Lecture 4: Search 3

Victor R. Lesser
CMPSCI 683
Fall 2010

First Homework

1st Programming Assignment – 2 separate parts (homeworks)

- First part due on (9/27) at 5pm
- Second part due on 10/13 at 5pm

Send homework writeup as .pdf file and code to TA

Today's lecture

- Overview of Search Strategies
- Blind Search (Most slides will be skipped)
- Informed Search
  - How to use heuristics (domain knowledge) in order to accelerate search?
  - A* and IDA*
  - Reading: Sections 4.1-4.2.

General Tree Search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem] applied to STATE(node) succeeds return node
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

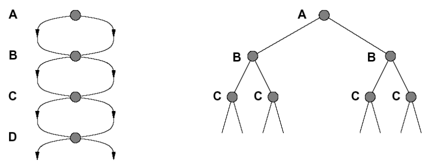
Fringe is often called open list

```
function EXPAND( node, problem) returns a set of nodes
   successors ← the empty set
   for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
      s ← a new NODE
      PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
   return successors
```

Set up state of node

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential



Avoiding Repeated States

- Do not re-generate the state you just came from.
- Do not create paths with cycles.
- Do not generate any state that was generated before (using a hash table to store all generated nodes)
  - Markov Assumption
- Add **Close list** to search algorithm or cleverly construct search space

