Artificial Intelligence

CSC348 Unit 3: Problem Solving and Search

Syedur Rahman

Lecturer, CSE Department

North South University

syedur.rahman@wolfson.oxon.org

Artificial Intelligence: Lecture Notes

The lecture notes from the introductory lecture and this unit will be available shortly from the following URL:

- http://www.geocities.com/syedatnsu/

Acknowledgements

- These lecture notes contain material from the following sources
  - *Intelligent Systems* by S.Clark, 2005
  - *Logical Programming and Artificial Intelligence* by S. Kapetanakis, 2004
  - *Artificial Intelligence: A modern approach* by S. Russell and P. Norvig, International Edition, 2nd edition

© 2006 Syedur Rahman

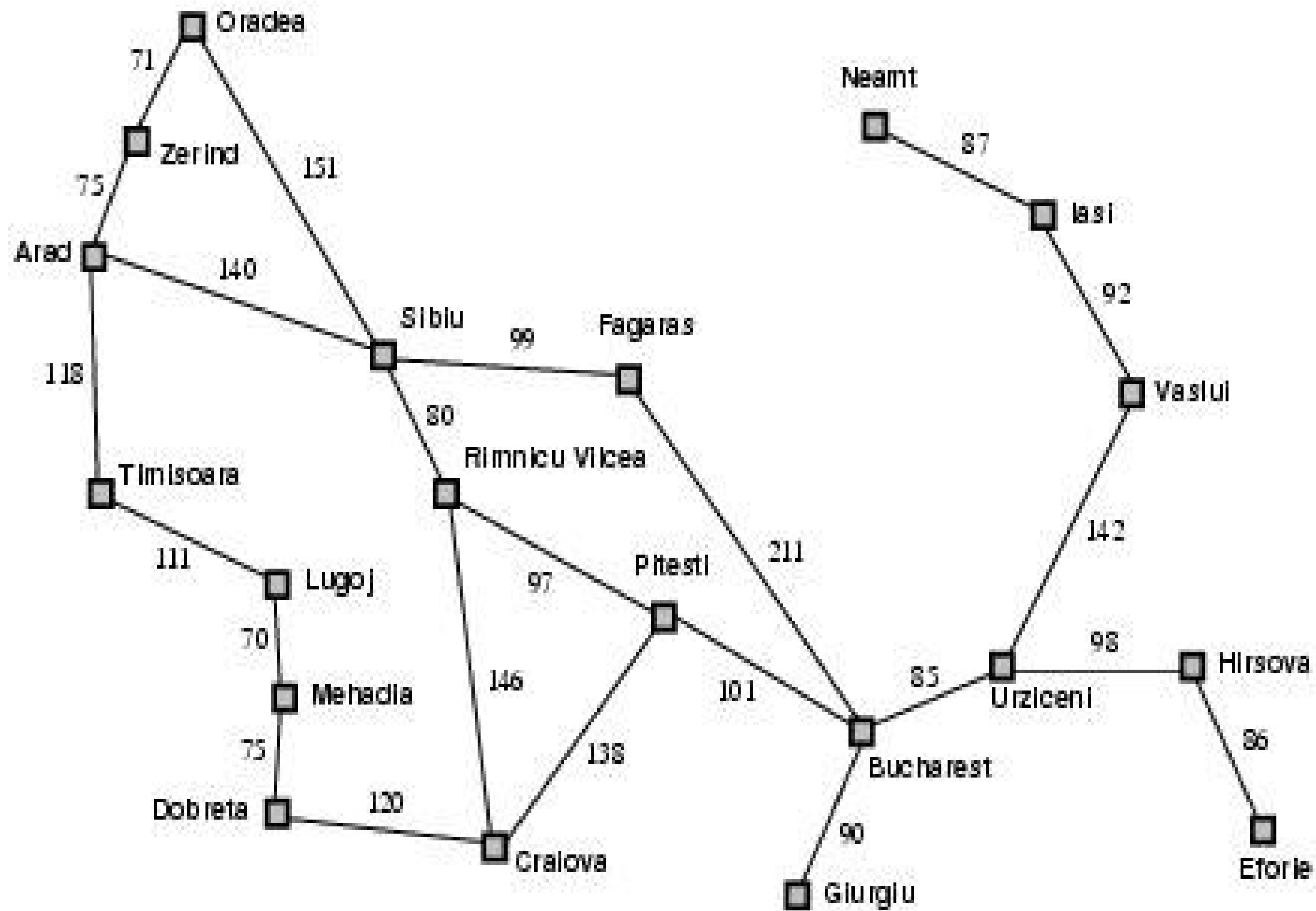
Unit 3: Problem Solving and Search

- Problem Solving Agents
- Problem Formulation and Search Spaces
- Tree Search Algorithm
- Breadth First Search
- Depth First Search
- Depth Limited Search
- Uniform Cost Search
- Iteratively Deepening Search
- Best First Search
- A* Search

Problem-solving agent

- Four general steps in problem solving:
  - **Goal formulation**
    - What are the successful world states
  - **Problem formulation**
    - What actions and states to consider given the goal
  - **Search**
    - Examine different possible sequences of actions that lead to states of known value and then choose the best sequence
  - **Execute**
    - Perform actions on the basis of the solution

Example: Romania



Example: 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, …

Problem type

- Given how we have defined the problem:
  - Environment is **fully observable**
  - Environment is **deterministic**
  - Environment is **sequential**
  - Environment is **static**
  - Environment is **discrete**
  - Environment is **single-agent**

Problem formulation

- A problem is defined by:
  - An initial state, e.g. *In(Arad)*
  - A successor function *S(X)*= set of action-state pairs
    - e.g. *S(In(Arad))={(Go(Sibiu), In(Sibiu)), (Go(Zerind), In(Zerind)),…}*

  initial state + successor function = state space
  - Goal test
    - Explicit, e.g. *x='In(Bucharest)'*
    - Implicit, e.g. *checkmate(x)*
  - Path cost (assume additive)
    - e.g. sum of distances, number of actions executed, …
    - *c(x,a,y)* is the step cost, assumed to be >= 0

A solution is a sequence of actions from initial to goal state;
the optimal solution has the lowest path cost

Example: 8-puzzle

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??

Example: 8-puzzle

7	2	4
5		6
8	3	1

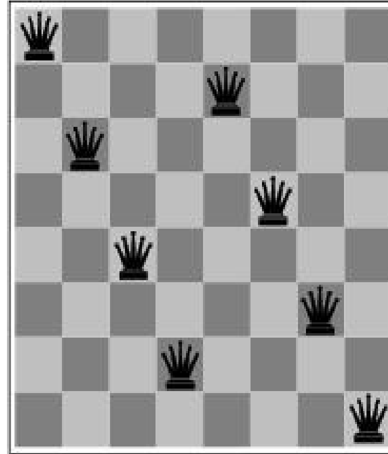
Start State

	1	2
3	4	5
6	7	8

Goal State

- States: location of each tile plus blank
- Initial state: Any state can be initial
- Actions: Move blank {*Left*, *Right*, *Up*, *Down*}
- Goal test: Check whether goal configuration is reached
- Path cost: Number of actions to reach goal

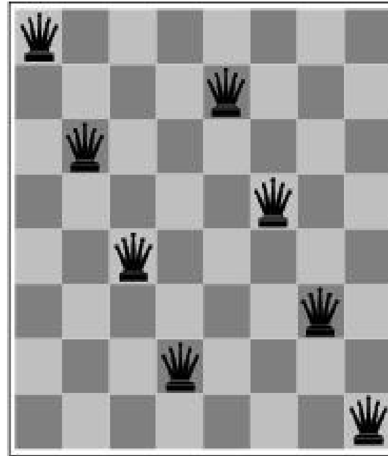
Example: 8-queens problem



Incremental formulation vs. complete-state formulation

- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??

