Problem Solving Via Search

UNIVERSITI TUNKU ABDUL RAHMAN

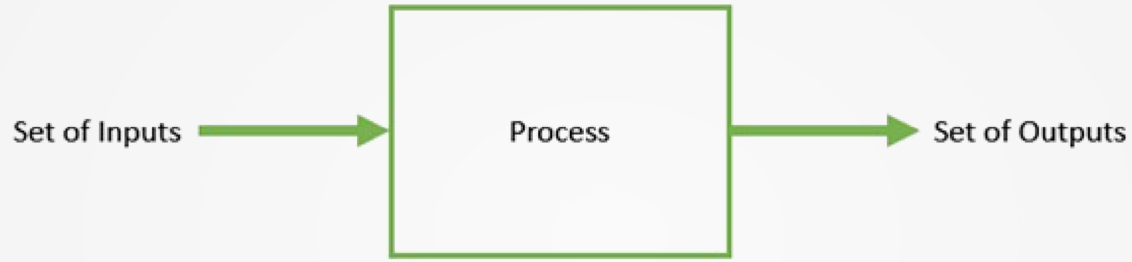
Learning Objectives

After completing this lecture, you will be able to:-

- Explain basic problem-solving terminology
- Describe the strengths and weaknesses of metaheuristics
- Explain the general workings of genetic algorithms as a problem-solving metaheuristic
- Describe the main motivations for using genetic algorithms

Optimization

- **Optimization** is aimed at making something better (solving a problem)



- The result of optimization is a set of ‘best’ inputs/values/parameters (as judged by the process’ output)
- The definition of ‘best’ depends on the problem
  - Generally a maximum/minimum of some kind

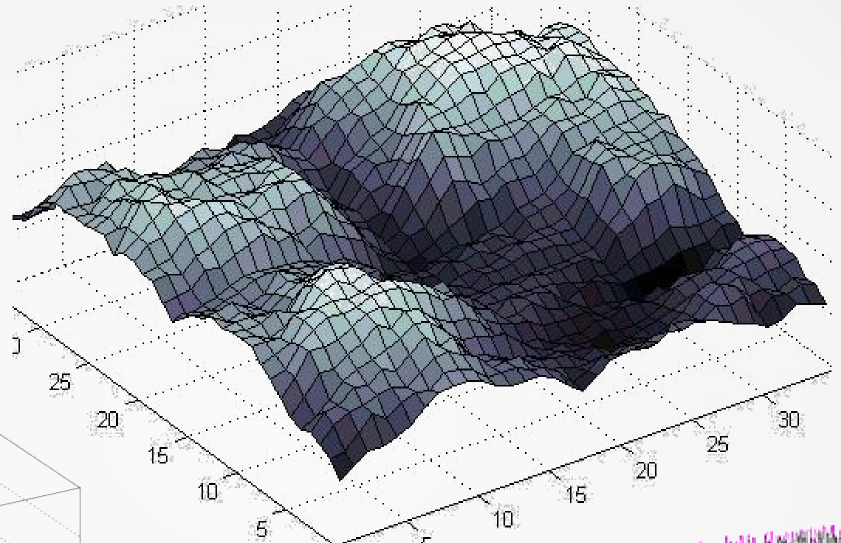
Objective Function

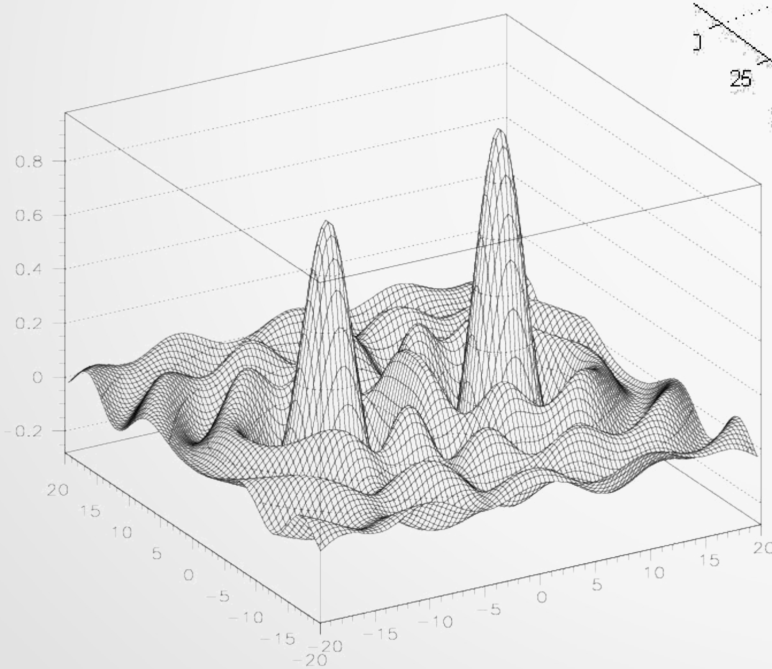
- Any problem being optimized has some **objectives**
  - Minimal travel time, minimal cost, maximum profit
- The mathematical function describing this objective is called the **objective function** (or the **cost function**)
- The **objective function** depends on one or more **decision variable** which are the inputs to the process

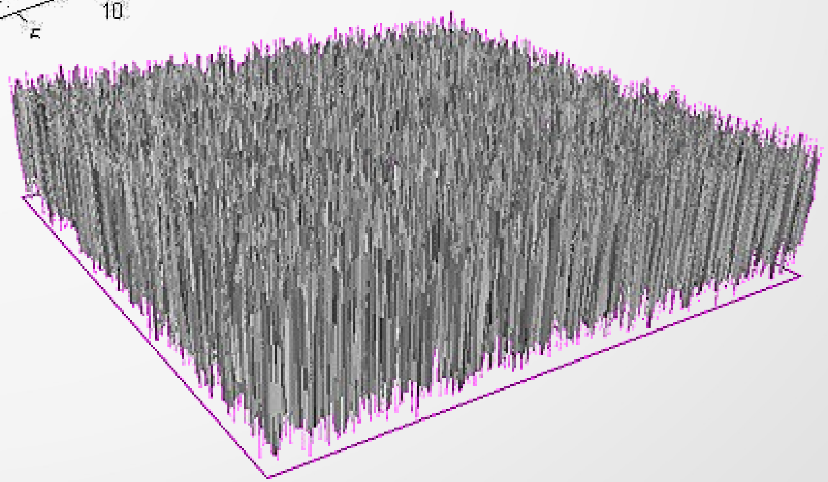
Search Space

- Solving a problem can be represented as a search task
  - ‘Find’ the solution
- The **search space** is then a set containing *all possible solutions* to the problem
- Optimization would therefore involve finding one (or a few) solutions in this search space
  - A good solution would maximize/minimize the objective function

Search Space



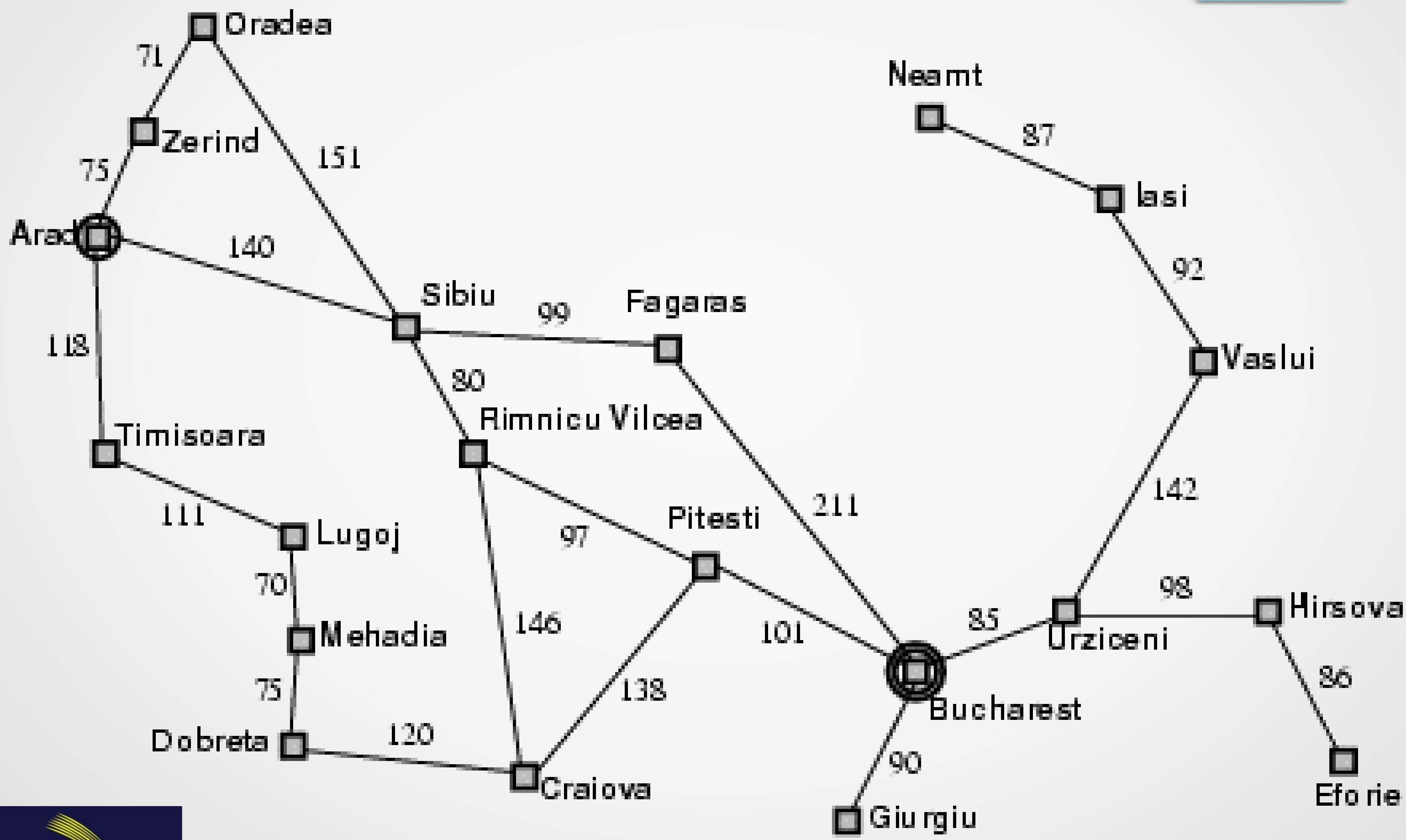




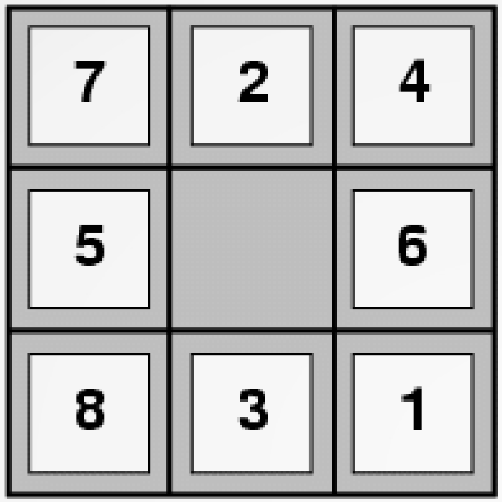
Example Search Space: Romania

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest
- **Formulate goal**:
  - Be in Bucharest
- **Formulate problem**:
  - *states*: various cities
  - *actions*: drive between cities
- **Find solution**:
  - Sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

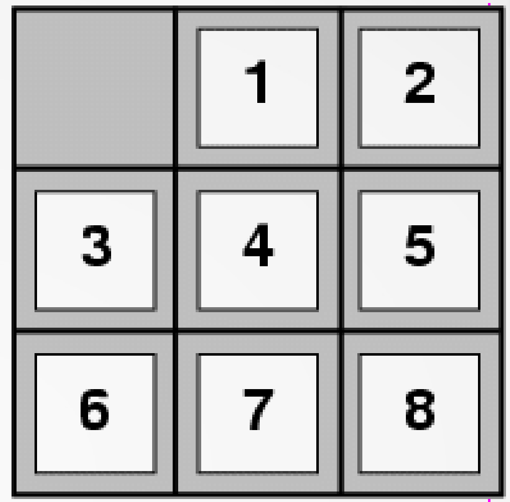
Example Search Space: Romania



Example Search Space: The 8-puzzle



Start State



Goal State

- States? - locations of tiles
- Actions? - move blank left, right, up, down
- Goal Test? - goal state (given)
- Path Cost? - 1 per move

Note:
Optimal solution of N-Puzzle family is NP-hard!

Algorithms vs Heuristics

In the context of problem-solving:-

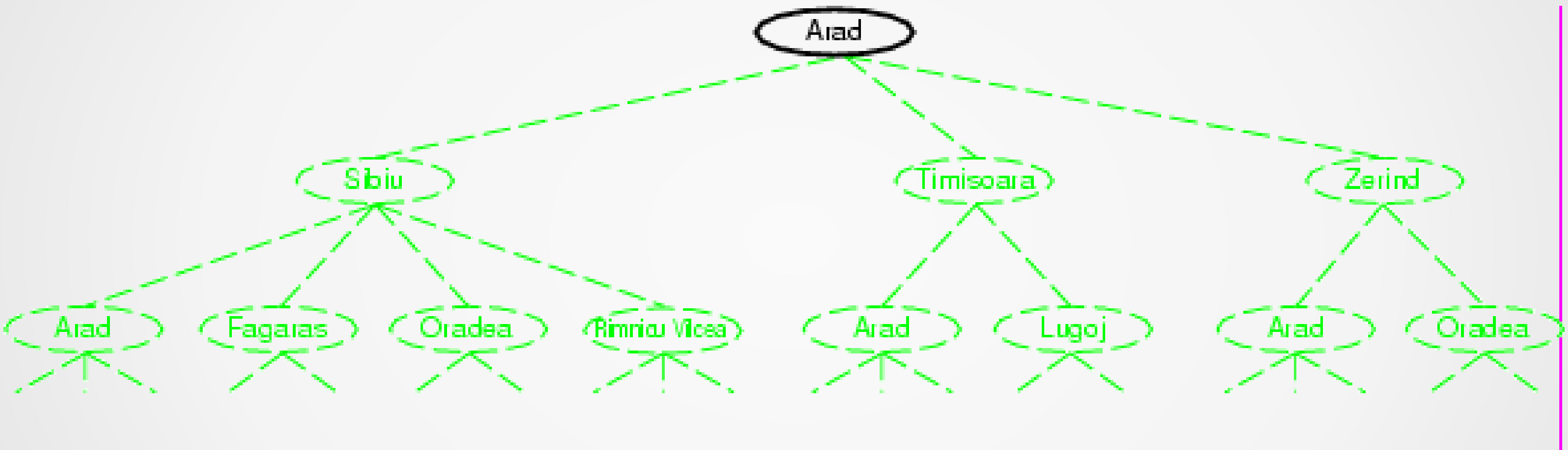
- An **algorithm** specifies *unambiguously* how to solve a problem
- A **heuristic** is a technique for solving a problem quickly
  - A shortcut, trades off exactness/accuracy/precision
  - Can be based on intuition, experience, rules-of-thumb
  - Can be very reliable (provably *admissible*) or more than useless (actively harmful)

Tree Search Algorithms

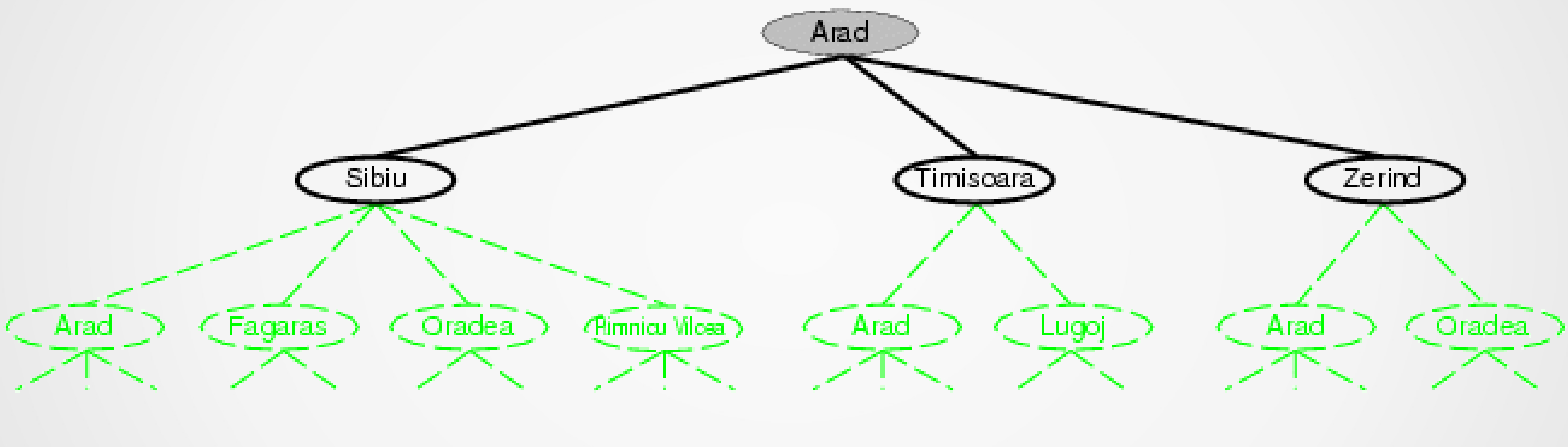
- Basic idea:
  - Offline, simulated exploration of state space by generating successors of already-explored states (a.k.a **expanding** states)

```
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
```

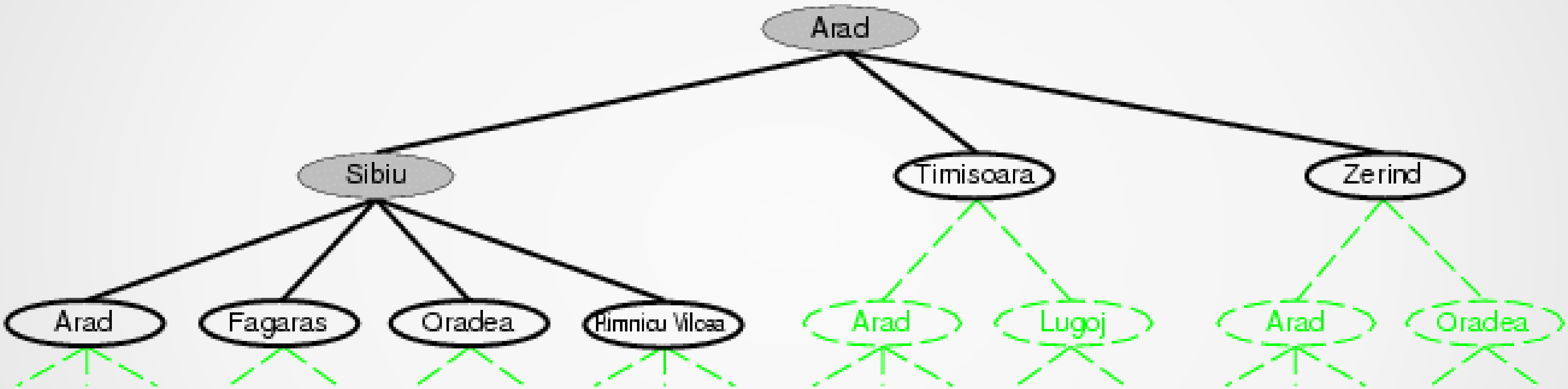
Tree Search Example



Tree Search Example



Tree Search Example



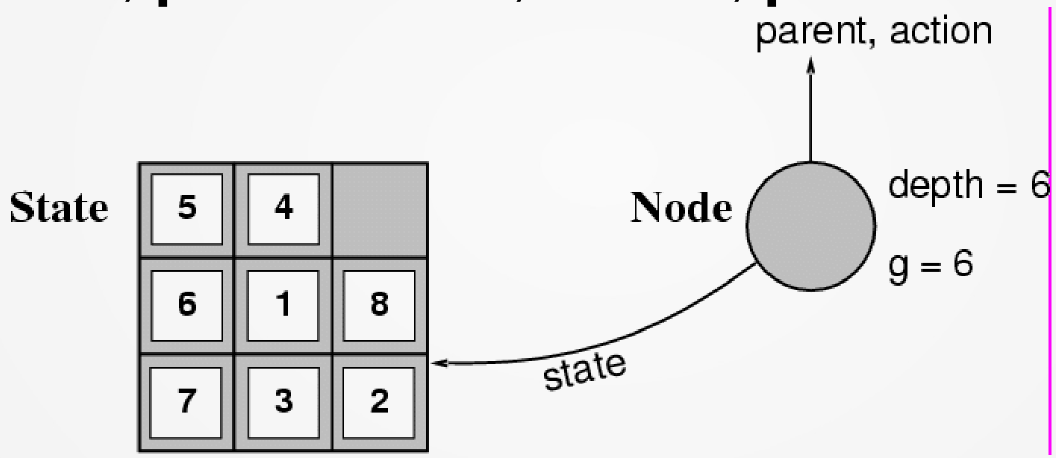
Implementation: General Tree Search

```
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](State[node]) then return Solution(node)
        fringe ← InsertAll(Expand(node, problem), fringe)
```

```
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
```

