Lecture 3
Solving Problems by Searching

Algorithm

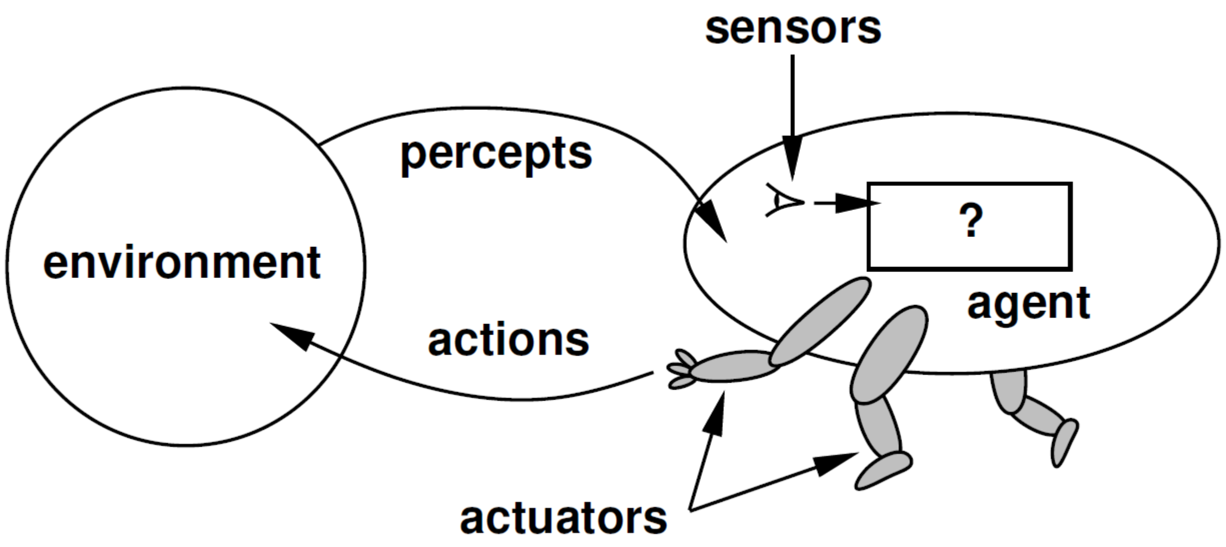
张子臻，中山大学计算机学院

zhangzizhen@gmail.com

Outline

- Problem solving agents
- Search examples
- Uninformed search
- Informed search

Agents and Environments



- Agents include humans, robots, softbots, thermostats, etc.
- The agent function maps from percept histories to actions:

$$f : \mathcal{P}^* \to \mathcal{A}$$

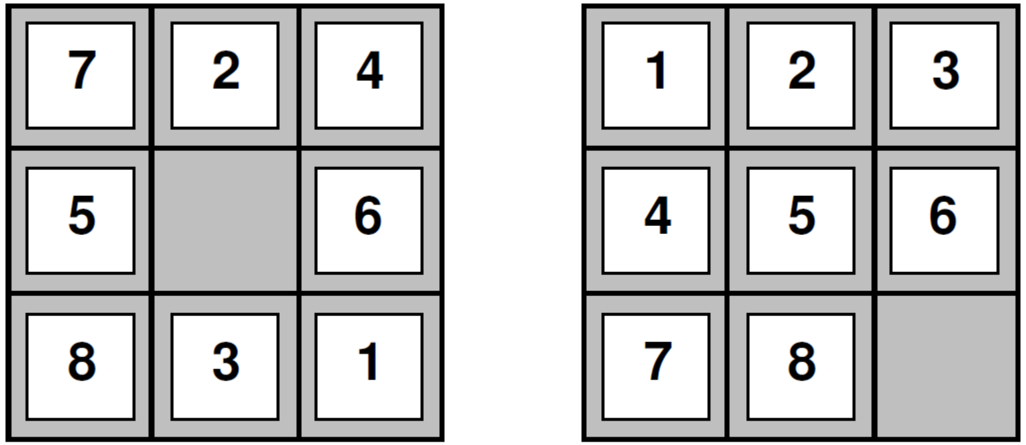
Agents

- Simple-reflex agents directly maps states to actions. They may not operate well in environments where the mapping is too large to store or takes too much to learn.
- Goal-based agents can succeed by considering future actions and desirability of their outcomes.
- Problem solving agent is a goal-based agent that decides what to do by finding sequences of actions that lead to desirable states.
  - Looking for such a sequence is called search.
  - A search algorithm takes a problem as input and returns a solution in the form of action sequence.
  - One a solution is found the actions it recommends can be carried out – execution phase.

Components of the Search

- A search problem can be defined formally by four components.
  - Initial state
  - Goal test
  - Successor function
  - Path cost function that assigns a numeric cost to each path. The cost of a path can be described as the some of the costs of the individual actions along the path – step cost
- Problem solution: sequences of actions to be taken successively.

Search Example: 8-Puzzle Problem

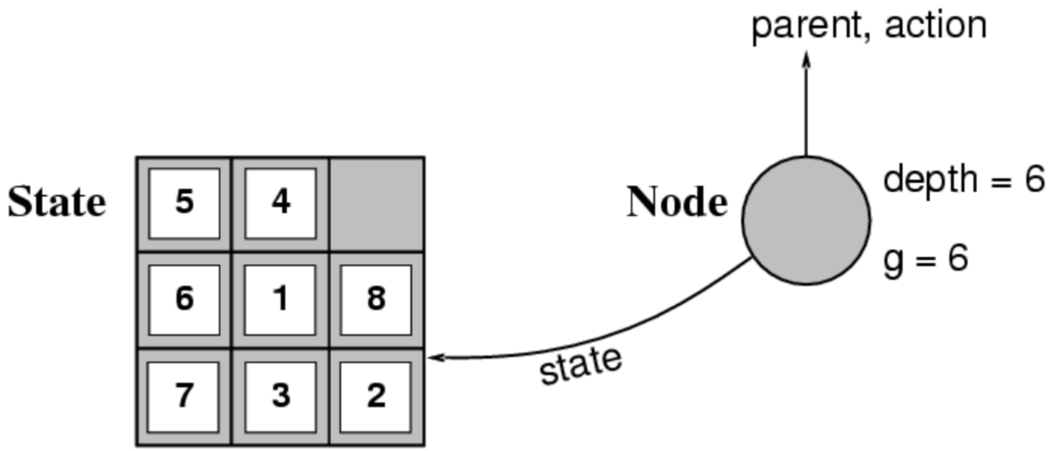


Start State Goal State

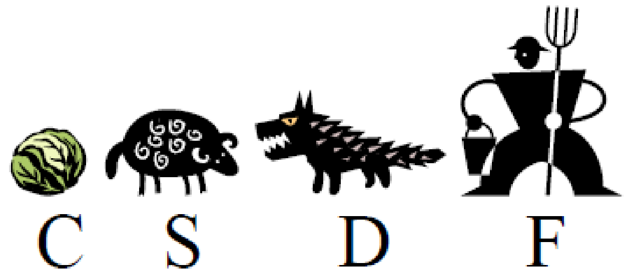
- States: locations of tiles
- Initial state: start state (given)
- Goal test: goal state (given)
- Successor function
  - Action: move blank left, right, up, down
- Path cost: 1 per move

State vs. Node

- A state is a (representation of) a physical configuration.
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth.

