CS 188: Artificial Intelligence
Fall 2009

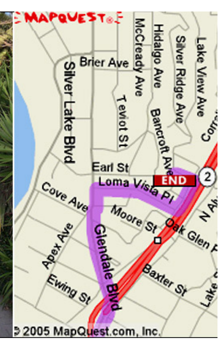
Lecture 3: A* Search

9/3/2009

Pieter Abbeel – UC Berkeley

Many slides from Dan Klein

Search Gone Wrong?



Announcements

- Assignments:
  - Project 0 (Python tutorial): due Friday 1/28 at 4:59pm
  - Project 1 (Search): due Friday 2/4 at 4:59pm
    - Watch for office hour specifics --- GSI project Czar!
  - Still looking for project partners? --- Come to front after lecture.
  - Try pair programming, not divide-and-conquer
  - Account forms available up front during break and after lecture
- Lecture Videos: will be linked from lecture schedule
- Sections start tomorrow
  - Have fun solving exercises! Solutions will be posted online on Friday after last section.
  - After 2 weeks of section we will evaluate potential overcrowdedness issues and find a solution

Today

- Time and space complexity of DFS and BFS
- Iterative deepening --- "best of both worlds"
- Uniform cost search
- Greedy search
- A* search
  - Heuristic design
  - Admissibility, Consistency
- Tree search → Graph search

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)

General Tree Search

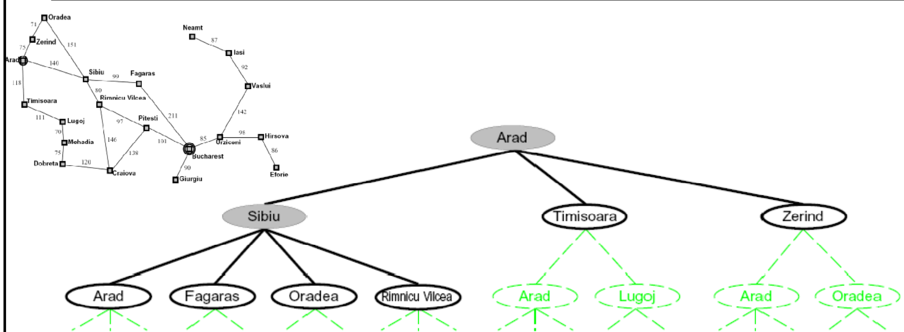
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
     if there are no candidates for expansion then return failure
     choose a leaf node for expansion according to strategy
     if the node contains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

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy

Detailed pseudocode is in the book!

- Main question: which fringe nodes to explore?

Example Search Tree



- Search:
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - Try to expand as few tree nodes as possible

Search Algorithm Properties

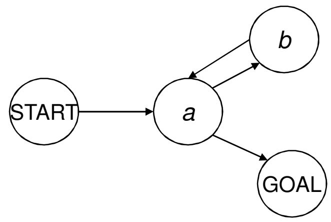
- Complete? Guaranteed to find a solution if one exists?
- Optimal? Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

Variables:

n	Number of states in the problem
b	The average branching factor B (the average number of successors)
$C*$	Cost of least cost solution
s	Depth of the shallowest solution
m	Max depth of the search tree

DFS

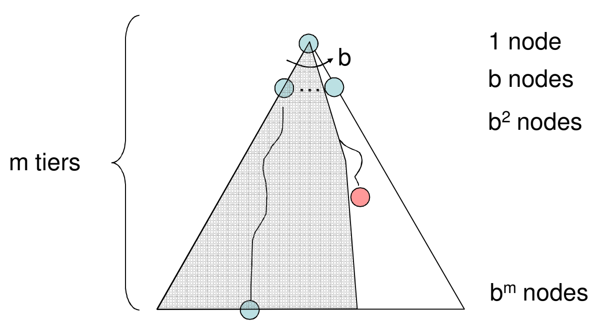
Algorithm		Complete	Optimal	Time	Space
DFS	Depth First Search	N	N	Infinite	Infinite



- Infinite paths make DFS incomplete…
- How can we fix this?