Graph Search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem](STATE[node]) then return node
      if STATE[node] is not in closed then
         add STATE[node] to closed
         fringe ← INSERTALL(EXPAND(node, problem), fringe)
   end
```

Don't expand node if already on closed list

Search Strategies

- A key issue in search is limiting the portion of the state space that must be explored to find a solution.
- The portion of the search space explored can be affected by the order in which states (and thus solutions) are examined.
- The search strategy determines this order by determining which node (partial solution) will be expanded next.

Search Strategies

- A key issue in search is limiting the portion of the state space that must be explored to find a solution.
- The portion of the search space explored can be affected by the order in which states (and thus solutions) are examined.
- The search strategy determines this order by determining which node (partial solution) will be expanded next.

Search Strategies

- A strategy: picking the order of node expansion
- Evaluation criteria:
  - **Completeness**: is the strategy **guaranteed** to find a **solution** when there is one?
  - **Time complexity**: how **long** does it take to find a solution? (number of nodes generated)
  - **Space complexity**: how much **memory** does it need to perform the search? (maximum number of nodes in memory)
  - **Optimality**: does the strategy find the **highest-quality solution** when there are **several solutions**?
- Time and space complexity are measured :
  - ***b*** – maximum branching factor of the search tree
  - ***d*** – depth of the least-cost solution
  - ***m*** – maximum depth of the state space (may be infinity)

Complexity of Search Strategies

- The extent to which partial solutions are kept and used to guide the search process
- The context used in making decisions
- The degree to which the search process is guided by domain knowledge
- The degree to which control decisions are made dynamically at run-time

Search Strategy Classification

- Search strategies can be classified in the following general way:
  - Uninformed/blind search;
  - Informed/heuristic search;
    - Relationship of nodes to goal state, and intra-node relationships
  - Multi-level/multi-dimensional/multi-direction;
  - Systematic versus Stochastic
  - Game/Adversarial search
    - Game search deals with the presence of an opponent that takes actions that diminish an agent's performance (see AIMA Chapter 6).

Uninformed/Blind Search Strategies

- Uninformed strategies do not use any information about how *close (distance,cost)* a node might be to a goal (additional cost to reach goal).
- They differ in the order that nodes are expanded (and operator cost assumptions).

Examples of Blind Search Strategies

- Breadth-first search (open list is FIFO queue)
- Uniform-cost search (shallowest node first)
- Depth-first search (open list is a LIFO queue)
- Depth-limited search (DFS with cutoff)
- Iterative-deepening search (incrementing cutoff)
- Bi-directional search (forward and backward)

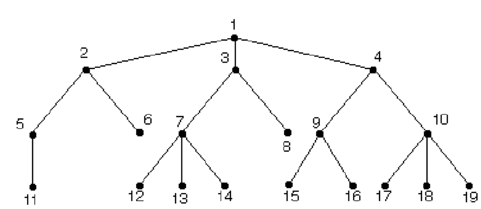
Will Skip Following Slides in Class Discussion

- Breadth-first Search
- Uniform-cost Search
- Depth-first Search
- Depth-limited Search

Breadth-First Search

Expand shallowest unexpanded node

Fringe: is a FIFO queue, new successors go to the end



Breadth-First Search

(b - branching factor, d -depth)

- **Completeness**:
- Yes
- **Time complexity**:
- $1+b+b^2+b^3+\ldots+b^d = O(b^{d+1})$
- **Space complexity**:
- $O(b^{d+1})$, keep every node in memory.
- **Optimality**:
- Yes (only if step costs are identical)

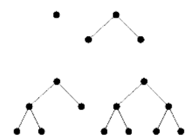


Figure Breadth-first search tress after 0, 1, 2, and 3 node expansions (b=2, d=2)

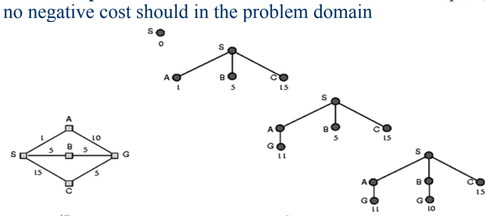
Breadth-First Search (cont)

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

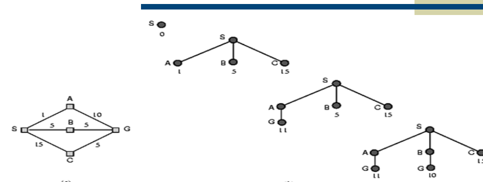
- Time and Memory requirements for a breadth-first search.
- The figures shown assume (1) branching factor b=10; (2) 1000 nodes/second; (3) 100 bytes/node
- **Time** and **Space complex** Cannot be use to solve any but the **smallest problem**

Uniform Cost Search

- BFS finds the ***shallowest*** goal state.
- Uniform cost search modifies the BFS by **expanding ONLY the lowest cost node** (as measured by the path cost *g(n))*
- The **cost of a path** must **never decrease** as we traverse the path, ie. no negative cost should in the problem domain



Uniform Cost Search



- **Completeness**:
- Yes
- **Time complexity**:
- $b^{C^*/e}$
  - ✓ C*: optimal cost, e:minimum cost of each step

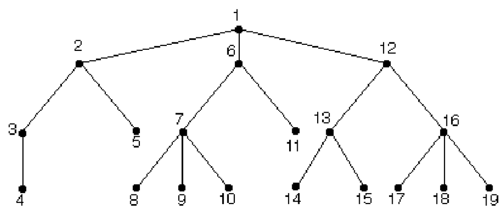
Space complexity:
$b^{C^*/e}$

Optimality:
Yes

Depth-First Search (Cont'd)

Expand deepest unexpanded node

Fringe: is a LIFO queue, new successors go to the front



Depth-First Search

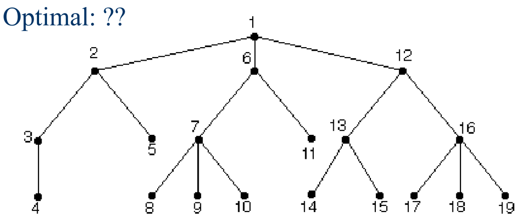
- DFS always **expands one** of the nodes at the deepest level of the tree.
- The search **only go back** once it **hits a dead end** (a non-goal node with no expansion)
- DFS have **modest memory requirements**, it only needs to store a single path from root to a leaf node.
- For **problems that have many solutions**, **DFS** may actually be faster than BFS, because it has a good chance of finding a solution after exploring only a small portion of the whole space.

Depth-First Search (cont)

- One problem with DFS is that it can get **stuck** going down the wrong path.
- Many problems have **very deep** or even **infinite** search trees.
- DFS should be **avoided** for search trees with **large** or **infinite maximum depths**.
- It is common to implement a **DFS** with a **recursive function** that calls itself on each of its children in turn.

Properties of Depth-first Search

- Complete: ??
- Time: ??
- Space: ??
- Optimal: ??

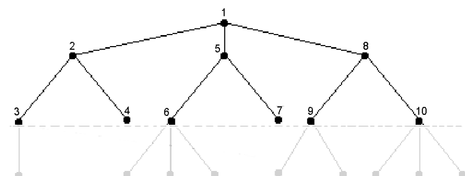


Properties of Depth-first Search

- Complete: No, fails in infinite-depth spaces, spaces with loops. Modify to avoid repeated states along path: complete in finite spaces
- Time: b^m: terrible if m is much larger than d (depth of solution), but if solutions are dense, may be much faster than breadth-first
- Space: $b \cdot m$, i.e., linear space!
- Optimal: no
  - ***b*** – maximum branching factor of the search tree
  - ***m*** – maximum depth of the state space

Depth-Limited Search

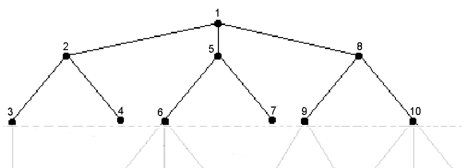
- "**Practical**" DFS
- DLS avoids the pitfalls of DFS by imposing a cutoff on the **maximum depth** of a path.
- However, if we choose a depth limit that is too small, then DLS is not even complete.
- The time and space complexity of DLS is similar to DFS.



Depth-Limited Search (cont)

✓ *(b-branching factor, l-depth limit)*

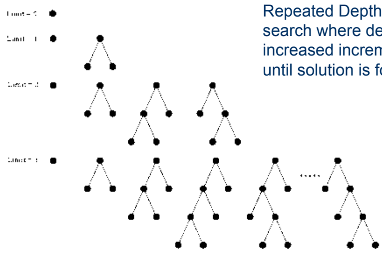
- **Completeness**:
- Yes, only if $l >= d$
- **Time complexity**:
- b^l
- **Space complexity**:
- bl
- **Optimality**:
- No



Iterative Deepening Search

- The **hard part** about DLS is **picking a good limit**.