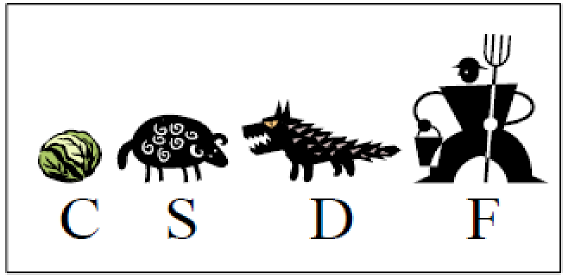


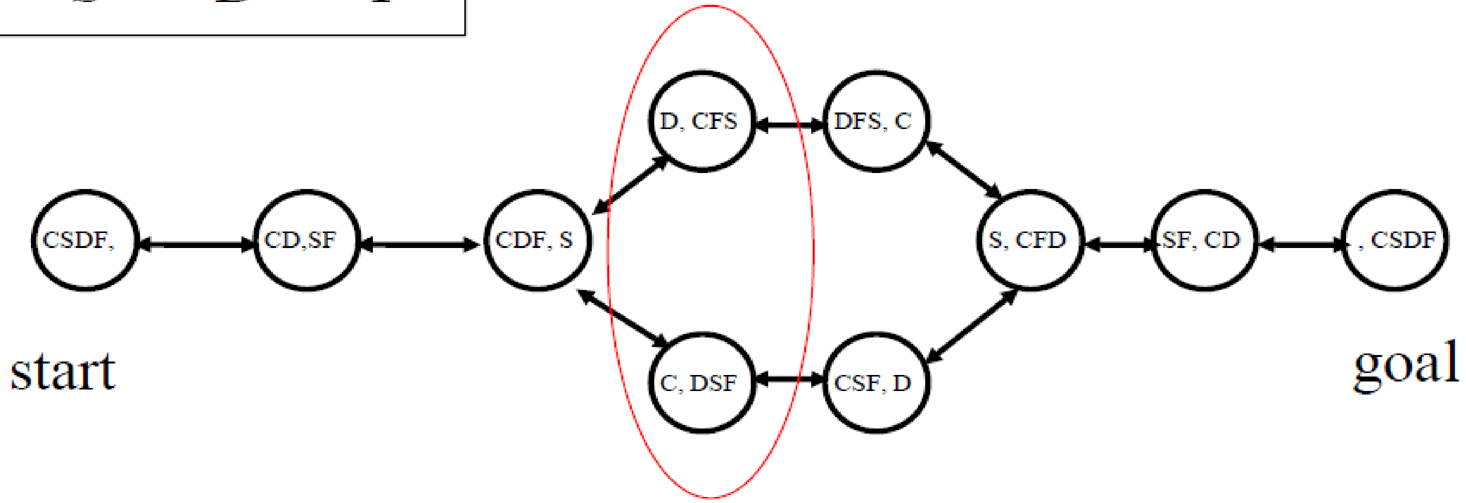
Search Example: River Crossing



- State space ***S***: all valid configurations
- Initial states (nodes) ***I* = {(CSDF,)} ⊆ *S***
  - Where’s the boat?
- Goal states ***G* = {(,CSDF)} ⊆ *S***
- Successor function ***succs(s)* ⊆ *S***: states reachable in one step (one arc) from ***s***
  - ***succs((CSDF,)) = {(CD, SF)}***
  - ***succs((CDF,S)) = {(CD,FS), (D,CFS), (C, DFS)}***
- cost(s,s’) = 1 for all arcs. (weighted for other problems)
- The search problem: find a solution path from a state in ***I*** to a state in ***G.***
  - Optionally minimize the cost of the solution.

Search Graph in State Space





- In general, there will be many generated, but un-expanded states at any given time.
- One has to choose which one to expand next.
- Search strategy: a function that selects the next node to be expanded.

Search Strategies

- A search strategy is defined by picking the order of node expansion.
- Strategies are evaluated along the following dimensions:
  - Completeness: does it always find a solution if one exists?
  - Time complexity: number of nodes generated
  - Space complexity: maximum number of nodes in memory
  - Optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - ***b***: maximum branching factor of the search tree
  - ***d***: depth of the least-cost solution
  - ***m***: maximum depth of the state space (may be ∞)

Uninformed Search

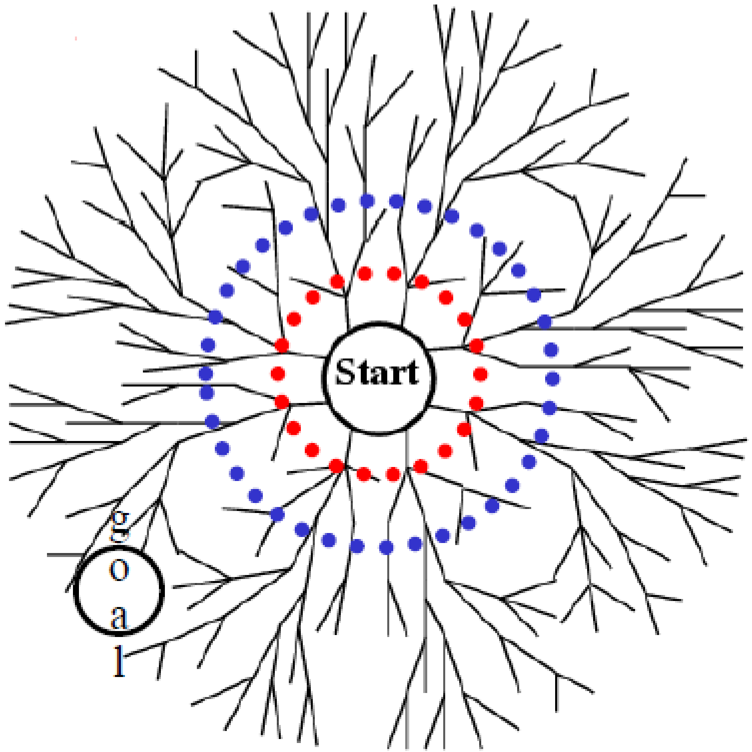
- Uninformed means we only know:
  - The goal test
  - The ***succs()*** function
- But not which non-goal states are better: that would be informed search.
- For now, we also assume ***succs()*** graph is a tree.
  - Won't encounter repeated states.
  - We will discuss it later.
- Search strategies: BFS, UCS, DFS, IDS.

