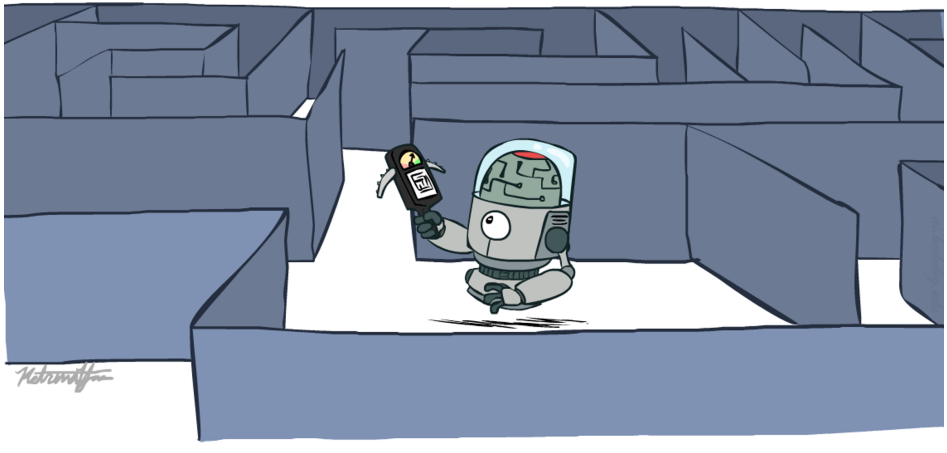
CS 188: Artificial Intelligence

Search Continued

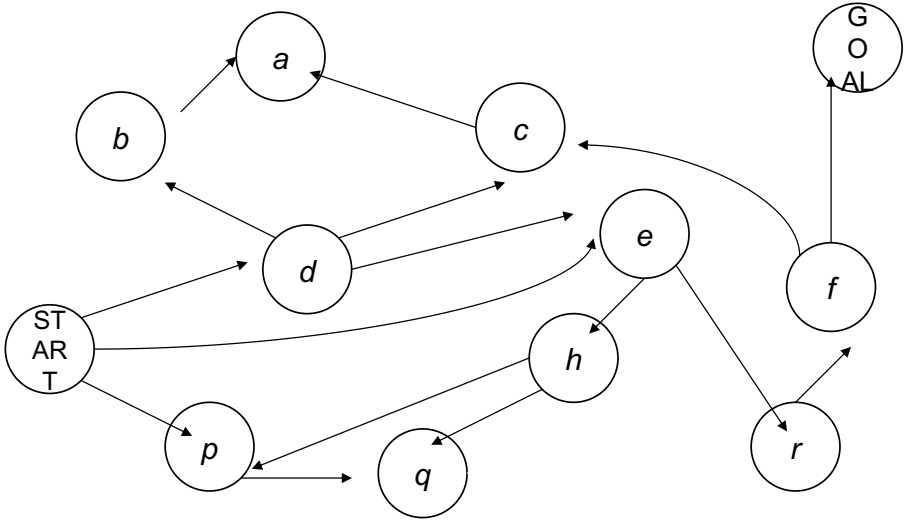


Instructors: Anca Dragan

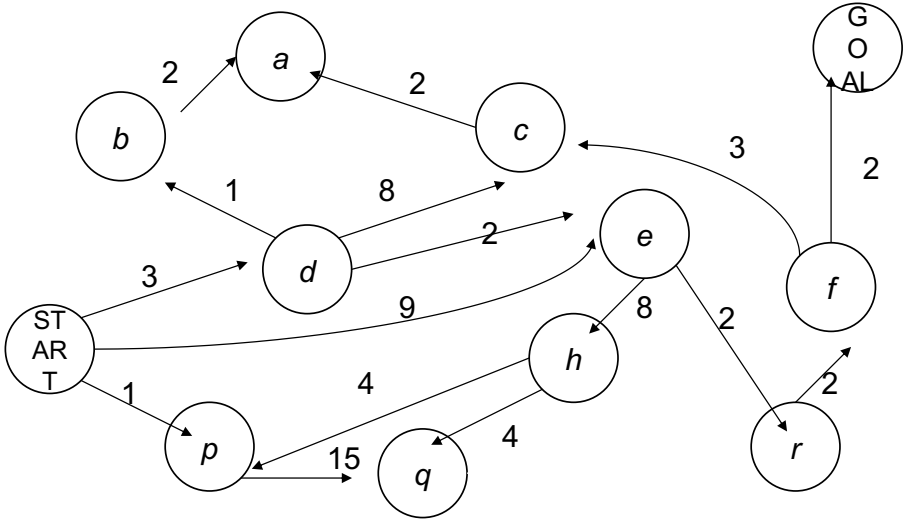
University of California, Berkeley

[These slides adapted from Dan Klein and Pieter Abbeel; ai.berkeley.edu]

Cost-Sensitive Search



Cost-Sensitive Search



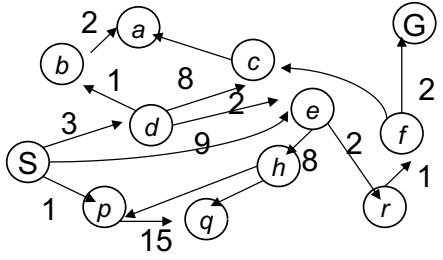
BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.

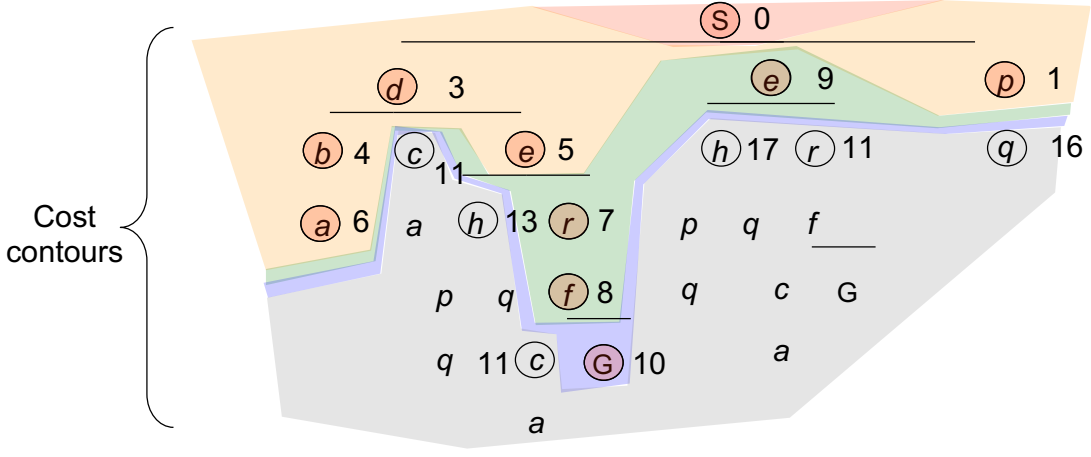
How?

