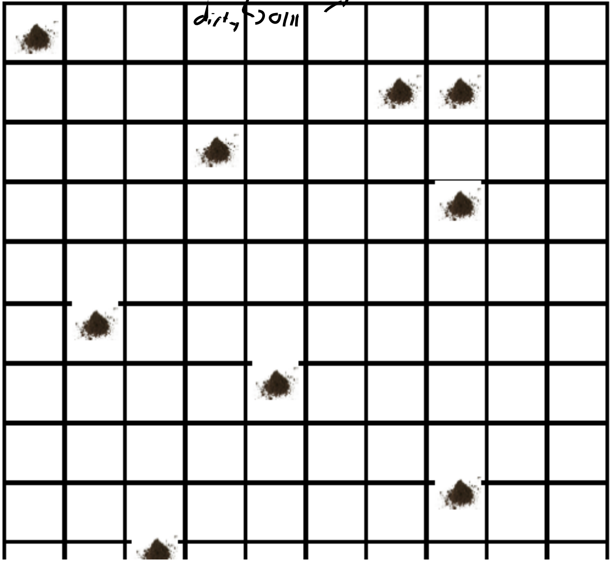
$\underbrace{1111}_{\text{dirty}} \to 1110 \to 0110$

$\to 0111 \to$

(no nb today, sorry ⌢̈)



branching factor:

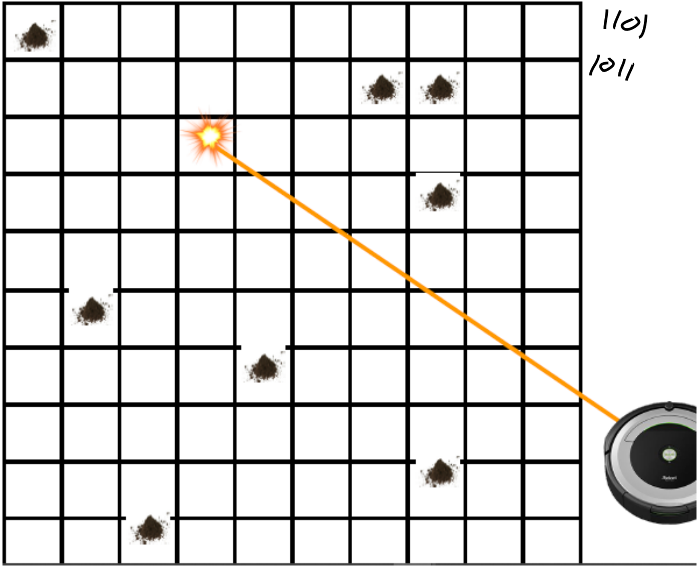
$b = N$

$O(b^{\varepsilon d})$

$\text{depth} = N$

$\therefore N^N$ ⌢̈

- Imagine we have a Roomba that has to clean N dirty tiles. Instead of moving, it just blasts tiles one at a time with a sweet laser!
- What is the optimal solution? Is it unique? How many paths will we explore?