Breadth-First Search (BFS)

- Use a queue (First-in First-out)

```
en_queue(Initial states)
While (queue not empty)
    s = de_queue()
    if (s==goal) success!
    T = succs(s)
    for t in T: t.prev=s
    en_queue(T)
endWhile
```

We need back pointers to recover the solution path.



Performance of BFS

- Assume:
  - the graph may be infinite.
  - Goal(s) exists and is only finite steps away.
- Will BFS find at least one goal?
- Will BFS find the least cost goal?
- Time complexity?
  - Number of states generated
  - Goal: ***d*** edges away
  - Branching factor: ***b***
- Space complexity?
  - Number of states stored

Performance of BFS

- Completeness: yes, BFS will find a goal.
- Optimality: yes, if edges cost 1 (more generally positive non-decreasing in depth), no otherwise.
- Time complexity (worst case): goal is the last node at radius ***d.***
  - Have to generate all nodes at radius ***d.***
  - $b + b^2 + \ldots + b^d \sim O(b^d)$
- Space complexity: $O(b^d)$

Uniform-Cost Search

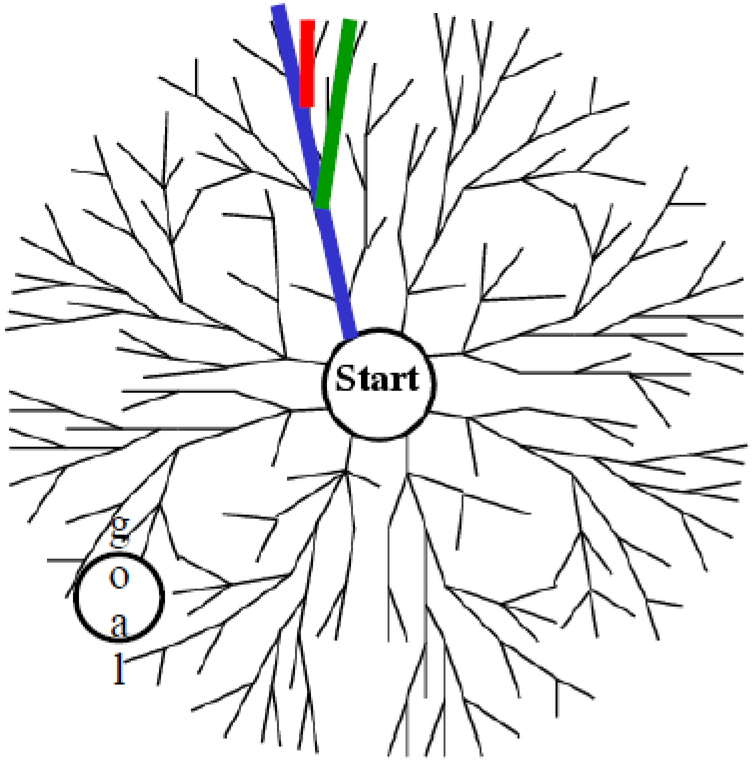
- Find the least-cost goal
- Each node has a path cost from start (= sum of edge costs along the path). Expand the least cost node first.
- Use a priority queue instead of a normal queue
  - Always take out the least cost item
  - Remember *heap?* time ***O(log(number of items in heap))***
- Complete and optimal (if edge costs >= ε > 0)
- Time and space: can be much worse than BFS
  - Let ***C**** be the cost of the least-cost goal
  - $O(b^{C^*/\varepsilon})$, possibly $C^*/\varepsilon \gg d$

Depth-First Search

- Use a stack (First-in Last-out)

```
push(Initial states)
While (stack not empty)
        s = pop()
        if (s==goal) success!
        T = succs(s)
        push(T)
endWhile
```

This is non-recursive implementation of DFS, recursive implementation is more common

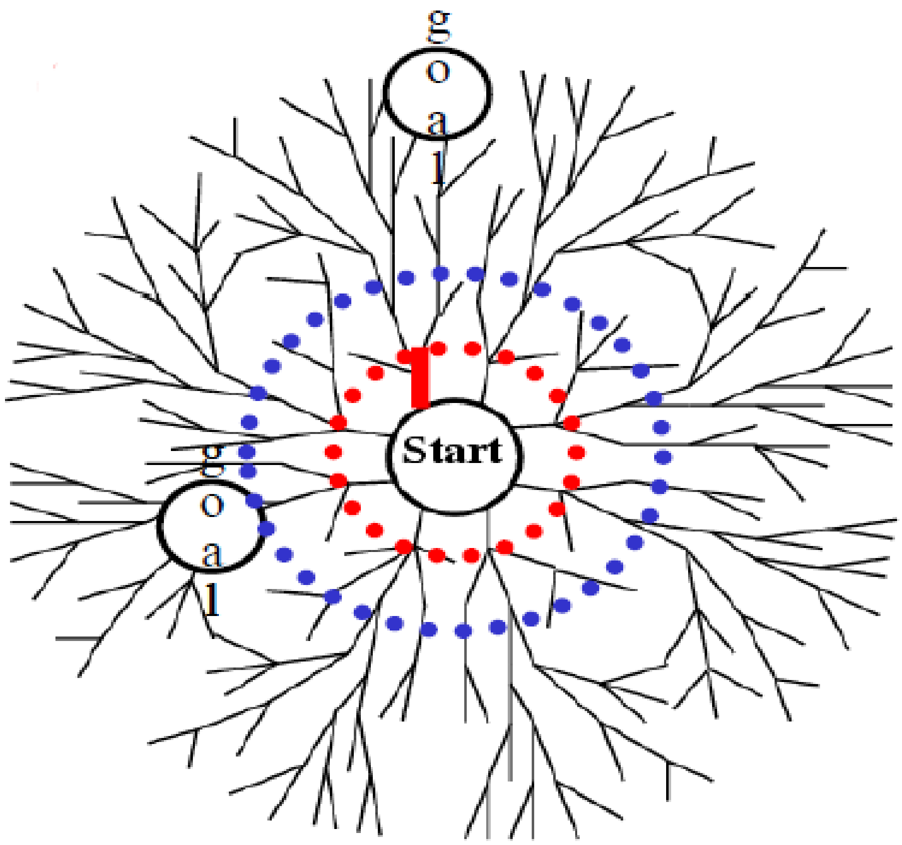


Performance of DFS

- ***m*** = maximum depth of graph from start
- Space complexity: $\boldsymbol{O(mb)}$
- Infinite tree: may not find goal (incomplete)
- May not be optimal
- Finite tree: may visit almost all nodes, time complexity $\boldsymbol{O(b^m)}$

