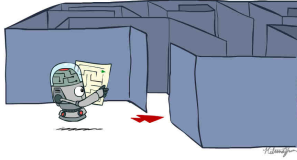


## CS 188x: Artificial Intelligence

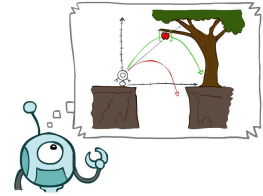
### Search



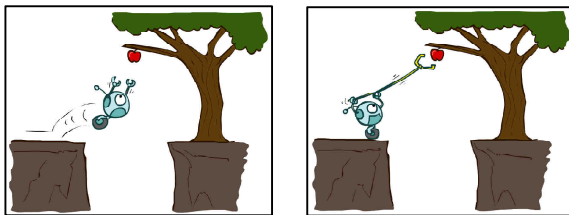
Dan Klein, Pieter Abbeel  
University of California, Berkeley

## Today

- Agents that Plan Ahead
- Search Problems
  - Uninformed Search Methods
    - Depth-First Search
    - Breadth-First Search
    - Uniform-Cost Search

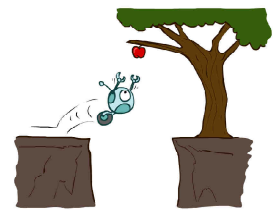


## Agents that Plan



## Reflex Agents

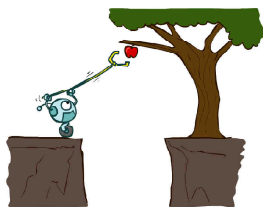
- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world's current state
  - Do not consider the future consequences of their actions
  - Consider how the world IS
- Can a reflex agent be rational?



[demo: reflex optimal / loop]

## Planning Agents

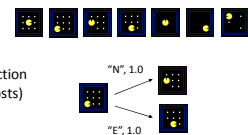
- Planning agents:
  - Ask "what if"
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE
- Optimal vs. complete planning
- Planning vs. replanning



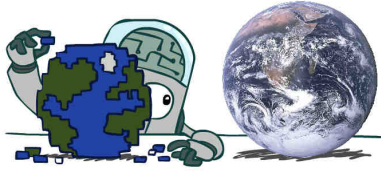
[demo: plan fast / slow]

## Search Problems

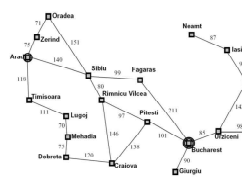
- A search problem consists of:
  - A state space
    - A successor function (with actions, costs)
    - A start state and a goal test
  - A solution is a sequence of actions (a plan) which transforms the start state to a goal state



## Search Problems Are Models



## Example: Traveling in Romania



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

## What's in a State Space?

The **world state** includes every last detail of the environment

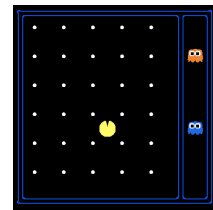


A **search state** keeps only the details needed for planning (abstraction)

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>▪ Problem: Pathing</li> <li>▪ States: (x,y) location</li> <li>▪ Actions: NSEW</li> <li>▪ Successor: update location only</li> <li>▪ Goal test: is (x,y)=END</li> </ul> | <ul style="list-style-type: none"> <li>▪ Problem: Eat-All-Dots</li> <li>▪ States: {(x,y), dot booleans}</li> <li>▪ Actions: NSEW</li> <li>▪ Successor: update location and possibly a dot boolean</li> <li>▪ Goal test: dots all false</li> </ul> |
|---|---|

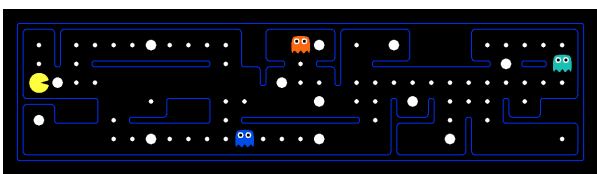
## State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW



- How many
  - World states?  $120 \times (2^{30}) \times (12^2) \times 4$
  - States for pathing? 120
  - States for eat-all-dots?  $120 \times (2^{30})$

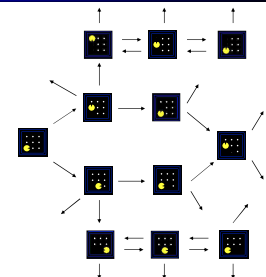
## Quiz: Safe Passage



- Problem: eat all dots while keeping the ghosts perma-scared
- What does the state space have to specify?
  - (agent position, dot booleans, power pellet booleans, remaining scared time)

