DD2380

Artificial Intelligence

Lecture 5
Search

Patric Jensfelt and Jana Tumova

Problem solving via search

- The problem is composed of 4 components:

–An initial state

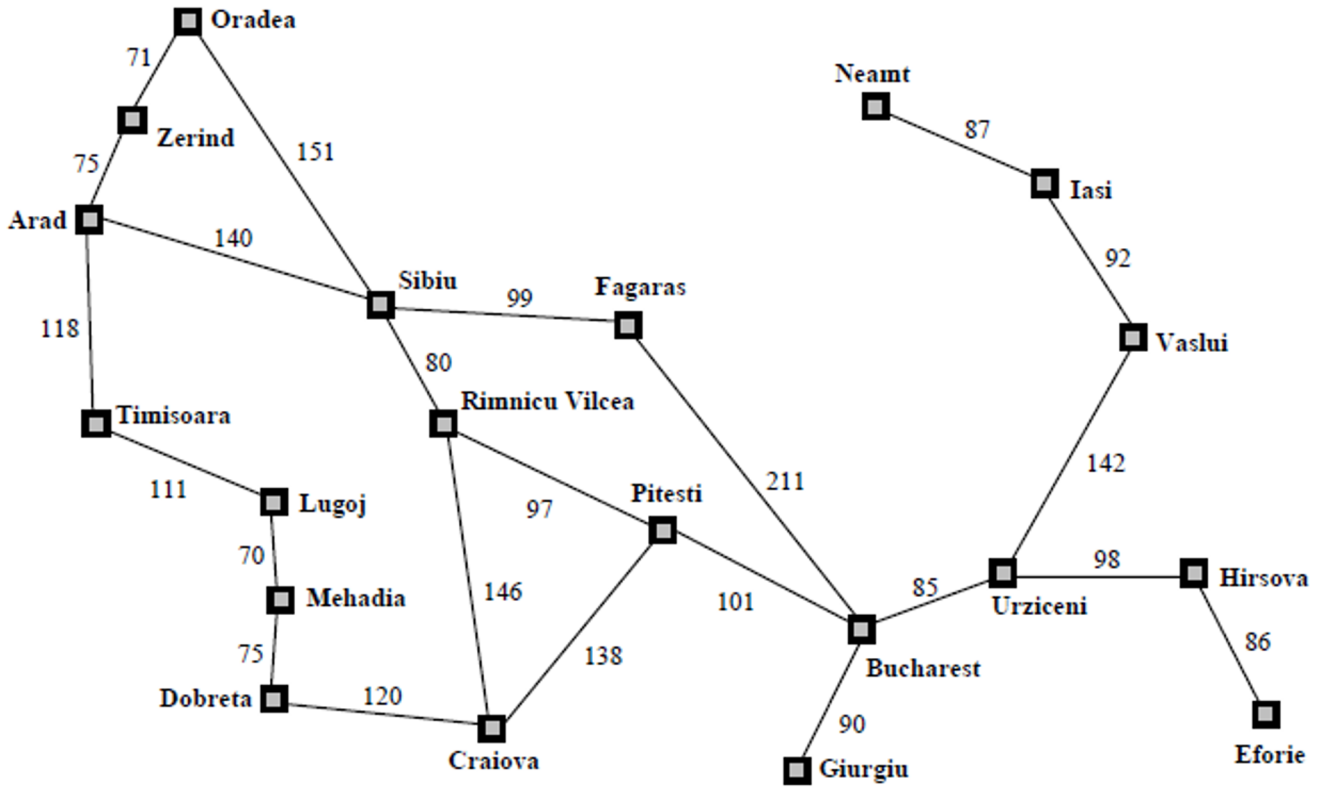
–Successor function, new_state=f(old_state, action)

–A goal test

–A path cost specification

- State space is the set of all states *reachable* from the initial state
- A solution is a sequence of actions.
- An optimal solution is a solution with minimum cost.

Example: Routing problem



Initial state:
Arad

Successor function:
Zerind = f(Arad,…)
…

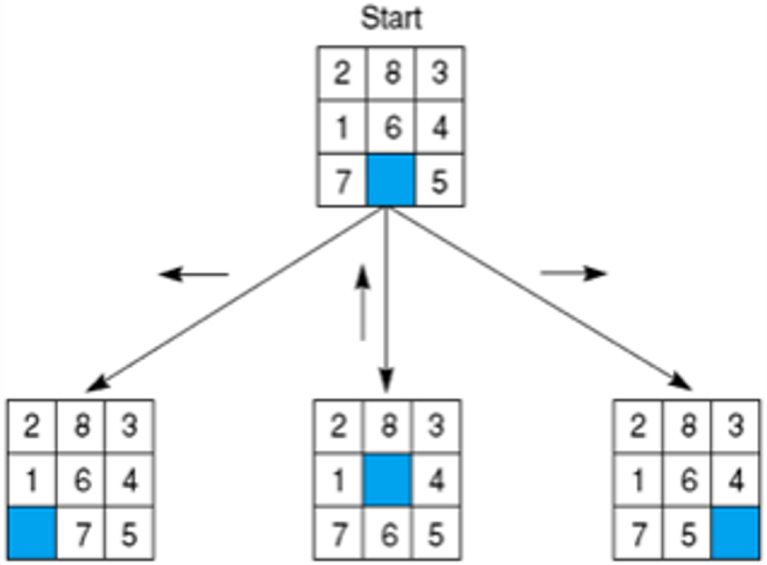
Goal test:
Is the state Bucharest?

Path cost specification: distance traveled

Example: 8-puzzle

Initial state:

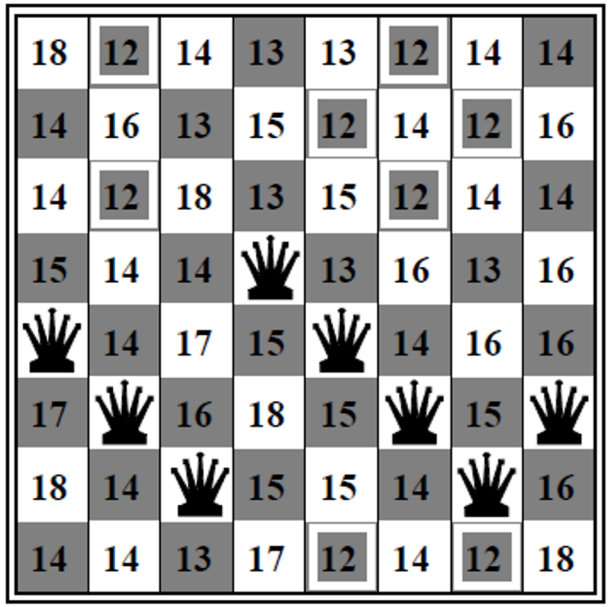
Successor function:



Goal test:

Path cost specification: number of moves

Example: 8-queens



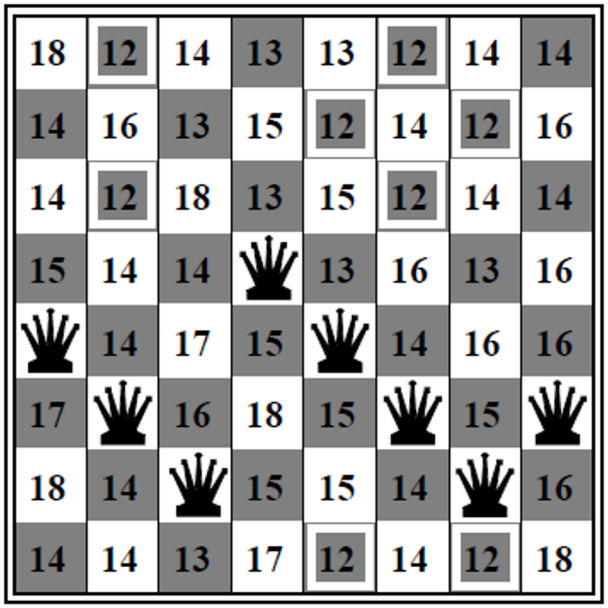
Initial state: ?

Successor function: ?

Goal test: ?

Path cost specification: ?

Example: 8-queens



Initial state:
An empty chessboard

Successor function:
Adding one queen to the board or moving one queen on the board

Goal test:
No queen is under attack

Path cost specification: Number of moves

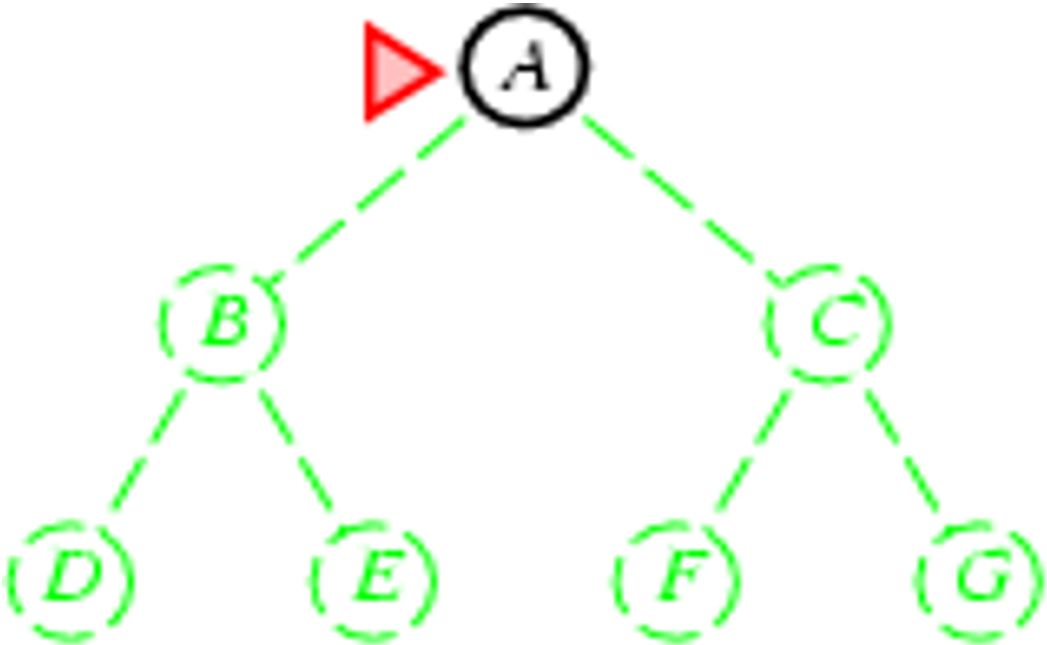
Characterization of search methods

- Time and space complexity is measured in terms of
  - **b** branch factor (max num of successors of any node)
  - **d** depth of optimal solution
  - **m** maximum length of a path in the spate space (could be ∞)

Breadth-first search (BFS)

- Expand the shallowest unexpanded node
- Implementation:

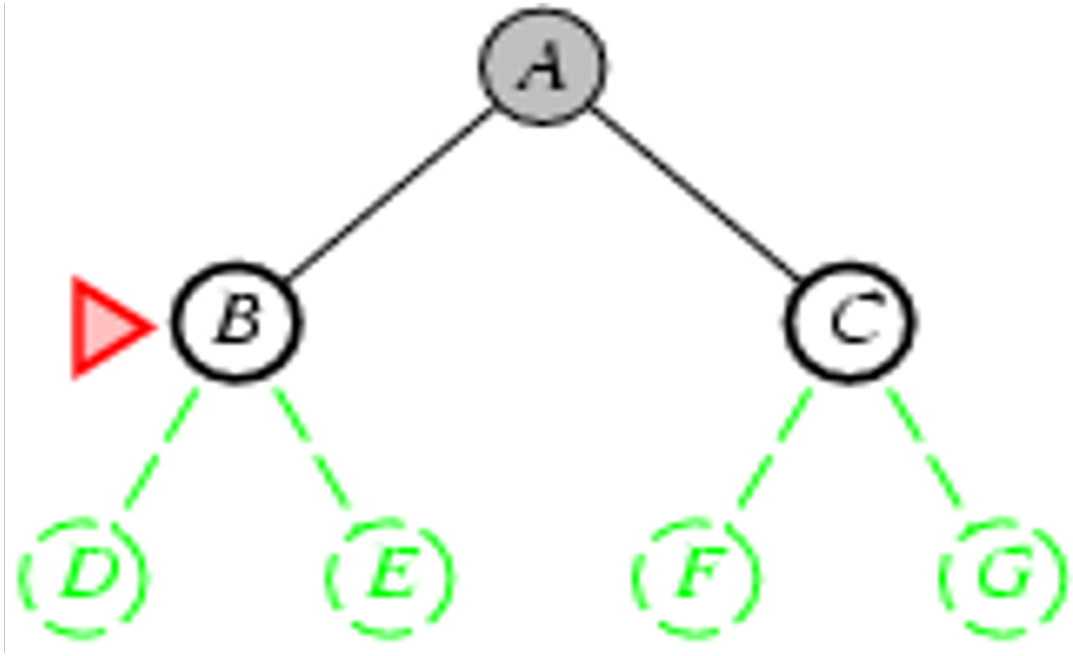
–*fringe* is a FIFO queue, i.e., new successors go at the end



Breadth-first search

- Expand the shallowest unexpanded node
- Implementation:

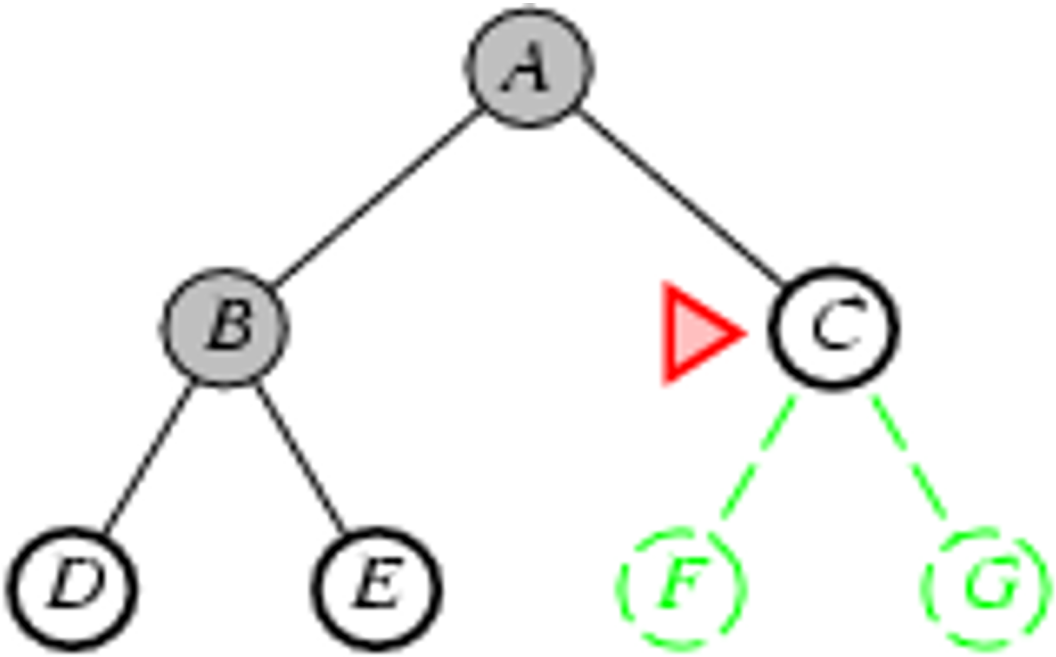
–*fringe* is a FIFO queue, i.e., new successors go at the end



Breadth-first search

- Expand the shallowest unexpanded node
- Implementation:

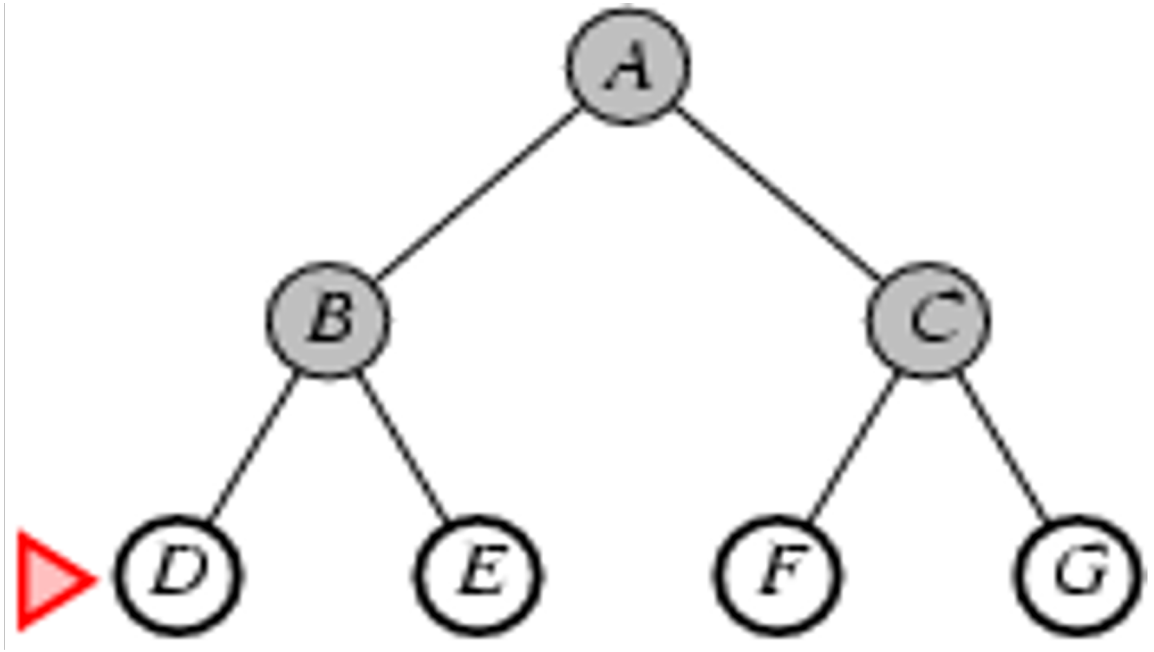
–*fringe* is a FIFO queue, i.e., new successors go at the end



Breadth-first search

- Expand the shallowest unexpanded node
- Implementation:

–*fringe* is a FIFO queue, i.e., new successors go at the end

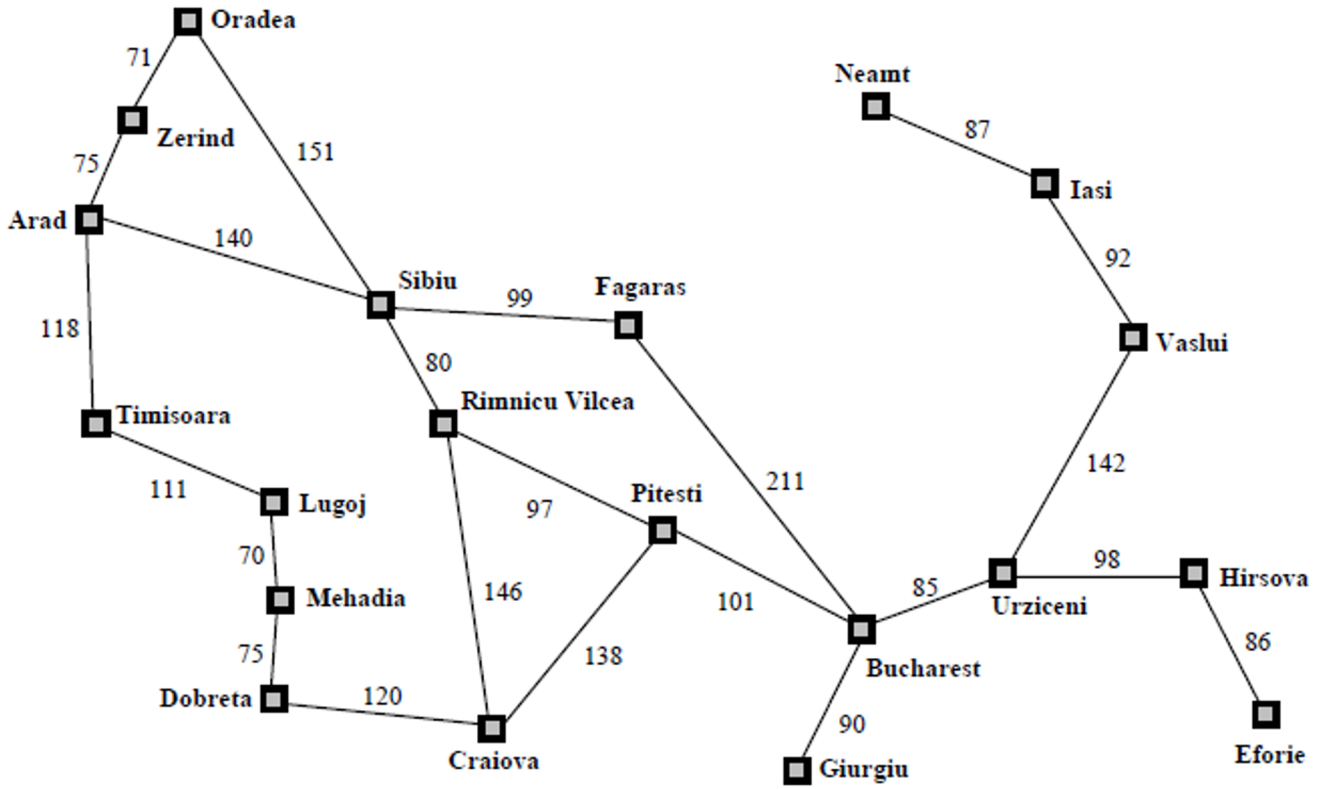


Breadth-first search properties
(Tree version with goal test at node generation)

- Completeness: Yes
- Time: $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- Space: keeps all nodes in memory $O(b^d)$
- Space can easily become a bottleneck
- Optimal: Not in general

– **b** branch factor
– **d** depth of optimal solution

Example: Routing problem



Initial state:
Arad

Successor function:
Zerind = f(Arad,…)
…

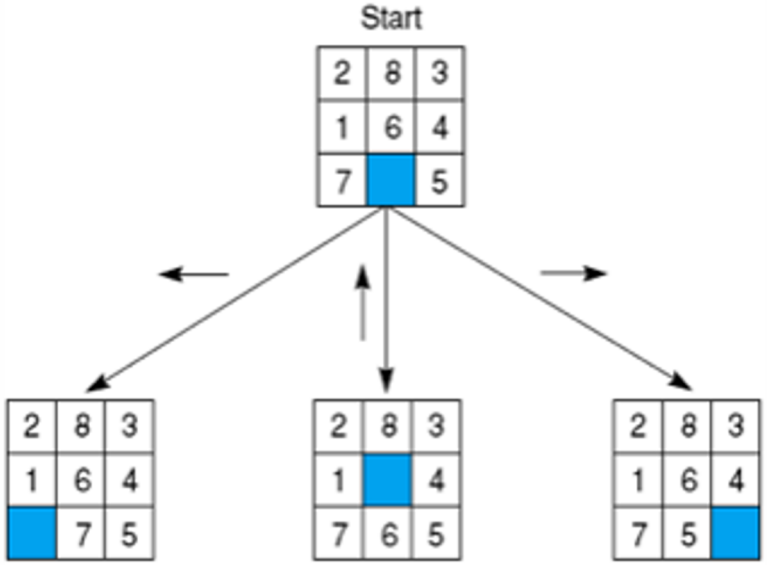
Goal test:
Is the state Bucharest?

Path cost specification: distance traveled

Example: 8-puzzle

Initial state:

Successor function:

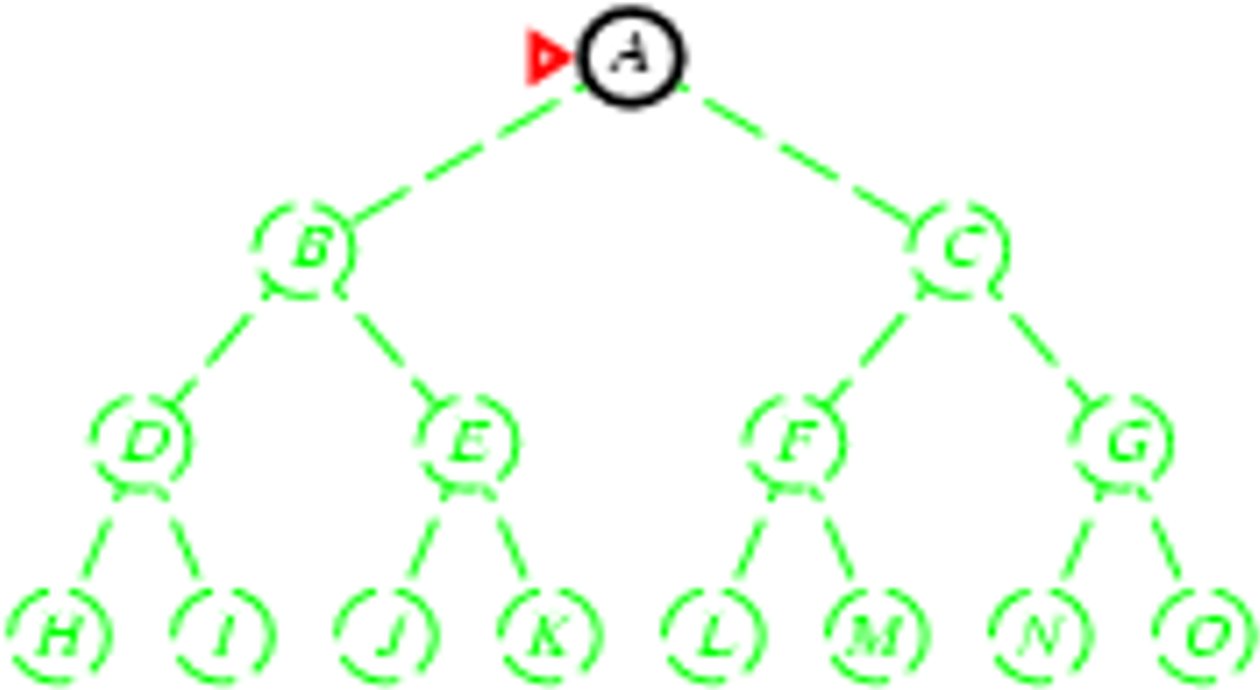


Goal test:

Path cost specification: number of moves

Depth-first search

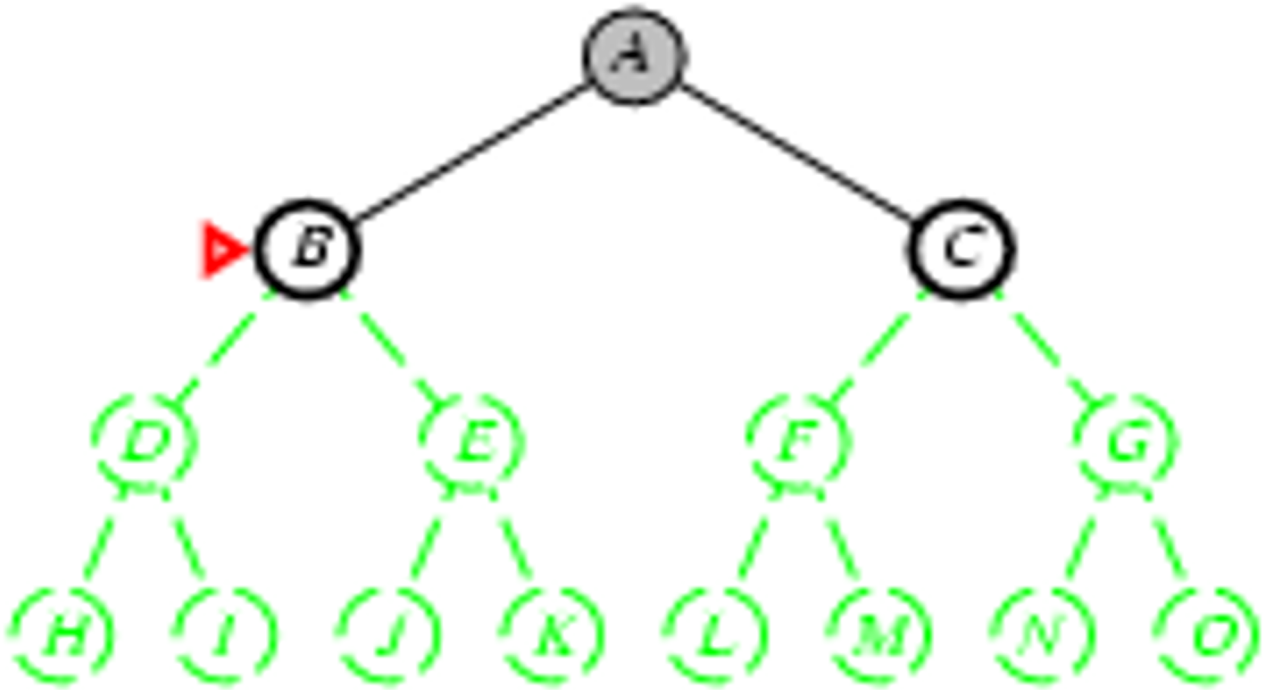
- Expand the deepest unexpanded node
- Implementation:
  - *fringe* = LIFO stack, i.e., put successors at front



Depth-first search

- Expand the deepest unexpanded node
- Implementation:

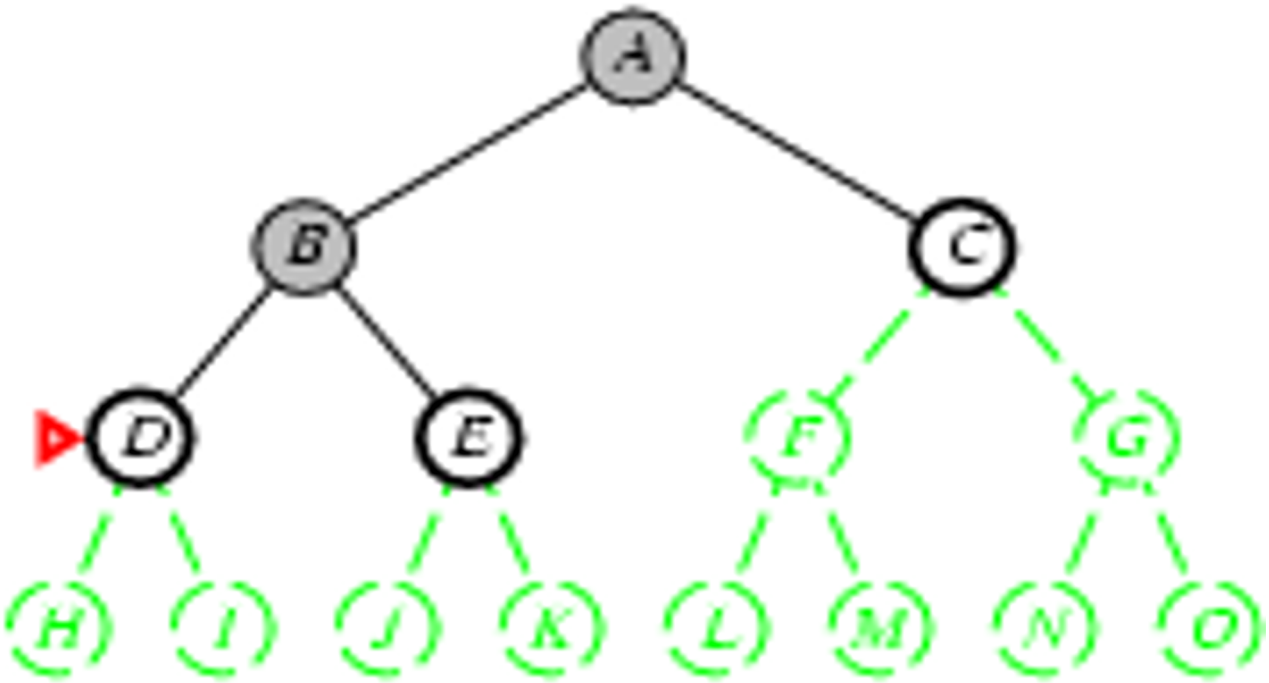
–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

- Expand the deepest unexpanded node
- Implementation:

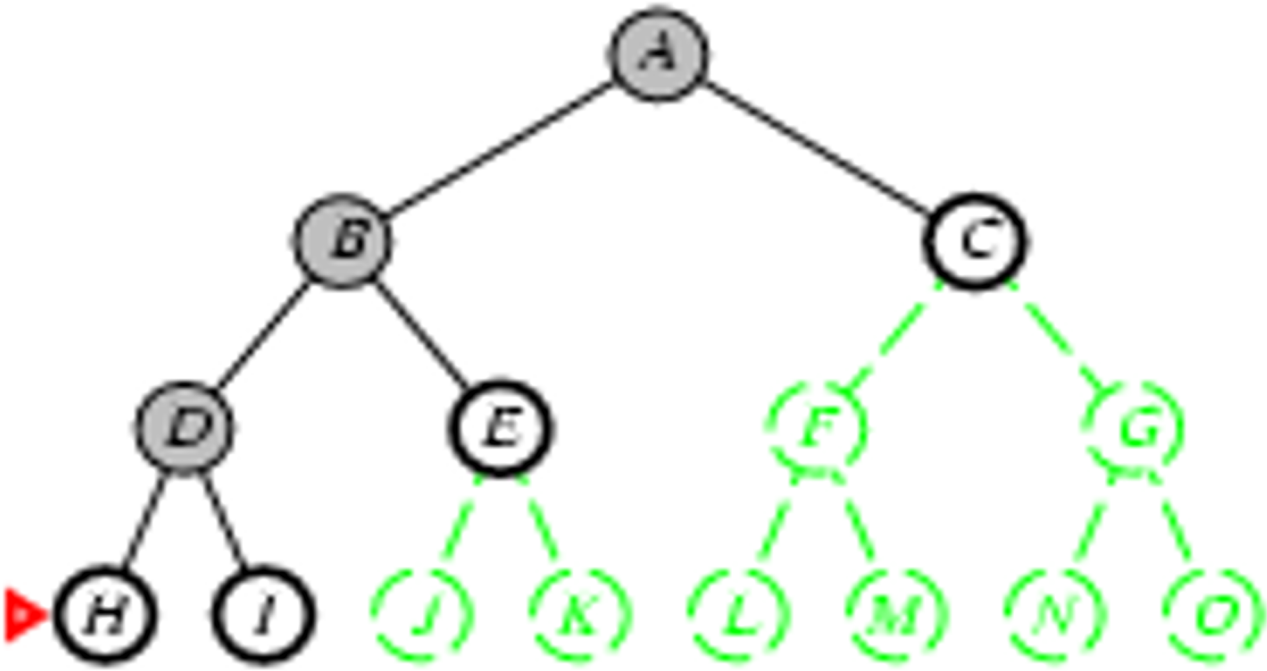
–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

- Expand the deepest unexpanded node
- Implementation:

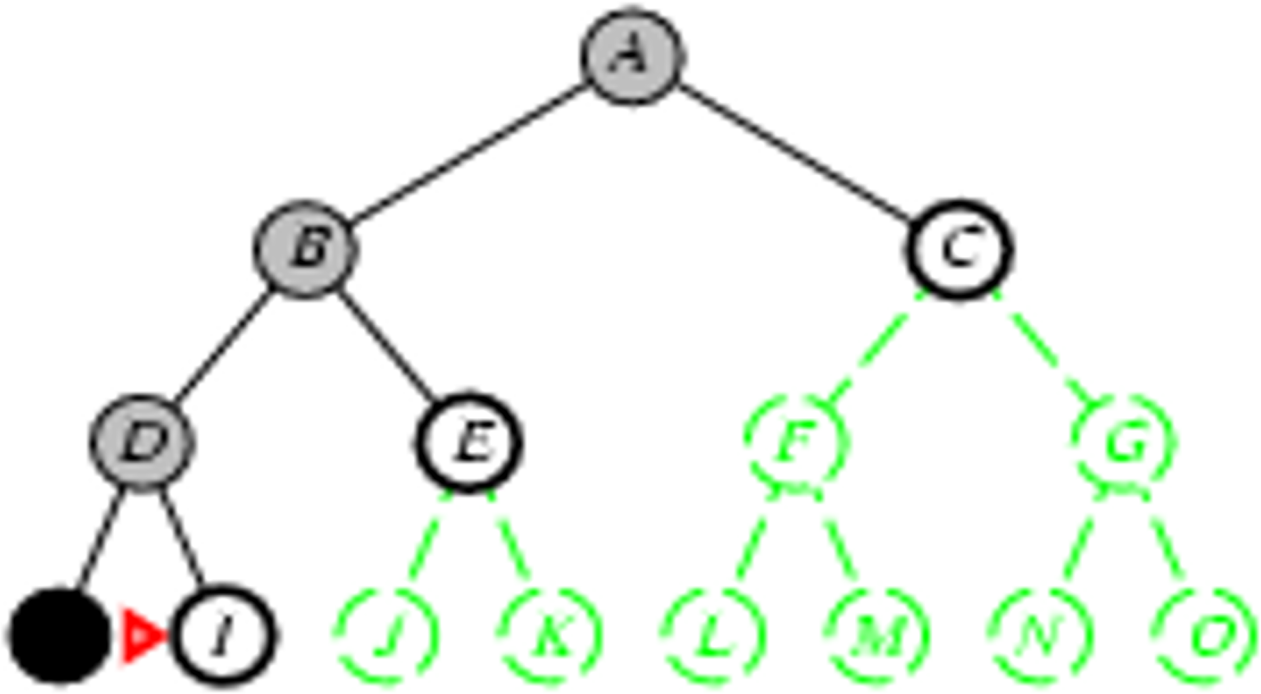
–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