Iterative Deepening Search

1. DFS, but stop if path length > 1.
2. If goal not found, repeat DFS, stop if path length >2.
3. And so on…

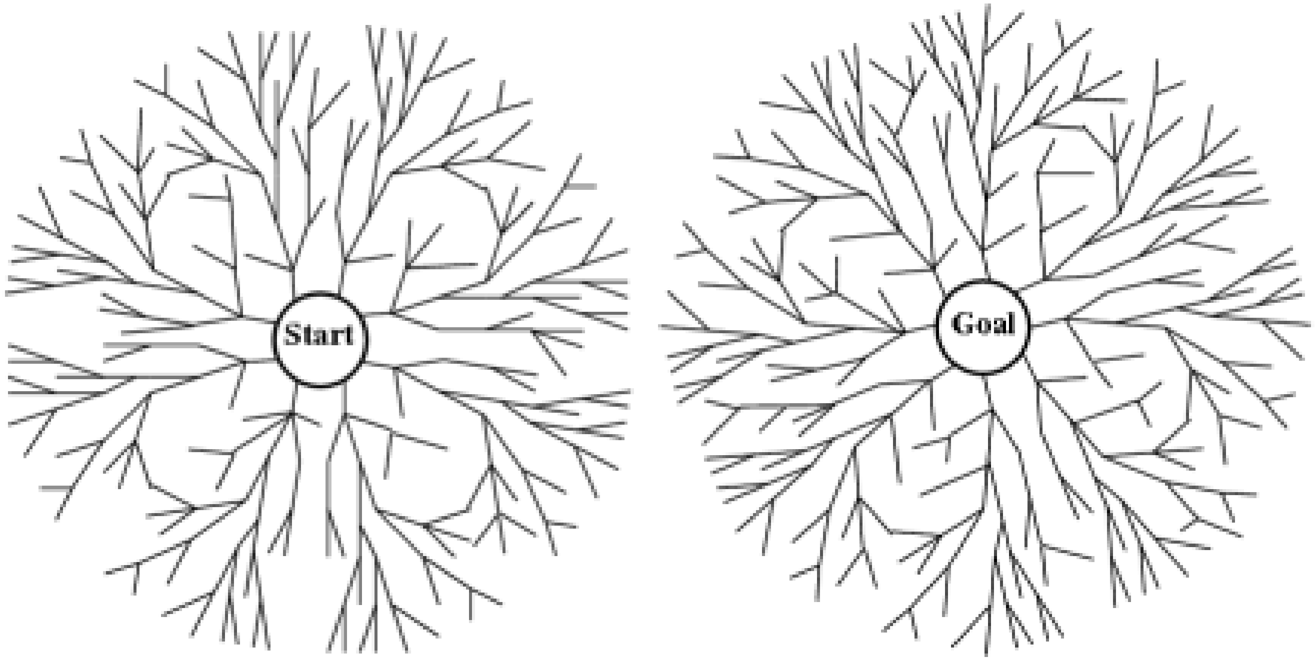


Performance of IDS

- BFS + DFS
  - Complete, optimal like BFS
  - Small space complexity like DFS
- A huge waste?
  - Each deepening repeats DFS from the beginning
  - No! $db+(d-1)b^2+(d-2)b^3+\ldots+b^d \sim O(b^d)$
  - Time complexity like BFS

Bidirectional Search

- Breadth-first search from both start and goal
- Stop when fringes meet
- The fringes(边缘) are $O(b^{d/2})$
- Generates $O(b^{d/2})$ instead of $O(b^{d})$ nodes



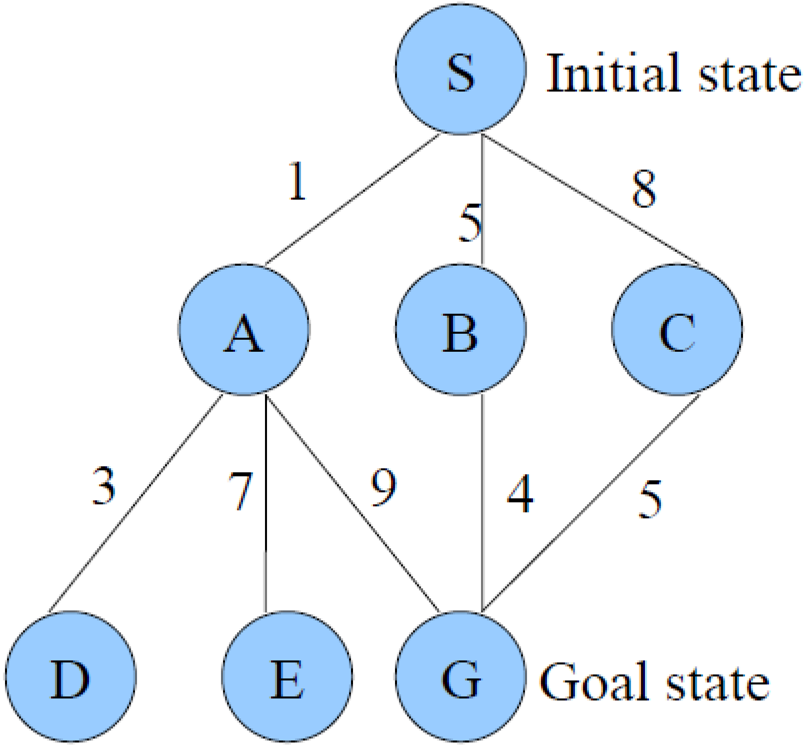
Performance of Search Algorithms

b: branching factor (assume finite) d: goal depth m: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$
Iterative deepening	Y	Y, if [1]	$O(b^d)$	$O(bd)$
Bidirectional search	Y	Y, if [1]	$O(b^{d/2})$	$O(b^{d/2})$

