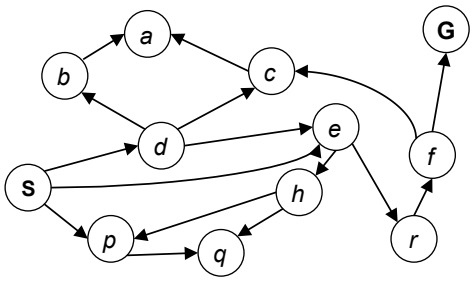
Queue-Based Search

Pieter Abbeel – UC Berkeley

Many slides from Dan Klein

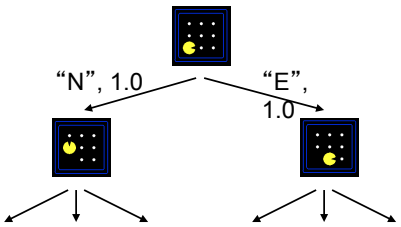
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - For every search problem, there's a corresponding state space graph
  - The successor function is represented by arcs
- We can rarely build this graph in memory (so we don't)



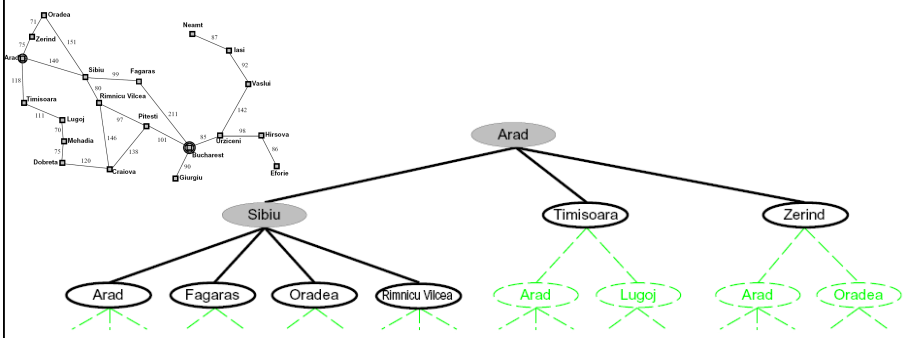
Ridiculously tiny search graph for a tiny search problem

Search Trees



- A search tree:
  - This is a “what if” tree of plans and outcomes
  - Start state at the root node
  - Children correspond to successors
  - Nodes contain states, correspond to PLANS to those states
  - For most problems, we can never actually build the whole tree

Another Search Tree



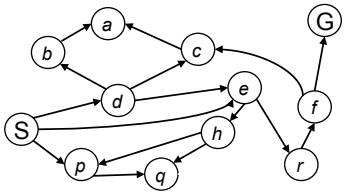
- Search:
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - Try to expand as few tree nodes as possible

General Tree Search

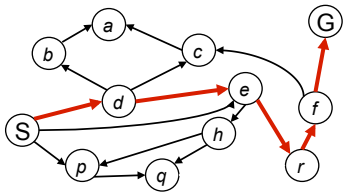
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the 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
- Main question: which fringe nodes to explore?

Example: Tree Search

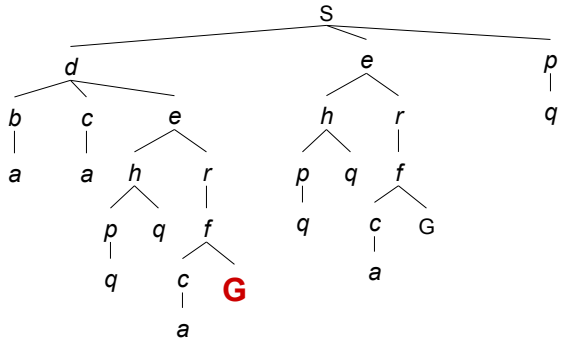


State Graphs vs. Search Trees



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

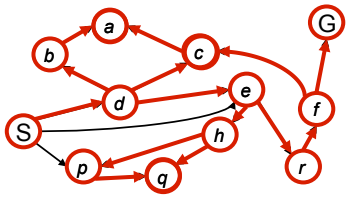
We construct both on demand – and we construct as little as possible.