Example: 8-queens problem

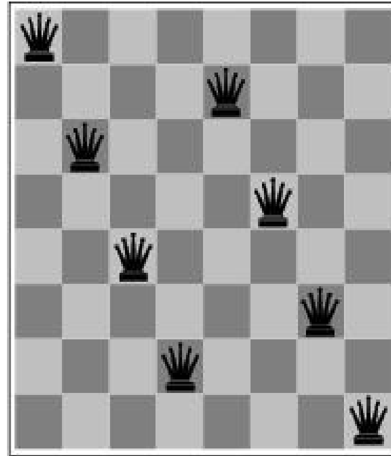


Incremental formulation

- States: Any arrangement of 0 to 8 queens on the board
- Initial state: No queens
- Actions: Add queen to any empty square
- Goal test: 8 queens on board and none attacked
- Path cost: N/A

  3×10^{14} possible sequences to investigate

Example: 8-queens problem



Incremental formulation (alternative)

- States?? n ($0 \leq n \leq 8$) queens on the board, one per column in the n leftmost columns with no queen attacking any other
- Actions?? Add queen in leftmost empty column such that is not attacking any other queen
- 2057 possible sequences to investigate; solutions are easy to find
  - But with 100 queens the problem is much harder

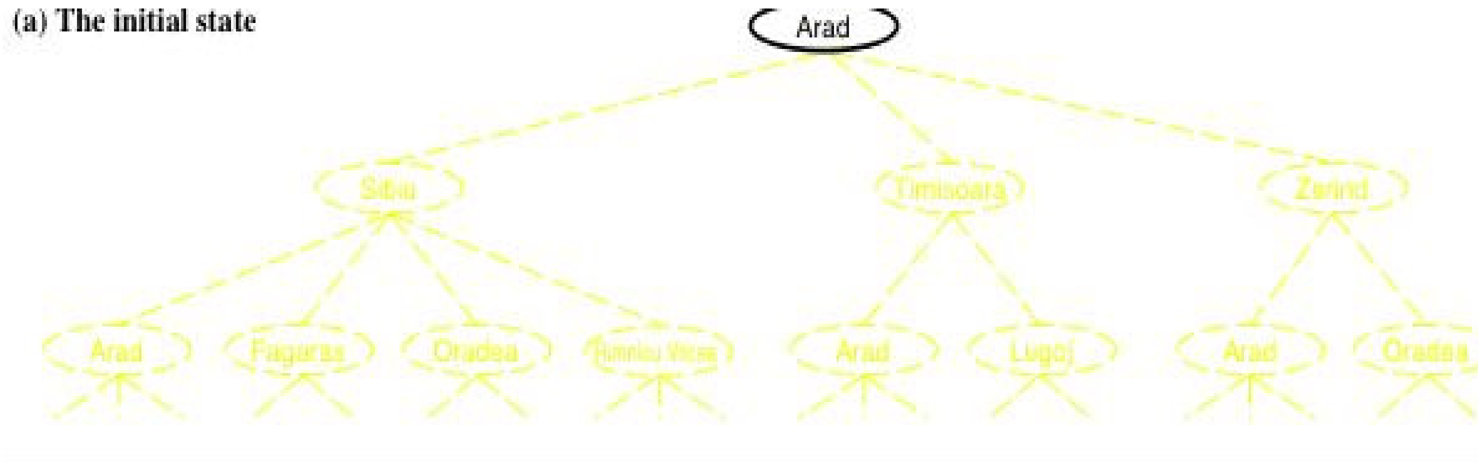
Real World Examples

- Route-finding problems
- Touring problems
- Travelling Salesman problem
- VLSI layout problem
- Robot navigation
- Automatic assembly sequencing
- Internet searching

Basic search algorithms

- How do we find the solutions of previous problems?
  - Search the state space
    - State space can be represented by a **search tree**
    - Root of the tree is the initial state
    - Children generated through successor function
  - In general we may have a search graph rather than tree (same state can be reached through multiple paths)

Simple tree search example



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

Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*

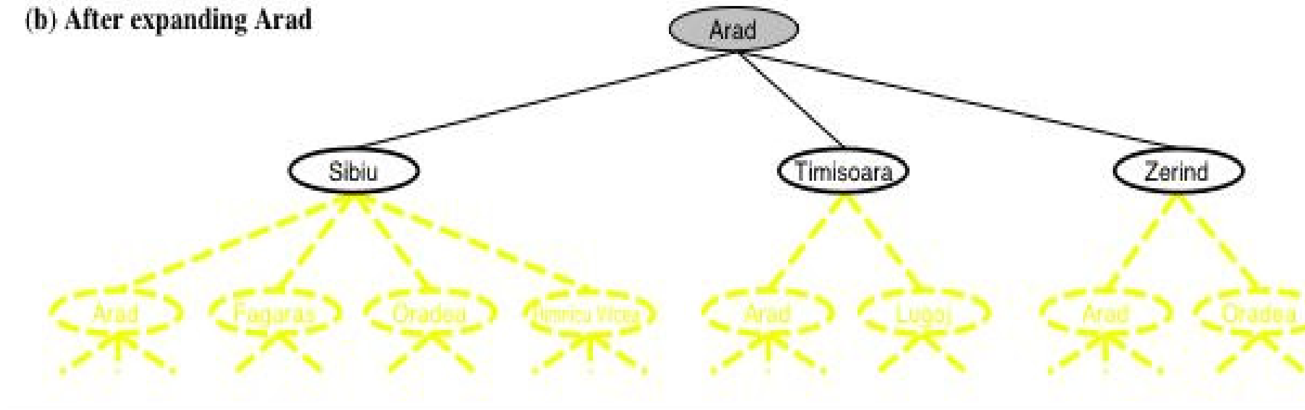
choose leaf node for expansion according to *strategy*

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

else expand the node and add resulting nodes to the search tree

enddo

Simple tree search example



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

Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*

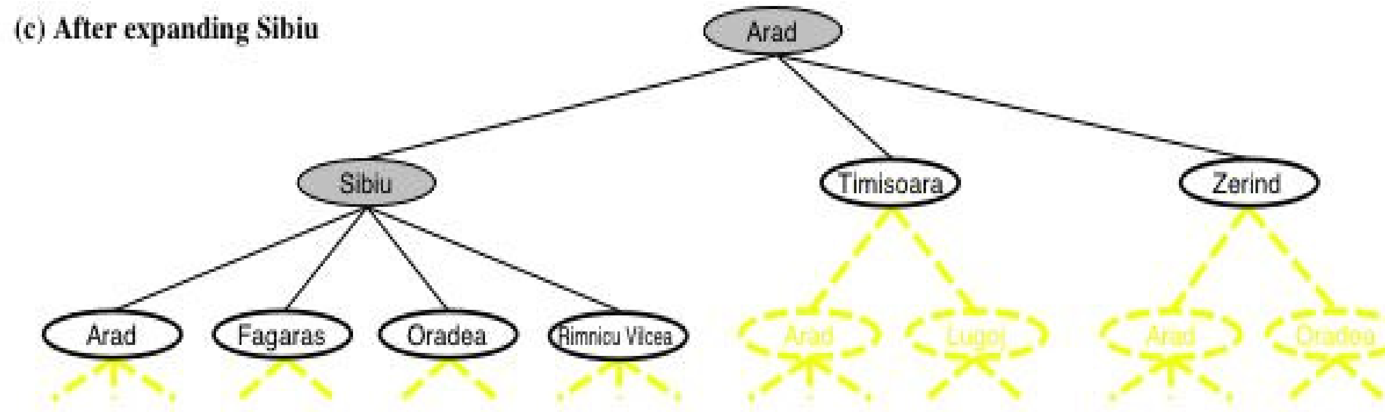
choose leaf node for expansion according to *strategy*

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

else expand the node and add resulting nodes to the search tree

enddo

Simple tree search example



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

Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*

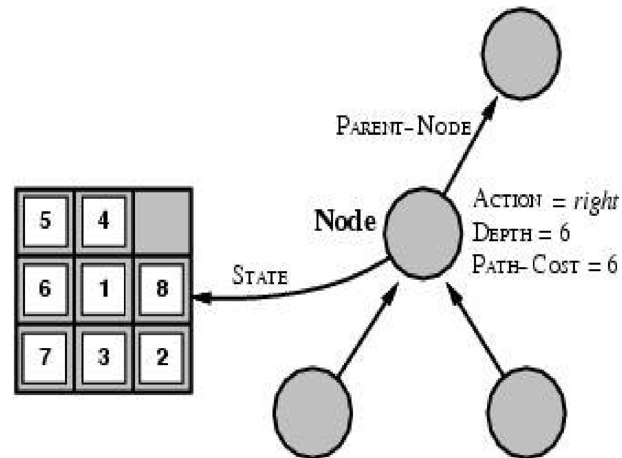
choose leaf node for expansion according to *strategy* ← **Determines search process**

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

else expand the node and add resulting nodes to the search tree

enddo

State space vs. search tree



- A *state* corresponds to a configuration of the world
- A *node* is a data structure in a search tree
  - *e.g. node= <state, parent-node, action, path-cost, depth>*

Search strategies

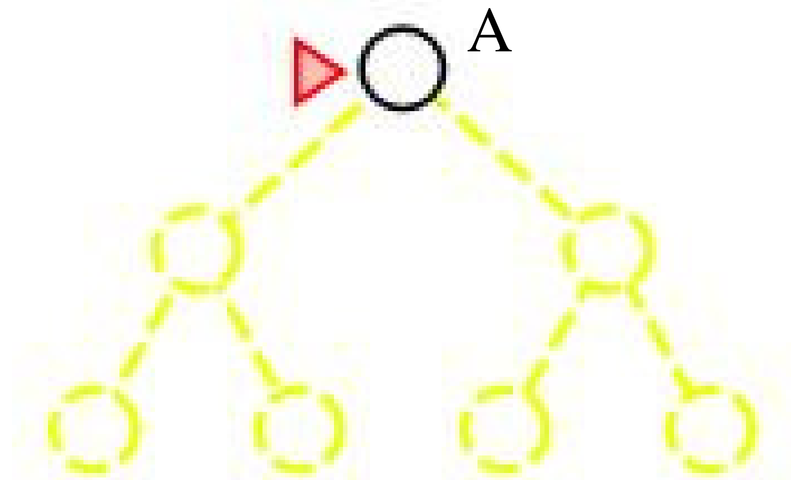
- A search strategy is defined by picking the order of node expansion
- Problem-solving performance is measured in four ways:
  - **Completeness**: Is a solution found if one exists?
  - **Optimality**: Does the strategy find the optimal solution?
  - **Time Complexity**: How long does it take to find a solution?
  - **Space Complexity**: How much memory is needed to perform the search?
- Time and space complexity are measured in terms of problem difficulty defined by:
  - b - branching factor of the search tree
  - d - depth of the shallowest goal node
  - m - maximum length of any path in the state space