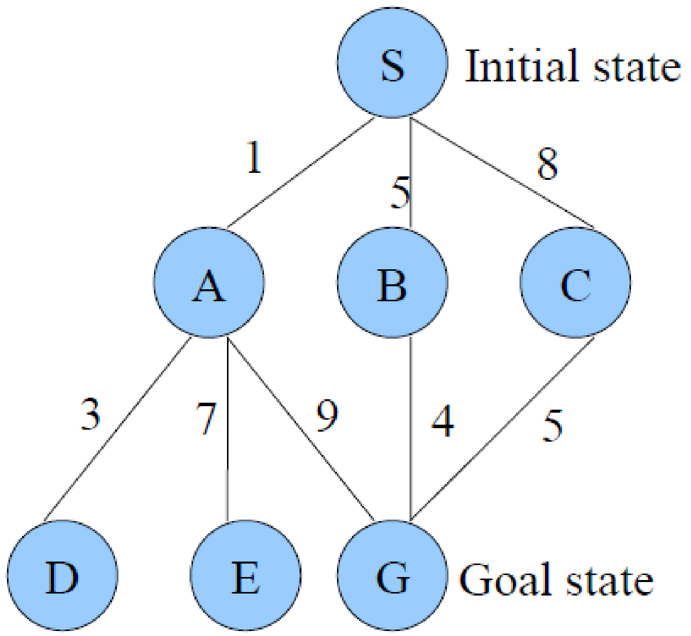
1. edge cost constant, or positive non-decreasing in depth

An Example



- All edges are directed, pointing downwards

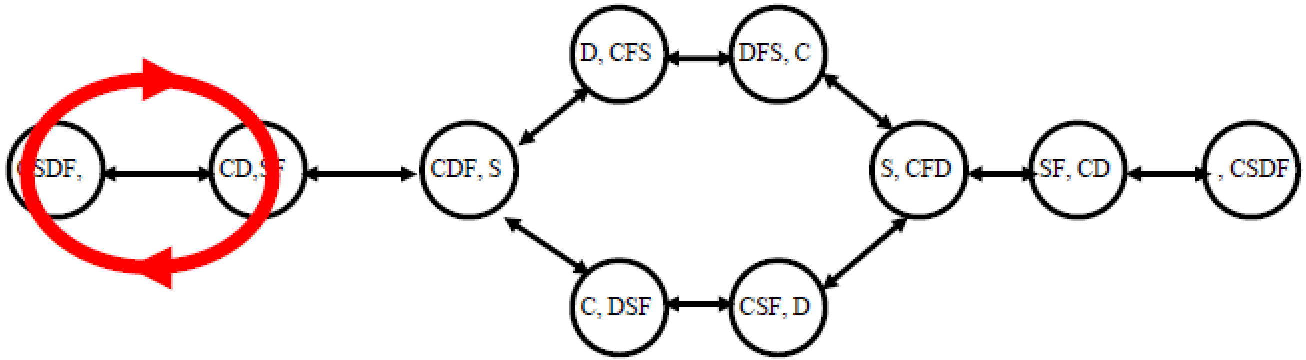
Node Expansion



- Depth-First Search:
  - S A D E G
  - Solution found: S A G
- Breadth-First Search:
  - S A B C D E G
  - Solution found: S A G
- Uniform-Cost Search:
  - S A D B C E G
  - Solution found: S B G (This is the only uninformed search that worries about costs.)
- Iterative-Deepening Search:
  - S A B C S A D E G
  - Solution found: S A G

General Graph Search

- The problem: repeated states



- We have to remember already-expanded states (CLOSED).
- When we take out a state from the fringe (OPEN), check whether it is in CLOSED (already expanded).
  - If yes, throw it away.
  - If no, expand it (add successors to OPEN), and move it to CLOSED.

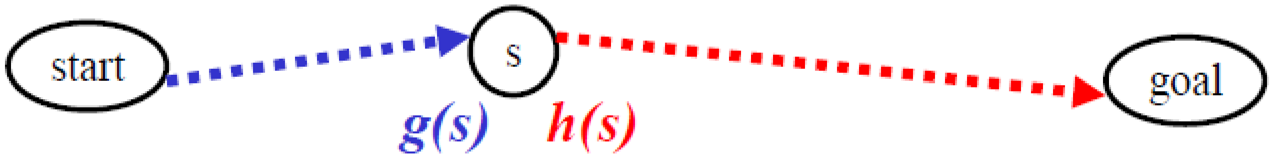
General Graph Search

- BFS:
  - Still $\boldsymbol{O(b^d)}$ space complexity
- DFS:
  - Memorizing DFS (MEMDFS): memorize every expanded states
  - Path Check DFS (PCDFS): remember only expanded states on current path (from start to the current node)

Informed Search

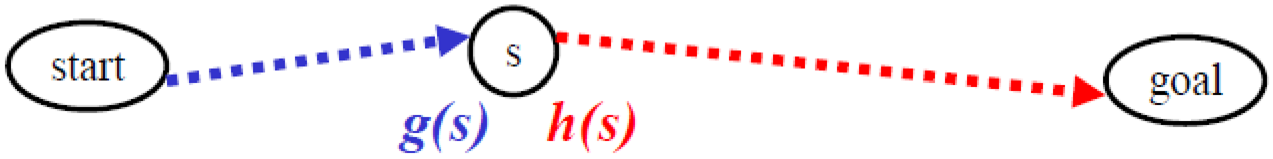
- Uninformed search
  - Knows the actual path cost ***g(s)*** from the start to a node ***s.***

- Informed search
  - Also has a heuristic ***h(s)*** of the cost from ***s*** to goal.
  - Can be much faster than uninformed search.