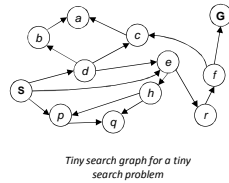
## State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a search graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea

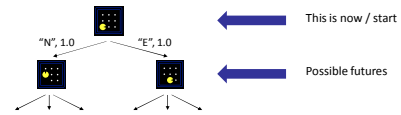


## State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a search graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea

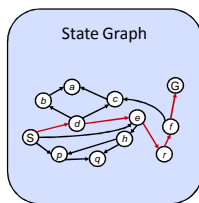


## Search Trees



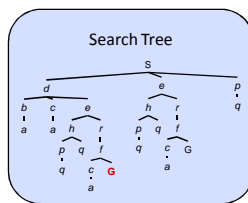
- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

## State Graphs vs. Search Trees



Each NODE in in the search tree is an entire PATH in the problem graph.

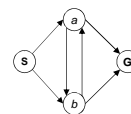
We construct both on demand – and we construct as little as possible.



## Quiz: State Graphs vs. Search Trees

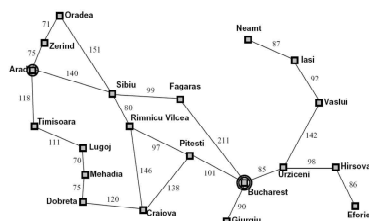
Consider this 4-state graph:

How big is its search tree (from S)?



Important: Lots of repeated structure in the search tree!

## Search Example: Romania



## Searching with a Search Tree



- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a **fringe** of partial plans under consideration
  - Try to expand as few tree nodes as possible

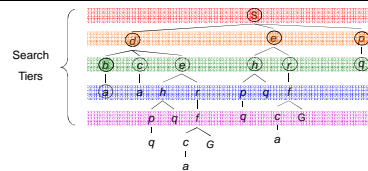
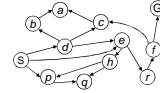


## Breadth-First Search



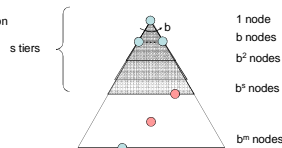
## Breadth-First Search

Strategy: expand a shallowest node first  
Implementation: Fringe is a FIFO queue



## Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - 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?
  - $s$  must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



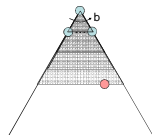
## Quiz: DFS vs BFS

- When will BFS outperform DFS?
- When will DFS outperform BFS?

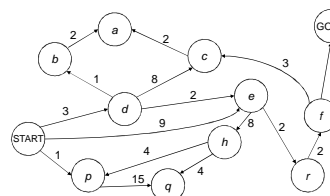
[demo: dfs/bfs]

## Iterative Deepening

- 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. ....
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!



## Cost-Sensitive Search



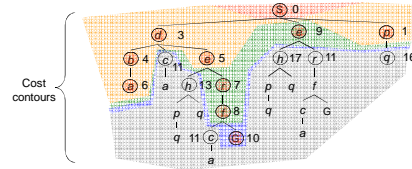
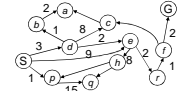
BFS finds the shortest path in terms of number of actions.  
It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.

## Uniform Cost Search



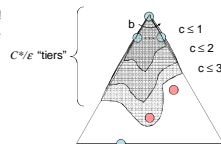
## Uniform Cost Search

Strategy: expand a cheapest node first:  
Fringe is a priority queue (priority: cumulative cost)



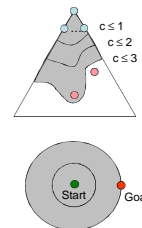
## Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs  $C^*$  and arcs cost at least  $\epsilon$ , then the "effective depth" is roughly  $C^*/\epsilon$
  - Takes time  $O(b^{C^*/\epsilon})$  (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly the last tier, so  $O(b^{C^*/\epsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes! (Proof next lecture via  $A^*$ )



## 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
- We'll fix that soon!



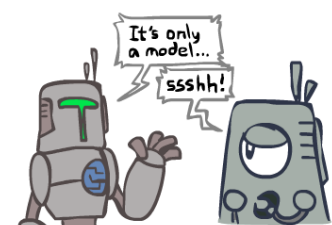
[demo: search demo empty]

## The One Queue: Priority Queues

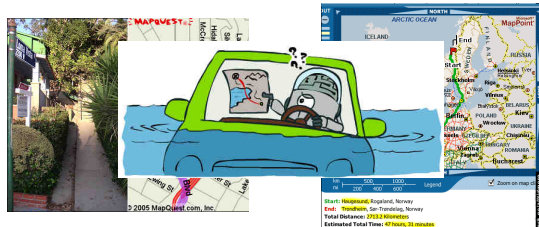
- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the  $\log(n)$  overhead from an actual priority queue with stacks and queues
  - Can even code one implementation that takes a variable queuing object

## Search and Models

- Search operates over models of the world
  - The agent doesn't actually try all the plans out in the real world!
  - Planning is all "in simulation"
  - Your search is only as good as your models...

