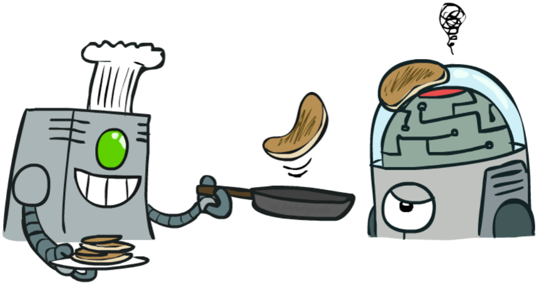
Informed Search

CSE 4711: Artificial Intelligence

Md. Bakhtiar Hasan

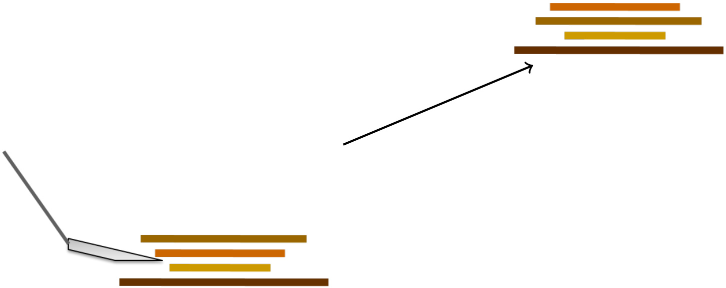
Assistant Professor
Department of Computer Science and Engineering
Islamic University of Technology

Example: Pancake Problem

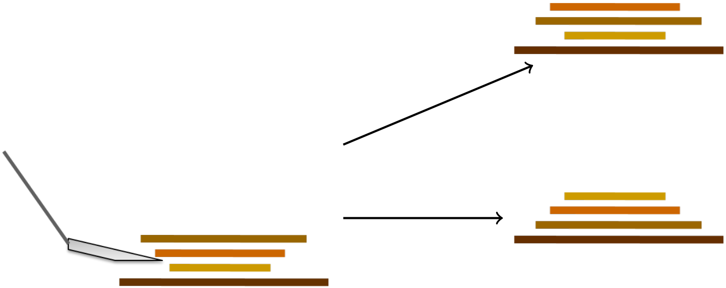


Example: Pancake Problem

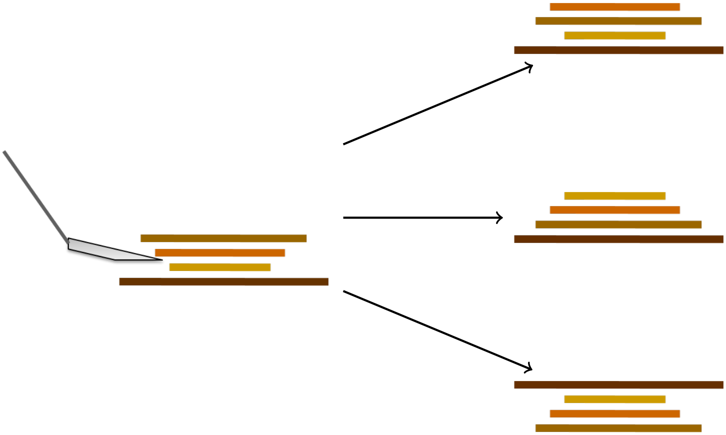
Example: Pancake Problem



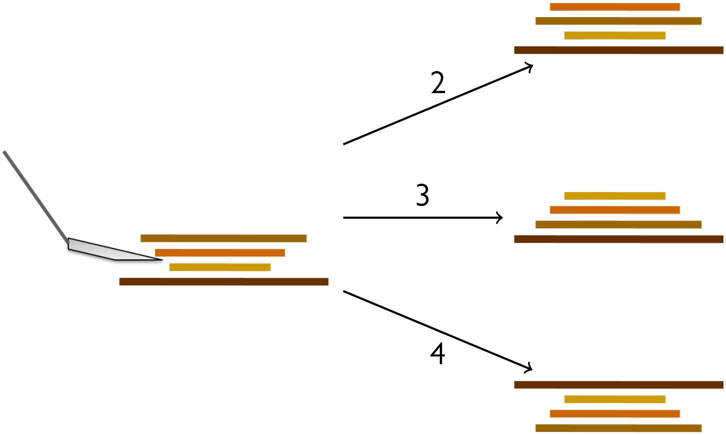
Example: Pancake Problem



Example: Pancake Problem



Example: Pancake Problem



Cost: Number of pancakes flipped

Example: Pancake Problem

BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES

Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†

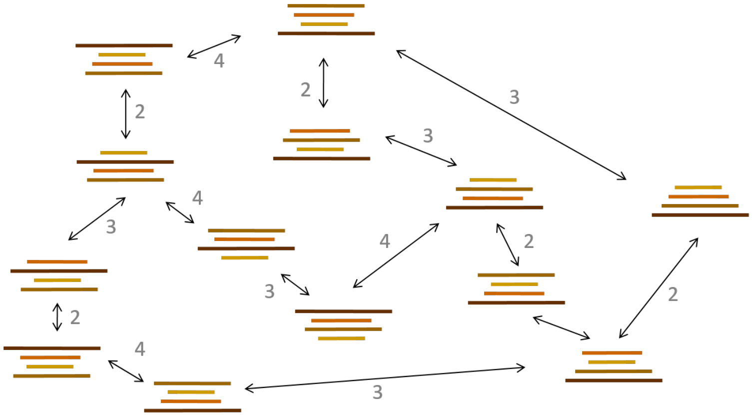
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation σ of the integers from 1 to n, let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n. We show that $f(n) \leqslant (5n+5)/3$, and that $f(n) \geqslant 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2 - 1 \leqslant g(n) \leqslant 2n + 3$.

Example: Pancake Problem

State space graph with costs as weights[1]



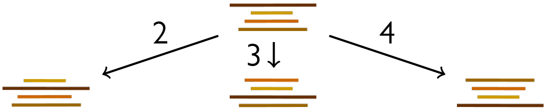
[1] Slide does not contain entire state space graph

General Search Tree

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 according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting node to the search tree
  end

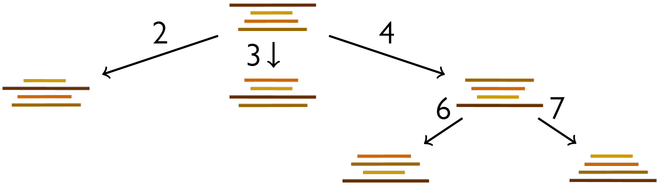
General Search Tree

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 according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting node to the search tree
  end



General Search Tree

```
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 according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting node to the search tree
  end
```



Informed Search

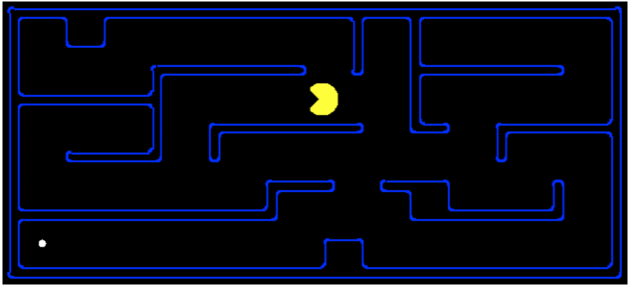
Video: ContoursPacmanSmallMaze-UCS

Search Heuristics

- A function that *estimates* how close a state is to a goal
- Designed for a particular search problem

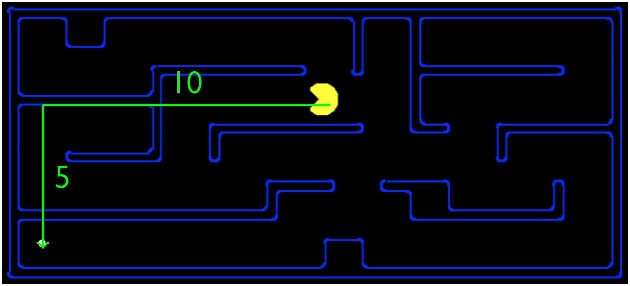
Search Heuristics

- A function that *estimates* how close a state is to a goal
- Designed for a particular search problem



