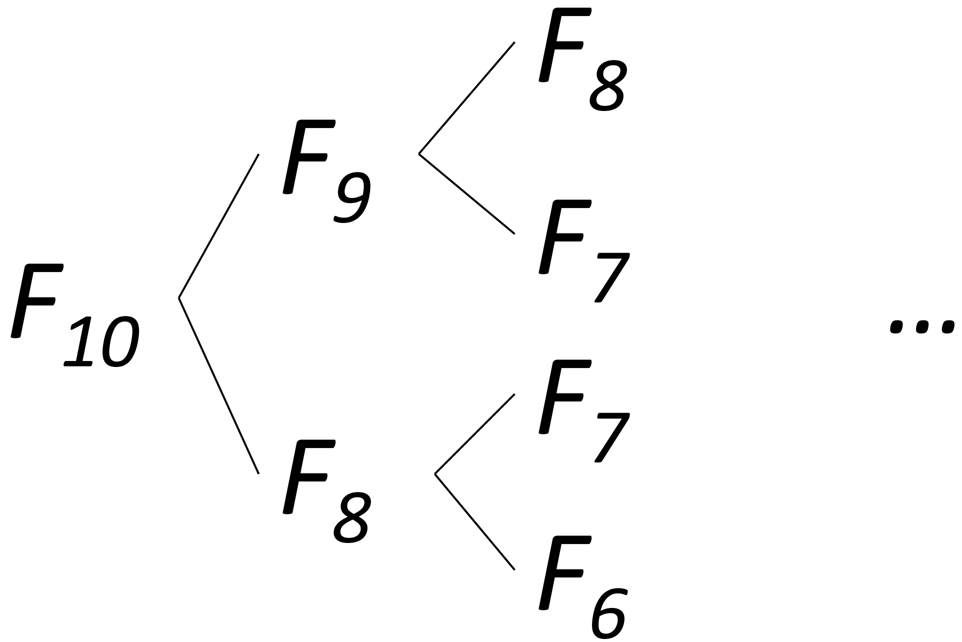
- The *Fibonacci numbers* are defined by the following recurrence:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1$$

Example: How to compute F_{10}

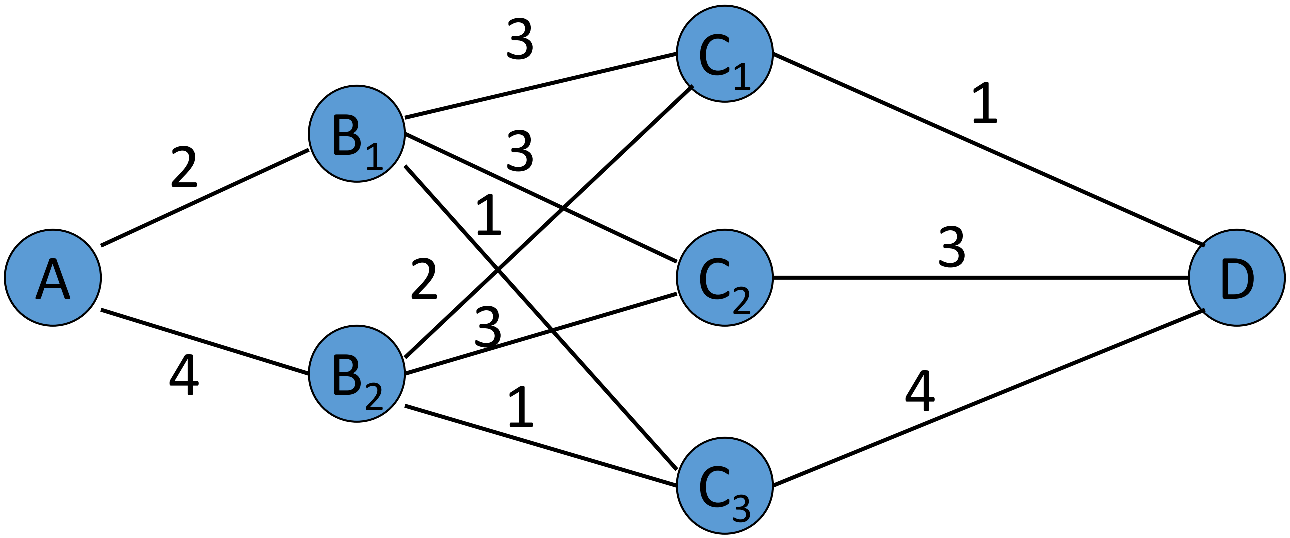


Example: Tabular computation

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

DP VS UCS

- DP can not work on undirected graph. (sub-problem can not be determined easily)



- UCS can not work on graph with negative cost.