Implementation: States vs. Nodes

- 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**



- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states

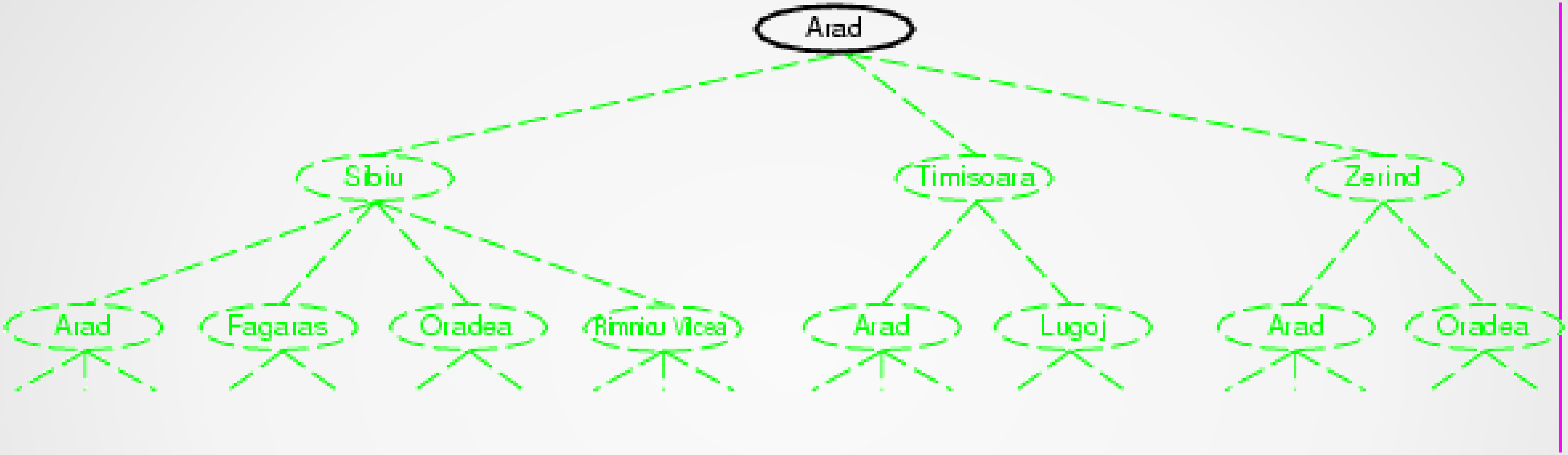
Useful Concepts

- **State space**: the set of all states reachable from the initial state by *any* sequence of actions
  - When several operators can be applied to each state, this gets large very quickly
- **Path**: a sequence of actions leading from one state s_j to another state s_k
- **Frontier**: those states that are available for *expanding*, for applying legal actions to
- **Solution**: a path from the initial state s_i to a state s_f that satisfies the goal state

Basic Search Algorithms: Tree Search

- How do we find the solutions for the previous problem formulations?
  - Enumerate in some order all possible paths from the initial state
  - Here: search through *explicit tree generation*
    - Root = initial state
    - Nodes and leafs generated through *transition model*
  - In general search generates a *graph* (same state through multiple paths) but we'll just look at *trees* in this lecture
    - Treats different paths to the same node as distinct

Simple Tree Search Example



function **TREE-SEARCH(*problem, strategy*)** return **a solution or failure**

Initialize frontier to the initial state of the problem

do

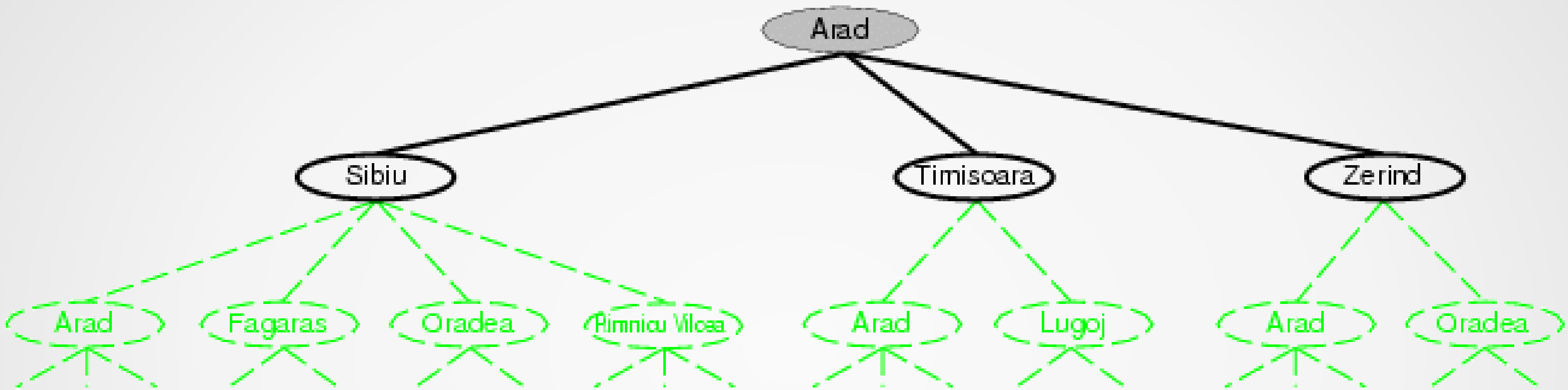
if **the frontier is empty** then return ***failure***

choose leaf node for expansion according to *strategy* & remove from frontier

if **node contains goal state** then return ***solution***

else **expand the node and add resulting nodes to the frontier**

Tree Search Example



function **TREE-SEARCH(*problem, strategy*)** return **a solution or failure**

Initialize frontier to the *initial state* of the *problem*

do

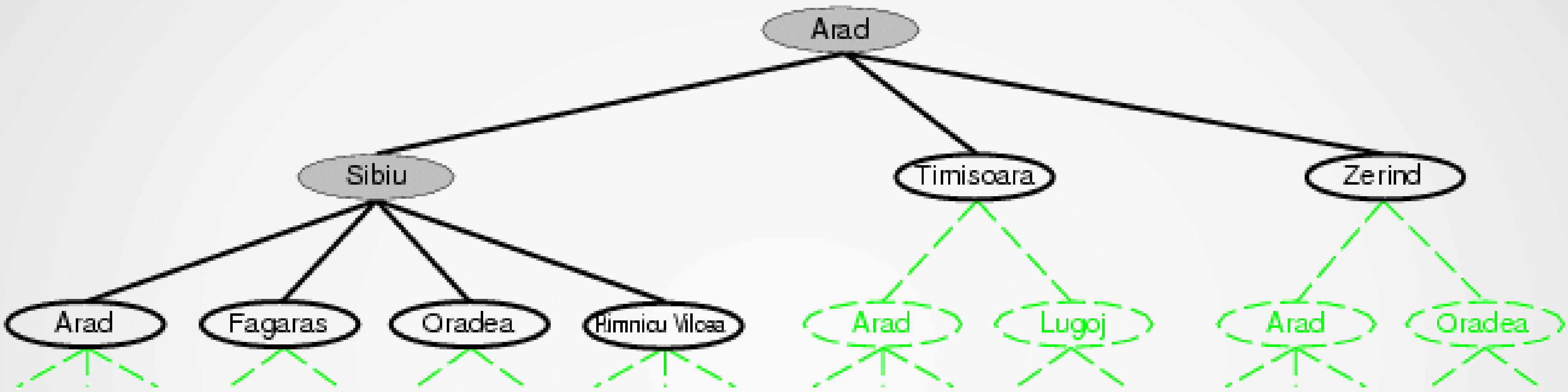
if **the frontier is empty** then return ***failure***

choose leaf node for expansion according to strategy & remove from frontier

if **node contains goal state** then return ***solution***

else **expand the node and add resulting nodes to the frontier**

Tree Search Example



function **TREE-SEARCH(*problem, strategy*)** return **a solution or failure**

Initialize frontier to the *initial state* of the *problem*

do

if **the frontier is empty** then return ***failure***

choose leaf node for expansion according to strategy & remove from frontier

if **node contains goal state** then return ***solution***

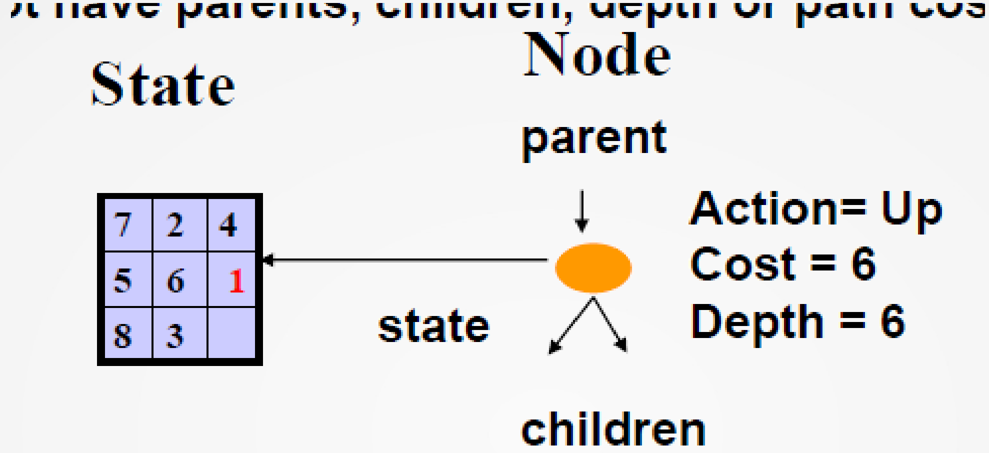
else **expand the node and add resulting nodes to the frontier**

Determines search *process!!*

8-Puzzle: States and Nodes

- A **state** is a (representation of a) **physical configuration**
- A **node** is a data structure constituting *part of* a **search tree**
  - Also includes **parent**, **children**, **depth**, **path cost** ***g(x)***
  - Here **node=<state,parent-node,action,path-cost,depth>**
- **States** *do not* have **parents**, **children**, **depth**, or **path cost**!

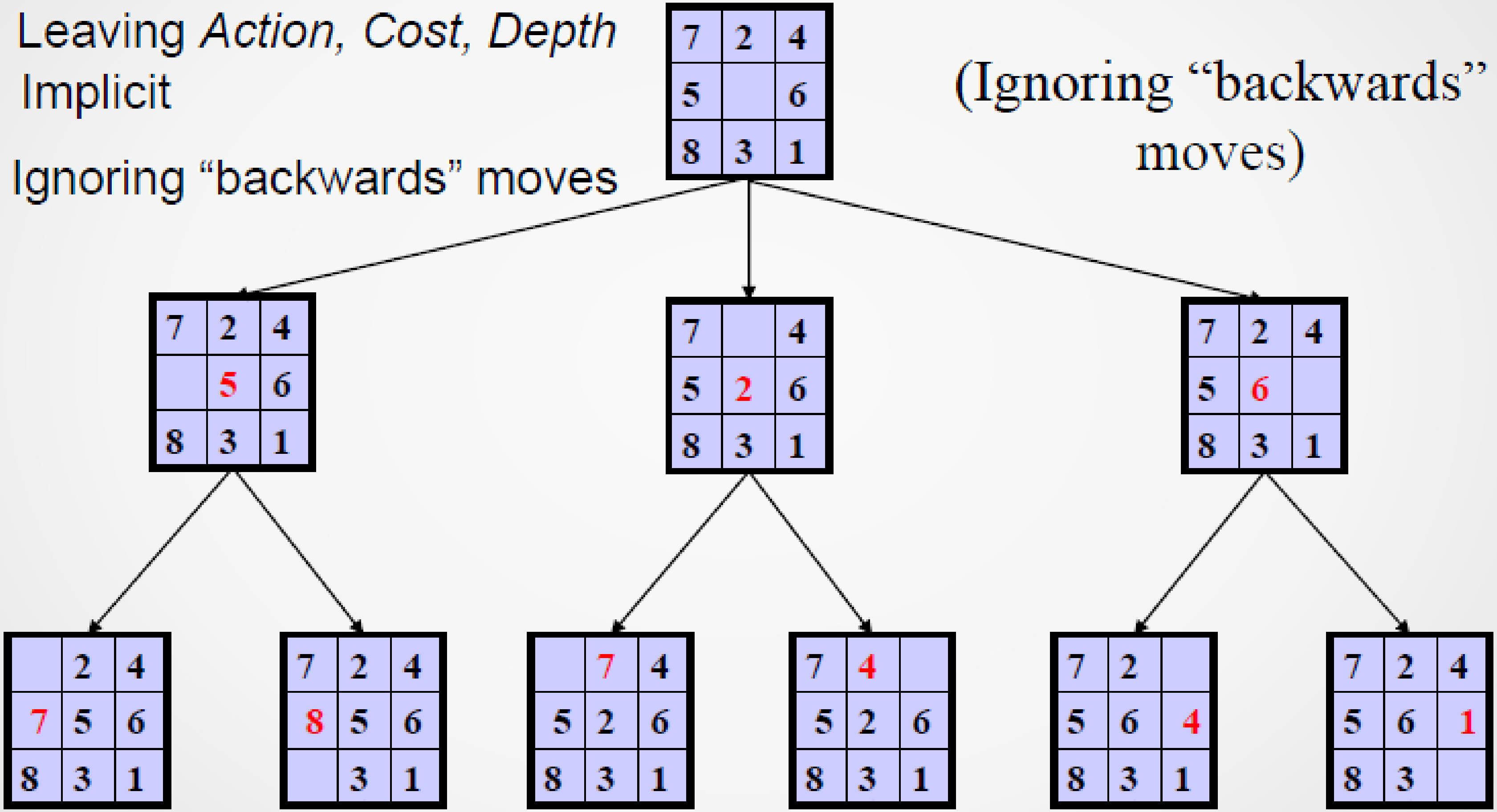
8-Puzzle: States and Nodes



- The **Expand** function
  - Uses the Actions and Transition Model to create the corresponding states
    - Creates new nodes
    - Fills in the various fields

8-Puzzle: Search Tree

- Leaving *Action, Cost, Depth* Implicit
- Ignoring “backwards” moves



(Ignoring “backwards” moves)

Search Strategies

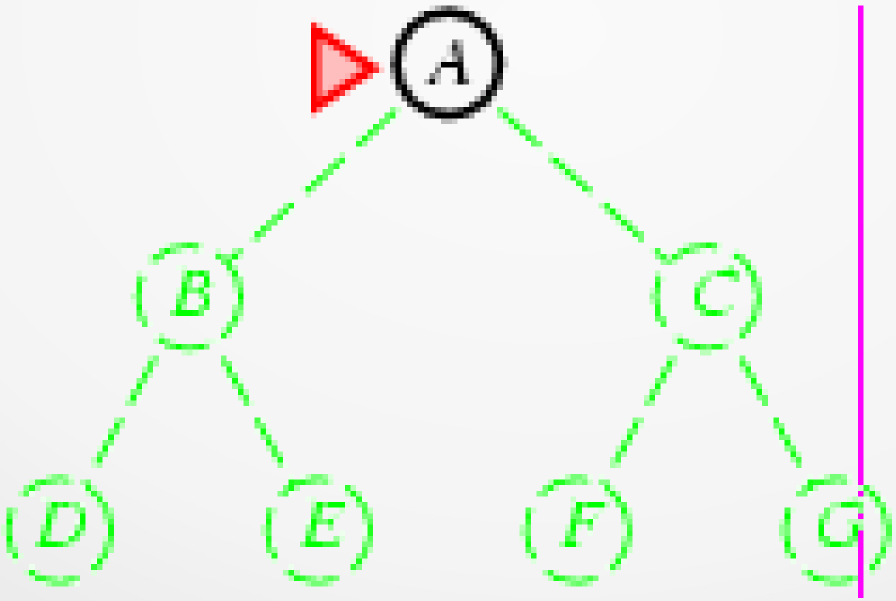
- A search strategy defines the **order of node expansion**
- Strategies are evaluated along the following dimensions:
  - **Completeness**: does it always find a solution if one exists?
  - **Time complexity**: how long does it take to find a solution?
  - **Space complexity**: how much memory is needed to perform the search?
  - **Optimality**: does it always find the optimal 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 Strategies

- **Uninformed** search strategies use only the information available in the problem definition (a.k.a blind search)
- Categories defined by expansion algorithm:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

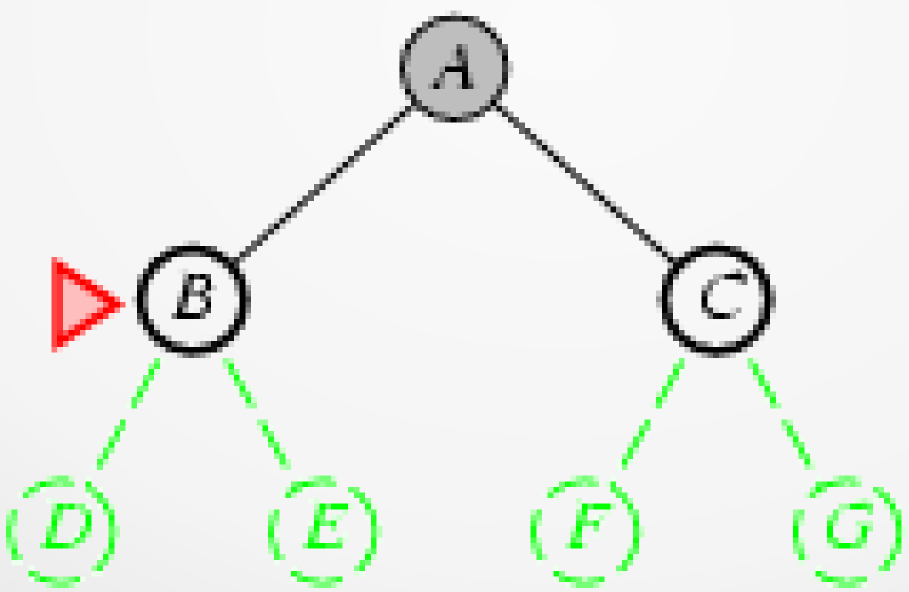
Breadth First Search

- Expand shallowest unexpanded node
- **Implementation**:
  - *Frontier* is a FIFO queue, i.e. new successors at end



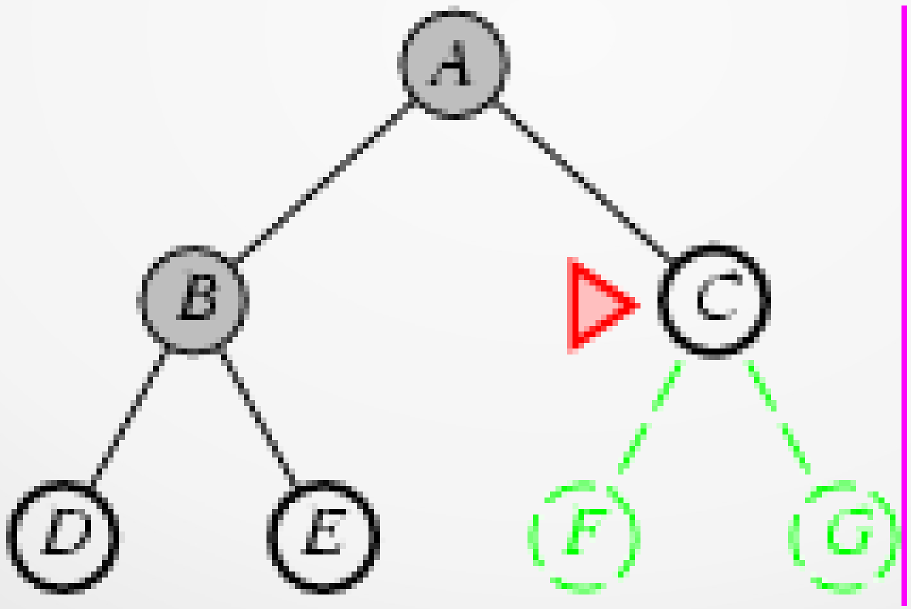
Breadth First Search

- Expand shallowest unexpanded node
- **Implementation**:
  - *Frontier* is a FIFO queue, i.e. new successors at end