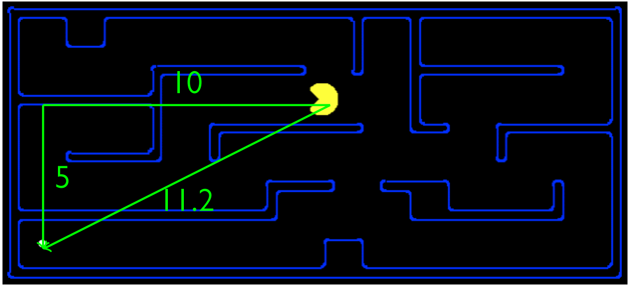
Search Heuristics

- A function that *estimates* how close a state is to a goal
- Designed for a particular search problem
- Example: Manhatten distance

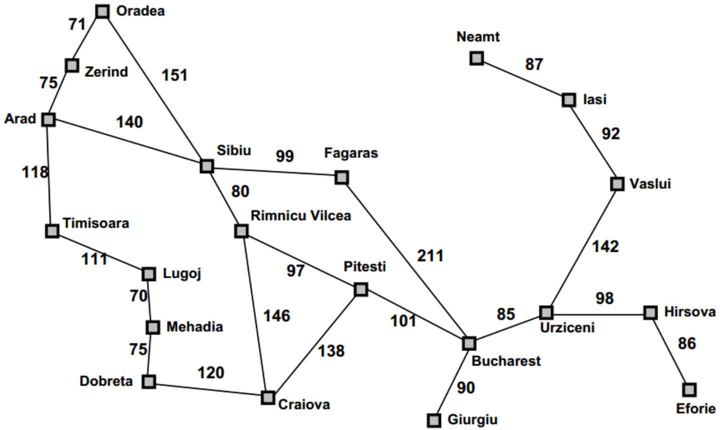


Search Heuristics

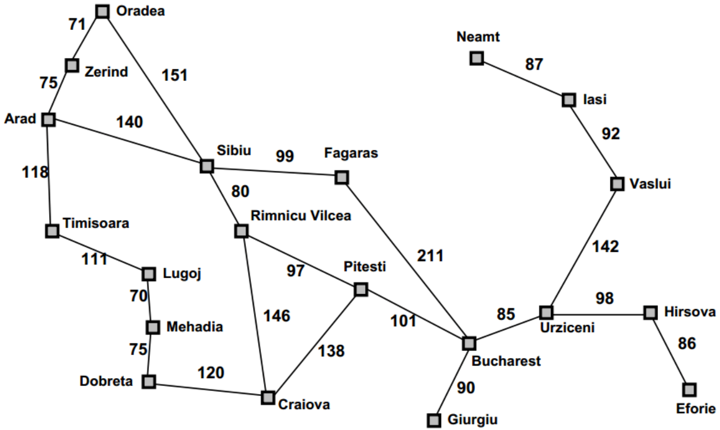
- A function that *estimates* how close a state is to a goal
- Designed for a particular search problem
- Example: Manhatten distance, Euclidean distance for pathing



Example: Heuristic Function

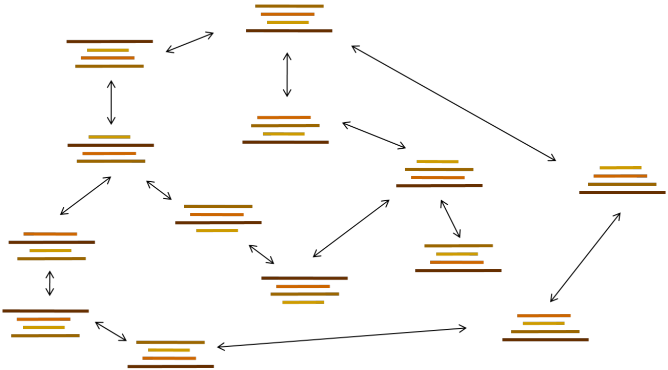


Example: Heuristic Function

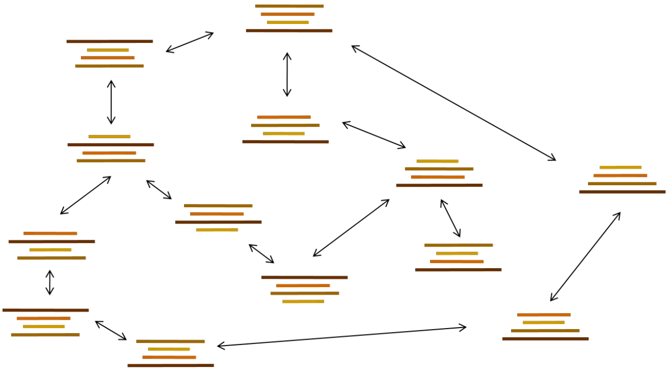


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnieu Vileea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: Heuristic Function

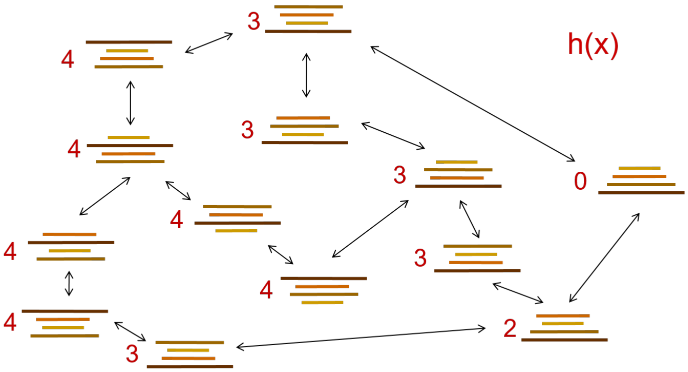


Example: Heuristic Function



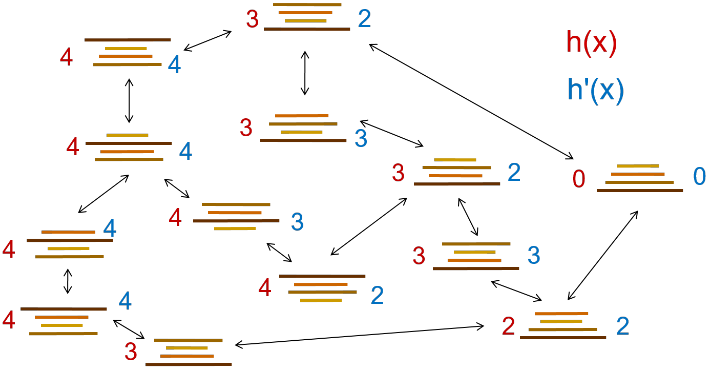
Bad heuristic: The number of correctly positioned pancakes

Example: Heuristic Function



$h(x)$ = The ID of the largest pancake that is still out of place

Example: Heuristic Function



$h(x)$ = The ID of the largest pancake that is still out of place
$h'(x)$ = The number of the incorrectly placed pancakes

