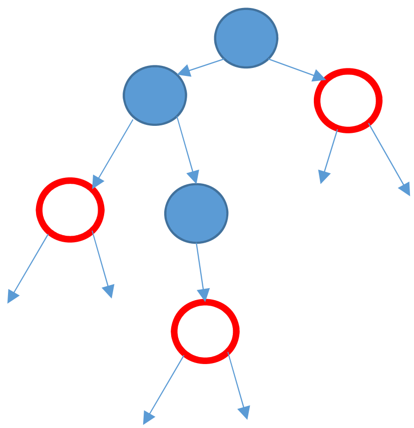
CS440/ECE448 Lecture 5: Search Order

Slides by Svetlana Lazebnik, 9/2016

Revised by Mark Hasegawa-Johnson, 1/2018

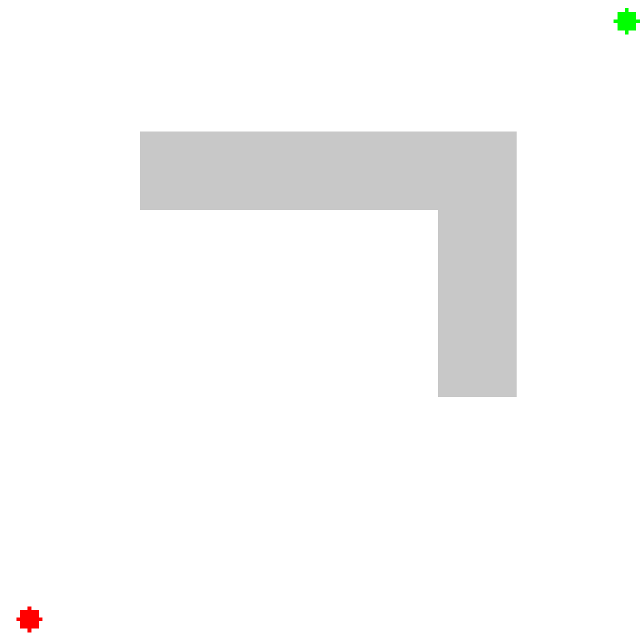


Prioritized Search

- Review: Tree Search vs. Dijkstra’s Algorithm
- Criteria for evaluating a search algorithm: completeness, optimality, computational cost, storage cost
- Search algorithms without side information: BFS, DFS, IDS, UCS
- Search algorithms with side information: GBFS vs. A*
  - Heuristics to guide search
  - Greedy best-first search
  - A* search
  - Admissible vs. Consistent heuristics
  - Designing heuristics: Relaxed problem, Sub-problem, Dominance, Max

Dijkstra's Shortest Path Algorithm

- Initialize:
  - d_{nl} = distance from n to l
  - $V_n = \infty$ for all vertices n
  - Unvisited = $\{all\ nodes\ but\ start\}$
  - k = Start Node
- While Goal ∈ Unvisited
  - For n ∈ Neighbor(k)
    - $V_n = \min(V_n, V_k + d_{nk})$
  - k ← $\underset{l \in Unvisited}{\operatorname{argmin}} V_l$



Dijkstra Algorithm Complexity

- Suppose there are V nodes, E edges
- Dijkstra's algorithm computational complexity
  - $V_n = \min(V_n, V_n + d_{nk})$: O{E} operations
  - k ← $\underset{l \in Unvisited}{\operatorname{argmin}} V_l$: O{|V|log|V|) operations
  - Total: O{|E|+|V|log|V|}
- Dijkstra storage space: O{|V|+|E|}

Tree Search Algorithm

- Initialize: Frontier = { startnode }
- While Frontier $\neq \emptyset$
  - Choose a node from the frontier
    - How do you choose a node?
    - Answer: using a search strategy – topic of this lecture
  - If it's the end node: terminate
  - If not, expand it: put its neighbors into the frontier

- Visited list: assume there isn't one, for now…

Tree Search Algorithm

- Computational complexity = $O\{MT_E + NT_Q\}$,
  - M = # nodes expanded, T_E=cost of choosing a node to expand
  - N = # nodes placed on frontier, T_Q=cost of doing so
  - If M<<V, N<<E then it's cheaper than Dijkstra's algorithm
  - If M=∞…

Prioritized Search

- Review: Tree Search vs. Dijkstra's Algorithm
- Criteria for evaluating a search algorithm: completeness, optimality, computational cost, storage cost
- Search algorithms without side information: BFS, DFS, IDS, UCS
- Search algorithms with side information: GBFS vs. A*
  - Heuristics to guide search
  - Greedy best-first search
  - A* search
  - Admissible vs. Consistent heuristics
  - Designing heuristics: Relaxed problem, Sub-problem, Dominance, Max

Analysis of search strategies

- Strategies are evaluated along the following criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Optimality:** does it always find a least-cost solution?
  - **Time complexity:** number of nodes generated
  - **Space complexity:** maximum number of nodes in memory
- Time and space complexity are measured in terms of
  - ***b**:* maximum branching factor of the search tree
  - ***d**:* depth of the optimal solution
  - ***m***: maximum length of any path in the state space (may be infinite)

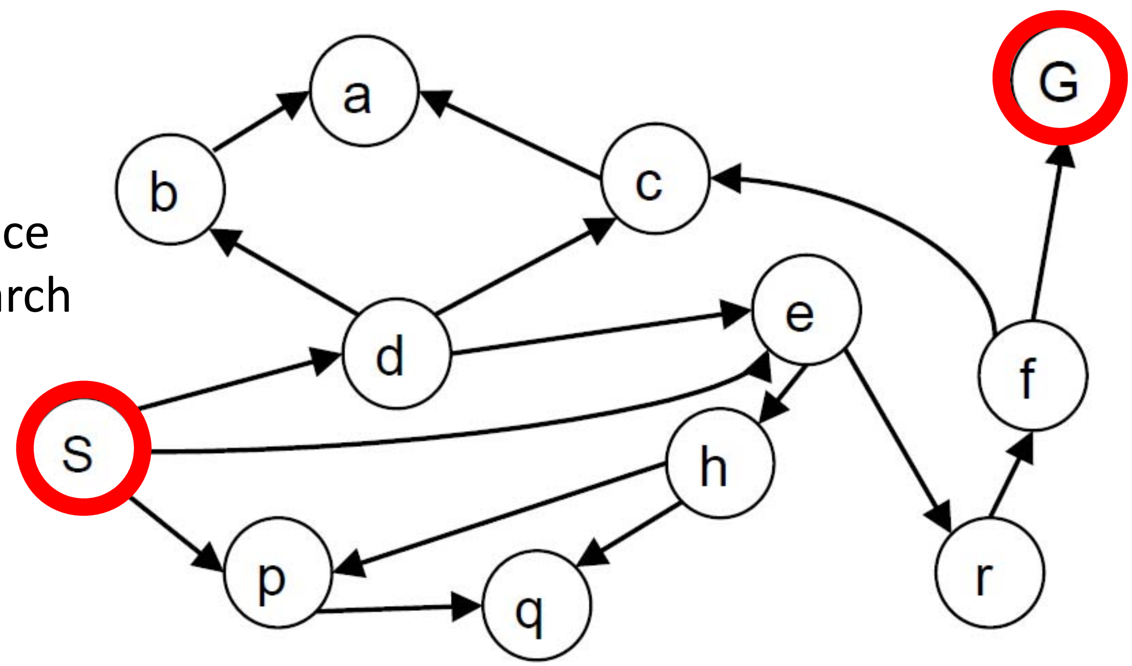
Prioritized Search

- Review: Tree Search vs. Dijkstra's Algorithm
- Criteria for evaluating a search algorithm: completeness, optimality, computational cost, storage cost
- Search algorithms without side information: BFS, DFS, IDS, UCS
- Search algorithms with side information: GBFS vs. A*
    - Heuristics to guide search
    - Greedy best-first search
    - A* search
    - Admissible vs. Consistent heuristics
    - Designing heuristics: Relaxed problem, Sub-problem, Dominance, Max