Breadth First Search

- Expand shallowest unexpanded node
- **Implementation**:
  - *Frontier* is a FIFO queue, i.e. new successors at end



Breadth First Search: Properties

- **Complete?** Yes (if **b** is finite)
- **Time?** $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d-1) = O(b^{d+1})$
- **Space?** $O(b^{d+1})$ (keeps every node in memory)
- **Optimal?** Yes (if step cost is equal)

Space is the bigger problem (more than time)

Breadth First Search: Properties

Depth	Nodes	Time	Memory
2	110	.11 ms	107 kB
4	11,110	11 msec	10.6 MB
6	10^6	1.1 sec	1 GB
8	10^8	2 mins	103 GB
10	10^{10}	3 hrs	10 TB
12	10^{12}	13 days	1 PB
14	10^{14}	3.5 yrs	99 PB
16	10^{16}	350 yrs	1 EB

Space is the bigger problem (more than time)

Uniform-Cost Search

- Similar to BFS
- Expands node *n* with the lowest path cost
- Does not consider the number of steps a path has
  - instead it considers the total cost of the path
- Will be stuck in infinite loop **if** expanding a node that has zero cost leads to the same state

Uniform-Cost Search

- Expand least-cost unexpanded node
- **Implementation**:
  - *Frontier* is a priority queue ordered by path cost
  - Equivalent to breadth-fist search **if** step costs are all equal

Uniform-Cost Search: Properties

- **Complete?** Yes (if step cost $\geq \varepsilon$)
- **Time?** # of nodes with *g* ≤ cost of optimal solution

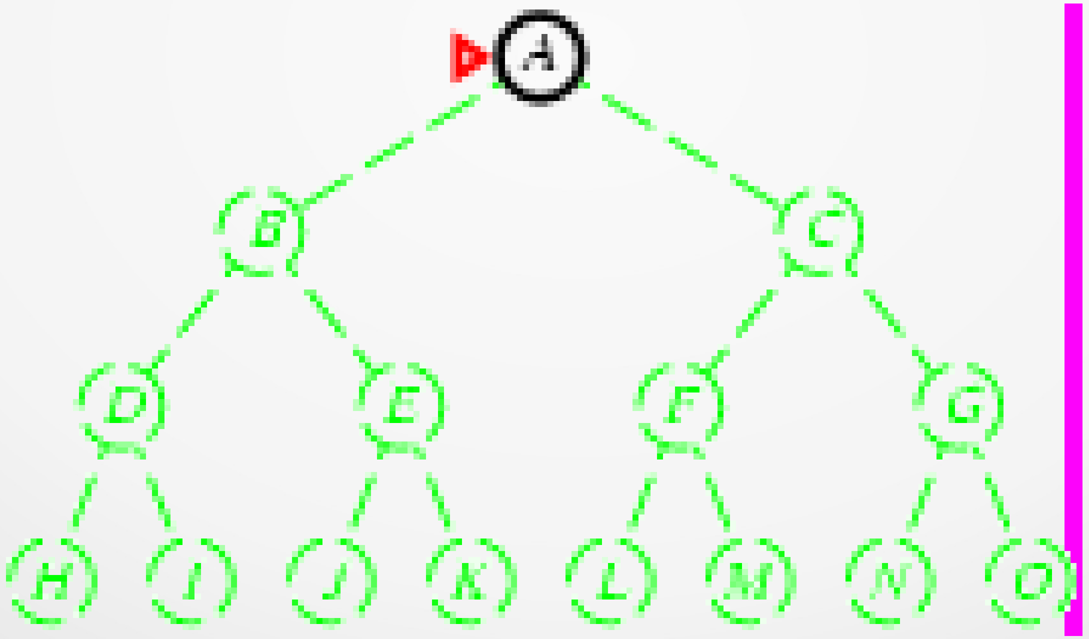
  $O(b^{ceiling(C^*/\varepsilon)})$ where C* is optimal solution cost
- **Space?** # of nodes with *g* ≤ cost of optimal solution

  $O(b^{ceiling(C^*/\varepsilon)})$
- **Optimal?** Yes, nodes expanded in increasing order of

  g(n)

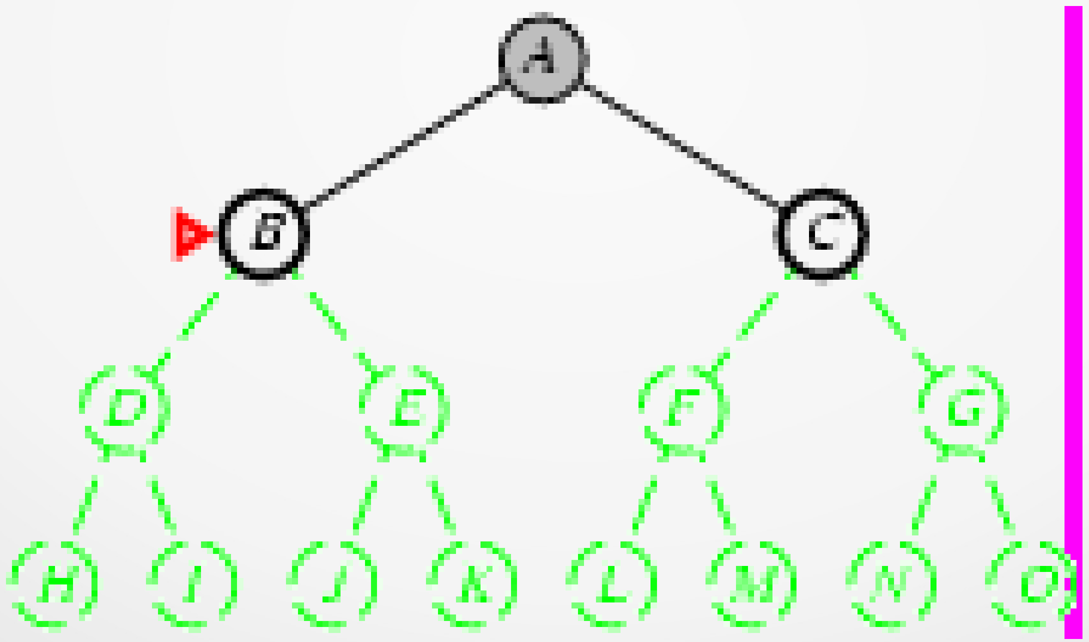
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



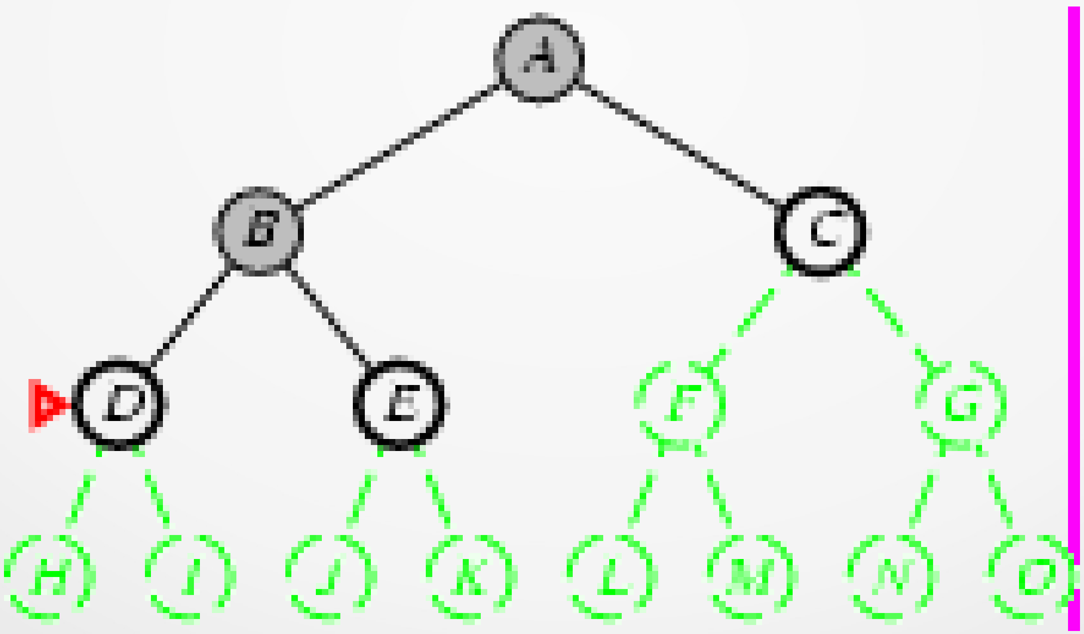
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



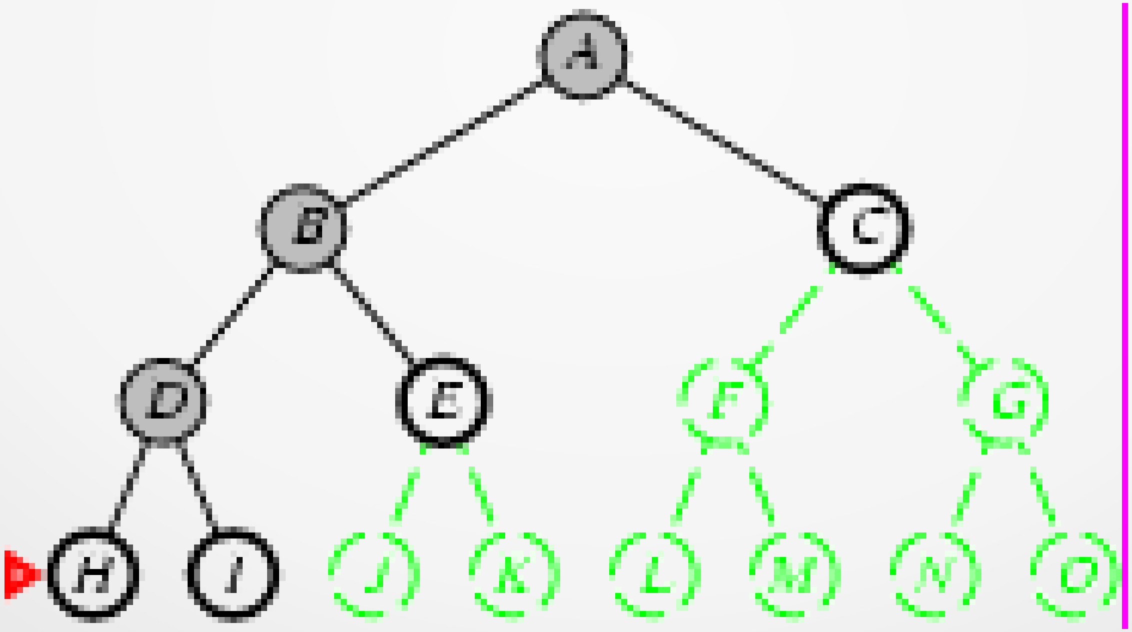
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



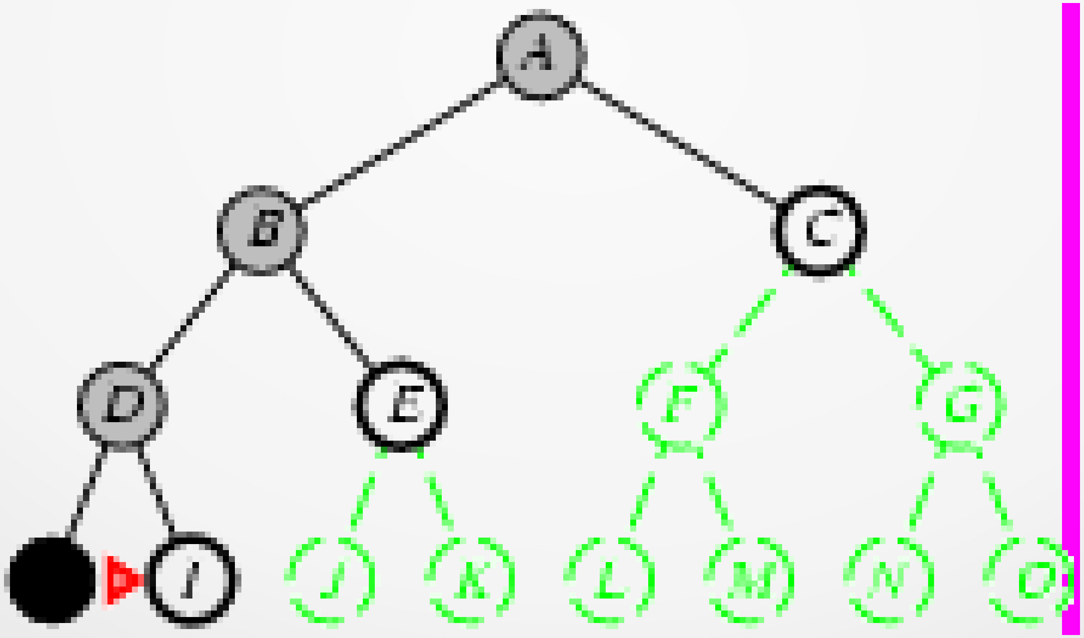
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



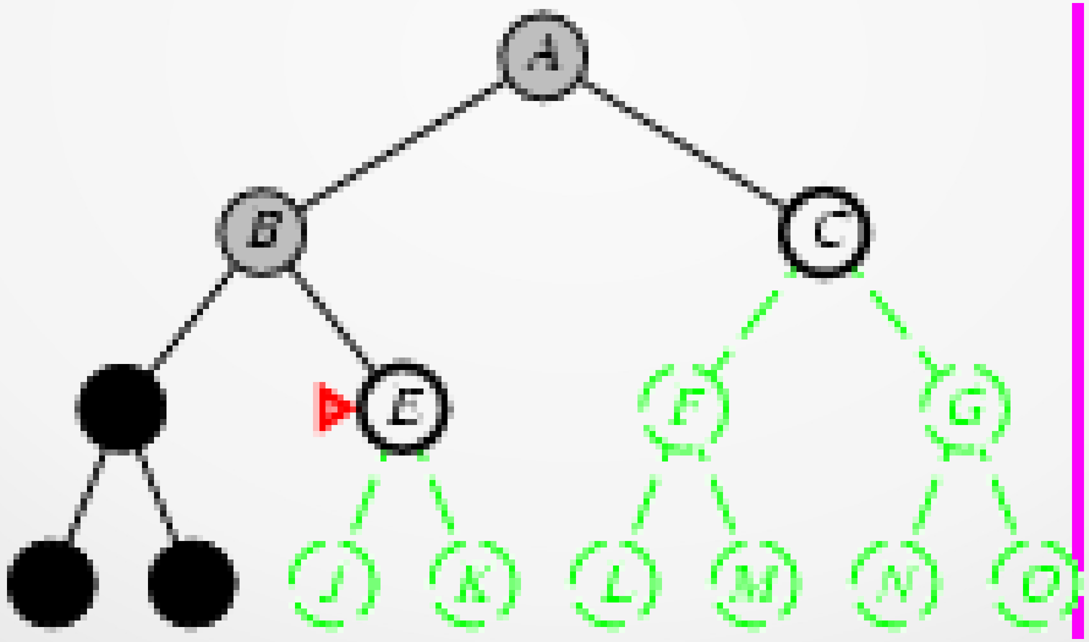
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



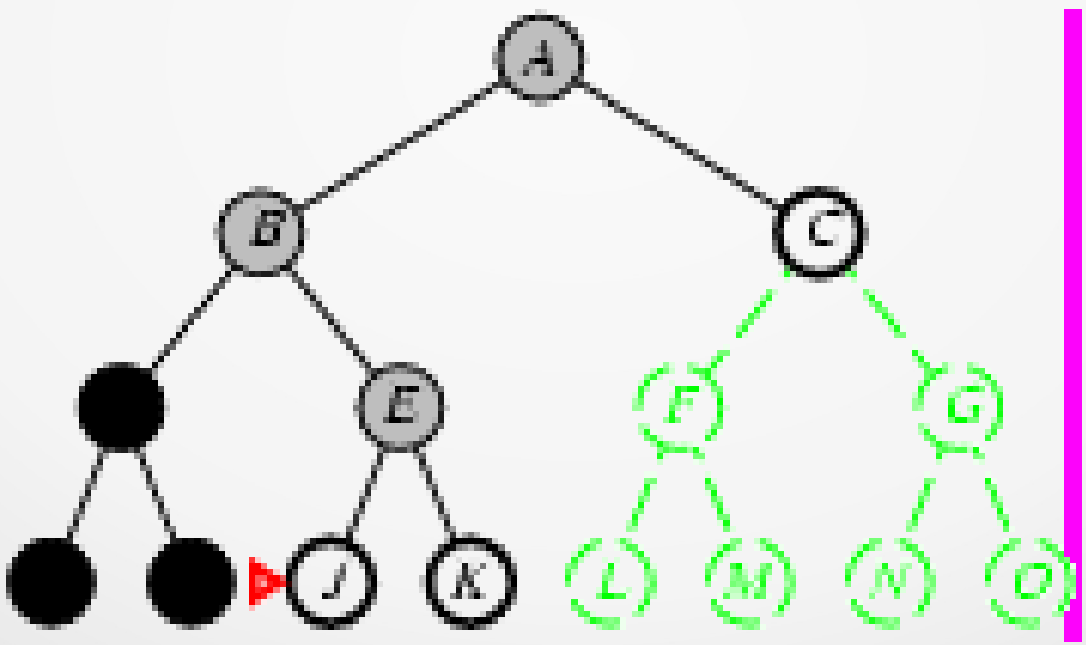
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



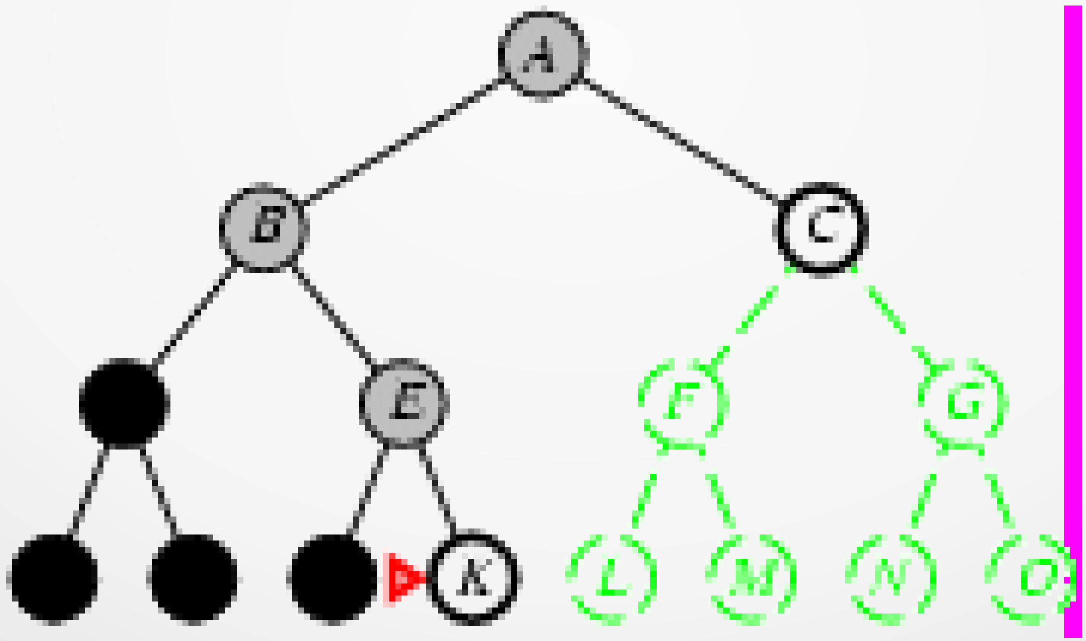
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