Uninformed search strategies

- **Uninformed search** (or **blind search**)
  - Strategies have no additional information about states beyond that provided in the problem definition
  - **Informed** (or **heuristic**) search strategies know whether one state is more promising than another
- Uninformed strategies (defined by order in which nodes are expanded):
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search

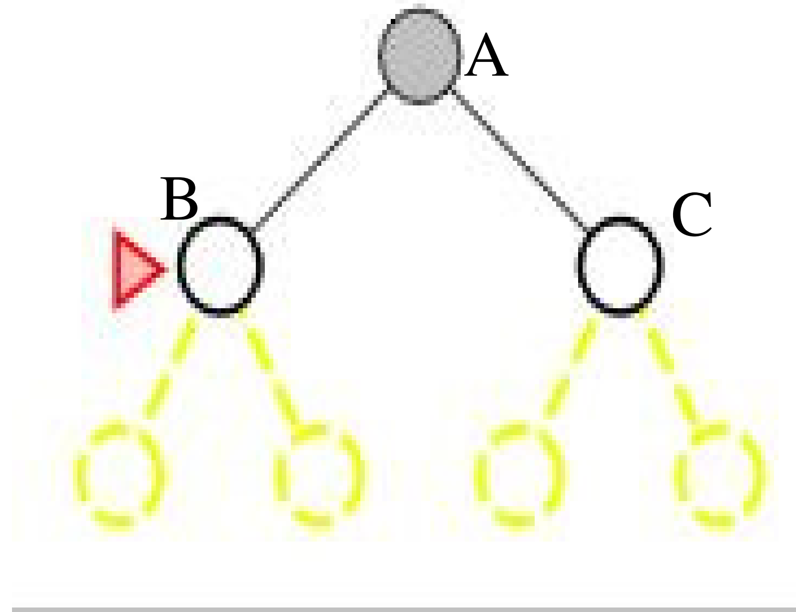
Breadth-first search

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



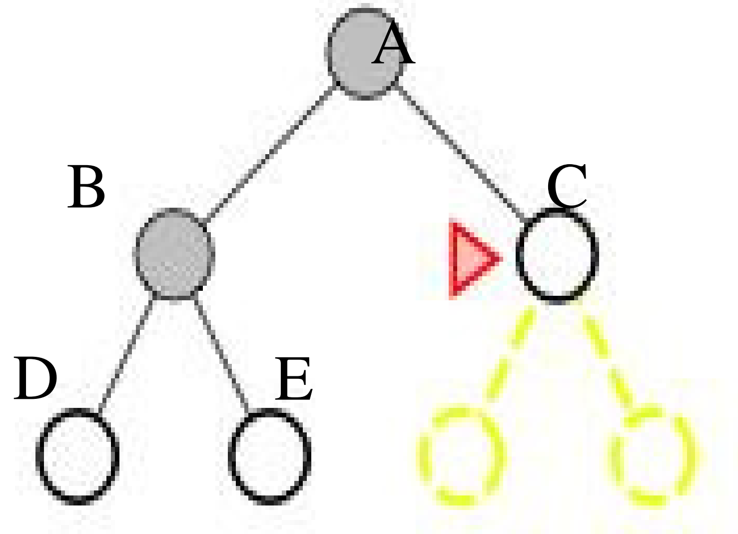
Breadth-first search

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



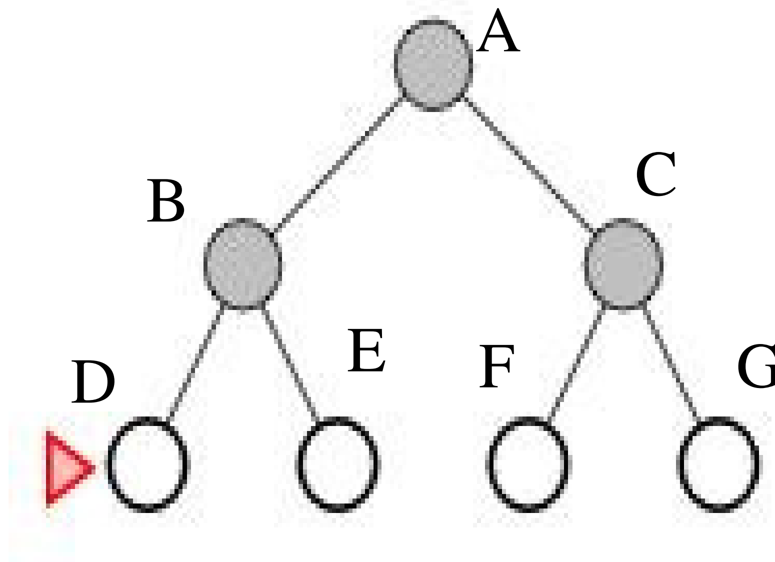
Breadth-first search

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



Breadth-first search

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



Breadth-first search

- Completeness: is a solution always found if one exists?
  - YES
    - If shallowest goal node is at some finite depth *d*
    - If branching factor *b* is finite
- BF search is optimal if the path cost is a non-decreasing function of the depth of the node

Breadth-first search

- Time complexity
- Assume a state space where every state has b successors
  - Assume solution is at depth d
  - Worst case: expand all but the last node at depth d
  - Total number of nodes generated:

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

- Space complexity: every node generated must remain in memory, so same as time complexity

Breadth-first search

- Memory requirements are a bigger problem than execution time
- Exponential complexity search problems cannot be solved by BF search (or any uninformed search method) for any but the smallest instances

DEPTH	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte

Uniform-cost search

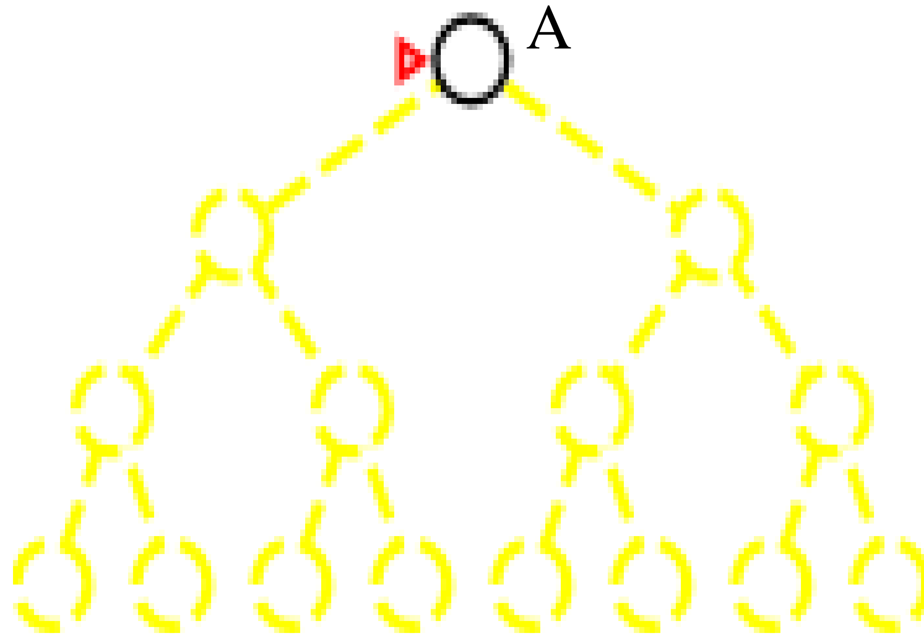
- Extension of BF-search:
  - Expand node with *lowest path cost*
- Implementation: *fringe* = queue ordered by path cost
- UC-search is the same as BF-search when all step-costs are equal

Uniform-cost search

- Completeness:
  - YES, if step-cost > ε
- Time and space complexity:
  - Assume C* is the cost of the optimal solution
  - Assume that every action costs at least ε
  - Worst-case: $O(b^{C^*/\varepsilon})$
- Optimality:
  - nodes expanded in order of increasing path cost
  - YES, if complete