- Expand the deepest unexpanded node
- Implementation:

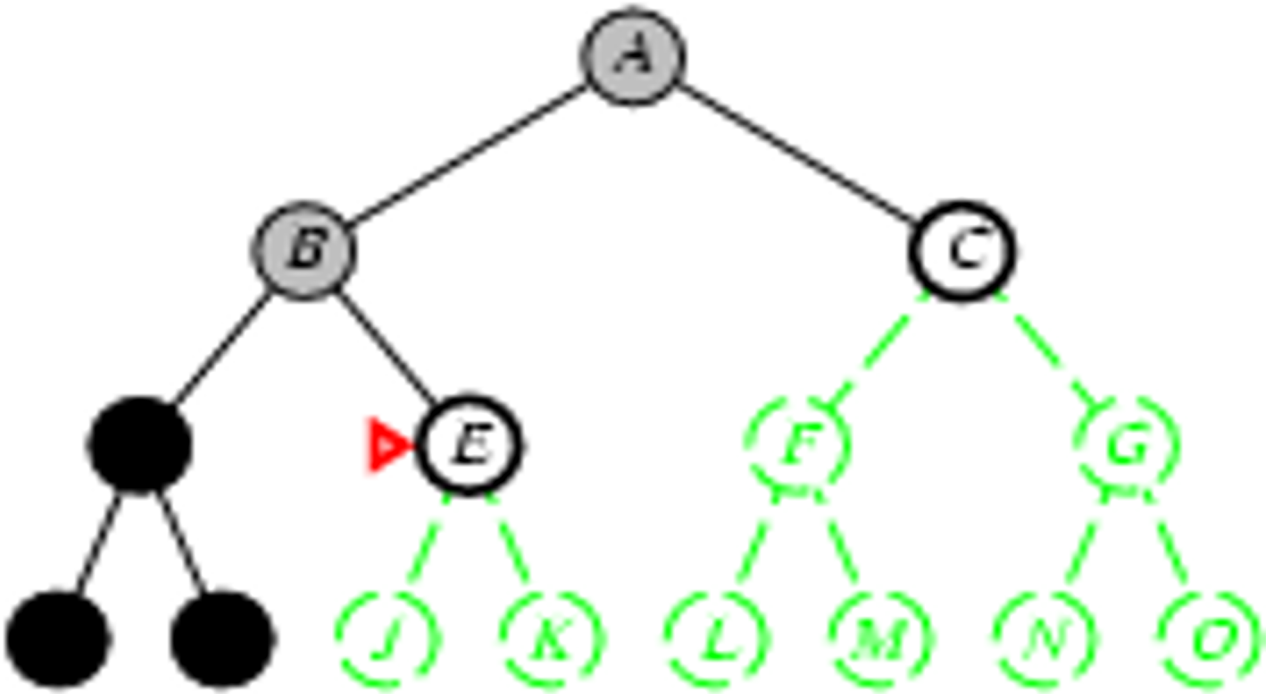
–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

- Expand the deepest unexpanded node
- Implementation:

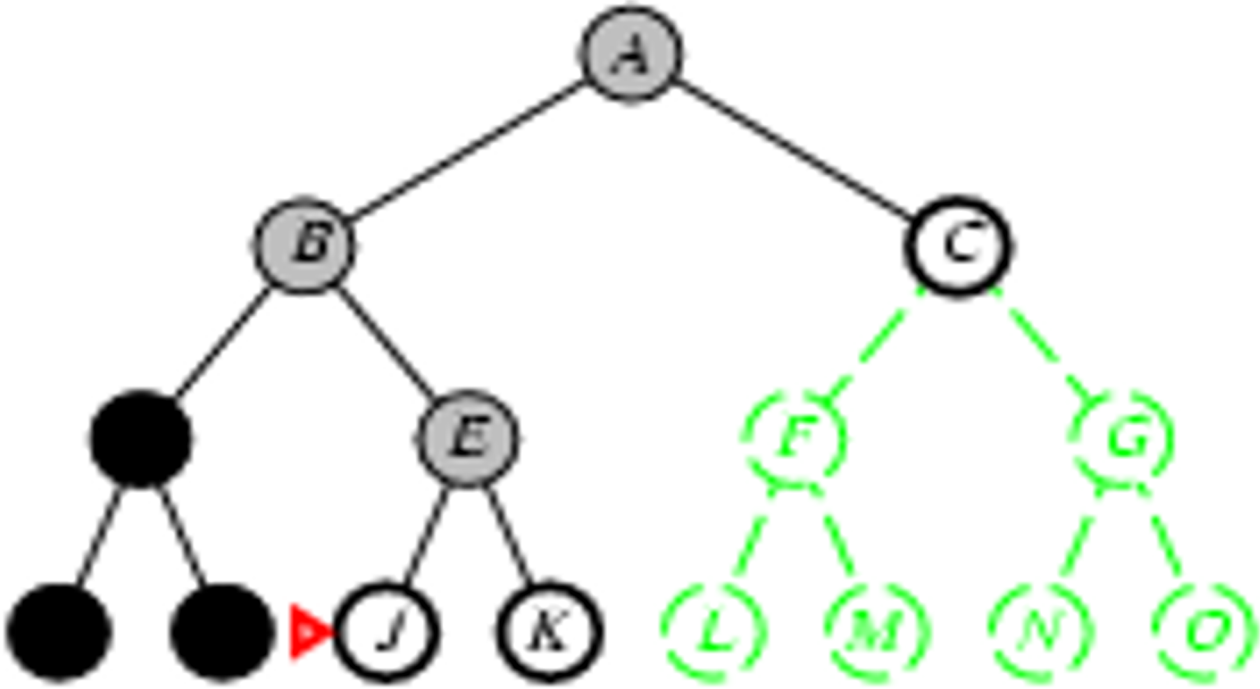
–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

- Expand the deepest unexpanded node
- Implementation:

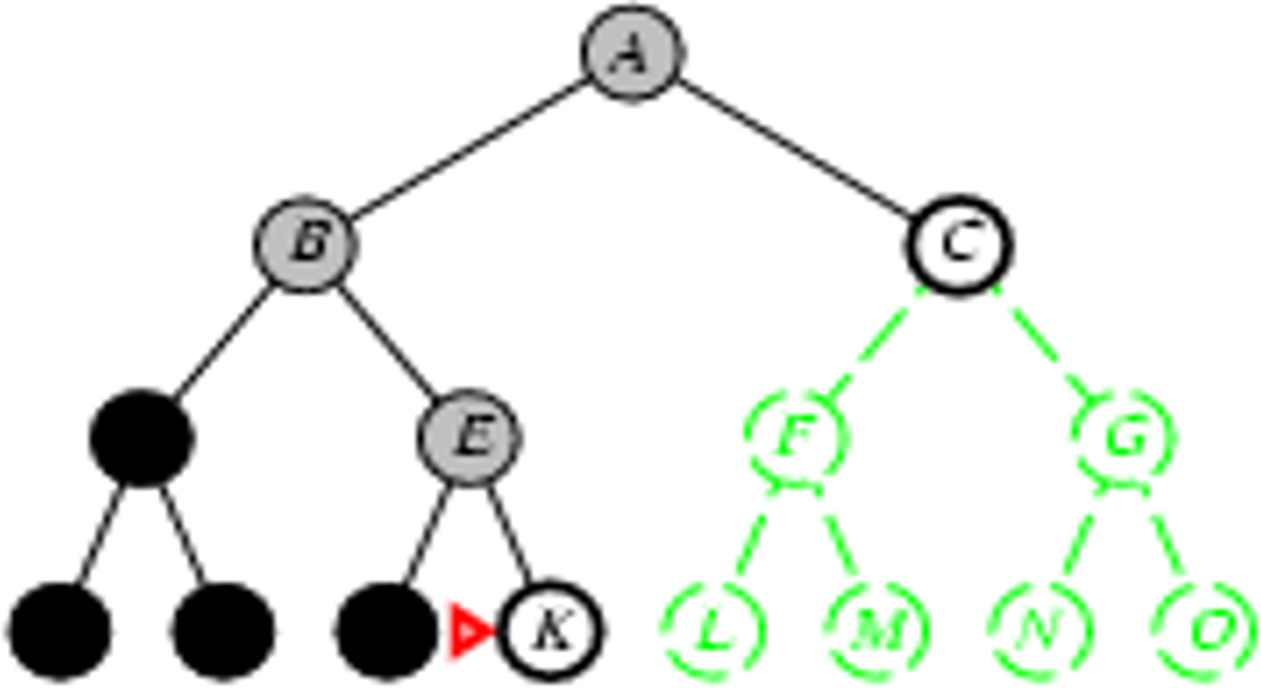
–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

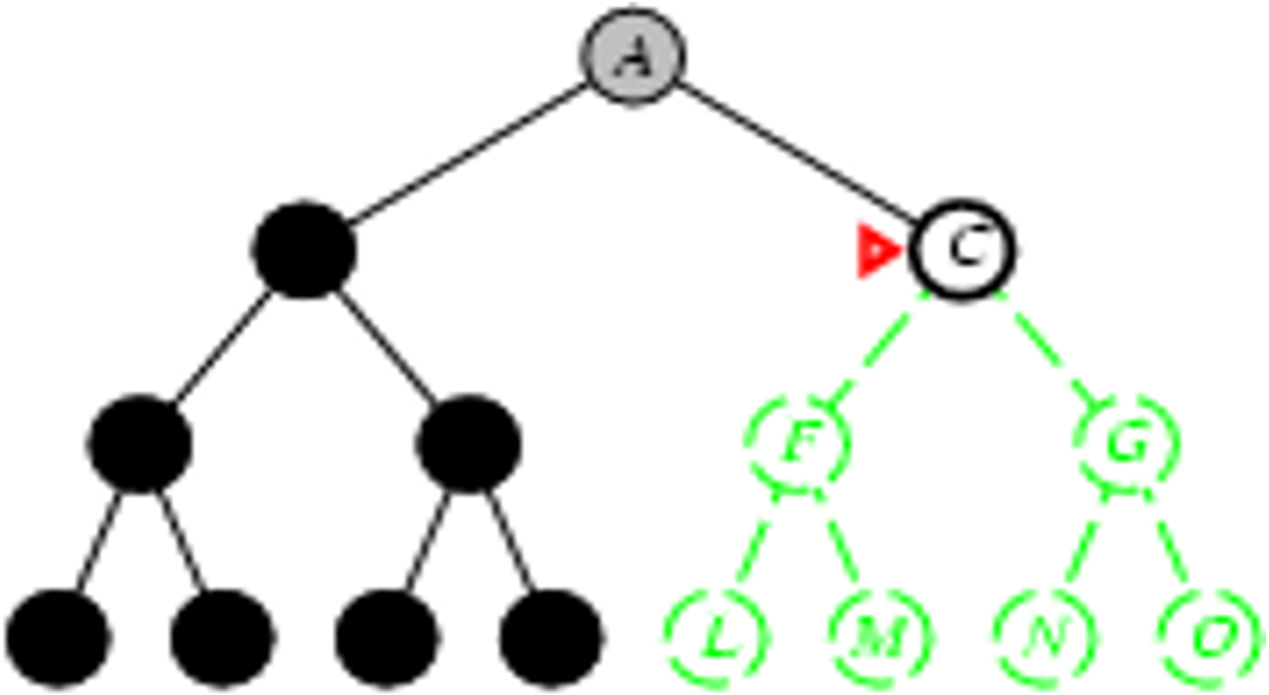
- Expand the deepest unexpanded node
- Implementation:

–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

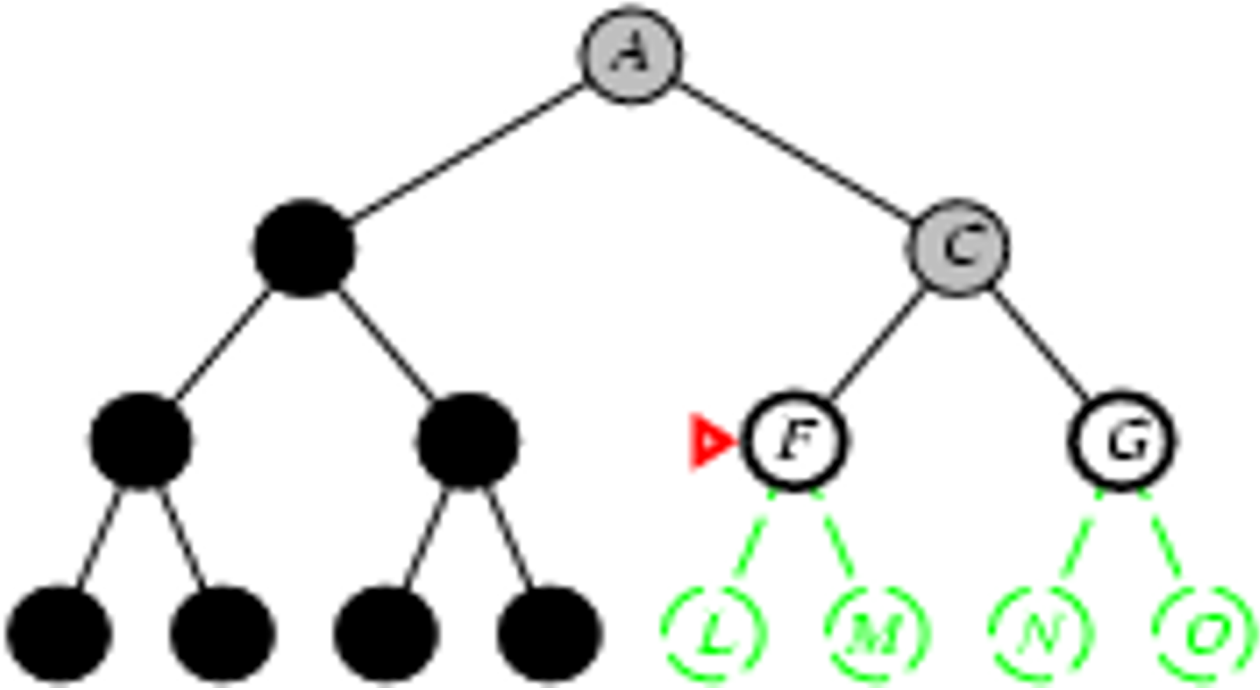
- Expand the deepest unexpanded node
- Implementation:
  - *fringe* = LIFO stack, i.e., put successors at front



Depth-first search

- Expand the deepest unexpanded node
- Implementation:

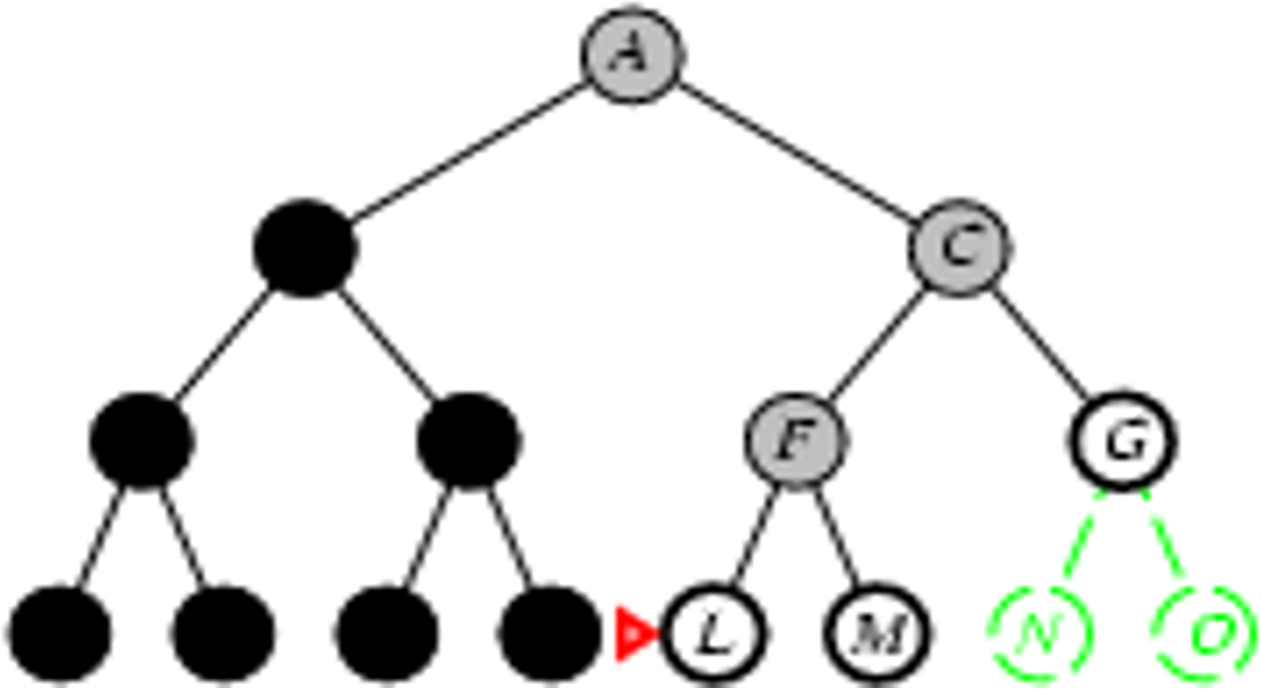
–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

- Expand the deepest unexpanded node
- Implementation:

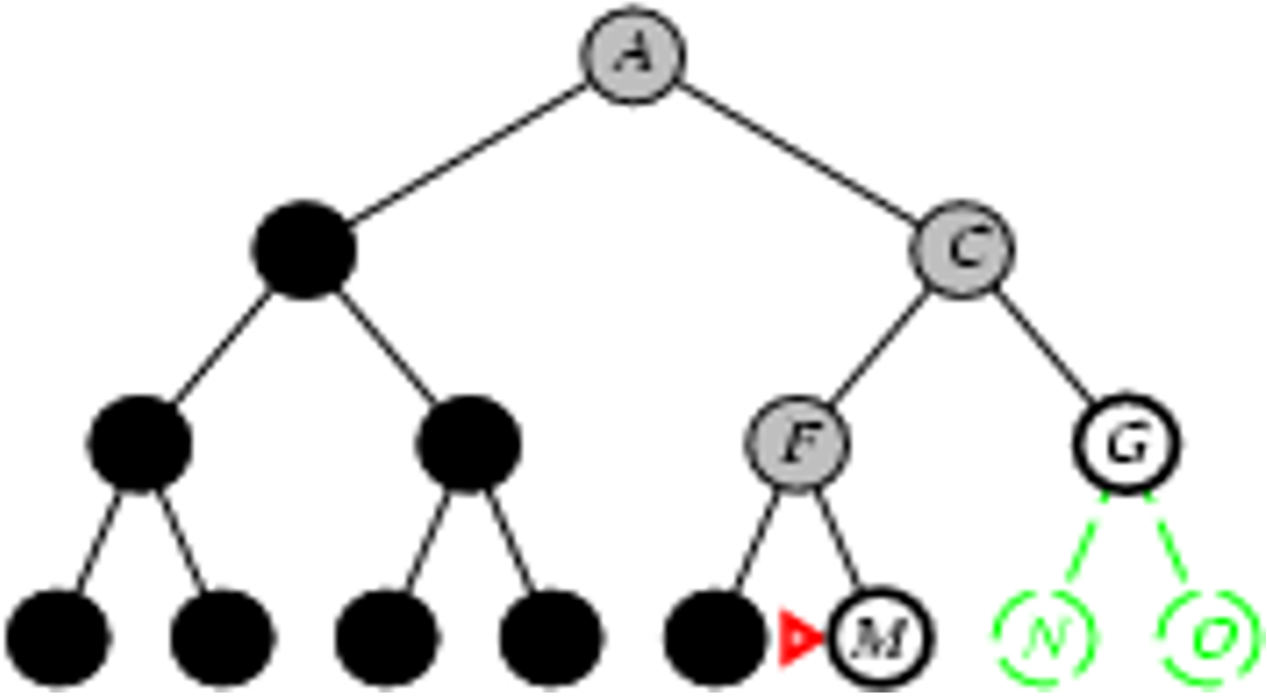
–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search

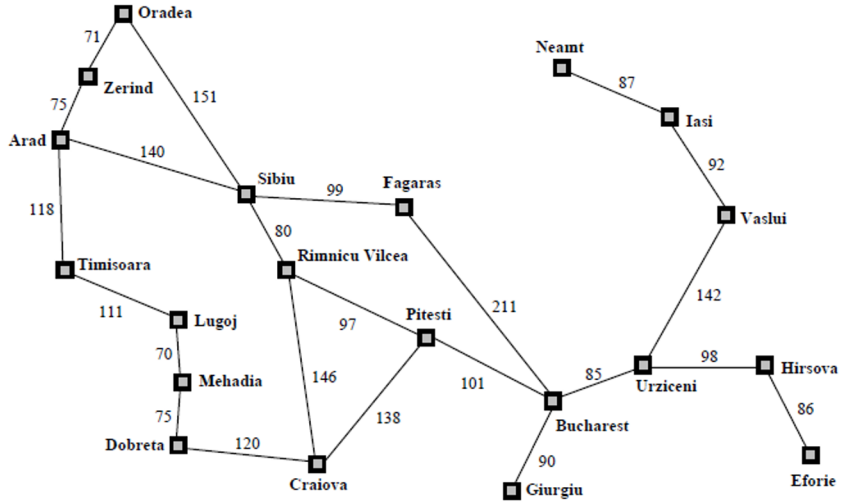
- Expand the deepest unexpanded node
- Implementation:

–*fringe* = LIFO stack, i.e., put successors at front



Depth-first search properties
(Tree version with goal test at node generation)

- Completeness: Fails in spaces with loops
- Time: $O(b^m)$ - m could be much larger than d
- Space: O(bm) - linear in space
- Optimal: No

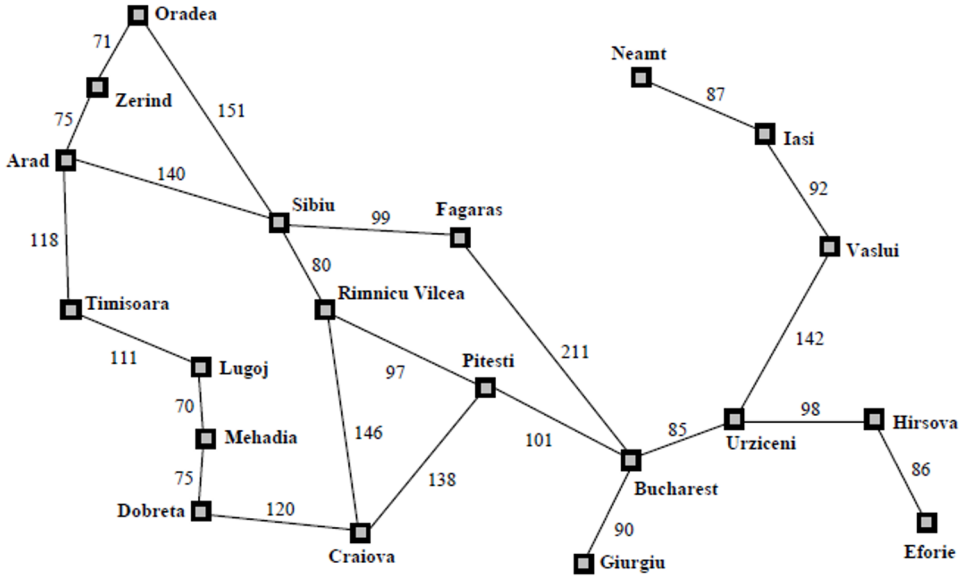


– **b** branch factor

– **d** depth of optimal solution

Tree vs. graph search variant

- Without vs. with remembering which states have been visited already
- DFS graph search is complete



BFS vs. DFS properties

	BFS	DFS tree	DFS graph
Completeness	Yes	No	Yes
Time	$O(b^d)$	$O(b^m)$	$O(b^m)$
Space	$O(b^d)$	$O(bm)$	$O(bm)$
Optimality	sometimes	no	no

DFS variation: Depth-limited search

- DFS with depth limit l
  - Do not expand nodes a depth l

- If we pick l < d we are in trouble

- How to set the depth limit if the depth is unknown?

DFS variation: Iterative deepening search (IDS)

- Idea: Iterate depth-limited search with increasing depth limit.

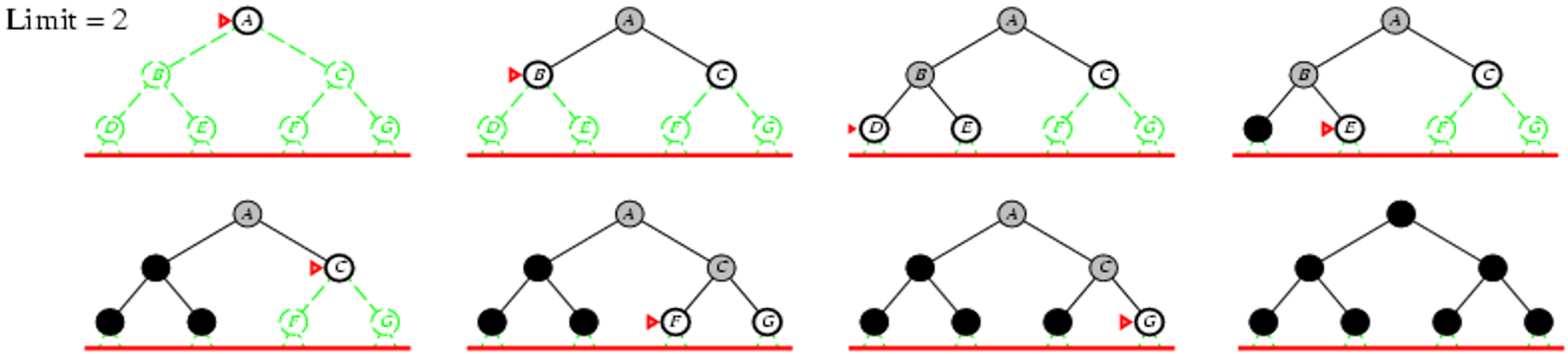
Iterative deepening search / =0

Limit = 0

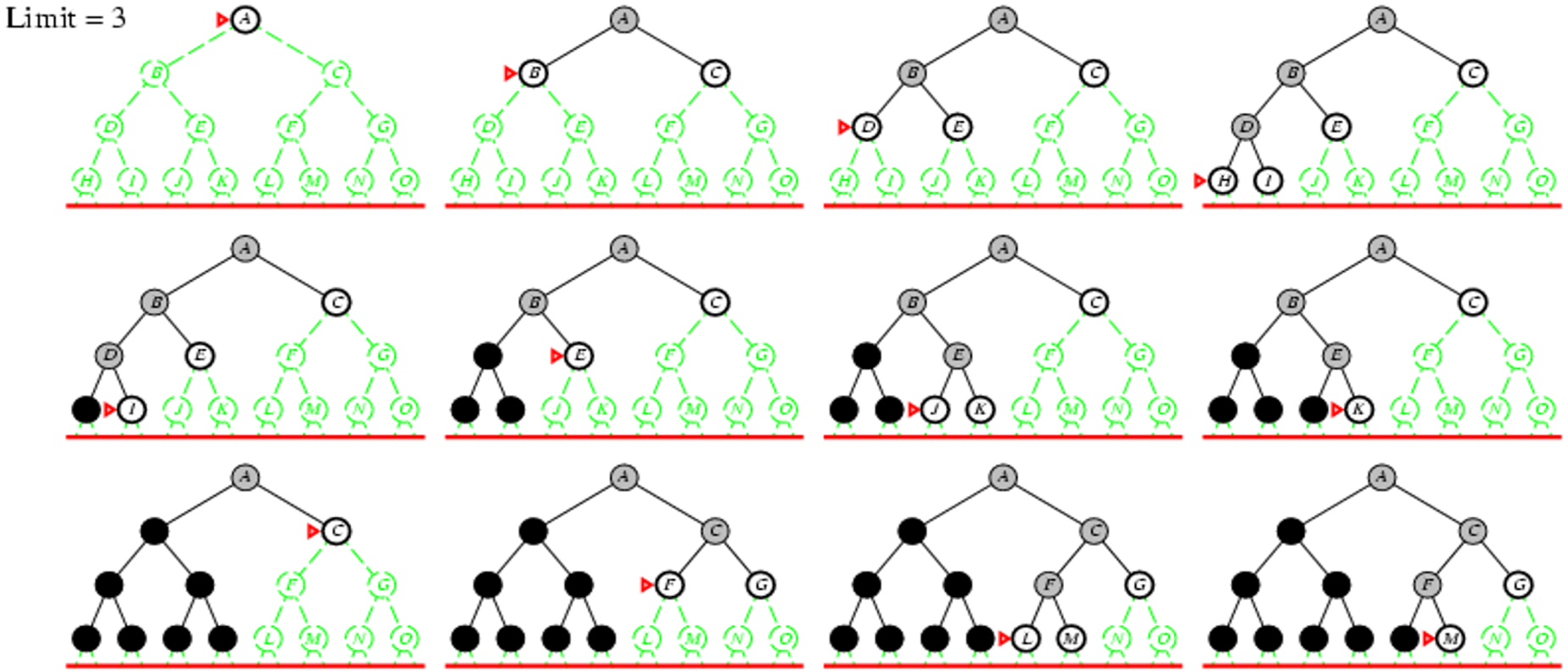
Iterative deepening search / =1

Limit = 1

Iterative deepening search / =2



Iterative deepening search / =3



Iterative deepening search (IDS)

- Does this not lead to a massive waste of time?

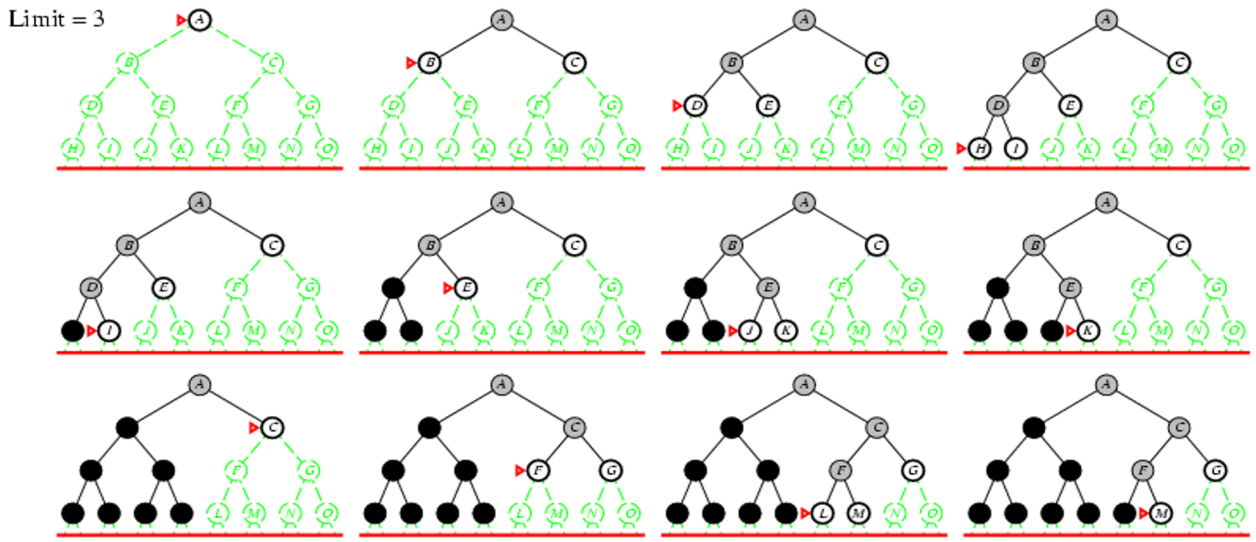
Iterative deepening search

- 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*:

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



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*:

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

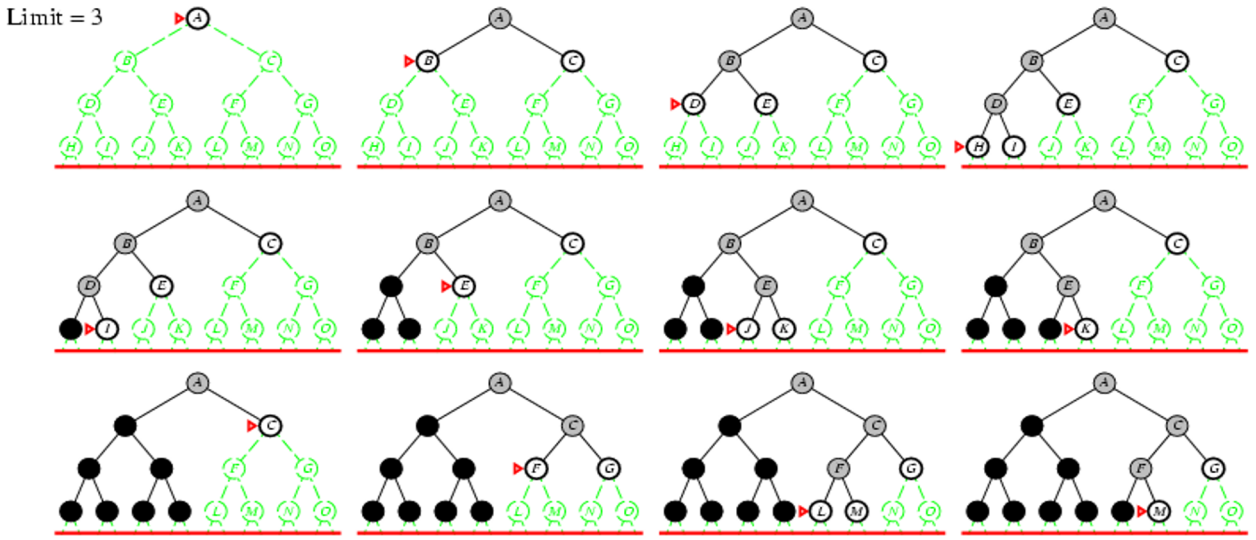
- For *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

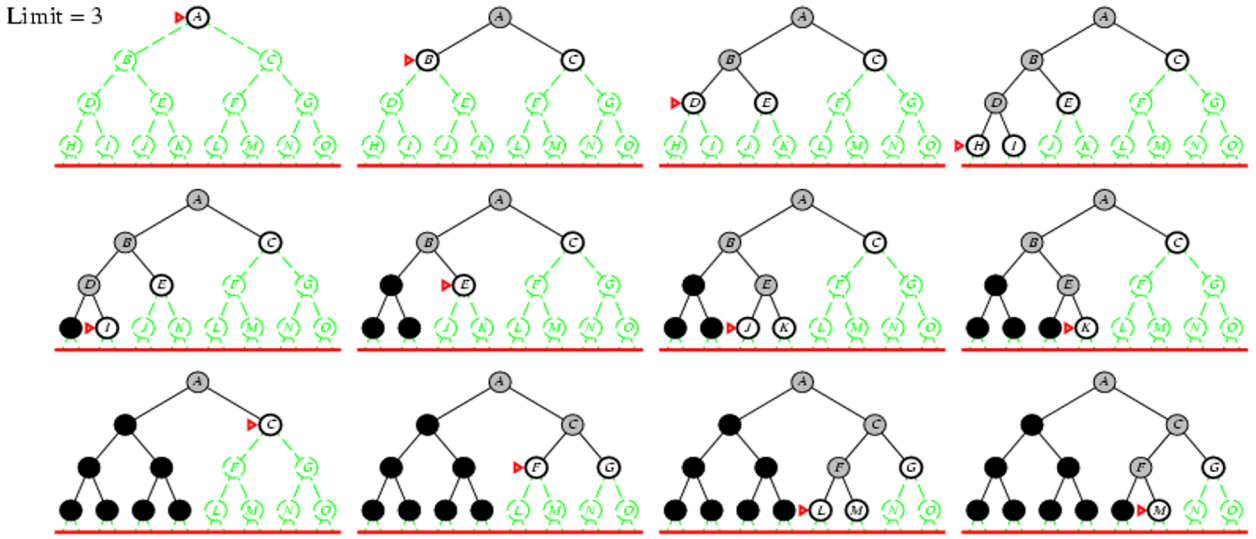
(Tree version with goal test at node generation)

- Completeness: ?
- Time: ?
- Space: ?
- Optimal: ?