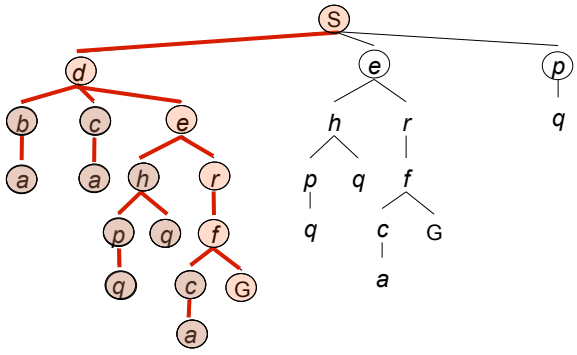


Depth First Search

Strategy: expand deepest node first

Implementation: Fringe is a LIFO stack

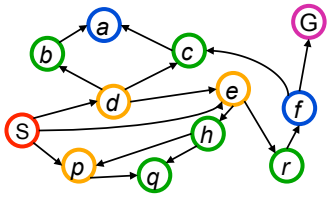


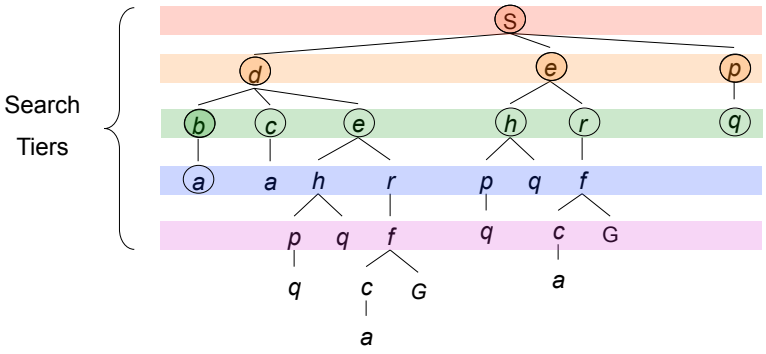


Breadth First Search

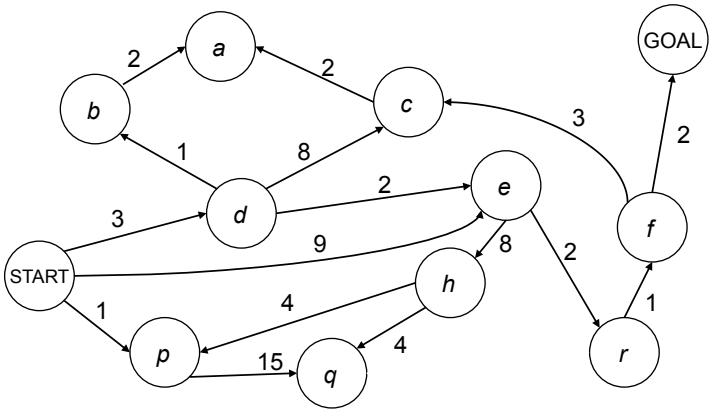
Strategy: expand shallowest node first

Implementation: Fringe is a FIFO queue





Costs on Actions



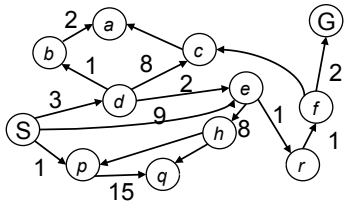
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

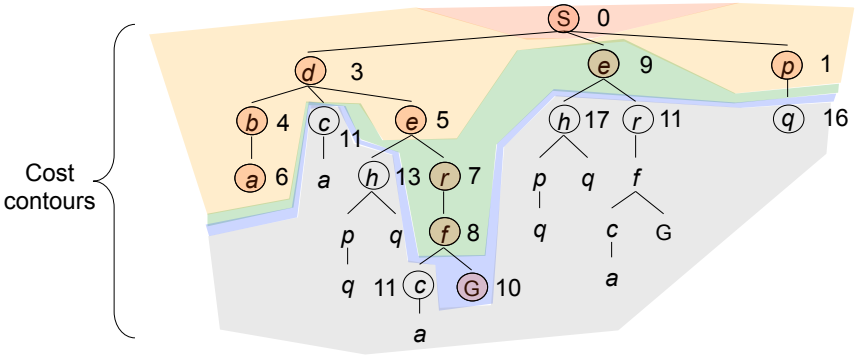
We will quickly cover an algorithm which does find the least-cost path.