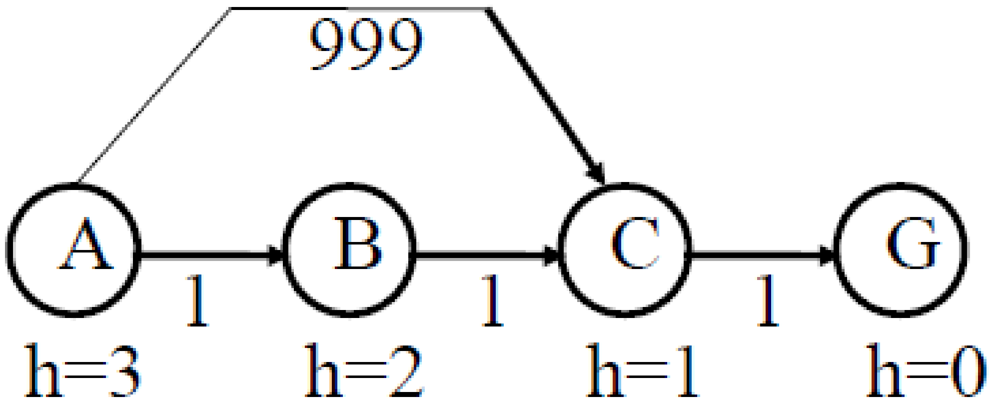
Recall: Uniform-Cost Search

- Uniform-cost search: uninformed search when edge costs are not the same.
- Complete (will find a goal) and Optimal (will find the least-cost goal).
- Always expand the node with the least ***g(s)***
- Use a priority queue:
  - Push in states with their first-half-cost ***g(s)***
  - Pop out the state with the least ***g(s)*** first
- Now we have an estimate of the second-half-cost ***h(s),*** how to use it?



Best-First Greedy Search

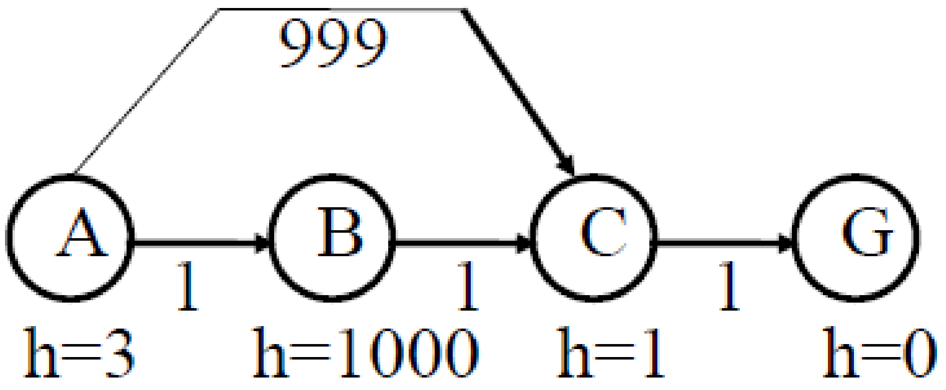
- Use ***h(s)*** instead of ***g(s)***
- Always expand the node with the least ***h(s)***
- Not optimal



It will follow the path *A -> C -> G*

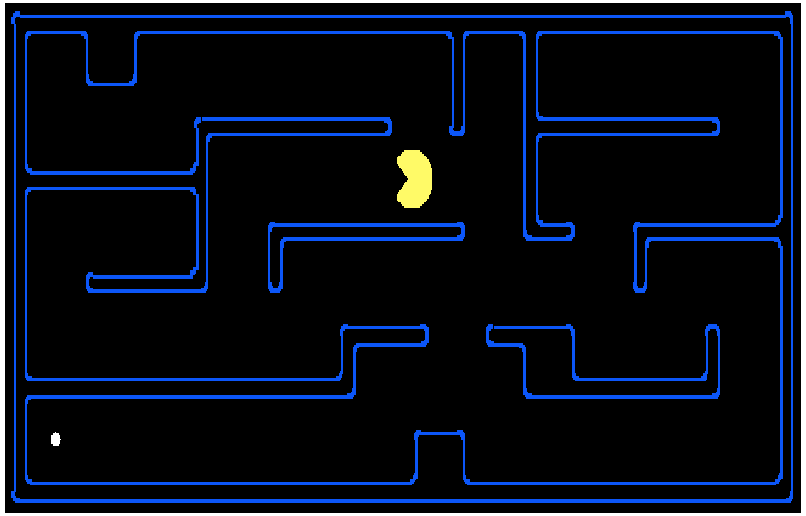
A Search

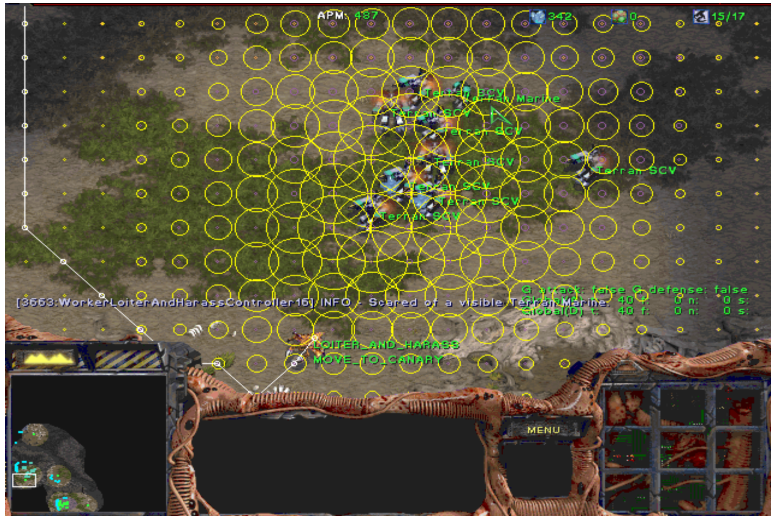
- Use ***f(s)=g(s)+h(s)***
- Always expand the node with the least ***g(s)+h(s)***
- A search is not always optimal



A* Search

- Same as A search, but the heuristic function ***h()*** has to satisfy ***h(s)<=h*(s)*,** where ***h*(s)*** is the true cost from node s to the goal.
- Such heuristic function ***h()*** is called ***admissible*.**
- An admissible heuristic never over-estimates.
- A search with admissible ***h()*** is called ***A* search.***





Admissible Heuristic Functions *h*

- 8-puzzle example

Example State

1		5
2	6	3
7	4	8

Goal State

1	2	3
4	5	6
7	8	

- Which of the following are admissible heuristics?
  - h(n)=number of tiles in wrong position
  - h(n)=0
  - h(n)=1
  - h(n)=sum of Manhattan distance between each tile and its goal location

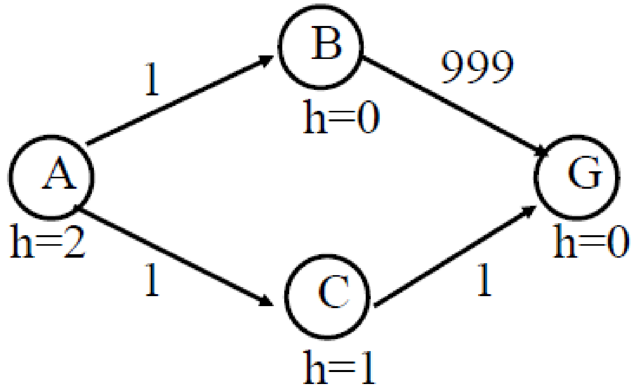
Admissible Heuristics

- A heuristic function $\boldsymbol{h_2}$ **dominates** $\boldsymbol{h_1}$ if for all $\boldsymbol{s}$