Properties of iterative deepening search
(Tree version with goal test at node generation)

- Completeness: Yes
- Time: $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)$
- Space: O(bd)
- Optimal: under the same conditions as BFS

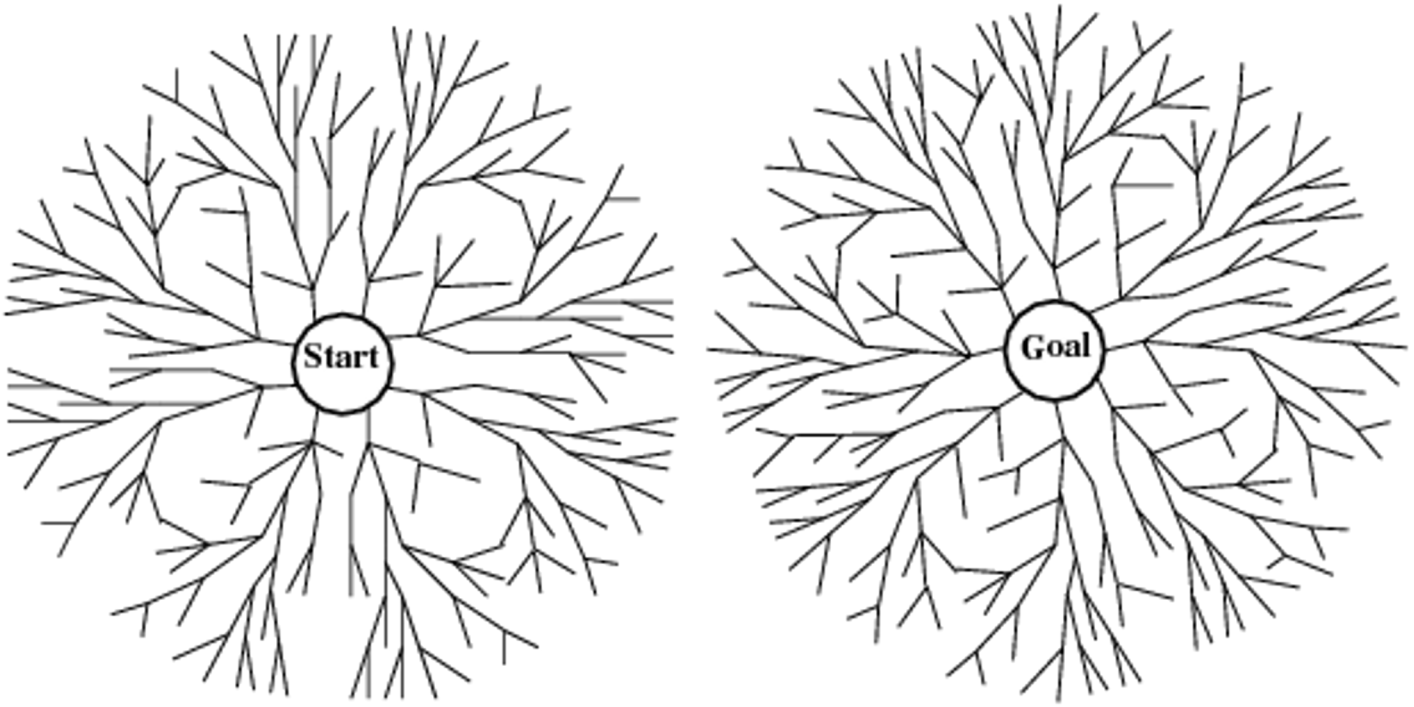


Properties of iterative deepening search
(Tree version with goal test at node generation)

- Completeness: Yes
- Time: $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)$
- Space: O(bd)
- Optimal: under the same conditions as BFS

- Faster than breadth-first search!
- Remember that most nodes in a search tree are at the last level (assuming the same branching factor)

Bi-directional search



- Motivation: $b^{d/2}+b^{d/2} << b^{d}$
- How do we see this in the figure?

The area of the two circles much smaller than one circle with twice the radius

Bi-directional search

- Search in parallel
  - start → goal
  - goal → start
- Look for them to meet
- Why may this be difficult?
  - Need predecessor function and not just successor
  - Need goal states not just goal test

Generic tree search algorithm

- Add root node (start state) to container
- Repeat
  - If container empty → FAILED
  - Choose a node to take out from the container
  - If node corresponds to goal → FOUND IT!
  - Add successors / children to the container

- Search method depends on how we add/take out node from the container

Generic graph search algorithm

- Add root node (start state) to container
- Repeat
  - If container empty → FAILED
  - Choose a node to take out from the container
  - Store the state to avoid it in the future
  - If node corresponds to goal → FOUND IT!
  - Add successors / children to the container if not already visited
- Search method depends on how we add/take out node from the container

Avoid visited states

- Saw that it can speed up search.
- Have to check for repeated states many times
  - Should be fast!

- Q: How to do this?

Avoid visited states

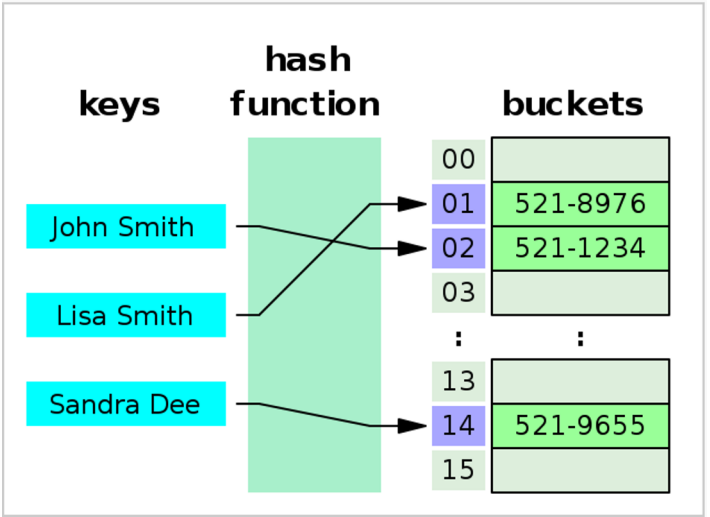
- Saw that it can speed up search.
- Have to check for repeated states many times
  - Should be fast!

- Q: How to do this?
  - Typically with a hash map / hash table

Hash table

- Want to find a way to know which of all previously visited states we should compare the new state to so we do not have to compare to all
- Use hash function to map the state ("key") into an index
- Compare only to previous states with the same index

→ those in the same "bucket"



Stolen from wikipedia

Hash table

- Lots of hash functions have been suggested
- Often implemented as

index = hash_function(key) % N

where N is the number of buckets or bins and % is modulo, ie maps to index [0, N-1]

- Zobrist hashing common in games

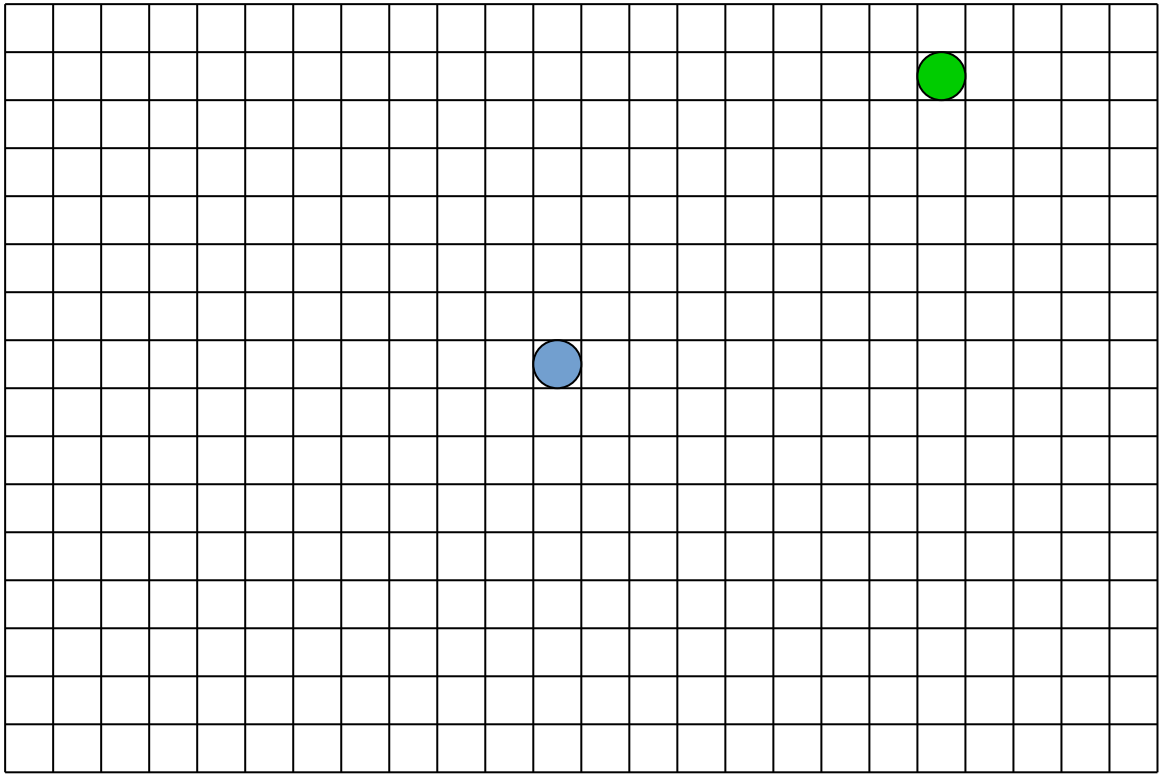
Informed search and exploration

- Un-informed search (e.g., BFS and DFS) often very inefficient
- Will look at two ways to improvements
  - Integrate task knowledge into the search
  - Use information for local search

- We want to make the expand function more clever, i.e. which node to look at next
- Ideally: Expand nodes heading directly for the goal and no others

Grid example

- Imagine a large grid
- Want to find a path to a cell 14 cells away

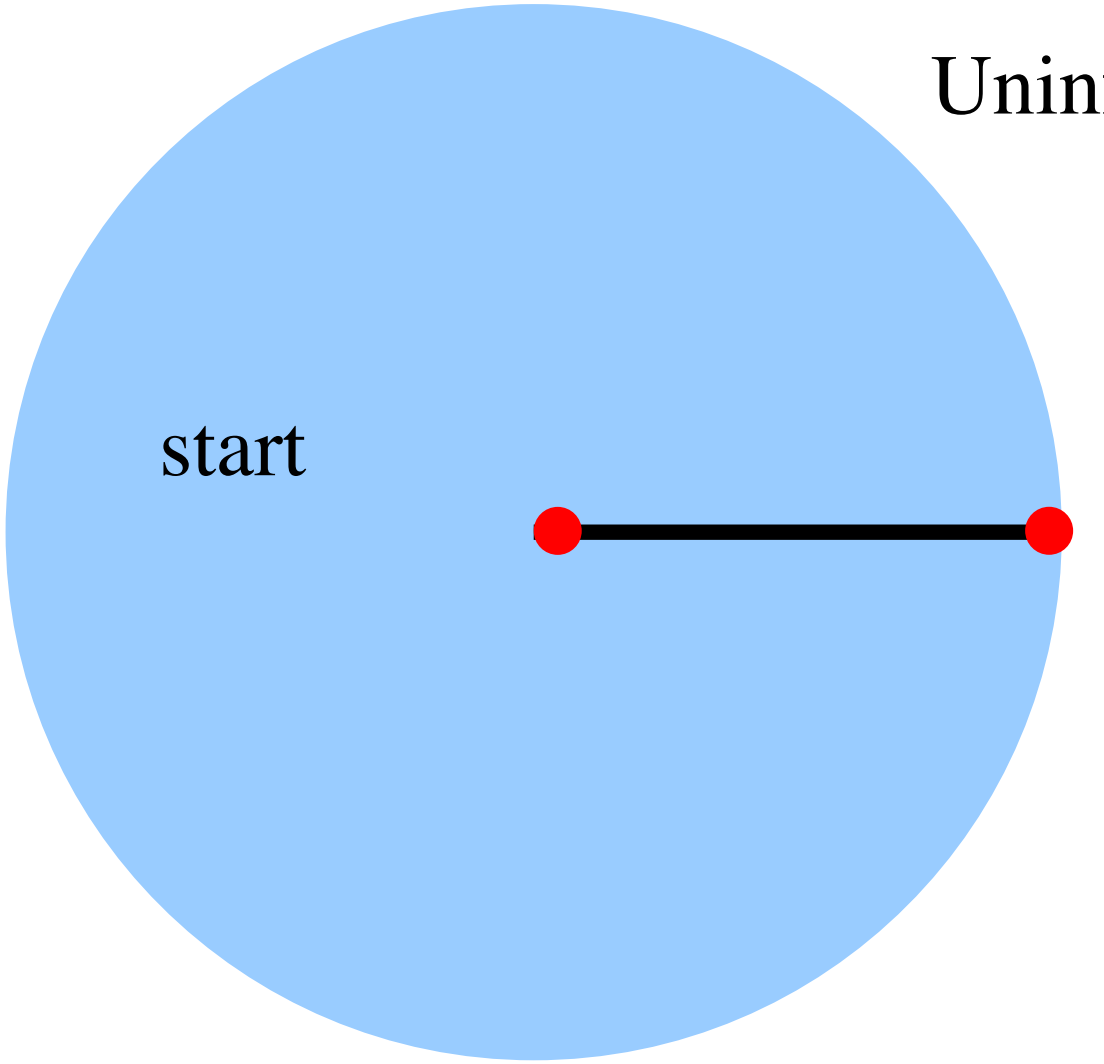


Grid example cont'd

- We want something that makes us expand nodes in the direction of the goal!
- Ideally we could get down to 14 nodes expanded
  - BFS naïve with goal test at node expansion: ~1.000.000.000 ($O(b^{d+1})$, b=4)
  - BFS avoid last state: ~10.000.000
  - BFS avoid past states: ~500 (area of square)
  - "Optimal" informed: 14

- A lot to gain!

Uninformed vs informed search



Uninformed search region

Ideal informed search region

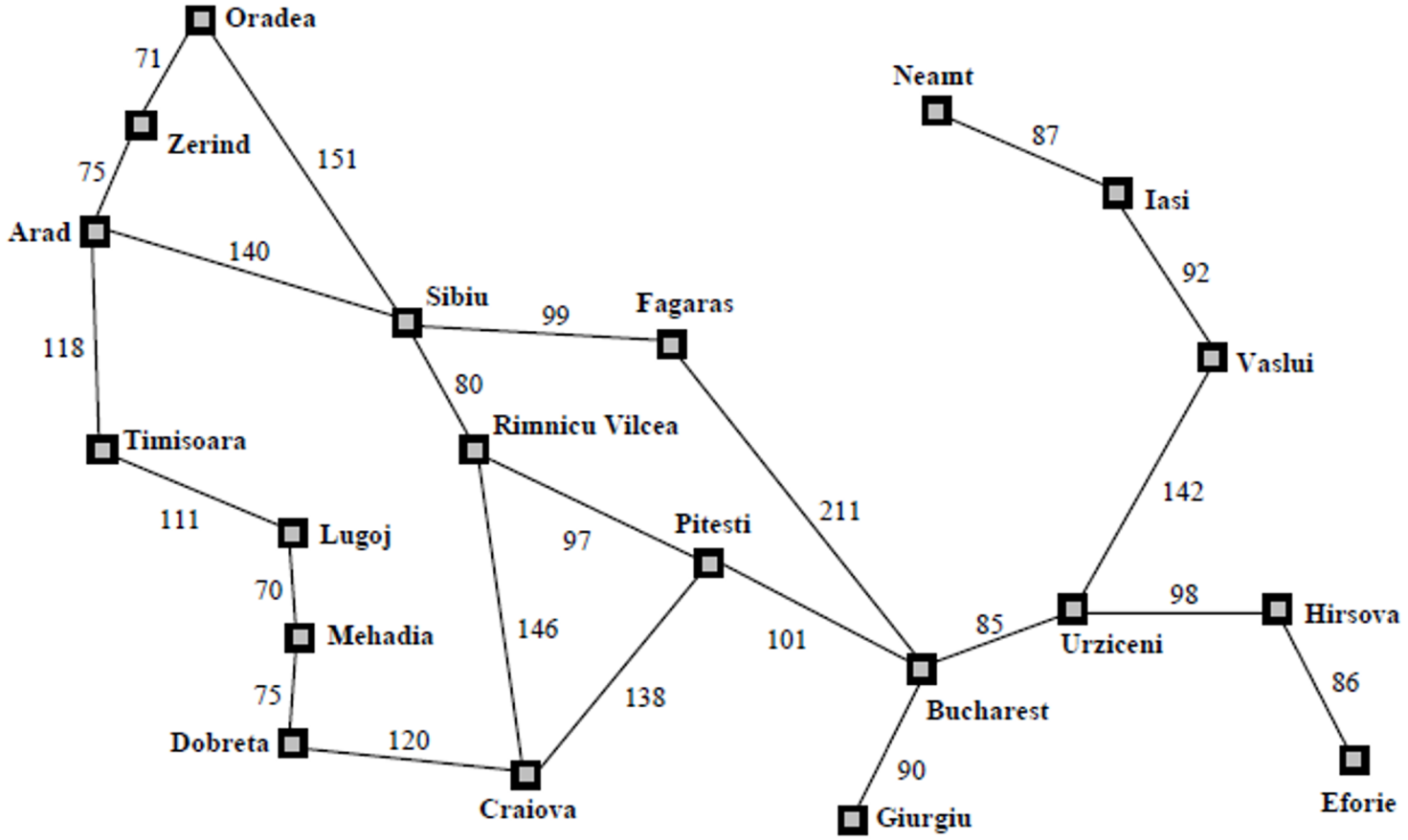
Informed search

- Often we have more information than
  - Initial state
  - Successor function
  - Goal test
  - Path cost

- We know something about the problem
- Use this to direct the search towards the goal

Example

- Where would you go first on the way from Arad to Bucharest? Why?