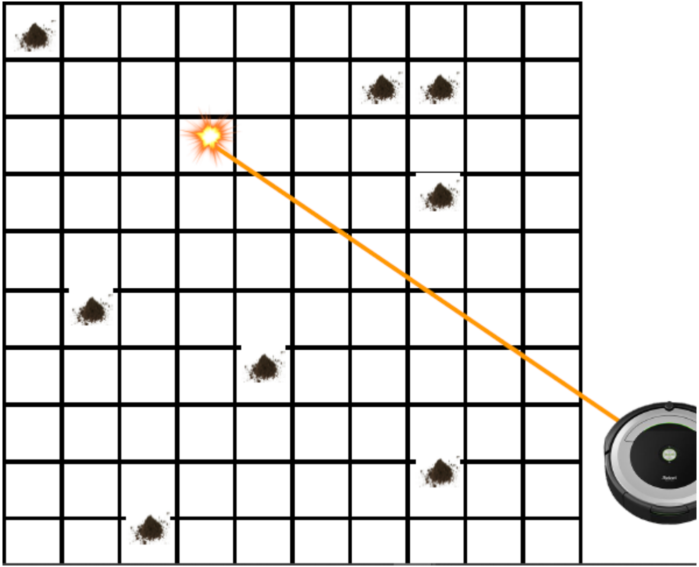


1101

1011

- Imagine we have a Roomba that has to clean N dirty tiles. Instead of moving, it just blasts tiles one at a time with a sweet laser!
- What is the optimal solution? Is it unique? How many paths will we explore?
- **Every** solution is equal. So our A^* will unfold all $\underline{2^N}$ states that might lead to one of the optimal solutions.

n! paths



- Imagine we have a Roomba that has to clean N dirty tiles. Instead of moving, it just blasts tiles one at a time with a sweet laser!
- What is the optimal solution? Is it unique? How many paths will we explore?

Announcements and To-Dos

Announcements:

1. HW 2 posted on Canvas!

if you're given a dictionary, you don't have reorder it

Last time we learned:

1. Some theory on A^*, and how its performance depends on heuristics.

$A*$ Recap:

1. A^* is optimal on a tree or graph if h is **consistent.** This means it must satisfy the triangle inequality (with edge costs as one edge), and is a slightly stricter requirement than *admissibility*: never overestimating.
2. We proved a Theorem: Any consistent heuristic is also admissible.
3. We also proved: If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing.
4. We also proved: Whenever A^* selects a node n for expansion, the optimal path to that node has been found.
5. From these, we concluded that A^* is **optimally efficient** for any given heuristic.

consistent

A* alternatives

Having to possibly track paths to many solutions can lead to a space complexity burden on A^*, since we keep all the generated nodes in memory.

Option: Iterative Deepening A^*.

1. Same idea as iterative depth-based search.
2. Instead of expanding the search depth, extend f up to an allowed cost threshold.
3. This searches out to a particular contour one at a time, then uses prior contours' solutions to generate paths out to the next deeper contour.

A* alternatives

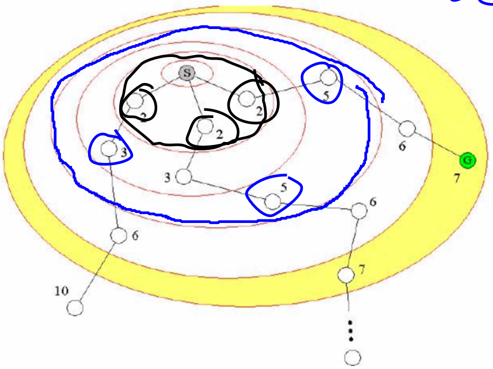
$f = h + g$

just did $f = g$: iterative UCS

Having to possibly track paths to many solutions can lead to a space complexity burden on A^*, since we keep all the generated nodes in memory.

Option: Iterative Deepening A^*.

1. Same idea as iterative depth-based search.
2. Instead of expanding the search depth, extend f up to an allowed cost threshold.
3. This searches out to a particular contour one at a time, then uses prior contours' solutions to generate paths out to the next deeper contour.

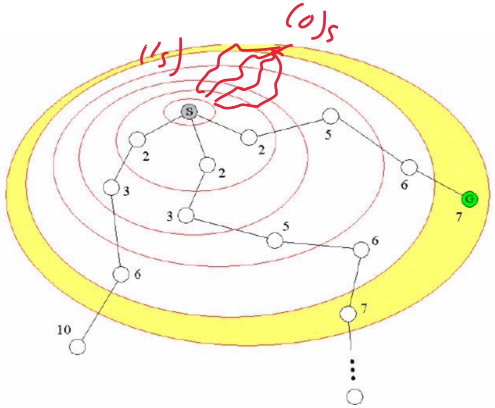


A^* alternatives

Having to possibly track paths to many solutions can lead to a space complexity burden on A^*, since we keep all the generated nodes in memory.