Greedy Search

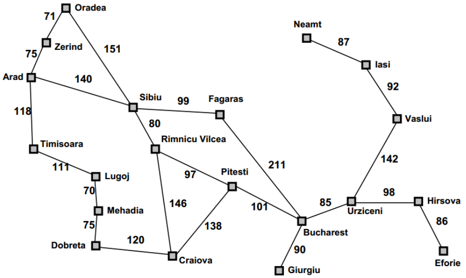
Greedy Search

- Expand the node that seems closest

Greedy Search

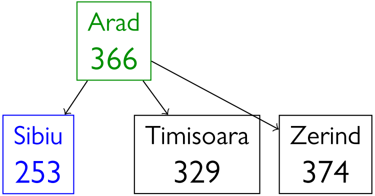
- Expand the node that seems closest

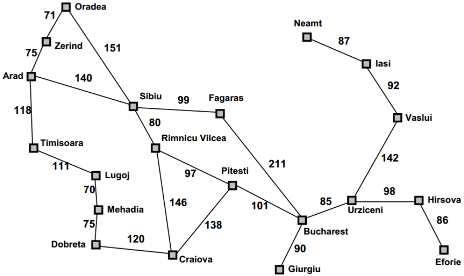
Arad
366



Greedy Search

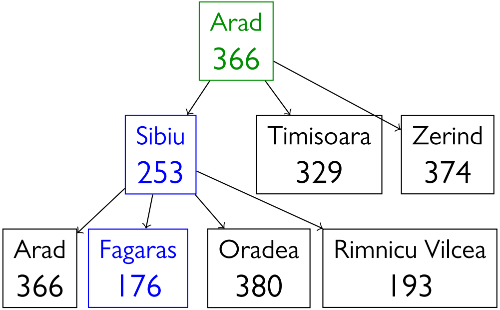
- Expand the node that seems closest

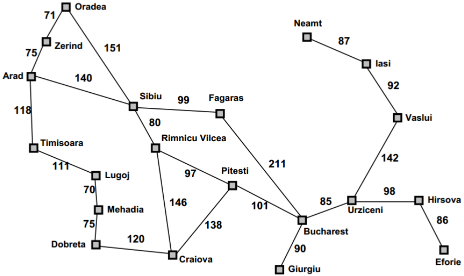




Greedy Search

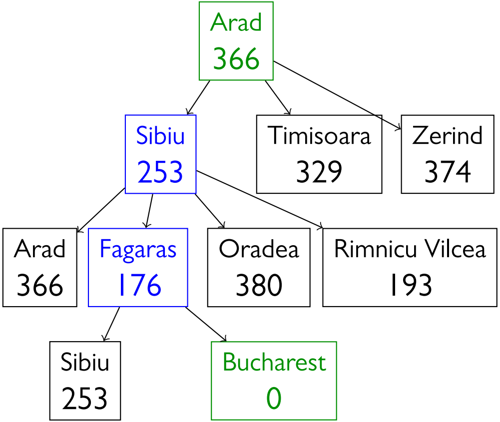
- Expand the node that seems closest

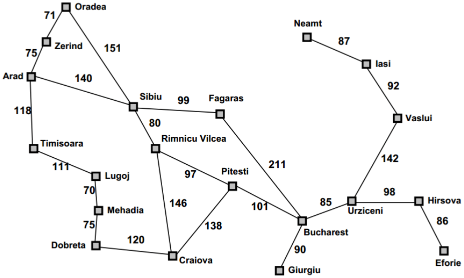




Greedy Search

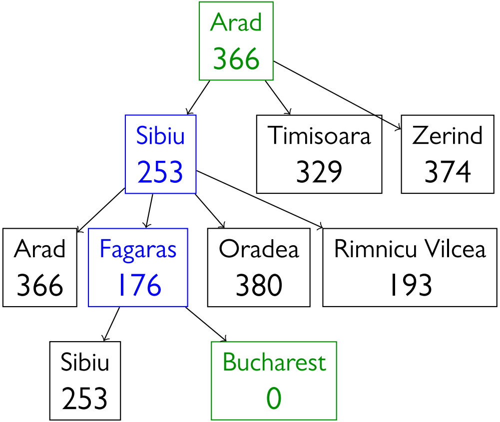
- Expand the node that seems closest

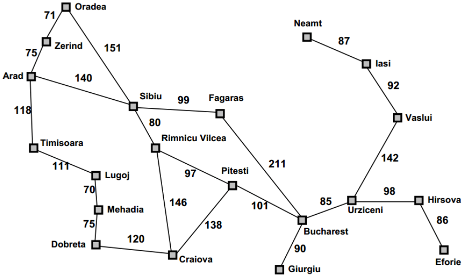


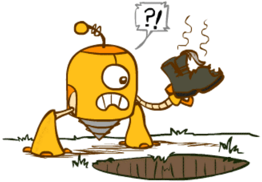


Greedy Search

- Expand the node that seems closest

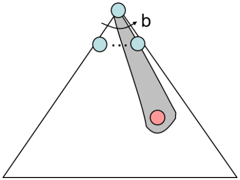






Greedy Search

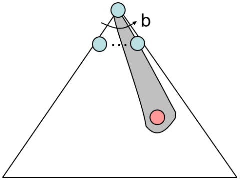
- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state



Video: Empty-greedy, ContoursPacmanSmallMaze-greedy

Greedy Search

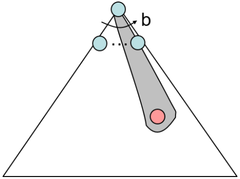
- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state
- A common case:
  - Best-first takes you straight to the (wrong) goal

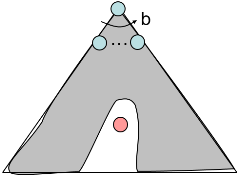


Video: Empty-greedy, ContoursPacmanSmallMaze-greedy

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





Video: Empty-greedy, ContoursPacmanSmallMaze-greedy

A* Search

A* Search

UCS

A* Search

UCS

Greedy

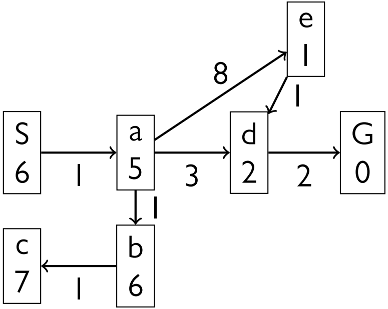
A* Search

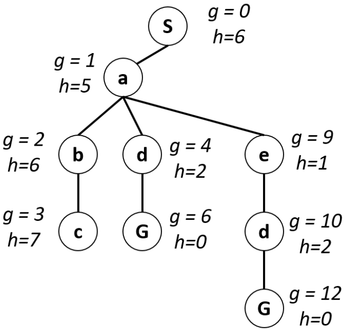
UCS

Greedy

A*

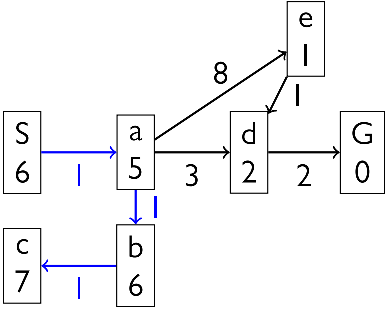
Combining UCS and Greedy

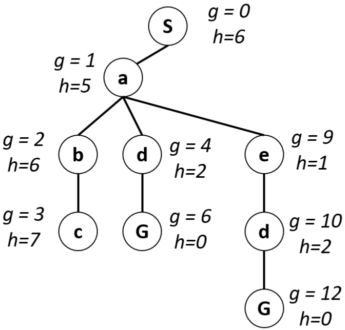




Combining UCS and Greedy

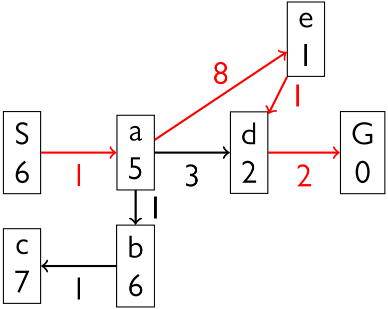
- Uniform-cost orders by path cost, or *backward cost* $g(n)$

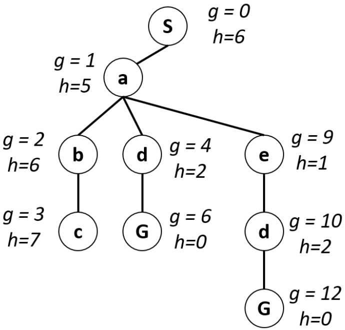




Combining UCS and Greedy

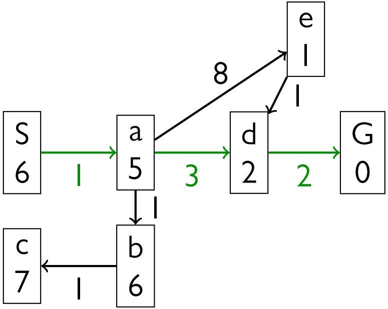
- Uniform-cost orders by path cost, or *backward cost* $g(n)$
- Greedy orders by goal proximity, or *forward cost* $h(n)$

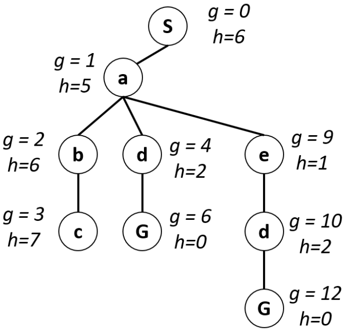




Combining UCS and Greedy

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



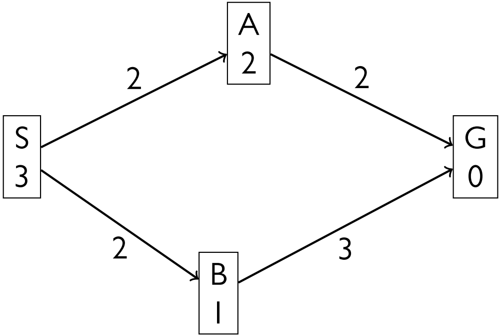


When should A* terminate?