Uniform Cost Search

Strategy: expand a cheapest node first:

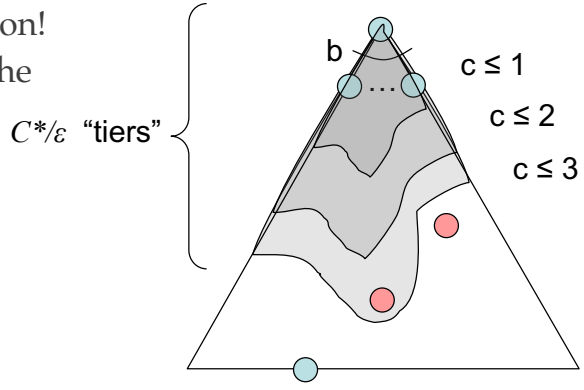
Fringe is a priority queue (priority: cumulative cost)





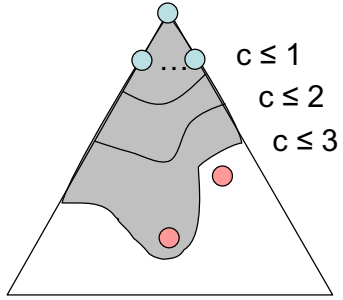
Uniform Cost Search (UCS) Properties

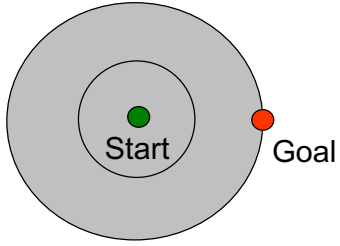
- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε , then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes! (if no solution, still need depth != ∞)
- Is it optimal?
  - Yes! (Proof 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!

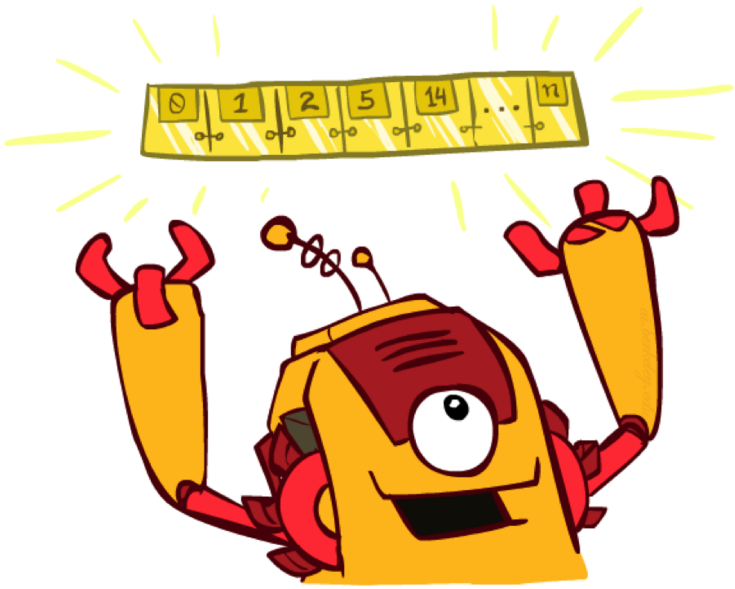




[Demo: empty grid UCS (L2D5)]
[Demo: maze with deep/shallow water DFS/BFS/UCS (L2D7)]

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



Up next: Informed Search

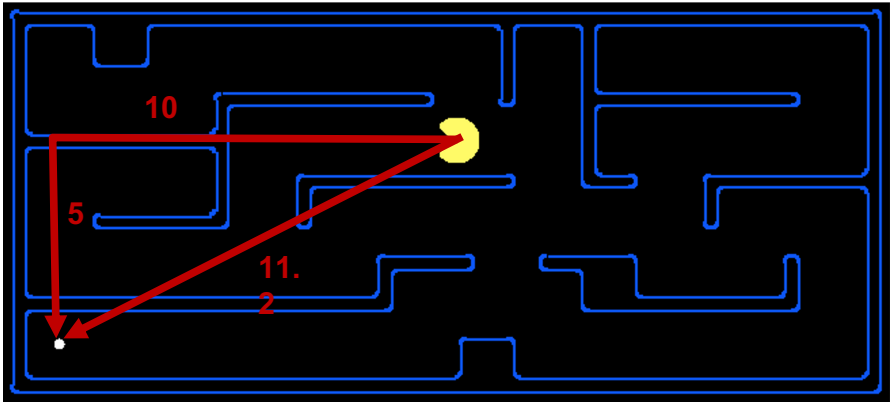
- Uninformed Search
  - DFS
  - BFS
  - UCS

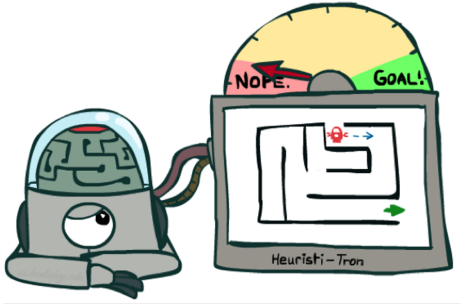
- Informed Search
  - Heuristics
  - Greedy Search
  - A* Search
  - Graph Search

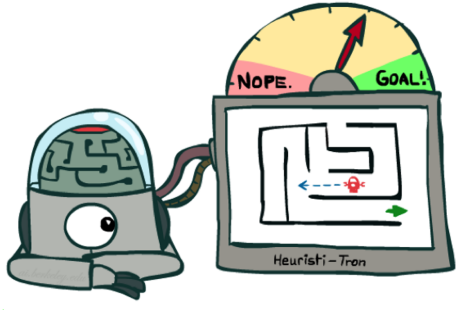
Search Heuristics

h - function

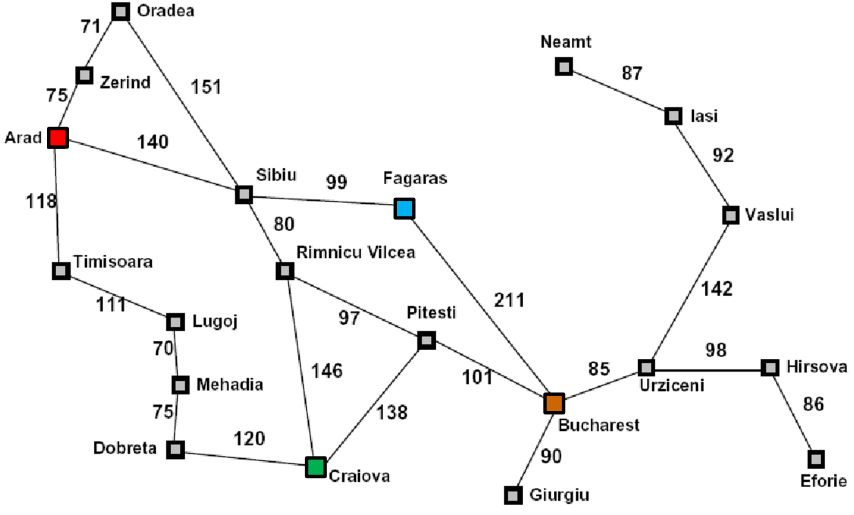
- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Pathing?
  - Examples: Manhattan distance, Euclidean distance for pathing







