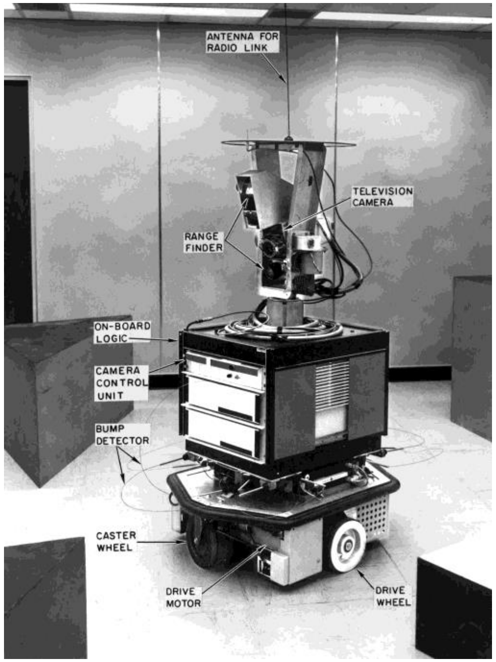
Artificial Intelligence

Steve James

Search


ANTENNA FOR RADIO LINK
TELEVISION CAMERA
RANGE FINDER
ON-BOARD LOGIC
CAMERA CONTROL UNIT
BUMP DETECTOR
CASTER WHEEL
DRIVE MOTOR
DRIVE WHEEL

Search

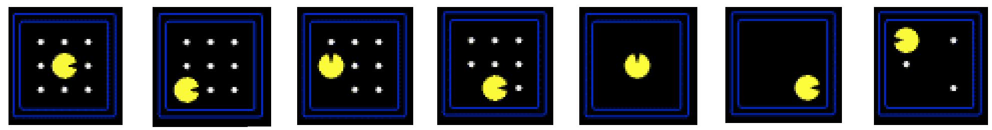
- Search = find solution to reach goal
- Automated theorem proving:
  - Start with axioms, find steps to reach theorem
- Checkers/Chess
  - Start at position, find moves to win

Definitions

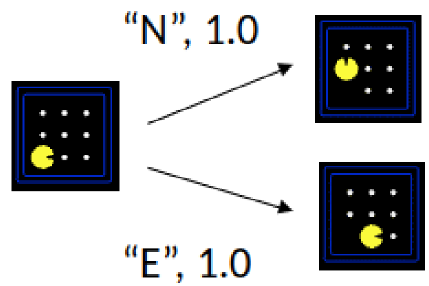
- Agent: a decision-making entity
  - E.g. robot, software code
- State: a representation of agent's world/environment at a given time
- State space: the space of all possible states
- Start in an initial state
- Successor function: what actions agent can take at a given time, the cost of doing so, and what happens next
- Goal test: function that, given state, returns true if a goal condition has been met

Search problem

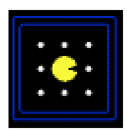
- Consists of:
  - A state space



  - Successor function



  - Start state

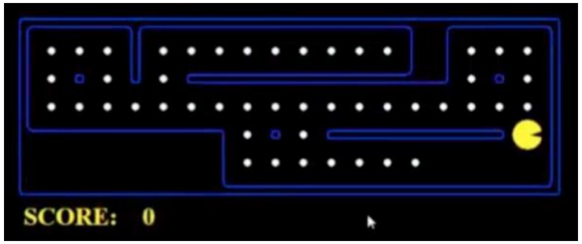


  - Goal test: $test(state) = \begin{cases} true\ if\ dots\ all\ eaten \\ false\ otherwise \end{cases}$

- Solution is sequence of actions (plan) that transforms start state into a goal state
- Optimal solution: minimises total cost from start to goal

Defining the problem

- Challenge is taking problem to be solved and deciding what the state space should be
- A state need only keep details necessary for planning
  - i.e. can be abstracted



- Problem: navigation
  - States: xy-location; actions: NSEW; successor: update location; goal test: (x, y) = TARGET
- Problem: eat all dots:
  - States: xy-location + dot booleans; actions: NSEW; successor: update location and dot boolean; goal test: dots all false

State space = locations x 2^dots

Formal Definition

- Set of states S
- Start state $s \in S$
- Set of actions A and rules $a(s) \to s'$
- Goal test $g(s) = \{0, 1\}$
- Cost function $C(s, a, s') \to \mathbb{R}^+$

- Search problem: $\langle S, s, A, g, C \rangle$

Problem statement

Find a sequence of actions $a_1, \ldots, a_n$

and corresponding states $s_1, \ldots, s_n$ such that:

$$s_0 = s$$
$$s_i = a_i(s_{i-1}), i = 1, \ldots, n$$
$$g(s_n) = 1$$

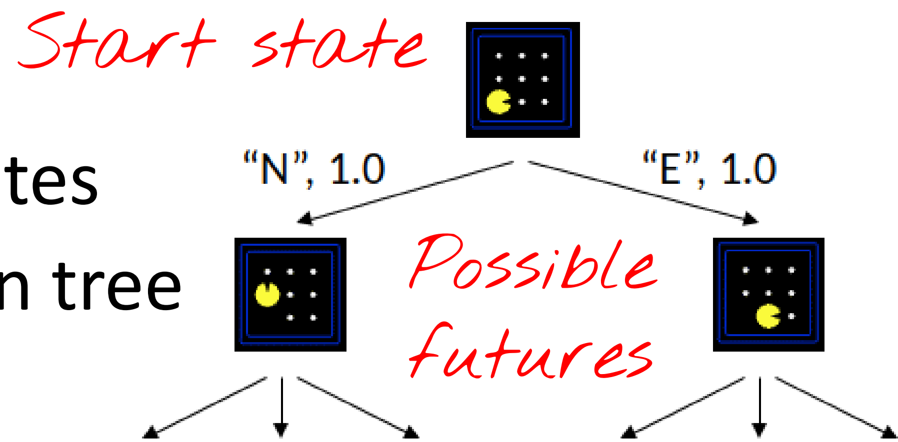
while minimising

$$\sum_{i=1}^{n} C(s_{i-1}, a_i, s_i)$$

Search tree

- A tree that shows future outcomes of actions!