- Should we stop when we enqueue a goal?

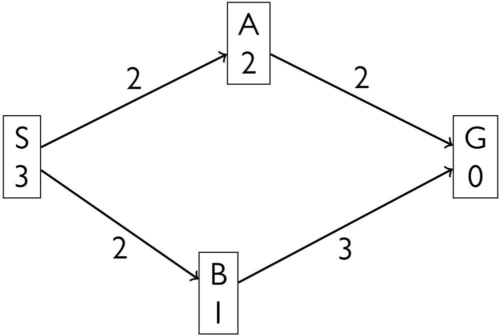
When should A* terminate?

- Should we stop when we enqueue a goal?



When should A* terminate?

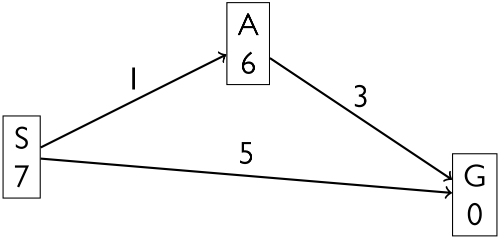
- Should we stop when we enqueue a goal?



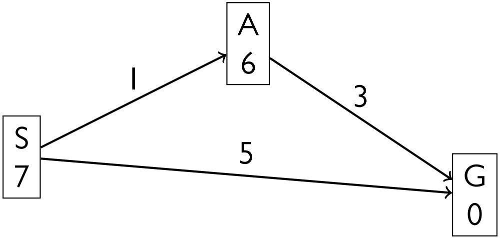
- No: only stop when you dequeue the goal

Is A* optimal?

Is A* optimal?

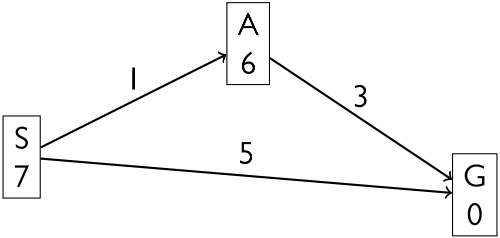


Is A* optimal?



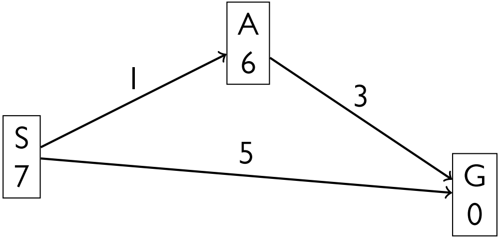
- What went wrong?

Is A* optimal?



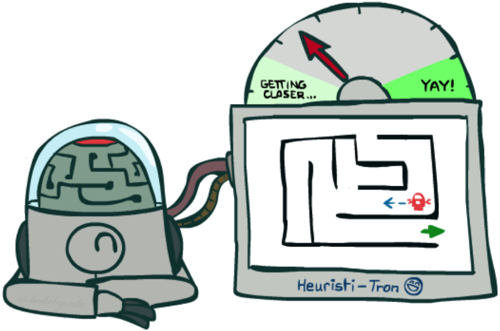
- What went wrong?
  - Actual bad goal cost $<$ estimated good goal cost

Is A* optimal?

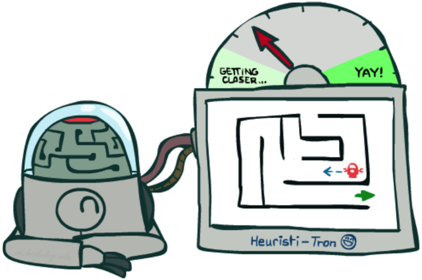


- What went wrong?
  - Actual bad goal cost < estimated good goal cost
- We need estimates to be less than the actual cost

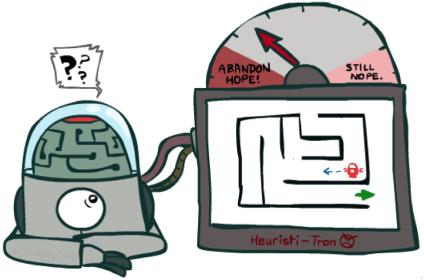
Admissible Heuristics



Admissible Heuristics



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



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

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

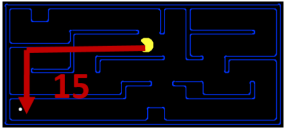
Admissible Heuristics

- 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

- Example:



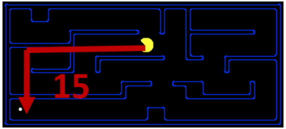
Admissible Heuristics

- 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

- Example:



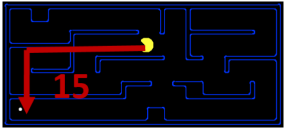
Admissible Heuristics

- A heuristic h is admissible (optimistic) if:

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

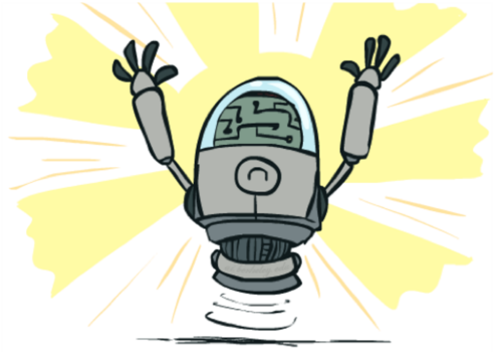
where $h^*(n)$ is the true cost to a nearest goal

- Example:



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

Optimality of A* Tree Search



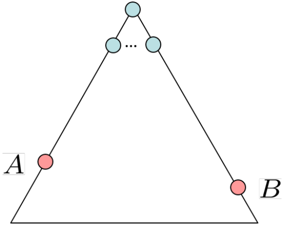
Optimality of A* Tree Search

Assume:

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

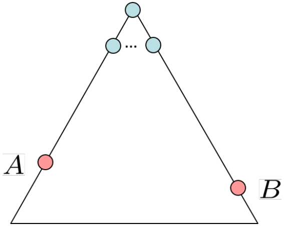
Claim:

- A will exit the fringe before B



Optimality of A* Tree Search

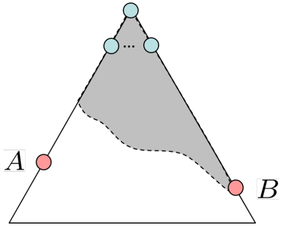
Proof:



Optimality of A* Tree Search

Proof:

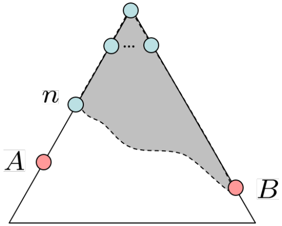
- Imagine B is on the fringe



Optimality of A* Tree Search

Proof:

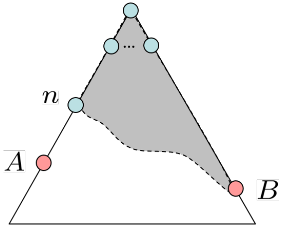
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)



Optimality of A* Tree Search

Proof:

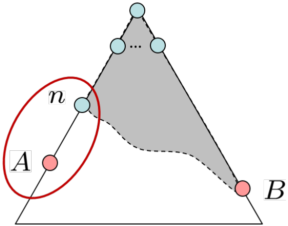
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B



Optimality of A* Tree Search

Proof:

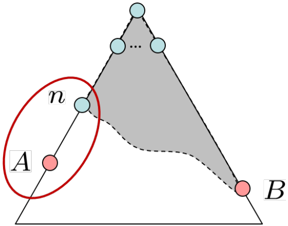
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$



Optimality of A* Tree Search

Proof:

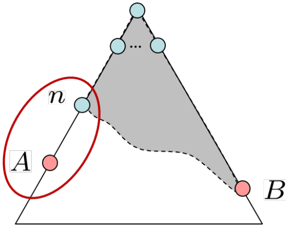
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]



Optimality of A* Tree Search

Proof:

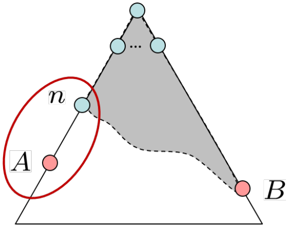
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]
     - $f(n) \leq g(A)$ [Admissiblity of heuristics]



Optimality of A* Tree Search

Proof:

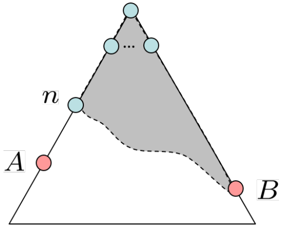
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]
     - $f(n) \leq g(A)$ [Admissiblity of heuristics]
     - $g(A) = f(A)$ [h(A)=0 at goal]



Optimality of A* Tree Search

Proof:

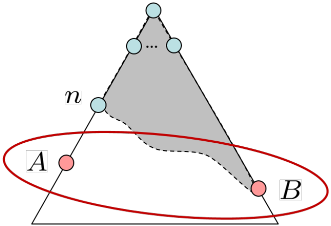
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]
     - $f(n) \leq g(A)$ [Admissiblity of heuristics]
     - $g(A) = f(A)$ [h(A)=0 at goal]
  2. $f(A) < f(B)$



Optimality of A* Tree Search

Proof:

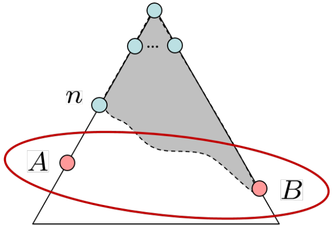
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]
     - $f(n) \leq g(A)$ [Admissiblity of heuristics]
     - $g(A) = f(A)$ [h(A)=0 at goal]
  2. $f(A) < f(B)$
     - $g(A) < g(B)$ [B is suboptimal]



Optimality of A* Tree Search

Proof:

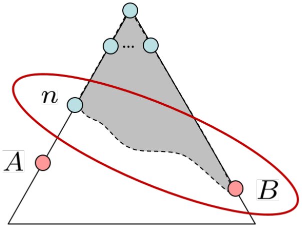
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]
     - $f(n) \leq g(A)$ [Admissiblity of heuristics]
     - $g(A) = f(A)$ [h(A)=0 at goal]
  2. $f(A) < f(B)$
     - $g(A) < g(B)$ [B is suboptimal]
     - $f(A) < f(B)$ [h=0 at goal]



Optimality of A* Tree Search

Proof:

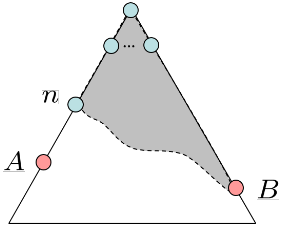
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]
     - $f(n) \leq g(A)$ [Admissiblity of heuristics]
     - $g(A) = f(A)$ [h(A)=0 at goal]
  2. $f(A) < f(B)$
     - $g(A) < g(B)$ [B is suboptimal]
     - $f(A) < f(B)$ [h=0 at goal]
  3. $f(n) \leq f(A) < f(B) \rightarrow n$ expands before B



Optimality of A* Tree Search

Proof:

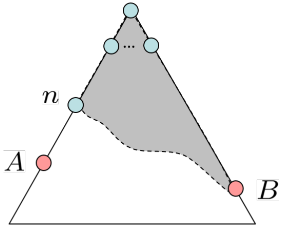
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]
     - $f(n) \leq g(A)$ [Admissiblity of heuristics]
     - $g(A) = f(A)$ [h(A)=0 at goal]
  2. $f(A) < f(B)$
     - $g(A) < g(B)$ [B is suboptimal]
     - $f(A) < f(B)$ [h=0 at goal]
  3. $f(n) \leq f(A) < f(B) \rightarrow n$ expands before B
- All ancestor of A expand before B



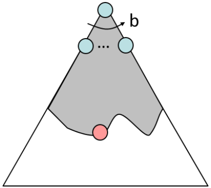
Optimality of A* Tree Search

Proof:

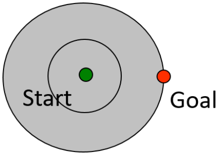
- Imagine B is on the fringe
- Some ancestor n of A is also on the fringe, too (maybe A)
- Claim: n will be expanded before B
  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$ [Definition of f-cost]
     - $f(n) \leq g(A)$ [Admissiblity of heuristics]
     - $g(A) = f(A)$ [h(A)=0 at goal]
  2. $f(A) < f(B)$
     - $g(A) < g(B)$ [B is suboptimal]
     - $f(A) < f(B)$ [h=0 at goal]
  3. $f(n) \leq f(A) < f(B) \rightarrow n$ expands before B
- All ancestor of A expand before B
- A expands before B $\rightarrow$ A* search is optimal

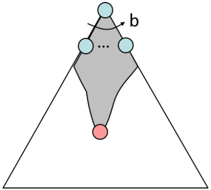


UCS vs A*

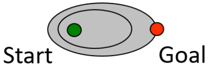


UCS





A*

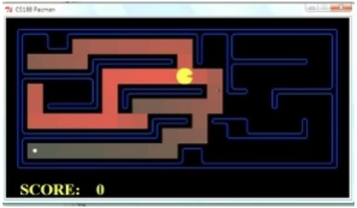


Video: Empty-UCS, Empty-astar, ContoursPacmanSmallMaze-astar.mp4

UCS vs A*

Greedy

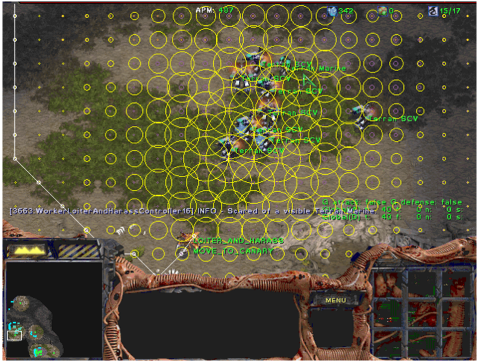
UCS



A*

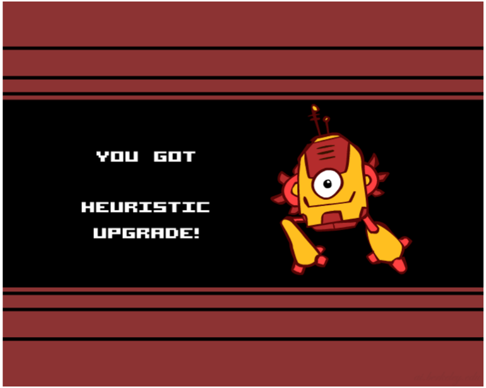
A* Applications

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



Video: tinyMaze, guessAlgorithm

Creating Admissible Heuristics

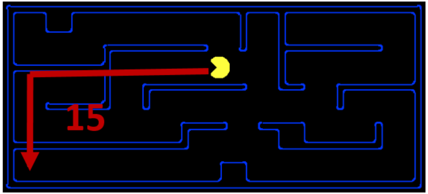


Creating Admissible Heuristics

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

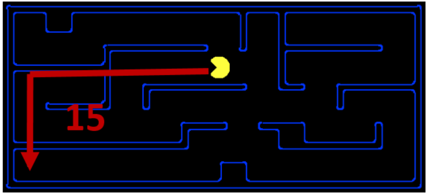
Creating Admissible Heuristics

- 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



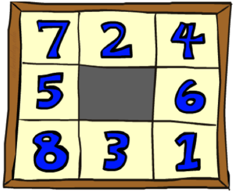
Creating Admissible Heuristics

- 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

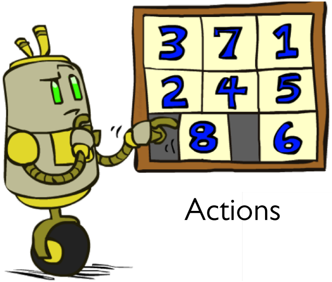


- Inadmissible heuristics are often useful too

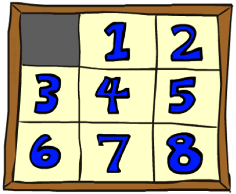
Example: 8 Puzzle



Start State



Actions



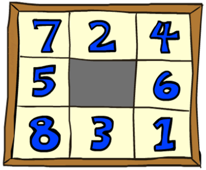
Goal State

- What are the states? $\rightarrow$ Puzzle configurations
- How many states? $\rightarrow 9!$
- What are the actions? $\rightarrow$ Move the empty piece in four directions
- How many successors are there from the start state? $\rightarrow 4$
- What should the cost be? $\rightarrow$ Number of moves