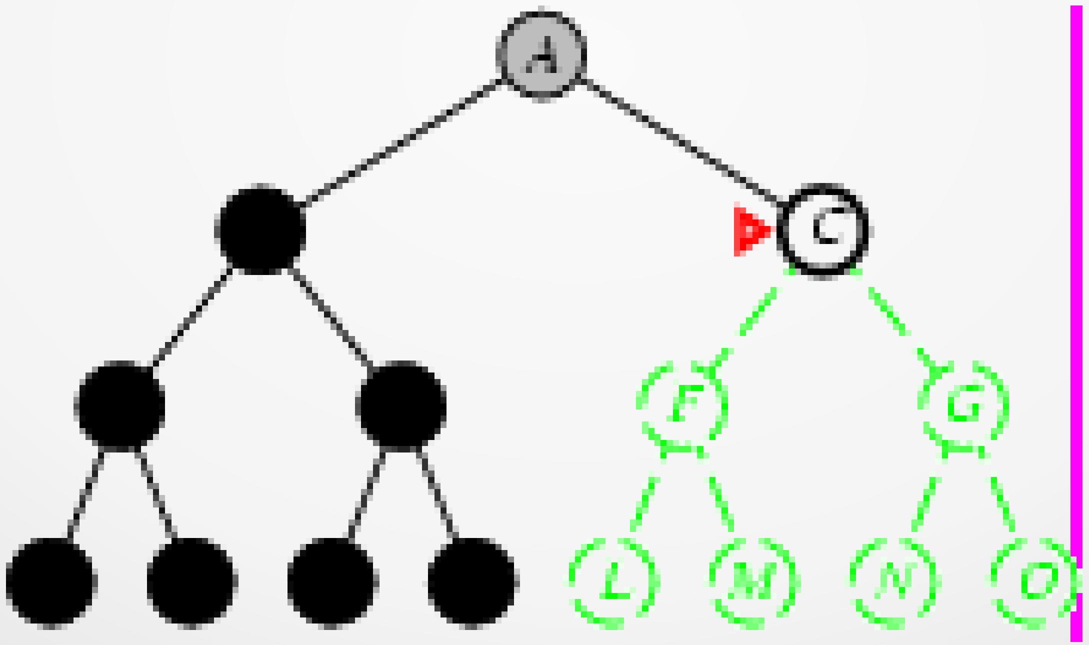
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



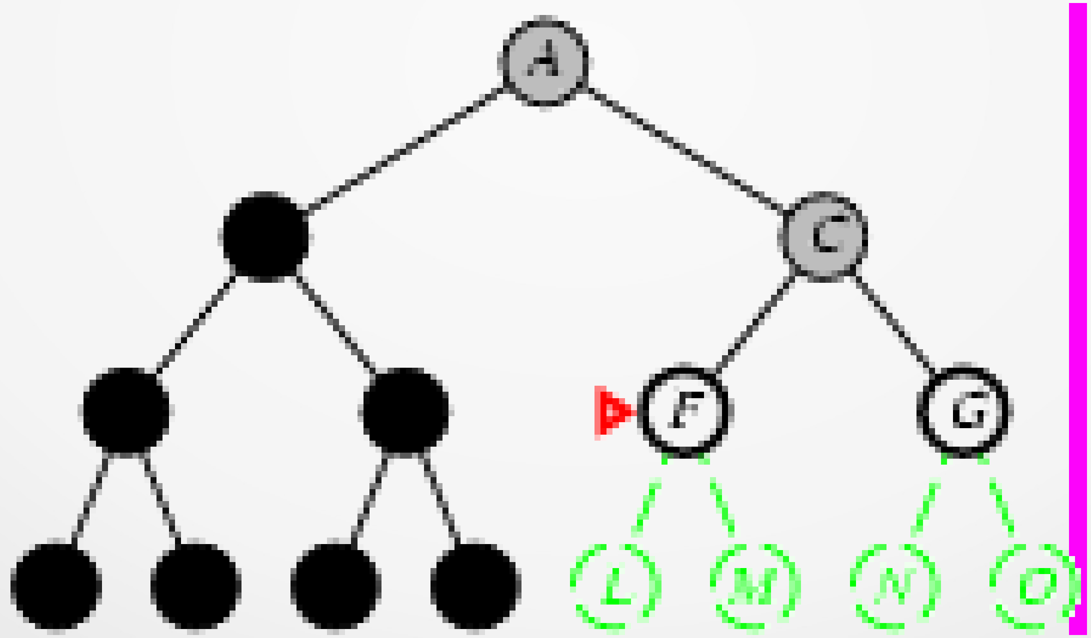
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



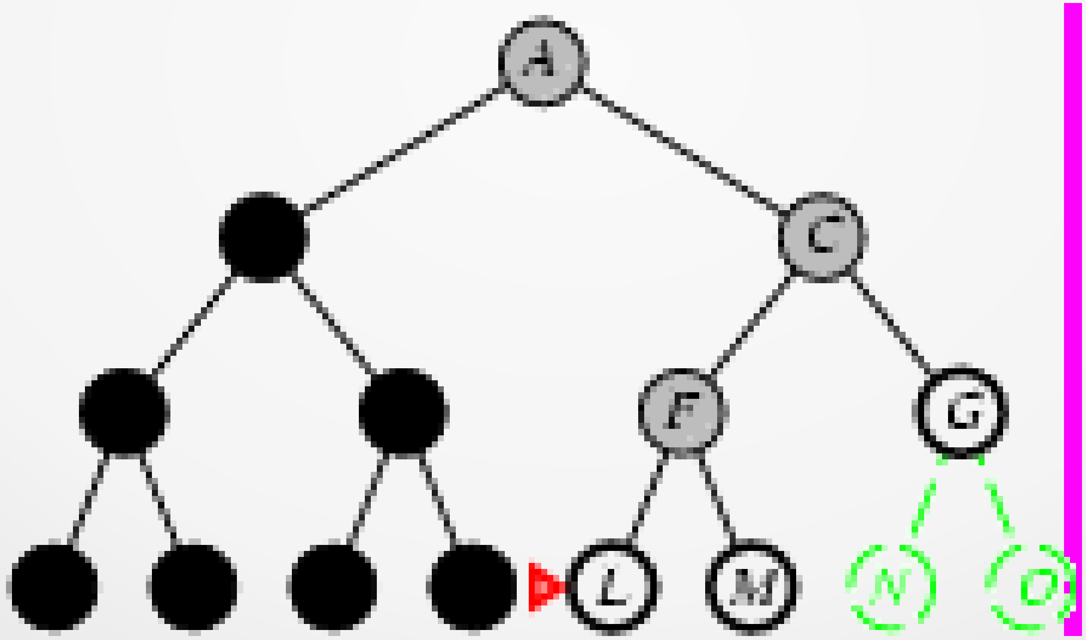
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



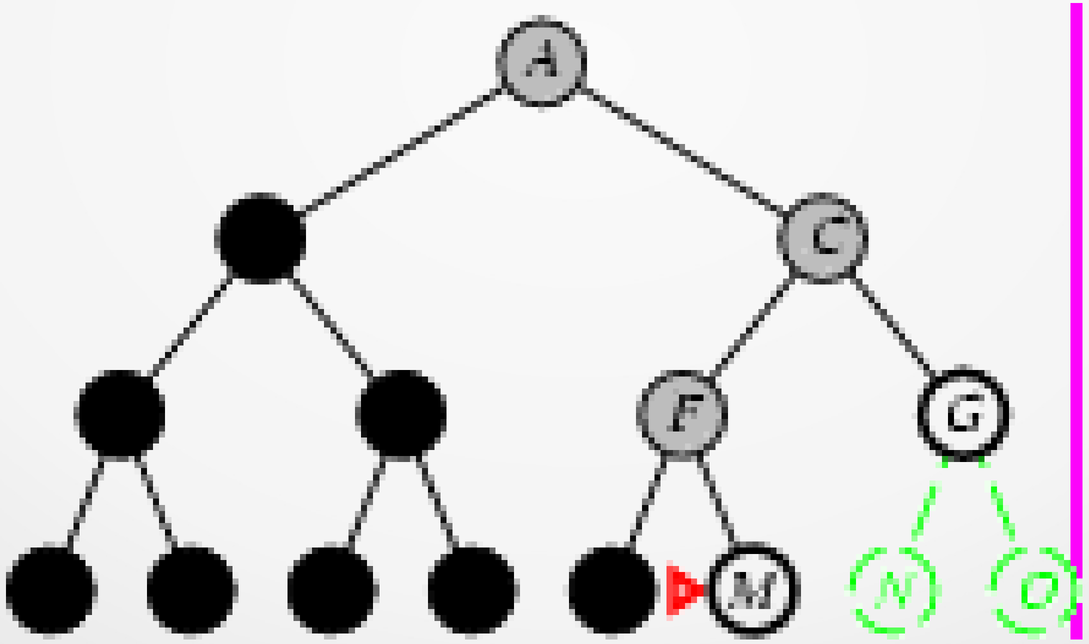
Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



Depth First Search

- Expand deepest unexpanded node
- **Implementation**:
  - *Frontier* is a LIFO queue, i.e. put successors at front



Depth First Search: Properties

- **Complete?** No: fails in infinite-depth spaces, spaces with loops. If modified to avoid repeated states along path, then complete in finite spaces.
- **Time?** $O(b^m)$: terrible if m is much larger than d but if solutions are dense, may be much faster than BFS
- **Space?** $O(bm)$ i.e. linear space!
- **Optimal?** No

Depth First vs Breadth First

- **Use depth first if:-**
  - Space is restricted
  - There are many possible solutions with long paths and wrong paths can be detected quickly
  - Search can be fine-tuned quickly
- **Use breadth-first if:-**
  - Possible infinite paths
  - Some solutions have short paths
  - Can quickly discard unlikely paths

Search Conundrum

- **Breadth First**
  - Complete
  - Uses $O(b^d)$ space
- **Depth First**
  - Not complete unless *m* is bounded
  - Uses $O(b^m)$ time; terrible if $m >> d$
  - *But* only uses $O(bm)$ space
- How can we get the best of both?

Depth-limited Search: A Building Block

- Depth First search but with depth limit *l*
  - i.e. nodes at depth *l* have no successors
- Solves the infinite-path problem
- If $l = d$ (by luck!) then optimal
- But:
  - If $l < d$ then incompleteness results
  - If $l > d$ then not optimal
- Time complexity: $O(bl)$
- *Space complexity:* $O(bl)$

Iterative Deepening Search

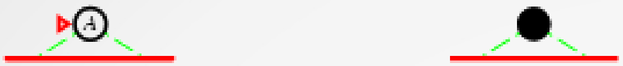
- *A general strategy to find best depth limit l*
  - Key idea: use Depth-limited search as subroutine, with increasing l
  - **Complete**: Goal is always found at depth d, the depth of the shallowest goal-node
- Combines benefits of Depth First search and Breadth First search

Iterative Deepening Search

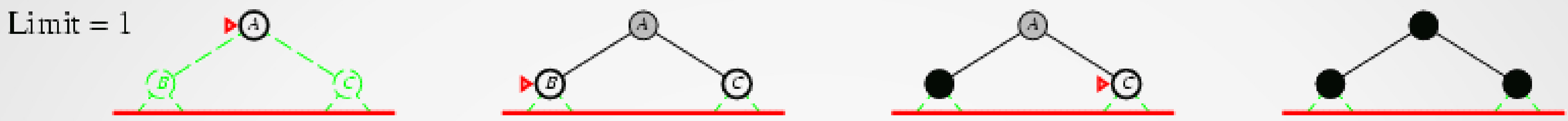
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Iterative Deepening Search (*l* = 0)

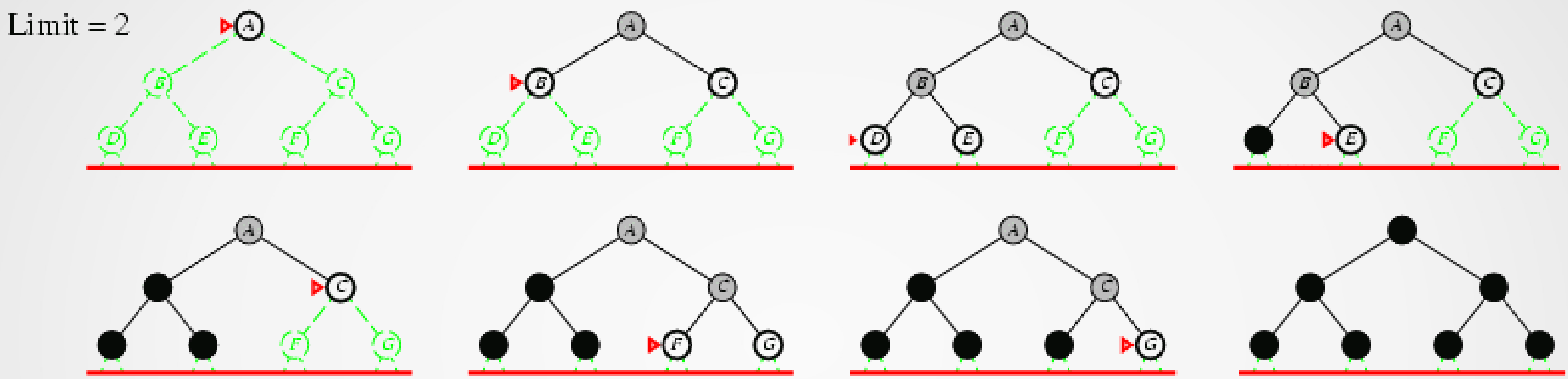
Limit = 0



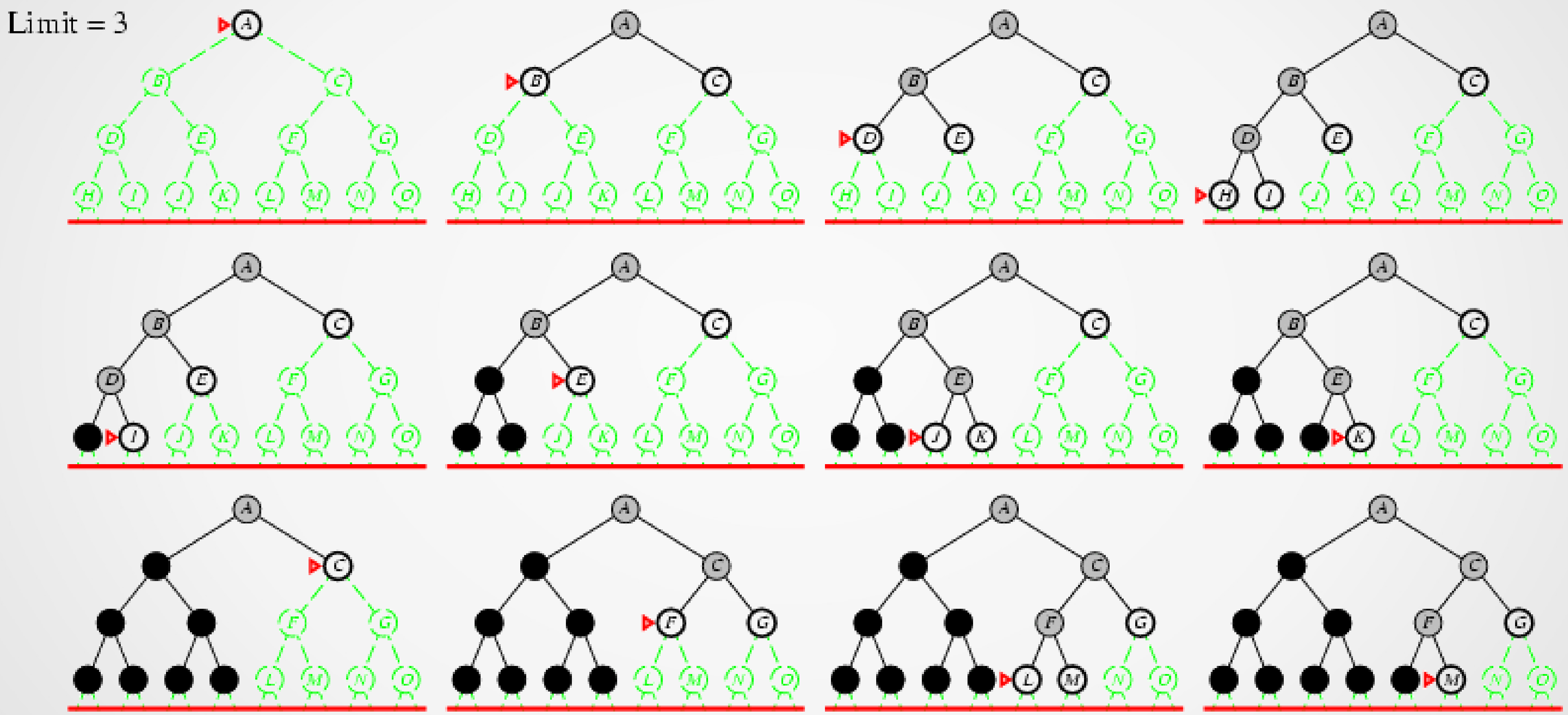
Iterative Deepening Search (*l* = 1)



Iterative Deepening Search (l = 2)



Iterative Deepening Search (l = 3)



Iterative Deepening Search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

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

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + db^1 + (d-b)b^2 \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
- Overhead = (123,456 – 111,111)/111,111 = 11%

Iterative Deepening Search: Properties

- **Complete?** Yes
- **Time?** $(d+1)b^0 + db^1 = (d-1)b^2 + \ldots + b^d = O(b^d)$
- **Space?** $O(bd)$
- **Optimal?** Yes, if step cost = 1

Summary of Search Algorithms so far

Criterion	Breadth First	Uniform Cost	Depth First	Depth-limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Informed/Heuristic Search Strategies

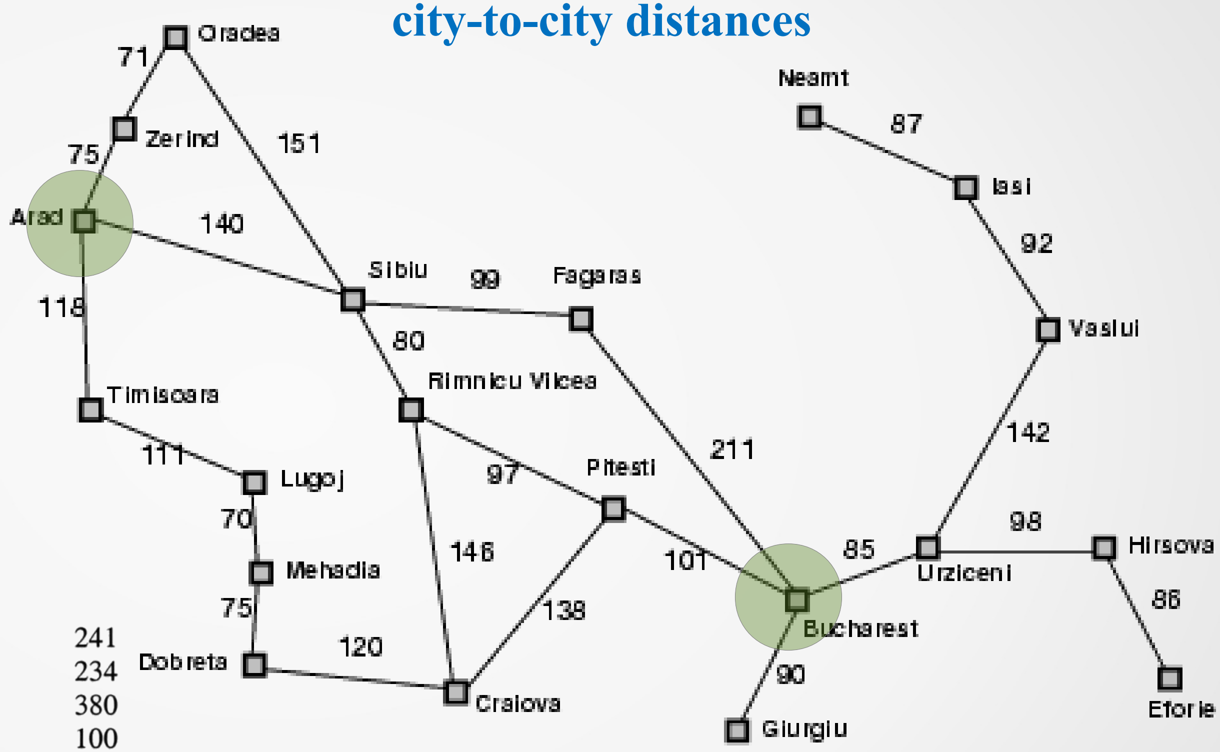
- Uses problem-specific knowledge in searching
- Find solutions more efficiently than an uninformed search
- It uses some scoring function to decide which paths or nodes seem promising
- Then the more promising nodes will be explored first before the less promising nodes