Depth-first search

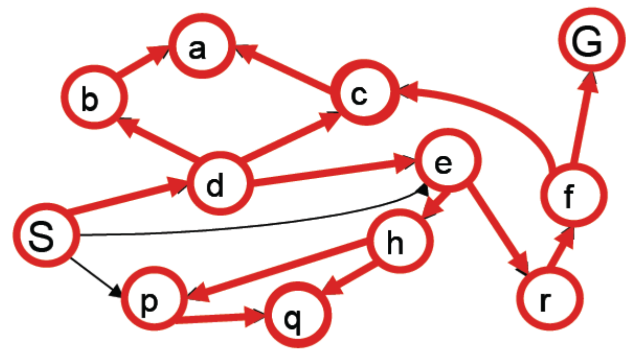
- Expand deepest unexpanded node
- Implementation: *frontier* is a LIFO stack

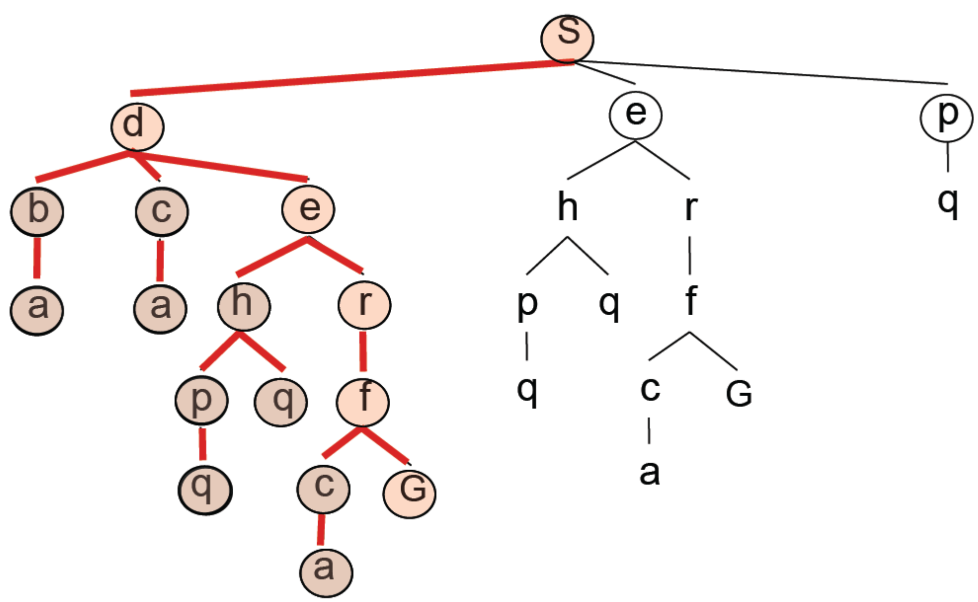
Example state space graph for a tiny search problem



Depth-first search

Expansion order:
(s,d,b,a,
c,a,
e,h,p,q,
q,
r,f,c,a,
G)





Properties of depth-first search

- **Complete? (always finds a solution if one exists?)**

  Fails in infinite-depth spaces, spaces with loops

  Modify to avoid repeated states along path

  → complete in finite spaces

- **Optimal? (always finds an optimal solution?)**

  No – returns the first solution it finds

- **Time? (how long does it take, in terms of b, d, m?)**

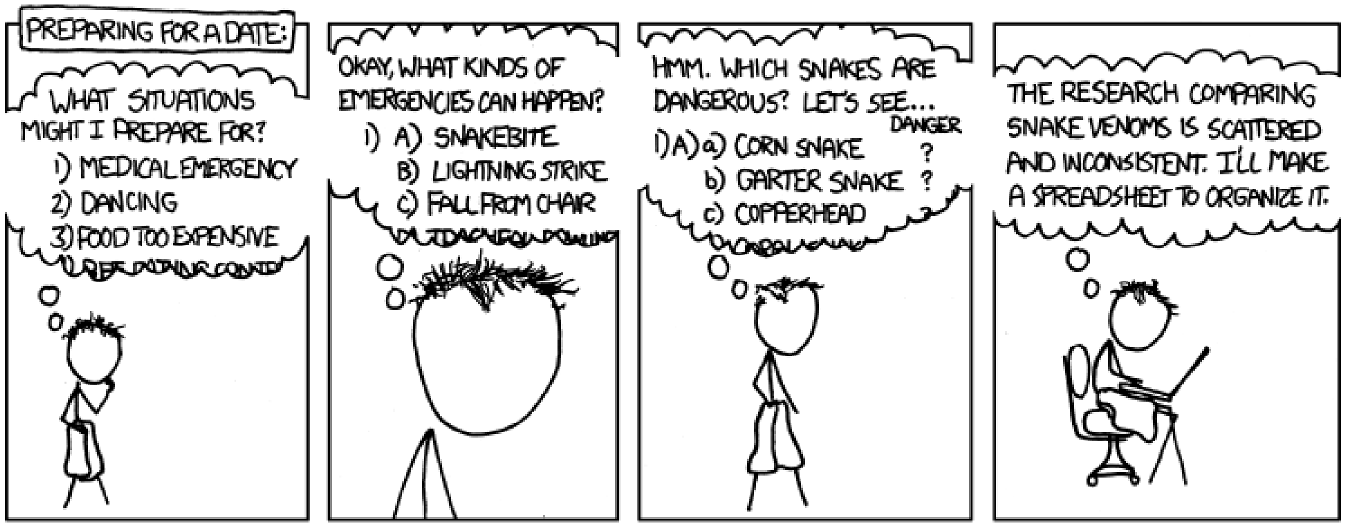
  Could be the time to reach a solution at maximum depth *m*: $O(b^m)$

  Terrible if *m* is much larger than *d*

  But if there are lots of solutions, may be much faster than BFS

- **Space? (how much storage space, in terms of b, d, m?)**

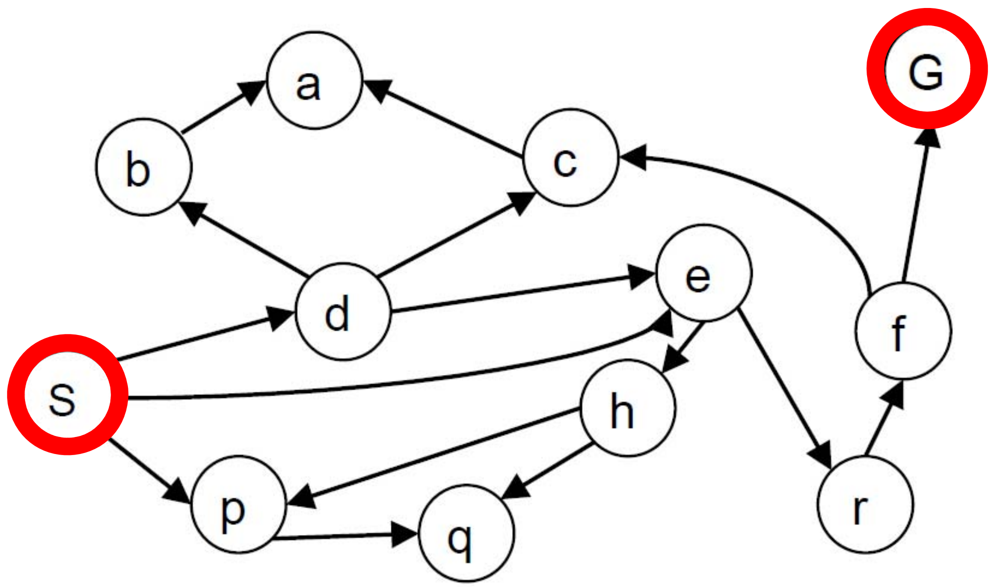
  $O(bm)$, i.e., linear space!



http://xkcd.com/761/

Breadth-first search

- Expand shallowest unexpanded node
- Implementation: *frontier* is a FIFO queue



Example from P. Abbeel and D. Klein

Breadth-first search

Expansion order:

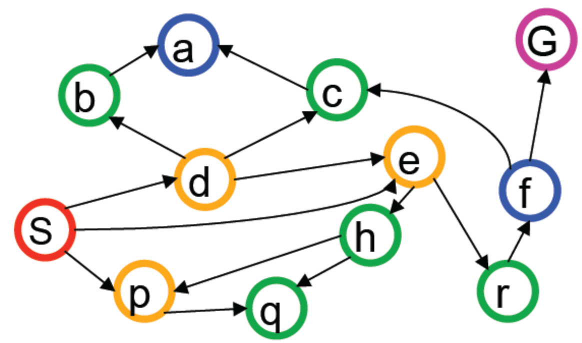
(s,

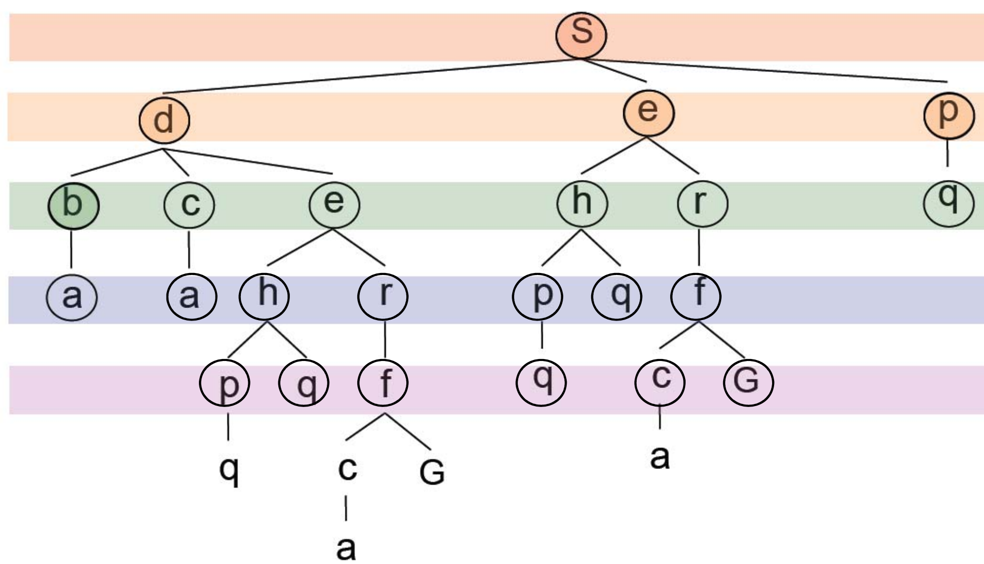
d,e,p,

b,c,e,h,r,q,

a,a,h,r,p,q,f,

p,q,f,q,c,G)





Properties of breadth-first search

- **Complete?**
  Yes (if branching factor *b* is finite).
  Even w/o repeated-state checking, it still works.
- **Optimal?**
  Yes – if cost = 1 per step (uniform cost search will fix this)
- **Time?**
  Number of nodes in a *b*-ary tree of depth *d*: $O(b^d)$
  (*d* is the depth of the optimal solution)
- **Space?**
  $O(b^d)$

- Space is the bigger problem (more than time)

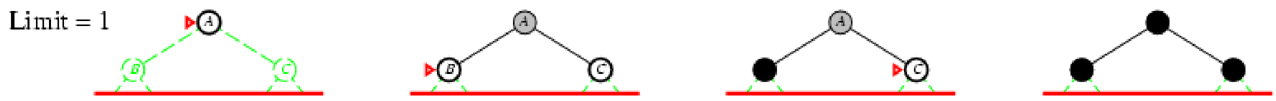
Iterative deepening search

- Use DFS as a subroutine
  1. Check the root
  2. Do a DFS searching for a path of length 1
  3. If there is no path of length 1, do a DFS searching for a path of length 2
  4. If there is no path of length 2, do a DFS searching for a path of length 3…