Option: Iterative Deepening A^*.

1. Same idea as iterative depth-based search.
2. Instead of expanding the search depth, extend f up to an allowed cost threshold.
3. This searches out to a particular contour one at a time, then uses prior contours' solutions to generate paths out to the next deeper contour.

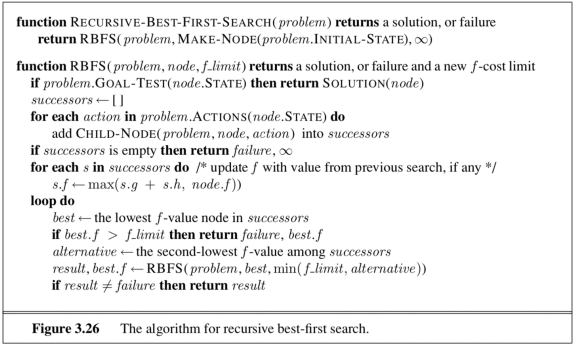


A^* alternative: RBFS

Option: Recursive best-first search (RBFS).

1. Kind of like a standard DFS, but track the best alternative path's cost f as we go.
2. Once the path we're on exceeds the currently saved best-available path, switch over to a backup path.
3. As RBFS back-tracks, each node along the back-tracked path it replaces the f-value with that of the cheapest child node. Put another way: we remember the best "leaf" in the sub-tree, so RBFS knows whether or not to go back down that road later.
4. Still has some indecision and replicated-computation problems, as expanding a new path adds to costs which makes unexpanded alternatives look better.

RBFS pseudo



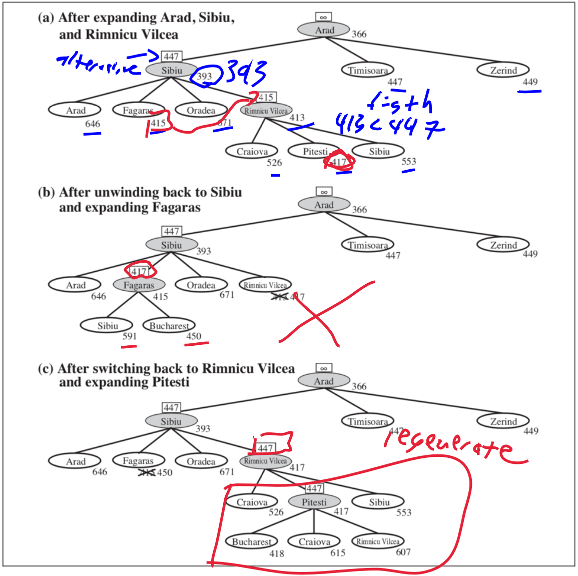
```
function RECURSIVE-BEST-FIRST-SEARCH($problem$) returns a solution, or failure
   return RBFS($problem$, MAKE-NODE($problem$.INITIAL-STATE), $\infty$)

function RBFS($problem, node, f\_limit$) returns a solution, or failure and a new $f$-cost limit
  if $problem$.GOAL-TEST($node$.STATE) then return SOLUTION($node$)
  $successors \leftarrow$ [ ]
  for each $action$ in $problem$.ACTIONS($node$.STATE) do
      add CHILD-NODE($problem, node, action$) into $successors$
  if $successors$ is empty then return $failure$, $\infty$
  for each $s$ in $successors$ do /* update $f$ with value from previous search, if any */
      $s.f \leftarrow \max(s.g + s.h,\ node.f)$)
  loop do
      $best \leftarrow$ the lowest $f$-value node in $successors$
      if $best.f > f\_limit$ then return $failure$, $best.f$
      $alternative \leftarrow$ the second-lowest $f$-value among $successors$
      $result, best.f \leftarrow$ RBFS($problem, best, \min(f\_limit, alternative)$)
      if $result \neq failure$ then return $result$
```

Figure 3.26 The algorithm for recursive best-first search.