DFS

- With cycle checking, DFS is complete.*



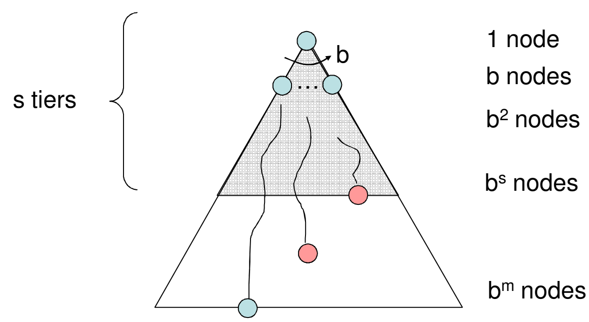
Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^m)$	$O(bm)$

- When is DFS optimal?

* Or graph search – next lecture.

BFS

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^m)$	$O(bm)$
BFS		Y	N*	$O(b^{s+1})$	$O(b^{s+1})$



- When is BFS optimal?

Comparisons

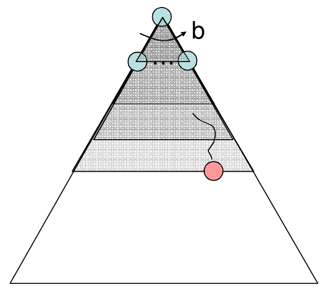
- When will BFS outperform DFS?
- When will DFS outperform BFS?

Iterative Deepening

Iterative deepening uses DFS as a subroutine:

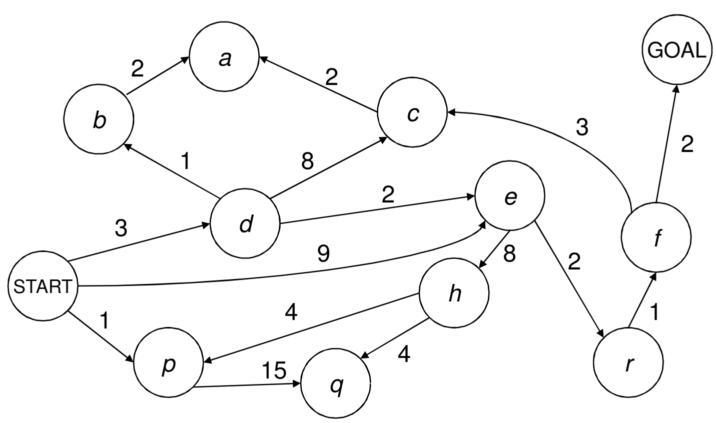
1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

   ….and so on.



Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^m)$	$O(bm)$
BFS		Y	N*	$O(b^{s+1})$	$O(b^{s+1})$
ID		Y	N*	$O(b^{s+1})$	$O(bs)$

Costs on Actions



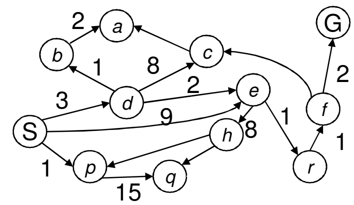
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

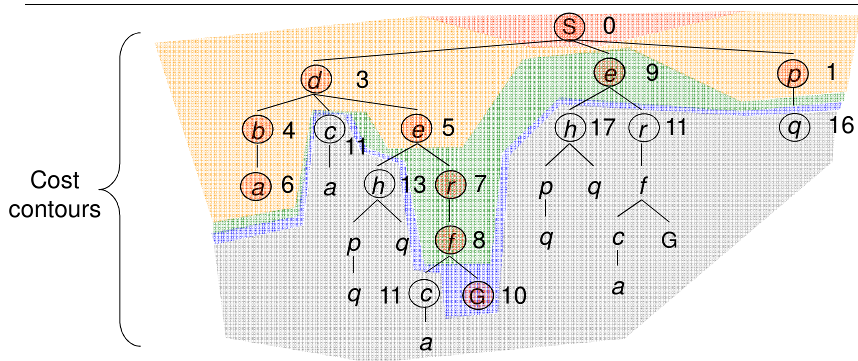
We will quickly cover an algorithm which does find the least-cost path.