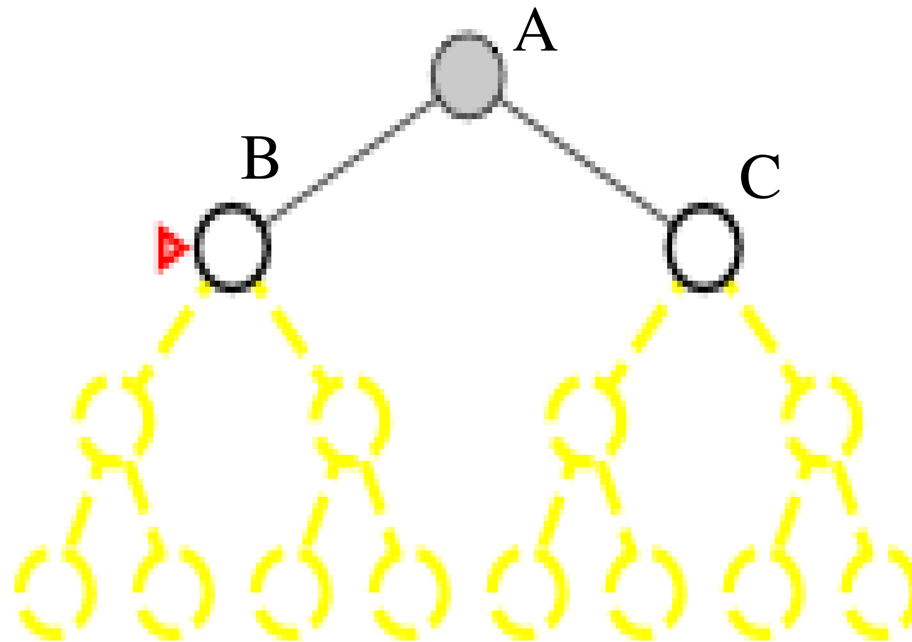
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



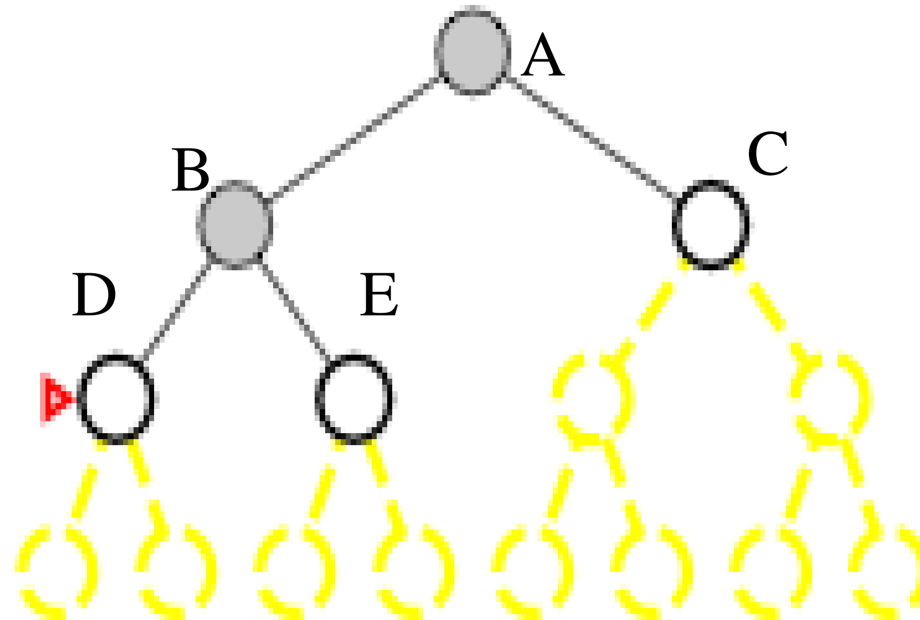
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



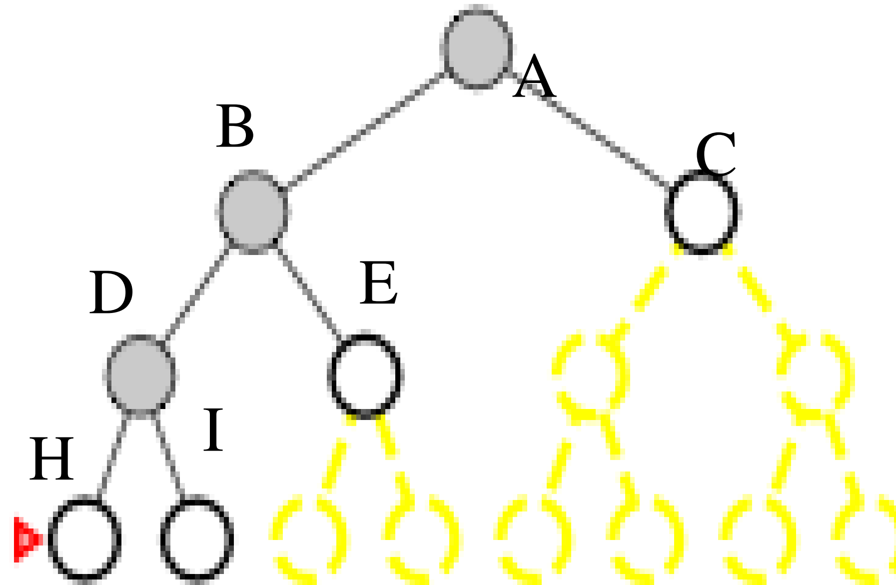
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



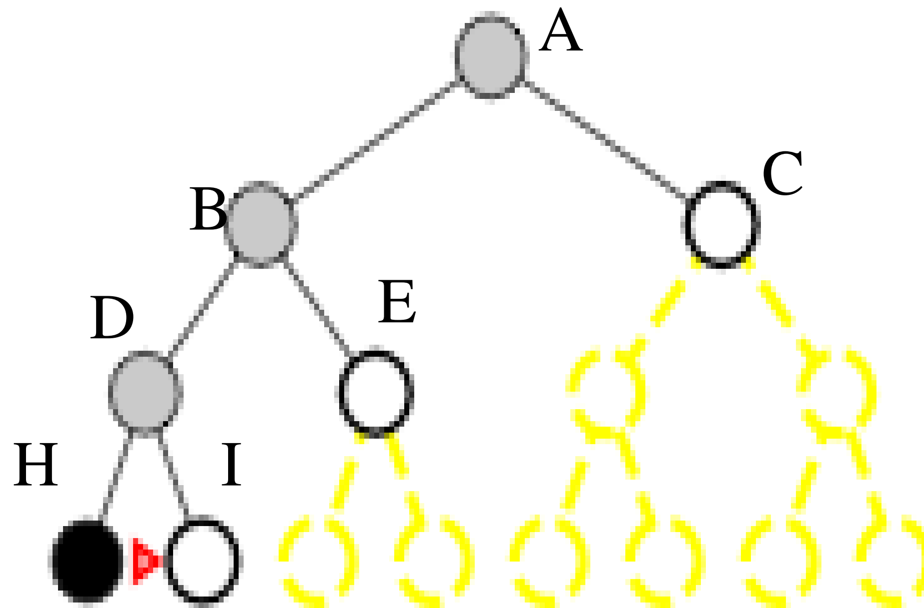
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



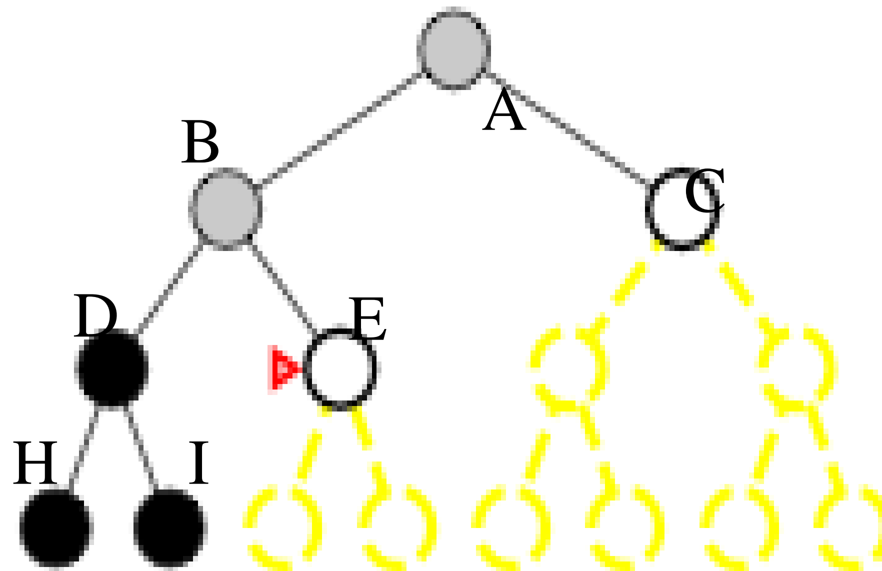
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