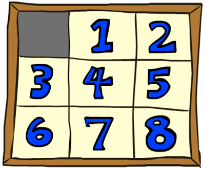
Example: 8 Puzzle

Attempt 1:

- Number of misplaced tiles



Start State

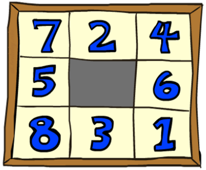


Goal State

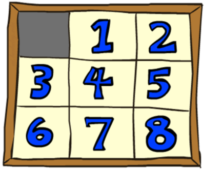
Example: 8 Puzzle

Attempt 1:

- Number of misplaced tiles
- Why is it admissible?



Start State

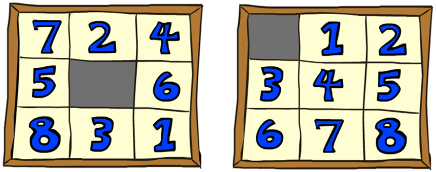


Goal State

Example: 8 Puzzle

Attempt 1:

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

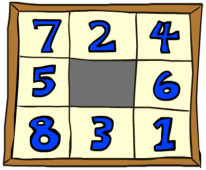


Start State Goal State

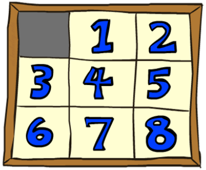
Example: 8 Puzzle

Attempt 1:

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



Start State



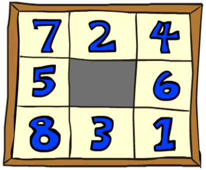
Goal State

	Average nodes expanded when the optimal path has...		
	...4 steps	...8 steps	...12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

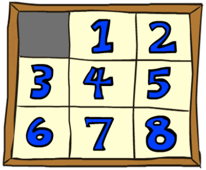
Example: 8 Puzzle

Attempt 1:

- Number of misplaced tiles
- Why is it admissible?
- $h(start) = 8$
- *Relaxed-problem* heuristic



Start State



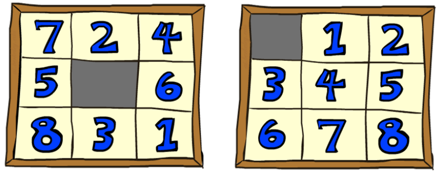
Goal State

	Average nodes expanded when the optimal path has...		
	...4 steps	...8 steps	...12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

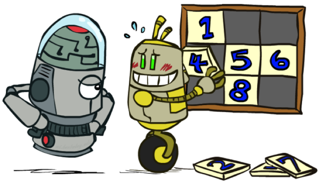
Example: 8 Puzzle

Attempt 1:

- Number of misplaced tiles
- Why is it admissible?
- $h(start) = 8$
- *Relaxed-problem* heuristic



Start State Goal State

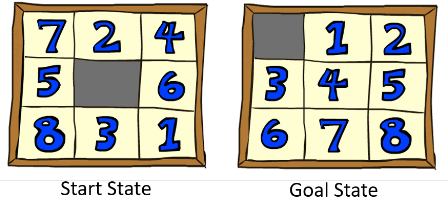


	Average nodes expanded when the optimal path has...		
	...4 steps	...8 steps	...12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

Example: 8 Puzzle

Attempt 2:

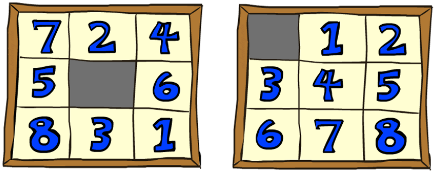
- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?



Example: 8 Puzzle

Attempt 2:

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

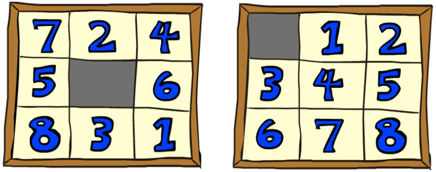


Start State Goal State

Example: 8 Puzzle

Attempt 2:

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

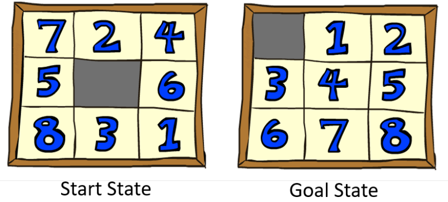


Start State Goal State

Example: 8 Puzzle

Attempt 2:

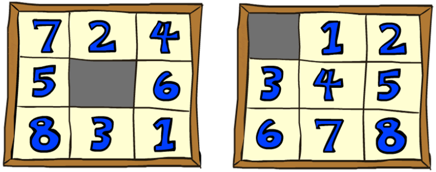
- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total *Manhatten* distance
- Why is it admissible?
- $h(start) = 3 + 1 + 2 + \cdots = 18$



Example: 8 Puzzle

Attempt 2:

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total *Manhatten* distance
- Why is it admissible?
- $h(start) = 3 + 1 + 2 + \cdots = 18$



Start State   Goal State

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

Example: 8 Puzzle

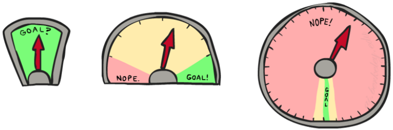
Attempt 3?

- What if we use the actual costs as heuristics?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?

Example: 8 Puzzle

Attempt 3?

- What if we use the actual costs as heuristics?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?

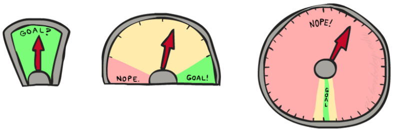


- With A*: a trade-off between quality of estimate and work per node

Example: 8 Puzzle

Attempt 3?

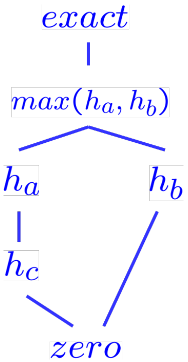
- What if we use the actual costs as heuristics?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?



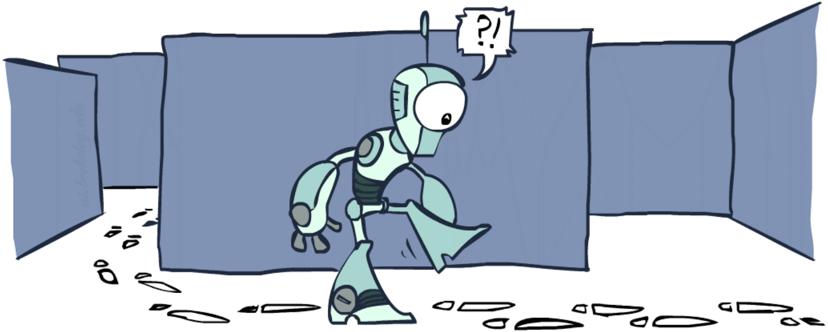
- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself

Semi-Lattice of Heuristics

- Trivial heuristics
  - Bottom of lattice is the zero heuristic
  - Top of lattice is the exact heuristic
- Dominance: $h_a \geq h_c$ if $\forall n : h_a(n) \geq h_c(n)$
- Heuristics can form a semi-lattice:
  - Max of admissible heuristics is admissible $h(n) = \max(h_a(n), h_b(n))$