Informed search strategies

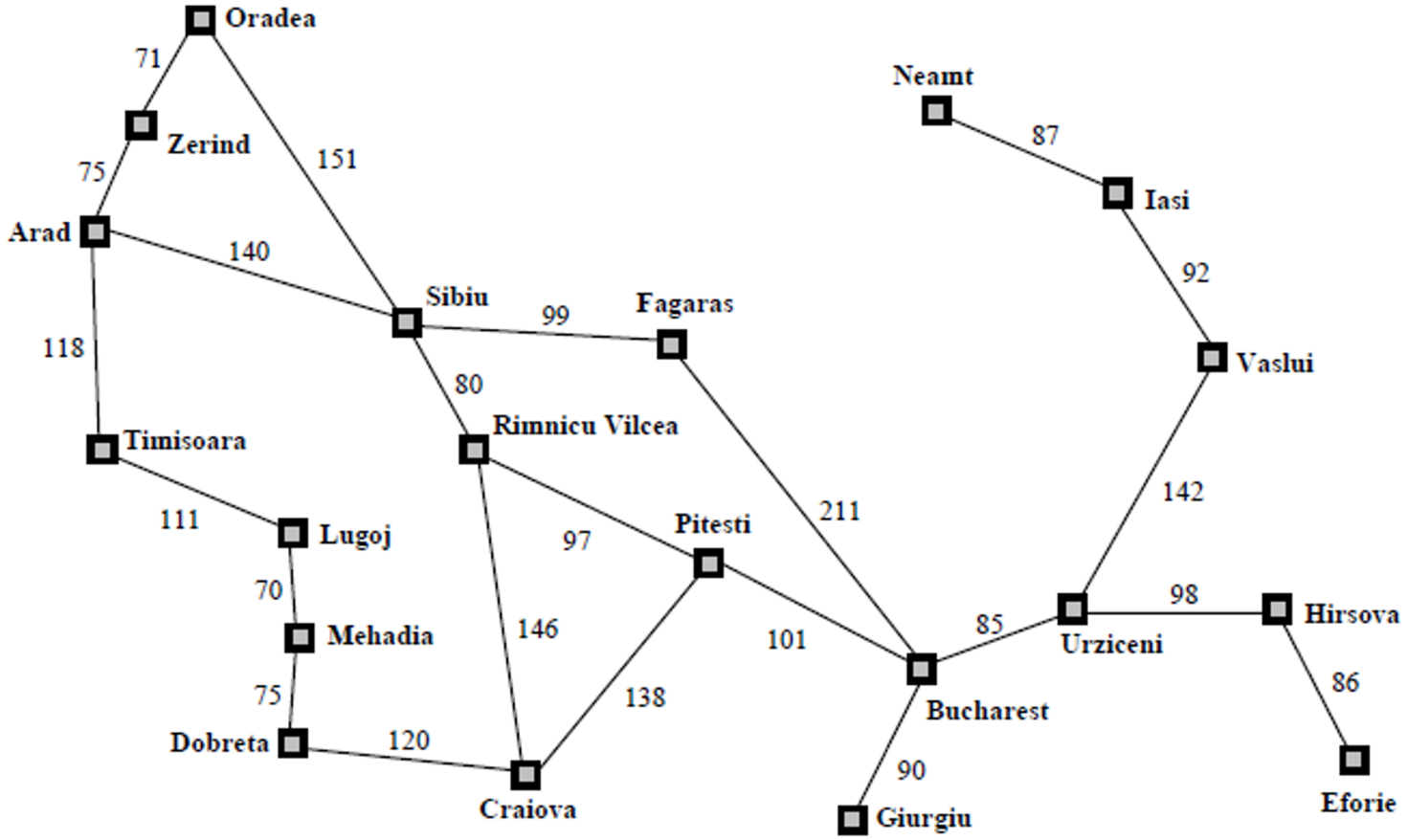
- General form: “Best First Search"
- Expand low-cost nodes before higher cost nodes
- Use evaluation function f(n) to estimate cost

A first go at the evaluation function

- Design of a heuristic function h(n)
  - Estimated cost of the cheapest path from node n to goal

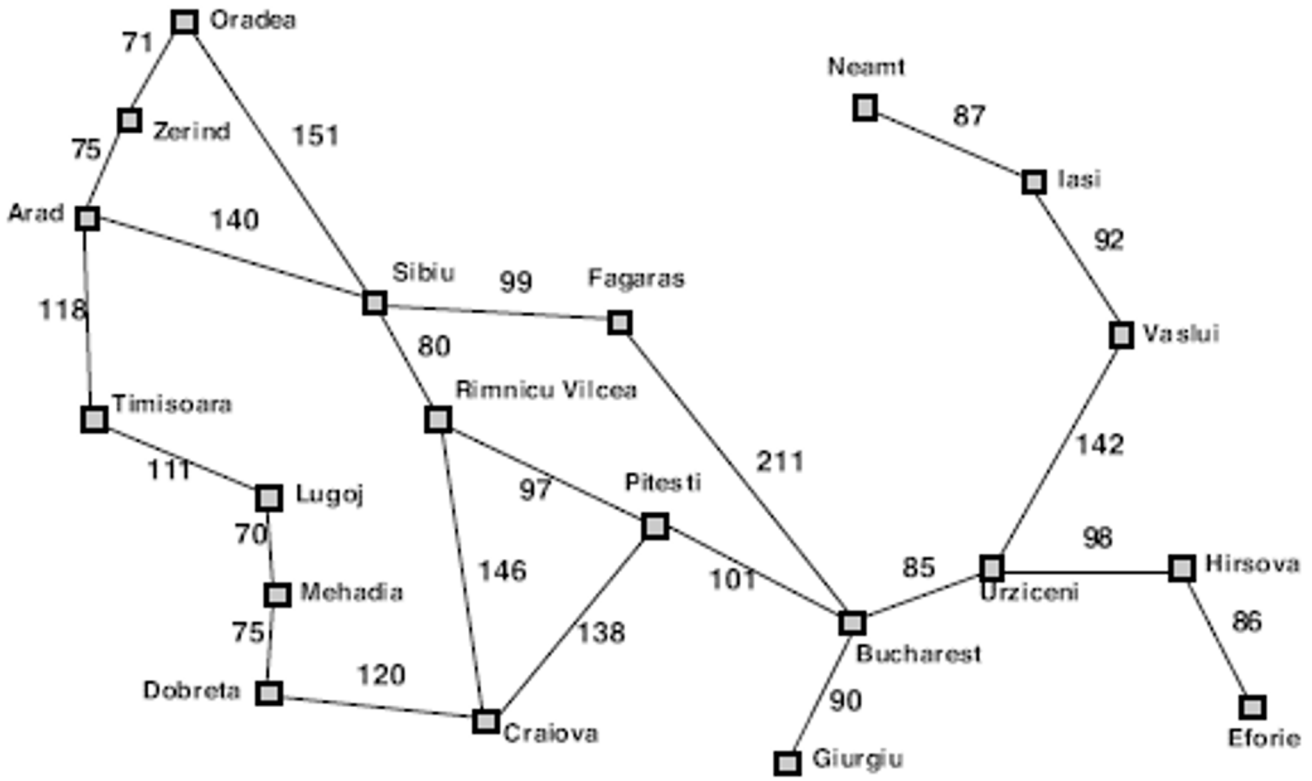
Example

- What would you pick for h(n) in this case?



Example

- Straight line distance from current city to Bucharest as h(n)



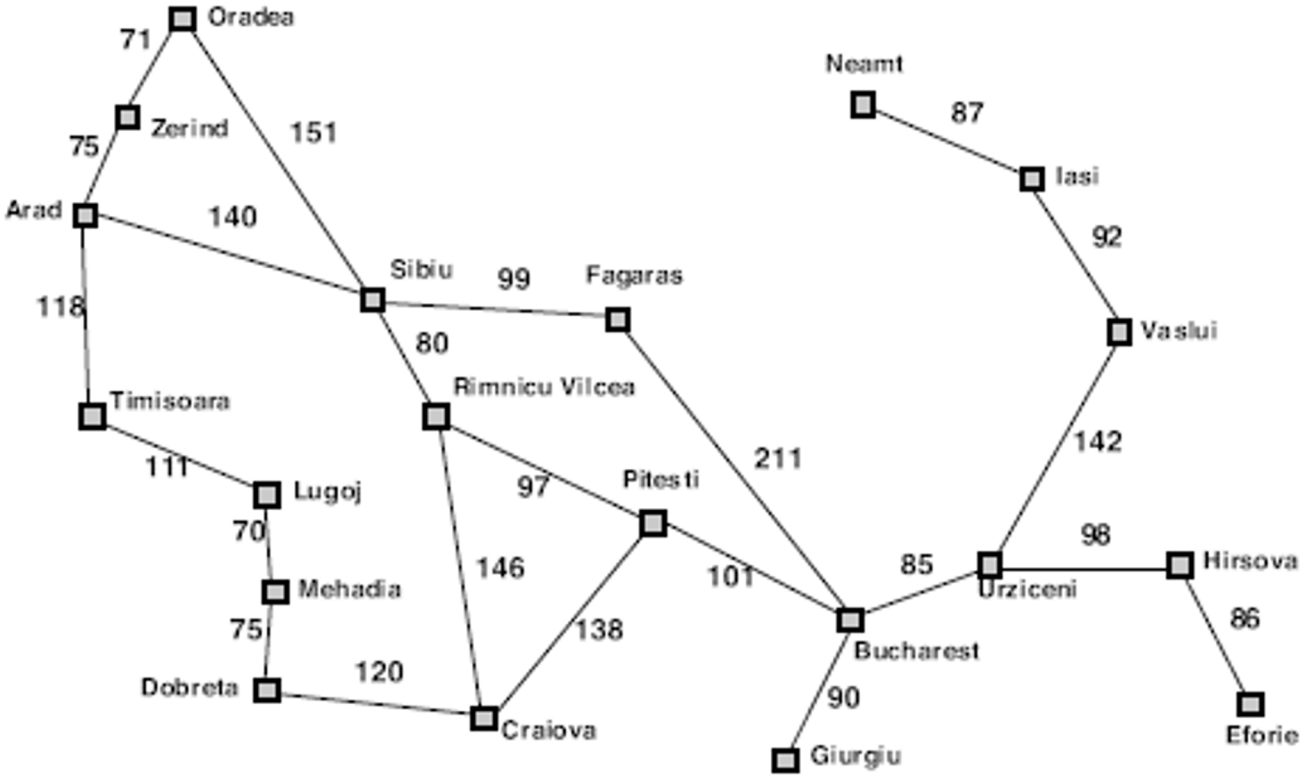
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

- Let h(n) be a heuristic for the search
  - Ex: $h_{SLD}(n)$ - straight line distance from n to Bucharest
- Greedy search uses f(n) = h(n) for expansion of nodes that appear to be closest to the goal.

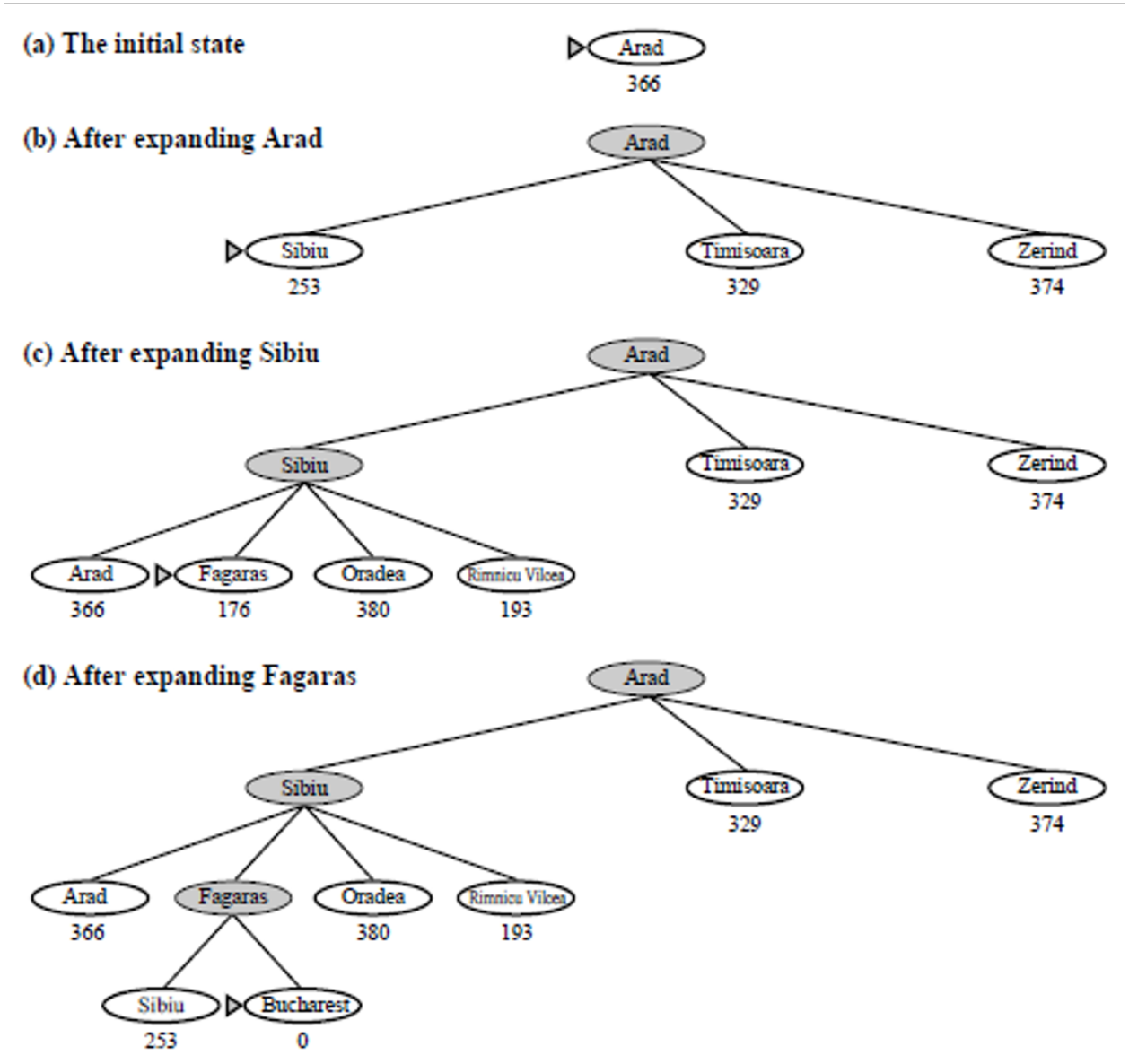
Q

- What path would greedy best-first find from Arad to Bucharest? Is it optimal?



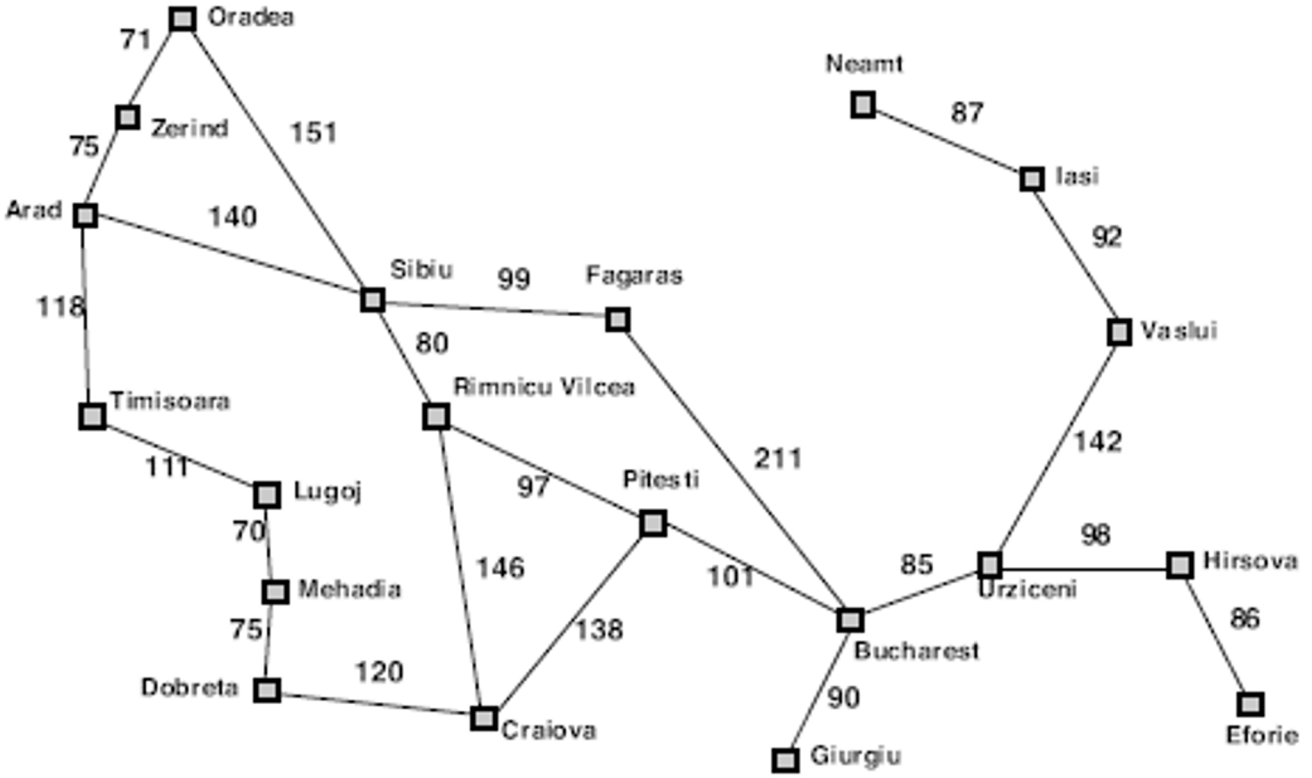
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Node expansion for greedy best-first search



Q

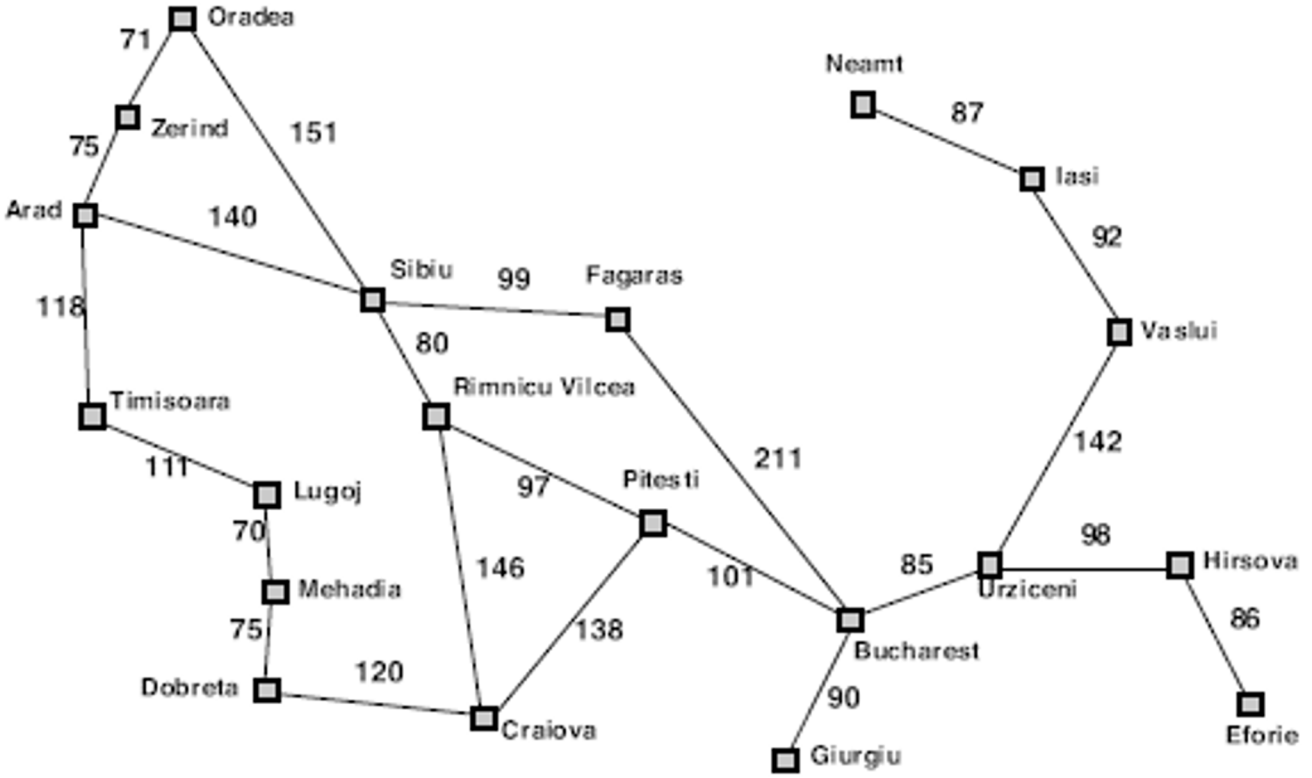
- Which node would the graph greedy best-first search (i.e. the one with repeated state checking) check first from Iasi to Fagaras?