Uniform Cost Search

Expand cheapest node first:

Fringe is a priority queue





Priority Queue Refresher

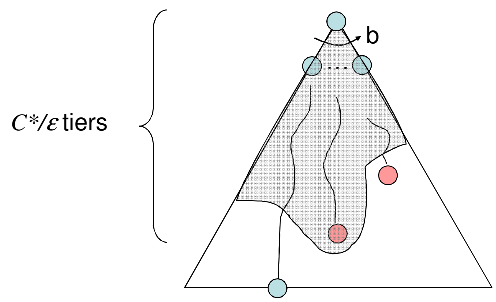
- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

pq.push(key, value)	inserts *(key, value)* into the queue.
pq.pop()	returns the key with the lowest value, and removes it from the queue.

- You can decrease a key's priority by pushing it again
- Unlike a regular queue, insertions aren't constant time, usually O(log *n*)
- We'll need priority queues for cost-sensitive search methods

Uniform Cost Search

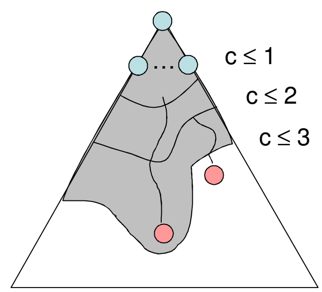
Algorithm		Complete	Optimal	Time (in nodes)	Space
DFS	w/ Path Checking	Y	N	$O(b^m)$	$O(bm)$
BFS		Y	N	$O(b^{s+1})$	$O(b^{s+1})$
UCS		Y*	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$