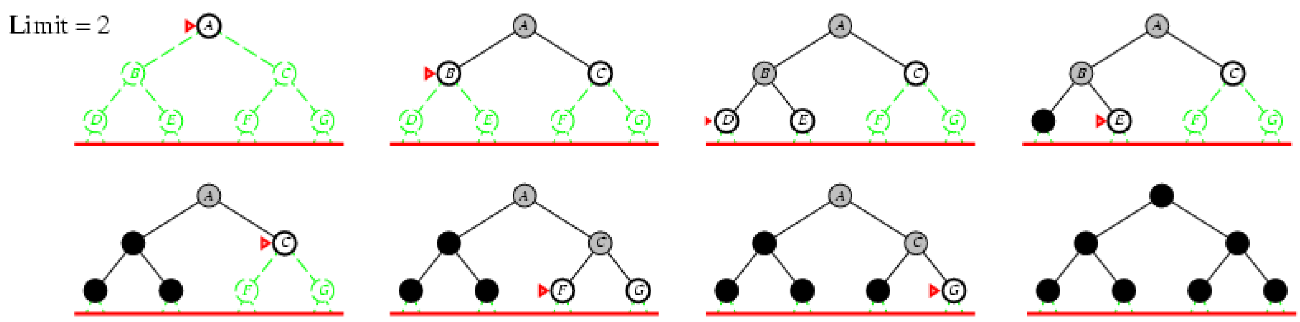
Iterative deepening search

Limit = 0

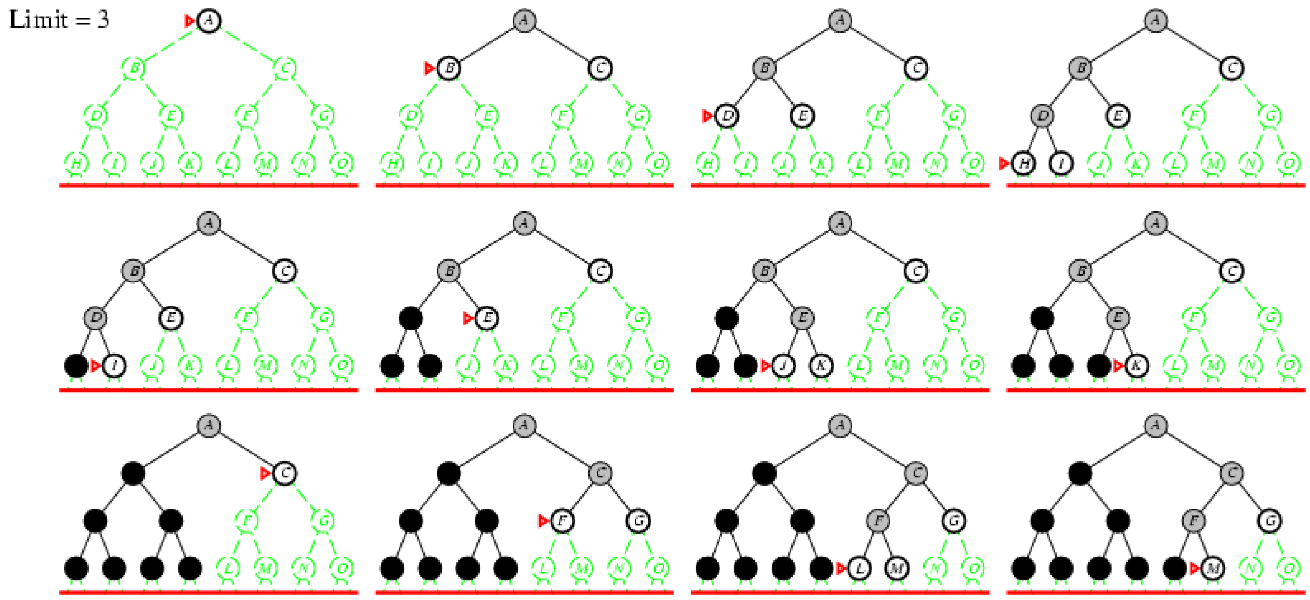
Iterative deepening search



Iterative deepening search



Iterative deepening search



Properties of iterative deepening search

- **Complete?**

  Yes – same completeness properties as BFS

- **Optimal?**

  Yes, if step cost = 1 – same as BFS

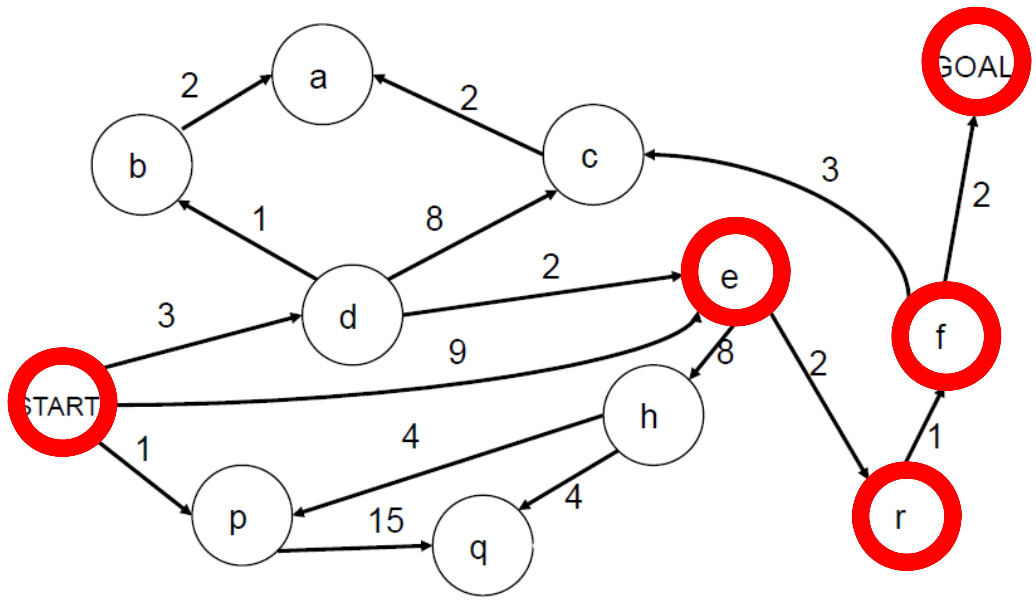
- **Time?**

  $1 + b + b^2 + \cdots + b^{d-1} + b^d = O\{b^d\}$ – same order as BFS! Increase in complexity is a factor of about (b+1)/b

- **Space?**

  $O(bd)$ – same as DFS!

Search with varying step costs

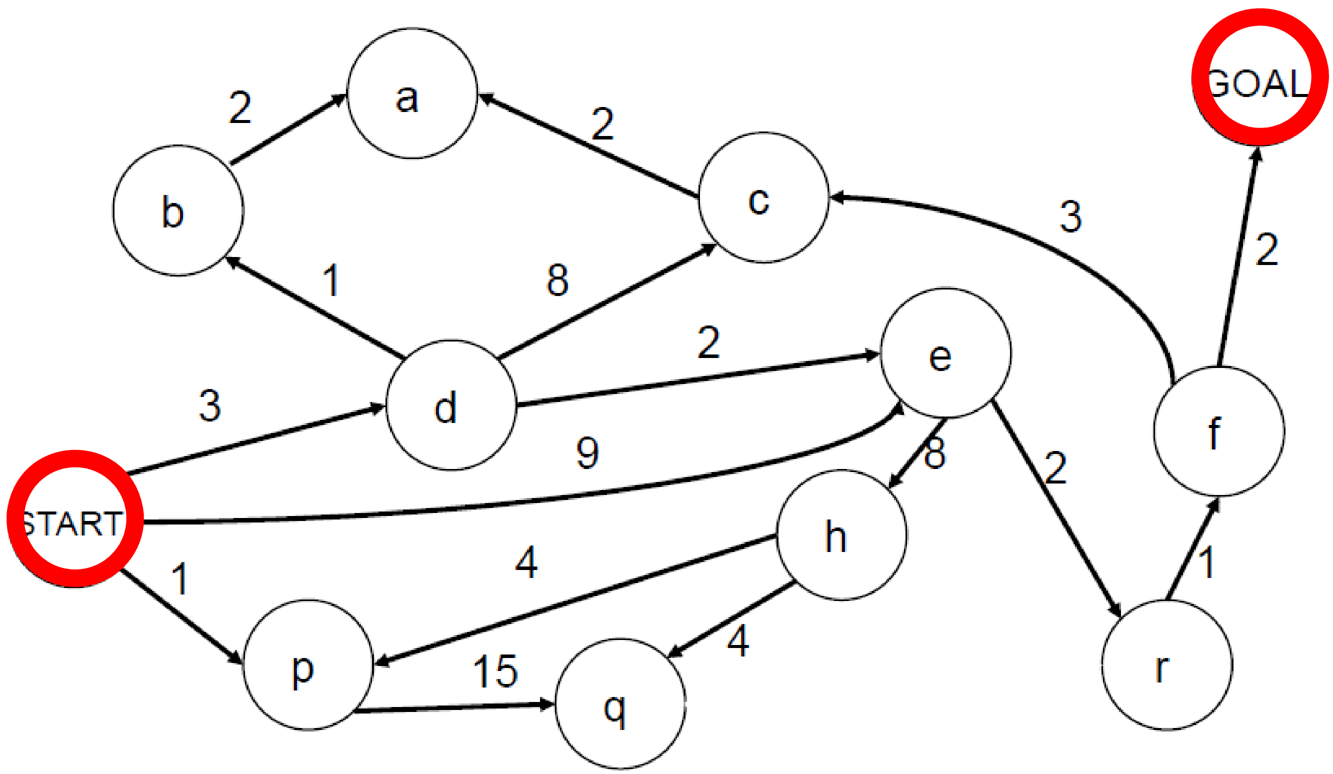


- BFS finds the path with the fewest steps, but does not always find the cheapest path

Uniform-cost search

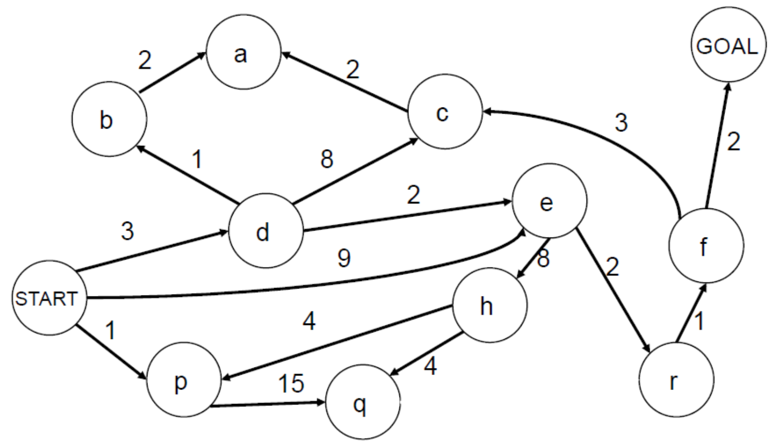
- For each frontier node, save the total cost of the path from the initial state to that node
- Expand the frontier node with the lowest path cost
- Implementation: *frontier* is a priority queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Equivalent to Dijkstra's algorithm, if Dijkstra's algorithm is modified so that a node's value is computed only when it becomes less than infinity