Informed/Heuristic Search Strategies

- Node expansion based on some **estimate** of distance to goal, extending current path
- General approach of informed search:
  - **Best First Search**: node selected for expansion based on an evaluation function *f(n)*
    - *f(n)* includes estimate of distance to goal
- Implementation:
  - Sort *frontier* queue monotonically by *f(n)*
  - Special cases: greedy search, A* search

Romania Revisited

city-to-city distances



Straight-line distances to Bucharest

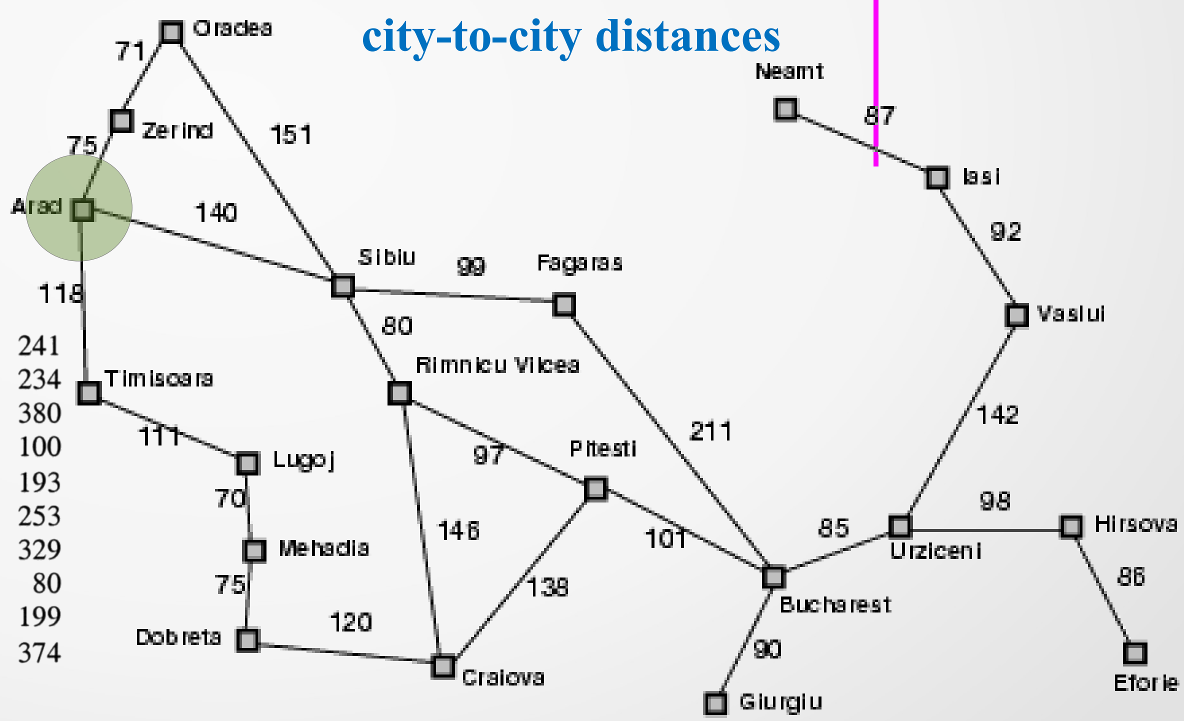
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy Best First Search

“A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood.”

- Let evaluation function $f(n) = h(n)$ (heuristic)
  - $h(n)$ = estimated cost of the cheapest path from node n to goal node
  - If n is goal then $h(n) = 0$
- Here: $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Ignores cost so far to get to that node $g(n)$

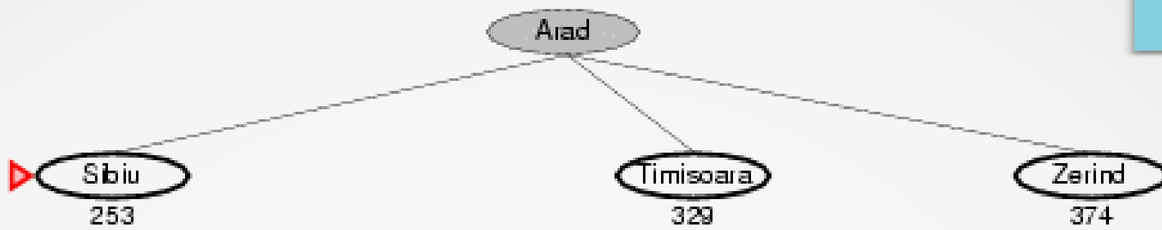
Greedy Best First Search: Example



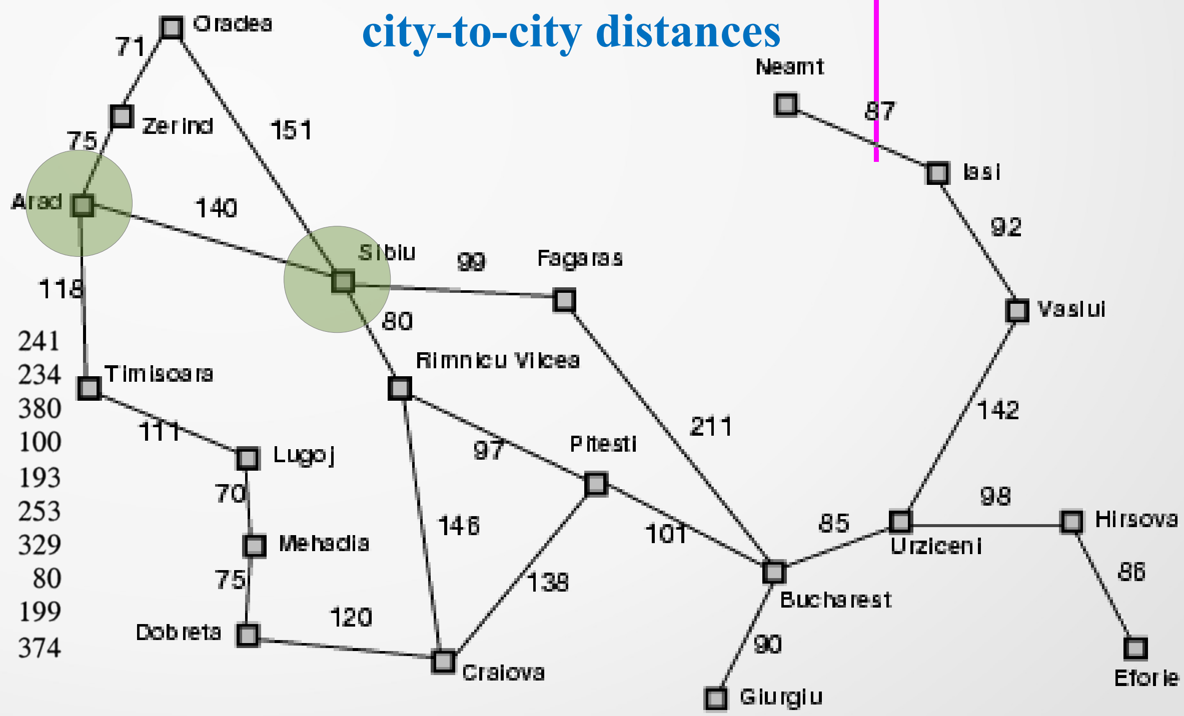
Straight-line distances to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy Best First Search: Example



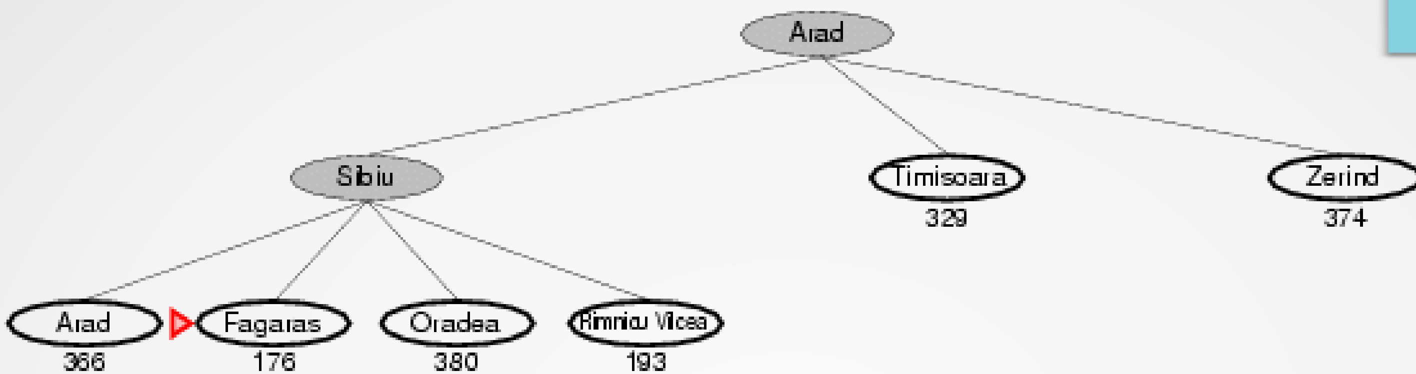
city-to-city distances



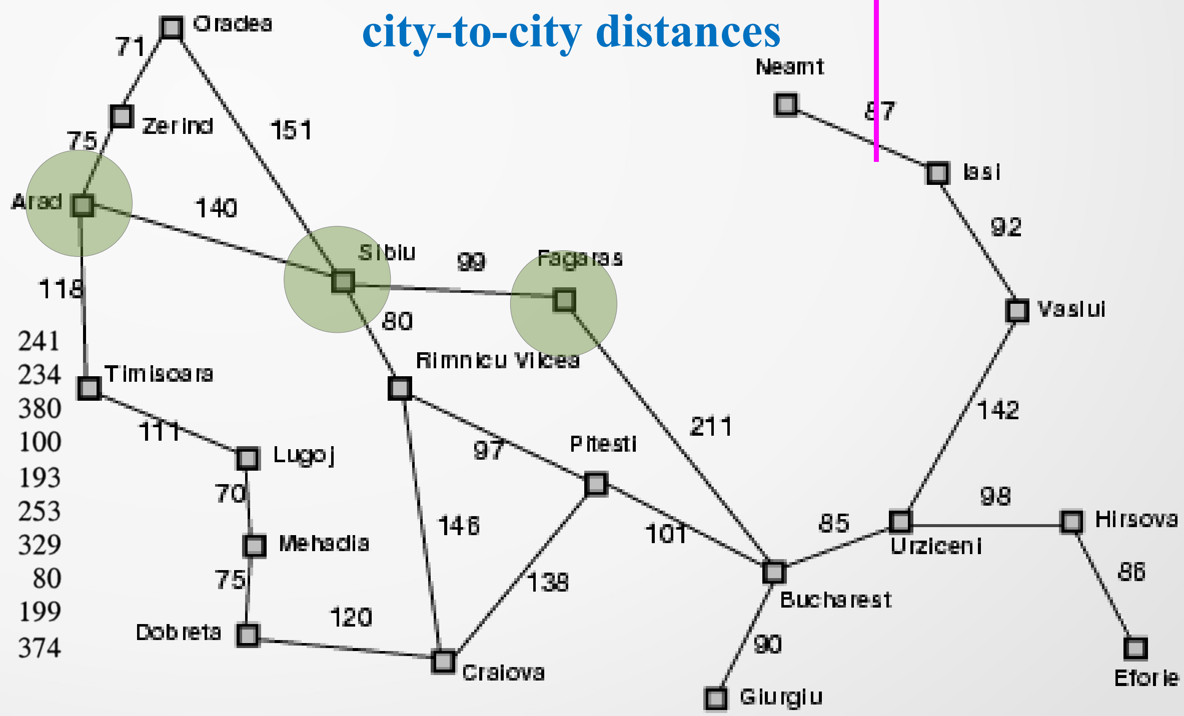
Straight-line distances to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy Best First Search: Example



city-to-city distances

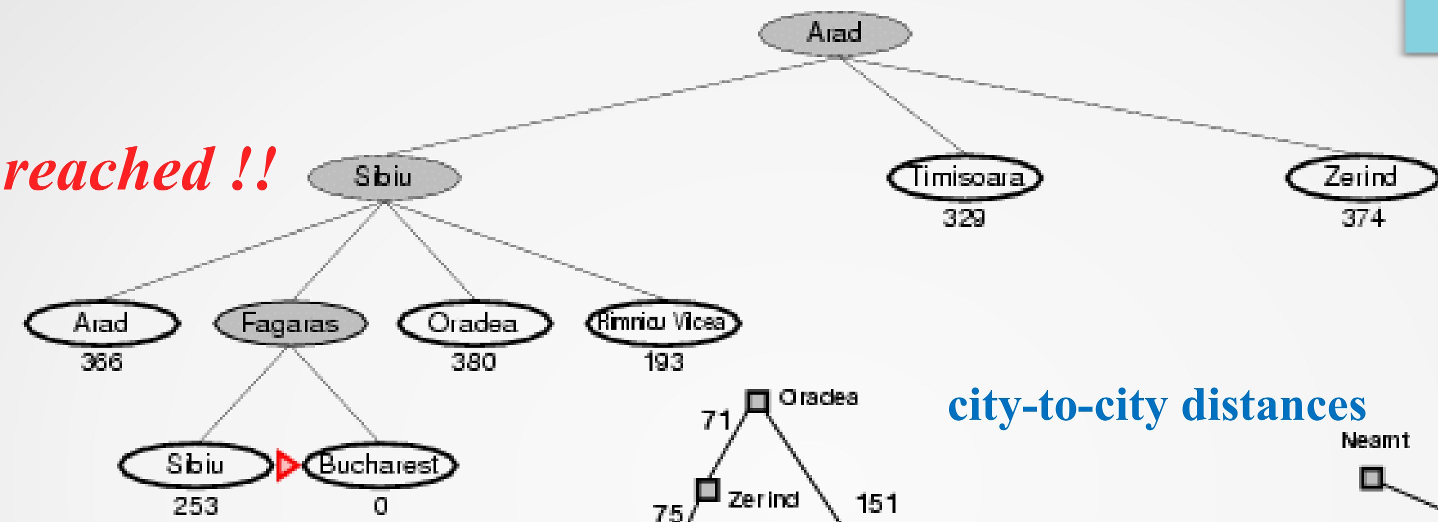


Straight-line distances to Bucharest

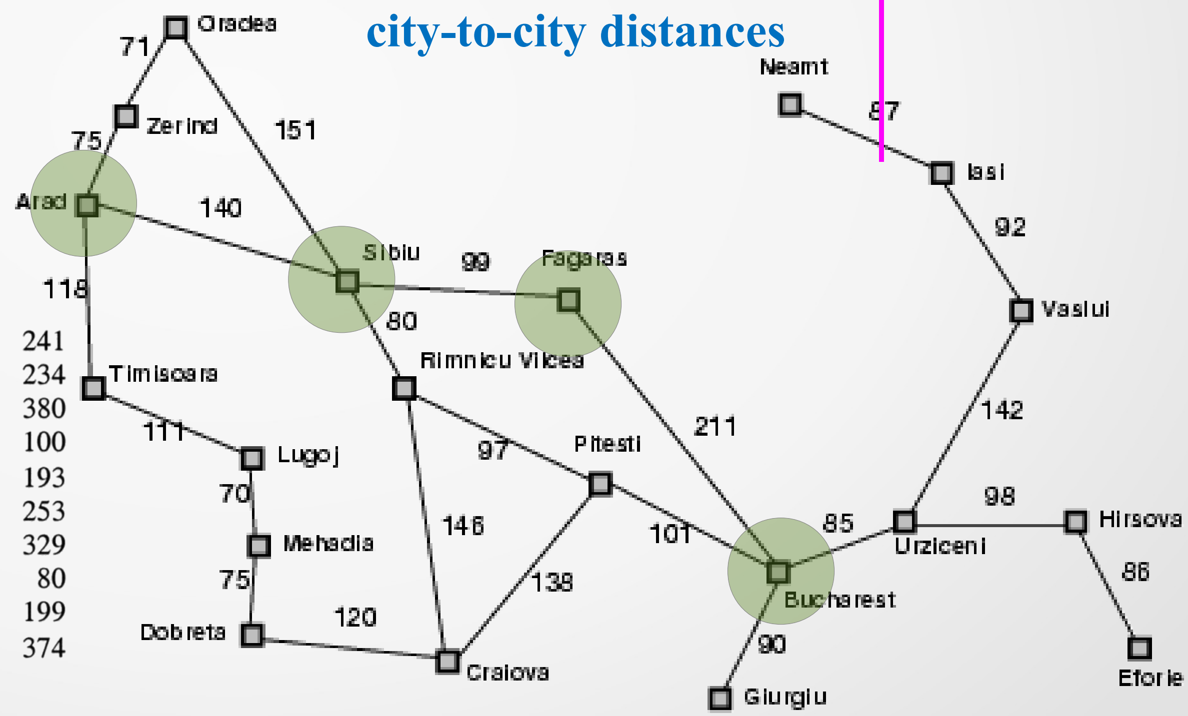
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy Best First Search: Example

Goal reached !!



city-to-city distances



Straight-line distances to Bucharest

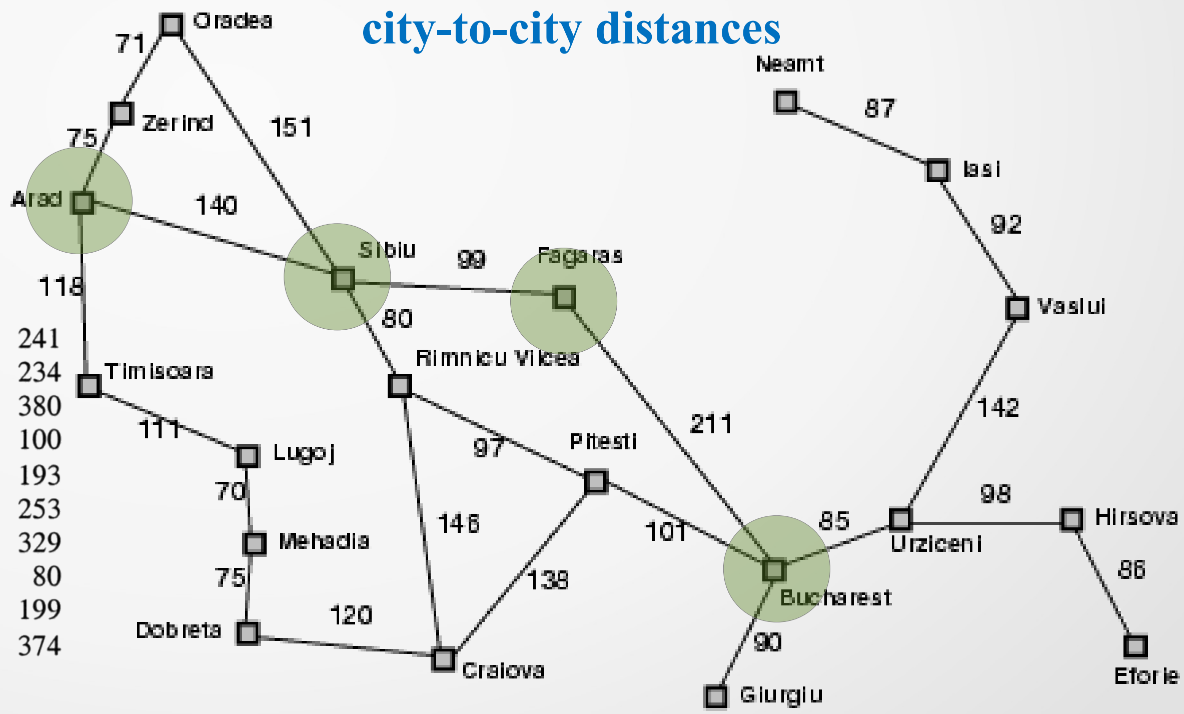
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy Best First Search: Properties

- **Optimal?** - No!
  - Found: *Arad → Sibiu → Fagaras → Bucharest (450km)*
  - Shortest: *Araid → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest (418 km)*

