COSC1125/1127 Artificial Intelligence

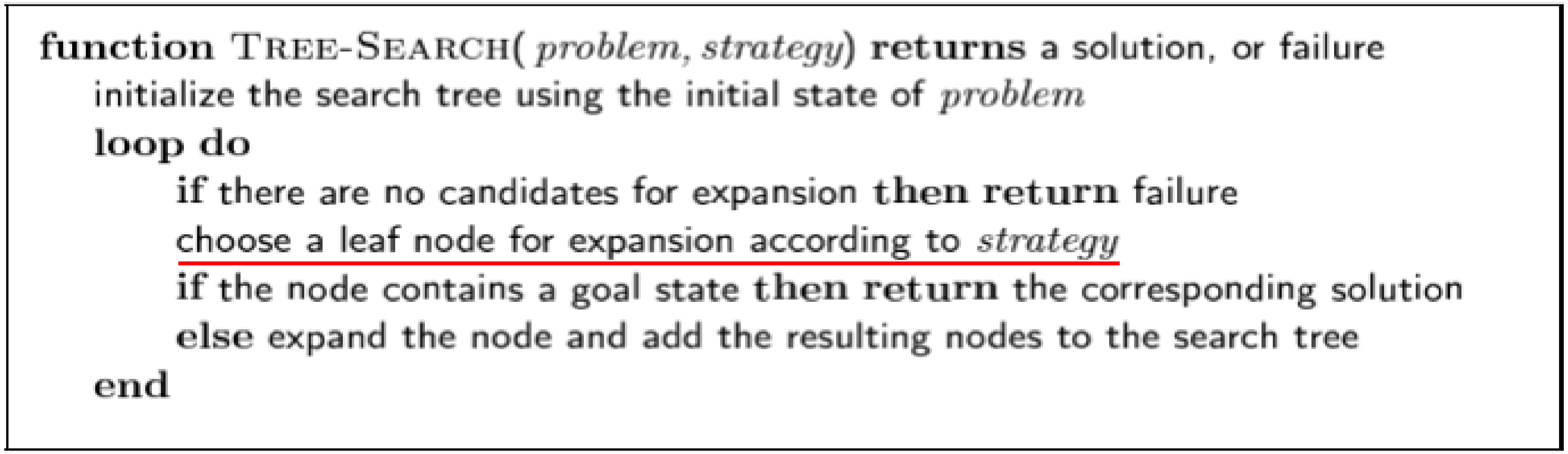
Week 2: Search Strategies

Readings:[RN2] Sec 4.1-4.2; or [RN3] Sec 3.5-3.6

Search Strategies

KEY CONCEPT: We want to find the solution while realizing in memory as few as possible of the nodes in the state space, which is represented as a search tree.

A strategy is defined by picking the order of node expansion during a search. The generalized search algorithm is shown as:



```
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
```

There are two main categories of search strategies: **Uninformed Search** and **Informed Search.**

Search Strategies…

Uninformed Search/Blind Search

- Can only distinguish goal and non-goal state
- Do not use information about number of steps from current state to goal
- The family members:
    - Depth First Search (DFS)
    - Breadth First Search (BFS)
    - Uniform Cost Search
    - Iterative Deepening Search
    - Bi-directional Search

Informed Search/Heuristic Search

- Use a guess/estimate of how far a state is from a goal in deciding which node to expand next
- The family members:
    - Greedy Search (or best-first search)
    - A* Search

[The word “Heuristic” derives from the Greek “Find”, “Discover”.]

Breadth First & Depth First Search

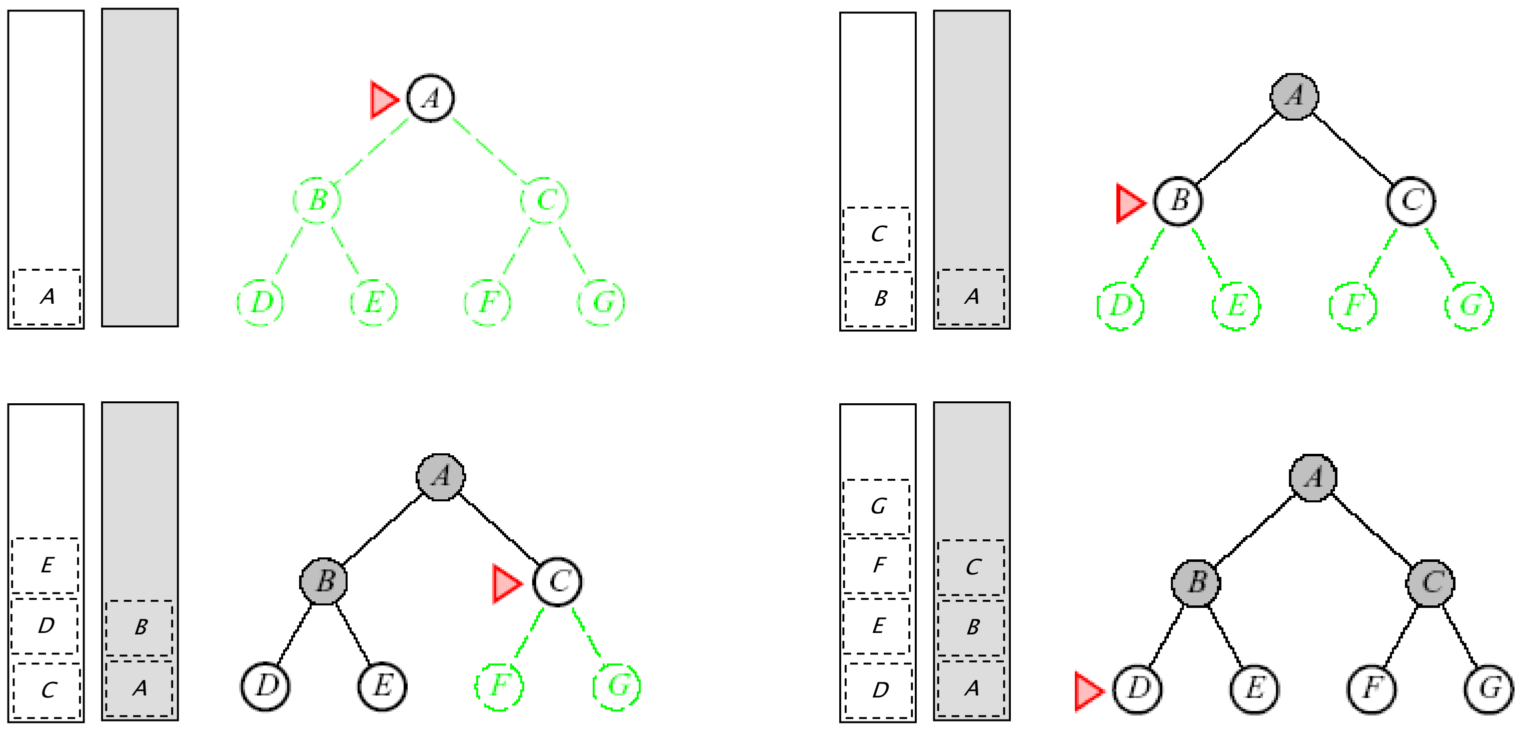
The pseudo code of a search algorithm

```
/* OPEN and CLOSED are lists of states. */
OPEN = Start node, CLOSED = empty

While OPEN is not empty do

   Remove the first state from OPEN, call it X

   If X is a goal return success

   Put X on CLOSED
   Generate all successors of X
   Eliminate any successors that are already on OPEN or CLOSED
   Put remaining successors on OPEN

End while
```

The algorithm is Breadth First Search (**BFS**) if the list OPEN is a queue (FIFO). Successors are added at **end** of the list.