RBFS depicted

recall: A* opens all nodes: $f(n) < C^*$ extive soln. cost.



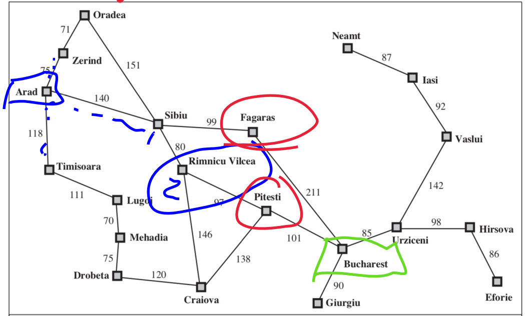


Figure 3.2 A simplified road map of part of Romania.

Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

Alternatives: complexity

Both IDA^* and RBFS use very little memory. For example, between iterations IDA^* keeps only the current $f-$cost limit, and RBFS has the memory benefits of DFS.

↳ 1 path at a time.

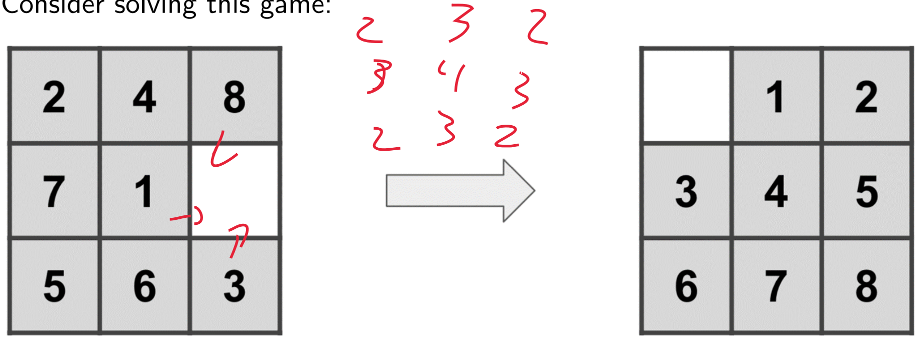
Unfortunately, those who do not remember the past are (maybe) doomed to repeat it!

Option: Memory-bounded A^* and Simplified Memory bounded A^* (MA^* and SMA^*.)
SMA^*:

1. Expands the best leaf *until memory is full*.
2. Then expands the best leaf and deletes the worst.
3. Concern: what if all leaves have the same $f-$ value? Then expand the newest and delete the oldest.
4. Concern: what if there is only one leaf? Then there is only one solution path from that root to the goal, and it's taking up all available memory as is. We can't solve the problem anyways.

Sept 18: Heuristics

Consider solving this game:



2	4	8
7	1	
5	6	3

2 3 2
3 4 3
2 3 2

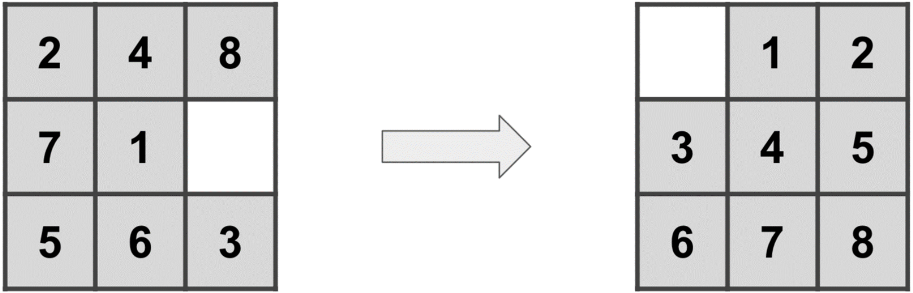
	1	2
3	4	5
6	7	8

What is the branching factor? Approximate or average solution depth? How many reachable states?