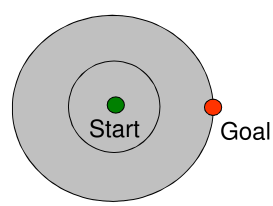


** UCS can fail if actions can get arbitrarily cheap*

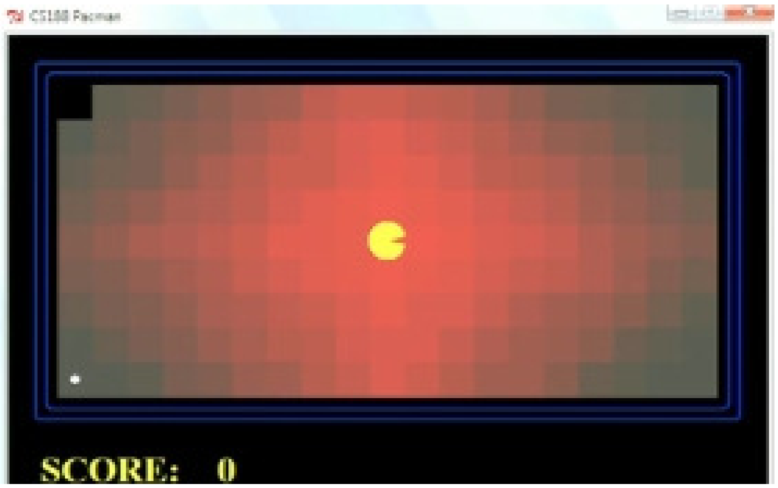
Uniform Cost Issues

- Remember: explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location



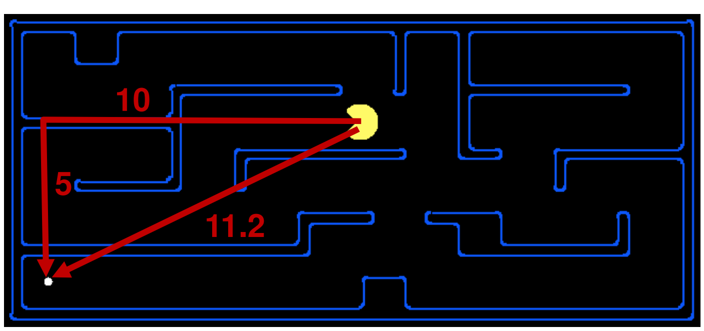


Uniform Cost Search Example

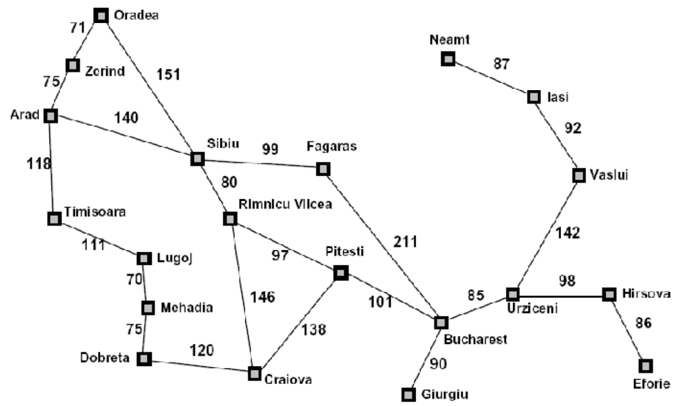


Search Heuristics

- Any *estimate* of how close a state is to a goal
- Designed for a particular search problem
- Examples: Manhattan distance, Euclidean distance



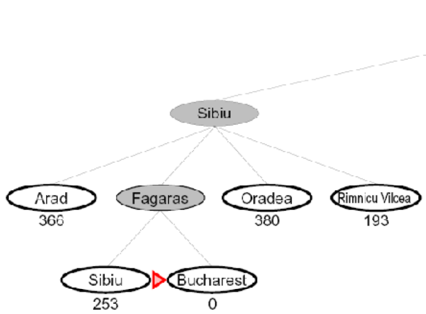
Heuristics

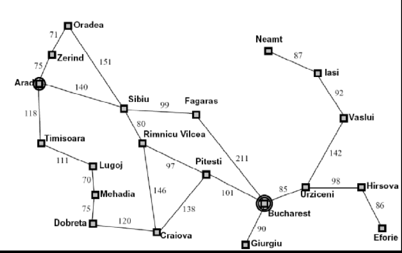


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Best First / Greedy Search

- Expand the node that seems closest…

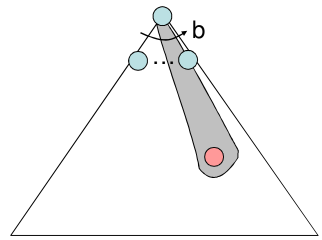


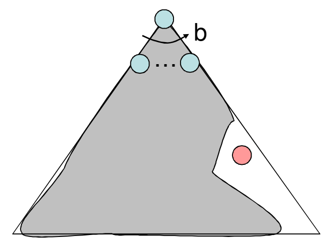


- What can go wrong?

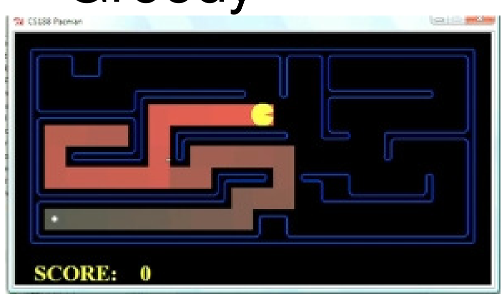
Best First / Greedy Search

- 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
- Like DFS in completeness (finite states w/ cycle checking)