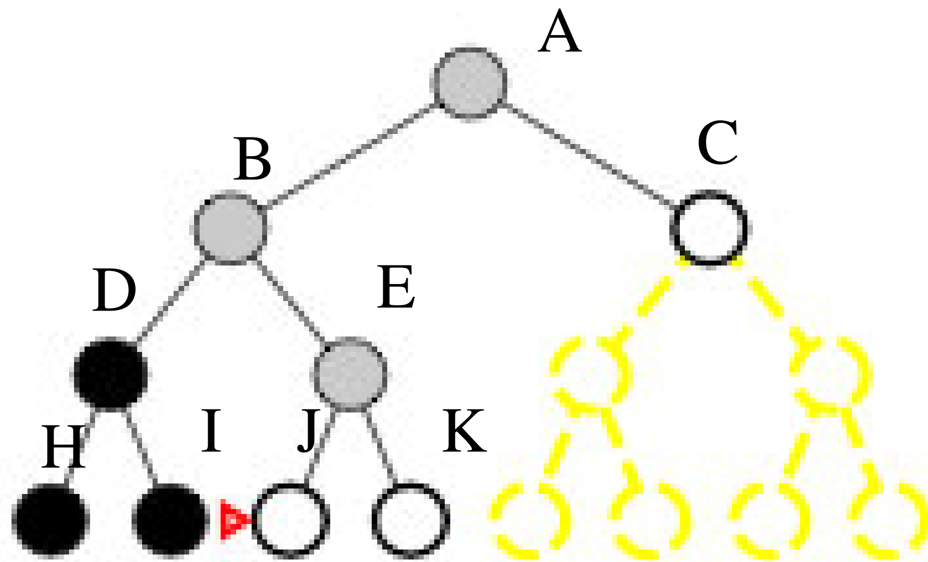
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



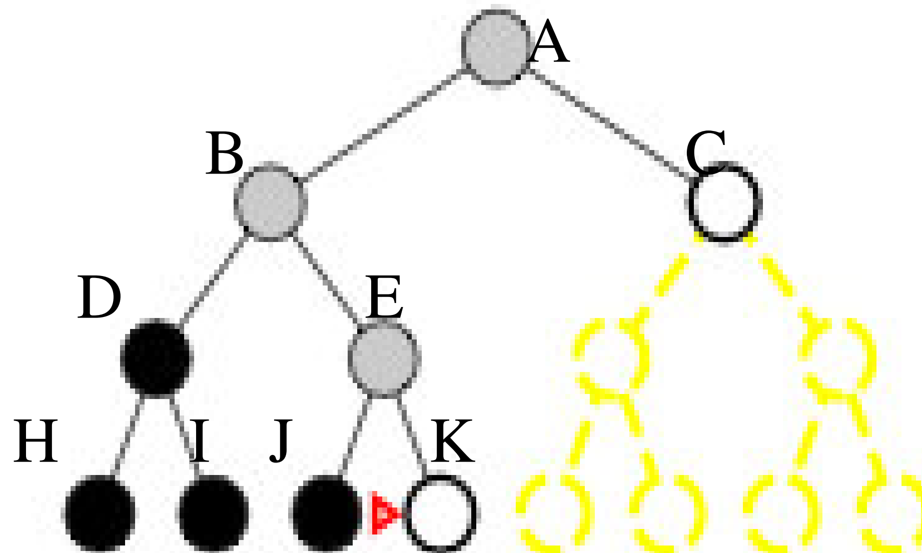
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



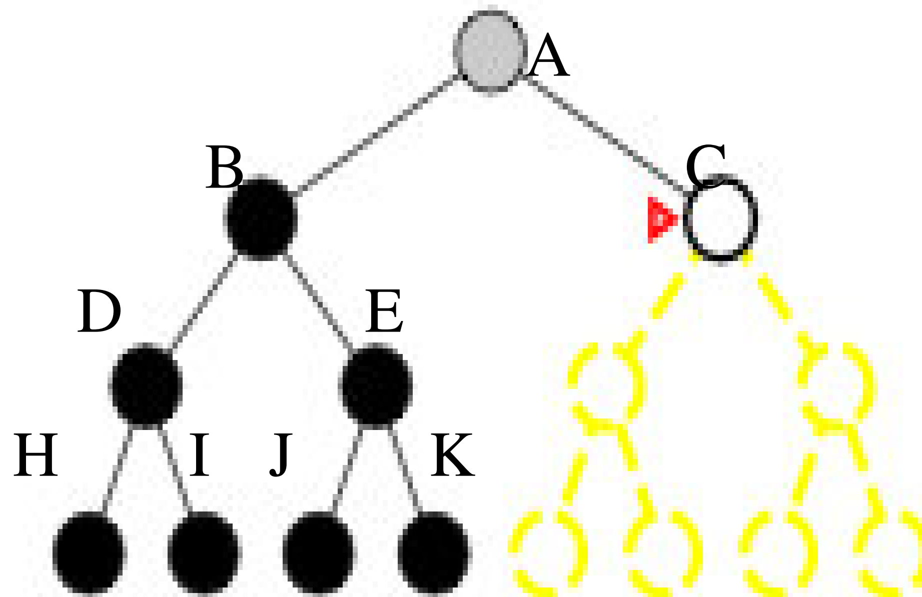
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



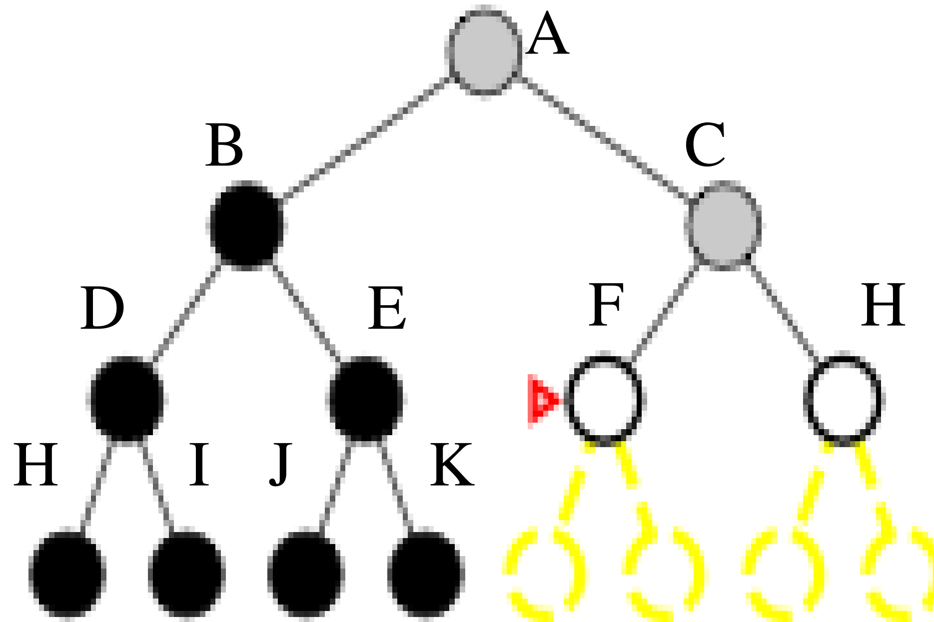
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



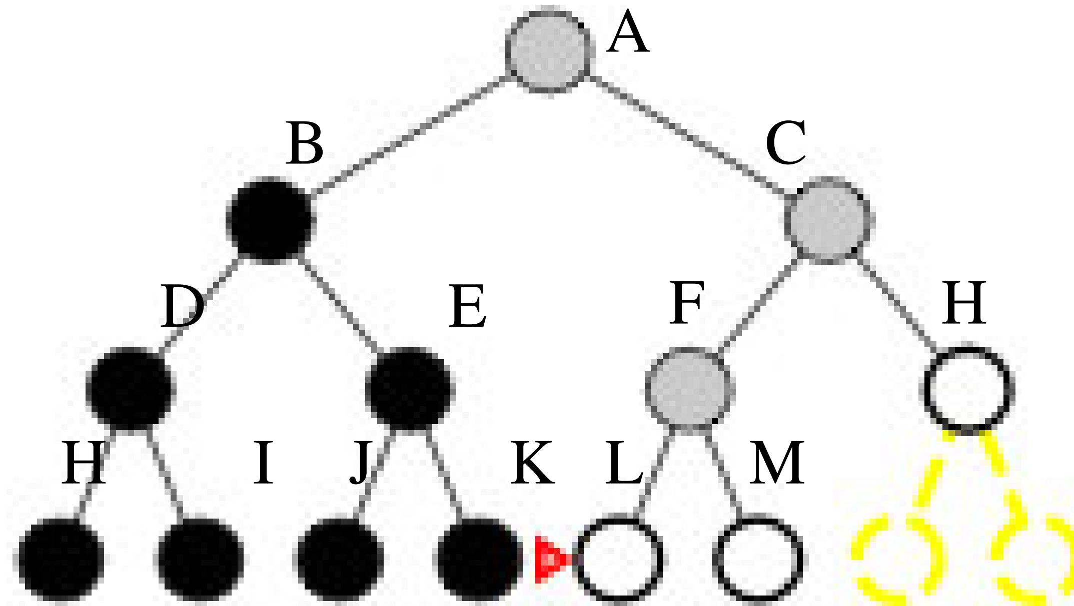
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