$b \approx 3$

Sept 18: Heuristics

Consider solving this game:



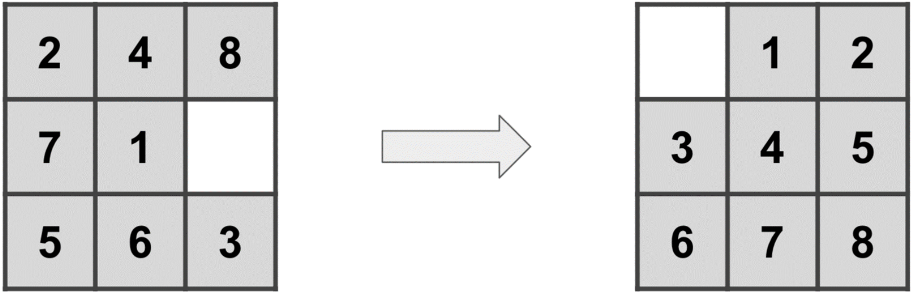
2	4	8
7	1	
5	6	3

	1	2
3	4	5
6	7	8

What is the branching factor? Approximate or average solution depth? How many reachable states?

$b \approx 3$, The average solution depth is around 22 moves. This means that DFS solution might expand $3^{22} \approx 3.1 \cdot 10^{10}$ nodes. The graph representation may require managing $9!$ total states, and for any given problem half of these are *reachable*.

Generating Heuristics



2	4	8
7	1	
5	6	3

	1	2
3	4	5
6	7	8

How do we come up with heuristics that might make A^* solve this type of game quickly? There are typically 3 ways:

1. Generate heuristics from relaxed problems. (What if we have more movements available?)
2. Generate heuristics from sub-problems. (What do we do on a 2×2 board? 3×3? $n \times n$?)

node!

3. Learn and use heuristics from experience. ("Rules-of-thumb")

The Tile Game

2	4	8
7	1	
5	6	3

1) 4
2) 0
3) 5
4) 0
5) 2
6) 2
7) 1

= $\boxed{14}$

even!

=

→

	1	2
3	4	5
6	7	8

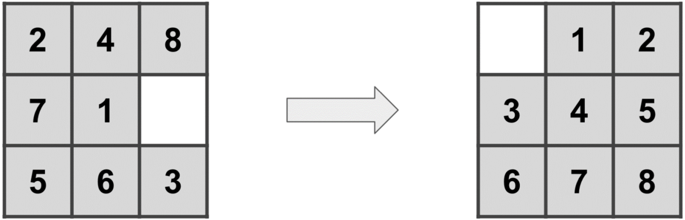
9! boards, only $\frac{9!}{2}$ are reachable.

1 2 → 0/1, , 2 or, 1,2

An aside: it turns out that whether this game is solvable - which half of the 9! states we can reach - is a function of the number of *inverted tile-pairs*.

Take the list [2,4,8,7,1,5,6,3] and compute how many swaps we'd have to make to put it into the goal state order [1,2,3,4,5,6,7,8]. If that value is even, the puzzle is solvable. If the value is odd, we can't reach the goal state.

The Tile Game



2	4	8
7	1	
5	6	3

	1	2
3	4	5
6	7	8

An aside: it turns out that whether this game is solvable - which half of the $9!$ states we can reach - is a function of the number of *inverted tile-pairs*.

Take the list $[2{,}4{,}8{,}7{,}1{,}5{,}6{,}3]$ and compute how many swaps we'd have to make to put it into the goal state order $[1{,}2{,}3{,}4{,}5{,}6{,}7{,}8]$. If that value is even, the puzzle is solvable. If the value is odd, we can't reach the goal state.

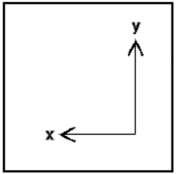
Solution: The parity as depicted here is 14. This is solvable.

