Informed (Heuristic) Search Strategies

A* Search

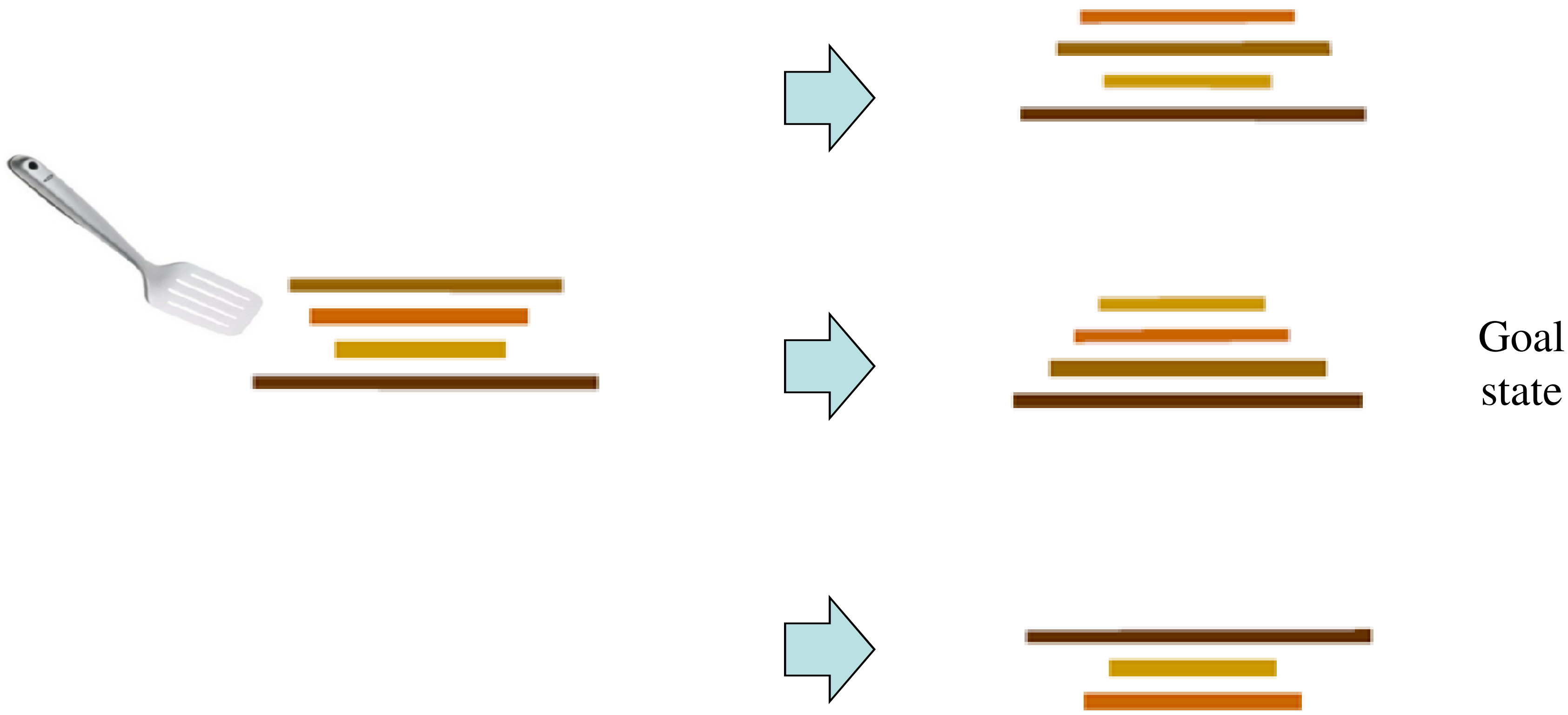
Outline

- Greedy Best First Search
- A* Search
- Graph Search
- Heuristic Design

Recap: Search

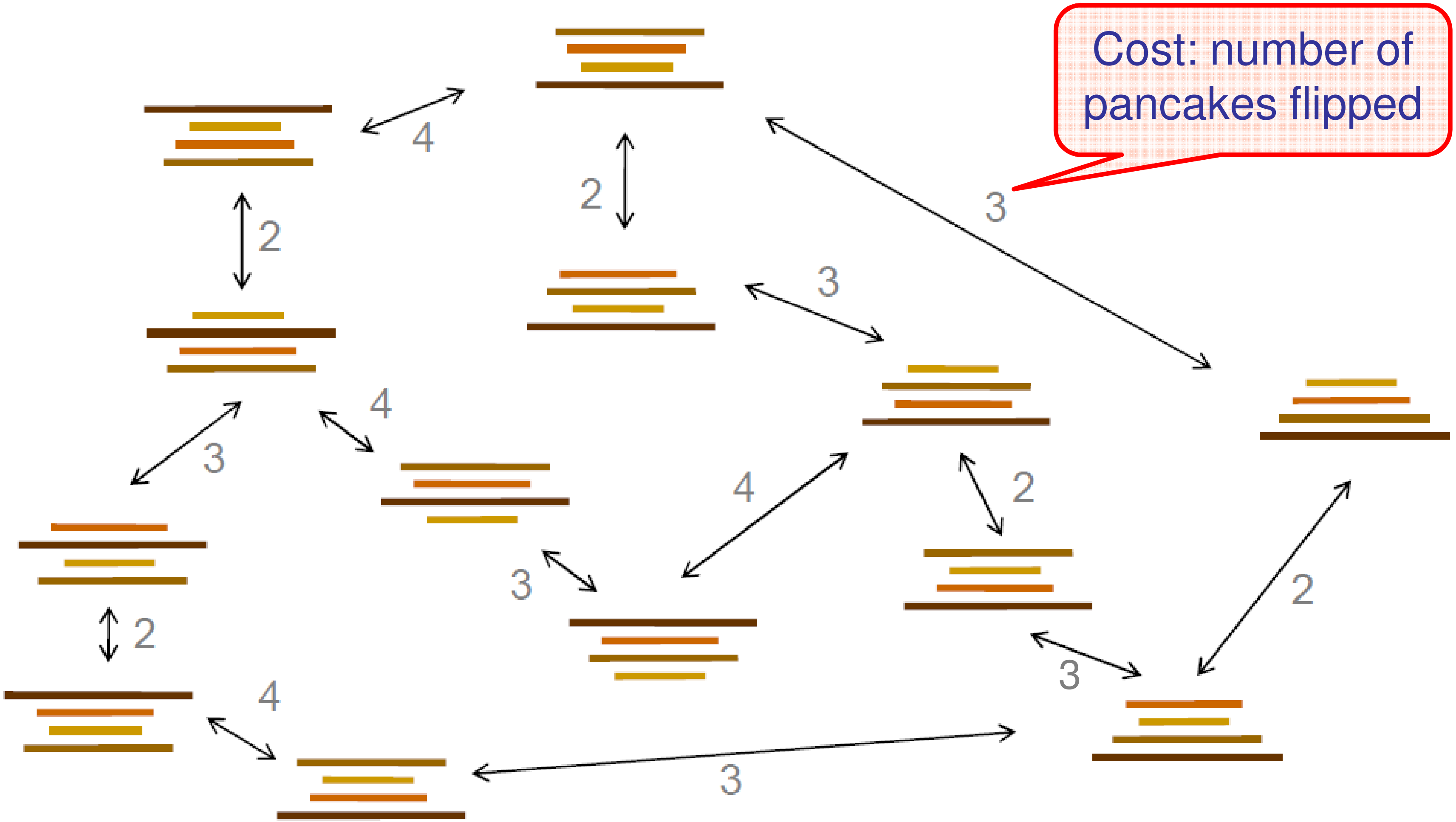
- Search problem:
  - States (configurations of the world)
  - Successor function: a function from states to lists of (state, action, cost) triples; drawn as a graph
  - Start state and goal test
- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)
- Search Algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
  - Optimal: finds least-cost plans