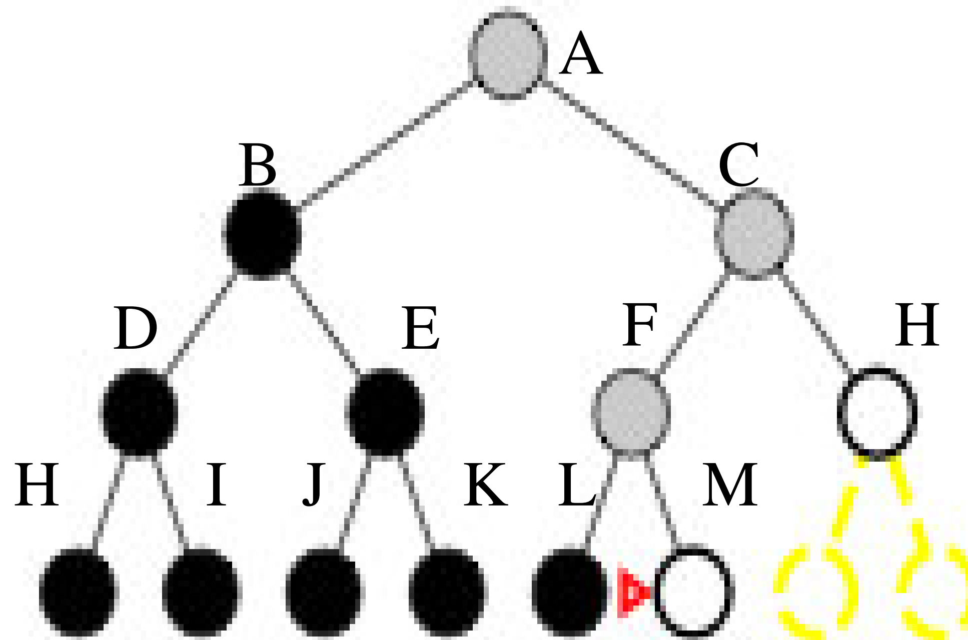
Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



Depth-first search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



DF-search: evaluation

- Completeness:
  - Is a solution always found if one exists?
  - No (unless search space is finite and no loops are possible)
- Optimality:
  - Is the least-cost solution always found?
  - No

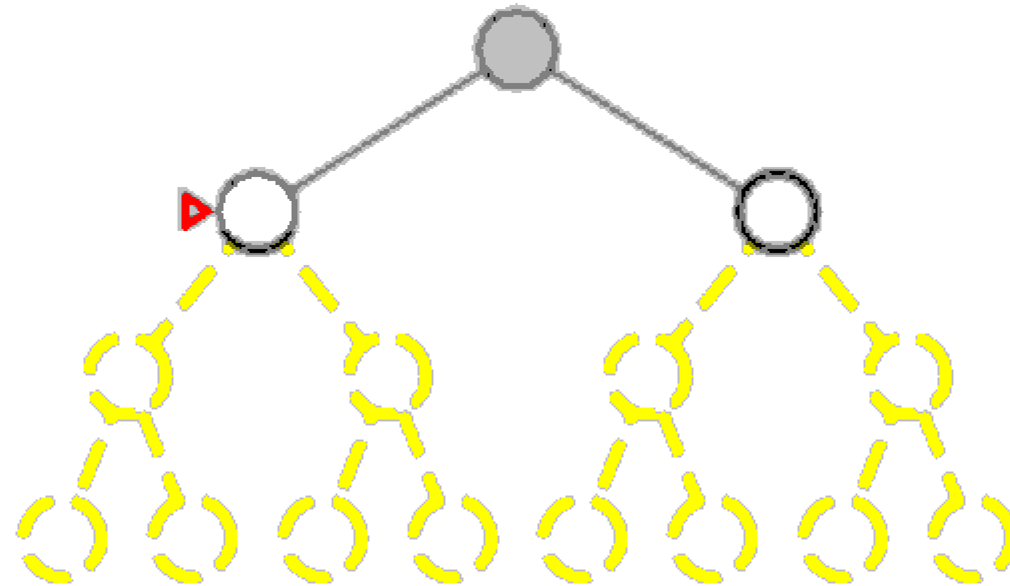
DF-search: evaluation

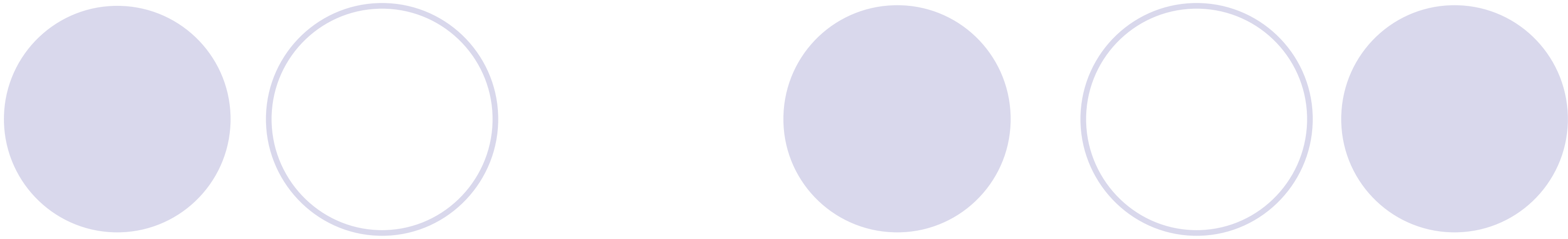
- Time complexity: $O(b^m)$
  - In general, time is terrible if *m* (maximal depth) is much larger than *d* (depth of shallowest solution)
  - But if there exist many solutions then faster than BF-search
- Space complexity: $O(bm+1)$
  - Backtracking search uses even less memory (one successor instead of all *b)*

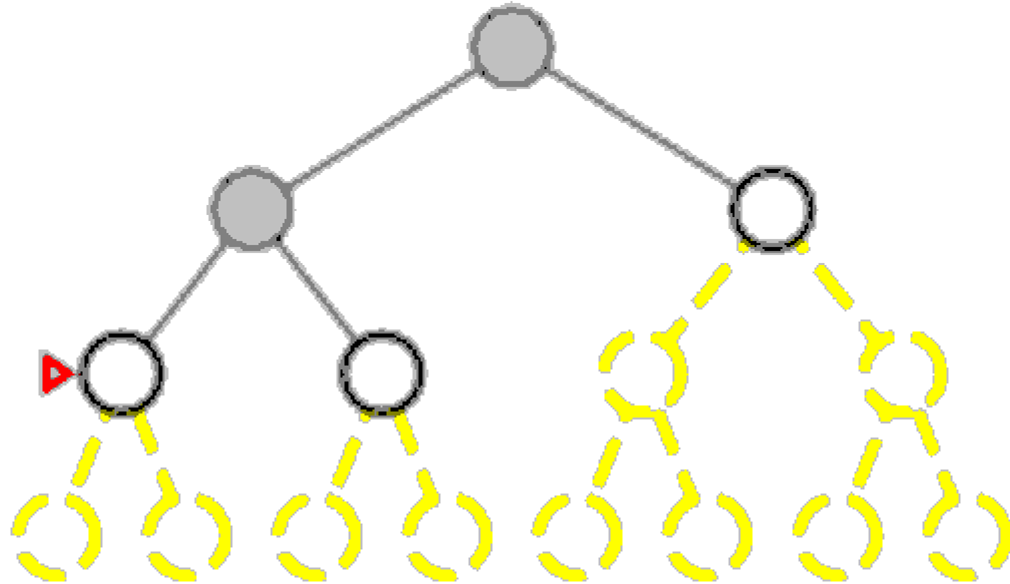
Depth-limited search

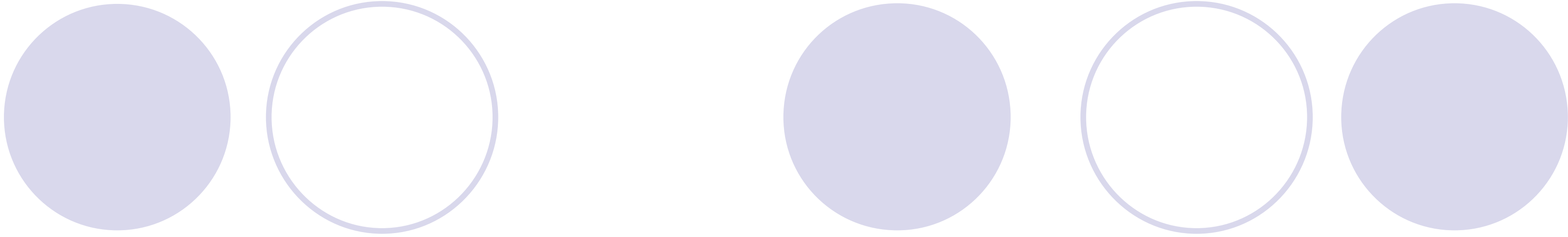
- DF-search with depth limit l
  - i.e. nodes at depth l have no successors
  - Problem knowledge can be used
- Solves the infinite-path problem
- If $l < d$ then incompleteness results
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$