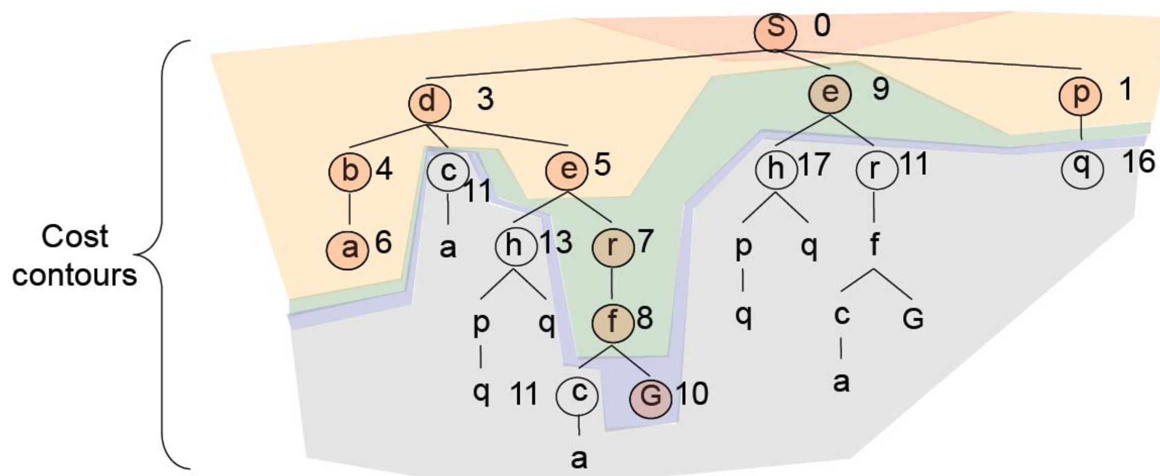
Uniform-cost search example



Uniform-cost search example

Expansion order:
(s,p(1),
d(3),b(4),
e(5),r(7),f(8)
e(9),
G(10))





Properties of uniform-cost search

- **Complete?**

  Yes, if step cost is greater than some positive constant ε (we don't want infinite sequences of steps that have a finite total cost)

- **Optimal?**

  Yes

Prioritized Search

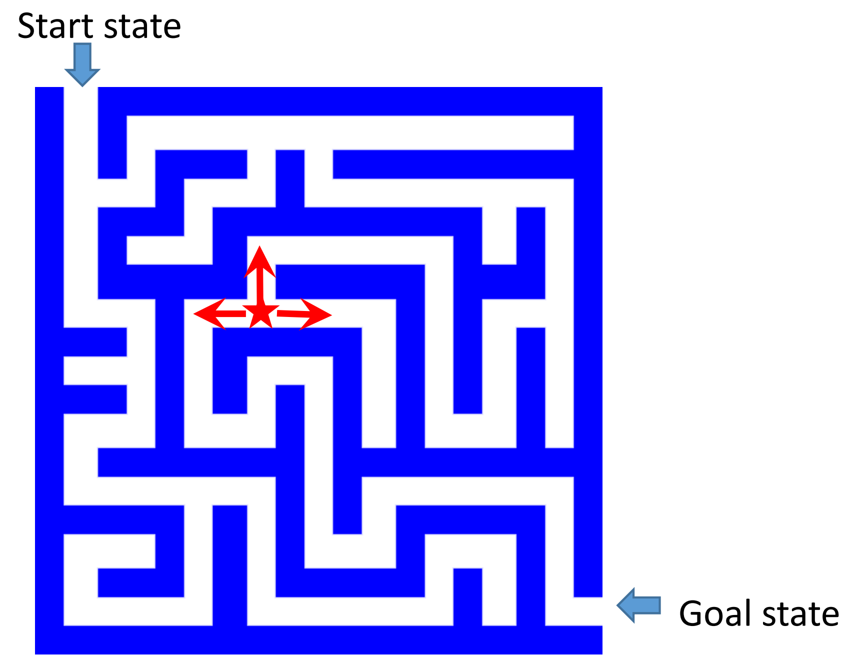
- Review: Tree Search vs. Dijkstra's Algorithm
- Criteria for evaluating a search algorithm: completeness, optimality, computational cost, storage cost
- Search algorithms without side information: BFS, DFS, IDS, UCS
- Search algorithms with side information: GBFS vs. A*
  - Heuristics to guide search
  - Greedy best-first search
  - A* search
  - Admissible vs. Consistent heuristics
  - Designing heuristics: Relaxed problem, Sub-problem, Dominance, Max

Informed search strategies

- Idea: give the algorithm “hints” about the desirability of different states
  - Use an ***evaluation function*** to rank nodes and select the most promising one for expansion
- Greedy best-first search
- A* search

Heuristic function

- ★ = node we're currently expanding
- **Most obvious thing to do: go toward the goal, i.e., →**



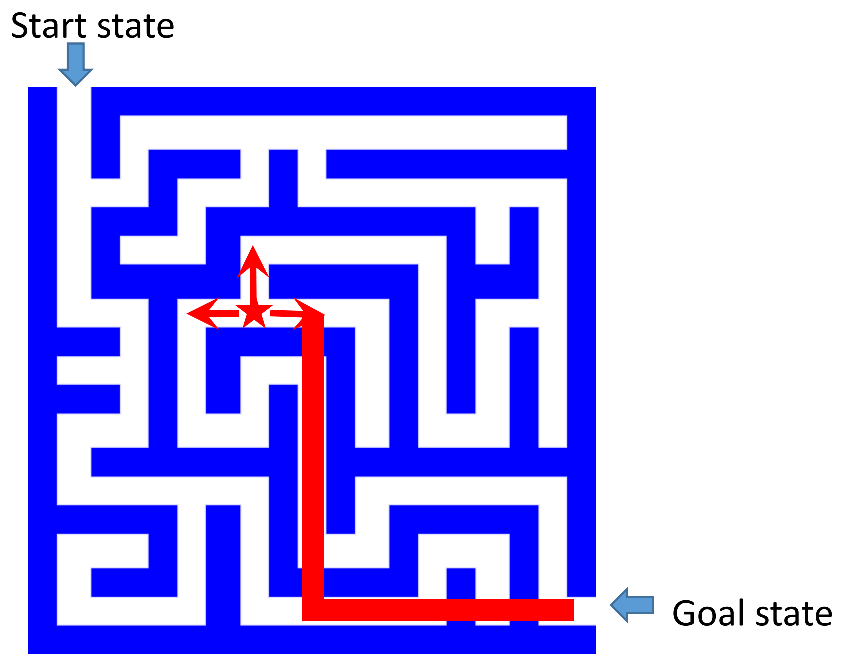
Heuristic function

- h[n] = estimate of the distance from node n to goal
- Requirements:
  - Very fast to compute
  - Approximate true cost to goal? Under-estimate?
- Example: Manhattan distance