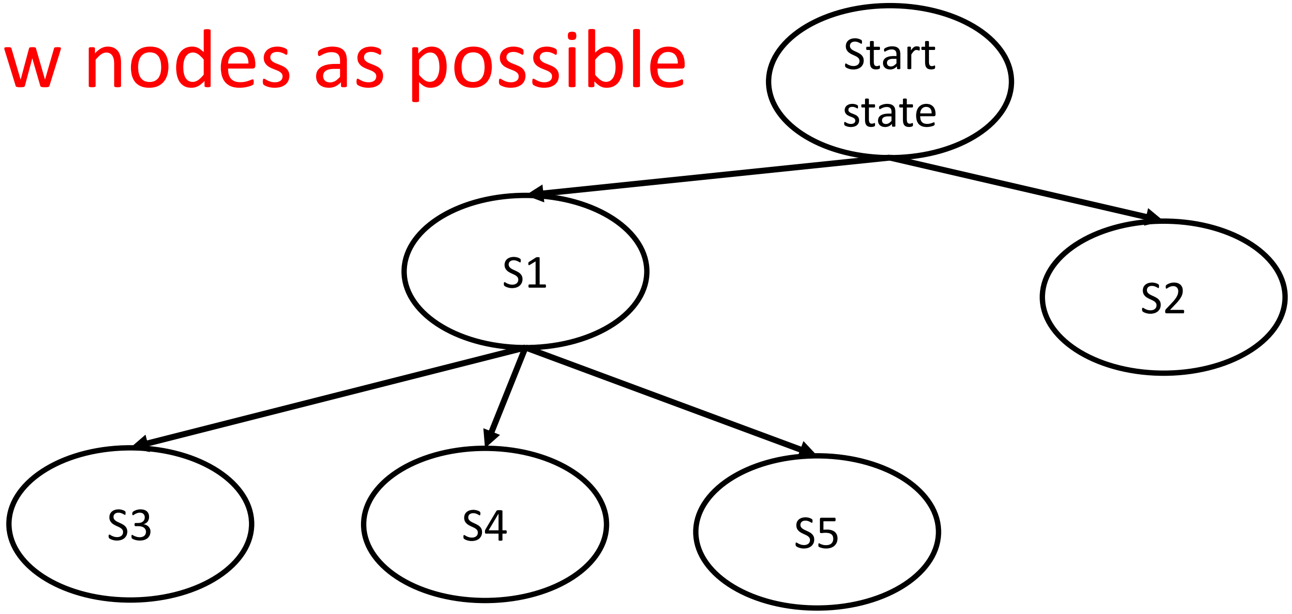
- Root is start state
- Children are successor states
- Edges from root to node in tree
  - Edges = plan!
  - Sum of edge costs = cost of plan
- For most problems, we can't build the full tree!
  - $O(b^d)$ where d is the max plan length



This is the challenge!