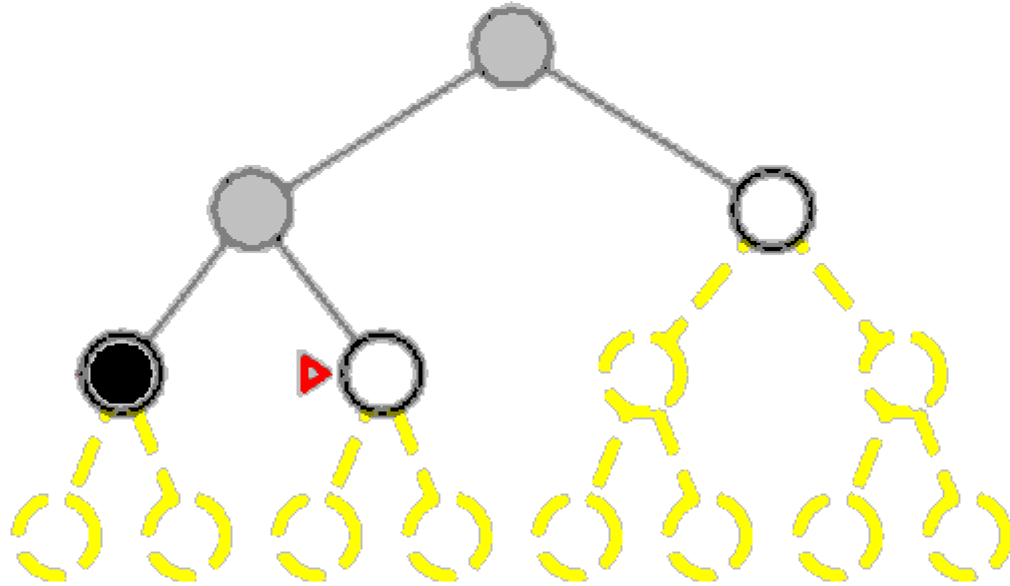
Depth-limited search with l=2

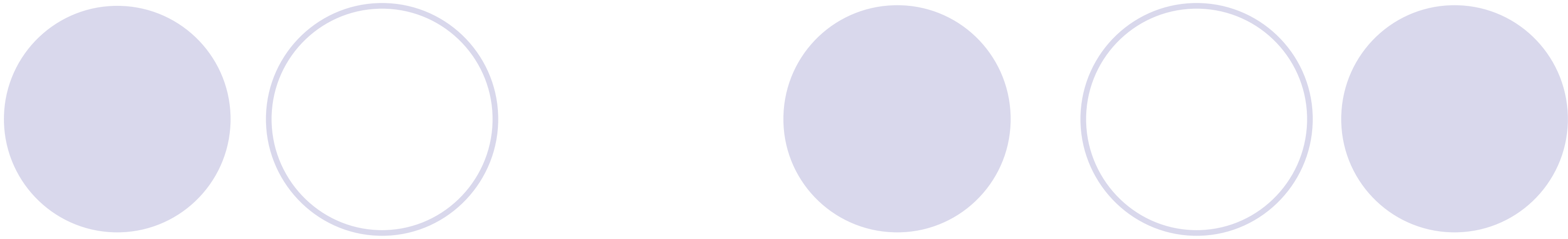


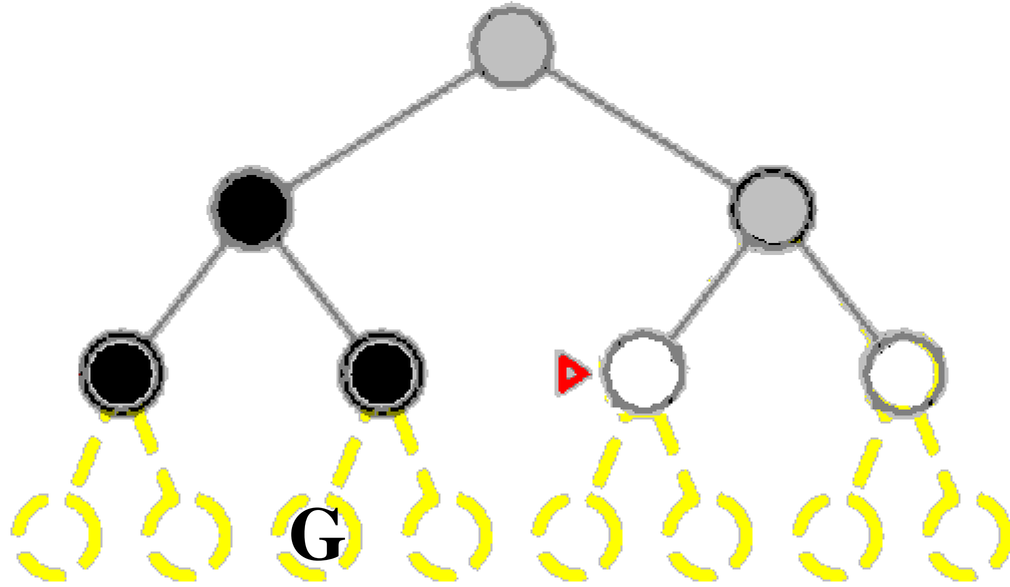











G

Depth-limited algorithm

```
function DEPTH-LIMITED-SEARCH(problem,limit) return a solution or
    failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(INITIAL-
    STATE[problem]),problem,limit)

function RECURSIVE-DLS(node, problem, limit) return a solution or failure/cutoff
    cutoff_occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] == limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result == cutoff  then cutoff_occurred? ←  true
        else if result ≠ failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

Iterative deepening search

- A general strategy to find best depth limit l
  - Goal is found at depth d, the depth of the shallowest goal-node
  - Often used in combination with DF-search
- Combines benefits of DF- and BF-search

Iterative deepening search

function ITERATIVE_DEEPENING_SEARCH(*problem*) **return** a solution or failure

inputs: *problem*

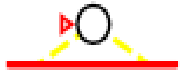
for *depth* ← 0 to ∞ **do**

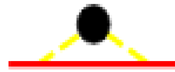
result ← DEPTH-LIMITED_SEARCH(*problem*, *depth*)

if *result* ≠ *cuttoff* **then return** *result*

ID-search, example

- Limit=0



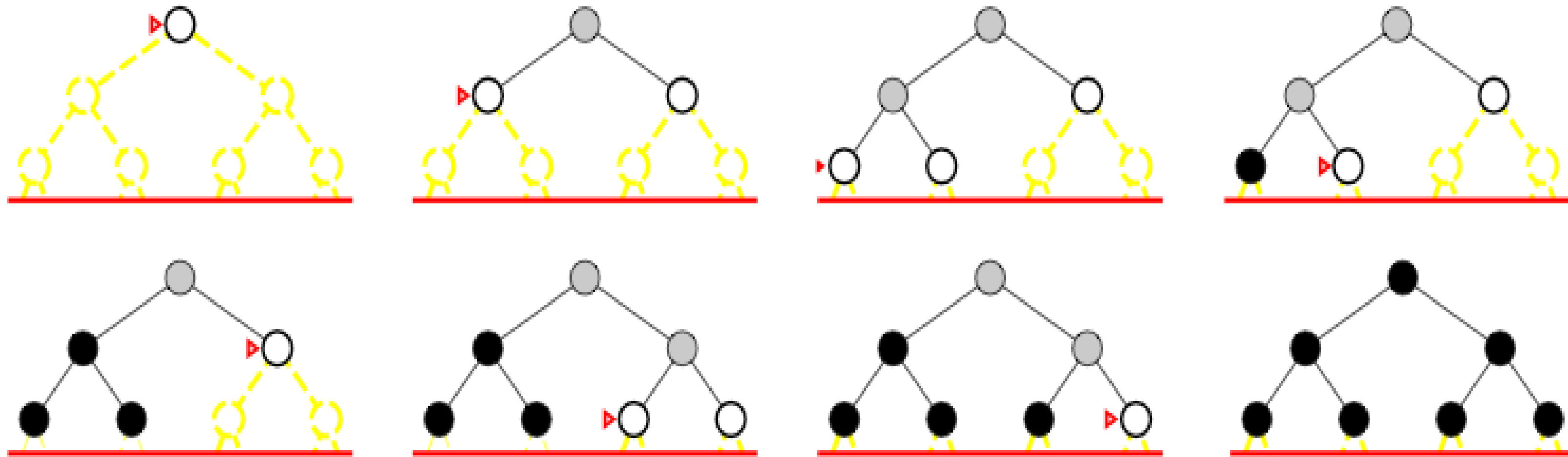


ID-search, example

- Limit=1

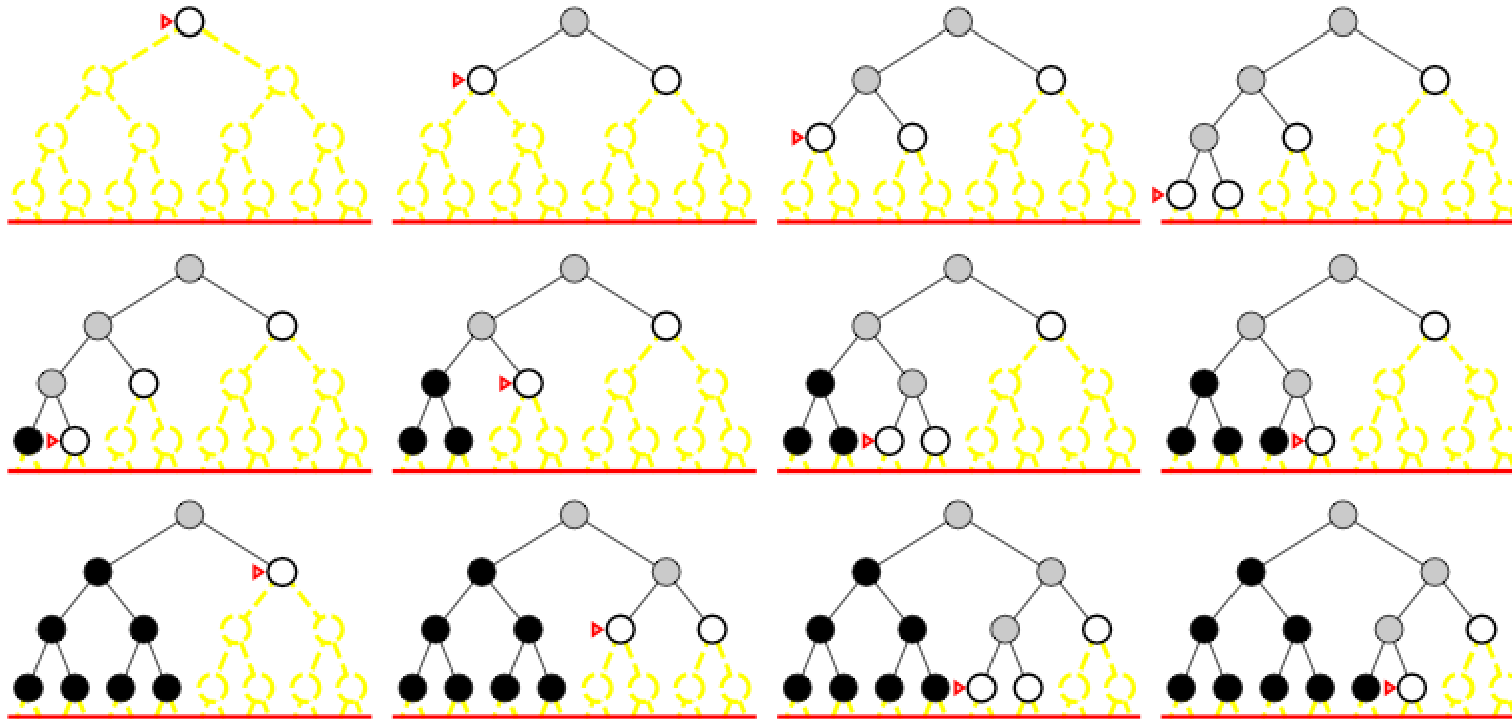
ID-search, example

- Limit=2



ID-search, example

- Limit=3



ID search: evaluation

- Time complexity: $O(b^d)$
  - Algorithm *seems* costly due to repeated generation of certain states
  - Node generation: $N(IDS) = (d)b + (d-1)b^2 + \ldots + (1)b^d$
    - level d: once
    - level d-1: 2
    - level d-2: 3
    - …
    - level 2: d-1
    - level 1: d

$$N(BFS) = b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

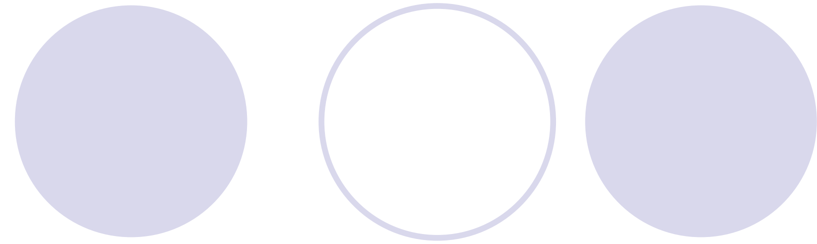
$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

Comparison for b=10 and d=5, solution at far right

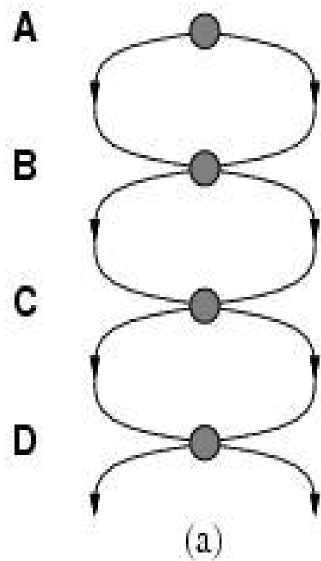
Summary of algorithms