The Tile Game Heuristics

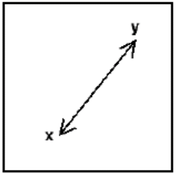
Making sure we find that solution without exploring millions or billions of nodes requires choosing a proper heuristic. Recall: consider A^* with a heuristic with relative error ε. This gave complexity $\underbrace{O(b^{\varepsilon d})}$ if step costs are constant (and solution is at depth d).

Two heuristics that might make sense: want ε small!

1. $h_1 :=$ the number of misplaced tiles. $h_1 \in \{0, \ldots, 8\}$
2. $h_2 :=$ sum of the distances between tiles and their goal positions. We would probably use *Manhattan* distance here, as our movements involve only Up-Down-Left-Right actions, which Manhattan distance captures. $h_2 \in \{0, \ldots, 32?\}$



Manhattan



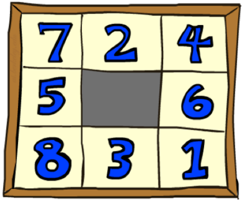
Euclidean

$$- : d(x,y) = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 + \cdots}$$

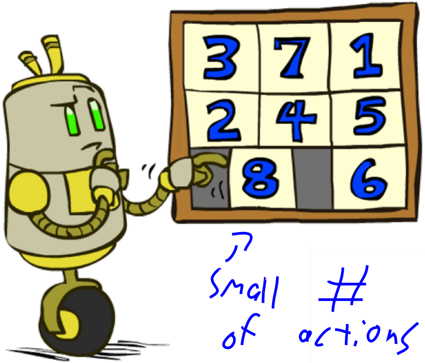
$$d(x,y)^{\text{manh.}} = |x_1-y_1| + |x_2-y_2| + |x_3-y_3|$$

Relaxed games

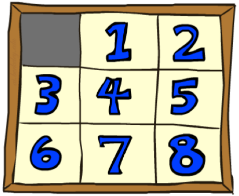
Why does Manhattan distance make sense here? It's representative of a solution cost to a *relaxed* game where we could "cheat" or make extra moves.



A starting state



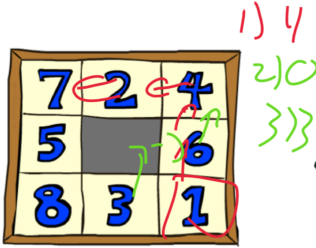
small # of actions



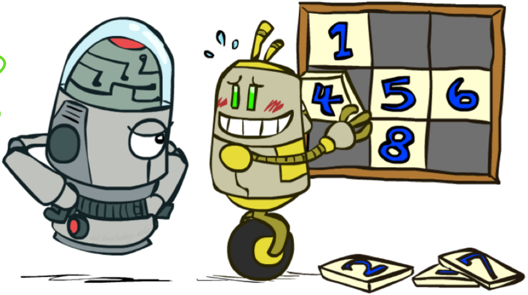
A goal state

Relaxed games

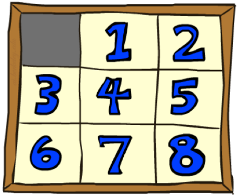
Why does Manhattan distance make sense here? It's representative of a solution cost to a *relaxed* game where we could "cheat" or make extra moves.



A starting state



More moves



A goal state

Suppose we took this 8-tile problem, but now you're allowed to "stack" tiles.

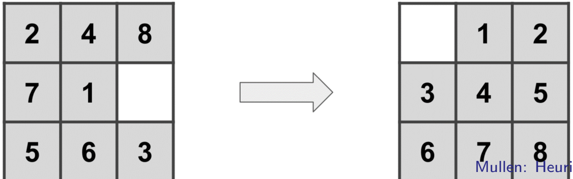
Just Relax!