$$h[n] = |x_n - x_G| + |y_n - y_G|$$

where (x_n, y_n) = location of node n

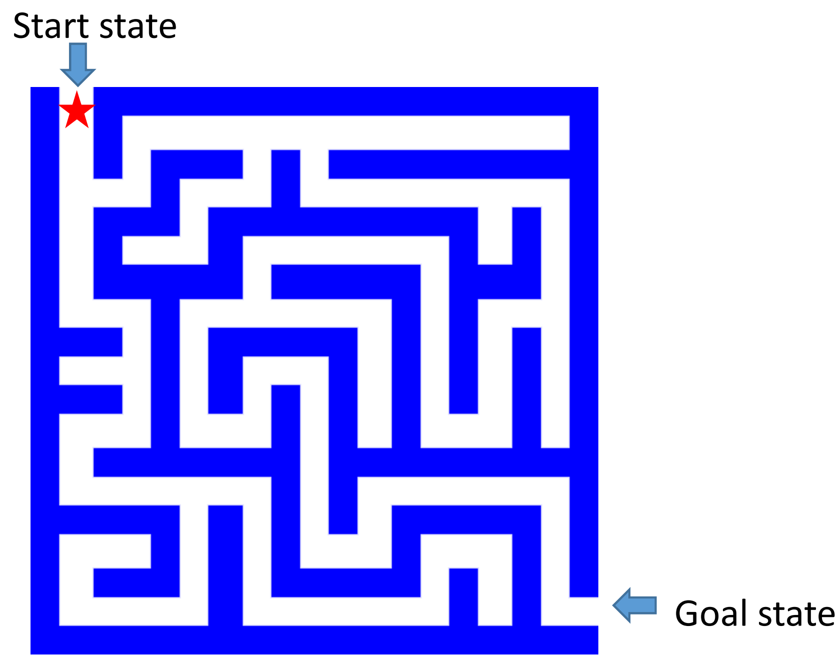
(x_G, y_G) = location of goal



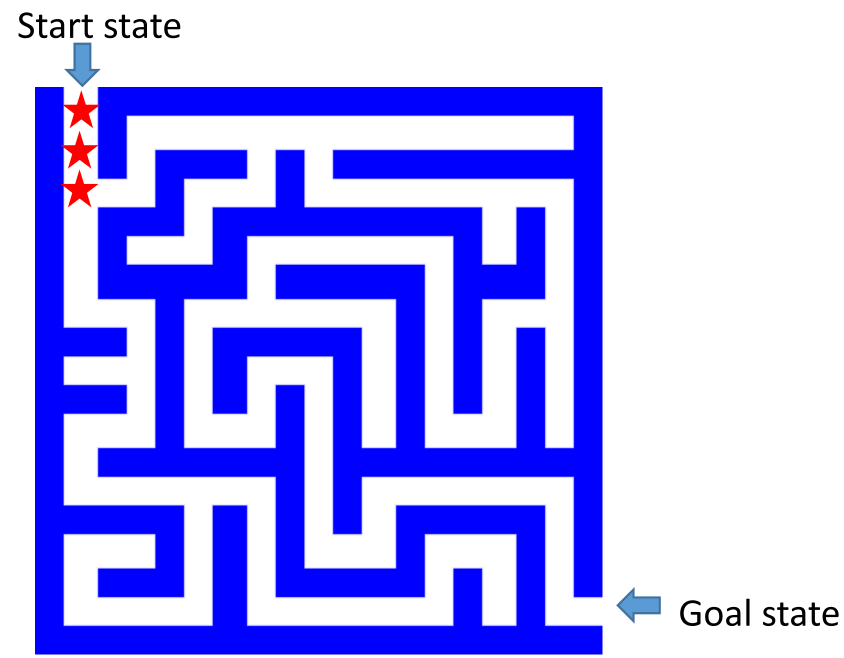
Greedy best-first search

Expand the node that has the lowest value of the heuristic function $h(n)$

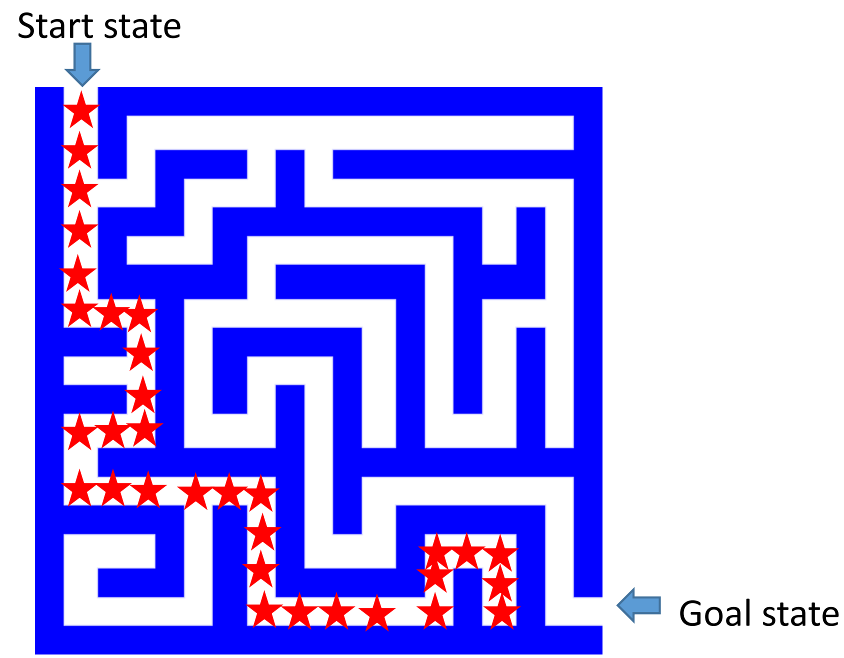
Greedy best-first search



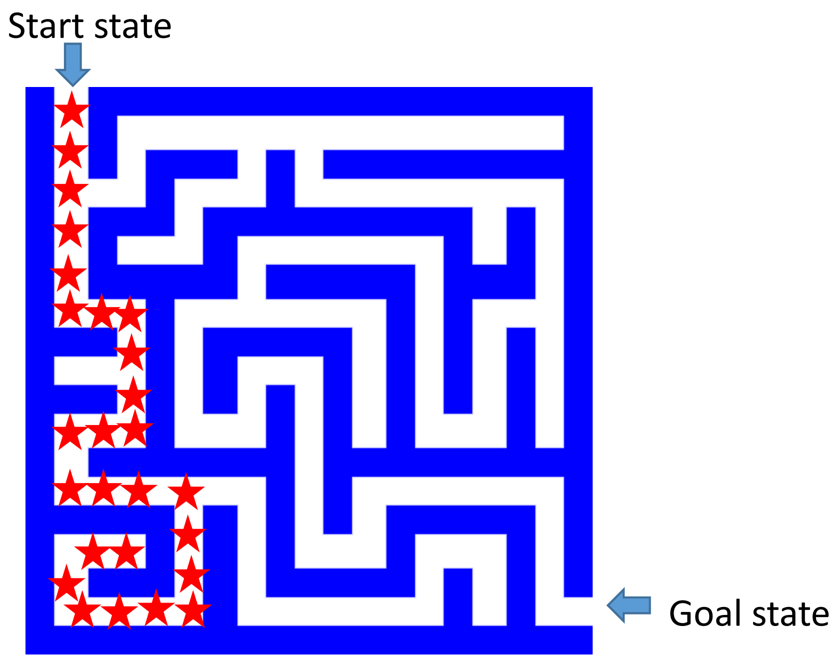
Greedy best-first search



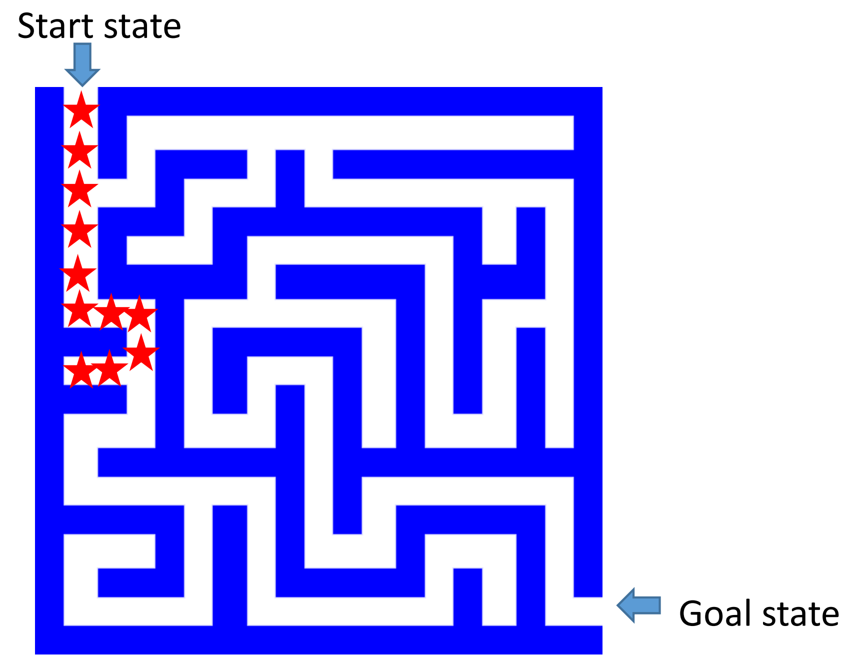
Greedy best-first search



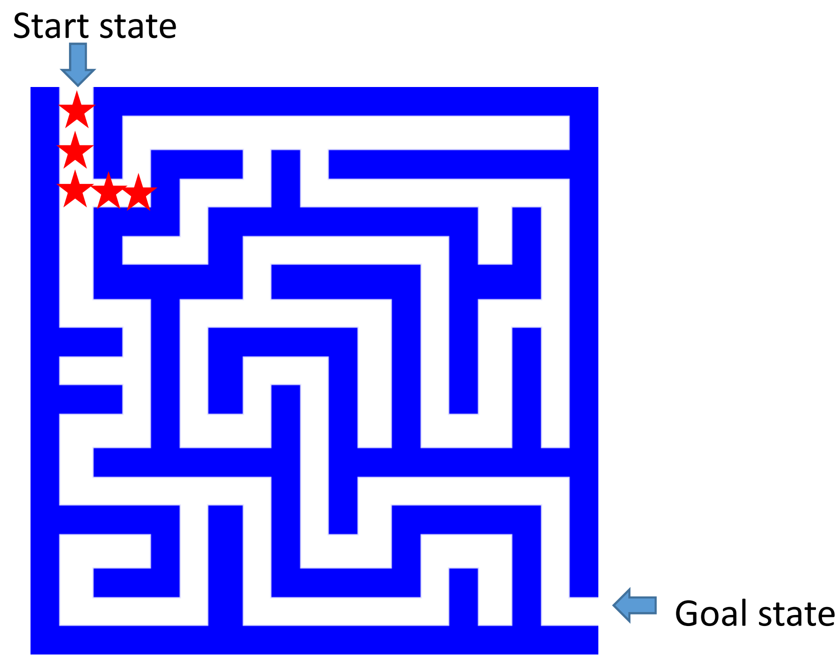
Greedy best-first search



Greedy best-first search



Greedy best-first search



Properties of greedy best-first search

- **Complete?**

  No – can get stuck in loops

- **Optimal?**

  No

- **Time?**

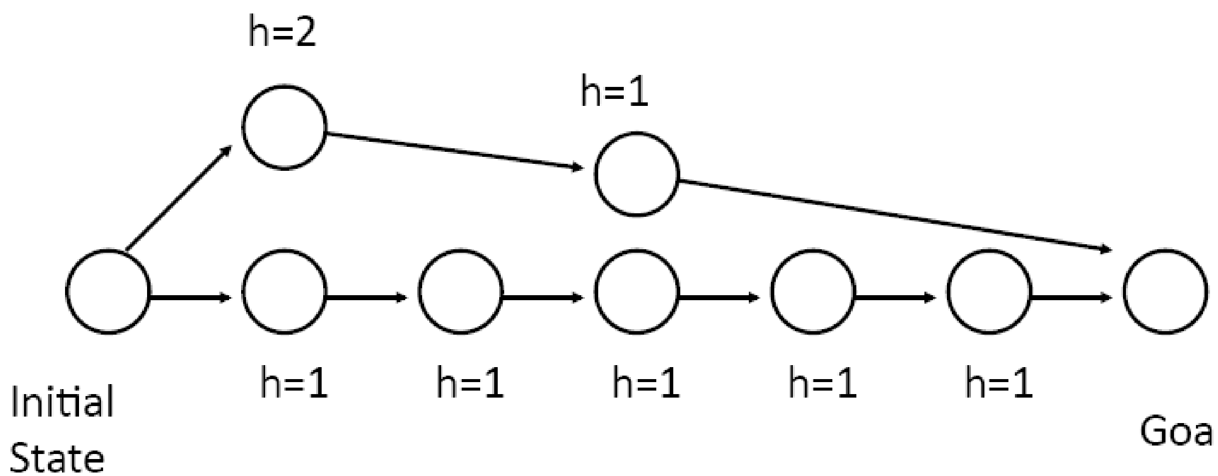
  Worst case: $O(b^m)$

  Can be much better with a good heuristic

- **Space?**

  Worst case: $O(b^m)$

How can we fix the greedy problem?



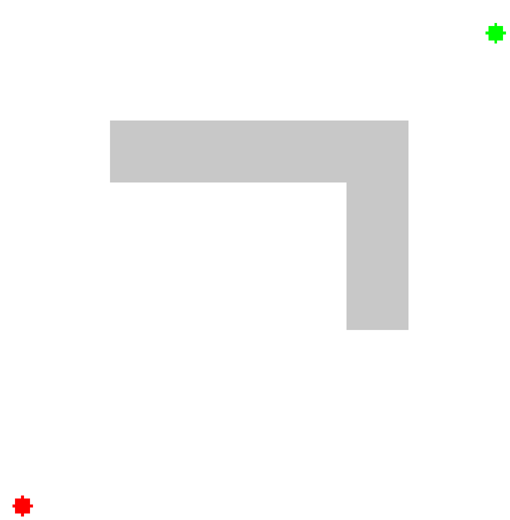
A^* search

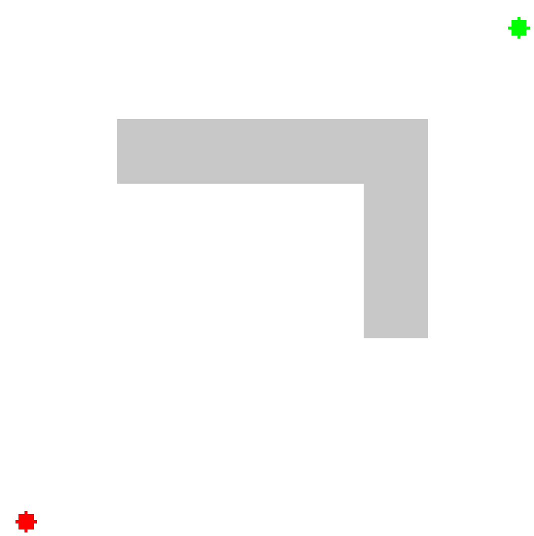
- Idea: avoid expanding paths that are already expensive
- The **evaluation function** $f(n)$ is the estimated total cost of the path through node n to the goal:

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