Example: Heuristic Function

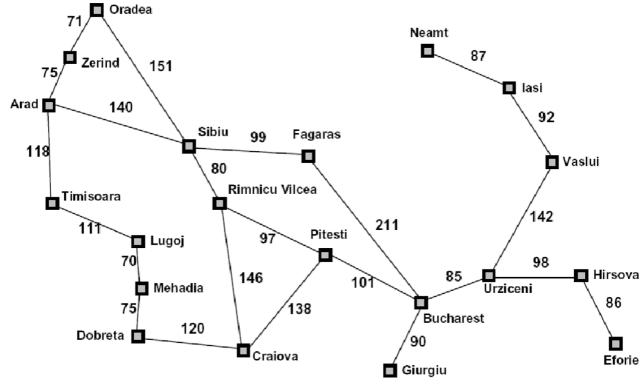


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

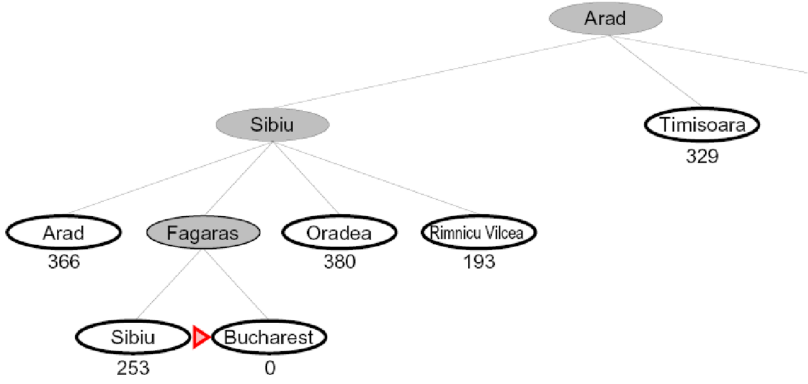
h(x)

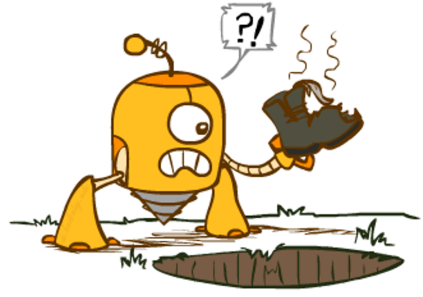
Greedy Search

Greedy Search



- Expand the node that seems closest…

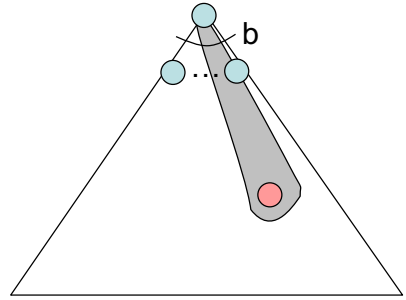




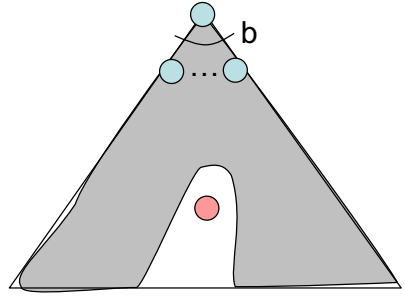
- Is it optimal?
  - No. Resulting path to Bucharest is not the shortest!

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



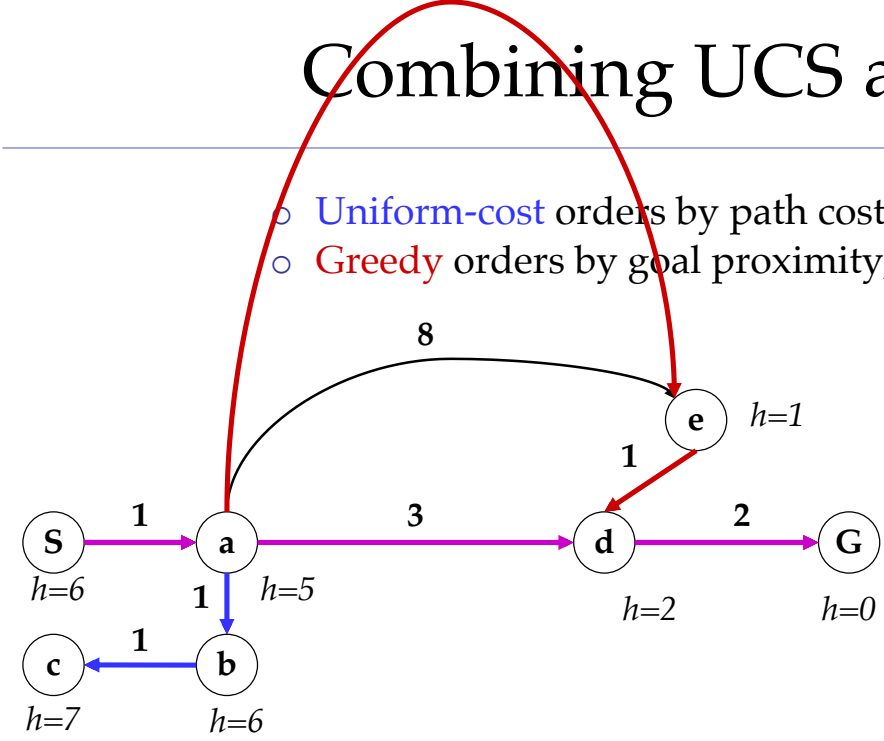
- Worst-case: like a badly-guided DFS

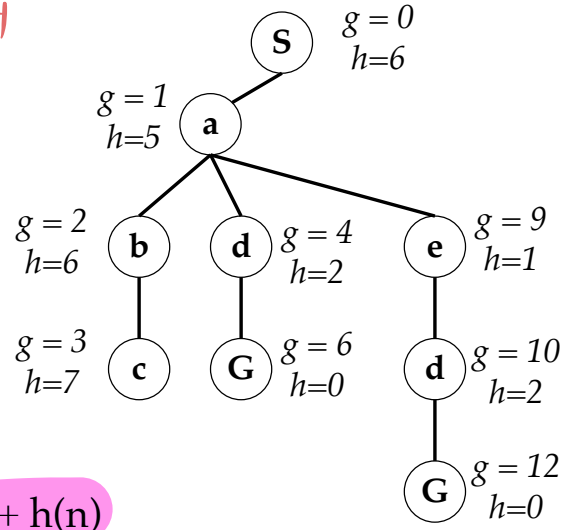
[Demo: contours greedy empty (L3D1)]
[Demo: contours greedy pacman small maze (L3D4)]

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

Algorithm Description

Suppose we are finding the shortest path from vertex a to a vertex z

The A* search algorithm initially:

- Marks each vertex as unvisited
- Starts with a priority queue containing only the initial vertex *a*
  - The priority of any vertex v in the queue is the weight $w(v)$ which assumes we have found the shortest path to v (initialize it to be infinity except for the initial vertex a)
  - Shortest weights have highest priority
- For each vertex *v*, *d(a, v)* is the shortest known distance from *a* to *v*, *d(a, a) = 0* and *d(a, v)= infinity* for all *v ≠ a*
- For each vertex *v*, *h(v, z)* is the heuristic distance from *v* to *z*

估值 到初始点 当前点到目标估值

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

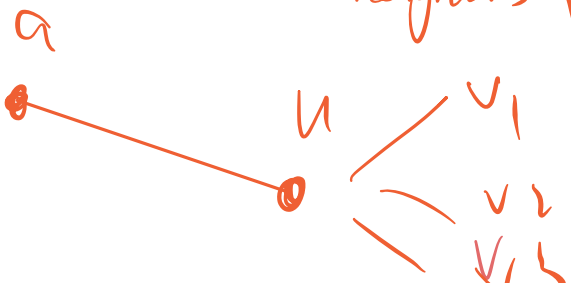
(多种选择)

Algorithm Description I (*Tree Search*)

The algorithm then iterates:

- Pop the vertex u with highest priority
- For each adjacent vertex (neighbor) v of u:
  - If $w(v) = d(a, u) + d(u, v) + h(v, z)$ is less than the current weight/priority of v, update the path leading to v and its priority
    - If v is not in the queue, push v into the queue