Straight-line distances to Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

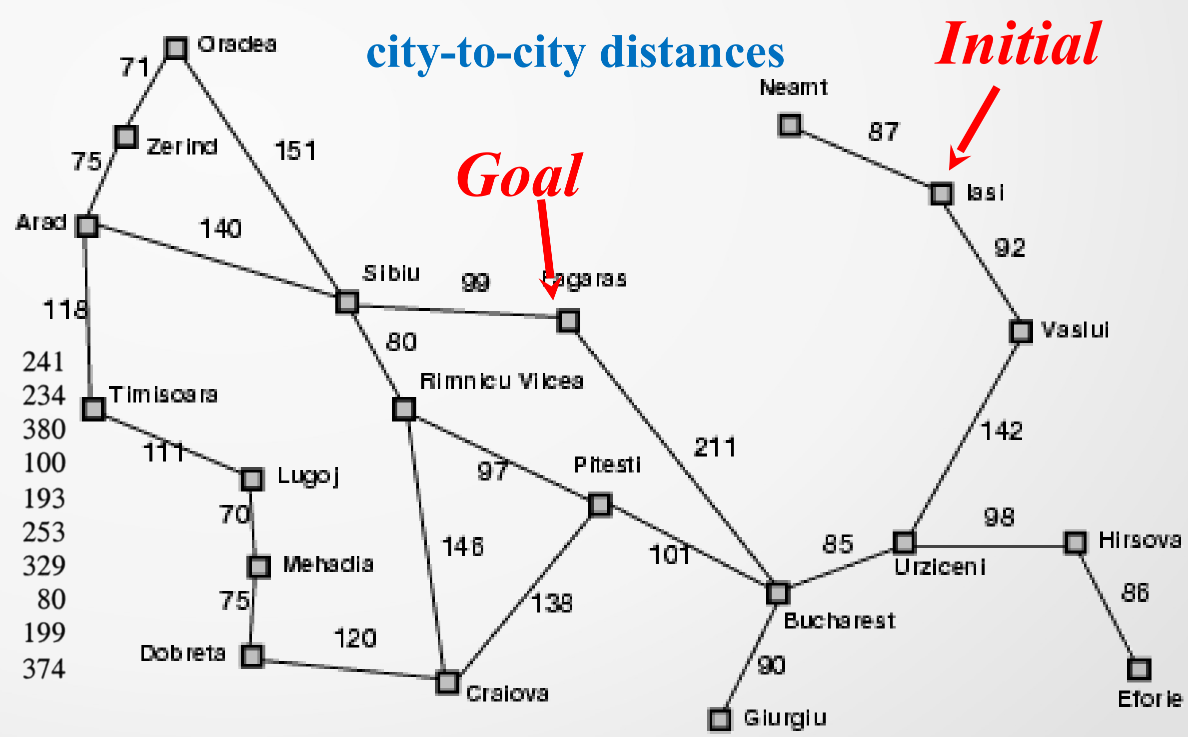


Greedy Best First Search: Properties

- **Complete?** - No
  - Can get stuck in loops e.g. *Iasi* → *Neamt* → *Iasi* → *Neamt* → …

Straight-line distances to Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



Greedy Best First Search: Properties

- **Optimal?** - No
- **Complete?** - No, can get stuck in loops
- **Time?**
  - *O(bm)* – worst case (like Depth First Search)
  - **But** a good heuristic can give dramatic improvement
- **Space?**
  - *O(bm)* – keeps all nodes in memory

A* Search

- Best-known form of Best First Search
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated cost from n to goal
  - $f(n)$ = estimated total cost of path through n to goal
- Implementation: Sort frontier queue by increasing $f(n)$

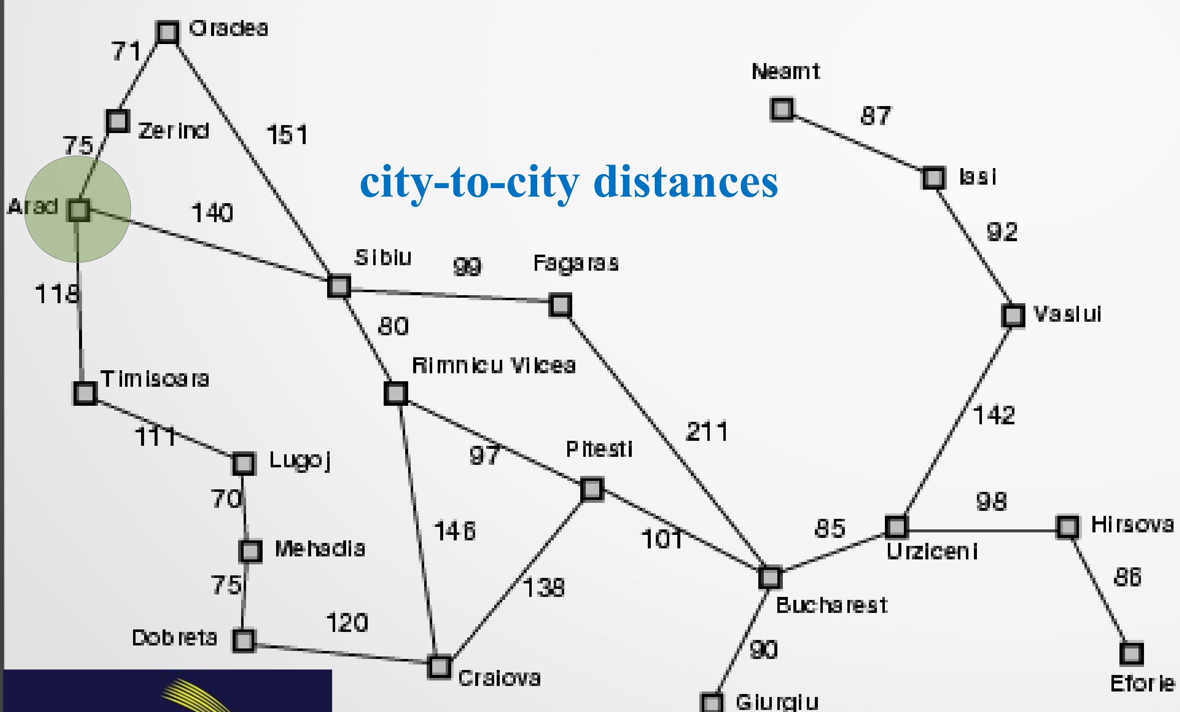
A* Search: Admissible Hueristics

- Let *h(n)* be an **admissible heuristic**
  - A heuristic is admissible if it **never overestimates** the cost to reach the goal i.e it is **optimistic**
  - Formally:

$$
\begin{array}{rl}
\forall n & \text{where } n \text{ is a node} \\
h(n) \leq h^*(n) & \text{where } h^*(n) \text{ is the true cost from } n \\
h(n) \geq 0 & \text{so } h(G) = 0 \text{ for any goal } G
\end{array}
$$

- Example
  - $h_{SLD}(n)$ never overestimates the actual road distance
- Theorem: if *h(n)* is **admissible**, A* using Tree Search is **optimal**

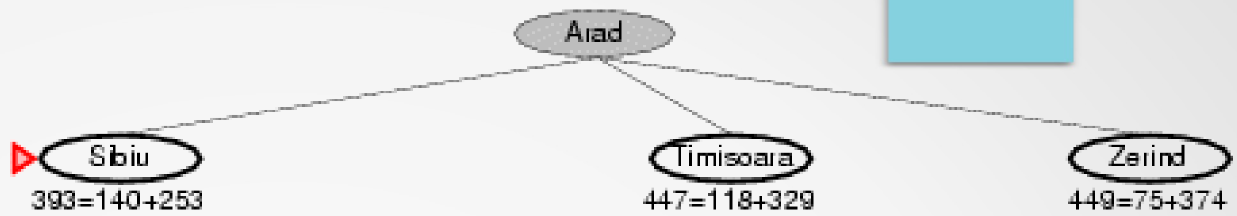
A* Search: Example

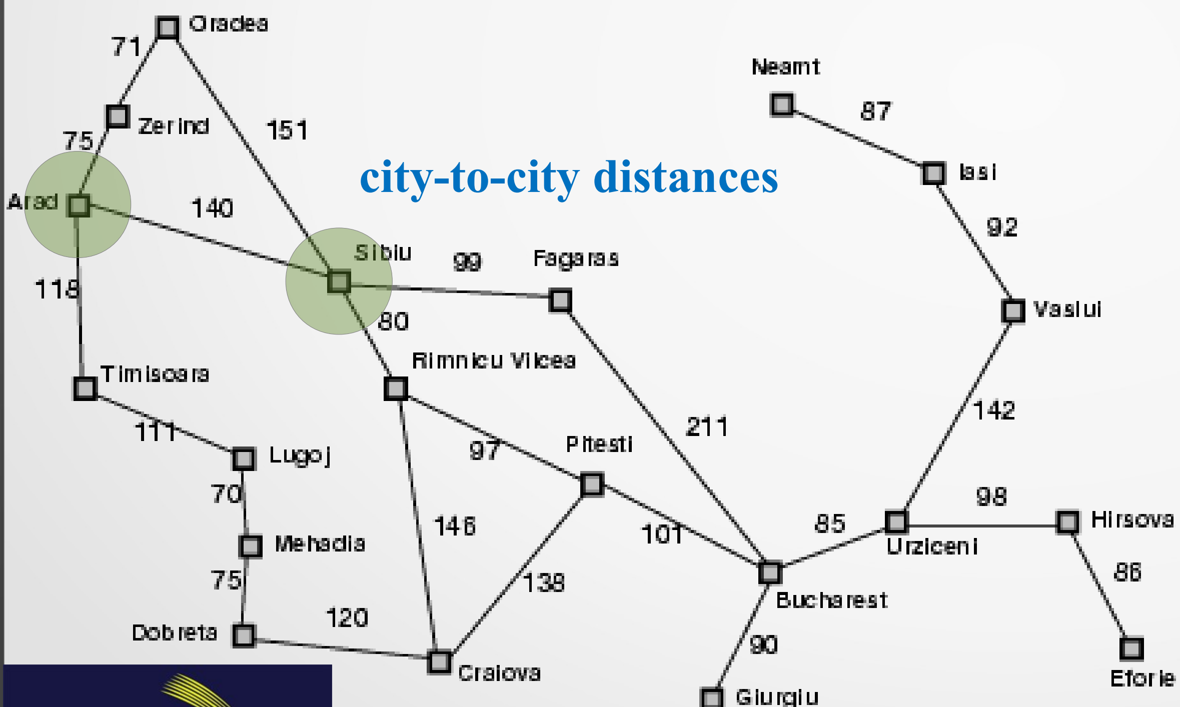


Straight-line distances to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A* Search: Example

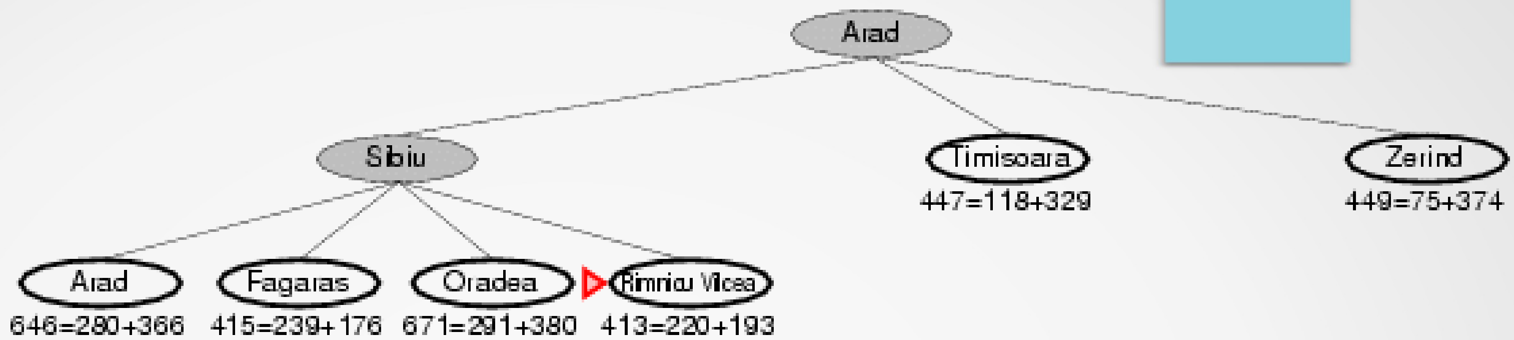


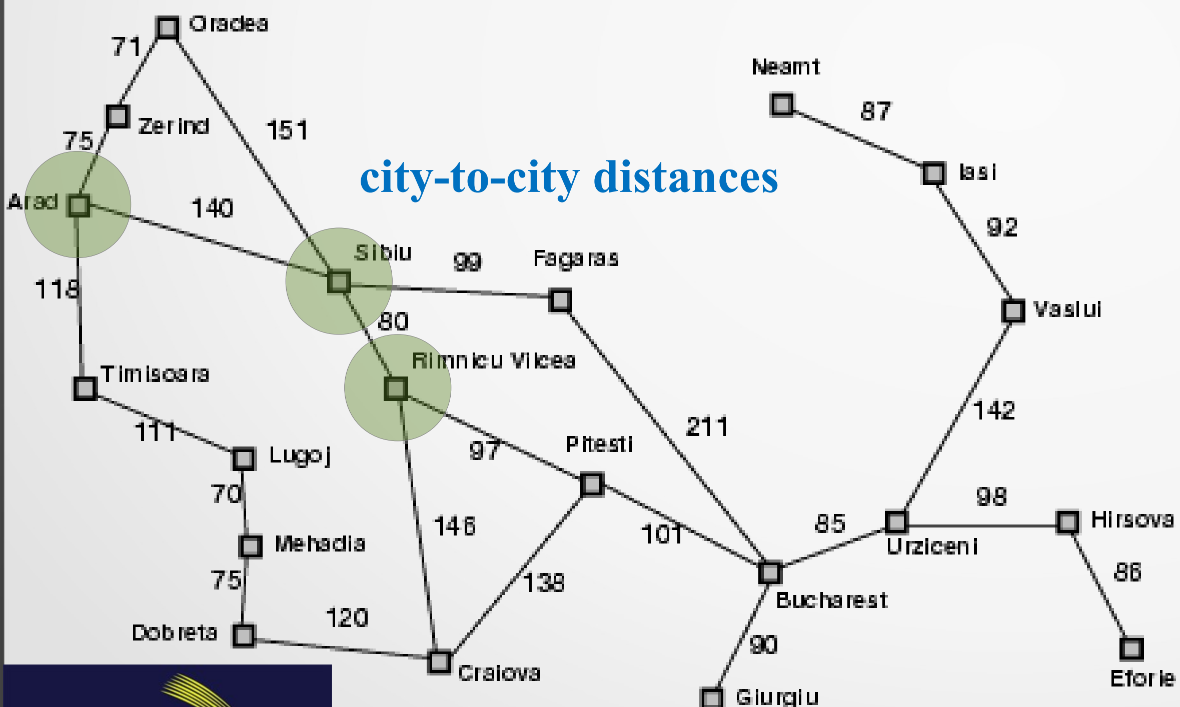


Straight-line distances to Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

A* Search: Example

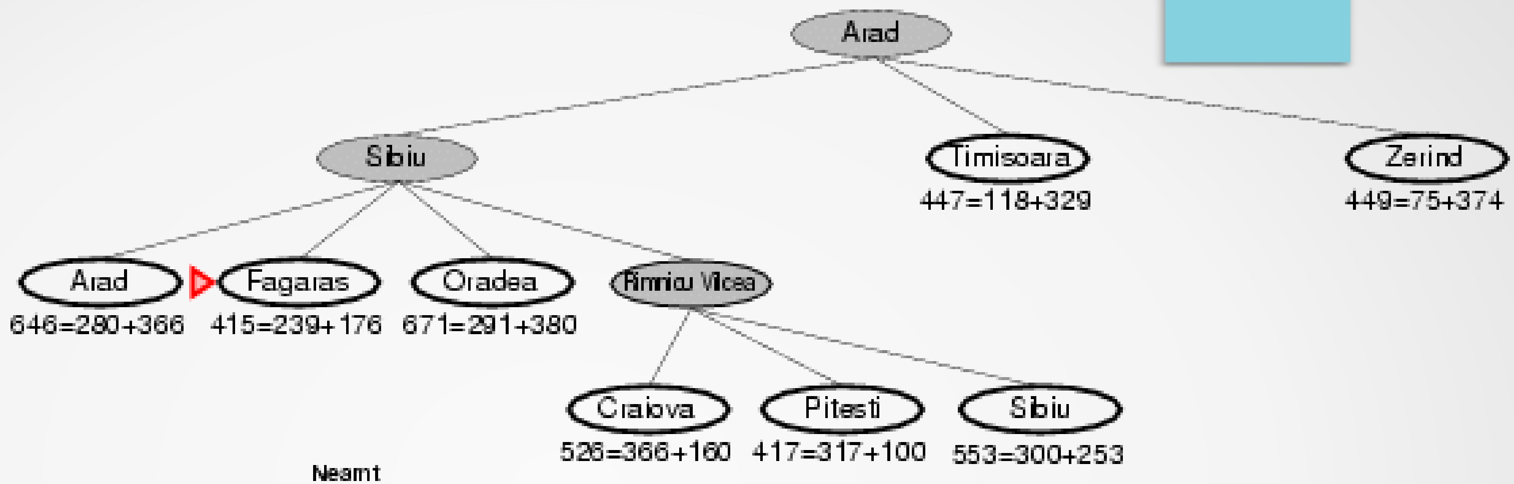


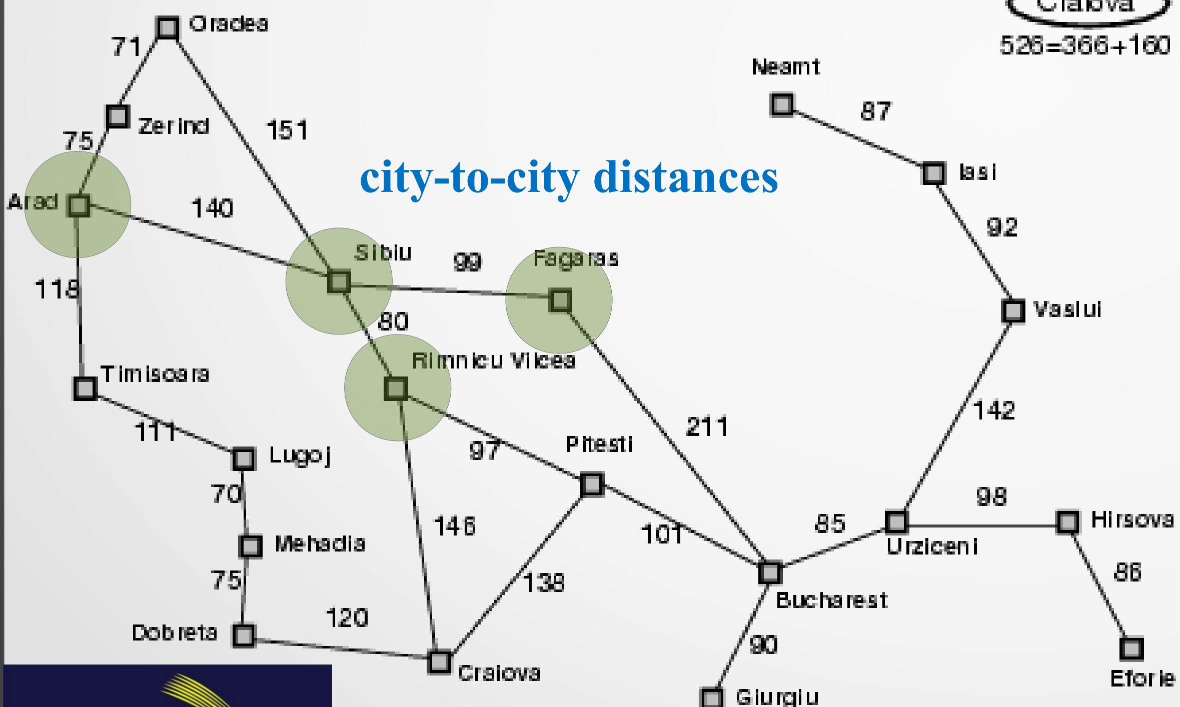


Straight-line distances to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A* Search: Example