$g(n)$: cost so far to reach n (path cost)
$h(n)$: estimated cost from n to goal (heuristic)

BFS vs. A* search





Source: [Wikipedia]

Admissible heuristics

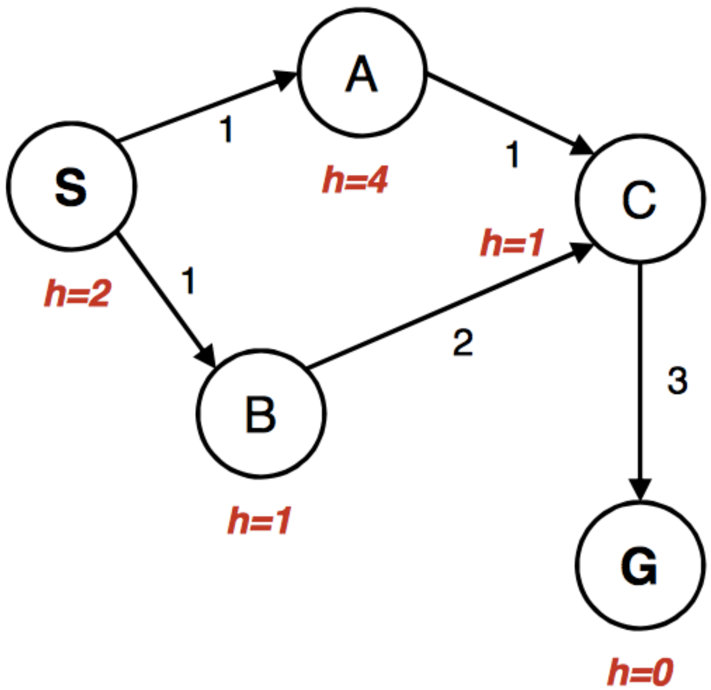
- An admissible heuristic never overestimates the cost to reach the goal
- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n
- Example:
  - straight line distance never overestimates the actual road distance
  - Manhattan distance never overestimates actual road distance if all roads are on a Manhattan grid
- **Theorem:** If $h(n)$ is admissible, and if we don't do repeated-state detection, then A^* is optimal

Optimality of A*

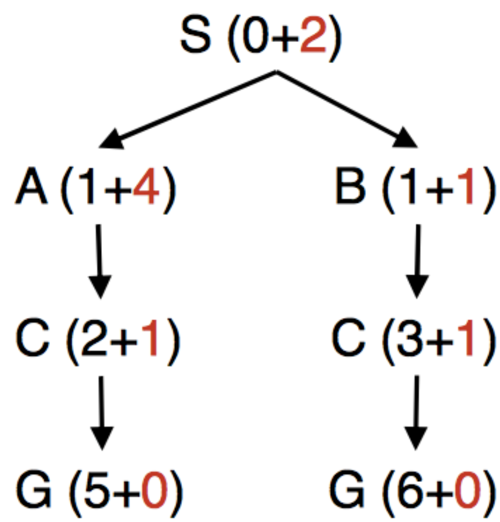
- **Theorem:** If $h(n)$ is admissible, A^* is optimal
  (if we don't do repeated state detection)
- Proof sketch:
  - A* expands all nodes for which $f(n) \leq C^*$, i.e., the *estimated* path cost to the goal is less than or equal to the *actual* path cost to the first goal encountered
  - When we reach the goal node, all the other nodes remaining on the frontier have *estimated* path costs to the goal that are at least as big as C^*
  - Because we are using an admissible heuristic, the *true* path costs to the goal for these nodes cannot be less than C^*

A* gone wrong?