Continue iterating until the item popped from the priority queue is the destination vertex z



Algorithm Description II (*Graph Search*)

The algorithm then iterates:

- Pop the vertex u with highest priority
  - mark u as visited
- For each unvisited adjacent vertex (neighbor) v of u:
  - If $w(v) = d(a, u) + d(u, v) + h(v, z)$ is less than the current weight/priority of v, update the path leading to v and its priority
    - If v is not in the queue, push v into the queue

Continue iterating until the item popped from the priority queue is the destination vertex z

1. 首先需要创建两个集合，一个存储 待访问 的节点（nodeLists），一个存储 已经访问过 的节点(visitedNodeLists)

2. 添加 起点（2,1） 到 nodeLists 列表，并且计算该点的 预估值。

这里就需要用到上面讲到的预估函数了，

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

我们可以很清楚的看到，起点到当前节点的距离为0，即：$g(n) = 0$。

而当前顶点（2,1）到目标顶点（2,5）的距离为4，即：$h(n) = 4$。

这里的h值可以采用 **曼哈顿距离公式** 计算得来：

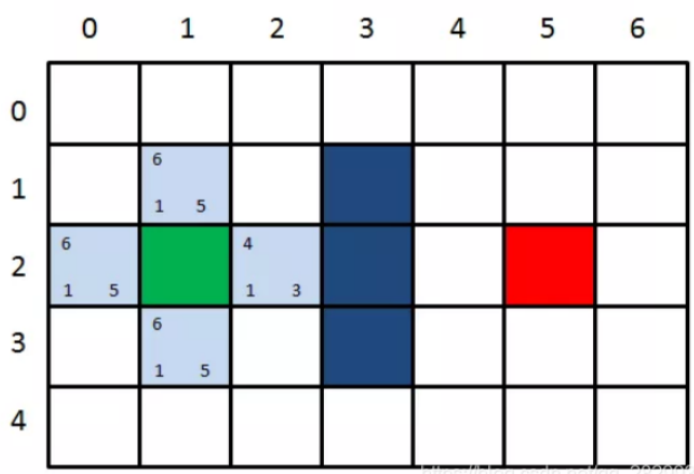
$h = |x_1 - x_2| + |y_1 - y_2|$

这里需要注意的是：h 值是在 不考虑障碍 的情况下计算的

3. 查找nodeLists 里预估值最小的节点，作为当前访问的节点，并且从nodeLists 删除 该节点

4. 获取 当前节点 的 邻居节点，计算出他们的预估值 并且添加到nodeLists列表中。

如图：




```
function startSearch(){
    //把起点添加到nodeLists
    nodeLists.push(startNode);
    //循环
    while (nodeLists.length > 0) {
        //查找预估值最小的节点
        let currentNode = findMinNode();
        //获取并且添加邻居节点到待访问列表
        findAndAddNeighborNode(currentNode);
        //把当前节点添加到已访问列表
        visitedNodeLists.push(currentNode);
        //判断是否是目标节点
        if (this.isTarget(currentNode)) {
            return currentNode
        }
    }
    return null
}
```

这里只考虑上下左右4个方位，并且分别计算了他们的各个预估值，

右下角：$h(n)$值

左下角：$g(n)$值

左上角：$f(n)$值

除了计算预估值，我们还需要把**当前节点**作为**每个邻居节点**的父节点，以便为了后面确定最终的路线。

这里需要注意的是：我们在添加到 **待访问列表**之前需要处理一下边界问题，并且判断一下是否是障碍物，如果是就不需要添加到列表中。

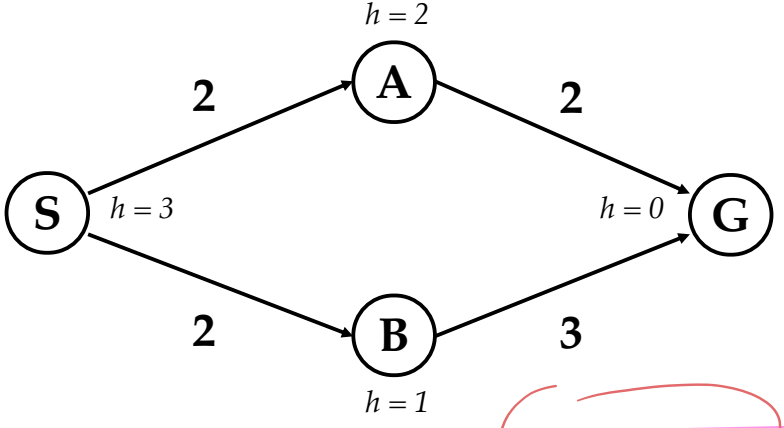
5. 把当前节点添加到visitedNodeLists中，代表已经访问过了。

6. 重复以上步骤 3 - 5 ，直到找到目标节点位置为止。

7. 循环输出 最终节点的父节点，就是我们需要的路径了。

When should A* terminate?

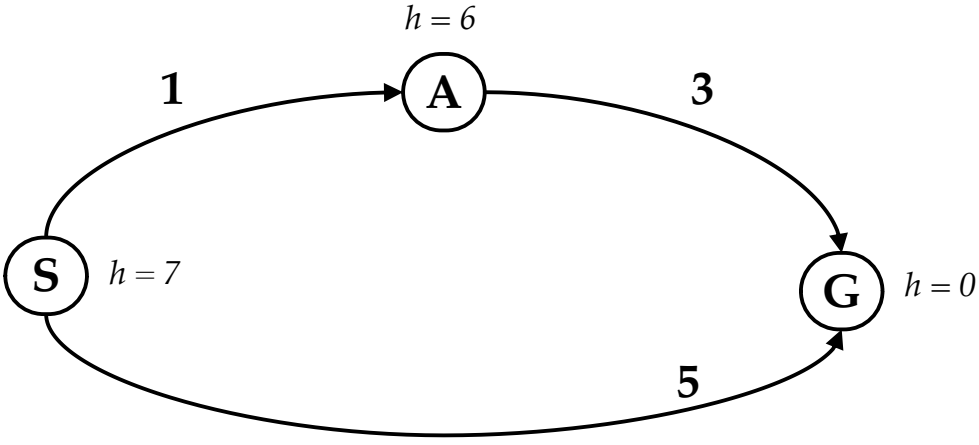
- Should we stop when we enqueue a goal?



	g	h	+
~~S~~	~~0~~	~~3~~	~~3~~
~~S->A~~	~~2~~	~~2~~	~~4~~
~~S->B~~	~~2~~	~~1~~	~~3~~
S->B->G	5	0	5
S->A->G	4	0	4

- No: only stop when we dequeue a goal

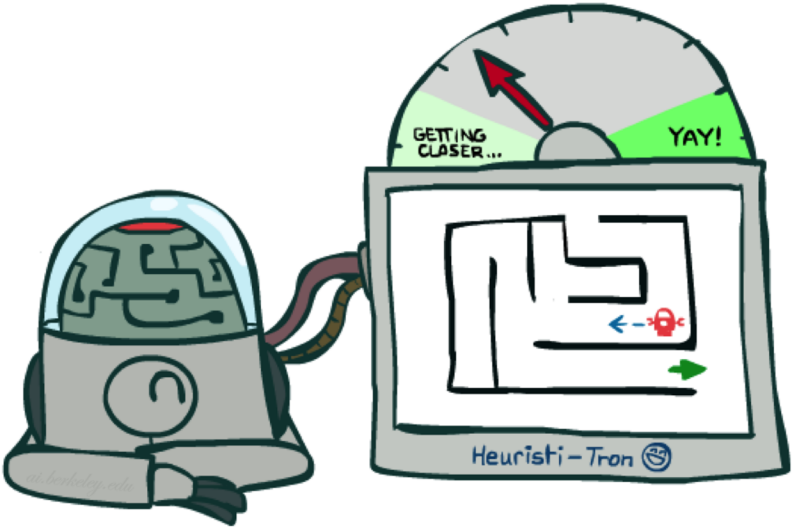
Is A* Optimal?