Example: Pancake Problem



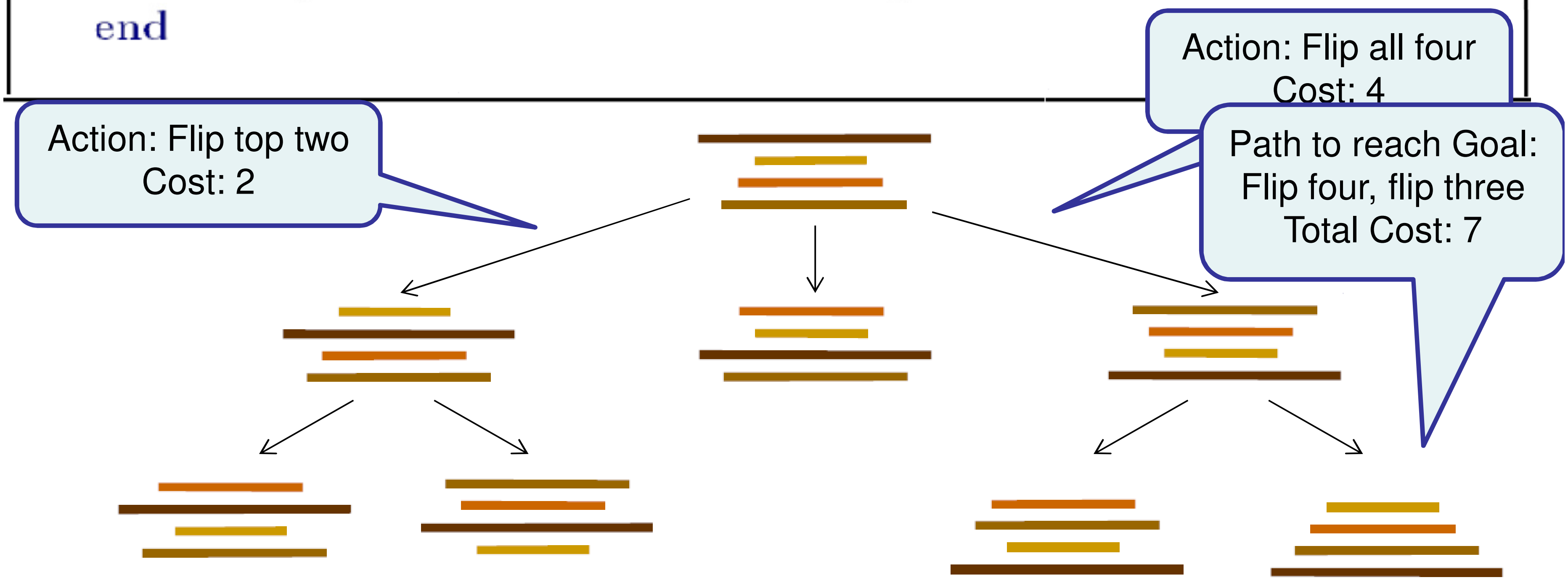
Example: Pancake Problem

State space graph with costs as weights



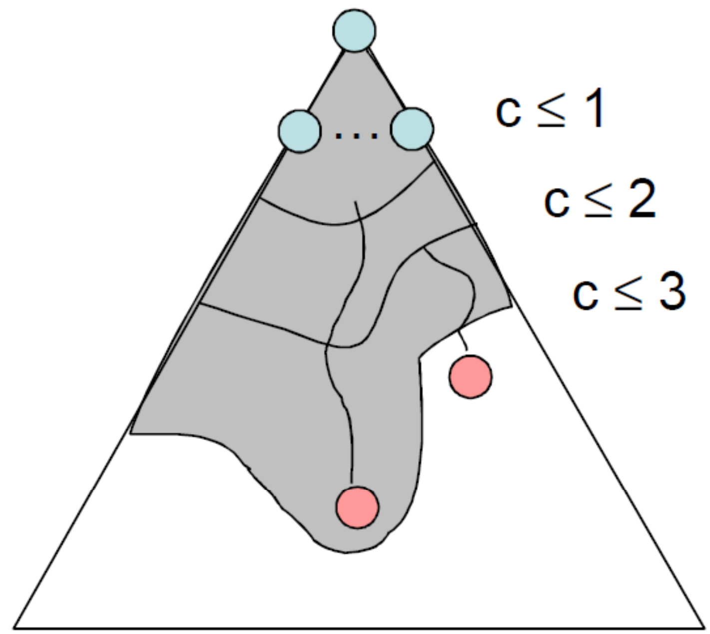
General Tree Search

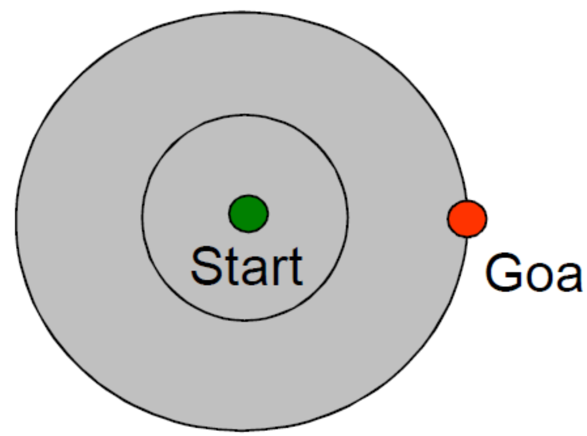
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the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



Uniform Cost Search

- Strategy: expand lowest path cost
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location