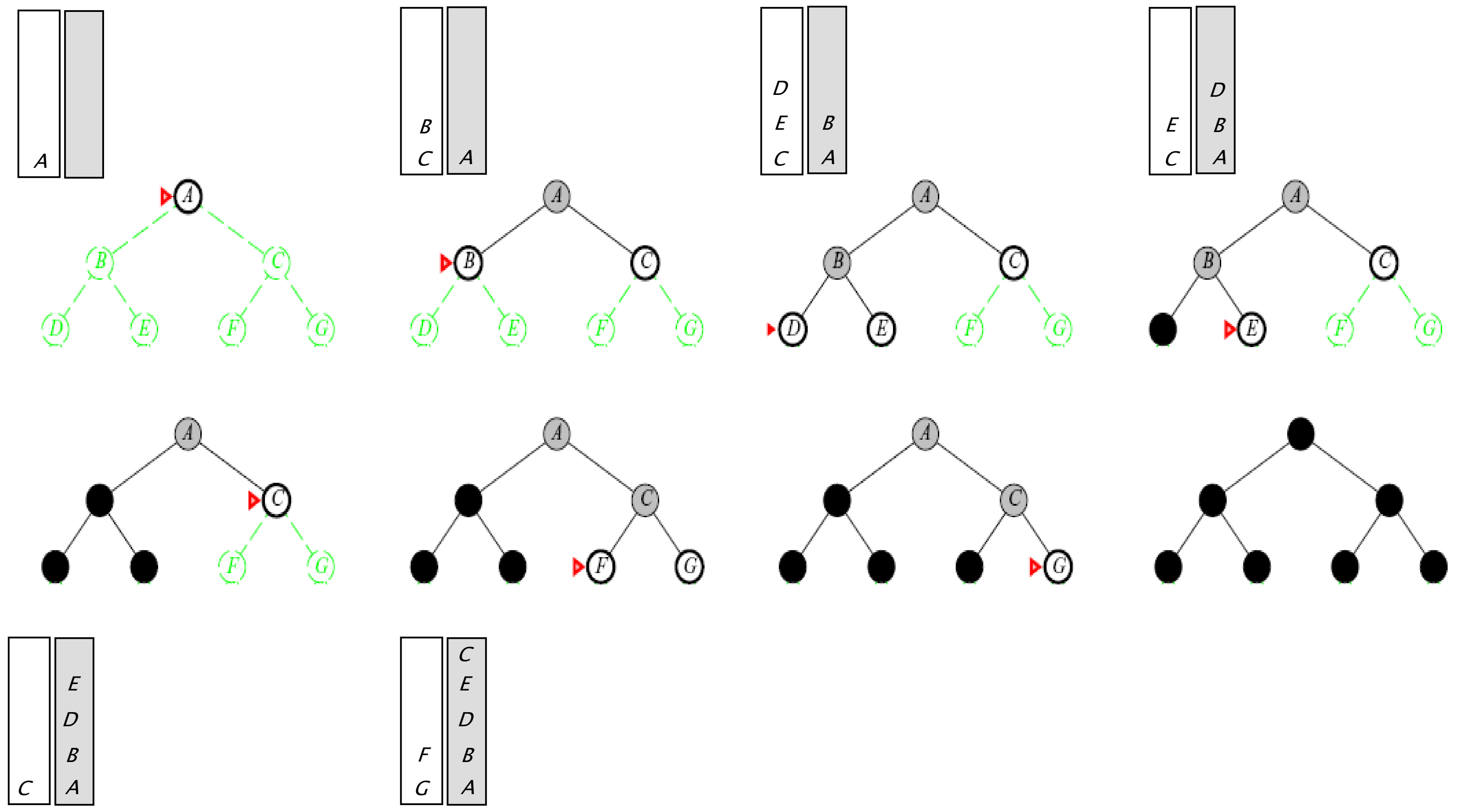
The algorithm is Depth First Search (**DFS**) if the list OPEN is a stack (LIFO). Successors are added at **front** of the list.

BFS Example



Nodes are explored in the order: A B C D E F G

DFS Example

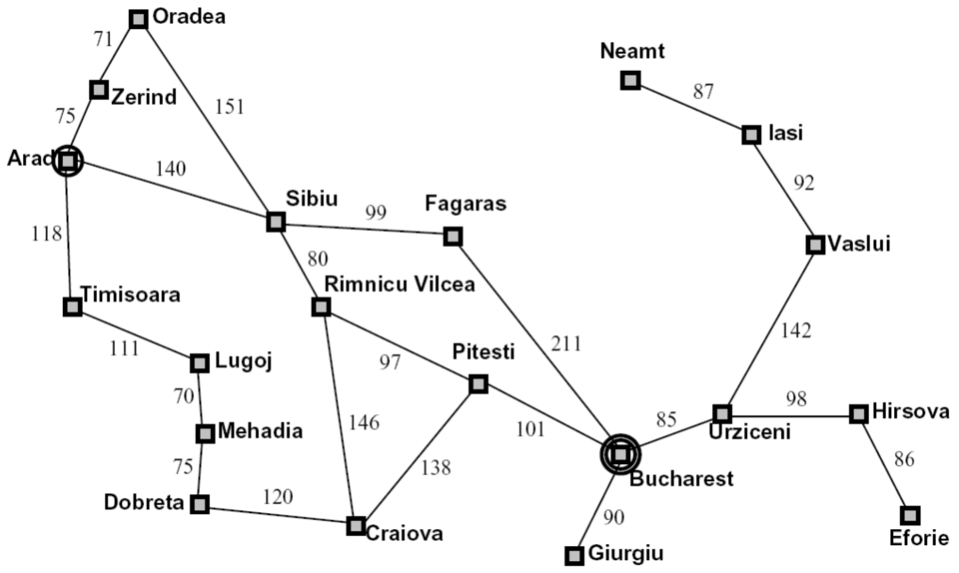


For the same search tree, nodes are explored in the order: A B D E C F G

[Compare this with the previous slide.]

BFS, DFS Example

Use travelling in Romania (from Arad to Bucharest) as an example.



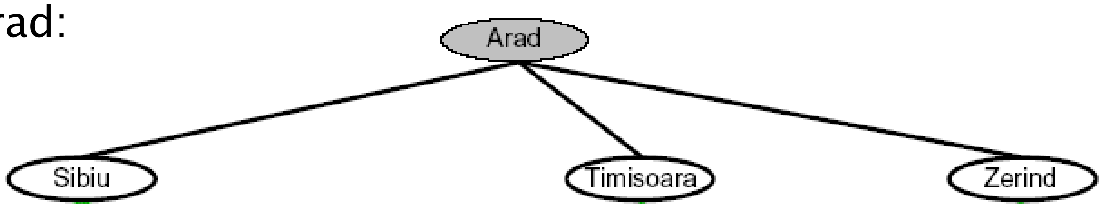
Question: how would you represent:

- The initial state
- The final state
- The operators
- Can the full state space be drawn?

BFS, DFS Example…

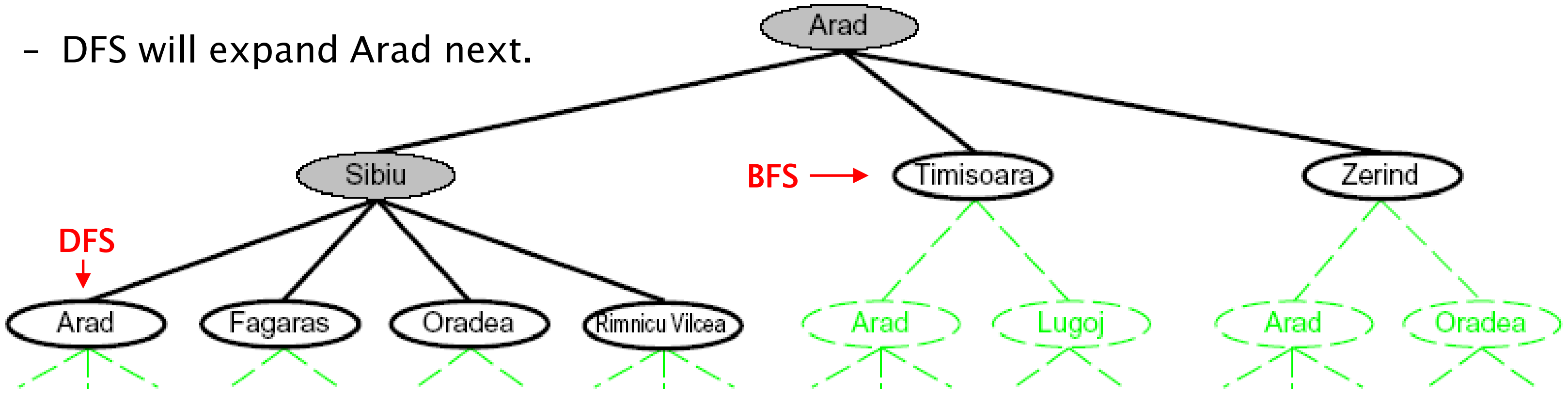
A state can be represented as a city. So the initial state is Arad.

After expanding Arad:



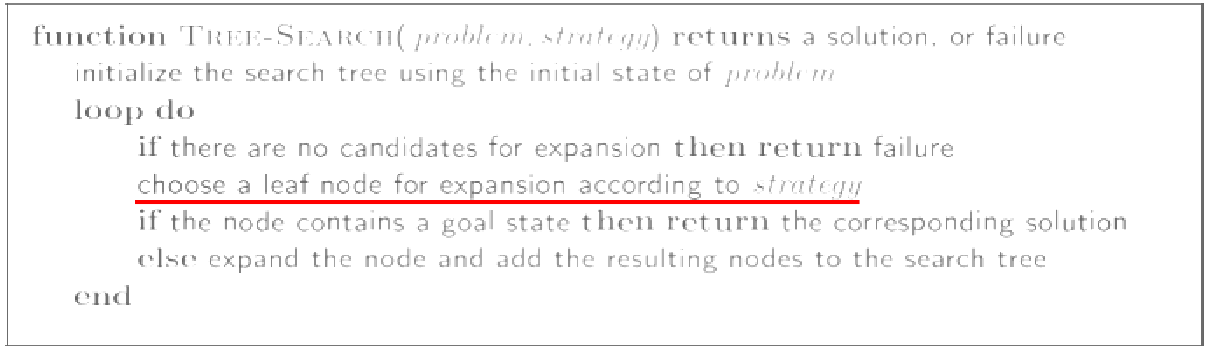
After expanding Sibiu:

- BFS will expand Timisoara next.
- DFS will expand Arad next.