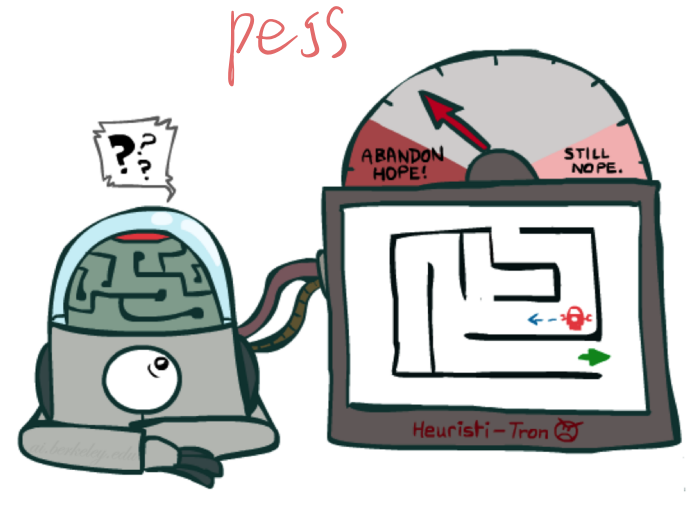
	g	h	+
~~S~~	~~0~~	~~7~~	~~7~~
S->A	1	6	7
S->G	5	0	5

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

Admissible Heuristics



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

Admissible heuristics h must always be optimistic:

- Let $d(u, v)$ represent the actual shortest distance from u to v
- A heuristic $h(u, v)$ is *admissible* if $h(u, v) \leq d(u, v)$
- The heuristic is *optimistic* or a *lower bound* on the distance

Using the Euclidean distance between two points on a map is clearly an admissible heuristic

- The flight of the crow is shorter than the run of the wolf

A problem with fewer restrictions on the actions is called a relaxed problem

- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

Theorem: If *h(n)* is admissible, A* using `TREE-SEARCH` is optimal

Consistent → non-decreasing

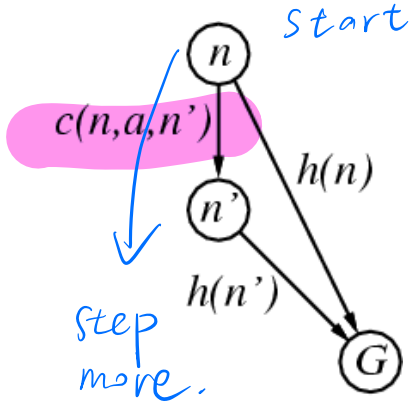
Consistent Heuristics

→ Graph

A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*, we have

$$h(n) \le c(n,a,n') + h(n')$$

consist → admissible.



If *h* is consistent, we have

w(n')	= d(n') + h(n')	(by def.)
	= d(n) + c(n,a,n') + h(n')	(d(n')=d(n)+c(n.a.n'))
	≥ d(n) + h(n) = w(n)	(consistency)
w(n')	≥ w(n)	

It's the triangle inequality !

i.e., *w(n)* is non-decreasing along any path.

keeps all checked nodes in memory to avoid repeated states

tree-search

Theorem:

If *h(n)* is consistent, A* using `GRAPH-SEARCH` is optimal

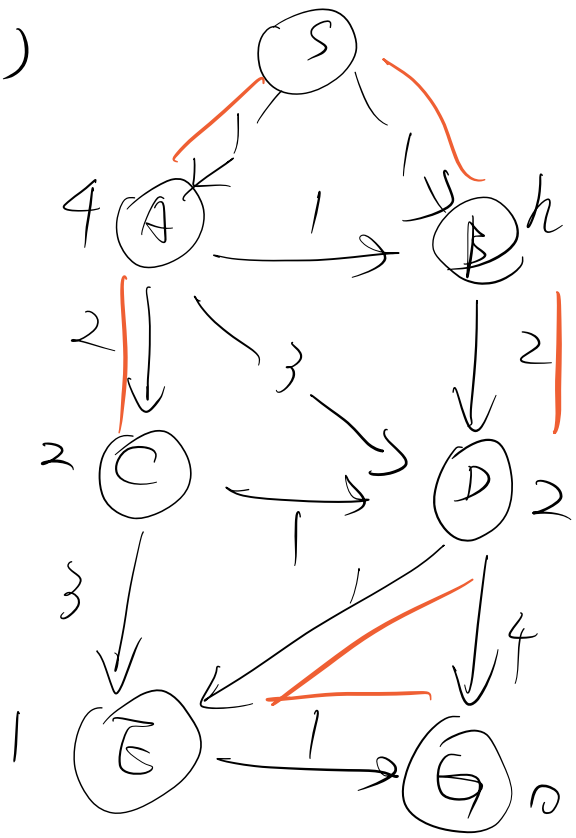
$h(n) \le c(n, a, n') + h(n')$

$h \le 2+2$.

$4 \le 1+h$

$5 \le 1+h$

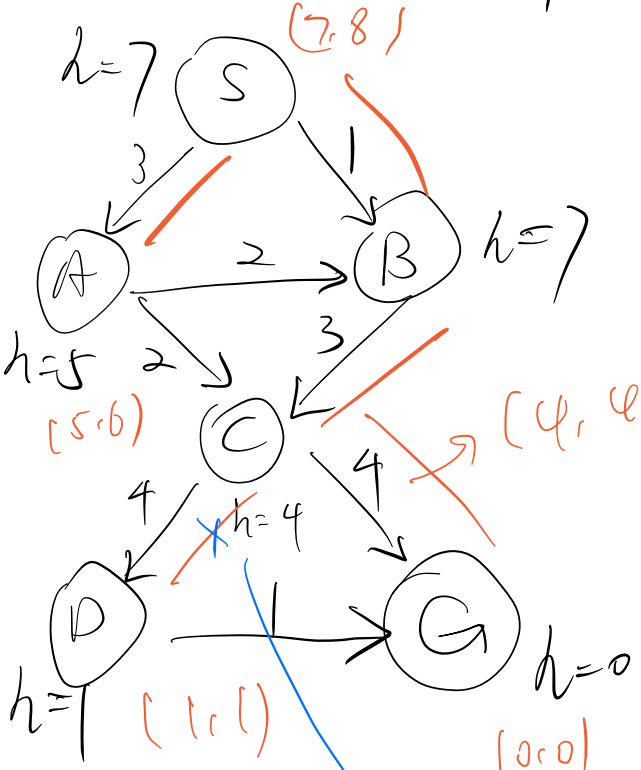
start



$h = 4$

$S \to A \to B \to C$

$\to D \to E \to G$



add: $h \le d$. ✓

consis: 多算, ok

G 先 expand, D 不算了

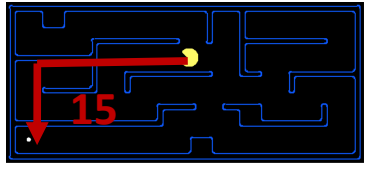
Admissible Heuristics

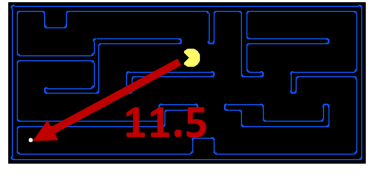
- A heuristic h is *admissible* (optimistic) if:

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

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

- Examples:

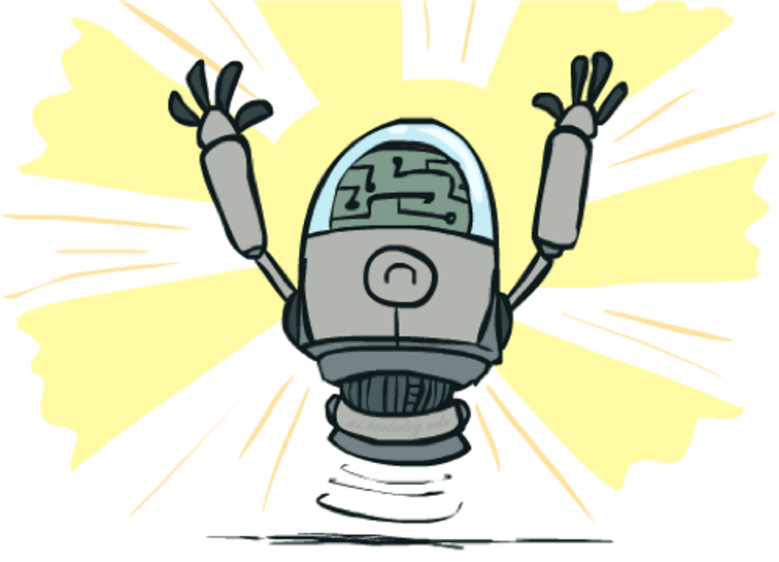




0.0

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

Optimality of A* Tree Search



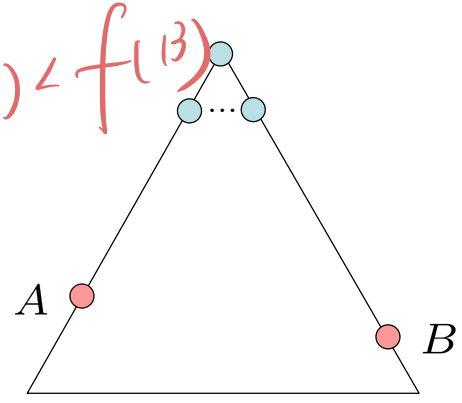
Optimality of A* Tree Search

Assume:

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

Claim:

- A will exit the fringe before B



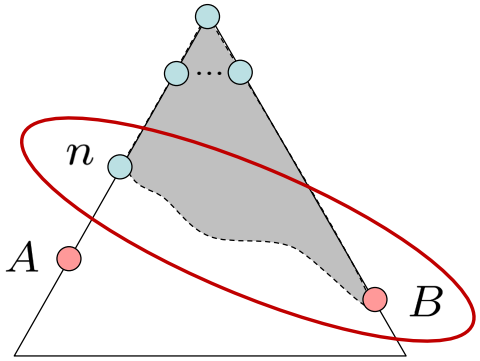
f(A) < f(B)

admissible → find optimal faster than suboptimal

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
  1. f(n) is less or equal to f(A)
  2. f(A) is less than f(B)
  3. *n* expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal



f(n) f(A) f(B)

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

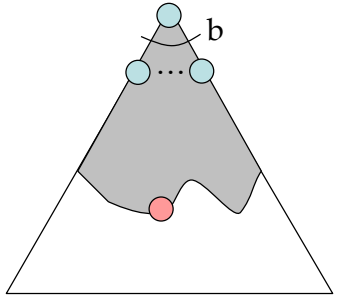
矛盾

all A's ancestor

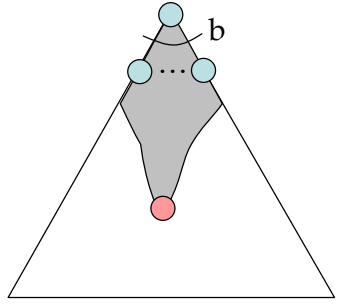
Properties of A*

expanded
→ A expanded

Uniform-Cost

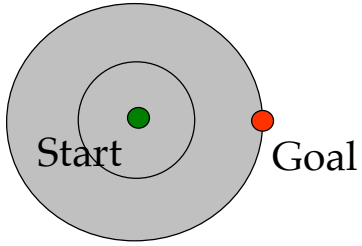


A*

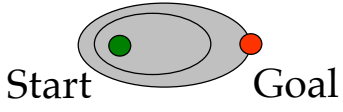


UCS vs A* Contours

- Uniform-cost expands equally in all "directions"

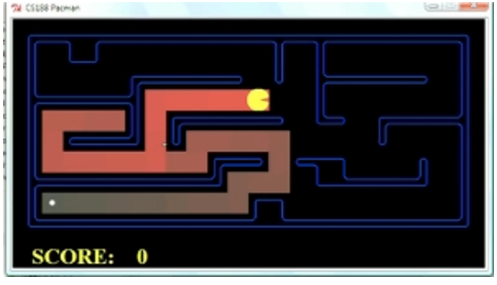


- A* expands mainly toward the goal, but does hedge its bets to ensure optimality

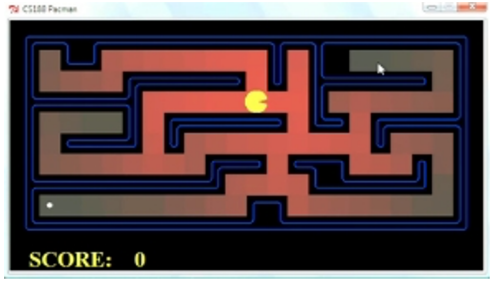


[Demo: contours UCS / greedy / A* empty (L3D1)]
[Demo: contours A* pacman small maze (L3D5)]

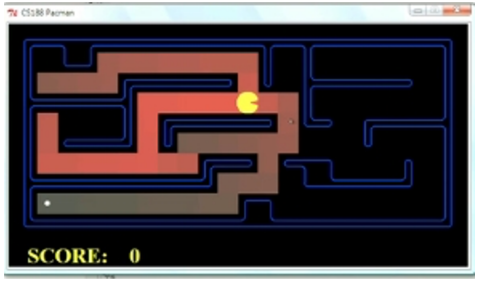
Comparison



Greedy



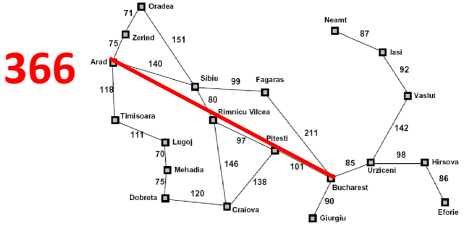
Uniform Cost

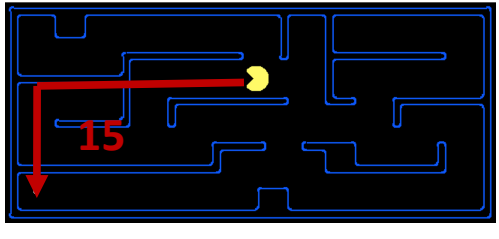


A*

Creating Admissible Heuristics

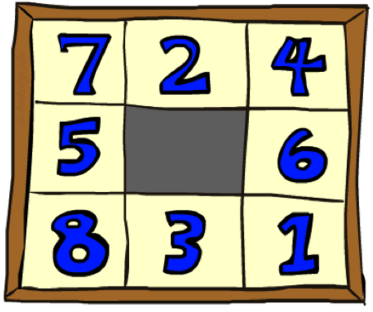
- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics
- Often, admissible heuristics are solutions to *relaxed problems*, where new actions are available