- IDS is a strategy that sidesteps the issue of choosing the best depth limit by **trying all possible depth limits**: first depth 0, then depth 1, the depth 2, and so on.

Iterative Deepening Search (cont)



Repeated Depth-limited search where depth-limit is increased incrementally until solution is found

Properties of iterative deepening search

- Complete:??
- Time: ??
- Space: ??
- Optimal: ??

Properties of iterative deepening search

- Complete: Yes
- Time:

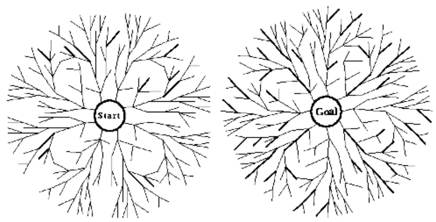
$$(d+1)1 + (d)b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d = O(b^d)$$

- Space: O(bd)
- Optimal: Yes, if step cost = 1
  Can be modified to explore uniform-cost tree

Iterative Deepening Search (cont)

- IDS may seem wasteful, because so many states are expanded multiple times.
- For most problems, however, the **overhead** of this multiple expansion is actually **rather small**.
  - Major cost is at fringe where solution is; this last fringe only occurs once
- In effect, it combines the **benefits of DFS and BFS**. It is **optimal** and **complete**, like BFS and has **modest memory requirements** of DFS.
- **IDS** is the **preferred** search method when there is a **large search space** and the **depth** of the solution is **not known**.

Bi-directional Search



A schematics view of a bi-directional BFS that is about to succeed, when a branch from the start node meets a branch from the goal node

Bi-directional Search

- **Search forward** from the Initial state
- And **search backwards** from the Goal state.
- Stop when two meets in the **middle**.
- Why is this good??; When can you do such a search??
- Each search checks each node before it is expanded to see if it is in the fringe of the other search.
  - **Completeness**:
  - Yes
  - **Time complexity**:
  - $b^{d/2}$
  - **Space complexity**:
  - $b^{d/2}$
  - **Optimality**:
  - Yes

Comparing Blind Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

- b - branching factor; d is the depth of the solution; m is the maximum depth of the search tree; l is the depth limit
- BFS, IDS, and BDS are optimal only if step costs are all identical.
- Iterative deepening search uses only linear space and not much more time

Informed/Heuristic Search

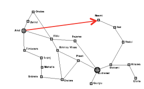
- While uninformed search methods can in principle find solutions to any state space problem, they are typically too inefficient to do so in practice.
- Informed search methods use *problem-specific knowledge* to improve *average* search performance.

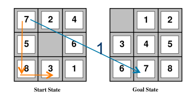
What are heuristics?

- Heuristic: *problem-specific knowledge* that reduces expected search effort.
- Informed search uses a heuristic evaluation function that denotes the relative desirability of expanding a node/state.
  - often include some estimate of the *cost to reach the nearest goal state* from the current state.
  - How much of the state of the search does it take into account in making this decision?
- In blind search techniques, such knowledge can be encoded only via state space and operator representation.

Examples of heuristics

- Travel planning
  - **Euclidean distance**
- 8-puzzle
  - **Manhattan distance**
  - **Number of misplaced tiles**
- Traveling salesman problem
  - **Minimum spanning tree**





Where do heuristics come from?

Heuristics from relaxed models

- Heuristics can be generated via simplified models of the problem
- Simplification can be modeled as deleting constraints on operators