Search Strategies

The breadth first search and depth first search are two cases of a general search algorithm.



```
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
```

- In breadth first search, the **next node at current level** is chosen for expansion.

- In depth first search, the **leftmost (unexpanded) node** is chosen for expansion.

- Other search algorithms use other selection methods to choose the node for expansion.

Before discussing other search algorithms, we will have a look at other aspects of search strategies: **data structures** and **evaluation** of search strategies.

Data Structures for Search

To facilitate a search algorithm, a data structure is needed for each node to maintain housekeeping information, which can be used during the search.

The information in a node includes:

- current state, the state in the search space to which the node corresponds,
- parent node of this node,
- operator applied to parent generate this node,
- depth of the node (number of links from root to this node)
- path cost (cost of the path from root to this node. More details later.)

Some searches, such as heuristic search algorithms, will require additional fields in the data structure.

Evaluation of Search Strategies

Search strategies are evaluated along the following four dimensions:

Completeness – whether this strategy is guaranteed to find a solution if one exists?

Time Complexity – how long does it take to find a solution?

Space Complexity – memory needed to perform the search.

Optimality – does the strategy find the optimal solution?
["the optimal solution" refers to a solution with the lowest path cost.]

Time and Space Complexity

Time and space complexity are measured in terms of

b – branching factor (or maximum number of successors) of any node

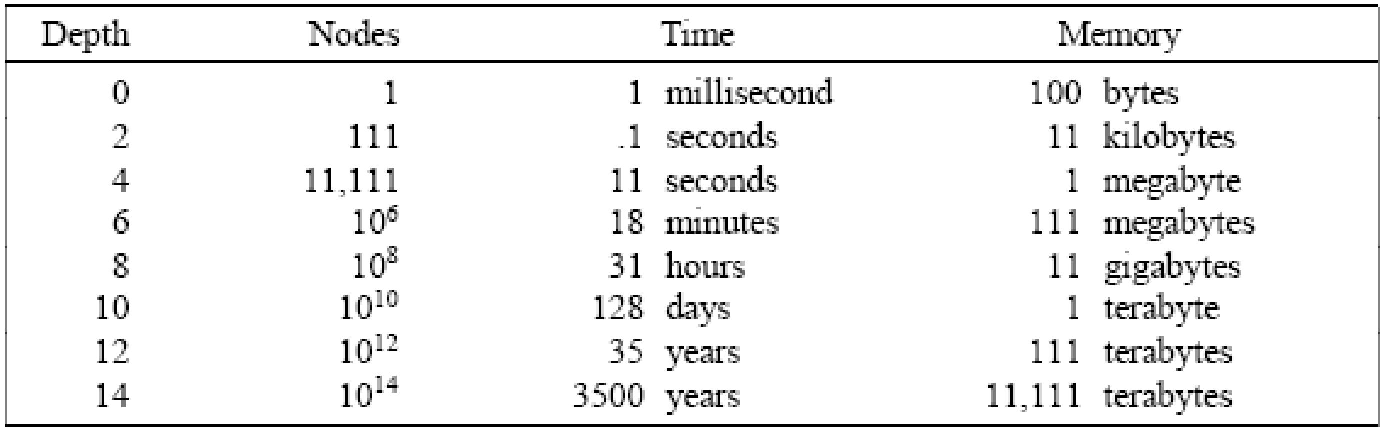
d – depth (of the shallowest goal node)

m – maximum depth of the state space graph (*m* may be ∞)

In case of breadth first search, the number of nodes expanded will be

$1 + b^1 + b^2 + b^3 + \ldots + b^d$

Suppose b = 10, the processing speed is 1000 node/sec and each node is 100 byte, the required time and space for different depth are:



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

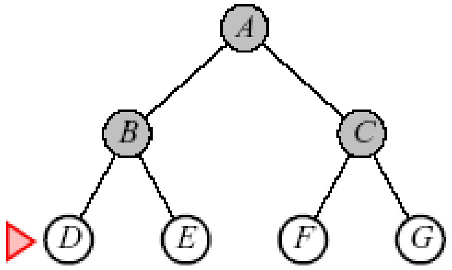
Properties of Breadth First Search

Completeness Yes, if *b* is finite.

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

Space Complexity $O(b^{d+1})$ [keeps every node in memory]

Optimality [Yes or No ?]



Question: do you think BFS can find the optimal solution first if there are several solutions?

Properties of Depth First Search

Completeness	No, if b is finite but m is infinite.
Time Complexity	$O(b^m)$. It is terrible if m is much lager than d
Space Complexity	$O(bm)$. It is linear! Only need to store $b \times m$ nodes.
Optimality	No. It might not find the shortest/optimal path. It might not recover from a bad early choice in a large/infinite search space.

Next, more search algorithms will be introduced and evaluated, such as Uniform cost search, Iterative deepening and Bi-directional search.

Uniform Cost Search

As mentioned before, a data structure of nodes should contain information of path cost.

In some problems the cost of expanding from one node to its successors might be different. Use route finding in Romania as the example, the distances from Arad to its three neighbouring cities are different.

The cost of each path should be maintained, which is the sum of the costs along the path.

Uniform Cost Search expands the node with the **smallest** path cost.

A function which assigns a cost to a path is called **path cost function**.

Path cost function is often denoted as $g()$.

Uniform Cost Search

Breath First Search can be viewed as a special case of Uniform Cost Search.

If the costs of going from one node to the successor nodes on next level are always the same, then Uniform Cost Search is equivalent to BFS.

In other words BFS is simply Uniform Cost Search where the cost function

$$g\,(\mathrm{n}) = depth\,(\mathrm{n})$$

Completeness: Yes

Optimality: Yes, because it selects least-cost unexpanded node.
So a solution with the shortest path can be found.

Question: how about the time/space complexity of Uniform Cost Search?

Iterative Deepening

As discussed previously

- BFS is complete and optimal, but has a big problem in space complexity.

- DFS is not complete and not optimal. However it has linear space complexity and could be faster than BFS.

Iterative deepening combines the benefits of BFS and DFS.

It performs Depth First Search with a depth limit.

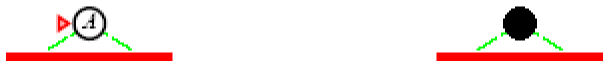
It tries all possible depth limits: first depth 0, then depth 1, depth 2 and so on.

It is an iterative process and the depth of search deeps at each iteration.

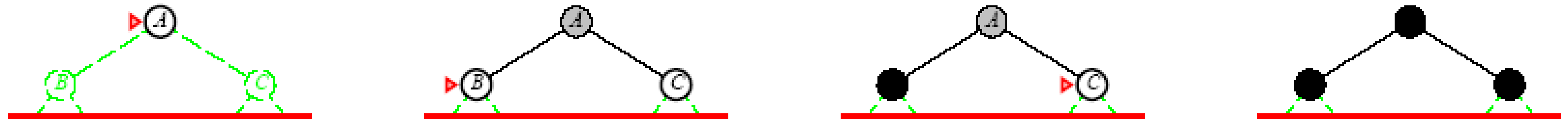
So it is called iterative deepening. See the illustration next.