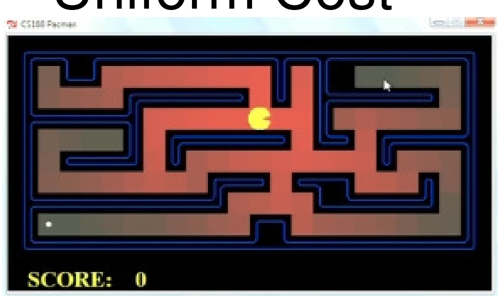




Greedy

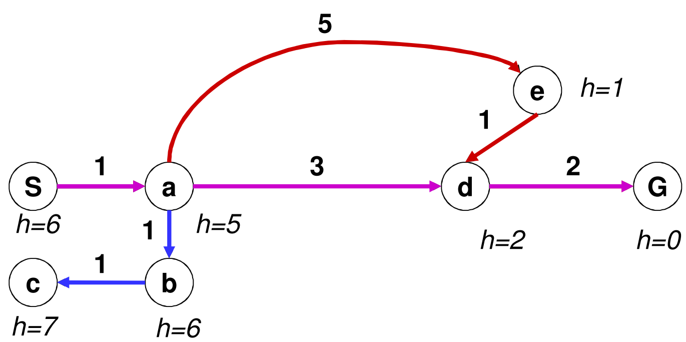


Uniform Cost



Combining UCS and Greedy

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

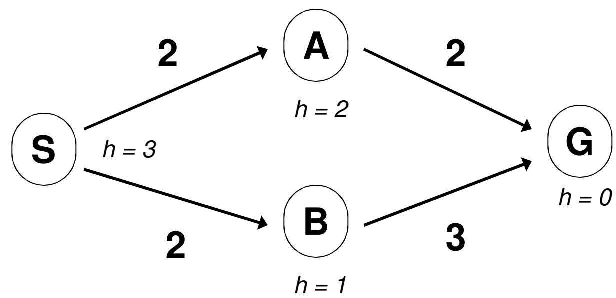


- A* Search orders by the sum: f(n) = g(n) + h(n)

Example: Teg Grenager

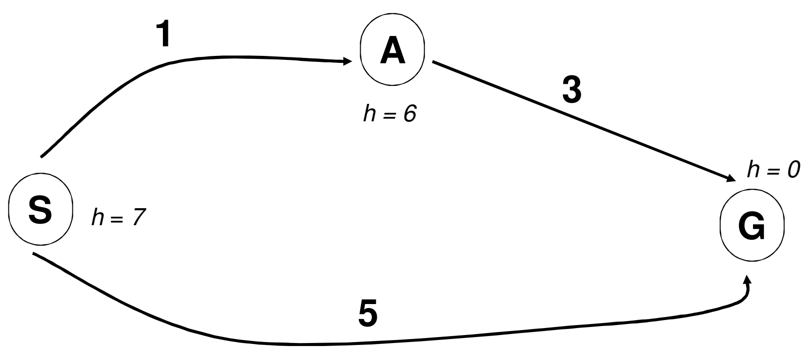
When should A* terminate?

- 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!

Admissible Heuristics

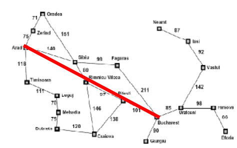
- A heuristic h is *admissible* (optimistic) if:

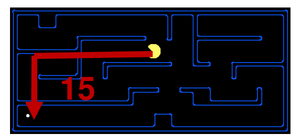
$$h(n) \leq h^*(n)$$

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

- Examples:

366



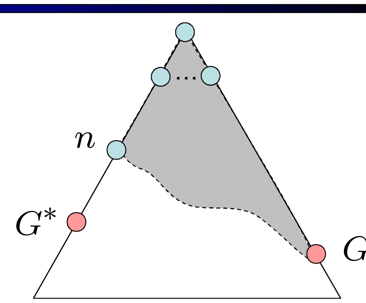


- Coming up with admissible heuristics is most of what's involved in using A* in practice.