Planning with search tree

- Expand tree nodes (potential plans)
- Maintain frontier of partial plans under consideration
- Try come up with solution while expanding as few 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
```

Here is where we can be clever!

- Important ideas:
  - Frontier
  - Expansion
  - Exploration strategy
- Which frontier nodes to explore?

Implementation detail:
Remember visited states.
Don't add them more than
once to tree!

Search strategies

- Tree search algorithms are about picking order of node expansion
- Things to consider:
  - Completeness: will it always find a solution if it exists?
  - Optimality: will it always find a least-cost solution?
  - Time complexity: number of nodes generated
  - Space complexity: max nodes in memory

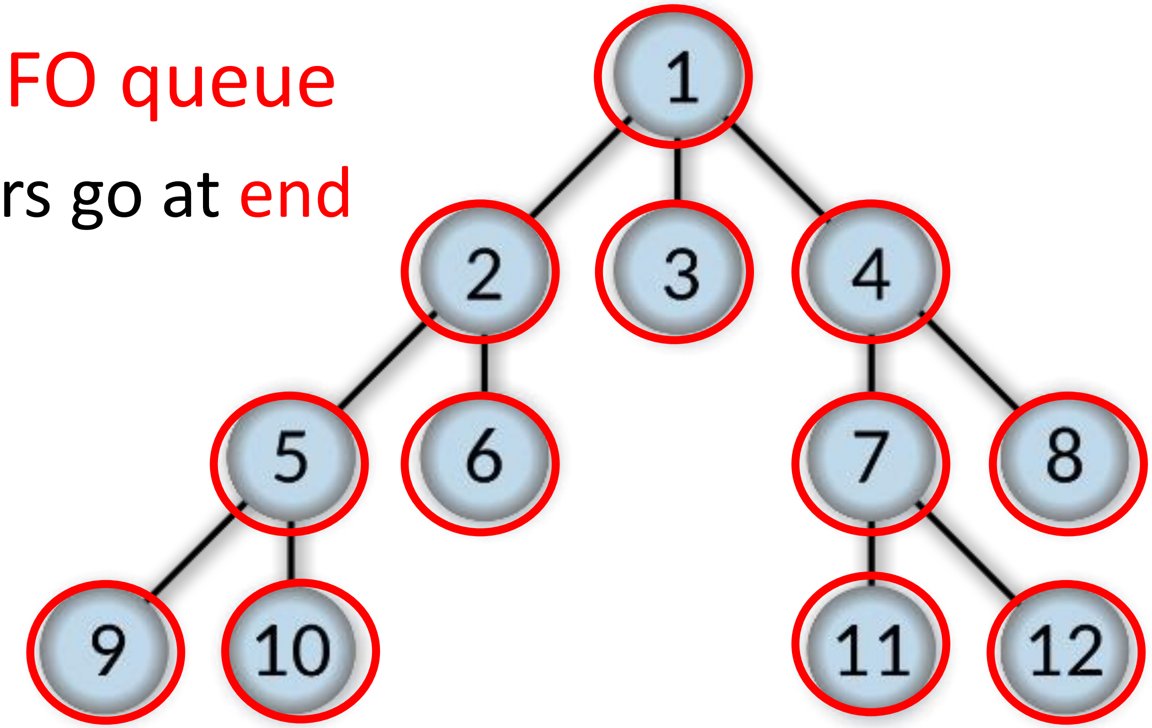
- Complexity depends on max branching factor, depth of optimal solution, max depth of state space

Uninformed search

- Uninformed search: we use only the info in the problem definition
- If we had additional domain knowledge, we could use informed/heuristic search (later)
- General uninformed strategies:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search

Breadth-first search

- Expand shallowest unexpanded node
- Frontier as a FIFO queue
  - New successors go at end



Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

BFS properties

- Let d be depth of shallowest solution

- Complete?
  – Yes (if b is finite). Shallowest solution returned
- Time?
  – $1 + b + b^2 + \ldots + b^d = O(b^d)$
- Space?
  – Keeps all frontier nodes in memory: $O\left(b^d\right)$
- Optimal:
  – Only if costs are constant

BFS Issues

- Both time and space are $O(b^d)$
- Assume:
  - we can generate 1m nodes/sec
  - 1 kbyte/node
  - $b = 10$
- Then, for $d = 8$
  - Time is 2 mins, but memory is 102GB
- For $d = 12$
  - Time is 13 days, but memory is 1000TB

Memory is massive issue!

Uniform-cost search

- BFS finds plan with shortest length
  - But what if cost of plan is not optimal? i.e. a longer plan may have smaller cost overall
- UCS almost same as BFS, but use priority queue instead of queue
  - Each node in queue ordered by cost to node
- BFS = UCS when costs are constant everywhere
- Very similar to Dijkstra's algorithm (but Dijkstra doesn't have a goal)

Uniform-cost search

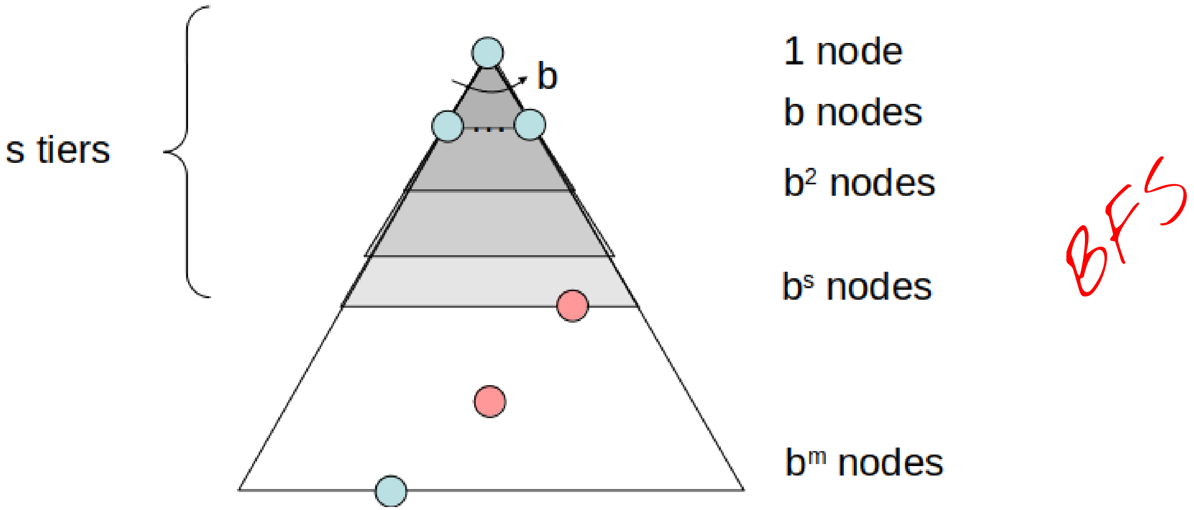
Priority queue instead of queue

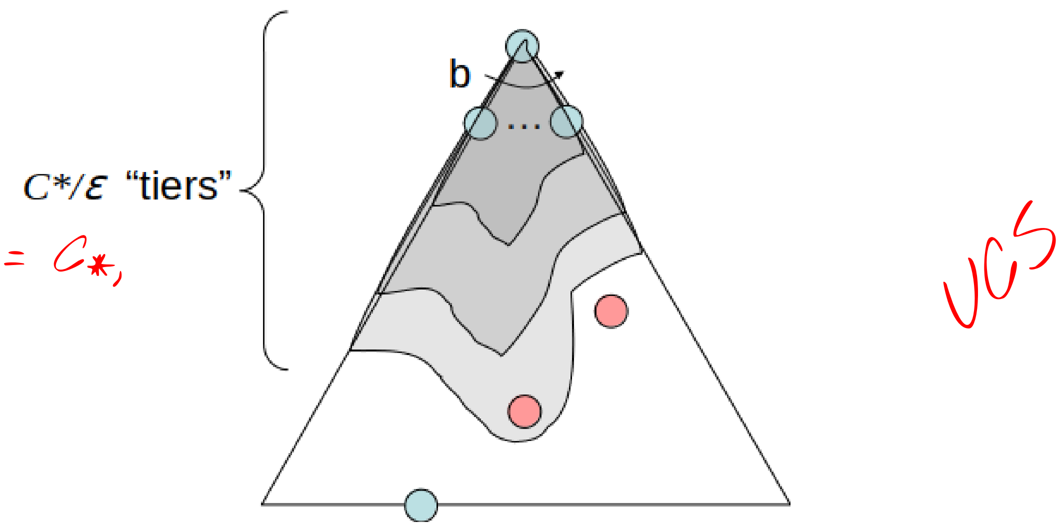
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
```

Extra check: if path to child shorter than previous one found, update it!