Uniform Cost Search

Expand cheapest node first:

Fringe is a priority queue





Priority Queue Refresher

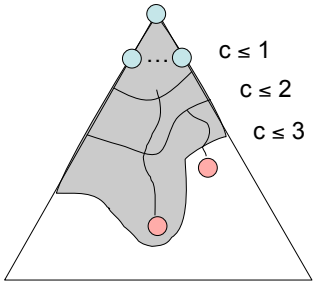
- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

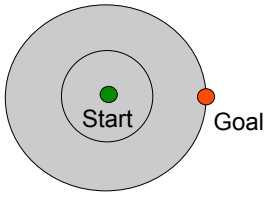
pq.push(key, value)	inserts *(key, value)* into the queue.
pq.pop()	returns the key with the lowest value, and removes it from the queue.

- 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 Issues

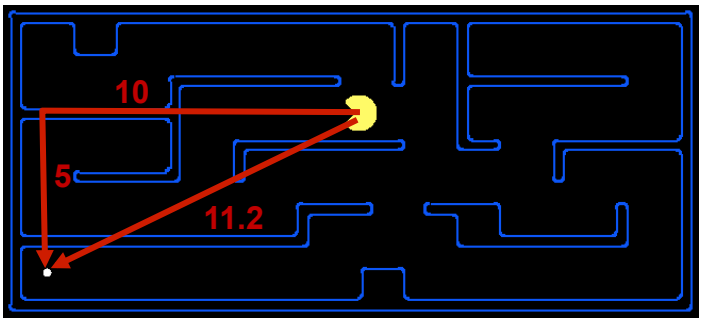
- Remember: explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location



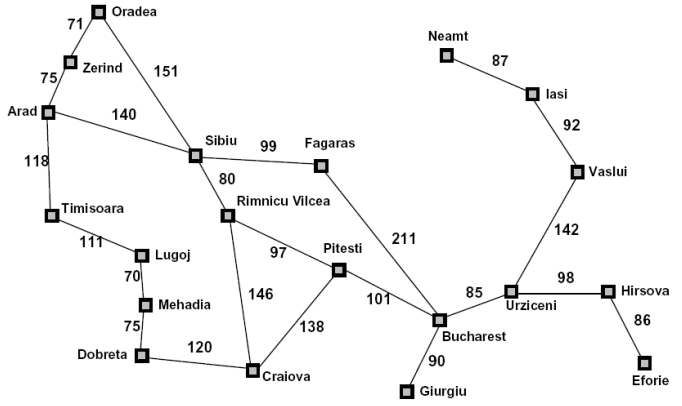


Search Heuristics

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



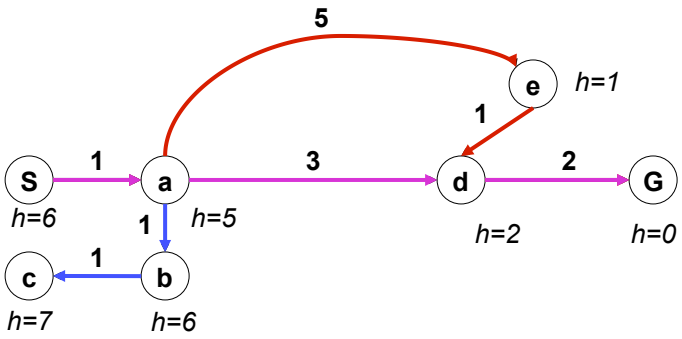
Heuristics



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

Combining UCS and a Heuristic

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

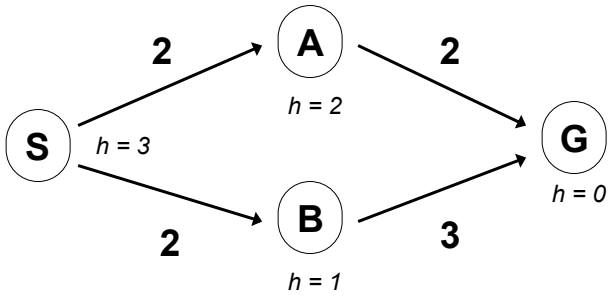


- A* Search orders by the sum: f(n) = g(n) + h(n)