State space graph

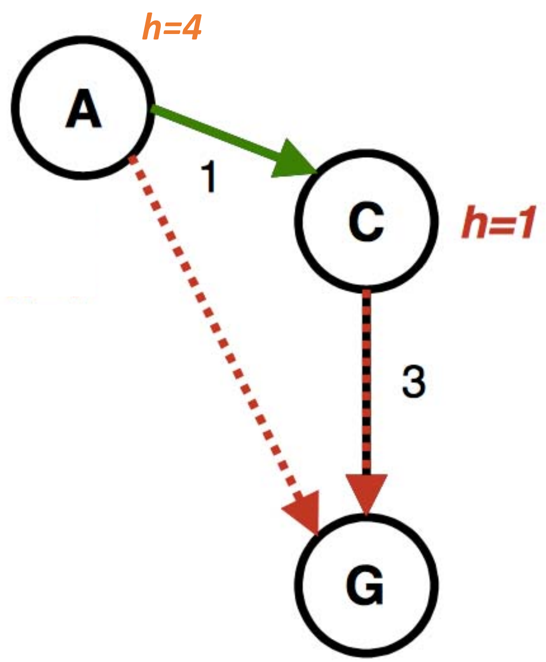


Search tree



Source: [Berkeley CS188x]

Consistency of heuristics



- Consistency: Stronger than admissibility
- Definition:

  cost(A to C) + h(C) ≥ h(A)

  cost(A to C) ≥ h(A) - h(C)

  real cost ≥ cost implied by heuristic
- Consequences:
  - The f value along a path never decreases
  - A* graph search is optimal

Source: Berkeley CS188x

Optimality of A*

- **Tree search** (i.e., search without repeated state detection):
  - A* is optimal if heuristic is ***admissible*** (and non-negative)
- **Graph search** (i.e., search with repeated state detection)
  - A* optimal if heuristic is ***consistent***
- Consistency implies admissibility
  - In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems

Source: [Berkeley CS188x]

Optimality of A*

- A* is *optimally efficient* – no other tree-based algorithm that uses the same heuristic can expand fewer nodes and still be guaranteed to find the optimal solution
  - Any algorithm that does not expand all nodes with $f(n) \leq C^*$ risks missing the optimal solution

Properties of A*

- **Complete?**
  Yes – unless there are infinitely many nodes with $f(n) \leq C^*$
- **Optimal?**
  Yes
- **Time?**
  Number of nodes for which $f(n) \leq C^*$ (exponential)
- **Space?**
  Exponential

Prioritized Search

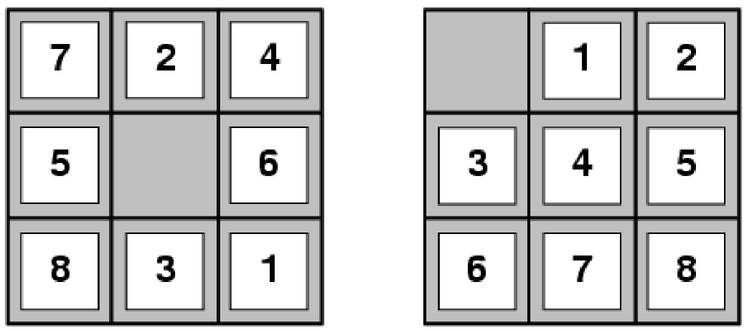
- Review: Tree Search vs. Dijkstra's Algorithm
- Criteria for evaluating a search algorithm: completeness, optimality, computational cost, storage cost
- Search algorithms without side information: BFS, DFS, IDS, UCS
- Search algorithms with side information: GBFS vs. A*
  - Heuristics to guide search
  - Greedy best-first search
  - A* search
  - Admissible vs. Consistent heuristics
  - Designing heuristics: Relaxed problem, Sub-problem, Dominance, Max

Designing heuristic functions

- Heuristics for the 8-puzzle

  $h_1(n)$ = number of misplaced tiles

  $h_2(n)$ = total Manhattan distance (number of squares from desired location of each tile)



Start State　　Goal State

$h_1(\text{start}) = 8$

$h_2(\text{start}) = 3+1+2+2+2+3+3+2 = 18$

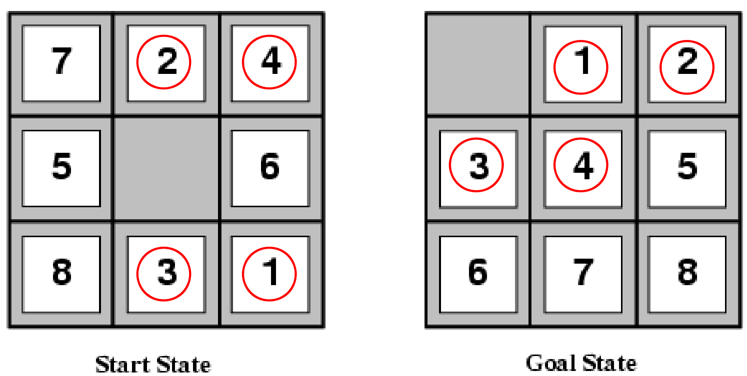
- Are h_1 and h_2 admissible?

Heuristics from relaxed problems

- 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
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Heuristics from subproblems

- Let $h_3(n)$ be the cost of getting a subset of tiles (say, 1,2,3,4) into their correct positions
- Can precompute and save the exact solution cost for every possible subproblem instance – *pattern database*



Dominance

- If h_1 and h_2 are both admissible heuristics and $h_2(n) \geq h_1(n)$ for all n, (both admissible) then h_2 dominates h_1
- Which one is better for search?
  - A* search expands every node with $f(n) < C^*$ or $h(n) < C^* - g(n)$
  - Therefore, A* search with h_1 will expand more nodes

Dominance

- Typical search costs for the 8-puzzle (average number of nodes expanded for different solution depths):

- d=12 IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes

- d=24 IDS ≈ 54,000,000,000 nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

Combining heuristics

- Suppose we have a collection of admissible heuristics $h_1(n)$, $h_2(n)$, …, $h_m(n)$, but none of them dominates the others
- How can we combine them?

$$h(n) = \max\{h_1(n), h_2(n), \ldots, h_m(n)\}$$

All search strategies

Algorithm	Complete?	Optimal?	Time complexity	Space complexity	Implement the Frontier as a…
BFS	Yes	If all step costs are equal	O(b^d)	O(b^d)	Queue
DFS	No	No	O(b^m)	O(bm)	Stack
IDS	Yes	If all step costs are equal	O(b^d)	O(bd)	Stack
UCS	Yes	Yes	Number of nodes w/ g(n) ≤ C*	Number of nodes w/ g(n) ≤ C*	Priority Queue sorted by g(n)
Greedy	No	No	Worst case: O(b^m) Best case: O(bd)	Worse case: O(b^m) Best case: O(bd)	Priority Queue sorted by h(n)
A*	Yes	Yes	Number of nodes w/ g(n)+h(n) ≤ C*	Number of nodes w/ g(n)+h(n) ≤ C*	Priority Queue sorted by h(n)+g(n)