BFS vs UCS





Solution cost $= C^*$,
costs $\geq \varepsilon$

UCS properties

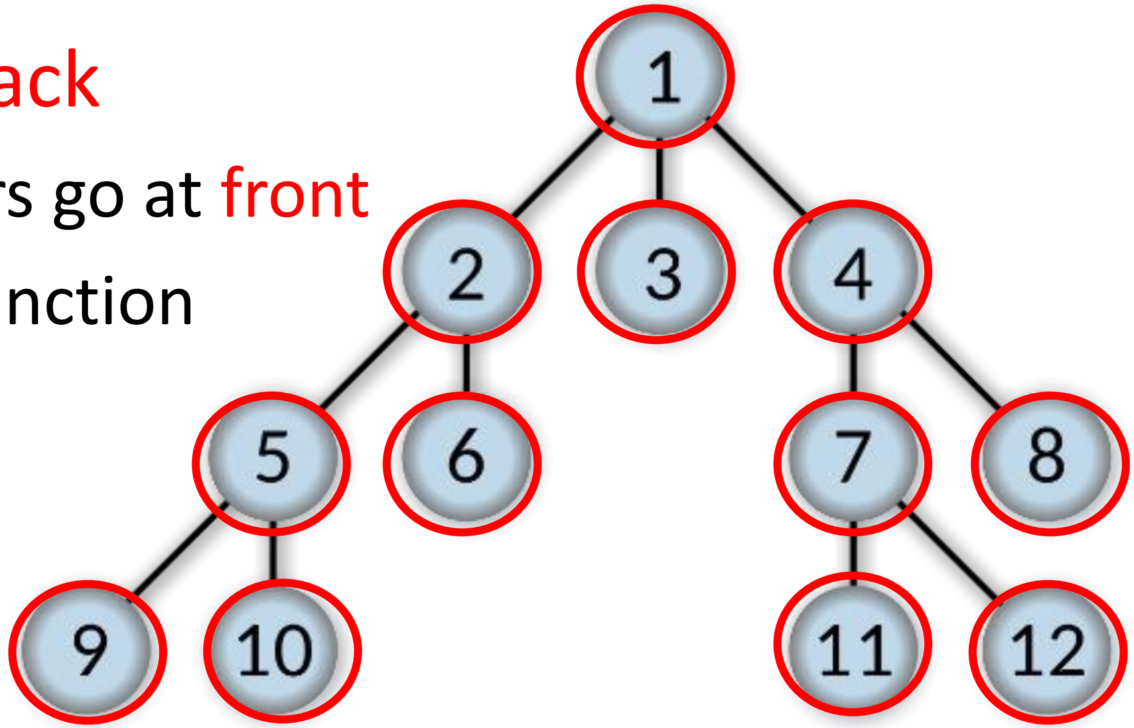
- Expands all nodes with cost less than cheapest solution
- If solution is costs C^* and each edge is ε, then effective depth is approx C^* / ε

- Complete?
  - Yes, if cost $\geq \varepsilon$ (positive costs) and best solution has finite cost
- Time?
  - $O(b^{ceiling(C^* / \varepsilon)})$
- Space?
  - Keeps all frontier nodes in memory: $O(b^{ceiling(C^* / \varepsilon)})$ Again ☹

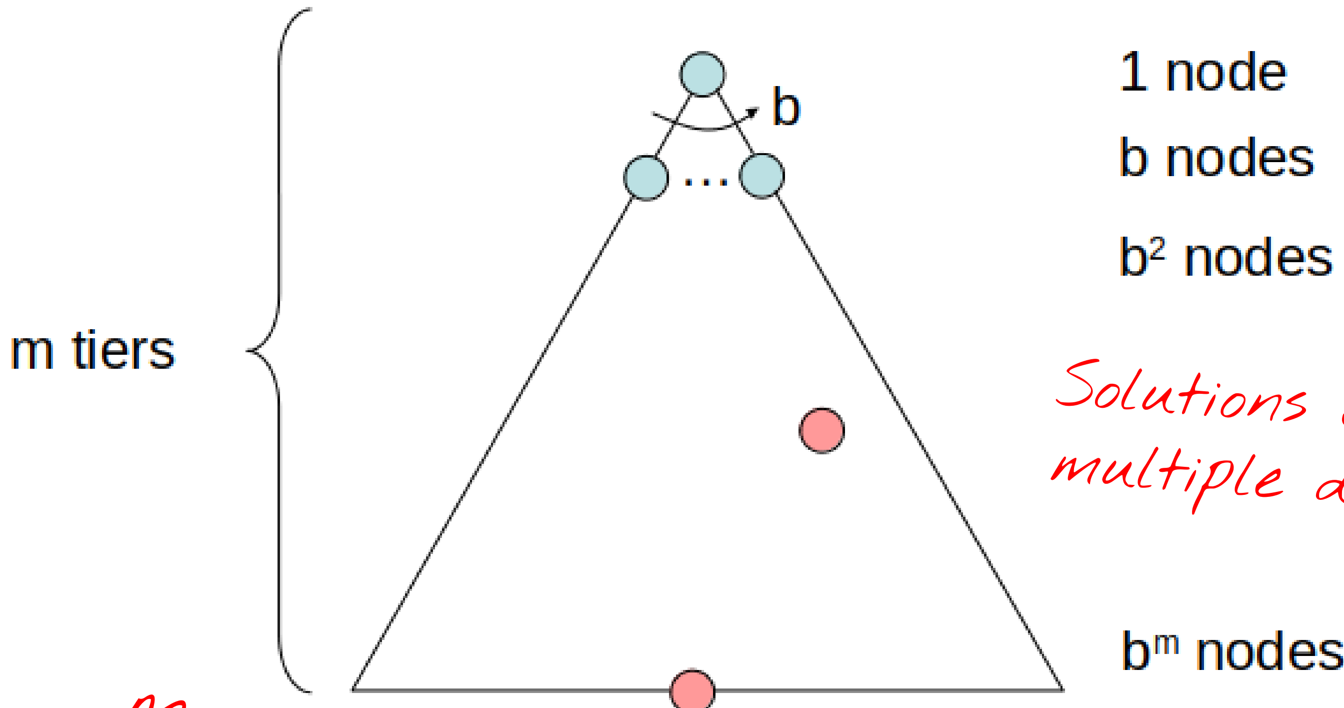
- Optimal:
  - Yes! Nodes expanded in increasing order of total cost