Iterative Deepening

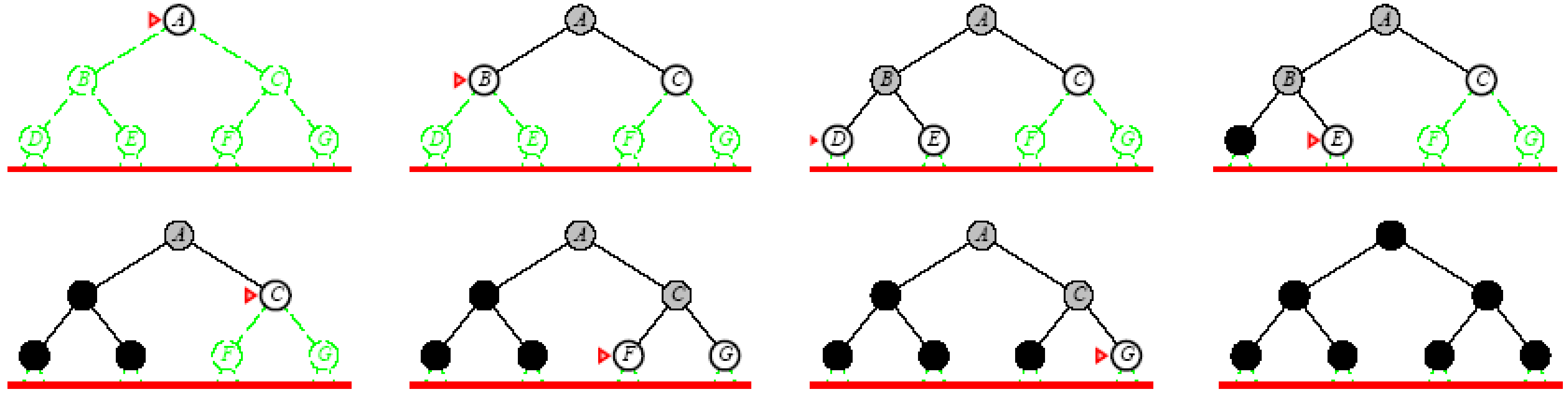
A DFS with depth limit = 0



A DFS with depth limit = 1

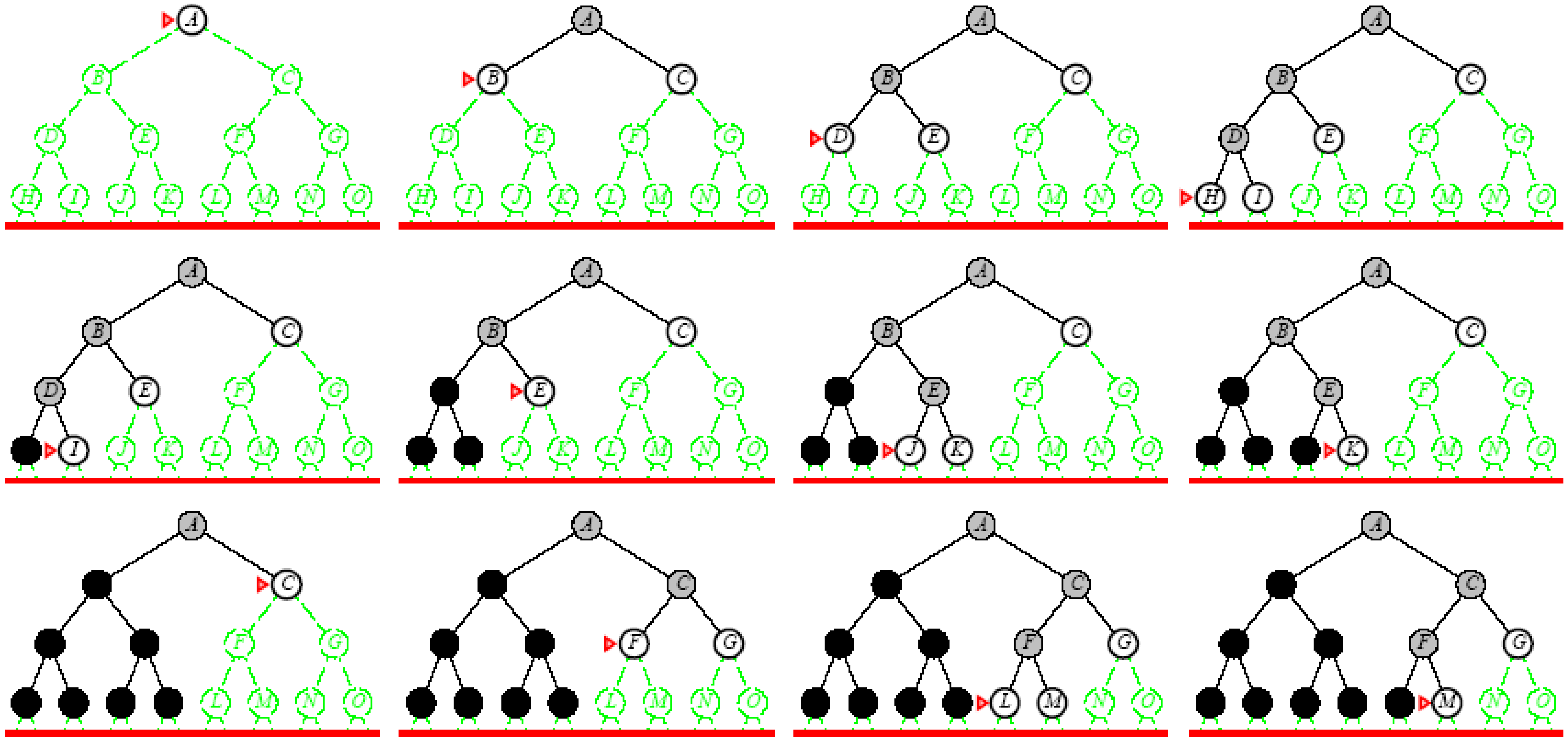


A DFS with depth limit = 2



Iterative Deepening

A DFS with depth limit = 3



Iterative Deepening

Iterative deepening combines the benefits of BFS and DFS.

It is complete and optimal.

It also has the linear space complexity as DFS has.

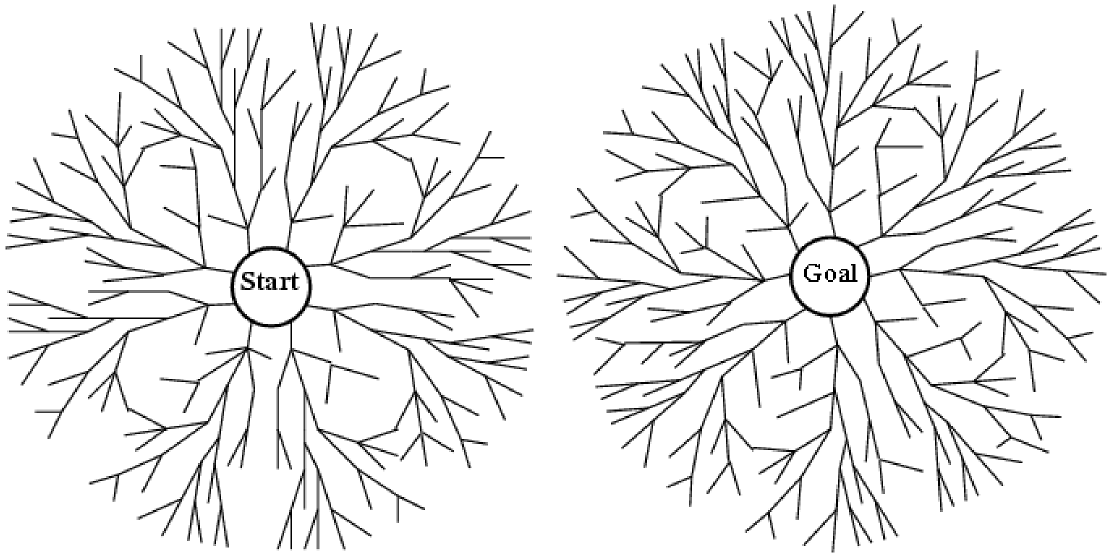
It may seem costly – many nodes are expanded multiple times.

However this is not a serious matter. Because most of the nodes on a tree are at the bottom level. It does not add much extra cost if the nodes on upper levels are expanded several times.

[Read more explanation in the text book.]

Bi-directional Search

Bi-directional search simultaneously search both forward from the initial state and backward from the goal. It stops when two searches meet in the middle.



The complexity is $O(b^{d/2})$ because the forward search and backward search each have to go only half way.

However there are several issues of bi-directional search.

Bi-directional Search

Issues of Bi-directional search:

- Bi-directional search must be able to generate predecessors to enable backward search. However in some problems that can be very difficult.

- How to deal with many possible goal states? Ideally the problem should have just one or a few goal states.

-There must be an efficient way to check whether each new node has already been visited from the other half of the search.

- Also we need to decide what kind of search is going to take place in each half of the search.

-In order to check whether the two search meet, the nodes of at least one of them must be retained in memory. So the space complexity is $O(b^{d/2})$, which might be impractical.

Repeated States

In the cannibals and missionaries problem, examples of repeated states (states already visited) have been shown. However the search algorithms so far don't deal with repeated states.

Avoiding repeated states is an important aspect of search process.

Various choices can be made depending on the search space.

- Not check for repeated states at all.

- Don't return to the state we just came from.

- Don't create paths with cycles in them.

- Don't generate any state that has been already generated.

Question: What would be a good data structure to use for checking for repeated states?

Constraint Satisfaction

In some problems the states are defined by the values of a set of variables and these values must obey certain **constraints**.

This kind of problem is known as **constraint satisfaction problem** (CSP).

The cannibals and missionaries problem is an example of CSP. The constraint is on either bank of the river

the number of missionaries >= the number of cannibals, or

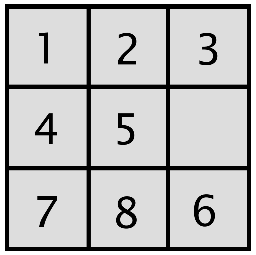
the number of missionaries = 0.

CPSs can be solved by general-purpose search algorithms.

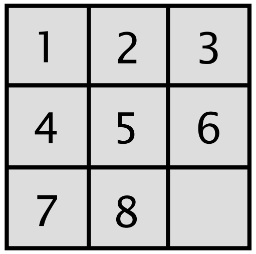
Because of CPS's special structure, algorithms designed specifically for them generally perform much better.