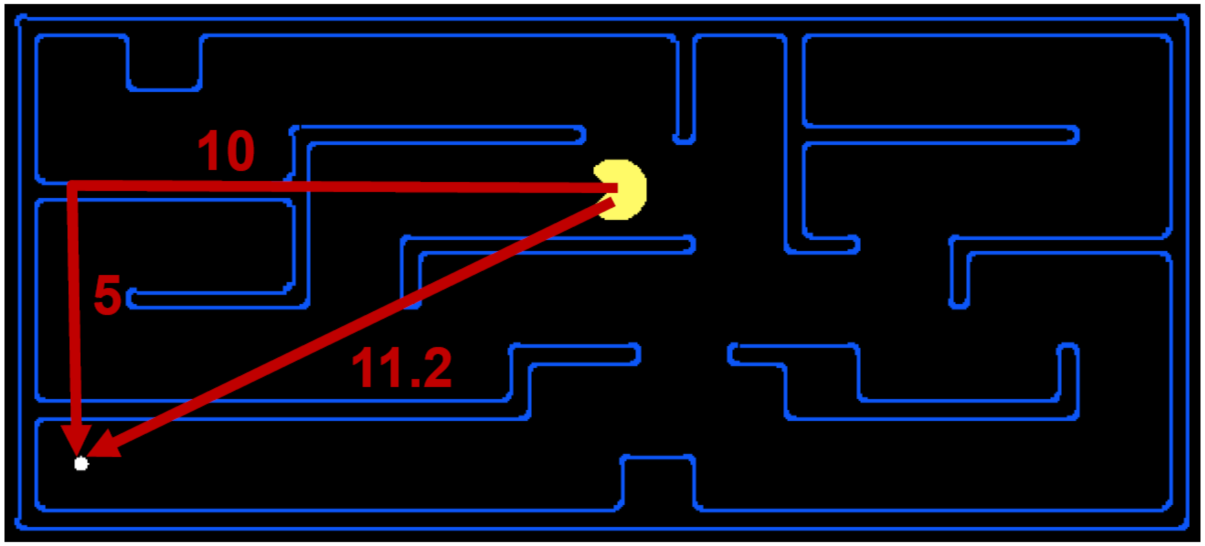




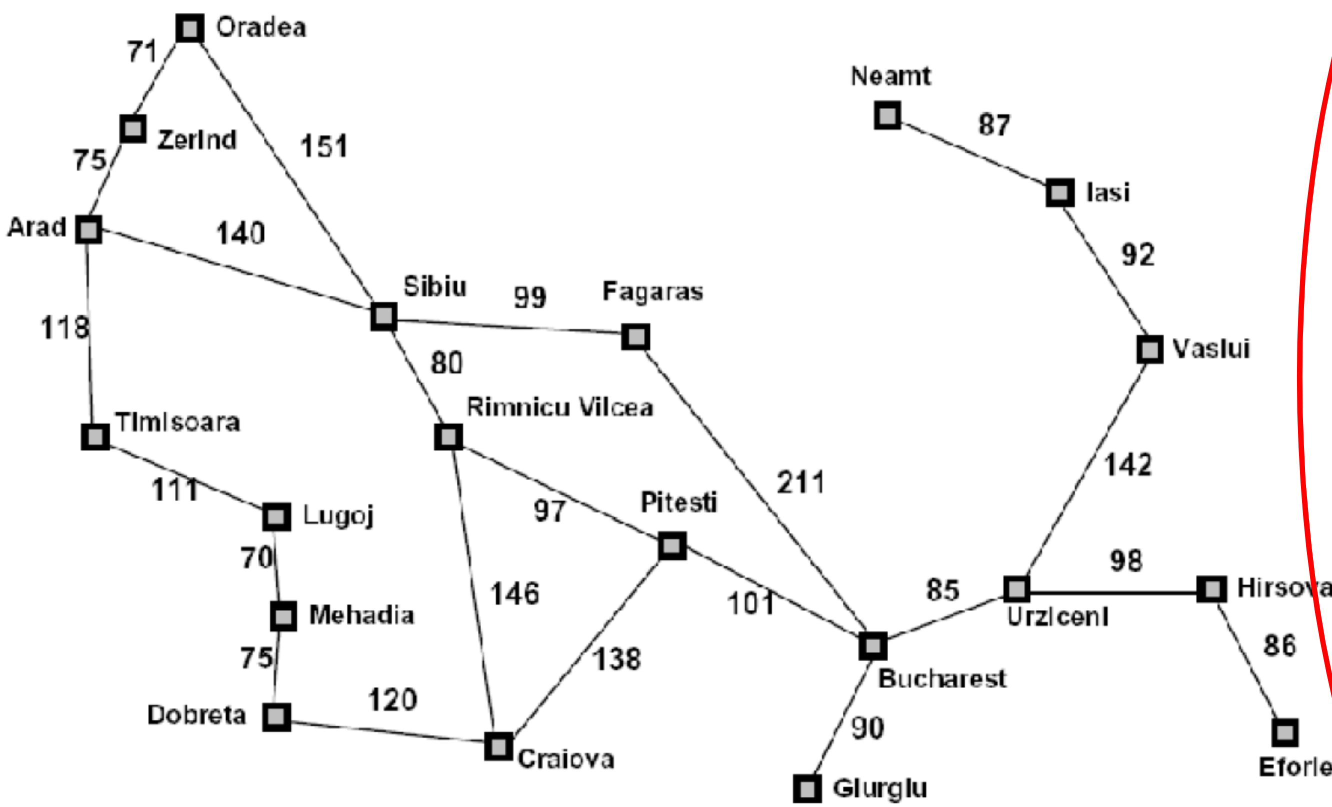
Search Heuristics

- Heuristic function $h(n)$ (a function from states to numbers):

  Any estimate of how close a state is to a goal ($h(n)$= 0 for goal node)
- Designed for each particular search problem
- Example: Manhattan distance, Euclidean distance