- Key property: Heuristic can be calculated efficiently (*low overhead -- why important?*)

Informed Search Strategies

- Best-first search (a.k.a. ordered search):
  - greedy (a.k.a. best-first)
  - A*
  - ordered depth-first (a.k.a. hill-climbing)
- Memory-bounded search:
  - Iterative deepening A* (IDA*)
  - Simplified memory-bounded A* (SMA*)
  - Recursive Best First Search (RBFS) Time-bounded search:
  - Anytime A*
  - RTA* (searching and acting)
- Iterative improvement algorithms (generate-and-test approaches):
  - Steepest ascent hill-climbing
  - Random-restart hill-climbing
  - Simulated annealing
- Multi-Level/Multi-Dimensional Search
  - Hierarchical A*
  - Blackboard

Informed/Heuristic Search

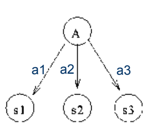
- While uninformed search methods can in principle find solutions to any state space problem, they are typically too inefficient to do so in practice.
- Informed search methods use *problem-specific knowledge* to improve *average* search performance.

What are heuristics?

- Heuristic: problem-specific knowledge that reduces ***expected*** search effort.
  - In blind search techniques, such knowledge can be encoded only via state space and operator representation.
- Informed search uses a ***heuristic evaluation function that denotes the relative desirability of expanding a node/state***.
  - often include some estimate of the *cost to reach the nearest goal state* from the current state.

One Way of Introducing Heuristic Knowledge into Search – Heuristic Evaluation Function

- heuristic evaluation function ***h* : Ψ --> R**, where **Ψ** is a set of all states and **R** is a set of real numbers, maps each state **s** in the state space **Ψ** into a measurement ***h (s)*** which is an ***estimate* of the cost extending of the cheapest path from *s to a goal node***.



Node A has 3 children.

h(s1)=0.8, *h*(s2)=2.0, *h*(s3)=1.6

The **value** refers to the **cost involved** for an action. A continued search at s1 based on *h*(s1) being the smallest is 'heuristically' the best.

Best-first search

- Idea: use an evaluation function for each node, which estimates its "desirability"
- Expand most desirable unexpanded node
- Implementation: open list is sorted in decreasing order of desirability
- A **combination** of depth first (**DFS**) and breadth first search (**BFS**).
  - Go depth-first until node path is no longer the most promising one (lowest expected cost) then backup and look at other paths that were previously promising (and now are the most promising) but not pursued. At each search step pursuing in a breath-first manner the paths that has lowest expected cost.

Best-First Search*

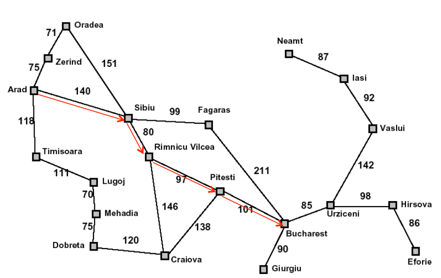
1) Start with *OPEN* containing just the initial state.
2) Until a goal is found or there are no nodes left on *OPEN* do:
   (a) Pick the **best** node (based on the heuristic function) on *OPEN*.
   (b) If it is a goal node, return the solution otherwise place node on the *CLOSED* list
   (b) Generate its successors.
   (c) For each successor node do:
      i. If it has not been generated before (i.e., not on *CLOSED* list), evaluate it, add it to *OPEN*, and record its parent.
      ii. *If it has been generated before,* ***change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have?***

Is this a complete and optimal search?

Greedy search

- Simple form of best-first search
- Heuristic evaluation function $h(n)$ estimates the cost from n to the closest goal