Formally, a relaxed problem is a problem related to the original, but with fewer restrictions on our actions. This means that there are more connections between states in the state space, but also that the depth of the solution will be much shallower.

(possibly more states!)

If we relax the problem enough, we can often find the depth of the solution without much work! If we can stack tiles in the 8-tile problem, the solution path is *exactly* the h_2 Manhattan distance to the goal state summed over all tiles.

Optimal solutions of the relaxed problem are **always** admissible heuristics, since more valid, equal-cost actions will always lead to a solution thats "at-least-as-fast" as the targeted full-game solution.



2	4	8
7	1	
5	6	3

	1	2
3	4	5
6	7	8

Relaxed Heuristics

The two heuristics proposed before are just versions of optimal solutions to relaxed games.

1. If tiles can teleport (any distance in one move), the optimal solution is $h_1 :=$ the number of misplaced tiles.
2. If tiles can stack atop one another, the optimal solution is $h_2 :=$ sum of the Manhattan distances between tiles and their goal positions.

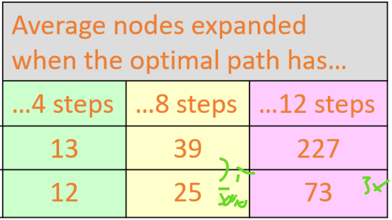
Is one of these better than another? Why or why not?

Relaxed Heuristics

The two heuristics proposed before are just versions of optimal solutions to relaxed games.

1. If tiles can teleport (any distance in one move), the optimal solution is $h_1 :=$ the number of misplaced tiles.
2. If tiles can stack atop one another, the optimal solution is $h_2 :=$ sum of the Manhattan distances between tiles and their goal positions.

Is one of these better than another? Why or why not?



#	Average nodes expanded when the optimal path has...		
	...4 steps	...8 steps	...12 steps
TILES : h_1	13	39	227
MANHATTAN h_2	12	25	73

3x faster!

admissible: $h^* \geq h_1$ ← $(h_1) \geq h_2$

true cost

closer to true.

Choice: $f = g + h$

cost to node

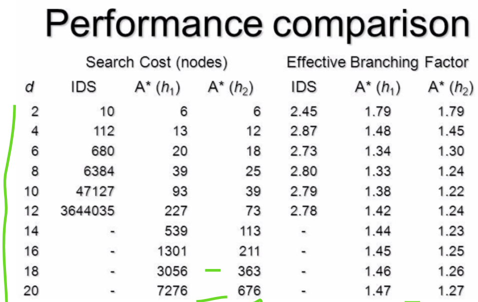
est. distance to sol.

(or not further)

Choosing Heuristics

When we have a bunch of heuristics, which do we choose?

1. The best case is one heuristics **dominates** the other. It's *always* closer to the true value, or: $h* \geq h_2 \geq h_1$ would represent h_2 dominating h_1. This leads to a fast A^*.
(p.)
2. More generally, we can just try to take the most accurate heuristic for whichever node we're currently at, or $h(n) = \max\left(h_1(n), h_2(n), \dots\right)$



Performance comparison

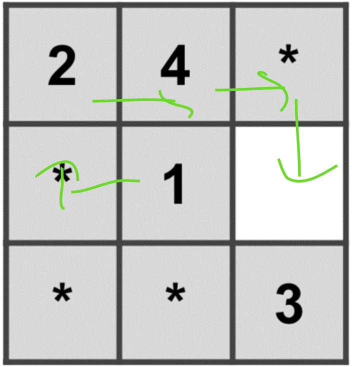
	Search Cost (nodes)			Effective Branching Factor		
d	IDS	A* (h_1)	A* (h_2)	IDS	A* (h_1)	A* (h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	-	539	113	-	1.44	1.23
16	-	1301	211	-	1.45	1.25
18	-	3056	363	-	1.46	1.26
20	-	7276	676	-	1.47	1.27

n times more per node

For the 8-tile game:
h_2 dominates h_1.