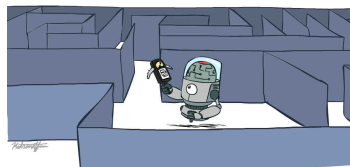


## Search Gone Wrong?



## CS 188x: Artificial Intelligence

### Informed Search



Dan Klein, Pieter Abbeel  
University of California, Berkeley

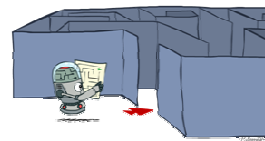
## Today

- Informed Search
  - Heuristics
  - Greedy Search
  - A\* Search
- Graph Search

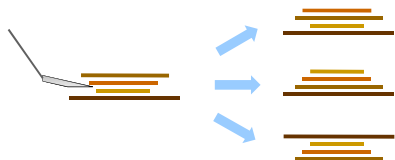


## Recap: Search

- Search problem:
  - States (configurations of the world)
  - Actions and costs
  - Successor function (world dynamics)
  - 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



Cost: Number of pancakes flipped

## Example: Pancake Problem

**BOUNDS FOR SORTING BY PREFIX REVERSAL**

William H. GATES  
Microsoft, Albuquerque, New Mexico

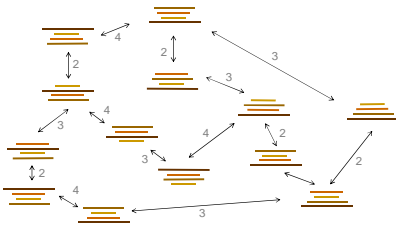
Christos H. PAPADIMITRIOU\*†  
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978  
Revised 28 August 1978

For a permutation  $\sigma$  of the integers from 1 to  $n$ , let  $f(\sigma)$  be the smallest number of prefix reversals that will transform  $\sigma$  to the identity permutation, and let  $f(n)$  be the largest such  $f(\sigma)$  for all  $\sigma$  in the symmetric group  $S_n$ . We show that  $f(n) \leq (5n + 5)/3$ , and that  $f(n) \geq 17n/16$  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  $5n/2 - 1 \leq g(n) \leq 2n + 5$ .

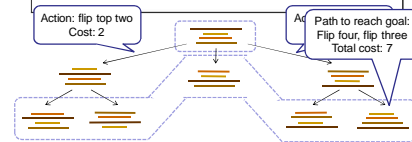
## Example: Pancake Problem

State space graph with costs as weights



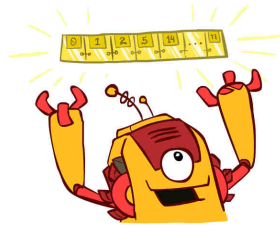
## General Tree Search

```
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
```



## The One Queue

- All these search algorithms are the same except for fringe strategies
- Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
- Practically, for DFS and BFS, you can avoid the  $\log(n)$  overhead from an actual priority queue, by using stacks and queues
- Can even code one implementation that takes a variable queuing object

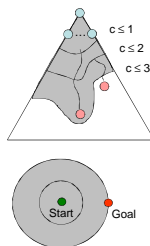


## Uninformed Search



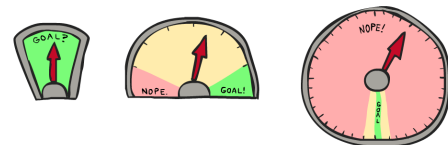
## Uniform Cost Search

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



[demo: contours UCS]

## Informed Search

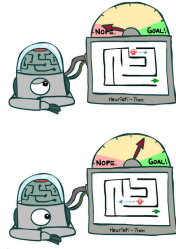
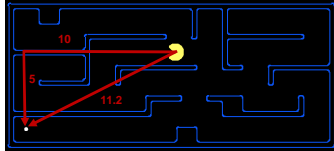