Optimality of A*: Blocking

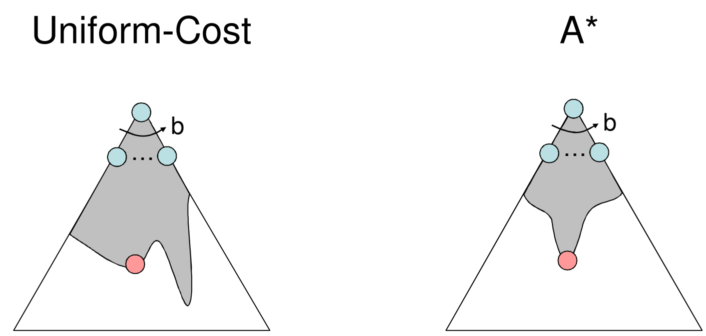
Proof:

- What could go wrong?
- We'd have to have to pop a suboptimal goal G off the fringe before G*
- This can't happen:
  - 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



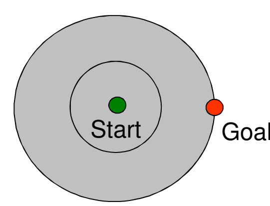
$$f(n) = g(n) + h(n)$$
$$g(n) + h(n) \leq g(G^*)$$
$$g(G^*) < g(G)$$
$$g(G) = f(G)$$
$$f(n) < f(G)$$

Properties of A*

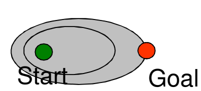


UCS vs A* Contours

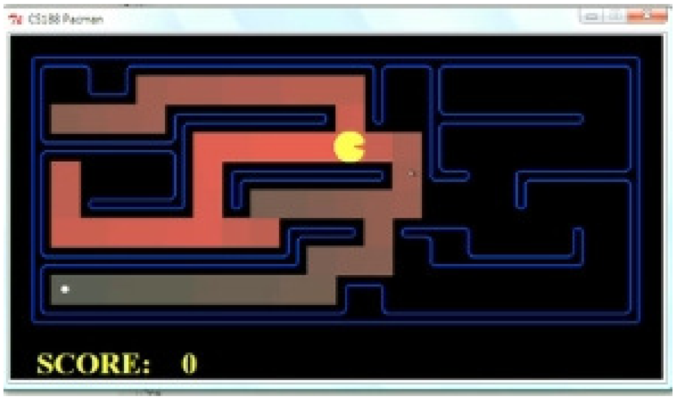
- Uniform-cost expanded in all directions



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



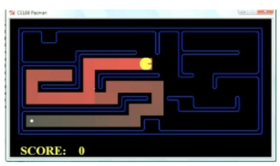
Example: Explored States with A*



Heuristic: manhattan distance ignoring walls

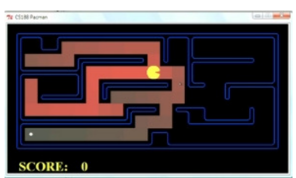
Comparison

Greedy



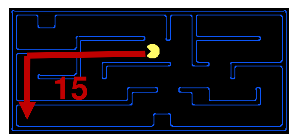
Uniform Cost

A star



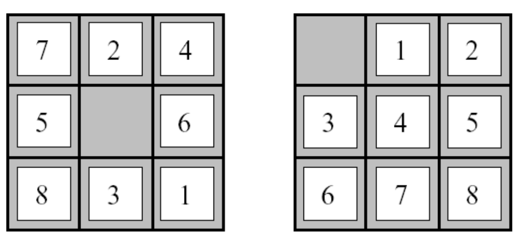
Creating Admissible Heuristics

- 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,* with new actions ("some cheating") 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: Number of tiles misplaced
- Why is it admissible?
- h(start) = 8
- This is a relaxed-problem heuristic

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

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

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 admissible?
- h(start) =

  3 + 1 + 2 + …

  = 18

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

	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!