Example: Teg Grenager

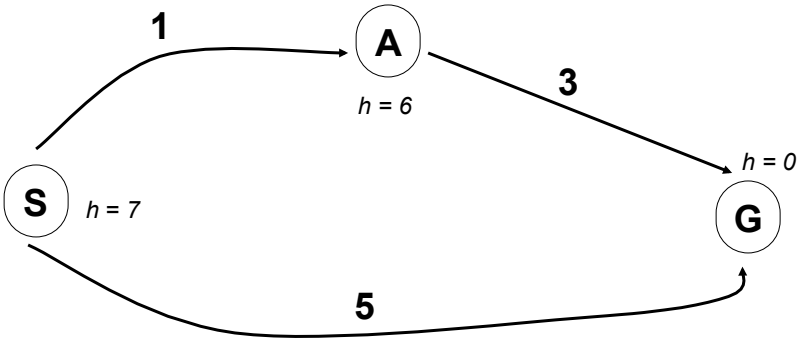
When should A* terminate?

- 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

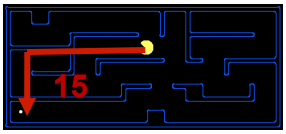
- A heuristic h is *admissible* (optimistic) if:

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

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

- Examples:

366

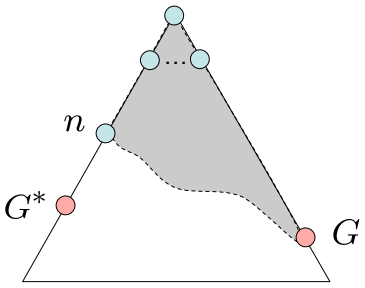


- Coming up with admissible heuristics is most of what's involved in using A* in practice.

Optimality of A*: Blocking

Proof:

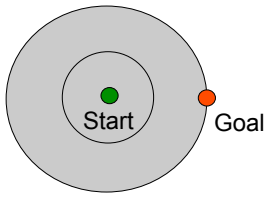
- What could go wrong?
- We'd have to have to pop a suboptimal goal G off the fringe before G*
- This can't happen:
  - Imagine a suboptimal goal G is on the queue
  - Some node *n* which is a subpath of G* must also be on the fringe (why?)
  - *n* will be popped before G



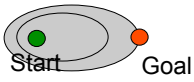
$$f(n) = g(n) + h(n)$$
$$g(n) + h(n) \leq g(G^*)$$
$$g(G^*) < g(G)$$
$$g(G) = f(G)$$
$$f(n) < f(G)$$

UCS vs A* Contours

- Uniform-cost expanded in all directions

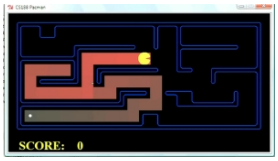


- A* expands mainly toward the goal, but does hedge its bets to ensure optimality

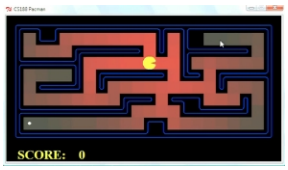


Comparison

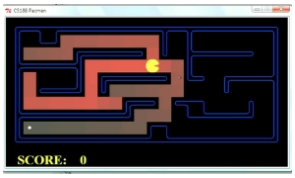
Greedy



Uniform Cost

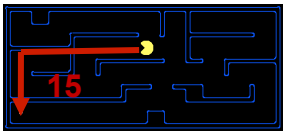


A star



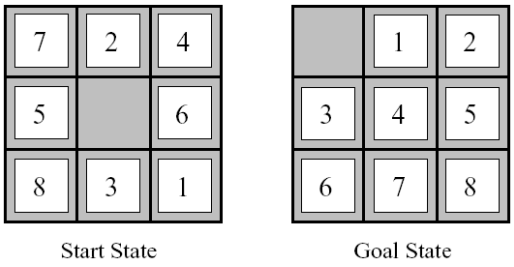
Creating Admissible Heuristics

- 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

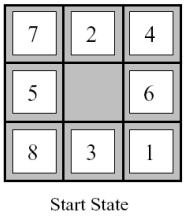


Start State Goal State

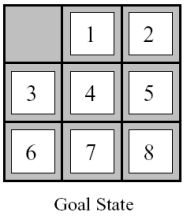
- What are the states?
- How many states?
- What are the actions?
- What states can I reach from the start state?
- What should the costs be?