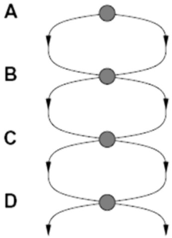
Graph Search



Graph Search

- Tree search requires extra work: Failure to detect repeated states can cause exponentially more work

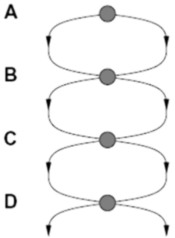
State Space
Graph



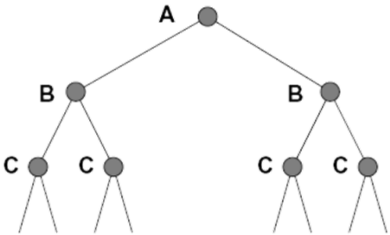
Graph Search

- Tree search requires extra work: Failure to detect repeated states can cause exponentially more work

State Space Graph



Search Tree



- Idea: never expand a state twice

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

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

Graph Search

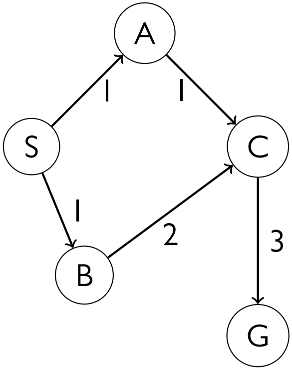
- Idea: never expand a state twice
- How to implement?
  - Tree search + set of expanded states (”closed set”)
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
- Important: store the closed set as a set, not a list
- Can graph search wreck completeness? Why/why not?

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?

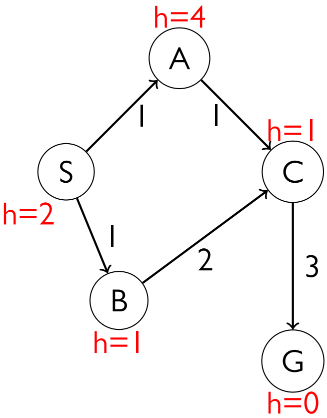
A* Graph Search Gone Wrong?

- State space graph



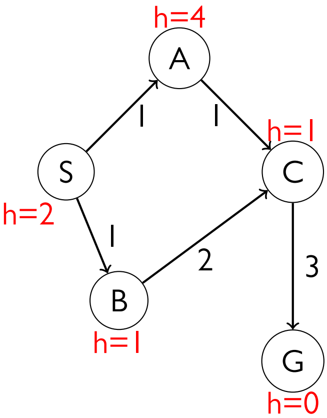
A* Graph Search Gone Wrong?

- State space graph



A* Graph Search Gone Wrong?

- State space graph

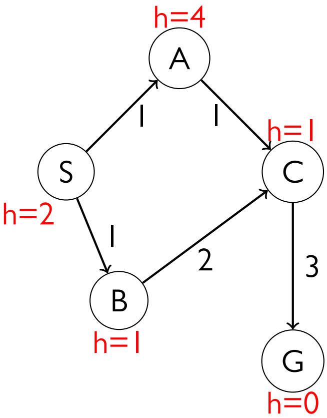


- Search tree

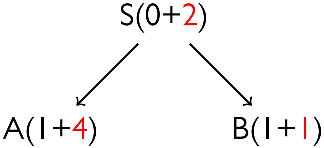
S(0+2)

A* Graph Search Gone Wrong?

- State space graph

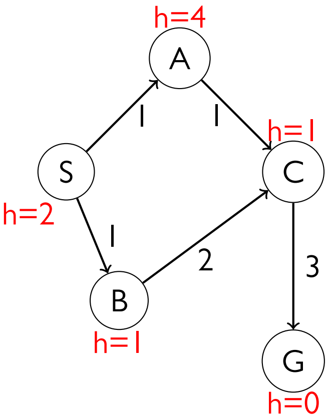


- Search tree

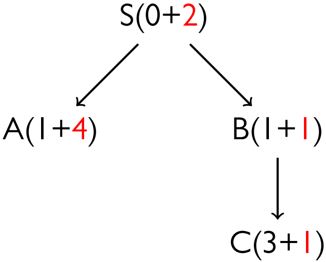


A* Graph Search Gone Wrong?

- State space graph

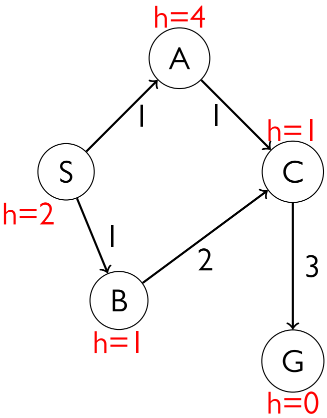


- Search tree

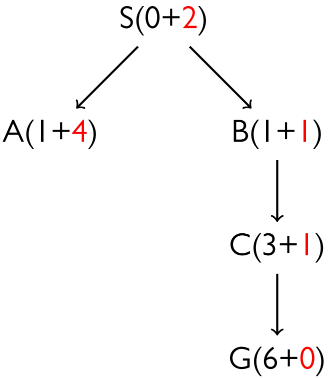


A* Graph Search Gone Wrong?

- State space graph

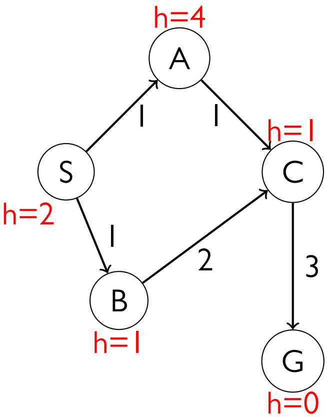


- Search tree

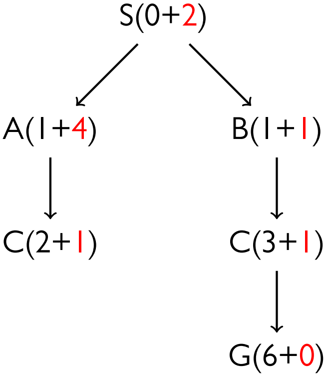


A* Graph Search Gone Wrong?

- State space graph

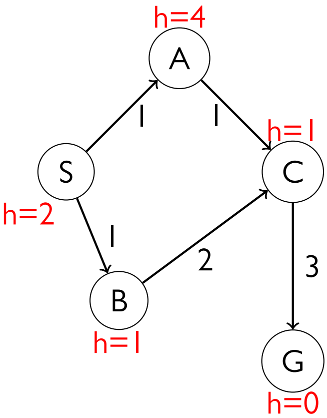


- Search tree

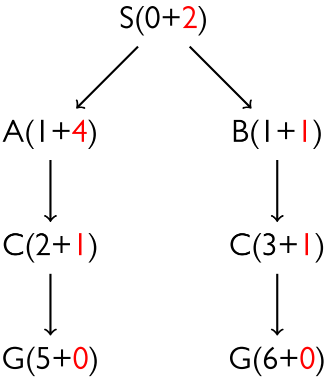


A* Graph Search Gone Wrong?