Heuristic Search

Suppose the problem is to solve the following 8-puzzle problem:

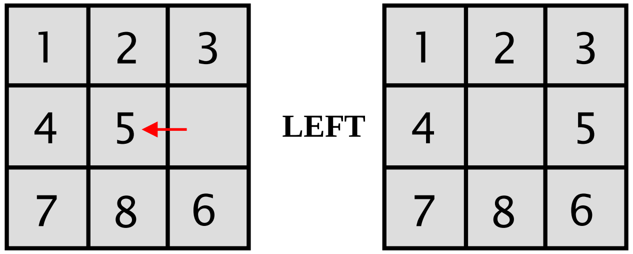


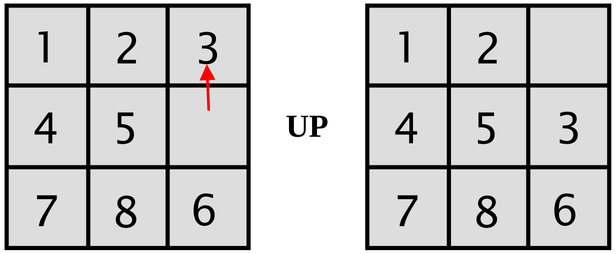
Initial State



Goal State

It can be solved by only one move. However if a depth first search is unlucky it could pick LEFT or UP to try next (move the blank to left or up).





This would result in worse positions than the starting position.

The search might never try DOWN ↓ and find the right move.

Heuristic (Informed) Search

To efficiently solve a problem, we can force the search to try the best or the most promising move first.

In the case of the last 8-puzzle, we can force DOWN to be the first move.

How can we do that? We can use some “domain knowledge” or “heuristic” information to guide the search.

This kind of search strategies are called informed search or heuristic search.

Heuristic search applies **a heuristic function *h* (*n*)** to a node *n*. It measures “how promising the node is”, or the “desirability” of the node.

The lower the heuristic value, the more promising or more desirable the node is.

Heuristic Functions

For 8-puzzles, useful heuristic functions could be

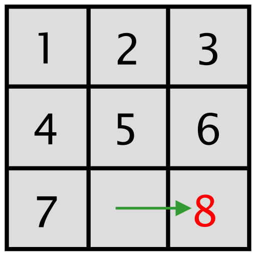
- **Tiles out of place**: number of tiles which are misplaced. We call it h_1.
- **Manhattan distance** from current state to the goal state. We call it h_2.

  Manhattan distance is also known as **city block distance.**

For example, if the goal state is:

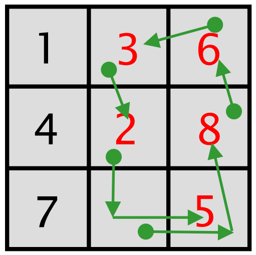
1	2	3
4	5	6
7	8	

Then the heuristic values of the above functions on these two states are:



$h_1 = 1$

$h_2 = 1$



$h_1 = 5$

$h_2 = 7$

Pseudo Code of Informed Search

```
/* OPEN and CLOSED are lists of states. */
OPEN = Start node, CLOSED = empty

While OPEN is not empty do

   Remove the first state from OPEN, call it X

   If X is a goal return path from start to X

   Put X on CLOSED
   Generate all successors of X

   for each successor of X do
       DEAL_WITH_IT( successor )
       Re-order the nodes on OPEN by value of f
   End for


End while
```

Pseudo Code of Informed Search...

```
DEAL_WITH_IT ( successor )
    case: the state of successor is not on OPEN or CLOSED
          assign the successor a value by function f(successor)
          add successor to OPEN
    case: successor is already on OPEN
          if its state was reached by a shorter path here
          then give the state on OPEN the shorter path
    case: successor is already on CLOSED
          if its state was reached by a shorter path here
          then
             remove the state from CLOSED
             update path
             add the successor to OPEN
    end case
```

Evaluation Function

The function $f(\mathrm{n})$ in pseudo code of informed search is evaluation function.

An evaluation function can be applied to any node.

Node on OPEN are sorted by return value of the evaluation function.

The node with the smallest evaluation value is expanded next.

Different choice for $f(\mathrm{n})$ give different kinds of searches.

If $f(\mathrm{n}) = h(\mathrm{n})$ [The evaluation function is a heuristic function]
then it is a greedy search.

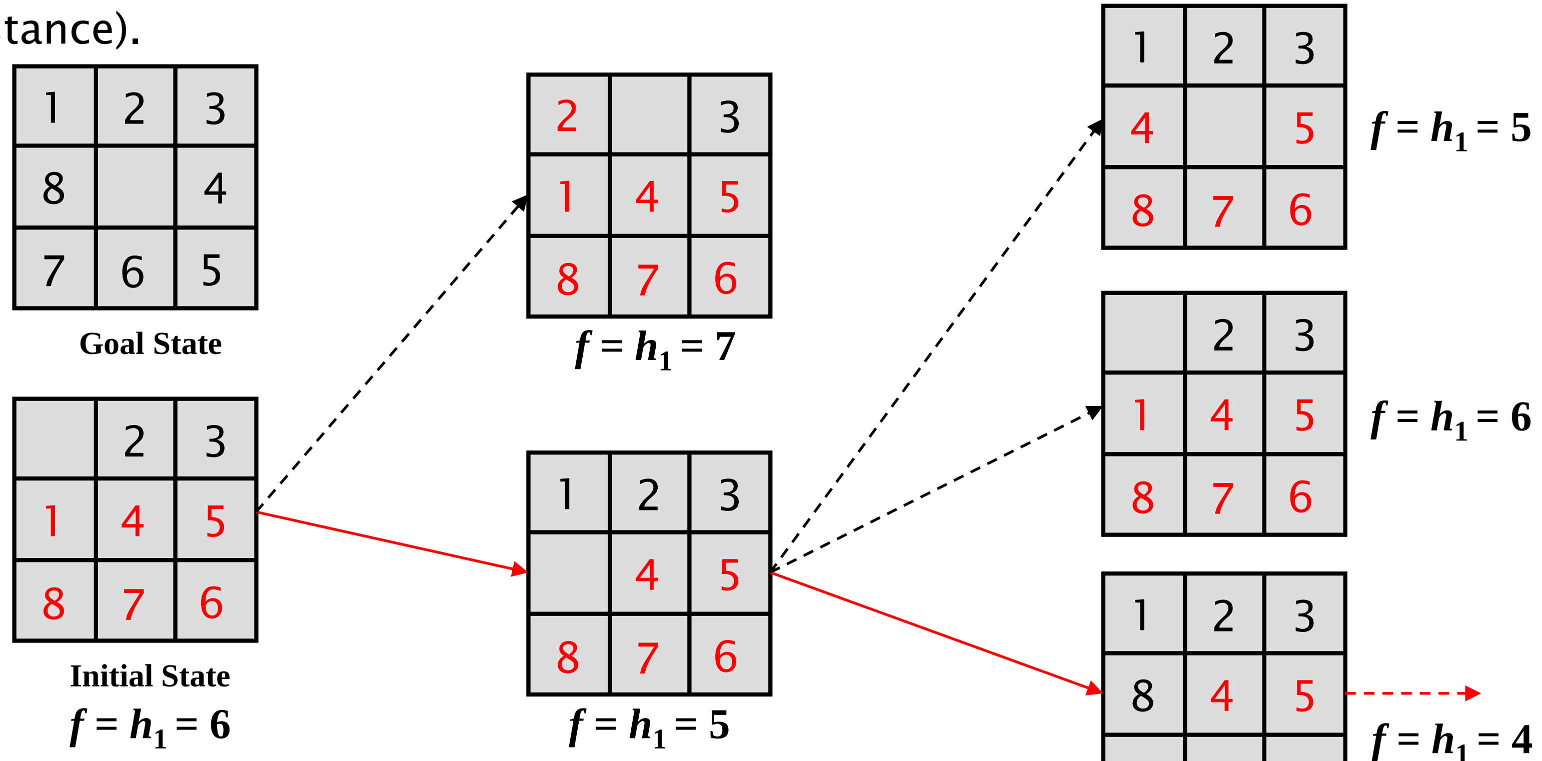
If $f(\mathrm{n}) = h(\mathrm{n}) + g(\mathrm{n})$ [Combines heuristic function and path cost function]
then it is an A* search.

Question: if $f(\mathrm{n}) = g(\mathrm{n})$, what kind of search it is?

Greedy Search

Greedy search uses heuristic function as evaluation function.