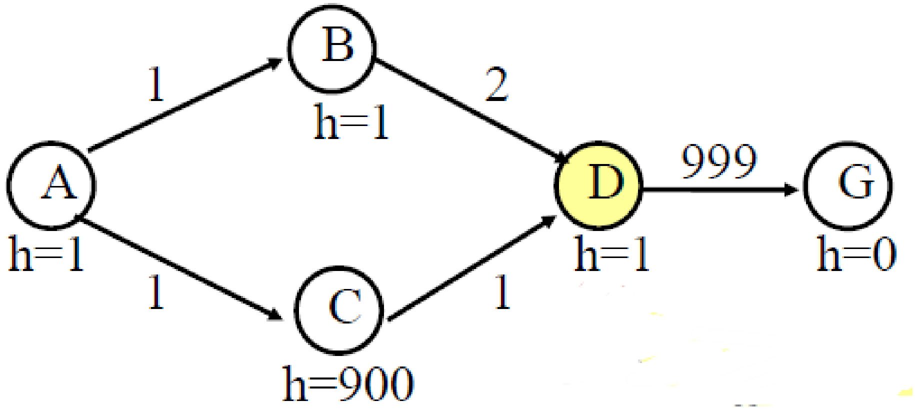
  $h_1(s) <= h_2(s) <= h^*(s)$
- d = 14,
  - $A^*(h_1)$ = 539 nodes
  - $A^*(h_2)$ = 113 nodes
- d = 24,
  - $A^*(h_1)$ = 39,135 nodes
  - $A^*(h_2)$ = 1,641 nodes
- We prefer heuristic functions as close to ***h**** as possible, but not over ***h****.
- Good heuristic function might need complex computation.
- Time may be better spent, if we use a faster, simpler heuristic function and expand more nodes.

Some Tricks

- A* should terminate only when a goal is popped from the priority queue



- A* can revisit an expanded state, and discover a shorter path



The A* Algorithm

Use priority queue

1. Put the start node ***S*** on the priority queue, called **OPEN**
2. If **OPEN** is empty, exit with failure
3. Remove from **OPEN** and place on **CLOSED** a node ***n*** for which ***f(n)*** is minimum
4. If ***n*** is a goal node, exit (trace back pointers from ***n*** to ***S***)
5. Expand ***n***, generating all its successors and attach to them pointers back to ***n***.
   For each successor ***n'*** of ***n*** not on **CLOSED**

   Use hashing

   1. If n' is not already on **OPEN**, estimate ***h(n')***,***g(n')=g(n)+c(n,n')***, ***f(n')=g(n')+h(n')***, and place it on **OPEN**.
   2. If ***n'*** is already on **OPEN**, then check if ***g(n')*** is lower for the new version of ***n'***. If so, then:
   - Redirect pointers backward from ***n'*** along path yielding lower ***g(n')***.
   - Put ***n'*** on **OPEN**.
   - If ***g(n')*** is not lower for the new version, do nothing.
6. Goto 2.

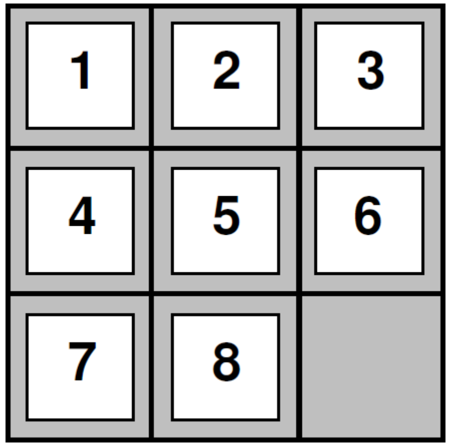
Choosing a Hash Function

- A hash function should be easy and quick to compute.
- A hash function should achieve an even distribution of the keys that actually occur across the range of indices.
- The usual way to make a hash function is to take the key, chop it up, mix the pieces together in various ways, and thereby obtain an index that will be uniformly distributed over the range of indices.

Hashing Functions for Permutations

- Sometimes, when keys are not integers, try to convert them to integers.
- For example, how to convert the permutation (2,4,1,3) to a number?
  - Cantor expansion / reverse Cantor expansion
  - *http://baike.baidu.com/view/437641.htm*

```
int PermutationToNumber(int permutation[],   int n) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        int count = 0;
        for (int j = i + 1; j < n; j++) {
            if (permutation[j] < permutation[i])  count++;
        }
        // factorials[j] records j!
        result += count * factorials[n - i - 1];
    }
    return result;
}
```



Consistent Heuristics

- Consistency is analogous to the triangle inequality from Euclidian geometry.
- A heuristic is consistent if

  $h(n) <= c(n, a, n') + h(n')$
- If h is consistent, then for

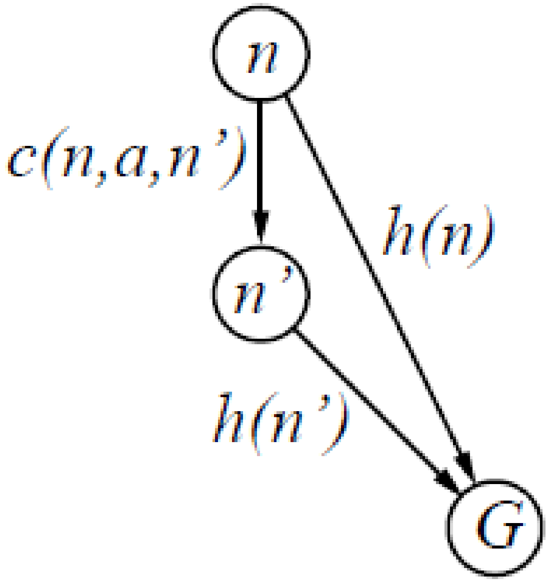
  every child n' of n, we have:

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

  $= g(n) + c(n, a, n') + h(n')$

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

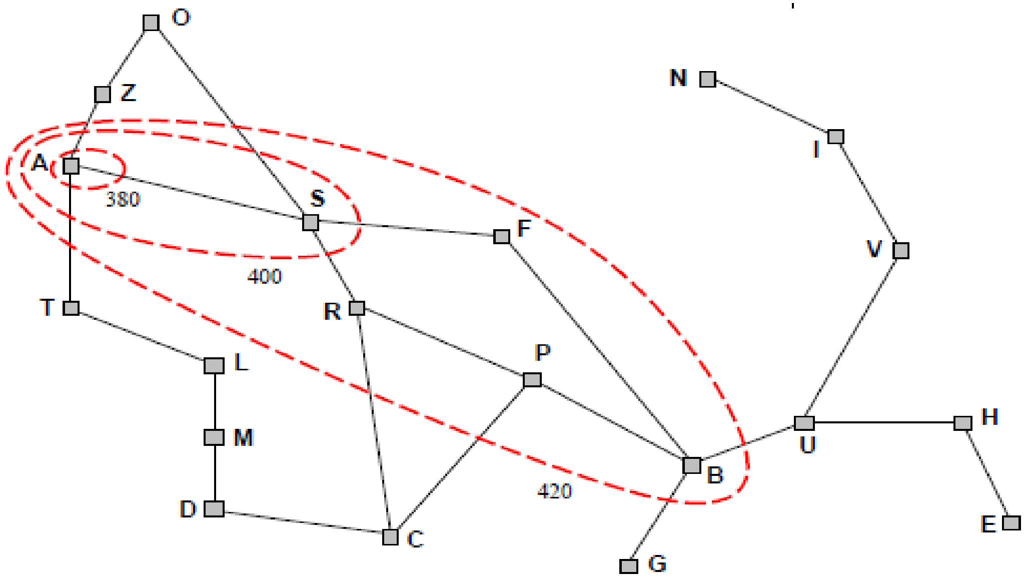
  $= f(n)$



- That is, $f(n)$ is non-decreasing along any path.

Behavior of A* with Consistent Heuristic

- If *h* is consistent, then A* expands nodes in order of increasing *f* value. In such a case, A* can be implemented more efficiently — no node needs to be processed more than once.
- Gradually adds “*f*-contours" of nodes (breadth-first adds layers)
- Contour *i* has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A*

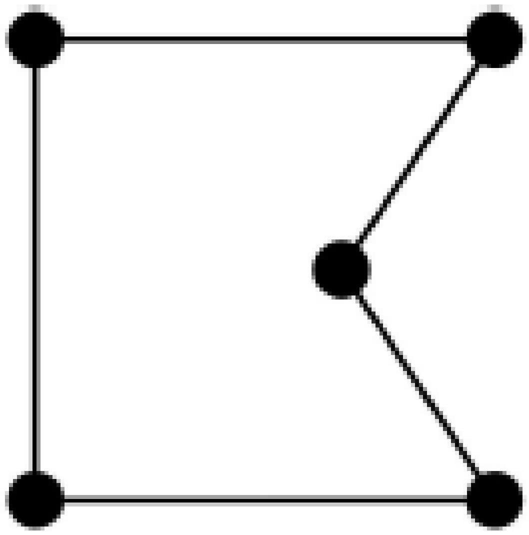
- Complete? Yes
- Time? *O(entire state space)* in worst case, *O(d)* in best case
- Space? Keeps all nodes in memory
- Optimal? Yes

A* for Traveling Salesman Problem

- For a node ***s***

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

- What is ***g(s)***, ***h(s)?***
- A* v.s. branch-and-bound



Iterative-Deepening A*

function IDA*(*problem*) **returns** a solution
  inputs: *problem*, a problem

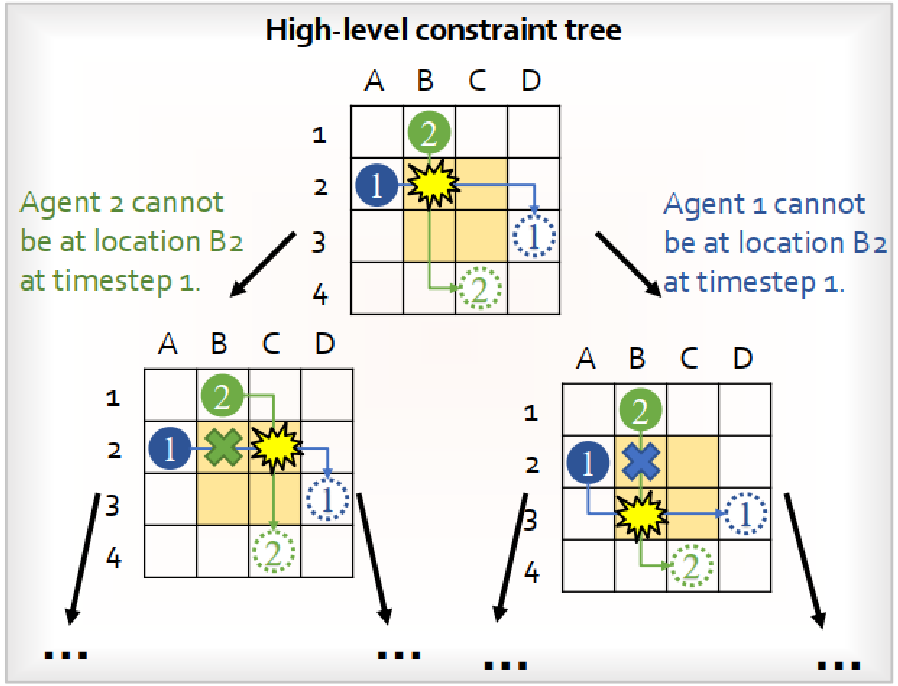
  $f_0 \leftarrow$ *h(initial state)*
  for $i \leftarrow$ 0 **to** ∞ **do**
    result $\leftarrow$ COST-LIMITED-SEARCH(*problem*, f_i)
    if *result* is a solution **then return** *result*
    else $f_{i+1} \leftarrow$ result
  end

function COST-LIMITED-SEARCH(*problem*, *fmax*) **returns** solution or number

  depth-first search, backtracking at every node n such that $f(n) > fmax$
  if the search finds a solution **then**
    return the solution
  else
    return $\min\{f(n) \mid$ the search backtracked at $n\}$

Variants of A*

- https://theory.stanford.edu/~amitp/GameProgramming/Variations.html
  - Beam search
  - Iterative deepening
  - Weighted A*
  - Bandwidth search
  - Bidirectional search
  - Dynamic A* and Lifelong Planning A*
  - …
- Multi-Agent Path Finding
  - Conflict-Based Search (CBS)



Thank you!