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

Q

- What happens for the tree greedy best-first search (i.e. the one without repeated state checking)?



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

Properties of greedy best-first search

- Completeness: Could get stuck in loops repeated-state checking needed to ensure completeness
- Time: O(bm) - a good heuristic can improve this significantly in practice
- Space: O(bm) - keep all nodes in memory
- Optimal: no

How can we make it better?

How can we make it better?

- Greedy best-first search ignores the past and only looks at what seems to be the best from where we are now
- Would be good to take cost so far into account

A* search

- Minimizing the total estimated solution cost
- Widely used best-first method
- Evaluation function f (n) = g(n) + h(n)

–g(n) the cost from start to node n, i.e. cost this far

–h(n) estimate of cost to get to goal

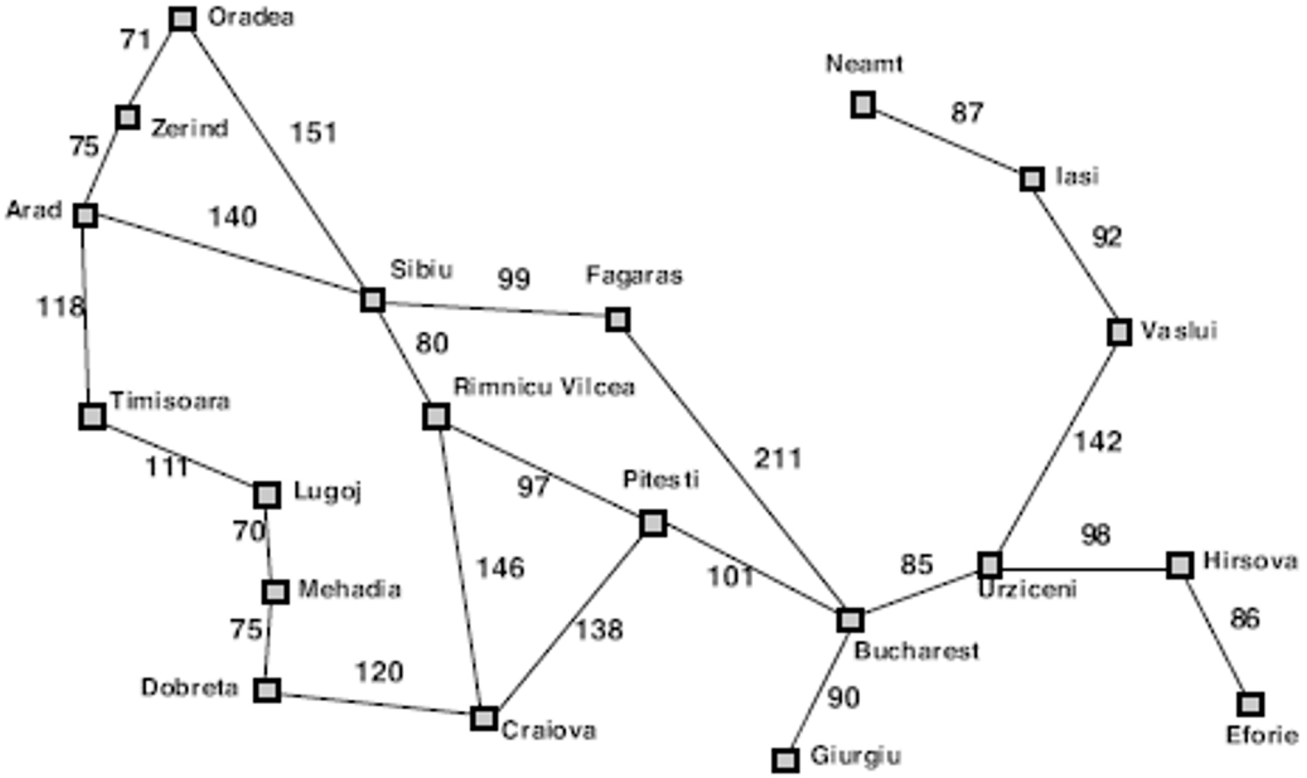
–f (n) estimate of the overall cost from start to goal via node n

A* and optimality

- Can show A* (tree search version) is optimal if h(n) is an admissible heuristic
  - h(n) admissible if it does not overestimate the cost

Q

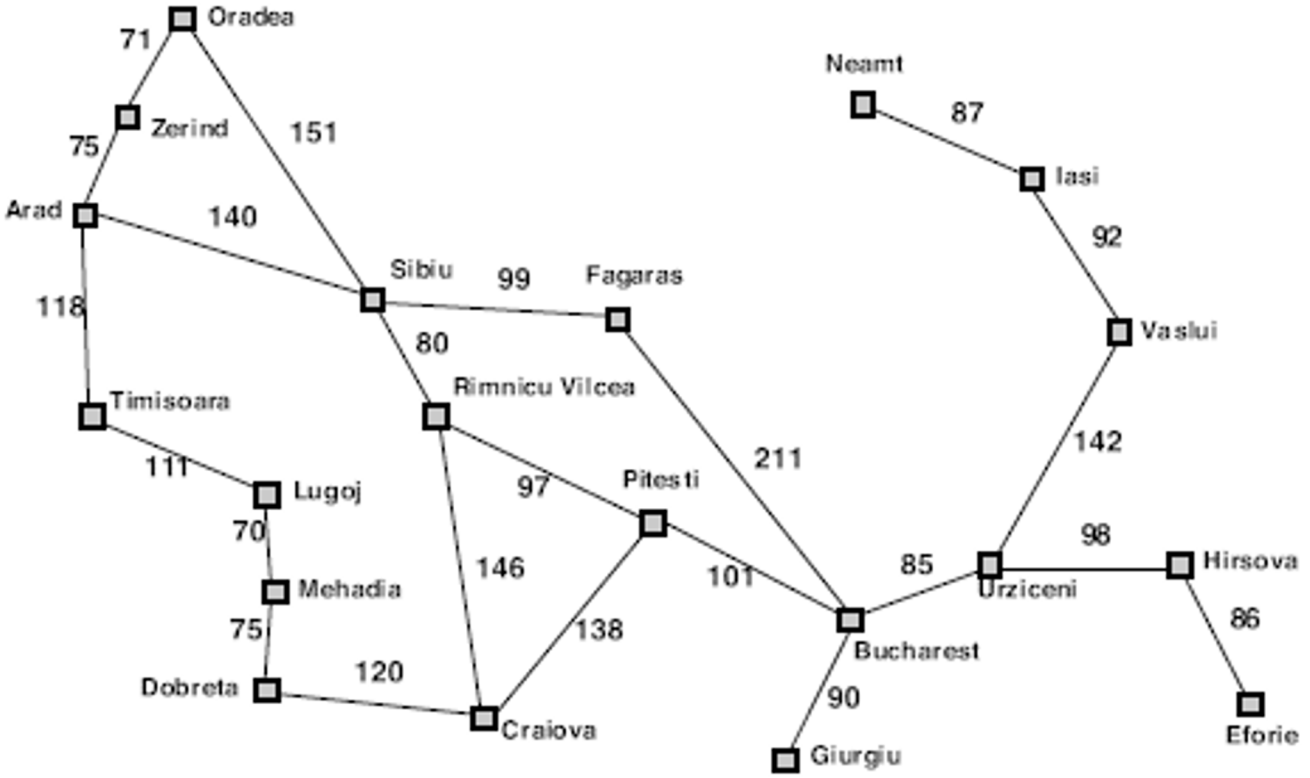
- What path would A* find from Arad to Bucharest using the straight line heuristic?



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

g(n) and h(n)

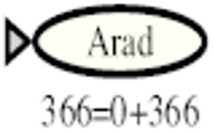
- g(n) is distance traveled so far
- h(n) is straight line distance from current city to Bucharest



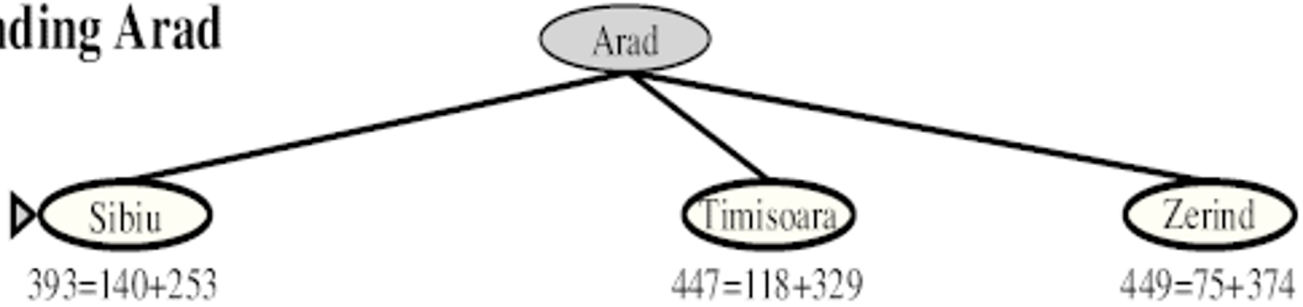
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Node expansion in A*

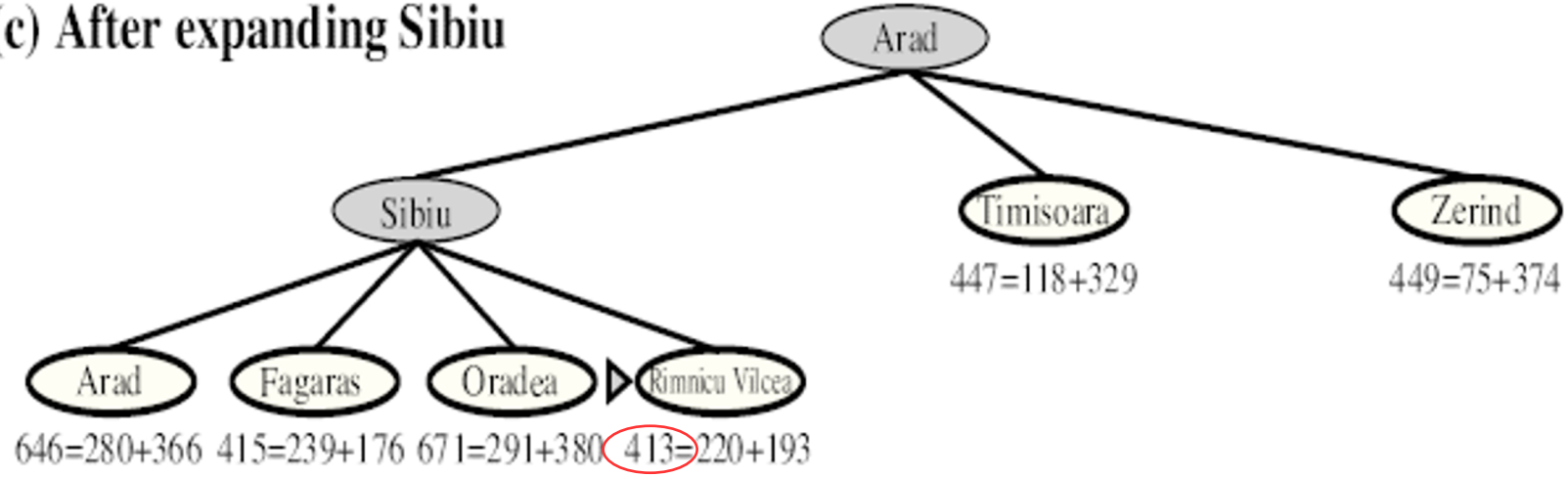
(a) The initial state



(b) After expanding Arad



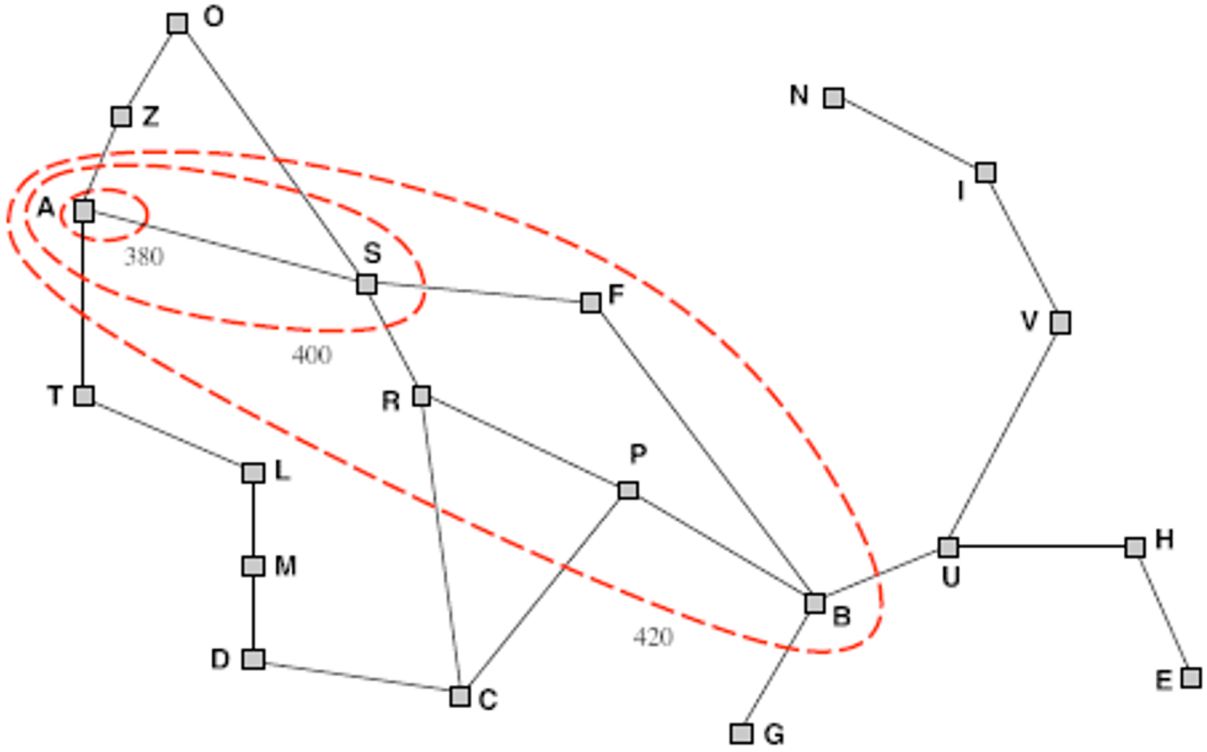
(c) After expanding Sibiu



Optimality of A*

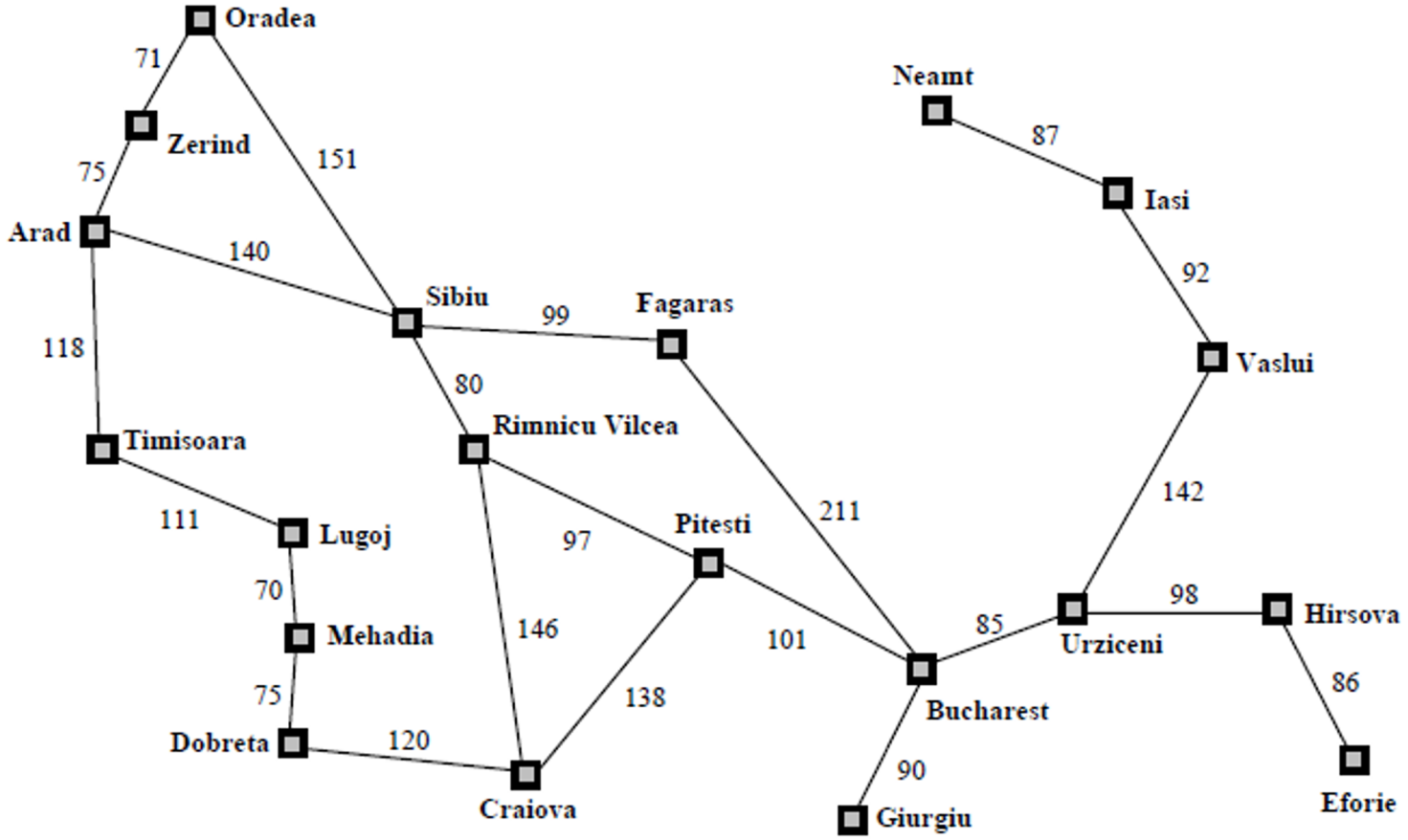
- Expands nodes according to increasing f value
- Gradually adds f-contours of nodes

a bit like breadth-first with layers



Q

- What is A* search for the routing problem with h(n) = 0? Is it still optimal?