Trivial Heuristics, Dominance

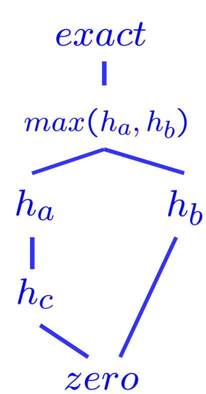
- Dominance: $h_a \geq h_c$ if

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

- 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

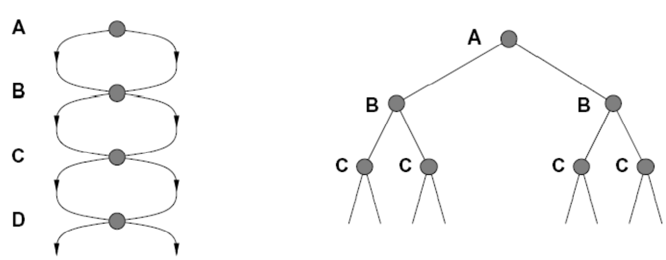


Other A* Applications

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

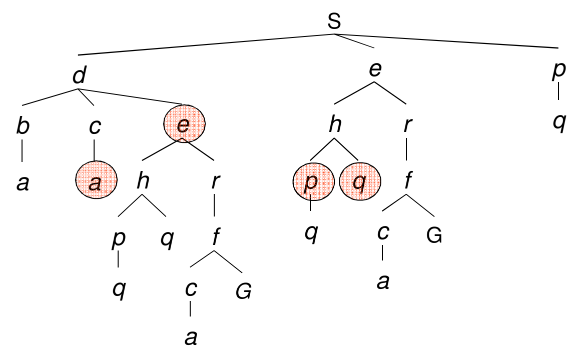
Tree Search: Extra Work!

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



Graph Search

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



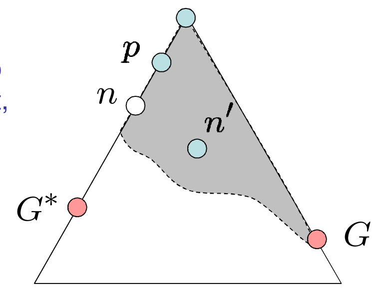
Graph Search

- Idea: never expand a state twice
- How to implement:
  - Tree search + list of expanded states (closed list)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state is new
- Python trick: store the closed list as a set, not a list
- Can graph search wreck completeness? Why/why not?
- How about optimality?

Optimality of A* Graph Search

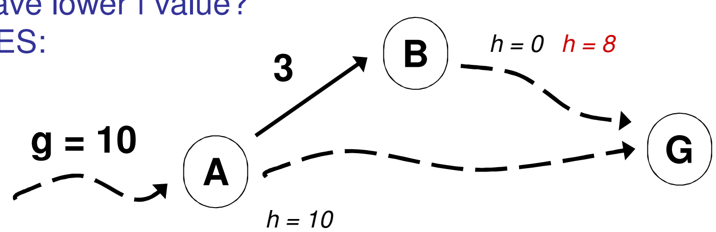
Proof:

- New possible problem: nodes on path to G* that would have been in queue aren't, because some worse *n'* for the same state as some *n* was dequeued and expanded first (disaster!)
- Take the highest such *n* in tree
- Let *p* be the ancestor which was on the queue when *n*' was expanded
- Assume *f(p) < f(n)*
- *f(n) < f(n')* because *n'* is suboptimal
- *p* would have been expanded before *n*'
- So *n* would have been expanded before *n*', too
- Contradiction!



Consistency

- Wait, how do we know parents have better f-vales than their successors?
- Couldn't we pop some node *n*, and find its child *n'* to have lower f value?
- YES:



- What can we require to prevent these inversions?
- Consistency: $c(n, a, n') \geq h(n) - h(n')$
- Real cost must always exceed reduction in heuristic

Optimality

- Tree search:
  - A* optimal if heuristic is admissible (and non-negative)
  - 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, natural admissible heuristics tend to be consistent

Summary: A*

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