8 Puzzle I

- Heuristic: Number of tiles misplaced
- Why is it admissible?
- h(start) = 8
- This is a relaxed-problem heuristic



Start State



Goal State

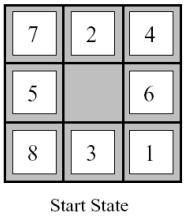
	Average nodes expanded when optimal path has length…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

8 Puzzle II

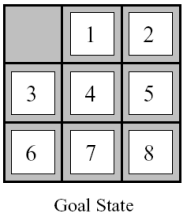
- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total *Manhattan* distance
- Why admissible?
- h(start) =

  3 + 1 + 2 + …

  = 18



Start State



Goal State

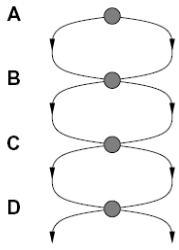
	Average nodes expanded when optimal path has length…		
	…4 steps	…8 steps	…12 steps
TILES	13	39	227
MANHATTAN	12	25	73

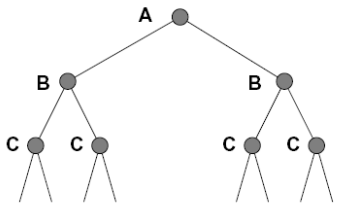
8 Puzzle III

- 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!

Tree Search: Extra Work!

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





287 Graph Search (~=188!)

- Very simple fix: check if state worth expanding again:
  - Keep around "expanded list", which stores pairs

  (expanded node, g-cost the expanded node was reached with)
  - When about to expand a node, only expand it if

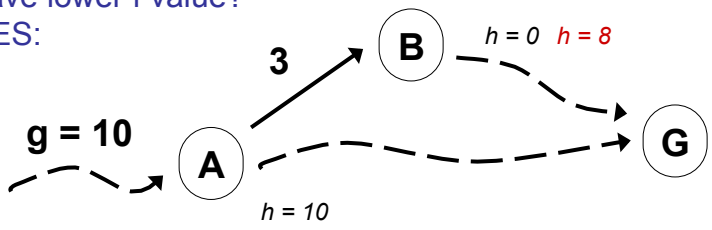
  either (i) it has not been expanded before (in 188 lingo: are not in the closed list)

  or (ii) it has been expanded before, but the new way of reaching this node is cheaper than the cost it was reached with when expanded before

- How about "consistency" of the heuristic function?
  - = condition on heuristic function
  - If heuristic is consistent, then the "new way of reaching the node" is guaranteed to be more expensive, hence a node never gets expanded twice;
  - In other words: if heuristic is consistent then when a node is expanded, the shortest path from the start state to that node has been found

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)
- 287 Graph search:
  - A* optimal if heuristic is admissible
    - A* expands every node only once if heuristic also consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, natural admissible heuristics tend to be consistent

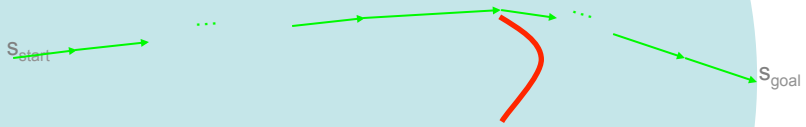
Weighted A* $f = g+\varepsilon h$

- **Weighted A*:** expands states in the order of $f = g+\varepsilon h$ values,

  $\varepsilon > 1$ = bias towards states that are closer to goal

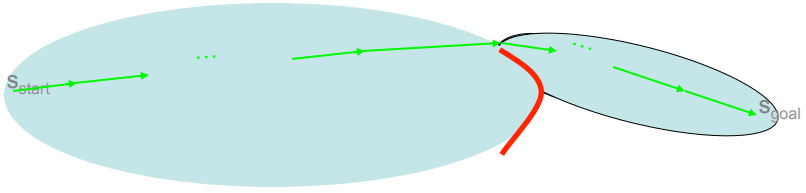
37

Weighted A* $f = g+\varepsilon h$: $\varepsilon = 0$ --- Uniform Cost Search



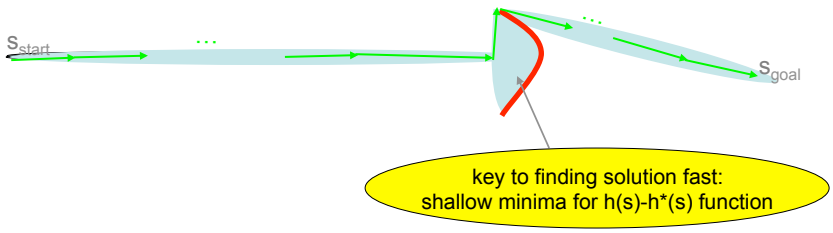
38

Weighted A* $f = g+\varepsilon h$: $\varepsilon = 1$ --- A*



39

Weighted A* $f = g+\varepsilon h : \varepsilon > 1$



40

Weighted A* $f = g+\varepsilon h : \varepsilon > 1$

- Trades off optimality for speed
- ε-suboptimal:
  - cost(solution) ≤ ε·cost(optimal solution)
  - Test your understanding by trying to prove this!
- In many domains, it has been shown to be orders of magnitude faster than A*
- Research becomes to develop a heuristic function that has shallow local minima