Example: Heuristic Function

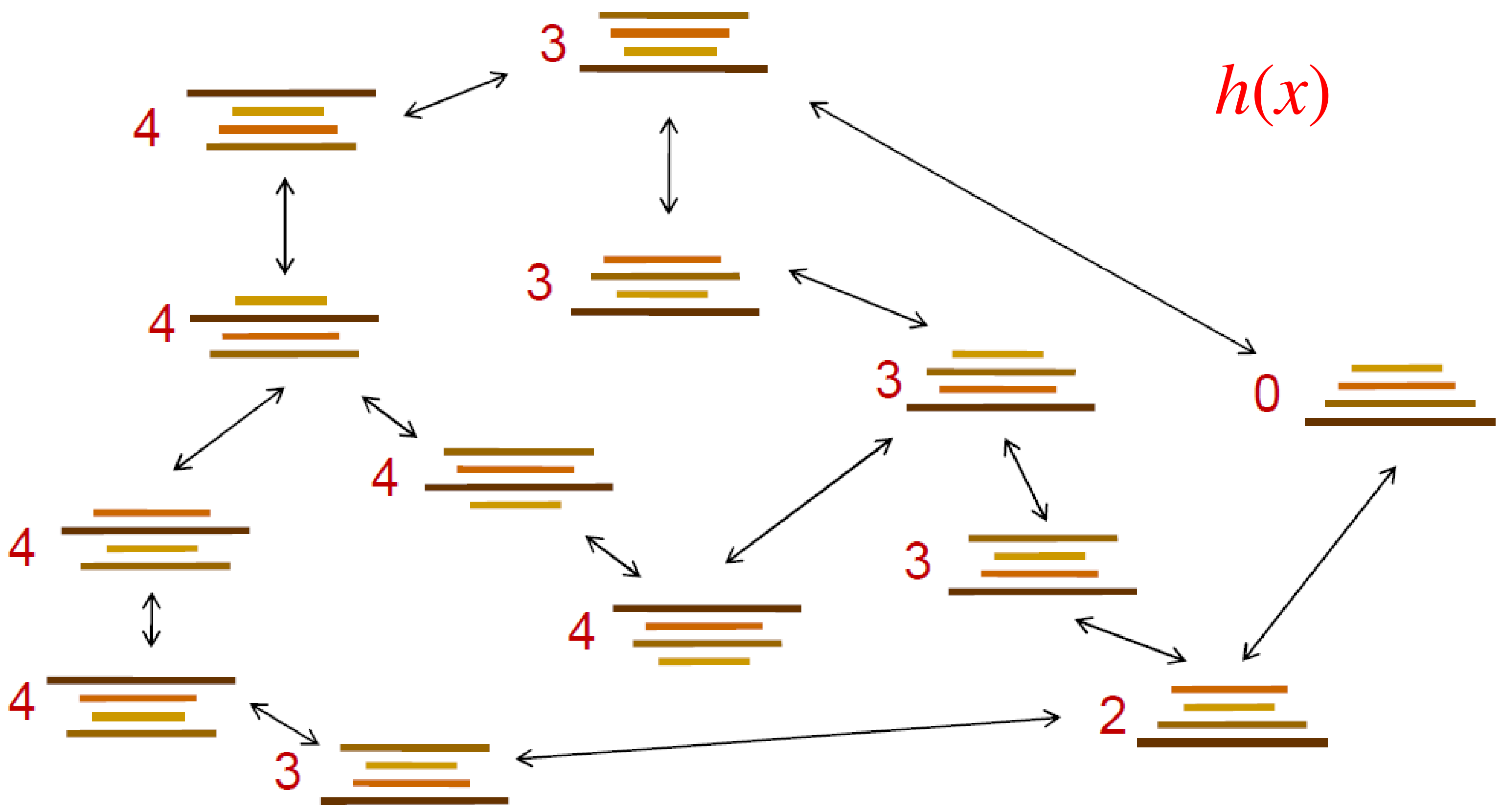


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(x)$

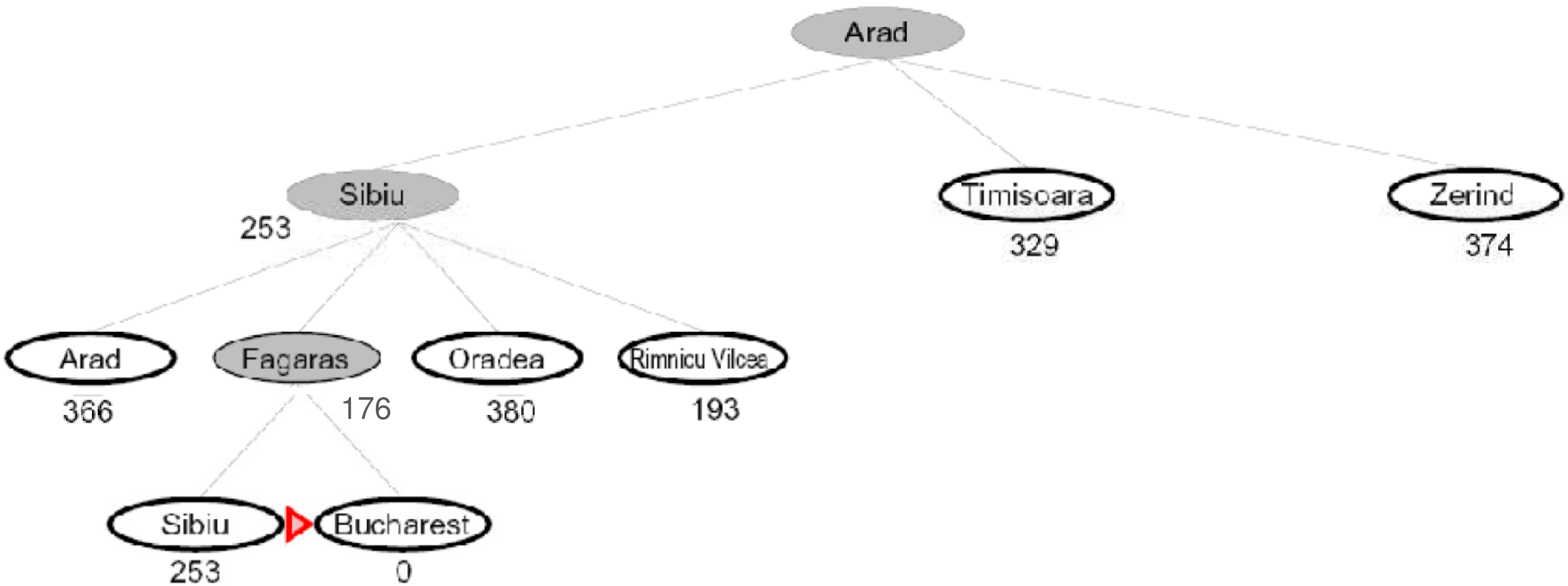
Example: Heuristic Function

Heuristic: the largest pancake that is still out of place



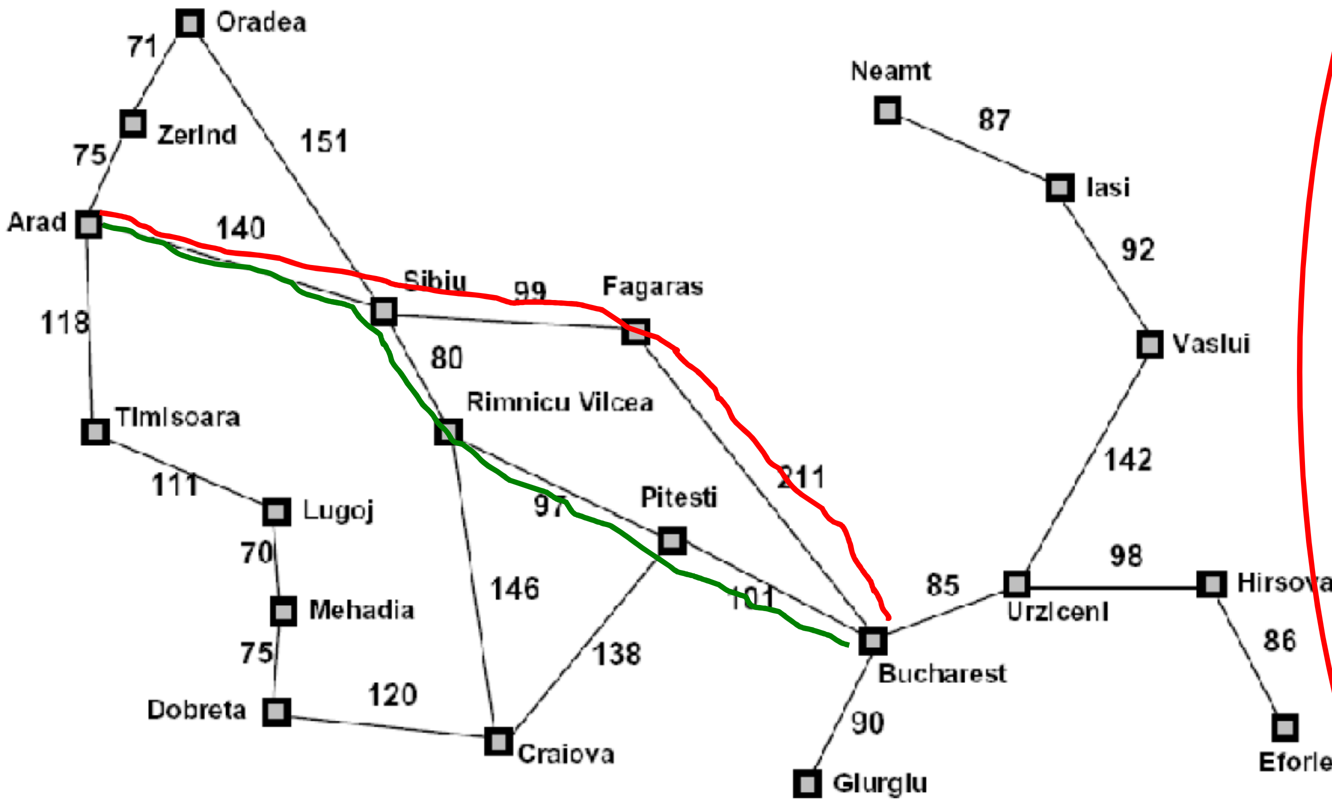
Greedy Best-First Search

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



- What can go wrong?

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

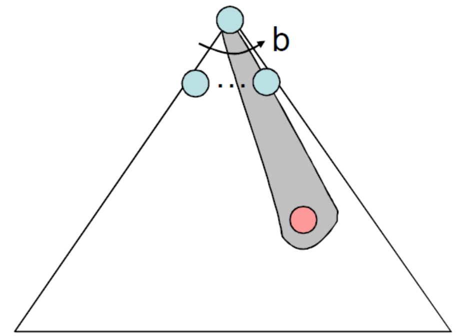
$h(x)$

Red Path (GBF) = 450

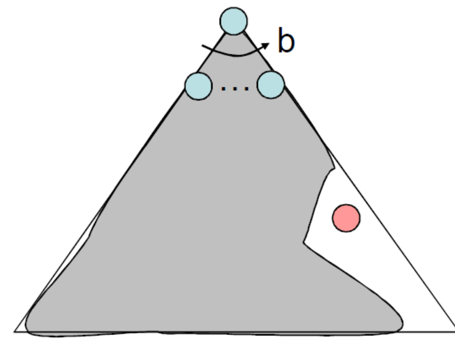
Green path(UCS)=418

Greedy Best-First Search

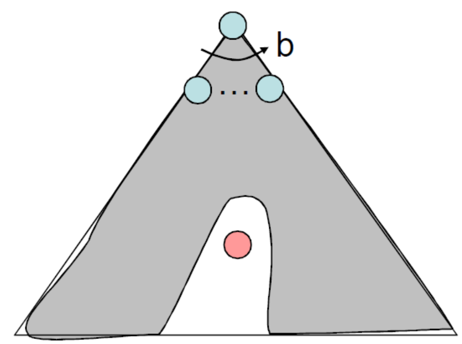
- A common case:
  - Best-first takes you straight to the (wrong) goal
- Worst-case: like a badly-guided DFS in the worst case
  - Can explore everything
  - Can get stuck in loops if no cycle checking
- Not optimal
  - heuristic is just an estimate to goal and GBF ignores the distance from root
- Like DFS in completeness
  - complete only if finite states with cycle checking