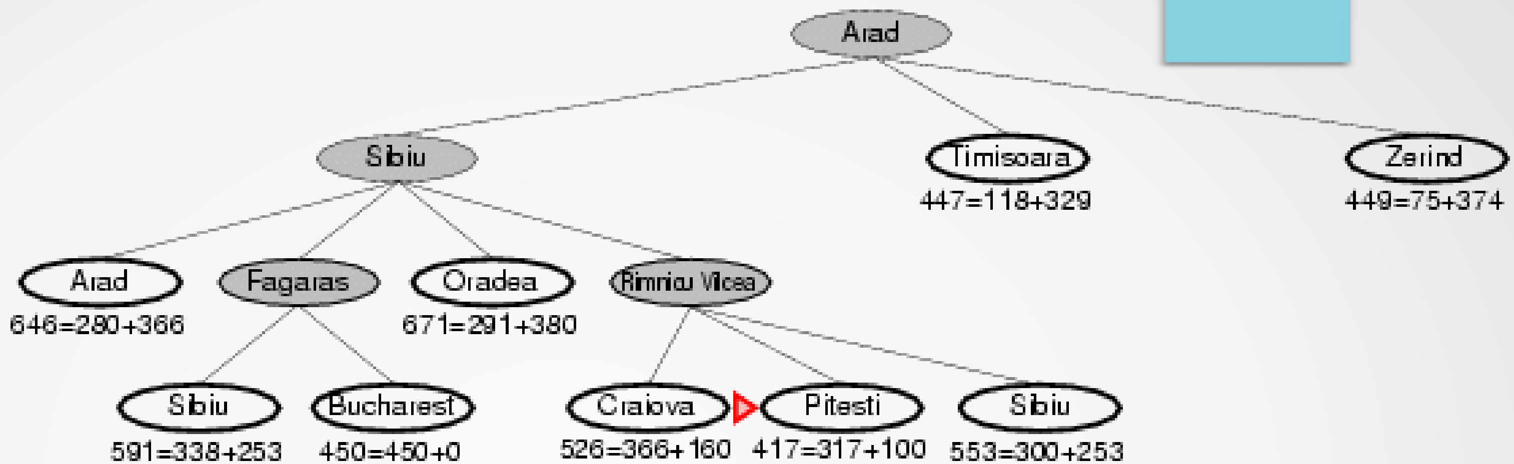


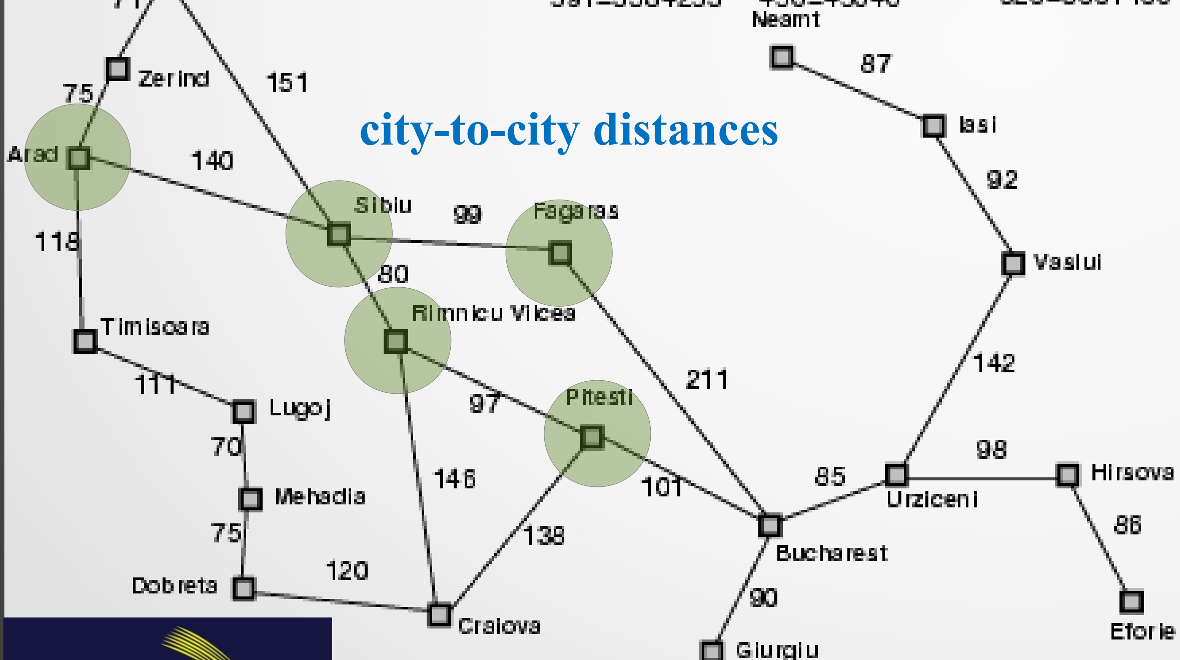


Straight-line distances to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A* Search: Example



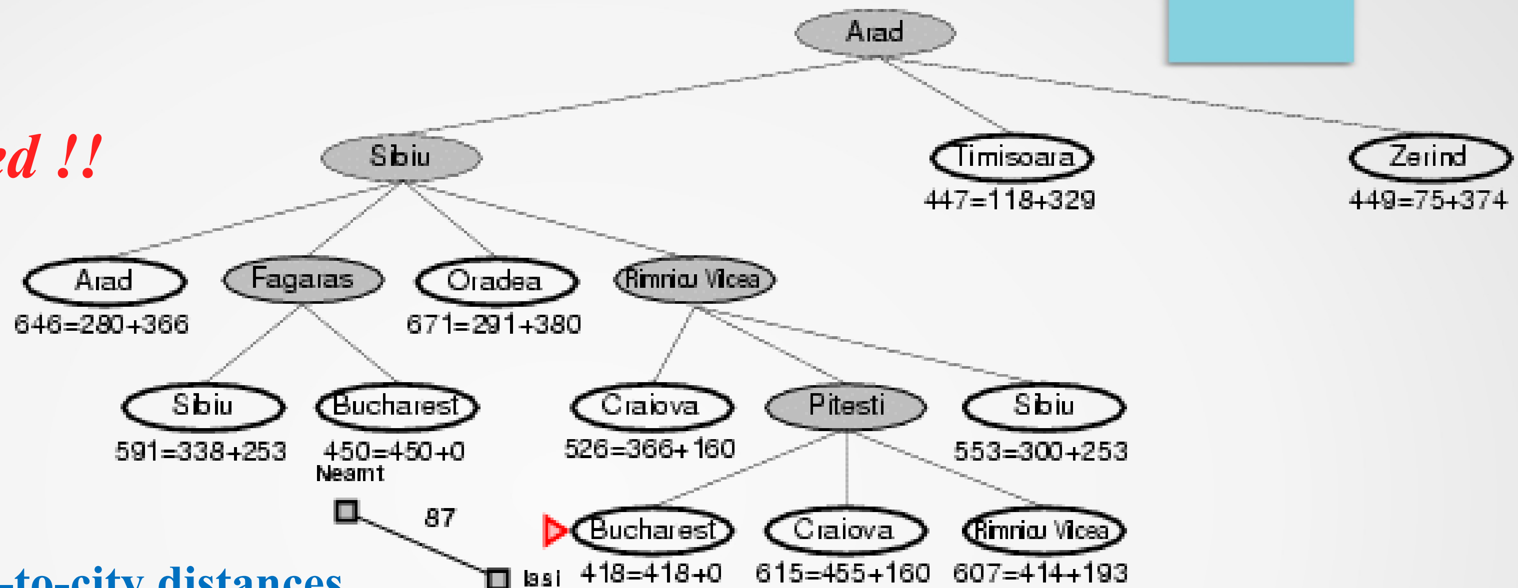


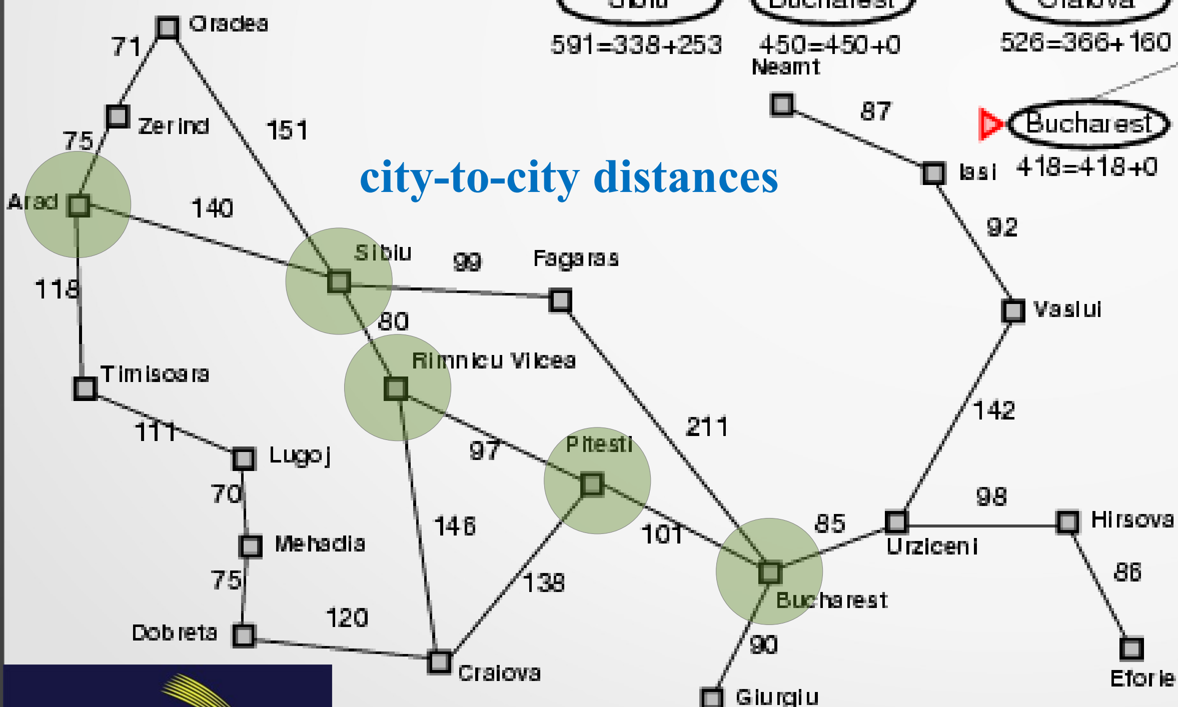
Straight-line distances to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A* Search: Example

Goal reached !!





city-to-city distances

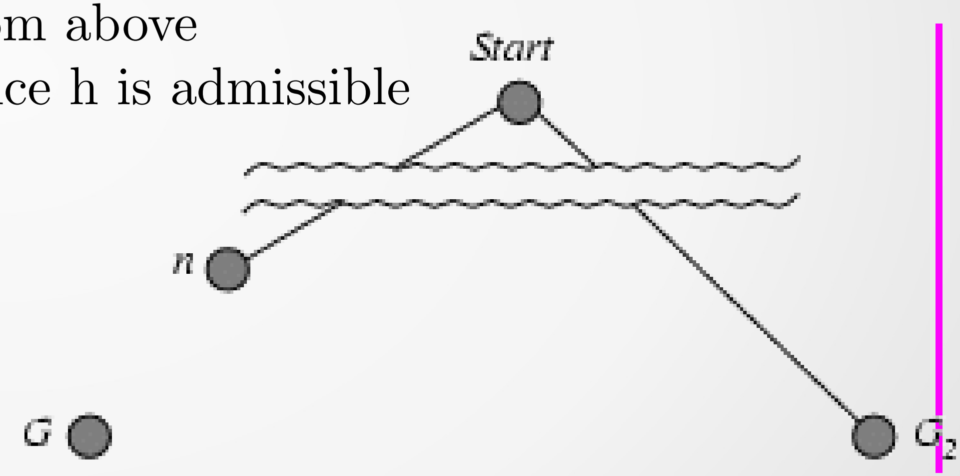
Straight-line distances to Bucharest

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A* Search: Proof of Optimality

- Suppose some suboptimal goal G_2 has been generated and is in the frontier. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.

$$\begin{array}{rcll} f(G_2) & > & f(G) & \text{from above} \\ h(n) & \leq & h^*(n) & \text{since h is admissible} \\ g(n)+h(n) & \leq & g(n)+h^*(n) & \\ f(n) & \leq & f(G) & \end{array}$$

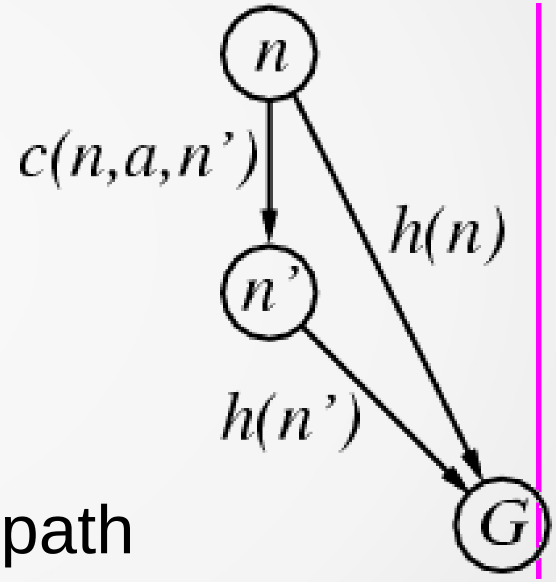


$\therefore f(G_2) > f(n)$ and A* will never select G_2 for expansion

Consistent Heuristics

- A heuristic is **consistent** if for every node *n*, every successor *n'* of *n* is generated by any action *a*

$$h(n) \leq c(n, a, n') + h(n')$$



- If *h* is consistent, we have

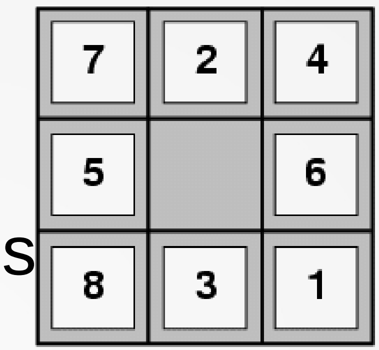
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) = f(n) \end{aligned}$$

i.e. *f(n)* is non-decreasing along any path

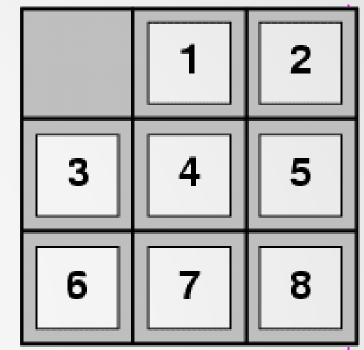
- Theorem: If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal

Admissible Heuristics: 8-puzzle

E.g. for the 8-puzzle



Start State

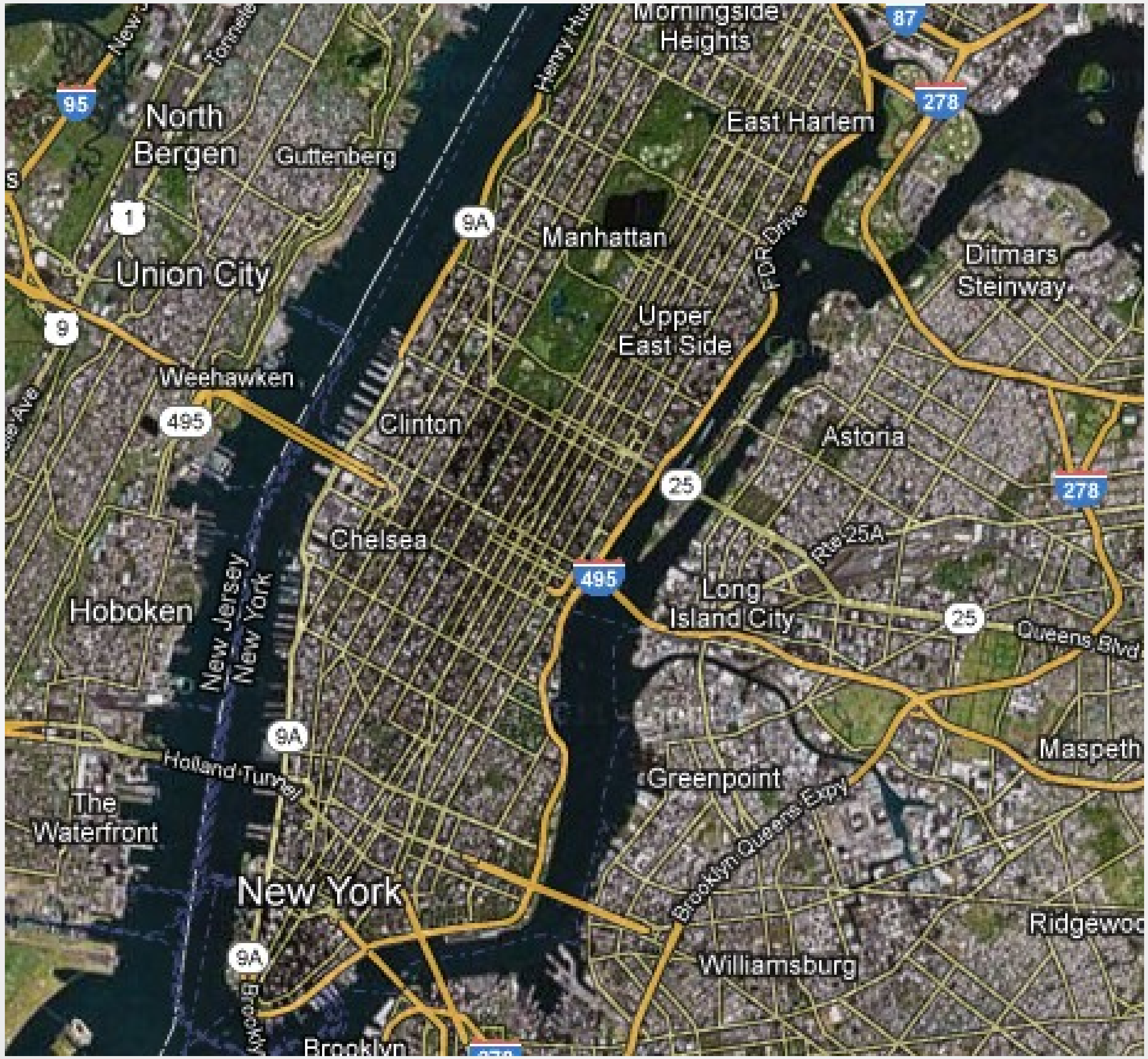


Goal State

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

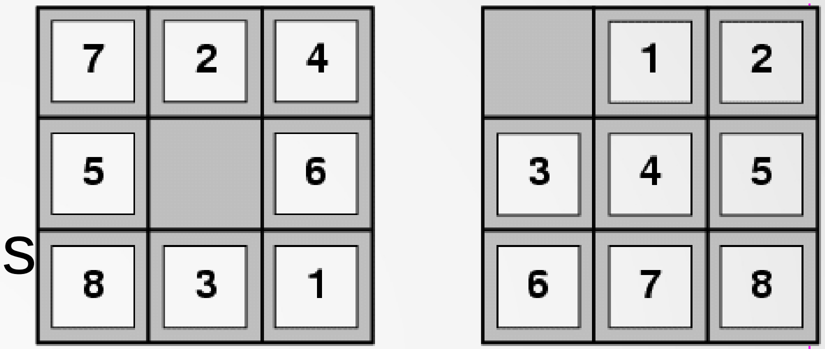
  (i.e. no. of squares from desired location of each tile)
- $\boldsymbol{h_1(S) = ?}$
- $\boldsymbol{h_2(S) = ?}$

Manhattan Distance



Admissible Heuristics: 8-puzzle

E.g. for the 8-puzzle



Start State   Goal State

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

  (i.e. no. of squares from desired location of each tile)
- $\boldsymbol{h_1(S) = 8}$
- $\boldsymbol{h_2(S) = 3+1+2+2+2+3+3+2 = 18}$

A* Search: Drawback

- Requires a lot of memory (still less than uninformed search) since it needs to keep all the generated nodes in memory
- Because of above, not practical for many large-scale problems

Metaheuristics

- A heuristic meant to find/generate/select a heuristic
  - Solution to optimization problem!
- Many are inspired by nature
  - Swarm intelligence
    - Particle Swarm Optimization
    - Ant Colony Optimization
  - Simulated Annealing
  - Evolutionary Programming/Genetic Algorithms

Metaheuristics

- A **higher-level** procedure/heuristic for finding a **sufficiently** good solution (which is itself an algorithm/heuristic)
- Good for search spaces too large to reasonably sample
- Good for incomplete/imperfect information
- Effectively are **strategies** to guide the search process in order to select solutions
  - Usable for a variety of problems (problem-independent)

Metaheuristics

- **Do not guarantee** a globally optimal solution (in general)
- **Not greedy** (in general)
- May sometimes lead to a (temporary) **deterioration of the solution**
  - This allows them to explore the solution space more thoroughly
- Goal is to **efficiently** explore the search space to find a **near-optimal** (good enough) solution