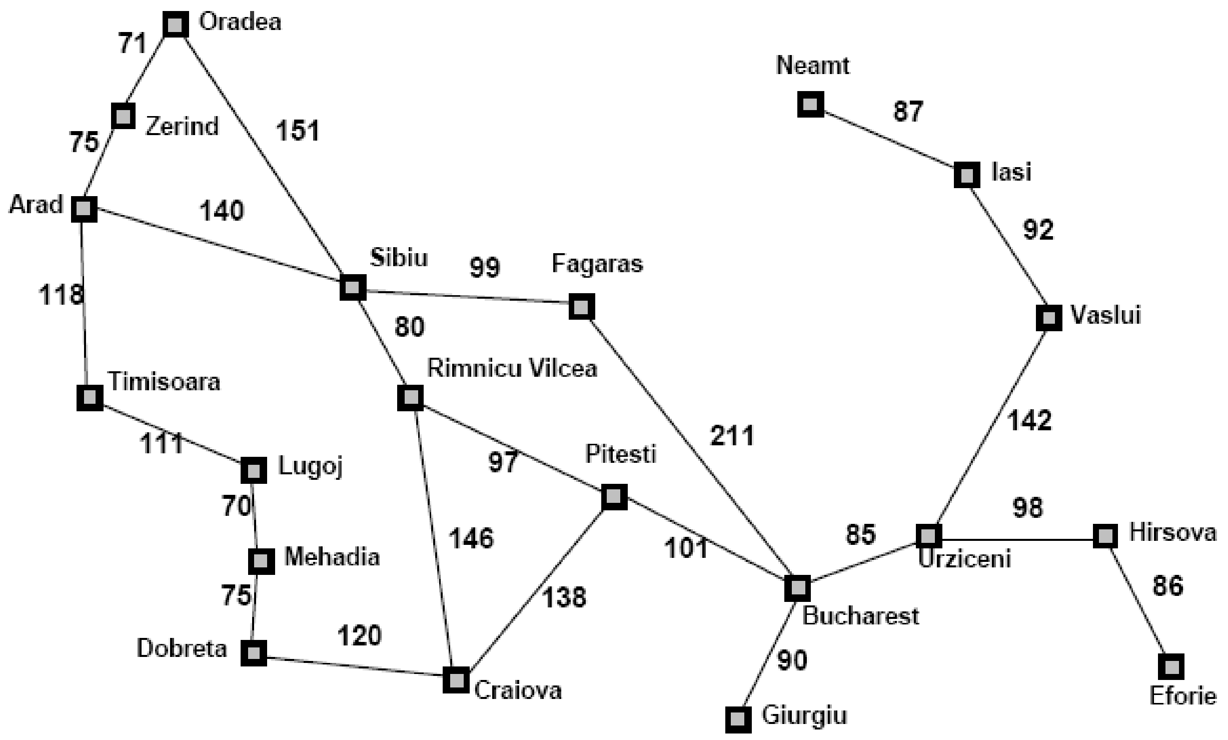
For example 8-puzzle problem could use h_1 (tiles out of place) or h_2 (Manhattan distance).



This example uses h_1 as evaluation function.

Greedy Search

Use route finding in Romania as another example. Suppose the straight-line distance from each city to Bucharest is known. We can use this distance as heuristic value.

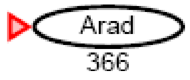


Straight−line distance to Bucharest

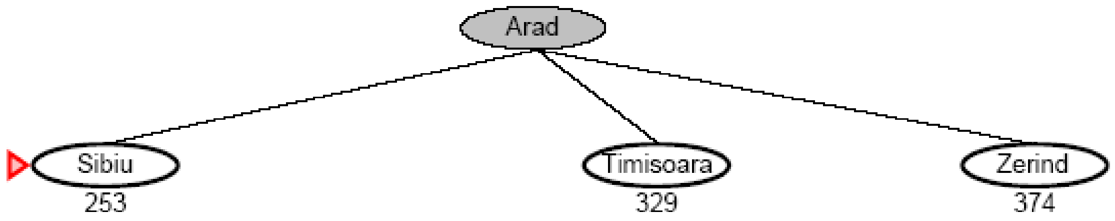
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Search

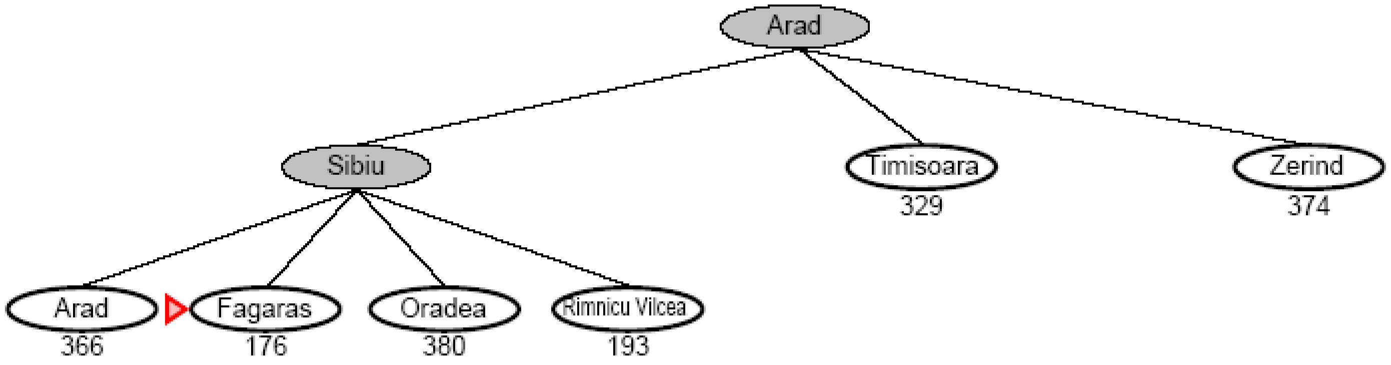
Depth 0: the heuristic value is under the node



Depth 1:

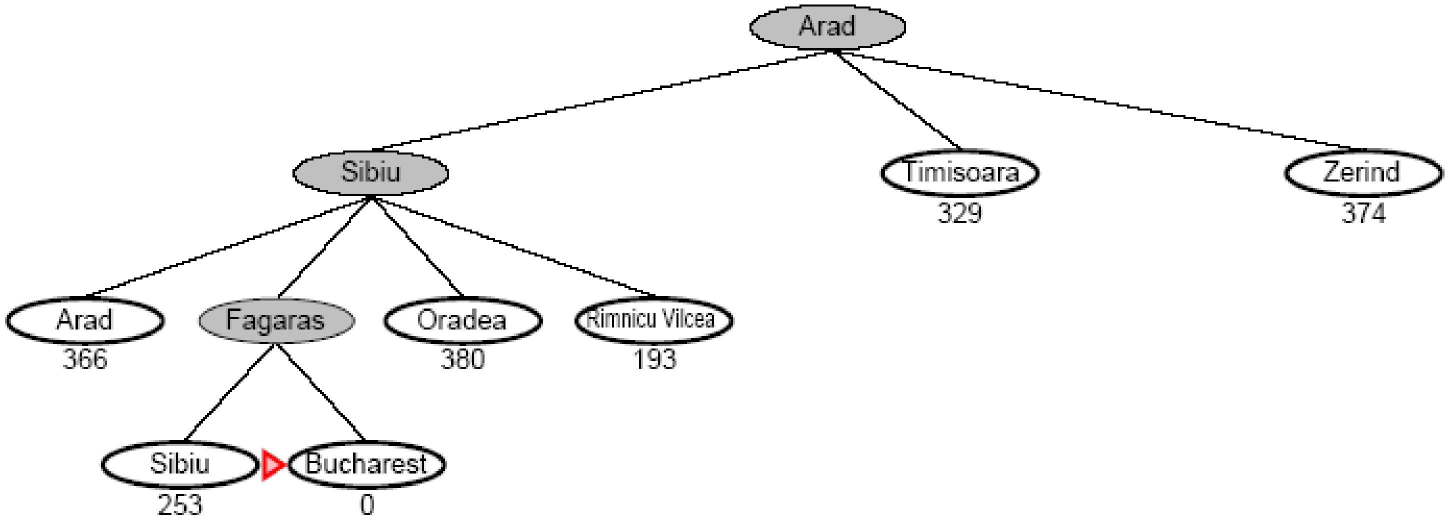


Depth 2: Sibiu is expanded because it has the least heuristic value.



Greedy Search

Depth 3:



Greedy search is not optimal. Like depth first search, it prefers to follow a single path all the way to goal. The path might not be the shortest one.

Greedy search is complete only if repeated states can be removed. Otherwise it can get stuck in loops.

A* Search

A* search uses heuristic function and path cost function as evaluation function.

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

The idea is try to expand a node that is on the least cost path to the goal.

$f(n)$ is the estimated cost of the cheapest solution through n.