Best case

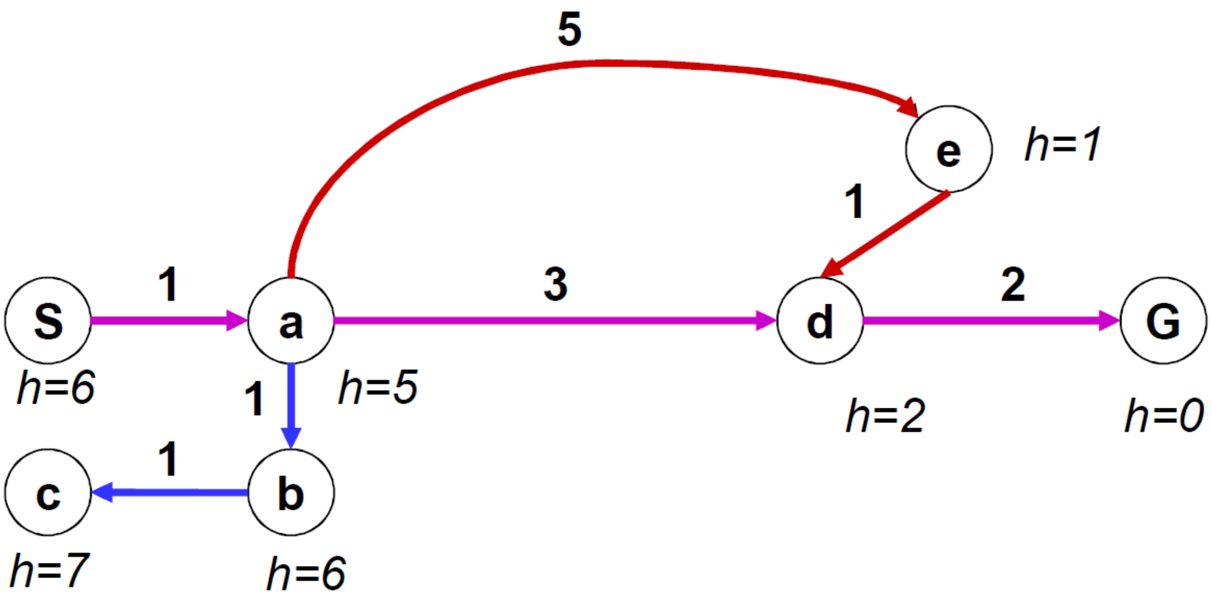


Worst case



Combining UCS and Greedy

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



UCS

s 0
s → a 1
s → a → b 2
s → a → d 4
s → a → e 6

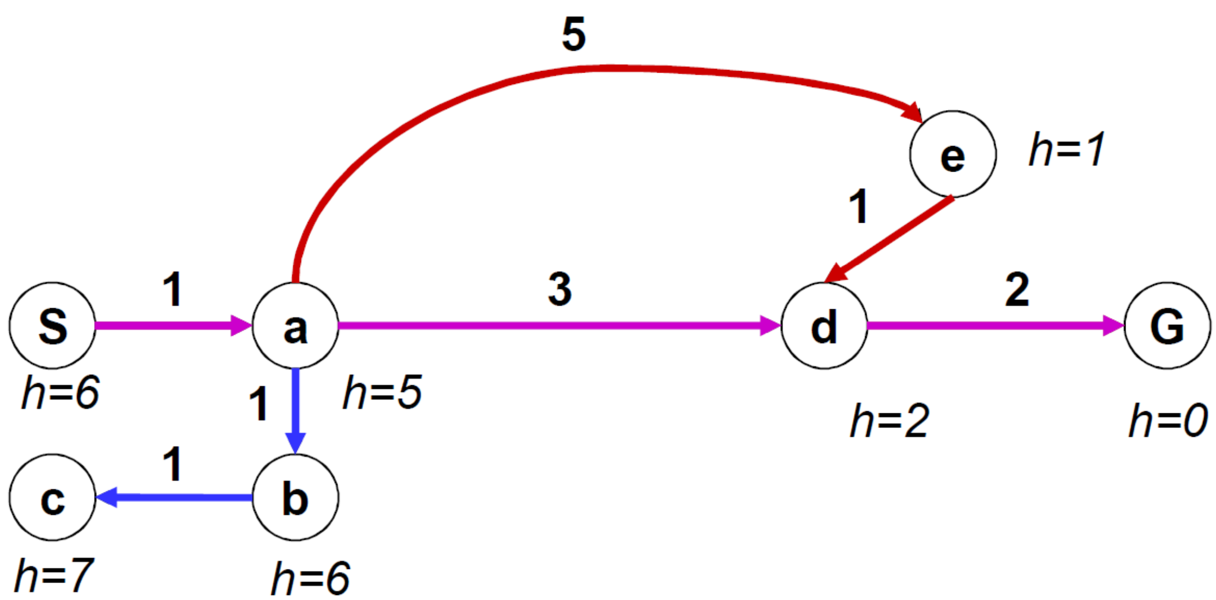
GBF

s 6
s → a 5
s → a → b 6
s → a → d 2
s → a → e 1
s → a → e → d 2

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

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*

s 0+6
s → a 1+5

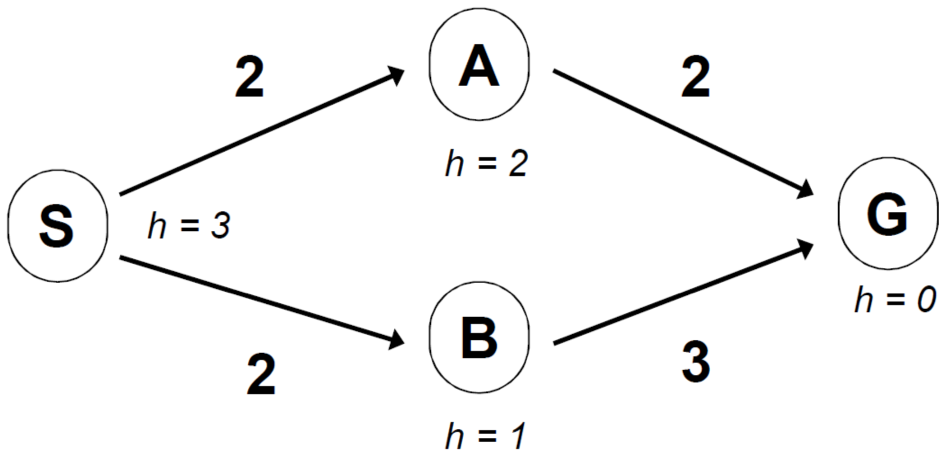
s → a → b 2+6
s → a → d 4+2
s → a → e 6+1

s → a → d → G 6+0

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

When should A* terminate?

- Should we stop when we enter a goal in the frontier?



A*

S 0+3

S → A 2+2

S → B 2+1

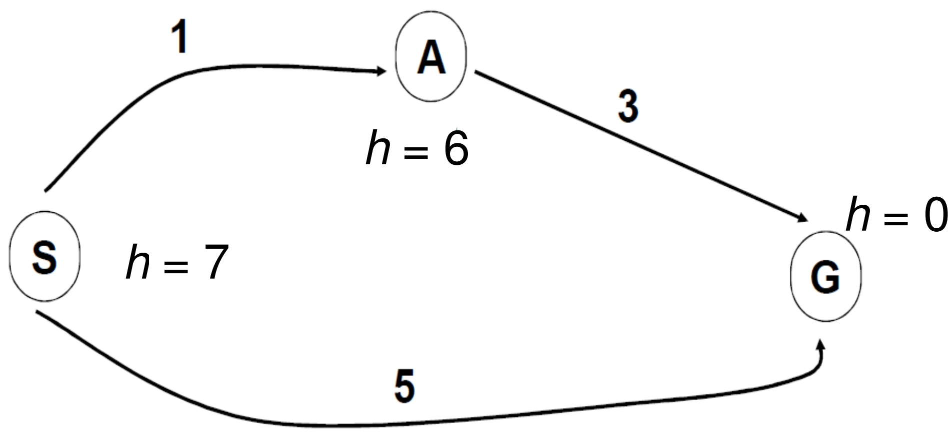
S → B → G 5+0

S → A → G 4+0

- No: only stop when we select a goal for expansion

Is A* optimal?

Overestimated *h*



A*

S 0+7

S → A 1+6

S → G 5+0

Answer:

S → G 5+0

Not optimal

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

Admissible Heuristics

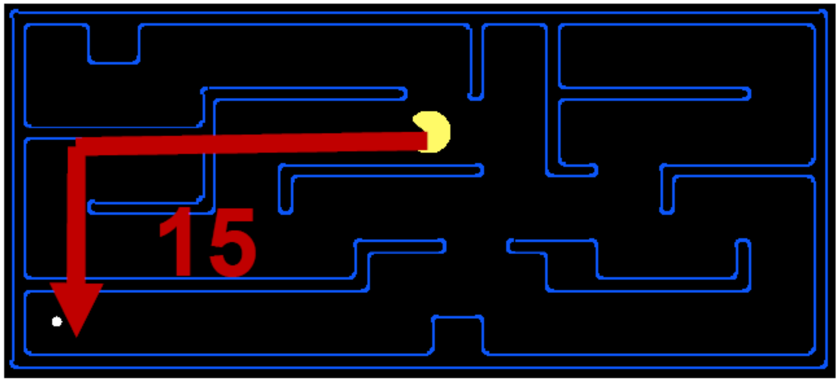
- A heuristic *h* is *admissible* if:

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

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

- Admissible heuristics are optimistic (underestimate the cost)

- Examples:



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

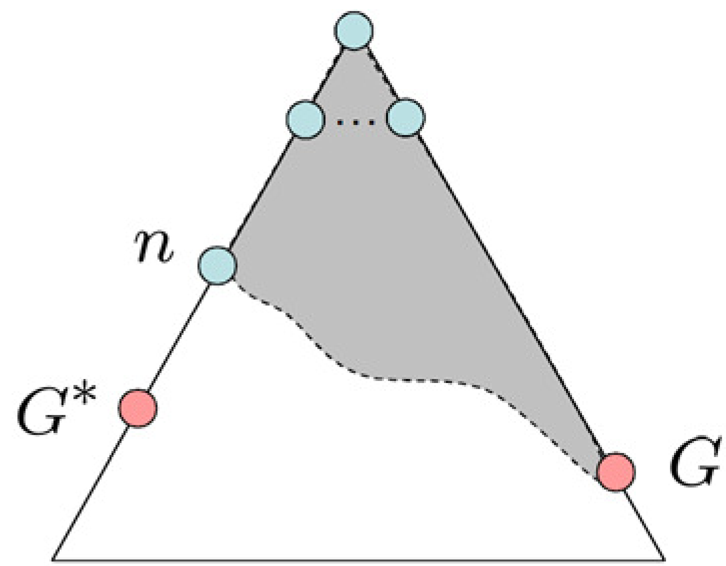
Optimality of A*: Blocking

Notation: …

- $g(n)$ = cost from root to node n
- $h(n)$ = estimated cost from n

  to the nearest goal (heuristic)
- $f(n) = g(n) + h(n)$ =

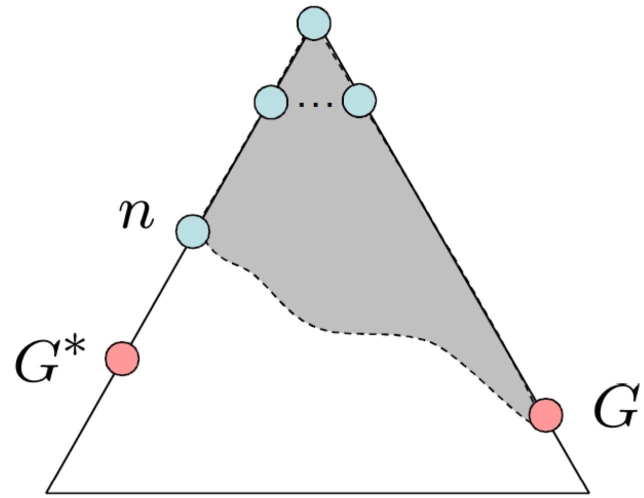
  estimated total cost via n
- G^*: a lowest cost goal node
- G: another goal node



Optimality of A*: Blocking

Proof:

- What could go wrong?
- We'd have to have to pop a suboptimal goal G off the fringe before G^*.



- This can't happen if h admissible:
  - Imagine a suboptimal goal G is on the queue
  - Some node n which is a subpath of G^* must also be on the fringe (why?)
  - n will be popped before G

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

$g(n) + h(n) \leq g(G^*)$ h admissible

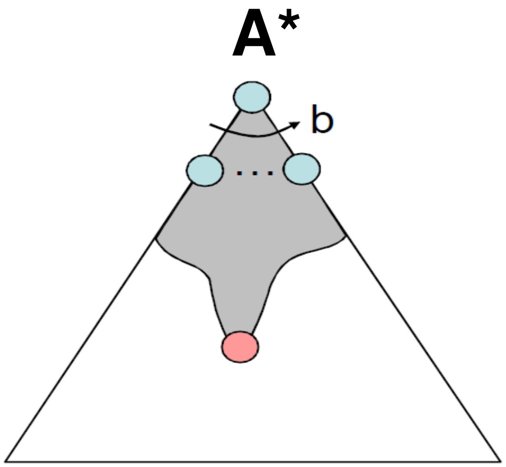
$g(G^*) < g(G)$ by assumption

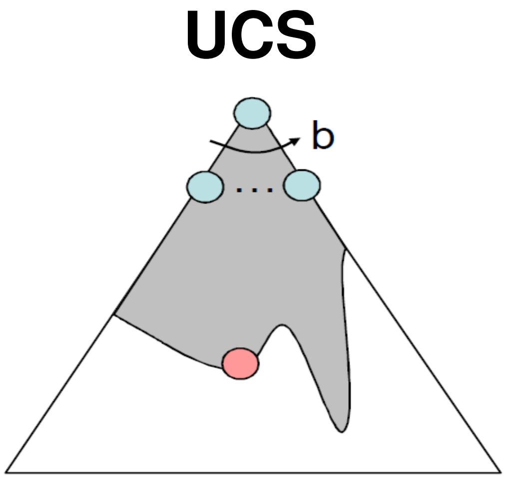
$g(G) = f(G)$ for goals $h(G)=0$

$f(n) < f(G)$

Properties of A*

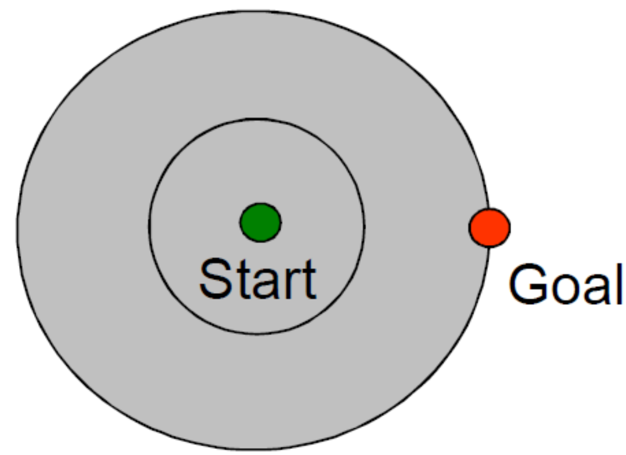
- A* does not expand any node with $f(n) > C^*$ (Pruning). While UCS might expand nodes with $g(n) < C^*$ but $f(n) > C^*$.
- Optimally efficient, no other algorithm guarantees to expand nodes less than A*. (but not good choice for every search problem)
- Complete if costs exceeds positive epsilon and b is finite
- Complexity $O(b^d)$!!!!



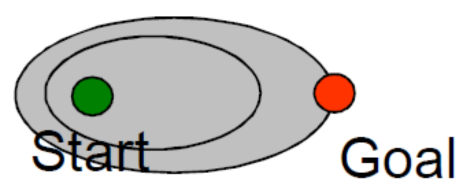


UCS vs. A* Contours

- Uniform-cost expanded in all directions (Contours of UCS are cheapest *g*)



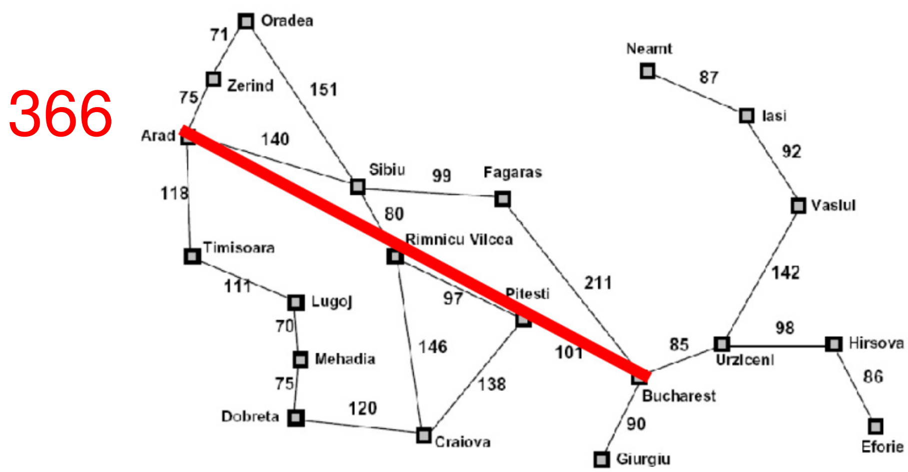
- A* expands mainly toward the goal, but does ensure optimality (Contours of A* are cheapest *f*.)