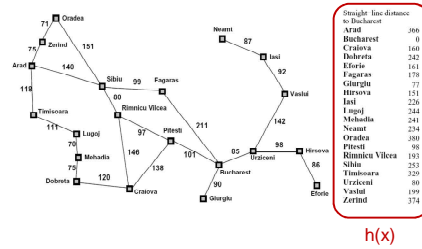
## Search Heuristics

### 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

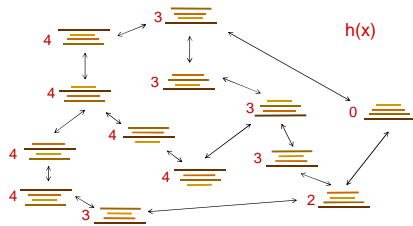


## Example: Heuristic Function



## Example: Heuristic Function

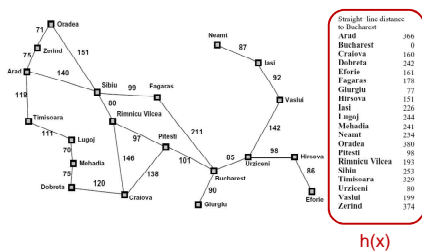
Heuristic: the number of the largest pancake that is still out of place



## Greedy Search

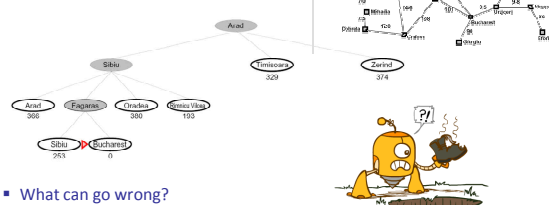


## Example: Heuristic Function



## Greedy Search

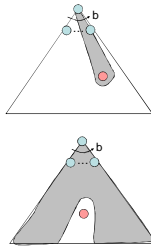
- Expand the node that seems closest...



- What can go wrong?

## 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



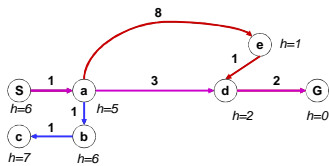
[demo: contours greedy]

## A\* Search



## Combining UCS and Greedy

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

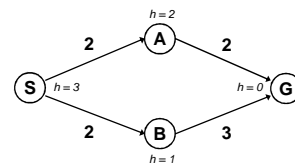


- A\* Search orders by the sum:  $f(n) = g(n) + h(n)$

Example: Teg Grenager

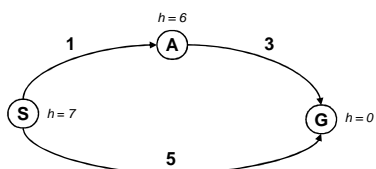
## When should A\* terminate?

- Should we stop when we enqueue a goal?



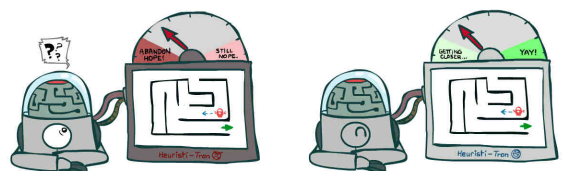
- No: only stop when we dequeue a goal

## Is A\* Optimal?



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

## 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.

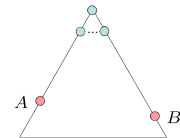
## Optimality of A\* Tree Search

Assume:

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

Claim:

- A will exit the fringe before B

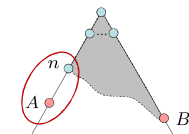


## Optimality of A\* Tree Search: Blocking

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  $f(A)$



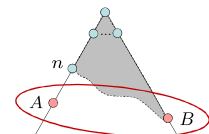
$$\begin{aligned} f(n) &= g(n) + h(n) && \text{Definition of f-cost} \\ f(n) &\leq g(A) && \text{Admissibility of } h \\ g(A) &= f(A) && h = 0 \text{ at a goal} \end{aligned}$$

## 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  $f(B)$



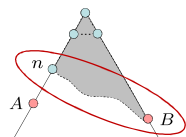
$$\begin{aligned} g(A) &< g(B) && \text{B is suboptimal} \\ f(A) &< f(B) && h = 0 \text{ at a goal} \end{aligned}$$

## 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

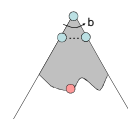


$$f(n) \leq f(A) < f(B)$$

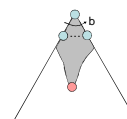
- All ancestors of A expand before B
- A expands before B
- A\* search is optimal