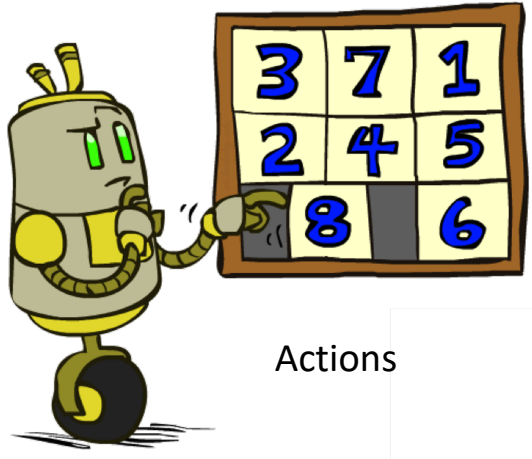


- Inadmissible heuristics are often useful too

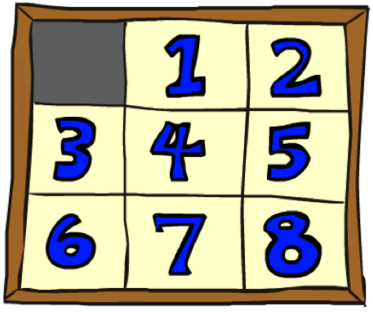
Example: 8 Puzzle



Start State



Actions



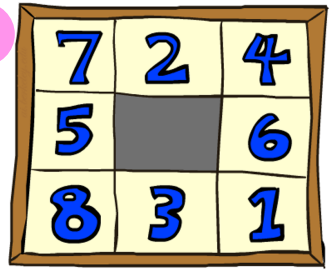
Goal State

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

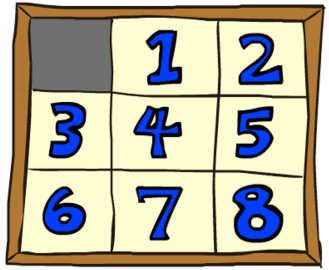
Admissible heuristics?

8 Puzzle I

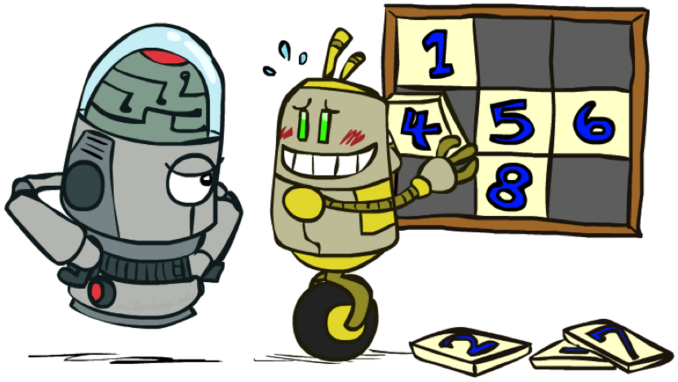
- Heuristic: Number of tiles misplace
- Why is it admissible?
- h(start) = 8
- This is a *relaxed-problem* heuristic



Start State



Goal State

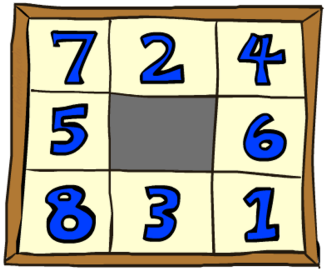


	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

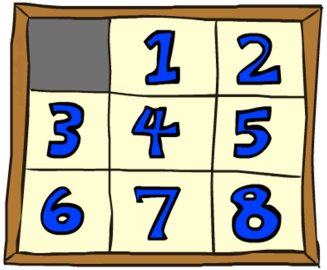
Statistics from Andrew Moore

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(start) = 3 + 1 + 2 + … = 18



Start State



Goal State

$|x_1 - x_2| + |y_1 - y_2|$

Euclidean

$\sqrt{(x_1 - x_2)^2 + (y - y_2)^2}$

$Lm = \left(|x_1 - x_2|^m + |y_1 - y_2|^m \right)^m$

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

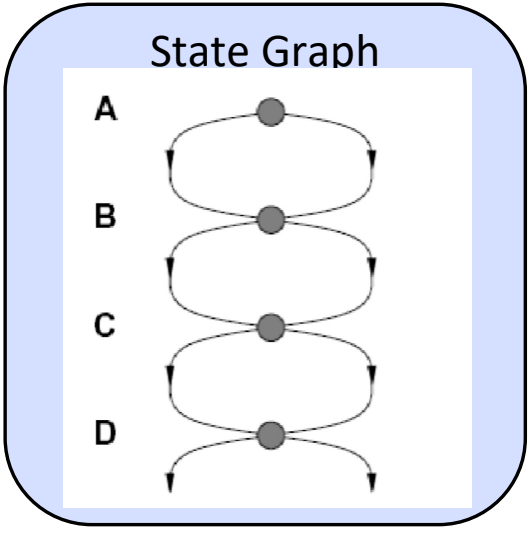
8 Puzzle III

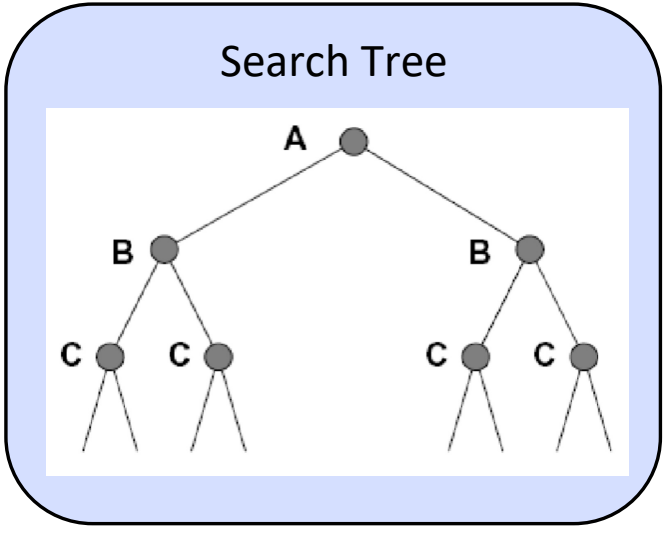
- How about using the *actual cost* as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?

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

Tree Search: Extra Work!

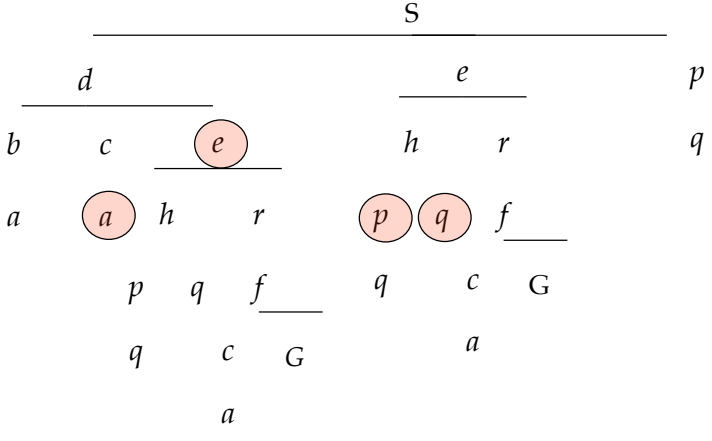
- Failure to detect repeated states can cause exponentially more work.