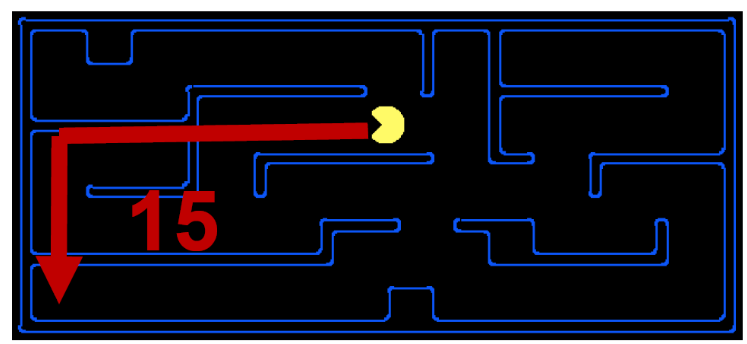


Creating Admissible Heuristics

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

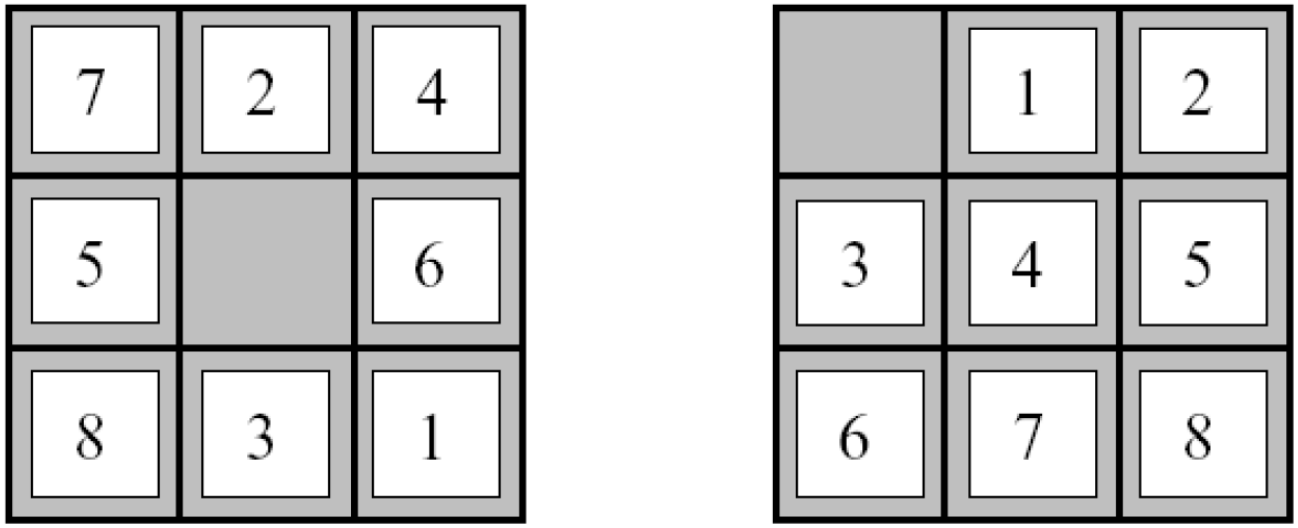
- Often, admissible heuristics are solutions to relaxed problems, where there are fewer restrictions on the actions (or new actions available)





- Inadmissible heuristics are often useful too (why?)

Example: 8 Puzzle



Start State Goal State

- What are the states?
- How many states?
- What are the actions?
- What states can I reach from the start state?
- What should the costs be?

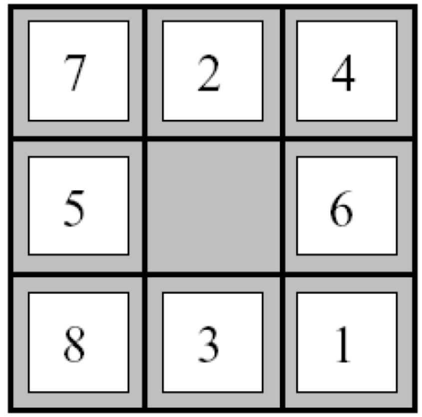
8 Puzzle (I)

- Heuristic 1:
  Number of tiles misplaced

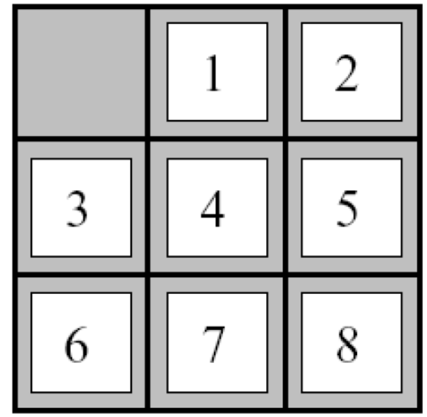
- Why is it admissible?

- h(start) = 8

- This is a relaxed-problem heuristic



Start State



Goal State

	Average nodes expanded when optimal path has length…		
	…4 steps	…8 steps	…12 steps
IDS	112	6,300	3.6×10^6
TILES	13	39	227

8 Puzzle (II)

- Heuristic 2:
  Sum of *Manhattan* distances of the tiles from their goal positions

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- Why admissible?

- h(start) =
  3+1+2+2+2+3+3+2 = 18

- This is also a relaxed-problem heuristic

	Average nodes expanded when optimal path has length…		
	…4 steps	…8 steps	…12 steps
TILES	13	39	227
MANHATTAN	12	25	73

8 Puzzle (III)

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

- With A*: a trade-off between quality of estimate and work per node!

Dominance

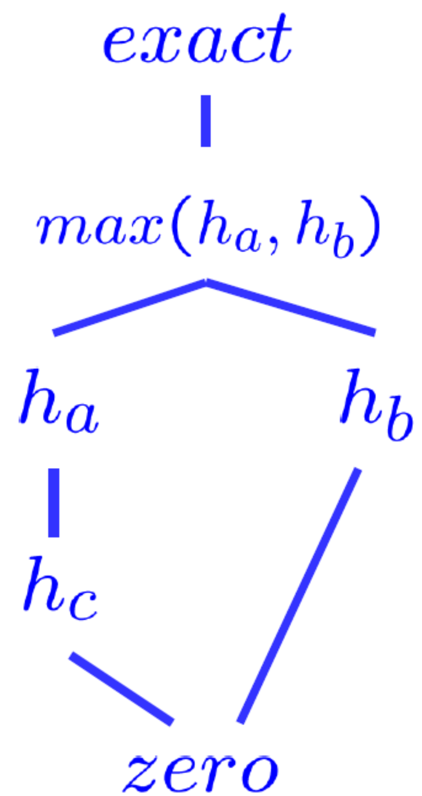
- Dominance: h_a dominates h_c if $\forall n : h_a(n) \geq h_c(n)$
- Dominance $\rightarrow$ efficiency

  A* using h_a never expands more nodes than A* using h_c.

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

  $$h(n) = \max\left(h_a(n) , h_b(n) \right)$$

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

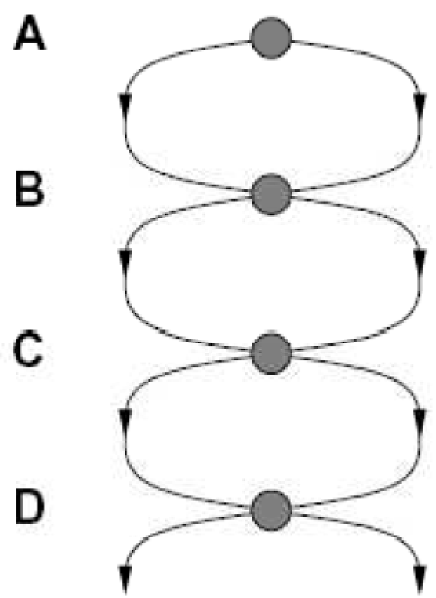


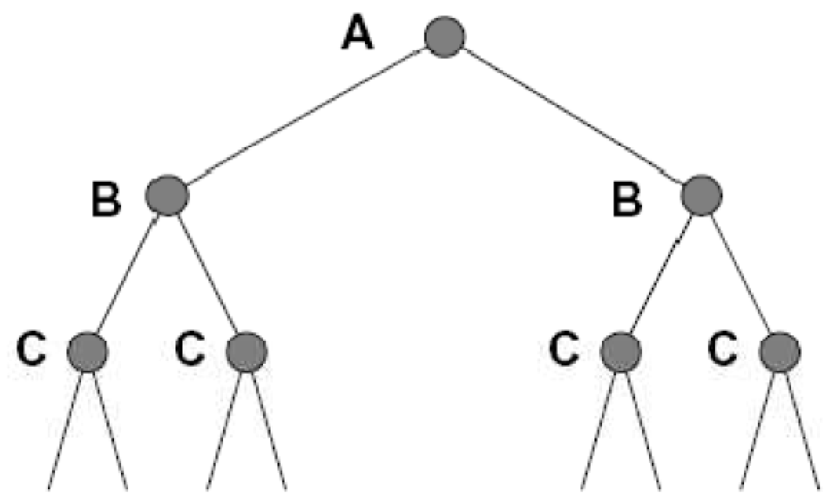
Other A* Applications

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

Tree Search: Extra Work!