Properties of A*

- Complete: Yes
- Time: Exponential in length of solution
- Space: Keeps all nodes in memory
  - memory main problem
- Optimal: Yes (under certain conditions)

Variations of A*

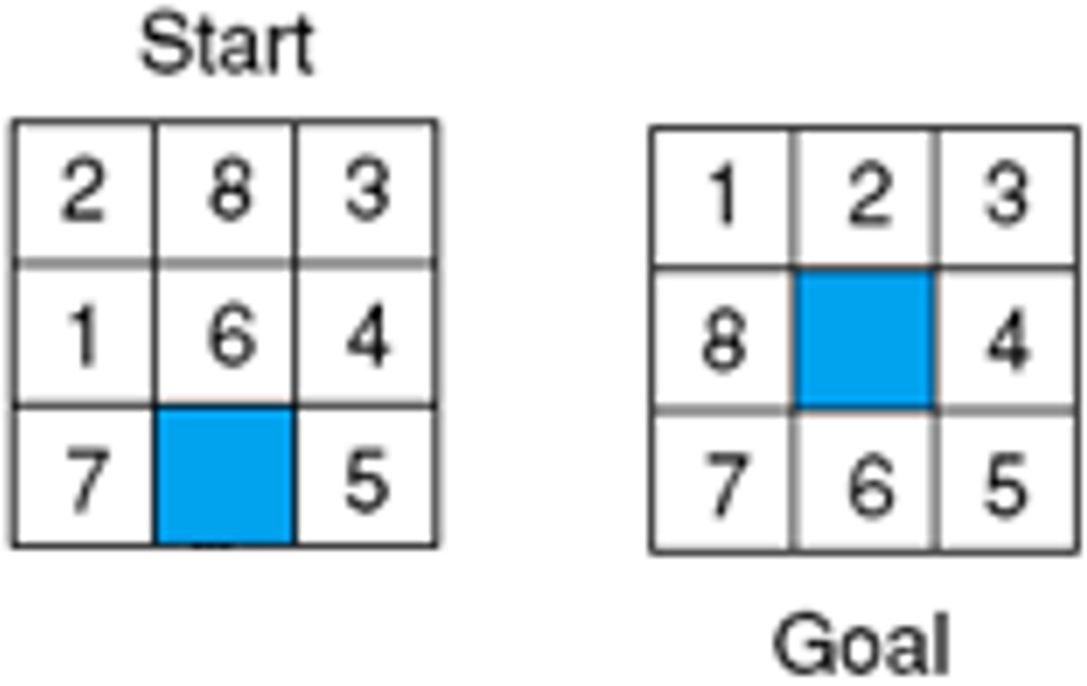
- Memory can pose a problem for A*
- Variations
  - Iterative Deepening A* (IDA)
  - Recursive best first (RBDF)
  - Memory Bounded A* (MA)
  - Simple MA (SMA)

- See the book for the details of the these methods

Design of heuristic function

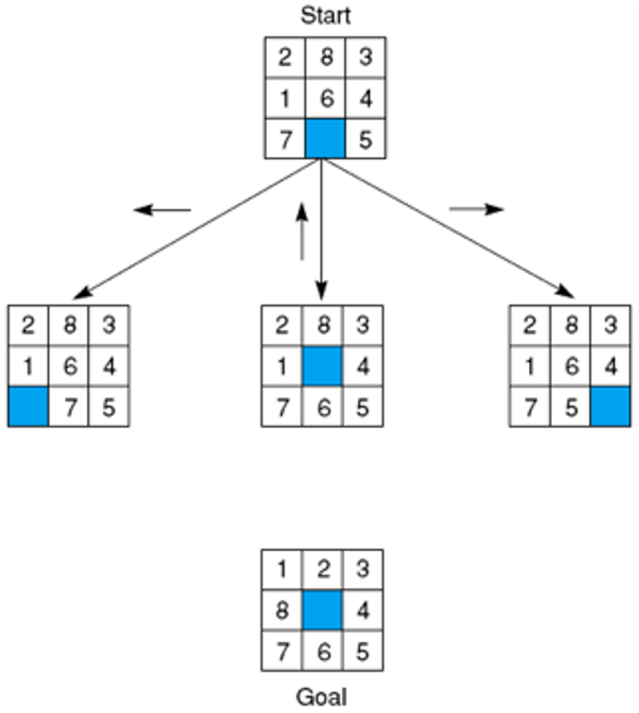
- One of the keys to making your search efficient

8-puzzle



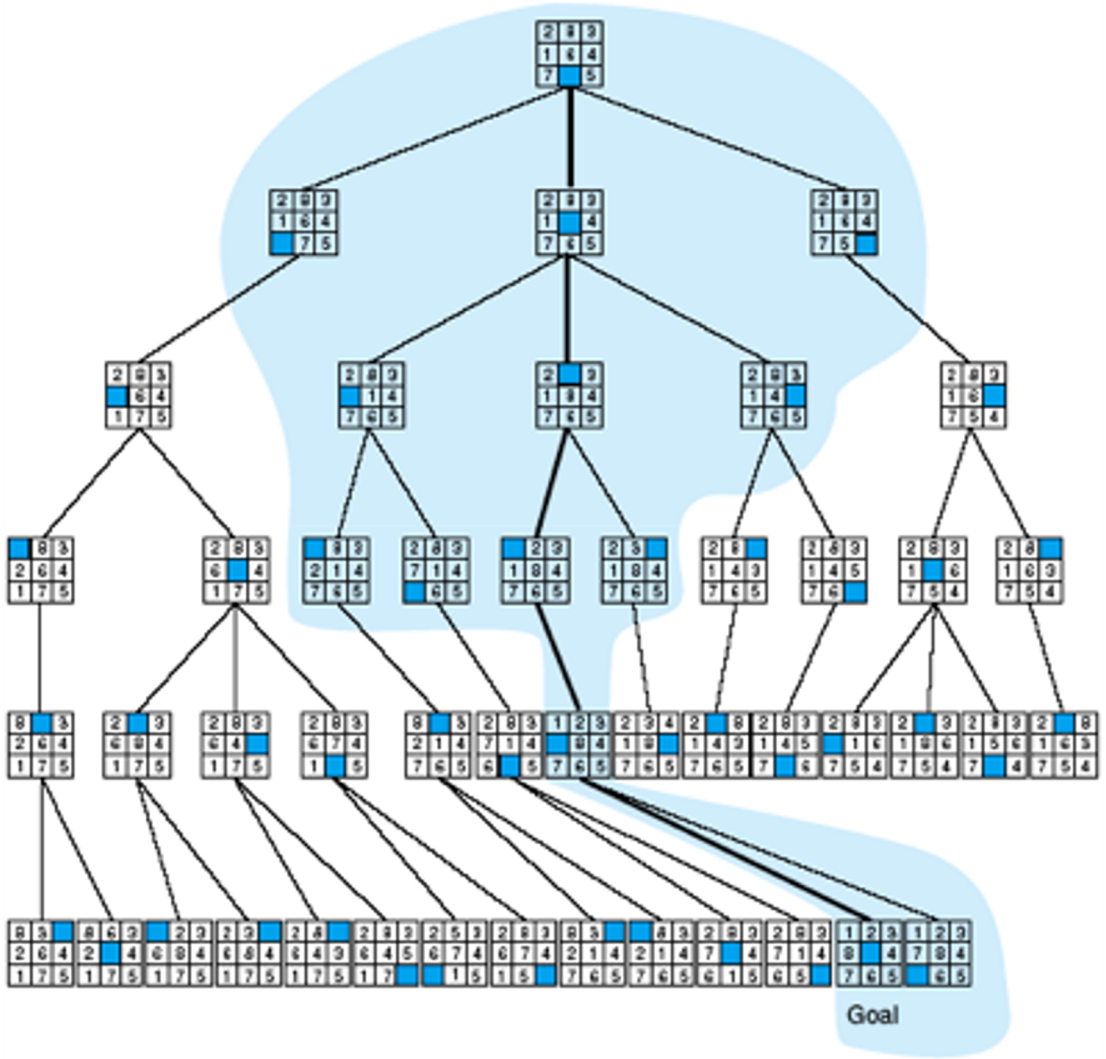
- One of the first problems where heuristics was studied
- Any ideas for a heuristic?

8-puzzle search tree example



- Two commonly used heuristics
  - $h_1(n)$: Number of misplaced tiles
  - $h_2(n)$: Sum of distances tiles are from goal position (Measured using city-block distance)

8-puzzle with A*



Notice how the search is “pulled” towards the goal
Blue area marks explored part of search space

Dominating heuristics

- Which of the heuristics is better?

–The one that needs to expand less nodes before finding an optimal solution

–If $h_2(n) \geq h_1(n)$ for all n, then h_2 dominates h_1

–Dominating heuristics is always better

Q

- Which of the heuristics is better?
- $h_1(n)$: Number of misplaced tiles
- $h_2(n)$: Sum of distances tiles are from goal position (Measured using city-block distance)

Inventing heuristics

- Strategies to invent heuristics include
  - Look at relaxed problems
- What are the relaxed problems for h_1 and h_2?

  - Analysis of sub-problems
- What is a sub-problem of the 8-puzzle?
  - Use of pattern databases

Local search methods

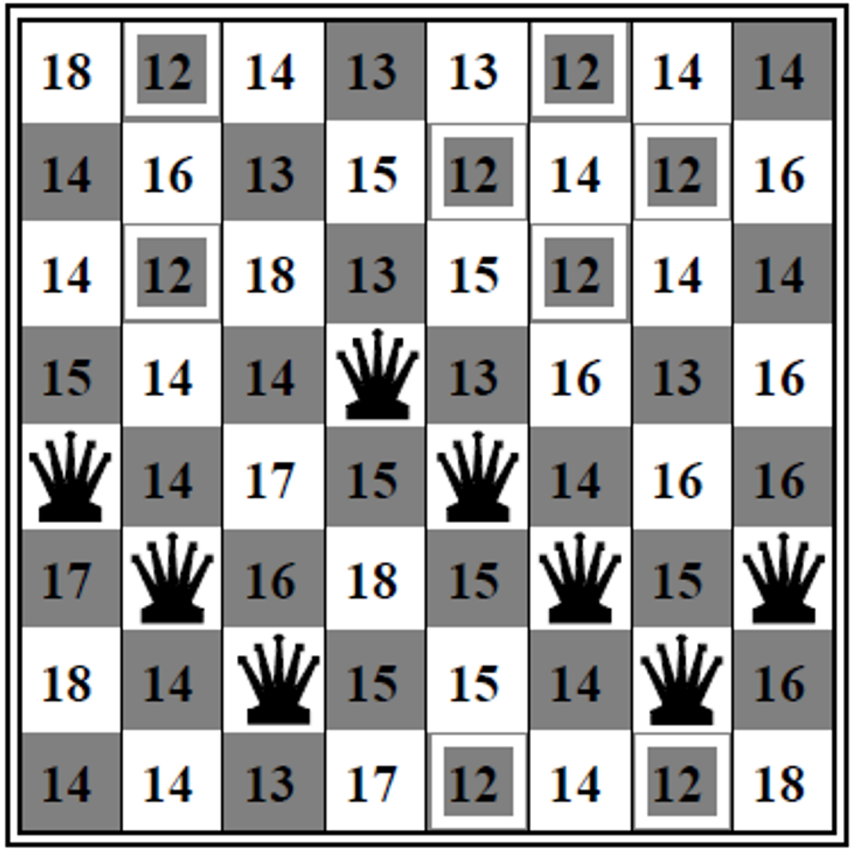
- In several cases the path is irrelevant
- The goal configuration itself is the solution, not the path to it
  - –train schedule, factory-floor layout, N-queens
- State space is the set of possible configuration
- Uses only a single current state, paths are not retained (mostly)
- Not systematic, but only little memory is needed and often find reasonable solutions

Hill climbing search

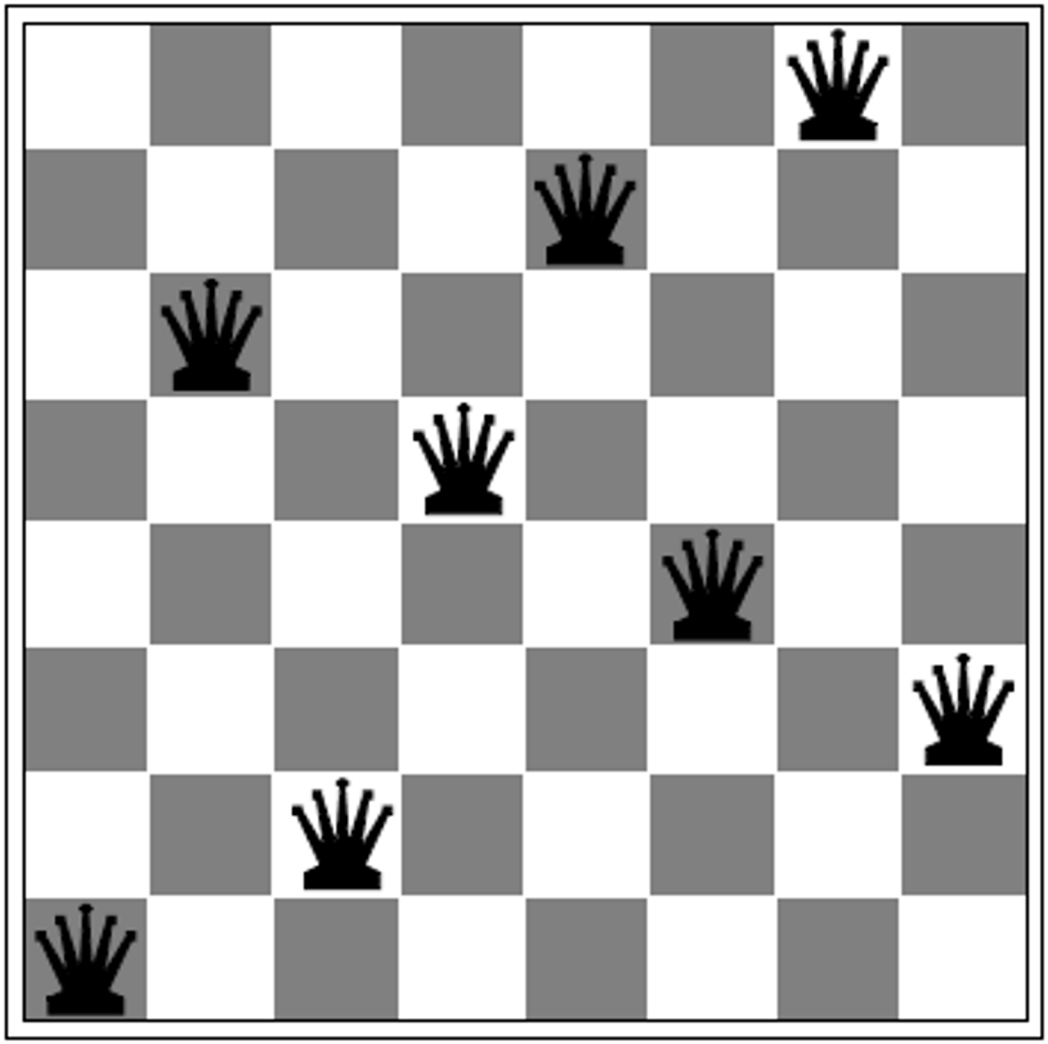
- Find the top of Mount Everest in a thick fog while suffering from amnesia
- Most basic (greedy) local search method
- Look at neighbors and move “uphill" in performance metric

Example: 8-queens

- h = Number of pairs of queens under attach
- h = 17 below



8-queens with local minima h=1



Hill climbing variations

- Stochastic hill climbing
  - Pick randomly among "improving successors"
- First-choice hill climbing
  - Pick randomly among all successors until finding an improving one
- Random restart hill climbing
  - Repeat with random initial states until goal is found

Simulated annealing

- Observation: Cannot move up in all landscapes in all steps!
- Make a “bad" move occasionally to escape.
- Reduce frequency/size over time

Summary informed search

- Use of heuristics to reduce search
- Best First Search - requires a heuristic
- A* - a method that is optimal and complete

- Local search – finding a goal state vs. a path