Depth-first search

- Expand deepest unexpanded node
- Frontier as a stack
  - New successors go at front
  - Or recursive function



DFS



Solutions exist at multiple depths

M is max solution depth

DFS properties

- We use special trick for visited states
  - Only remember states along path from root to current node
- Complete?
  - Yes, for finite spaces
- Time?
  - $O(b^m)$ - terrible if $m \gg d$
- Space?
  - Remember only path from root to current node (and unexplored siblings along path: $O(bm)$ Finally! ☺
- Optimal:
  - No, finds "leftmost" solution!

Depth-limited DFS

- Solution might be at finite depth, but if $m = \infty$,
  - DFS will never find it!
- Solution: just limit max depth to l
  - Time is now $O(b^l)$ and space is $O(bl)$
- But how to pick l? And what if $l < d$?
  - Could use domain knowledge e.g. if you know solution is at most k, pick $l = k$
  - Or use diameter: max shortest distance between any 2 states

Iterative deepening

- Instead, we can just try multiple depths!
- i.e. RUN DFS with $l = 1$
  - If solution, great! If not, run it again with $l = 2$
  - Keep going until you find solution!
- Combines the benefits of DFS + BFS

function ITERATIVE-DEEPENING-SEARCH($problem$) **returns** a solution, or failure
  for $depth = 0$ **to** ∞ **do**
    $result \leftarrow$ DEPTH-LIMITED-SEARCH($problem$, $depth$)
    if $result \neq$ cutoff **then return** $result$

- Seems wasteful!
  - But is it?

IDS Analysis

- If we ran depth-limited DFS to depth d

  $\mathrm{N_{DLS}} = b^0 + b^1 + b^2 + \ldots + b^d$

- If we run IDS $l = 0, 1, .., d$

  $\mathrm{N_{IDS}} = (d+1)b^0 + db^1 + (d-1)b^2 + \ldots + 2b^{d-1} + b^d$

- For $b = 10, d = 5$
  - $\mathrm{N_{DLS}} = 111\,111$
  - $\mathrm{N_{IDS}} = 123\,456$
  - 11% difference!

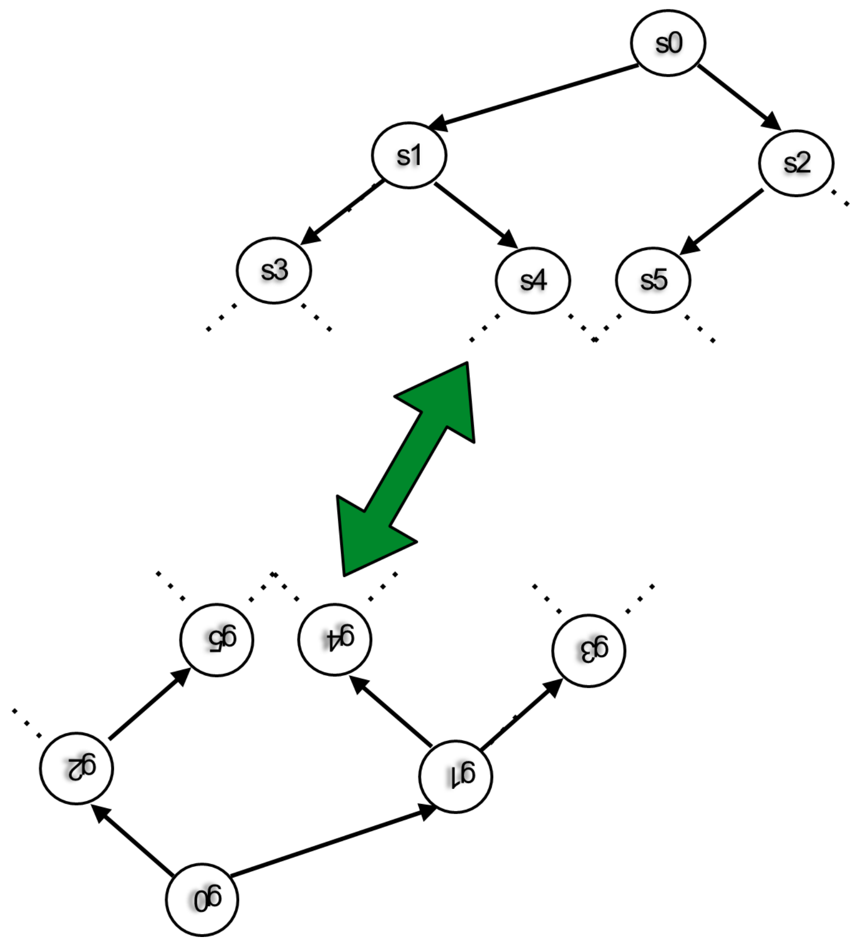
Most work happens in lowest levels

- Rule of thumb: if search space is large and depth of solution unknown, use IDS

IDS properties

- Complete?
  - Yes, it's like BFS
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ b^d = O(b^d)$
- Space? Like DFS!
  - $O(bd)$
- Optimal:
  - Only if step costs are same everywhere (like BFS)