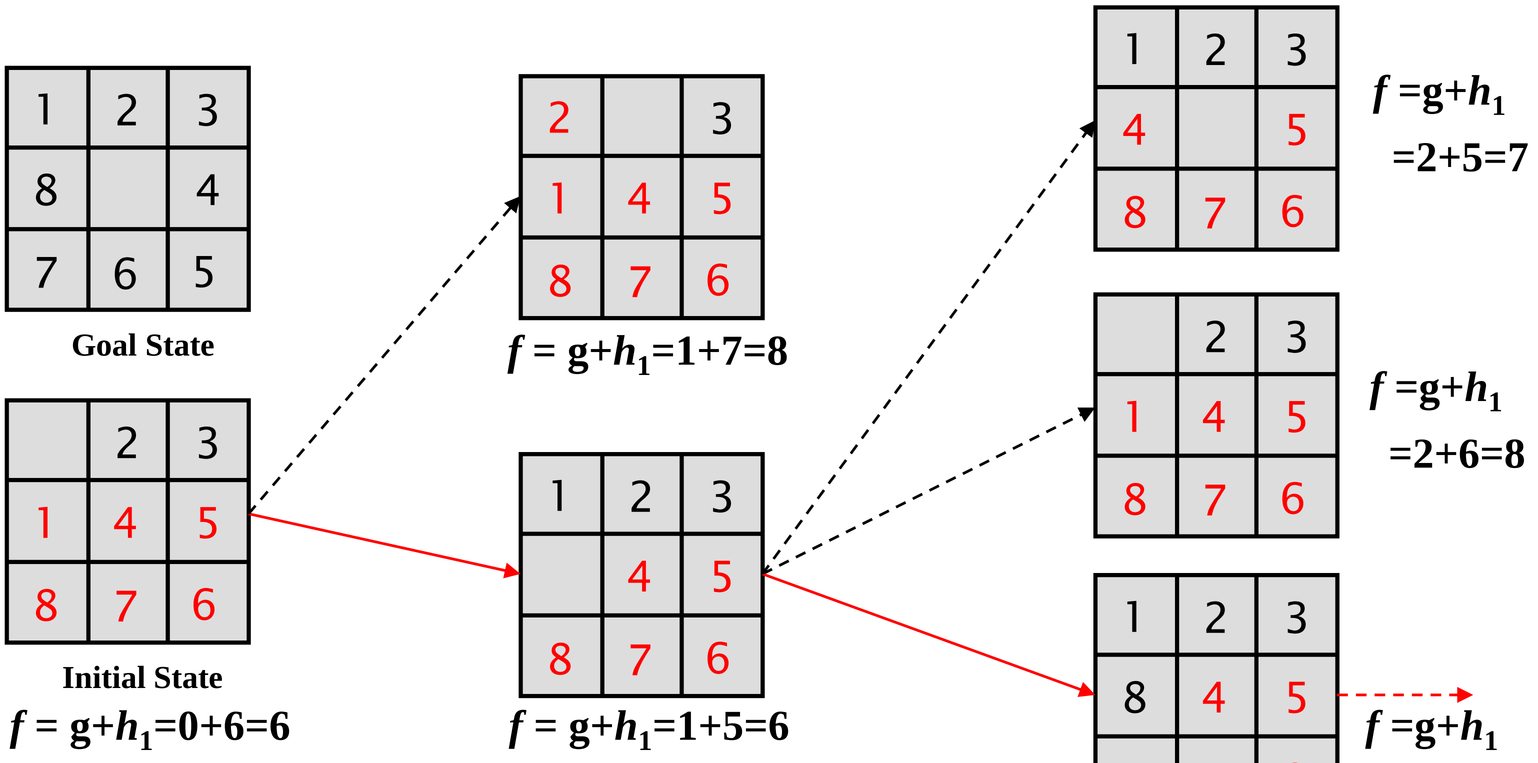
A* search is complete.

A* search is optimal.

A* search is Optimally Efficient, i.e., no other algorithms using $h(n)$ is guaranteed to expand fewer states than A*.

A* Search

A* search in 8-puzzle.

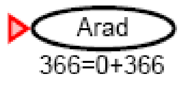


This example uses g + h_1 as evaluation function.

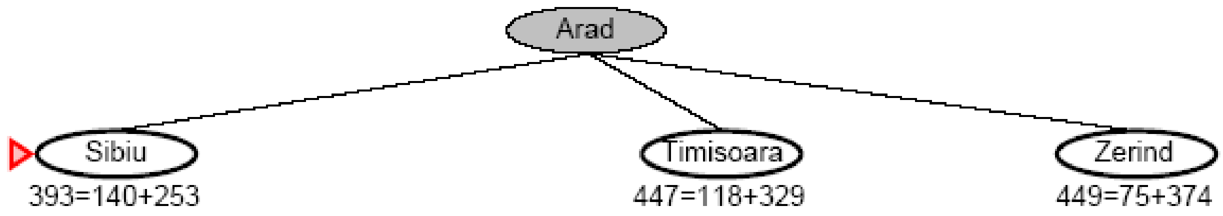
A* Search

A* search in route finding problem.

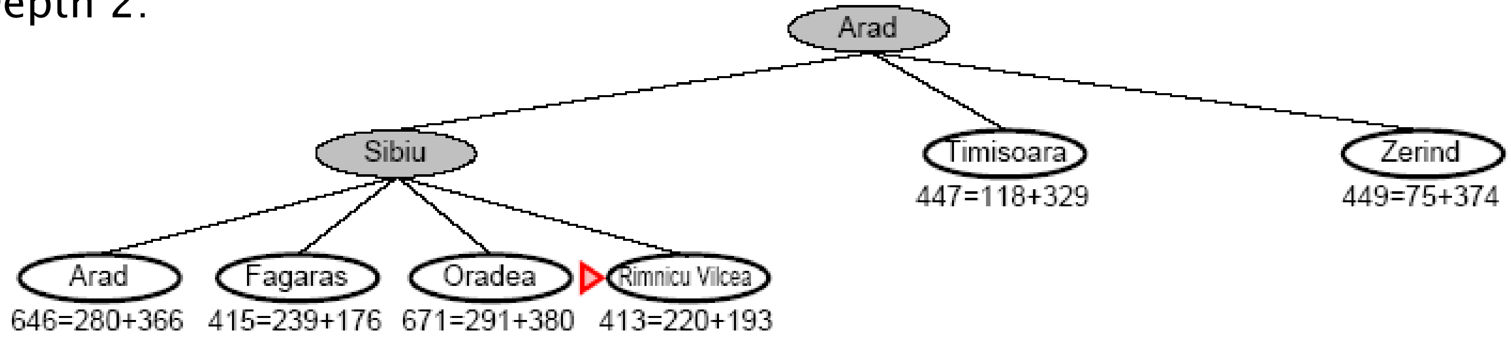
Depth 0:



Depth 1:



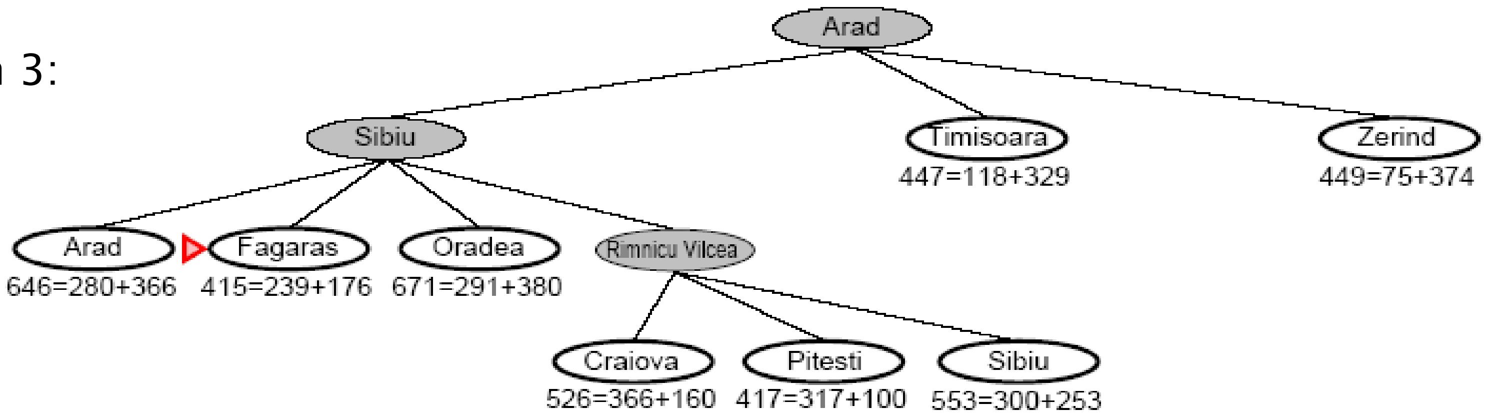
Depth 2:



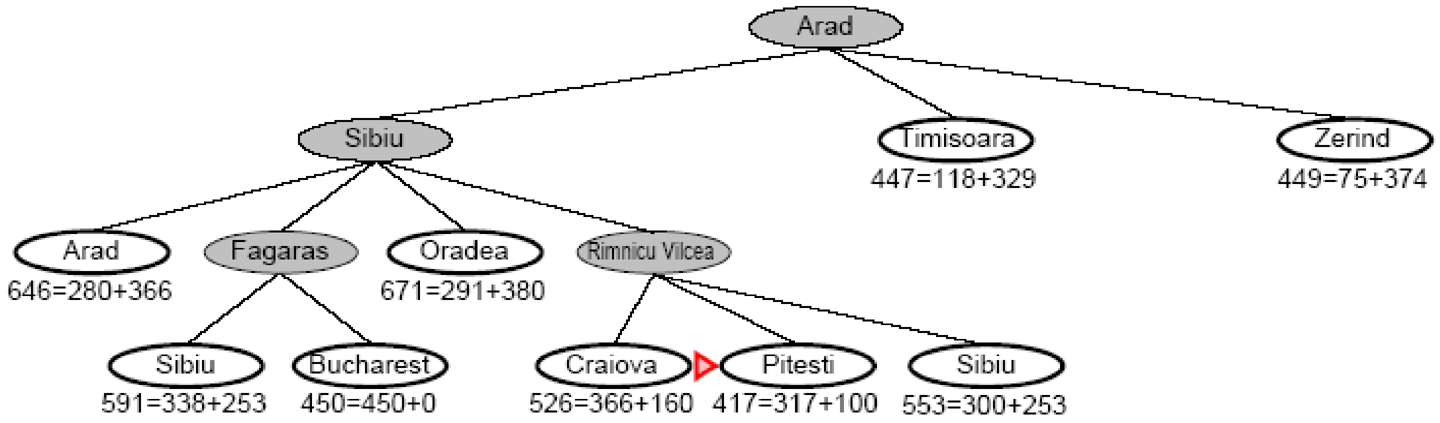
The next node will be expanded is Rimnicu Vilcea, not Fagaras.

A* Search

Depth 3:



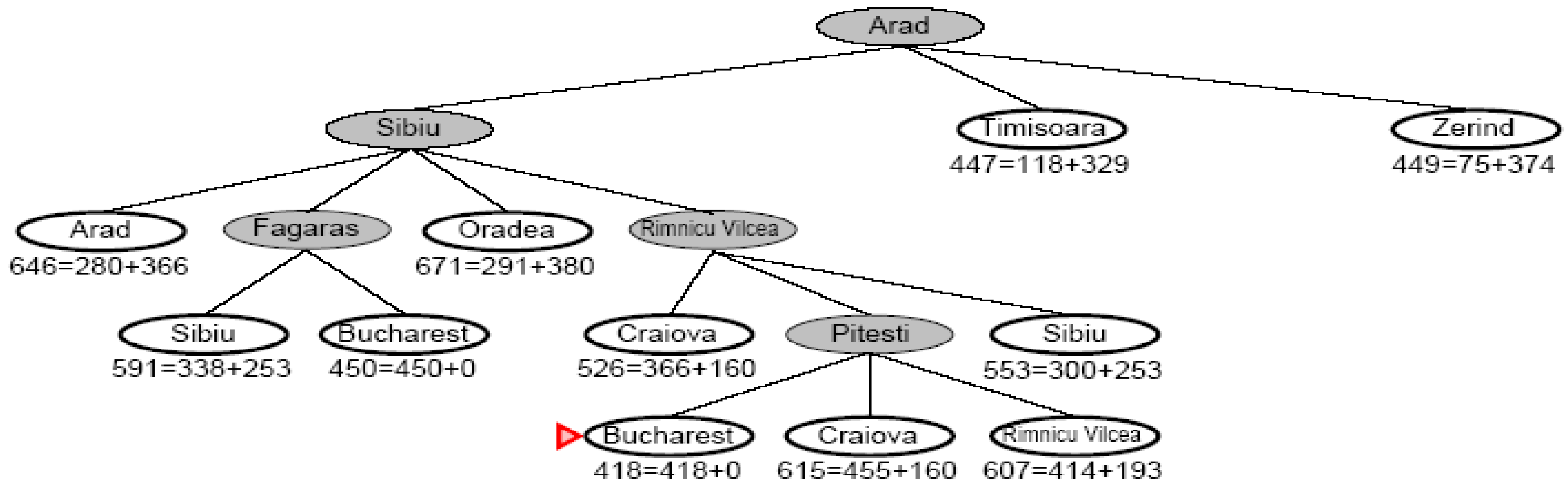
The next node will be expanded is Fagaras, because now it has the least evaluation value among the unexpanded nodes.



Pitesti will be expanded next. Its value is even lower than Bucharest.

A* Search

Depth 4:



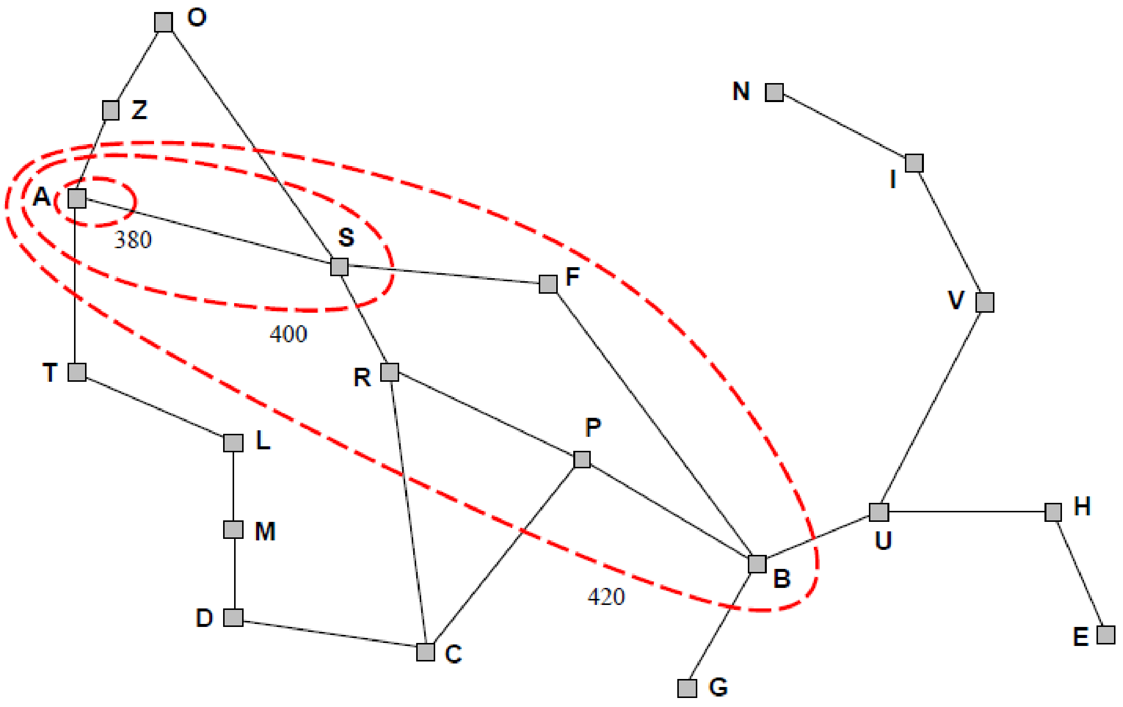
The second Bucharest has the least evaluation value and it is the destination city. So the route is Arad – Sibiu – Rimnicu Vilcea – Pitesti – Bucharest.

It is the optimal solution.

It is cheaper than route Arad – Sibiu – Fagaras – Bucharest.

A* Search

- A* expands nodes in order of increasing *f* value;
- Gradually adds "*f*-contours" of nodes (cf. BFS adds layers);
- Nodes inside a given contour have *f* costs less than or equal to the contour value, e.g., inside contour labelled 400, all nodes have *f(n)* less than or equal to 400.



Admissible Heuristic

Admissible heuristic is a heuristic that **never overestimates** the cost to the goal.

The straight line distance for the route finding problem is also admissible.

In 8-puzzle h_1 and h_2 are both admissible. Is one better than the other? “Better” means that fewer nodes will be expanded in searches.

If we have two heuristic functions and $h_X(n) >= h_y(n)$ for every node n, we say $h_X(n)$ **dominates** $h_y(n)$ or $h_X(n)$ is **more informed** than $h_y(n)$.

Intuitively, if both heuristics are underestimates then $h_x(n)$ is a more accurate estimate than $h_y(n)$.

More Informed Heuristic

In the 8–puzzle, h_2 is more informed than h_1.

Using $f = g + h_2$, a search expands less nodes.

Using $f = g + h_1$, it expands every state expanded by $f = g + h_2$ and a few more.

Using a more informed heuristic is guaranteed to expand fewer nodes of the search space and to be more computationally efficient.

Question: can you show that “Manhattan distance” dominates “Tiles out of place”?

Next week we will look at a very intriguing area of search – game playing.

Summary

- search, how to formulate search problem
- state, operators and state space
- representation of search problems
- search tree and search strategy
- Uninformed search and informed search
- DFS, BFS, Uniform Cost Search, Iterative Deepening, Bi-directional Search
- evaluation of search strategies: completeness, time/space complexity, optimality
- constraint satisfaction
- heuristic function & evaluation function
- Greedy Search, A* search
- admissible heuristic

Start working on Assignment#1 now.

Acknowledgement: the slides were developed based on notes from Russell & Norvig, by several computer science staff members over the years.