- Example: straight-line distance from city n to goal city (Bucharest)
- Greedy search expands the node (on *OPEN* list) that appears to be closest to the goal
- Properties of greedy search?

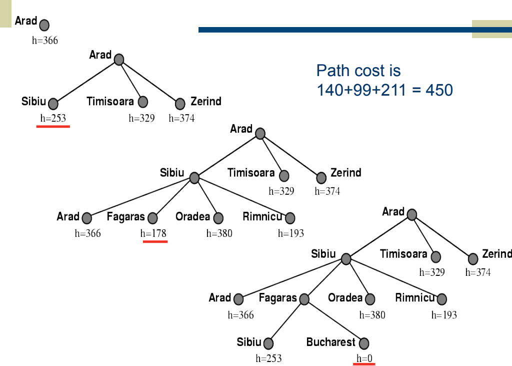
Roadmap of Romania



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

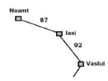
Greedy Search

Path cost is
140+99+211 = 450



Problems with Greedy Search (cont)

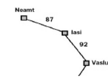
- Complete?
  - No, can get stuck in loops if not maintaining Closed list.
    - e.g. (Lasi to Fagaras) Lasi to Neamt to Lasi to Neamt
    - Neamt dead-end need to turn back



- Time??
  - O(b^m), but a good heuristic can give dramatic improvement *where m is the maximum depth of the search space*
- Space??
  - O(b^m), keeps all nodes in memory
- Optimal??
  - No (minimum cost path in example is 418 rather than 450)

Problems with Greedy Search (cont)

- Complete?
  - No, can get stuck in loops if not maintaining *Closed list*.
    - e.g. (Lasi-> Fagaras) Lasi->Neamt->Lasi->Neamt
    - Neamt dead-end need to turn back



- When h is admissible, monotonicity can be maintained when combined with pathmax:

  $f(n') = \max(f(n), g(n')+h(n'))$

  Equivalent to defining a new heuristic function h' and save f of parent as part of state of node

  $f(n') = g(n') + h'(n')$, $h'(n') = \max(f(n) - g(n'), h(n'))$

Minimizing total path cost: A*

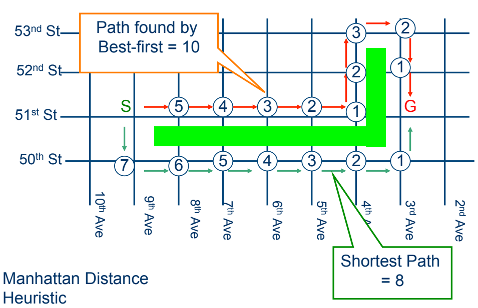
- **G(greedy)S** minimizes **the estimate cost to the goal, *h(n)*, - not optimal** and **incomplete**.
- **U(uniform)C(cost)S** minimizes **the cost of the path so far, *g(n)*** and is **optimal** and **complete** but can be **very inefficient**.
- A* Search **combines both GS** *h(n)* and **UCS** *g(n)* to give *f(n)* which estimates the cost of the **cheapest solution through n.**
- A* is similar to best-first search except that the evaluation is based on total path (solution) cost:

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

  $g(n)$ = cost of path from the initial state to n

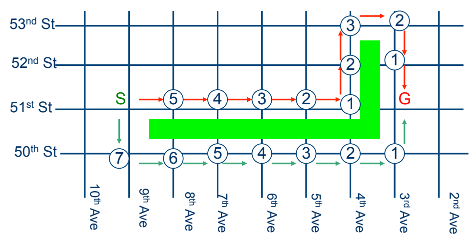
  $h(n)$ = estimate of the remaining distance

Greedy -- Complete, but not optimal…

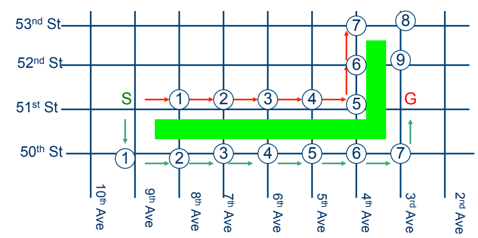


Manhattan Distance Heuristic

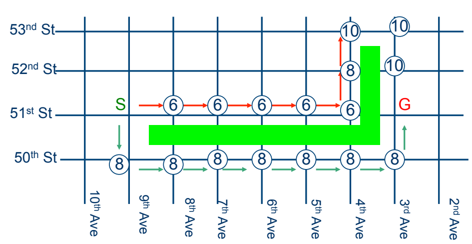
h values

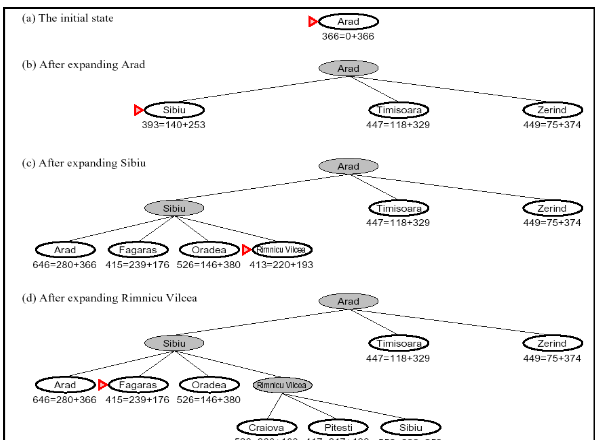


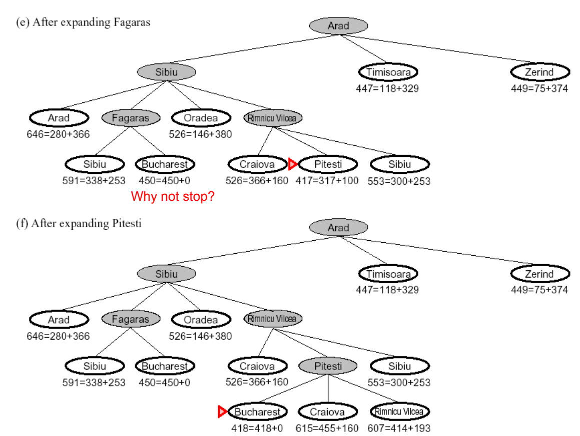
g values



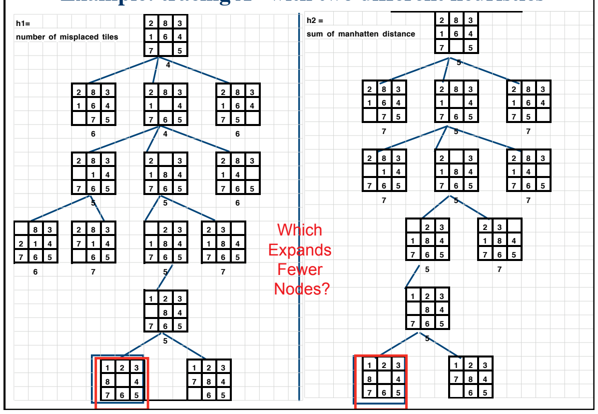
f = g + h







Example: tracing A* with two different heuristics



Admissibility and Monotonicity*

- Admissible heuristic *h(n)* = never overestimates the actual cost *h** *(n)* to reach a goal & ***h(n)>=0*** & ***h*(*goal*)=0**
- Monotone (Consistency) heuristic
  - For every node n and every successor n' reached from n by action a
  - ***h*(*n*) < = cost of (*n, a, n'*) + *h*(*n'*)** & ***h*(*goal*)=0**
    - The deeper you go along a path the better (or as good) the estimate of the distance to the goal state
  - **the *f* value (*which is g+h*) never decreases along any path.**
  - Implies ===> Each state reached has the minimal *g(n)*
- When *h* is admissible, monotonicity can be maintained when combined with pathmax: $f(n') = \max(f(n), g(n')+h(n'))$
  - Create new h -- $h'(n') = \max(f(n) - g(n'), h(n'))$
    $f(n') = g(n') + h'(n')$,
  - Force f to never decrease along path since $f(n') >= f(n)$

Does monotonicity in *f* imply admissibility?

Next lecture

- Finish off Discussion of A*
- IDA*
- Other Time and Space Variations of A*
  - RBFS
  - SMA*
  - RTA*