Bidirectional Search



Bidirectional Search

- Run searches forward and backward until intersection
  - $O(b^{d/2}) + O(b^{d/2}) \ll O(b^d)$
- Extra requirements:
  - Invert action rules (sometimes hard)
  - Not always unique
  - Need single solution
- When to stop:
  - Candidate solution when frontier nodes intersect
  - First solution may not be optimal – must check possible shortcuts

Summary of methods

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

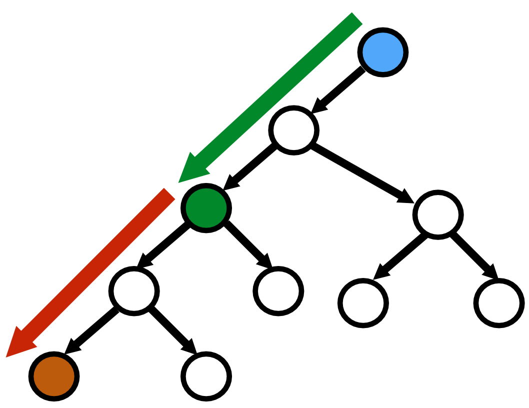
Now

- What if we know something about the problem?
  - How can we incorporate knowledge?
  - Task-specific expansion strategy?

What does UCS suggest?

- Cost-so-far: how much it costs to get to a node
  - UCS says: cheapest node first!

- What remains?
  - Total-cost = Cost-so-far + Cost-to-go



From UCS to heuristics

- Recall: UCS is like BFS with priority queue
- Nodes ordered by priority from smallest to largest
  - Priority is cost estimate $f(n)$ Evaluation function
- In UCS, $f(n) = g(n)$, where g is cost of reaching n from root
- Heuristic function $h(n) =$ estimate of cost from n to goal
- g is backward cost (start to node), h is (estimated) forward cost (node to goal)

Greedy best-first search

- UCS is $f(n) = g(n)$
- GBFS is $f(n) = h(n)$
  - Order nodes by estimated cost to goal

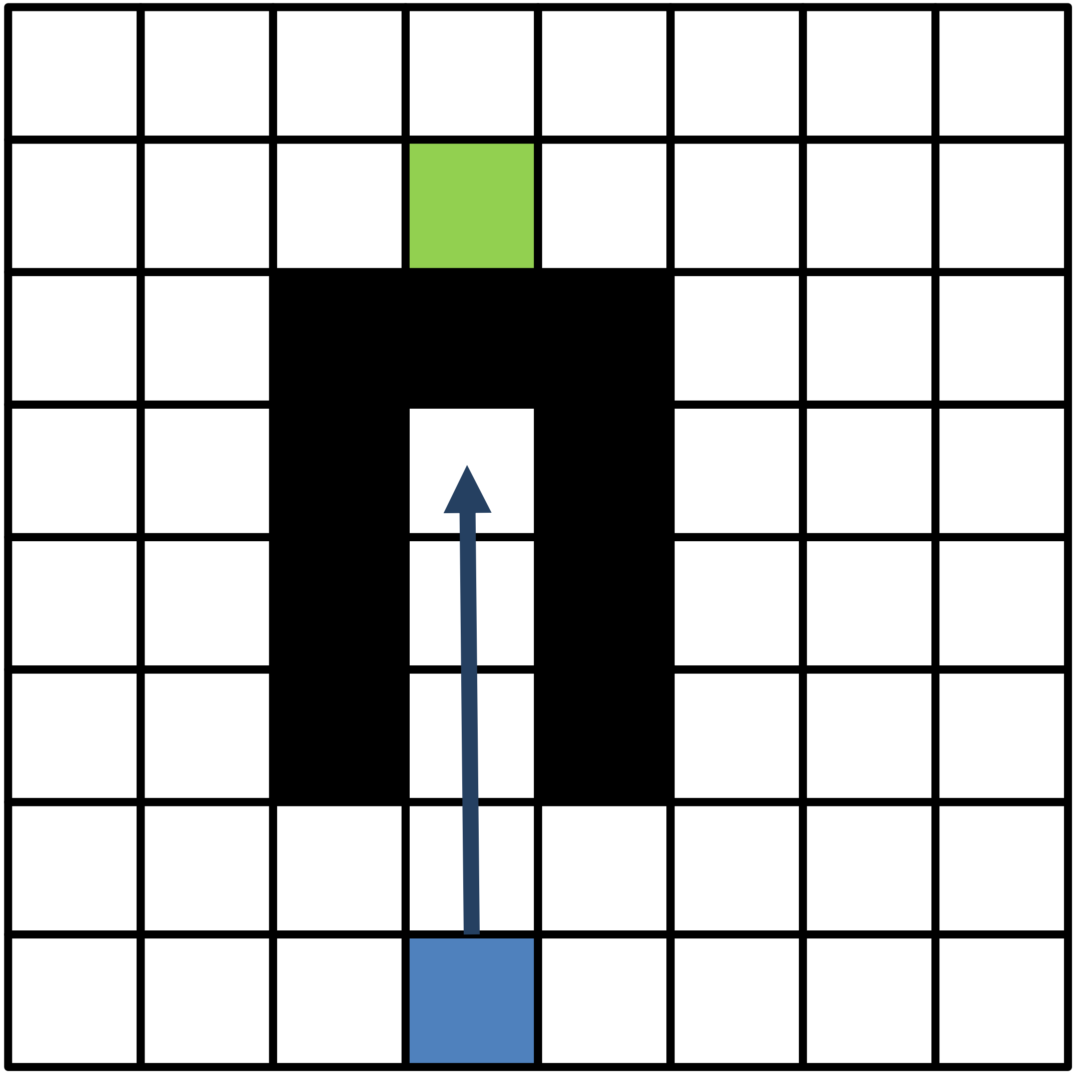
- Where does this estimate come from?
  - Domain knowledge!