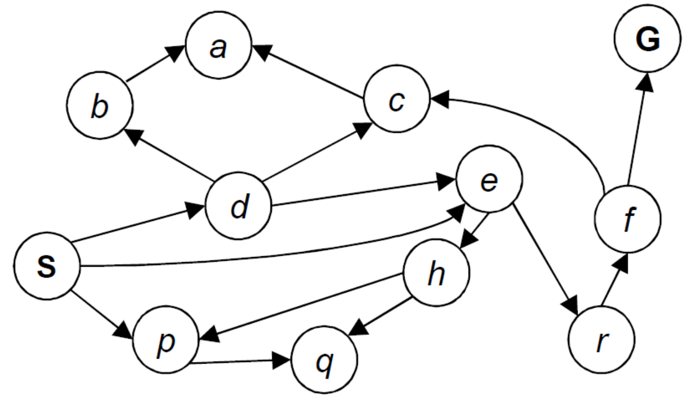
- Failure to detect repeated states can cause exponentially more work. Why?

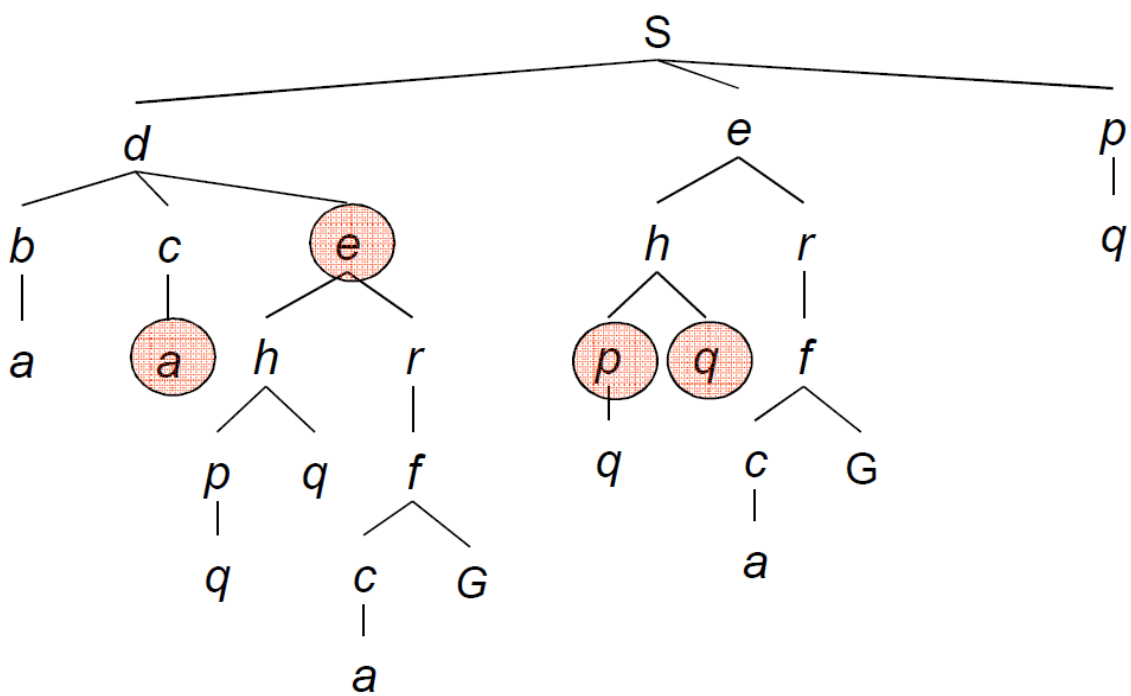




Example

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





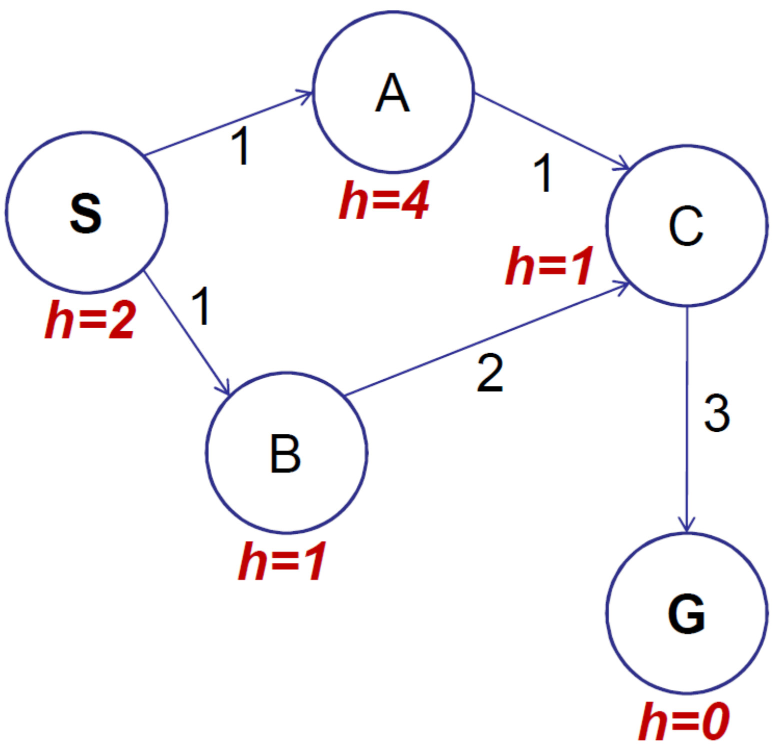
Graph Search

- 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 is new (neither in expanded set nor in frontier)
  - If not new, skip it

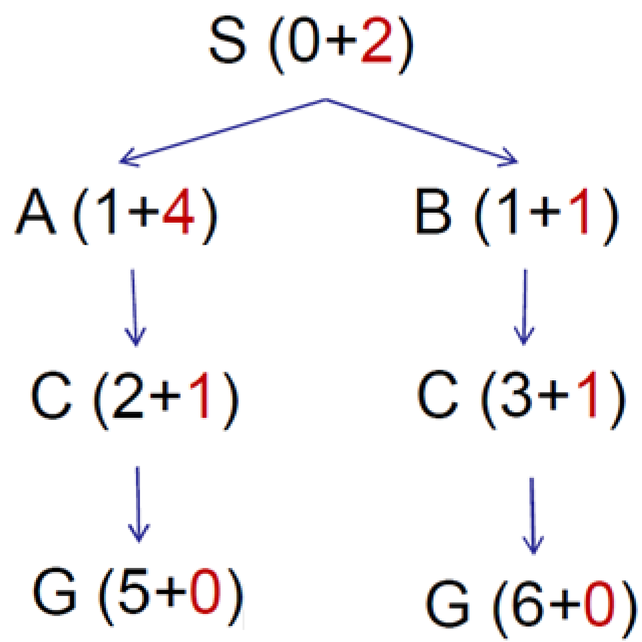
- Important: store the closed-set as a set, not a list
- Can graph search wreck completeness? Why/why not?
- How about optimality?

A* Graph Search Gone Wrong?

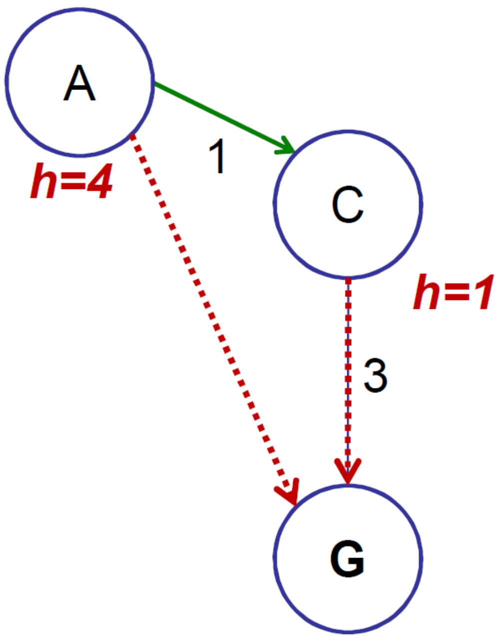
State space graph



Search tree



Consistency of Heuristics



- Stronger than admissibility

- Definition:

  cost(A to C) + h(C) $\geq$ h(A)

  cost(A to C) $\geq$ h(A) - h(C)

  real arc cost $\geq$ cost implied by heuristic

- Consequences:
  - The *f* value along a path never decreases
  - A* graph search is optimal

Optimality

- Tree search:
  - A* is optimal if heuristic is admissible (and non-negative)
  - UCS is a special case of A* (with $h = 0$)

- Graph search:
  - A* optimal if heuristic is consistent
  - UCS is optimal (h = 0 is consistent)

- Consistency implies admissibility

- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems

Summary: A*

- 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