Genetic Algorithms

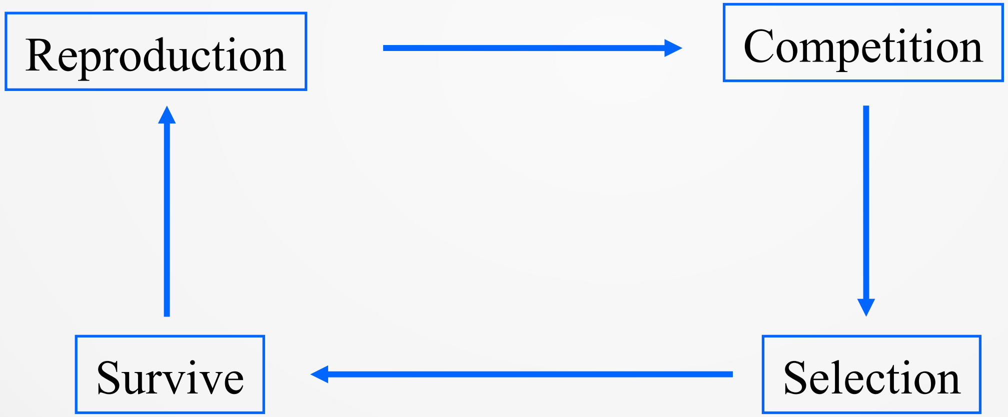
- Search-based optimization techniques based on the principles of **genetics** and **natural selection**
- **Adaptive** heuristic search algorithms that belong to the class of **evolutionary algorithms** (these simulate processes in natural system for evolution)
- Frequently used to solve optimization problems, finding near-optimal solutions to difficult problems

Genetic Algorithms History

- Developed in 1960s by John Holland, students, and colleagues in U. Michigan (including David E Goldberg)
- Result of a formal study of the phenomenon of adaptation as it occurs in nature
- Attempt to import the mechanisms of natural adaptation into computer systems
- Popularity increased in late 1980's
- Based on Darwin's theory of evolution, where the **best** should **survive** and create new **offspring**

Genetic Algorithms History

- "Survival of the Fittest" – the process of **natural selection** which means species who can adapt to changes in their environment are more likely to survive and reproduce, with adaptations passed on

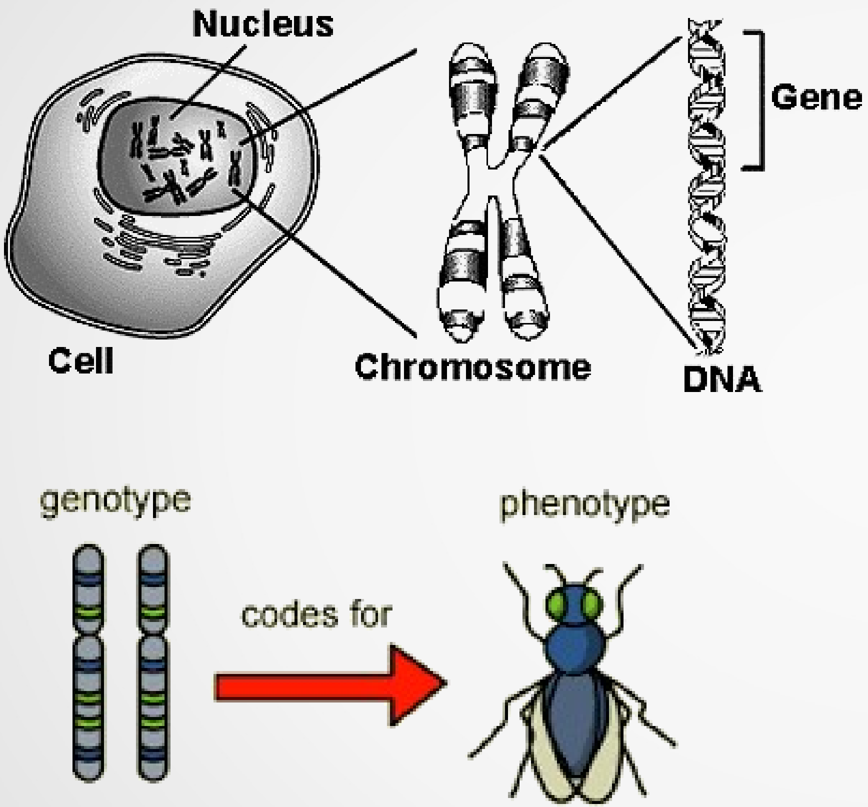


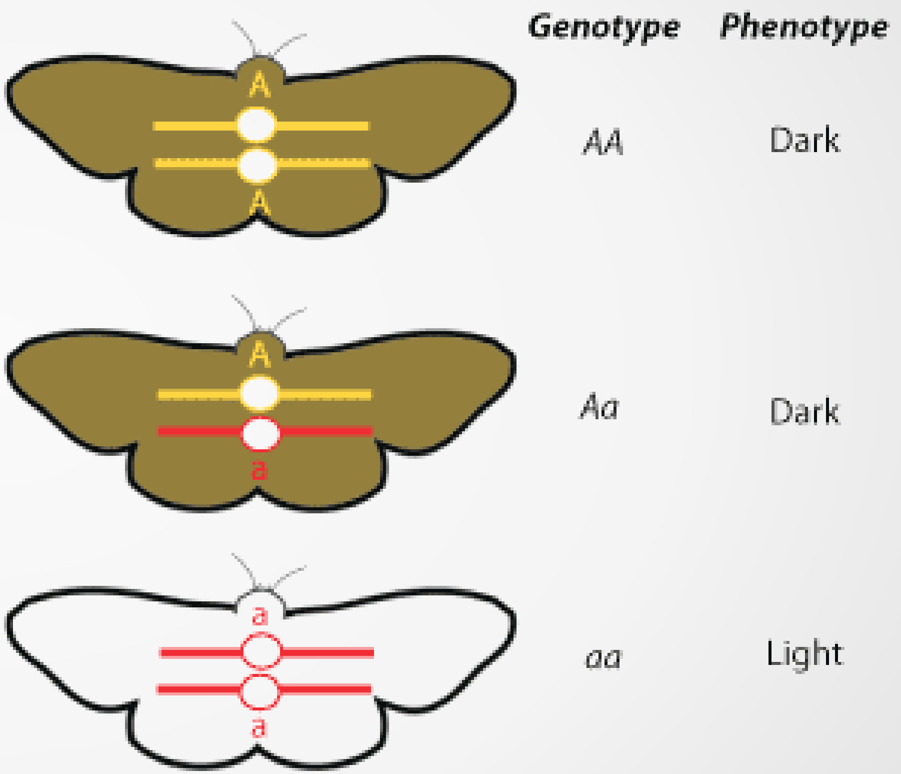
- A robust **search and optimization** mechanism

Biology of Natural Selection

- Cells contain **chromosomes**, which contain **genes**
- Aspects of an organism (e.g. hair colour, height) depend on the organism's genes
- A collection of genes is called a **genotype**
- A collection of traits/characteristics is called a **phenotype**
- **Reproduction** recombines genes from parents, along with some amount of mutation (errors) in copying
- **Fitness** of an organism determines its likelihood of reproducing

Biology of Natural Selection





Biology of Natural Selection

- **Genes** are the basic **building blocks** for an organism
- A **chromosome** is a sequence of genes
- Biologists distinguish between the **genotype** (genes and chromosomes) and the **phenotype** (the actual physical characteristic/expression in the organism)
  - You may have 'tall genes' but a separate medical condition affecting your spine may leave you short
- In genetic algorithms, "genes" may describe a **possible solution** without actually being the solution itself

Finding Solutions

- Suppose you have a problem that you don’t know how to solve...
  - No algorithm for solving it exists/is known
- There are a seemingly unlimited number of possible ways to solve it
  - Search space very large/infinite
- What can you do?

Finding Solutions

- An exhaustive search may work
  - Time taken depends on search space
- Gradient descent may be helpful
  - Directs the exhaustive search, but time taken still depends on search space
- Blind search (random generate and test) may actually be better than the above
  - Need to know what sort of solution is ‘good enough’

Finding Solutions

Try this “smarter” idea

- Generate a set of random solutions
- Repeat the following
  - Test each solution and rank them
  - Remove bad solutions
  - Keep good solutions, duplicate/modify them
- Stop when the solution is ‘good enough’
- GA! Intelligently exploits random search to direct search

Evolution

- Individuals (a candidate solution) in **population** (a set of solutions) **compete** for resources and mating opportunity
- Those individuals who are **successful** (fitness) then **mate** (recombine) to create more **offspring** (better solutions) than others
- Genes from ‘fittest’ parent propagate through the generations, sometimes creating better offspring (than either parent)
- Each successive generation is thus more suited for their environment

Genetic Algorithm Fundamentals

- We have a **pool/population** of possible solutions
- These solutions undergo **recombination** and **mutation** (like in natural genetics) producing new **children** (better than earlier solutions) and the process is repeated over various generations
- Each **individual** (candidate solution) has a **fitness value** (based on its **objective function** value), with the fitter individuals having a higher chance of recombining (mating) to yield “fitter” individuals
- Keep “**evolving**” till we reach a **stopping** criterion till we get a good enough solution

Genetic Algorithms Fundamentals

Nature	Genetic Algorithm
The environment (full of challenges)	Optimization problem
Individuals living in that environment	Feasible solutions (population)
Individual’s degree of adaptation	Solution quality (fitness)
A population of individuals	A set of feasible solutions
Selection, recombination, and mutation	Stochastic operators (selection, crossover, and mutation)
Evolution of populations to suit their environment	Iteratively applying stochastic operators on the set of feasible solutions

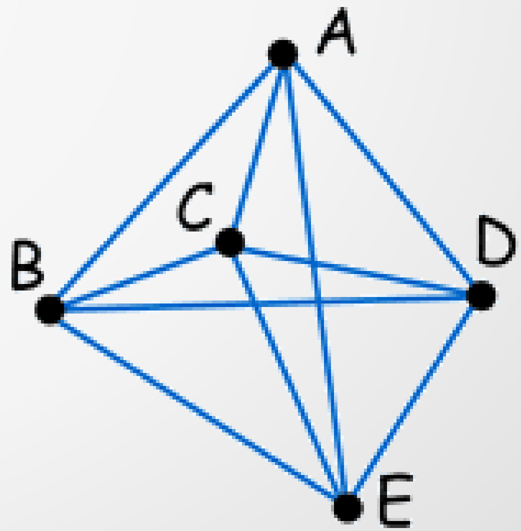
Motivation for Genetic Algorithms

- In general, genetic algorithms are used to deliver a ‘good enough’ (not necessarily the best) solution ‘fast-enough’
  - We are not perfectionists; we are engineers!
- Normally, genetic algorithms are used when the following conditions are fulfilled:-
  - Problem is difficult
  - Gradient-based methods fail
  - Fast (and good) solution needed

Motivation 1: Difficult Problems

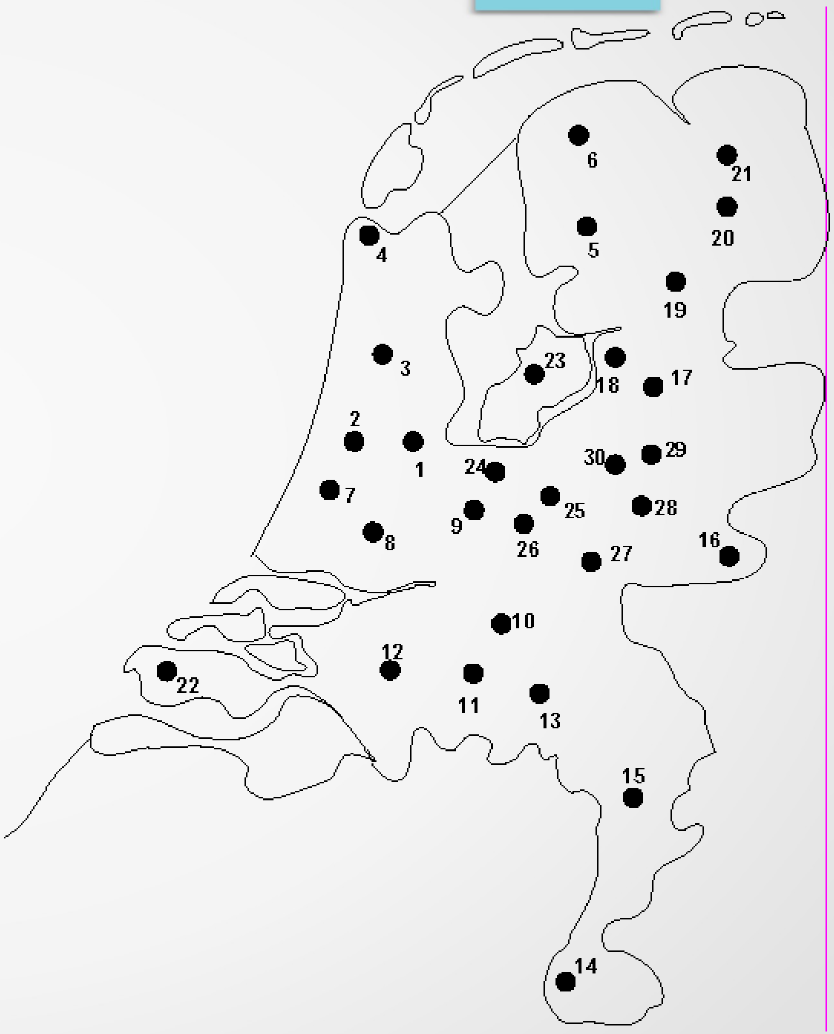
Traveling Salesman Problem (TSP)

- Imagine you need to visit 5 cities (you know all the distances)
- What is the shortest round-trip to follow? ABCDEA? ADECBA?
- How to solve this?
  - Check all possibilities (brute force)
  - Only works for small problems
  - Take factorial time **n!**



Motivation 1: Difficult Problems

- The TSP is an **NP-Complete problem**
- Complexity of such problems (for brute force) is **O(N!)**
- Even the most powerful computing systems could take a very long time (years, decades, centuries) to solve these problems
- GAs can be an efficient tool to provide usable **near-optimal** solutions in a shorter amount of time
- What if a tour guide needed to solve TSP for 30 cities?

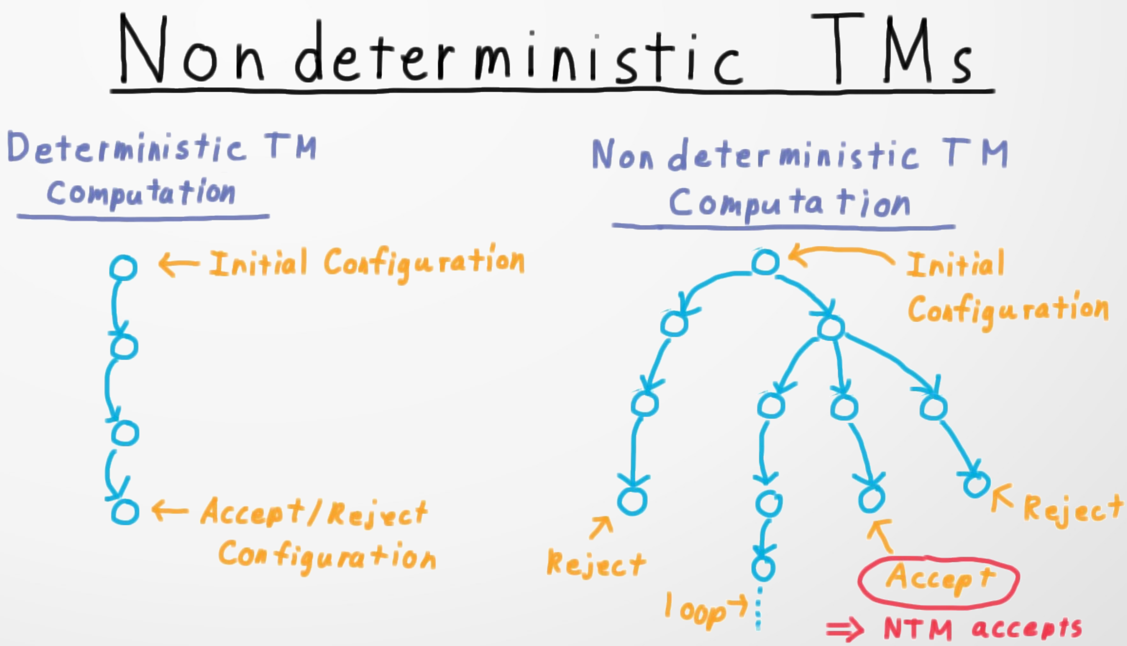


Motivation 1: Difficult Problems

n	*n*!	*n*	*n*!
0	1	25	$1.551121004 \times 10^{25}$
1	1	50	$3.041409320 \times 10^{64}$
2	2	70	$1.197857167 \times 10^{100}$
3	6	100	$9.332621544 \times 10^{157}$
4	24	450	$1.733368733 \times 10^{1000}$
5	120	1000	$4.023872601 \times 10^{2567}$
6	720	3249	$6.412337688 \times 10^{10000}$
7	5,040	10000	$2.846259681 \times 10^{35659}$
8	40,320	25206	$1.205703438 \times 10^{100000}$
9	362,880	100000	$2.824229408 \times 10^{456573}$
10	3,628,800	205023	$2.503898932 \times 10^{1000004}$
11	39,916,800	1000000	$8.263931688 \times 10^{5565708}$
12	479,001,600		
13	6,227,020,800		
14	87,178,291,200		
15	1,307,674,368,000		
16	20,922,789,888,000		
17	355,687,428,096,000		
18	6,402,373,705,728,000		
19	121,645,100,408,832,000		
20	2,432,902,008,176,640,000		

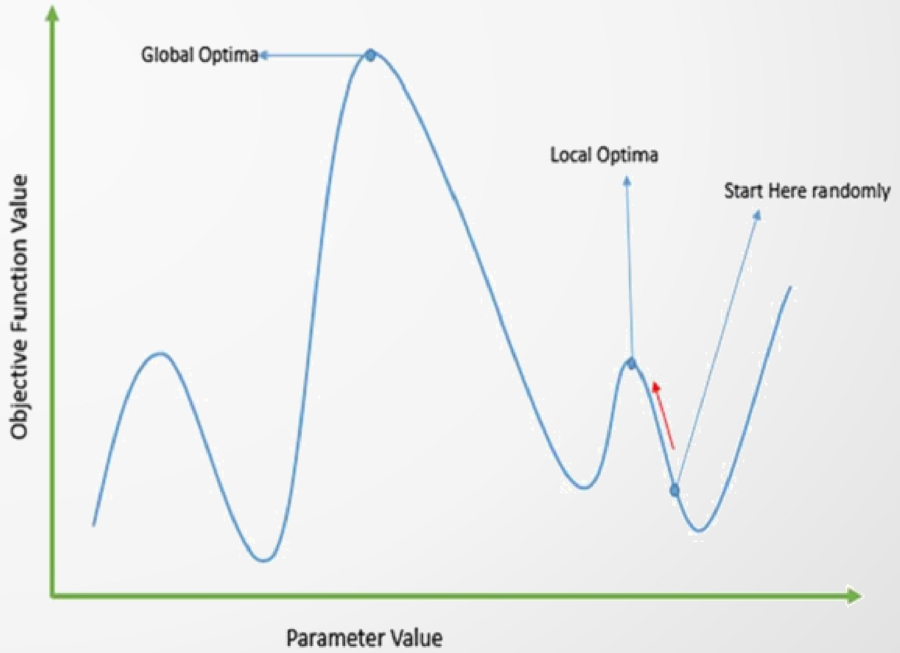
Motivation 1: Difficult Problems

- **Deterministic Polynomial Time**: A Turing Machine takes at most $O(n^c)$ steps for a string of length *n*
- **Non-deterministic Polynomial Time**: A Turing Machine takes at most $O(n^c)$ steps on <u>each computation path</u> for a string of length *n*



Motivation 2: Gradient Based Methods Fail

- Gradient-based methods start at a random point and move in the gradient direction to reach maxima/minima
- This can be very efficient for **single-peaked objective functions**
- In most real-world situation, the problem space has **many peaks** and **many valleys**, so gradient based methods will fail due to local optima



Motivation 3: Fast (and Good) Solution Needed

- Some **difficult problems** (like the TSP) have real-world applications like path finding and VLSI design
- For example, GPS navigation systems need to compute the ‘optimal’ path from one arbitrary location to another
- Multi-hour (or even multi-minute) computations are not acceptable for such applications, even if the results are perfect
- A ‘good enough’ solution which is delivered on time (a few seconds) is required

End of Lecture