## Properties of A\*

Uniform-Cost

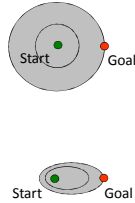


A\*



## UCS vs A\* Contours

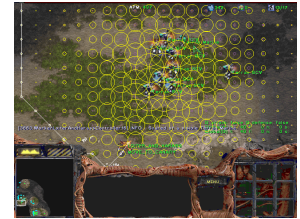
- Uniform-cost expands equally in all "directions"
- A\* expands mainly toward the goal, but does hedge its bets to ensure optimality



[demo: contours UCS / A\*]

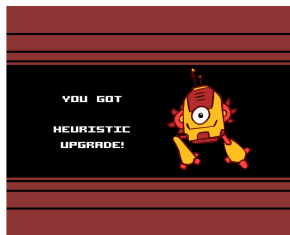
## A\* Applications

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



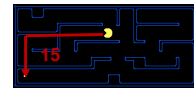
[demo: plan tiny UCS / A\*]

## Creating Heuristics



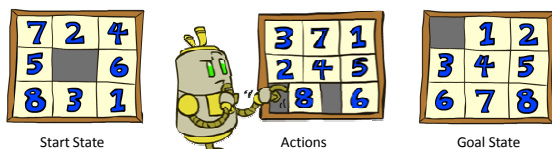
## Creating Admissible Heuristics

- 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

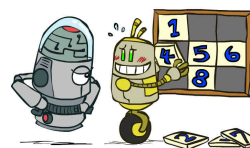
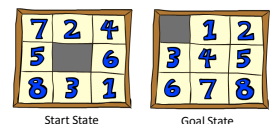
## Example: 8 Puzzle



- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

## 8 Puzzle I

- Heuristic: Number of tiles misplaced
- Why is it admissible?
- $h(\text{start}) = 8$
- This is a *relaxed-problem* heuristic



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

Statistics from Andrew Moore

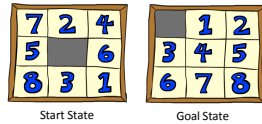
## 8 Puzzle II

- 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?

- $h(\text{start}) = 3 + 1 + 2 + \dots = 18$

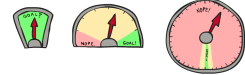


TILES	Average nodes expanded when the optimal path has...		
	...4 steps	...8 steps	...12 steps
MANHATTAN	13	39	227
MANHATTAN	12	25	73

## 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

- 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

## Trivial Heuristics, Dominance

- Dominance:  $h_a \geq h_c$  if

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

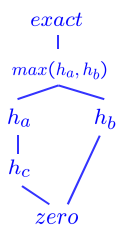
- Heuristics form a semi-lattice:

- Max of admissible heuristics is admissible

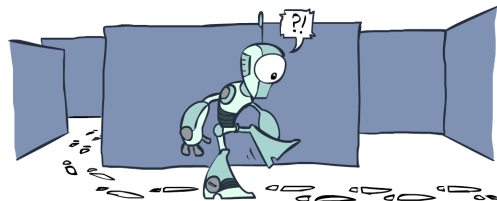
$$h(n) = \max(h_a(n), h_b(n))$$

- Trivial heuristics

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

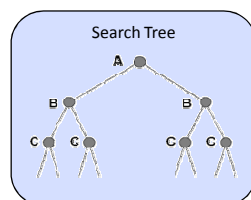
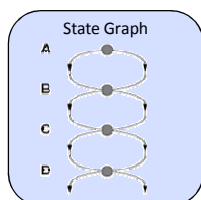


## Graph Search



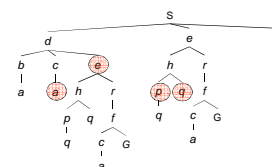
## Tree Search: Extra Work!

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



## Graph Search

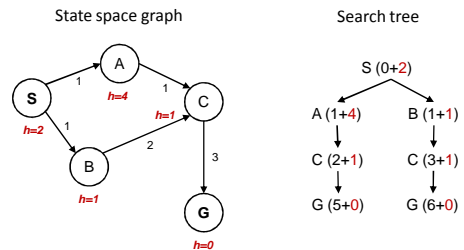
- In BFS, for example, we shouldn't bother expanding the circled nodes (why?)



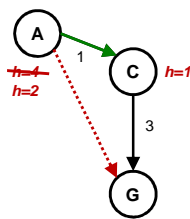
## Graph Search

- 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\* Graph Search Gone Wrong?



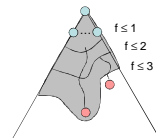
## Consistency of Heuristics



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

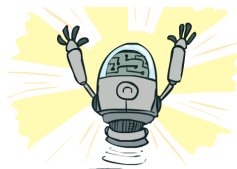
## Optimality of A\* Graph Search

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



## 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