41

Anytime A*

- Weighted A*
  - Trades off optimality for speed
  - ε-suboptimal
- Anytime A*
  - For $\epsilon \in \{ \epsilon_1, \epsilon_2, \ldots, 1\}$
    - Run weighted A* with current ϵ
- Anytime Repairing A* [Likhachev, Gordon, Thrun 2004]
  - efficient version of above that reuses state values within each iteration

Anytime Repairing A* (ARA*)

- Starting point:

Anytime A*

- For $\epsilon \in \{ \epsilon_1, \epsilon_2, \ldots, 1\}$
  - Run weighted A* with current ϵ

- When about to expand node, if already expanded before, don't expand again to save time, instead put on INCON list (for consistent heuristic this is fine and will still give us epsilon optimality guarantee)
- When epsilon decreased
  - Initialize priority queue with current priority queue union INCON
  - Update priorities
- Why is this "union INCON" needed? epsilon*h need not be consistent, hence need to keep track of potential optimality violations and re-consider later

Anytime Nonparametric A* (ANA*) [van den Berg, Shah, Huang, Goldberg, 2011]

- Tricky issue with ARA*:
  - How much to decrease epsilon in each step?
  - In practice: some tweaking
- ANA*: provides a theoretically justified and empirically shown to be superior scheme that can (a bit crudely) be thought of as always picking the right next epsilon

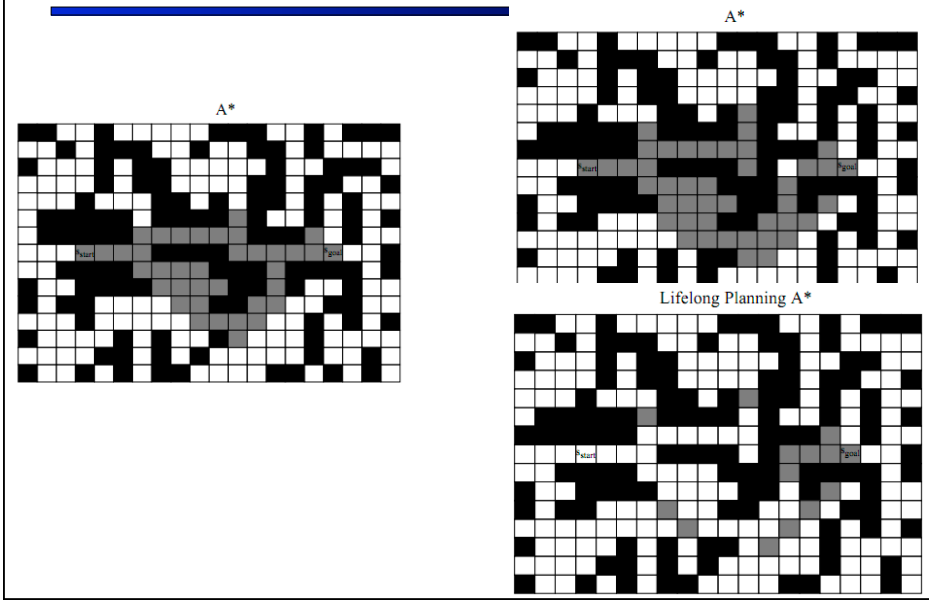
Lifelong Planning A* (LPA*)

[Koenig, Likhachev, Furcy, 2004]

- LPA* is able to handle changes in edge costs efficiently
  - Example application: find out a road has been blocked

Lifelong Planning A* (LPA*)