- e.g. In navigating task with location x

$$h(x) = |x - g|_2$$ Euclidean distance

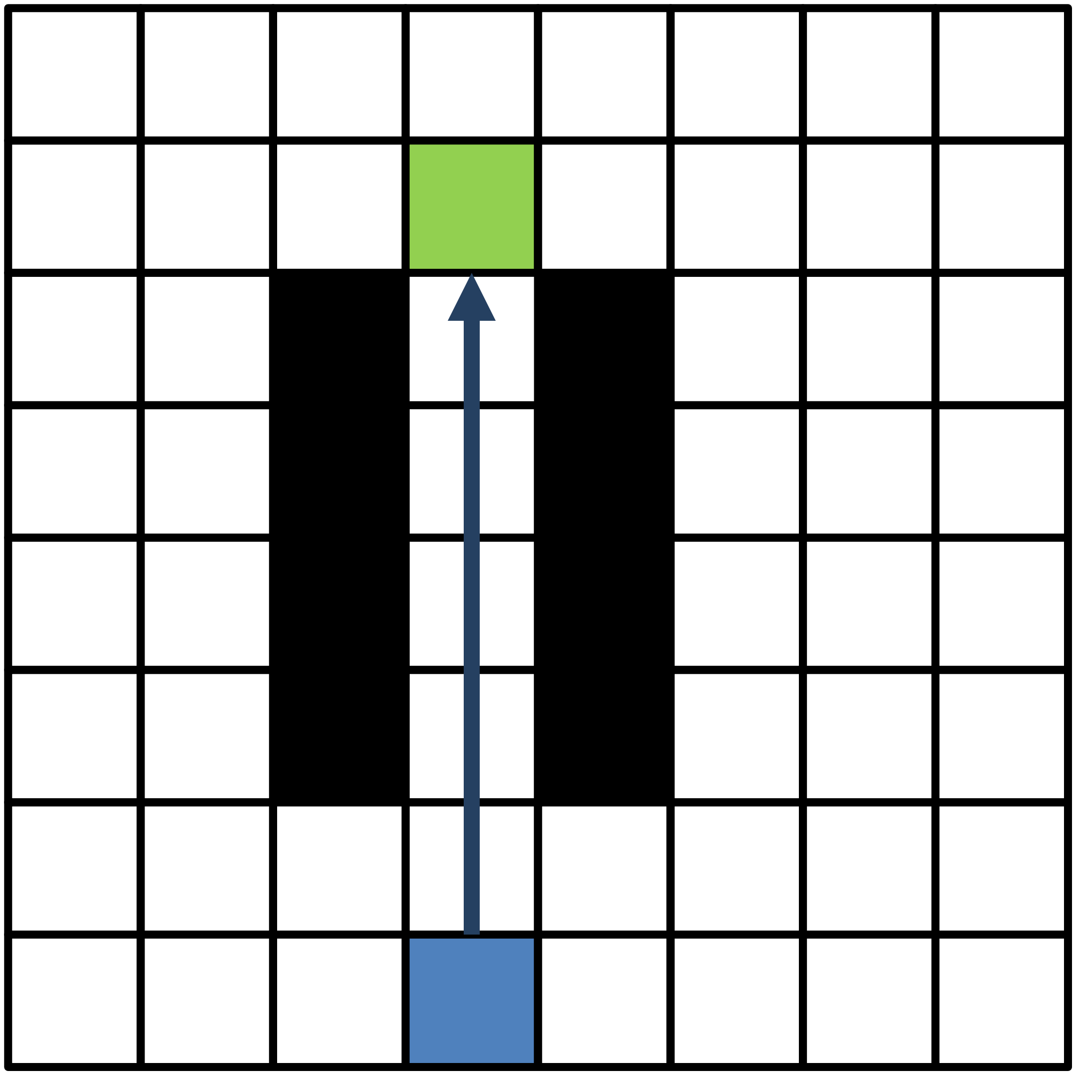
Example

If allowing repeated nodes, will be stuck! (incomplete)



Euclidean distance heuristic

Example



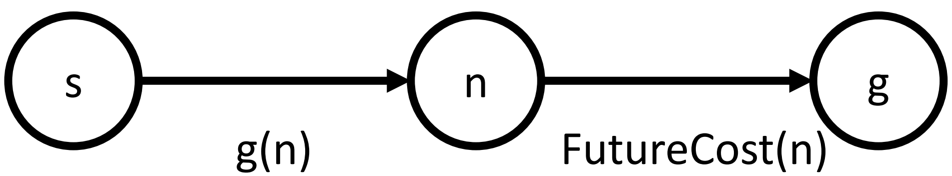
Euclidean
distance
heuristic

GBFS properties

- Incomplete for tree search (i.e. repeated nodes allowed)
  - Complete for graph search in finite space
- Space and time complexity are both $O(b^m)$
  - For max search depth
- But can be reduced substantially depending on heuristic and problem