- State space graph



- Search tree



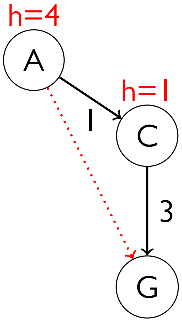
Consistency of Heuristics

Consistency of Heuristics

- Main idea: estimated heuristics cost $\leq$ actual costs

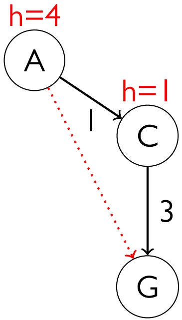
Consistency of Heuristics

- Main idea: estimated heuristics cost $\leq$ actual costs
  - Admissibility: heuristic cost $\leq$ actual cost to goal
    $h(A) \leq$ Actual cost from A to G



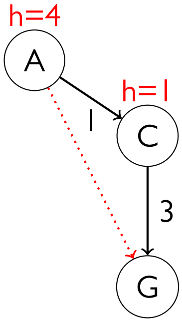
Consistency of Heuristics

- Main idea: estimated heuristics cost $\leq$ actual costs
  - Admissibility: heuristic cost $\leq$ actual cost to goal
    $h(A) \leq$ Actual cost from A to G
  - Consistency: heuristic "arc" cost $\leq$ actual cost for each arc



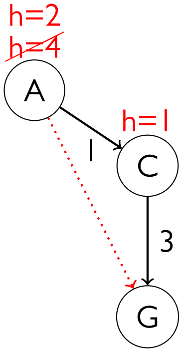
Consistency of Heuristics

- Main idea: estimated heuristics cost $\leq$ actual costs
  - Admissibility: heuristic cost $\leq$ actual cost to goal
    $h(A) \leq$ Actual cost from A to G
  - Consistency: heuristic "arc" cost $\leq$ actual cost for each arc
    h(A) - h(C) $\leq$ cost(A to C)



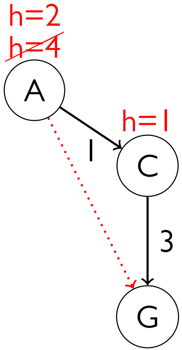
Consistency of Heuristics

- Main idea: estimated heuristics cost $\leq$ actual costs
  - Admissibility: heuristic cost $\leq$ actual cost to goal
    $h(A) \leq$ Actual cost from A to G
  - Consistency: heuristic "arc" cost $\leq$ actual cost for each arc
    h(A) - h(C) $\leq$ cost(A to C)
    h(A) $\leq$ h(C) + cost(A to C)



Consistency of Heuristics

- Main idea: estimated heuristics cost $\leq$ actual costs
  - Admissibility: heuristic cost $\leq$ actual cost to goal
    $h(A) \leq$ Actual cost from A to G
  - Consistency: heuristic "arc" cost $\leq$ actual cost for each arc
    h(A) - h(C) $\leq$ cost(A to C)
    h(A) $\leq$ h(C) + cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases



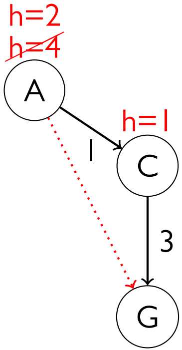
$$h(A) \leq \qquad cost(A\ to\ C) + h(C)$$
$$f(A) = g(A) + h(A) \leq g(A) + cost(A\ to\ C) + h(C) = f(C)$$

Consistency of Heuristics

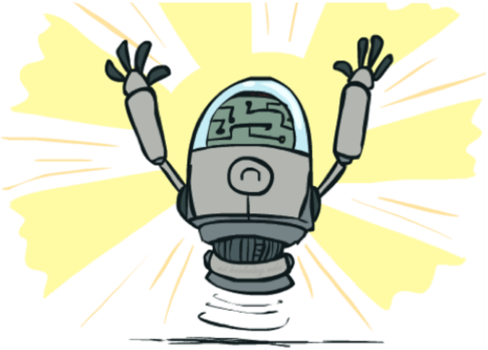
- Main idea: estimated heuristics cost $\leq$ actual costs
  - Admissibility: heuristic cost $\leq$ actual cost to goal
    $h(A) \leq$ Actual cost from A to G
  - Consistency: heuristic "arc" cost $\leq$ actual cost for each arc
    h(A) - h(C) $\leq$ cost(A to C)
    h(A) $\leq$ h(C) + cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases

$$h(A) \leq \qquad cost(A\ to\ C) + h(C)$$
$$f(A) = g(A) + h(A) \leq g(A) + cost(A\ to\ C) + h(C) = f(C)$$

  - A* graph search is optimal

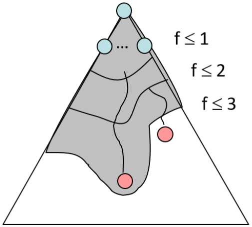


Optimality of A* Graph Search



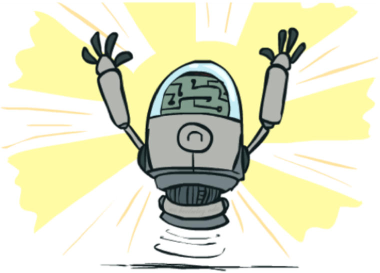
Optimality of A* Graph Search

- 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 suboptimally
  - Result: A* graph search is optimal



Optimality

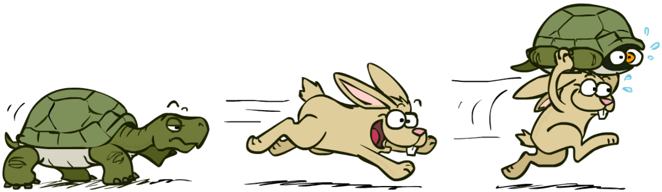
- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissiblity
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



A* Search: Summary

A* Search: 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

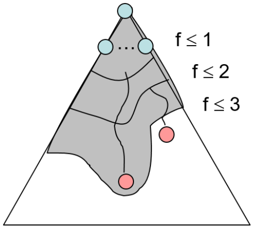
```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    for child-node in EXPAND(STATE[node],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

Graph Search Pseudo-Code

```
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
```

Optimality of A* Graph Search

- Consider what A* does:
  - Expands nodes in increasing total f value (f-contours)
    Reminder: $f(n) = g(n) + h(n)$ = cost to n + heuristic
  - Proof idea: the optimal goal(s) have the lowest f value, so it must get expanded first
- There's a problem with this argument. What are we assuming is true?



Optimality of A* Graph Search

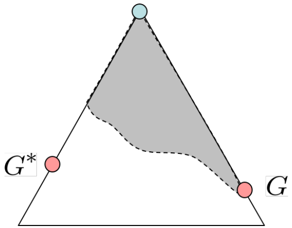
Proof:

- New possible problem: some n on path to G* isn't in queue when we need it, because some worse n' for the same state dequeued and expanded first (disaster!)
- Take the highest such n in tree
- Let p be the ancestor of n that was on the queue when n' was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because n' is suboptimal
- p would have been expanded before n'
- Contradiction!

Optimality of A* Graph Search

Proof:

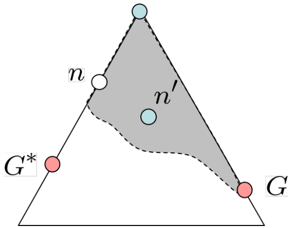
- New possible problem: some n on path to G* isn't in queue when we need it, because some worse n' for the same state dequeued and expanded first (disaster!)
- Take the highest such n in tree
- Let p be the ancestor of n that was on the queue when n' was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because n' is suboptimal
- p would have been expanded before n'
- Contradiction!



Optimality of A* Graph Search

Proof:

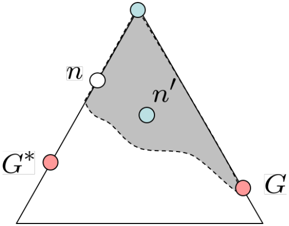
- New possible problem: some n on path to G* isn't in queue when we need it, because some worse n' for the same state dequeued and expanded first (disaster!)
- Take the highest such n in tree
- Let p be the ancestor of n that was on the queue when n' was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because n' is suboptimal
- p would have been expanded before n'
- Contradiction!



Optimality of A* Graph Search

Proof:

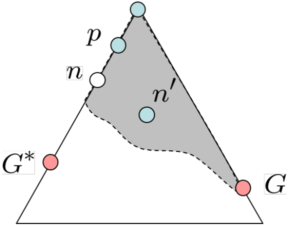
- New possible problem: some n on path to G* isn't in queue when we need it, because some worse n' for the same state dequeued and expanded first (disaster!)
- Take the highest such n in tree
- Let p be the ancestor of n that was on the queue when n' was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because n' is suboptimal
- p would have been expanded before n'
- Contradiction!



Optimality of A* Graph Search

Proof:

- New possible problem: some n on path to G* isn't in queue when we need it, because some worse n' for the same state dequeued and expanded first (disaster!)
- Take the highest such n in tree
- Let p be the ancestor of n that was on the queue when n' was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because n' is suboptimal
- p would have been expanded before n'
- Contradiction!



Suggested Reading

- Russell & Norvig: Chapter 3.5-3.6
- Poole & Mackworth: Chapter: 3.6-3.7