Sub-problem

Another source of heuristics might be solving sub-problems, and using induction, recursion, or estimation to apply those distances to the larger problem.



2	4	*
*	1	
*	*	3

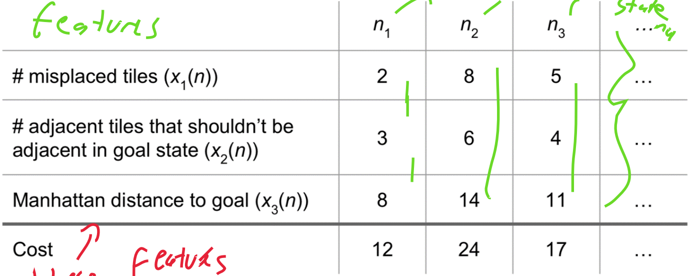
Consider only arranging the tiles $\{1, 2, 3, 4\}$ to their proper spots.

1. The cost of a solution to the subproblem is definitely cheaper than the cost of the optimal solution to the full problem.
2. This can lead us to constructing a pattern **database**
   2.1 Solve all or many possible configurations of the sub-problem
   2.2 Record the cost of the optimal solution of that sub-problem
   2.3 Use this as a heuristic
   2.4 Or include heuristics from *multiple* sub-problems, combined somehow.

Heuristic Learning

Suppose we solved thousands of 8-tile problems. We now have tons of initial states, but also tons of data: lengths of solution paths, initial state values of h_1, h_2, etc.

We could try to *model* the true h^* as a function of our easier-to-compute h_1 and h_2!



features	n_1	n_2	n_3	…
# misplaced tiles ($x_1(n)$)	2	8	5	…
# adjacent tiles that shouldn't be adjacent in goal state ($x_2(n)$)	3	6	4	…
Manhattan distance to goal ($x_3(n)$)	8	14	11	…
Cost	12	24	17	…

State

n_4

these

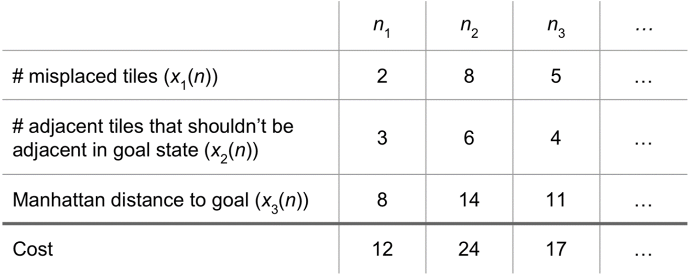
Features predict: cost-to-solve

Heuristic Learning

We could try to *model* the true h^* as a function of our easier-to-compute h_1 and h_2! Maybe a multiple linear regression model? We could predict the cost from the features of the initial states:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) + c_3 x_3(n) + \dots$$

Those features might include other heuristics, or whatever else you can think of!



	n_1	n_2	n_3	…
# misplaced tiles ($x_1(n)$)	2	8	5	…
# adjacent tiles that shouldn't be adjacent in goal state ($x_2(n)$)	3	6	4	…
Manhattan distance to goal ($x_3(n)$)	8	14	11	…
Cost	12	24	17	…

may not be
consistent/admissible

Heuristic Underview

Fundamentally, these things represent a computation trade-off between domination and efficiency.

1. A learning heuristic might not even be admissible or consistent, as this depends on the coefficients and features!
2. More generally, A^* using h_2 on these problem will never expand more nodes than using h_1: recall that every node with $f(n) < C^*$ gets expanded, but since $\underline{f(n)} = g(n) + \underline{h(n)}$, a smaller h shrinks that left-hand-term and might lead to many fewer expansions. So it's best to use a heuristic with higher values.
3. But there's a cost! 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.

Moving Forward

- Next Week:
  1. Finish up Searching
  2. Additional Office Hours!
- Next time: Local Search!