A* search

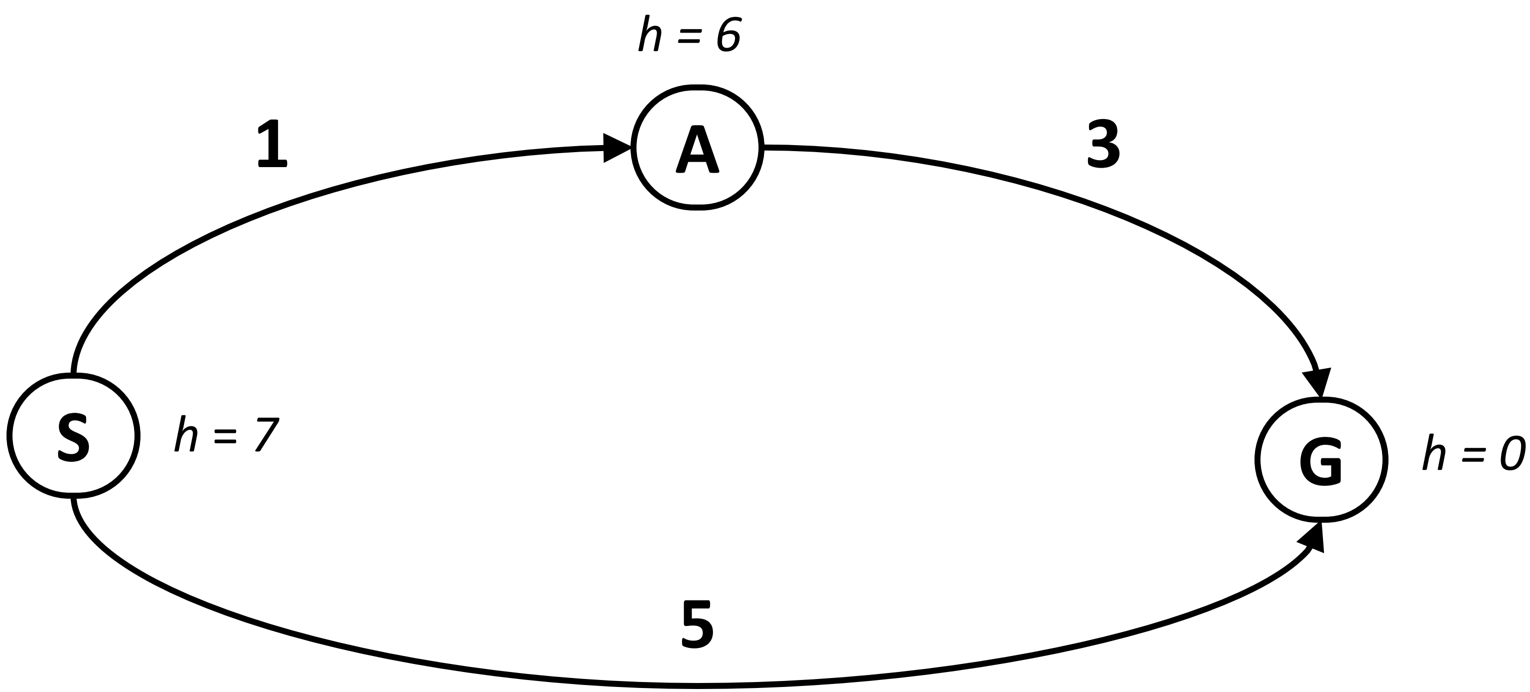
- UCS: $f(n) = g(n)$
- GBFS: $f(n) = h(n)$



- Ideally: explore in order of $g(n) + CostToGo(n)$
- A*: explore in order $f(n) = g(n) + h(n)$
  - h is estimate of future cost
- Implement A* as UCS with different priority!

Optimality of A*

- Depends on heuristic



- Fails! Bad cost goal < estimated good cost

Heuristics in A*

- h admissible if $h(n) \leq TrueFutureCost(n)$
  - We never overestimate the cost!
  - $f(n) = g(n) + h(n)$ means we never overestimate cost from start to goal through n
  - Optimistic: estimates cost as less than it actually is

- h is consistent if $h(n) \leq c(n, a, n') + h(n')$
  - All consistent heuristics are admissible

Triangle inequality

- A* tree search is optimal if h is admissible
- A* graph search is optimal if h is consistent

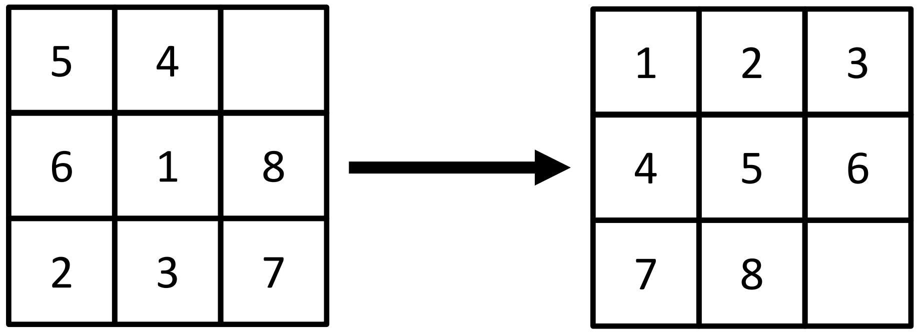
Properties of A*

- A* is complete, optimal
- A* is optimally efficient w.r.t. heuristic
  - No other algorithm will expand fewer nodes
- Time complexity is $O\left(b^{\epsilon d}\right)$
  - ϵ is relative error of heuristic, d is solution depth
- Alternate view, $O((b^*)^d)$ where b^* is effective branching factor
  - A* doesn't need to consider certain nodes (prunes) and so branching factor is reduced!
- Since exponential, memory is biggest issue
  - Can do iterative deepening equivalent (IDA*) Like BFS

Creating heuristics

- Main challenge for A*: how to make good heuristics that are admissible/consistent?
- For navigation, straight-line path is good
    - Can never be faster than that!
- Could learn heuristics from data (e.g. machine learning/precomputed database lookups)
- Use relaxations – solve an easier, less constrained version of a problem
    – E.g. Relaxed version of a maze → remove all the walls!

8-puzzle relaxation



- Relaxation 1: allow tiles to be placed anywhere directly
  - i.e. heuristic is number of misplaced tiles
  - Clearly underestimates true cost of moving tiles
- Relaxation 2: allow tiles to be slid around, ignoring other tiles
  - i.e. heuristic is sum of distances of each tile to its final location

Effect of heuristics

	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
Relax 1	13	39	227
Relax 2	12	25	73

- Tradeoff between quality of estimate and work per node
  - Closer heuristic is to true cost, fewer nodes are expanded…
  - But more work to compute heuristic per node!