Graph Search

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

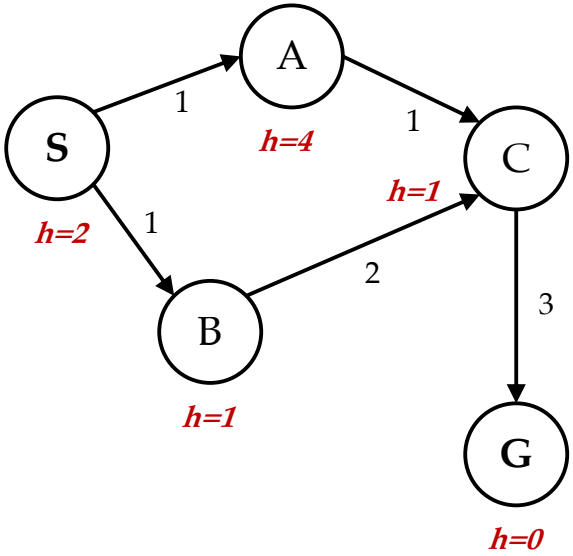


Graph Search

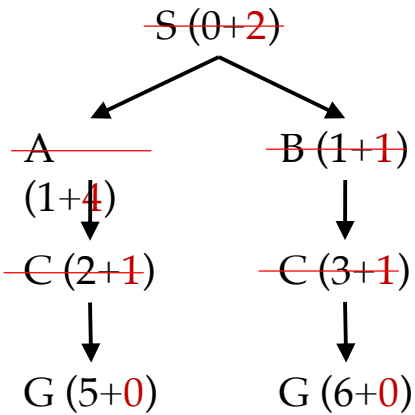
- Idea: never expand a state twice
- How to implement:
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
- Important: 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?

State space graph

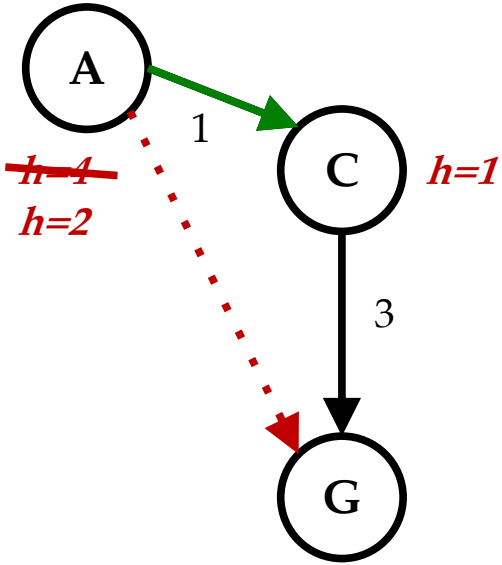


Search tree



Closed Set:S B C A

Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal

Consistent → non-decreasing

Consistent Heuristics

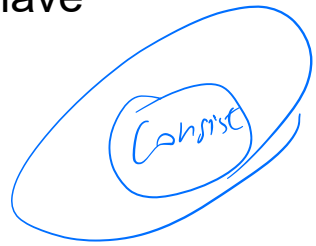
→ Graph

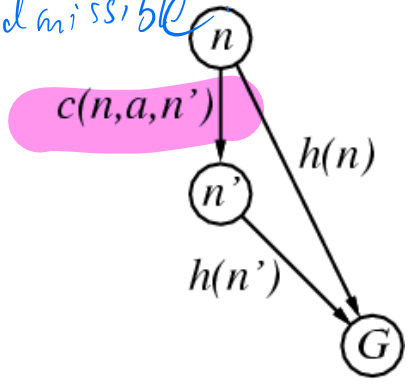
繼承父母

A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*, we have

$$h(n) \leq c(n,a,n') + h(n')$$

admissible





If *h* is consistent, we have

w(n')	= d(n') + h(n')	(by def.)
	= d(n) + c(n,a,n') + h(n')	(d(n')=d(n)+c(n.a.n'))
	≥ d(n) + h(n) = w(n)	(consistency)
w(n')	≥ w(n)	

It's the triangle inequality !

i.e., *w(n)* is non-decreasing along any path.

keeps all checked nodes in memory to avoid repeated states

tree-search

Theorem:

If *h(n)* is consistent, A* using `GRAPH-SEARCH` is optimal

Optimality of A* Search

- With a admissible heuristic, Tree A* is optimal.
- With a consistent heuristic, Graph A* is optimal.
  - See slides, also video lecture from past years for details.
- With h=0, the same proof shows that UCS is optimal.

tree is also.

UCS : h=0 A*
(all)

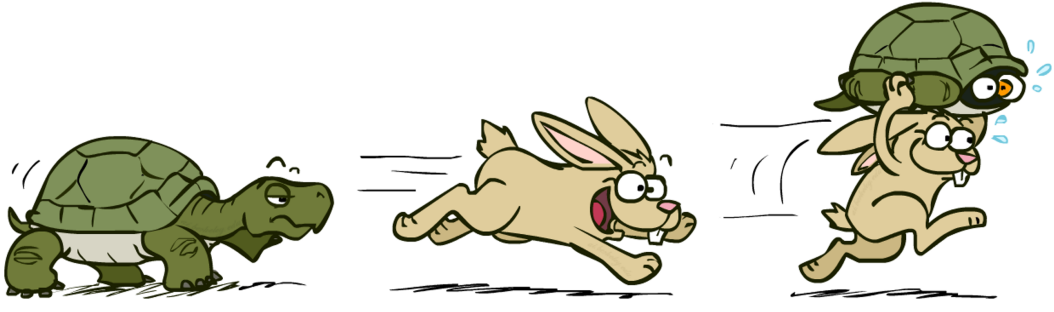
A*: Summary

- A* uses both backward costs and (estimates of) forward costs

count

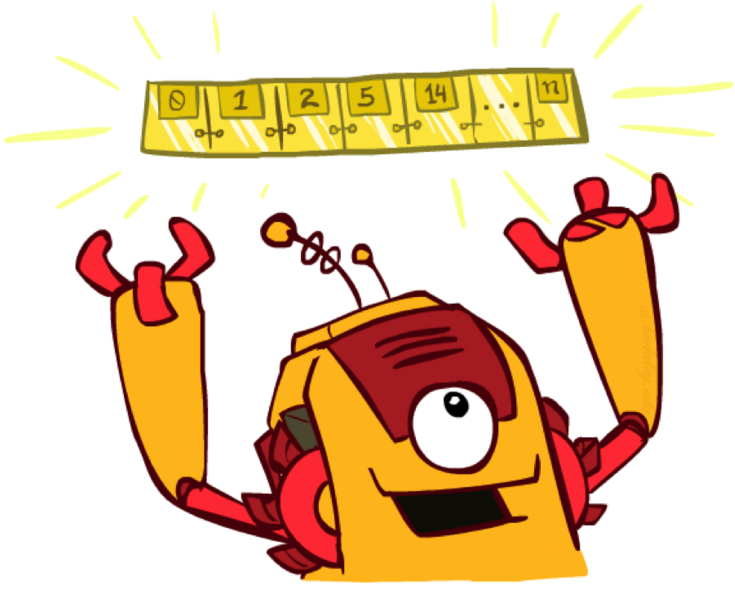
g + h

- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



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



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…