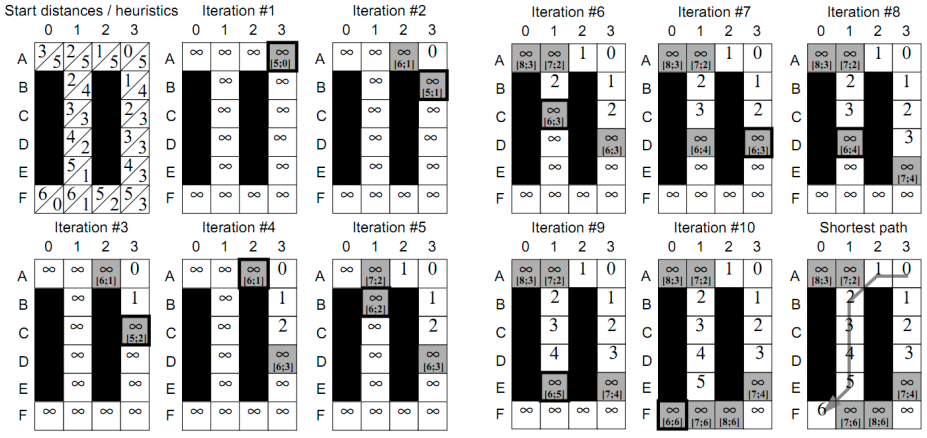
[Koenig, Likhachev, Furcy, 2004]



Lifelong Planning A* (LPA*)

[Koenig, Likhachev, Furcy, 2004]

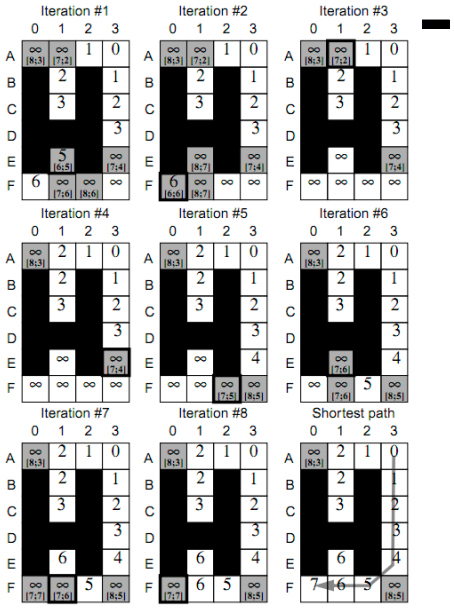
- First search (when no search has been done before) = A* search



Lifelong Planning A* (LPA*)

[Koenig, Likhachev, Furcy, 2004]

- Once (D,1) is blocked → re-search:



Lifelong Planning A* (LPA*)

[Koenig, Likhachev, Furcy, 2004]

procedure CalculateKey(s)
{01} return $[\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))]$;
procedure Initialize()
{02} $U = \emptyset$;
{03} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{04} $rhs(s_{start}) = 0$;
{05} U.Insert$(s_{start}, [h(s_{start}); 0])$;
procedure UpdateVertex(u)
{06} if $(u \neq s_{start})$ $rhs(u) = \min_{s' \in pred(u)}(g(s') + c(s', u))$;
{07} if $(u \in U)$ U.Remove(u);
{08} if $(g(u) \neq rhs(u))$ U.Insert$(u,$ CalculateKey$(u))$;
procedure ComputeShortestPath()
{09} while (U.TopKey() $<$ CalculateKey(s_{goal}) OR $rhs(s_{goal}) \neq g(s_{goal})$)
{10} $u =$ U.Pop();
{11} if $(g(u) > rhs(u))$
{12} $g(u) = rhs(u)$;
{13} for all $s \in succ(u)$ UpdateVertex(s);
{14} else
{15} $g(u) = \infty$;
{16} for all $s \in succ(u) \cup \{u\}$ UpdateVertex(s);
procedure Main()
{17} Initialize();
{18} forever
{19} ComputeShortestPath();
{20} Wait for changes in edge costs;
{21} for all directed edges (u, v) with changed edge costs
{22} Update the edge cost $c(u, v)$;
{23} UpdateVertex(v);

Fig. 6. Lifelong Planning A*

A* From Goal to Start

- A* with consistent heuristic finds shortest path from start state to all expanded states
- → If we flip roles of goal and start, it gives us shortest paths from all expanded states to the goal
  - →We obtain a closed-loop policy!
  - →Can account for dynamics noise
- Lifelong Planning A* + Flip-Start-Goal + some other optimizations → D* Lite [Koenig and Likhachev]
  - → Can account for observations, which can be encoded into changes in edge costs! (e.g., blocked path, etc.)