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	YES*	YES*	NO	YES, if $l \geq d$	YES	YES*
Time	b^{d+1}	$b^{C^*/e}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{C^*/e}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES*	YES*	NO	NO	YES	YES

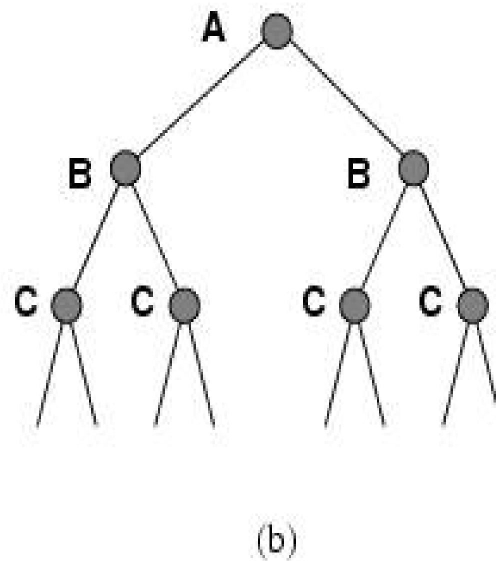
Repeated states



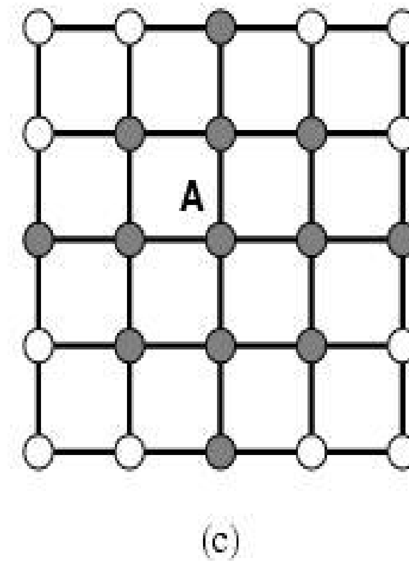
- Failure to detect repeated states can turn solvable problems into unsolvable ones



(a)



(b)



(c)

Graph search algorithm

- Closed list stores all expanded nodes

function GRAPH-SEARCH(*problem,fringe*) **return** a solution or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
    loop do
        if EMPTY?(*fringe*) **then return** failure
        node ← REMOVE-FIRST(*fringe*)
        if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
            then return SOLUTION(*node*)
        if STATE[*node*] is not in *closed* **then**
            add STATE[*node*] to *closed*
            fringe ← INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

Graph search, evaluation

- Optimality:
  - GRAPH-SEARCH discards newly discovered paths
    - This may result in a sub-optimal solution, except when uniform-cost search or BF-search with constant step cost is used
- Time and space complexity:
  - proportional to the size of the state space

    (may be much smaller than $O(b^d)$)
  - DF- and ID-search with closed list no longer have linear space requirements since all nodes are stored in the closed list

Best-first search

- Node is selected for expansion based on an *evaluation function* $f(n)$
- Evaluation function estimates distance to the goal
- Choose node which *appears* best
- Implementation:
  - fringe is a priority queue sorted in ascending order of f-values

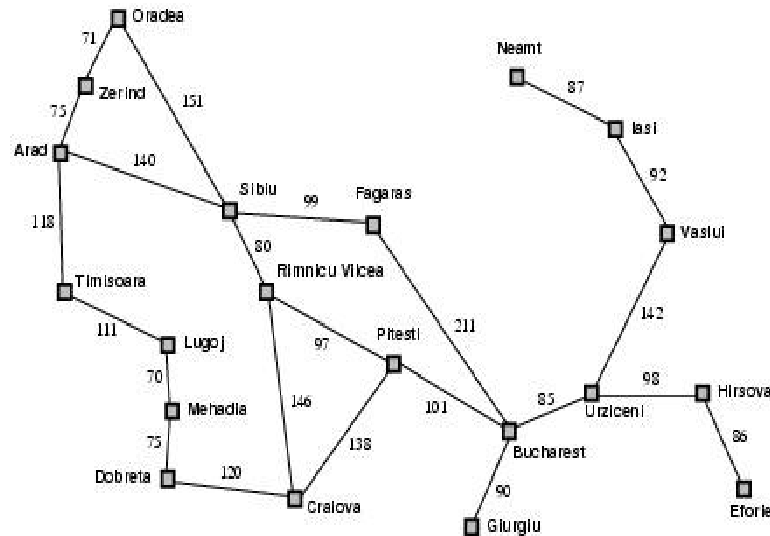
A heuristic function h(n)

- Dictionary defn: *"A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood"*
- For best-first search:

  $h(n)$ = estimated cost of the cheapest path from node n to goal node

Romania with step costs in km

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



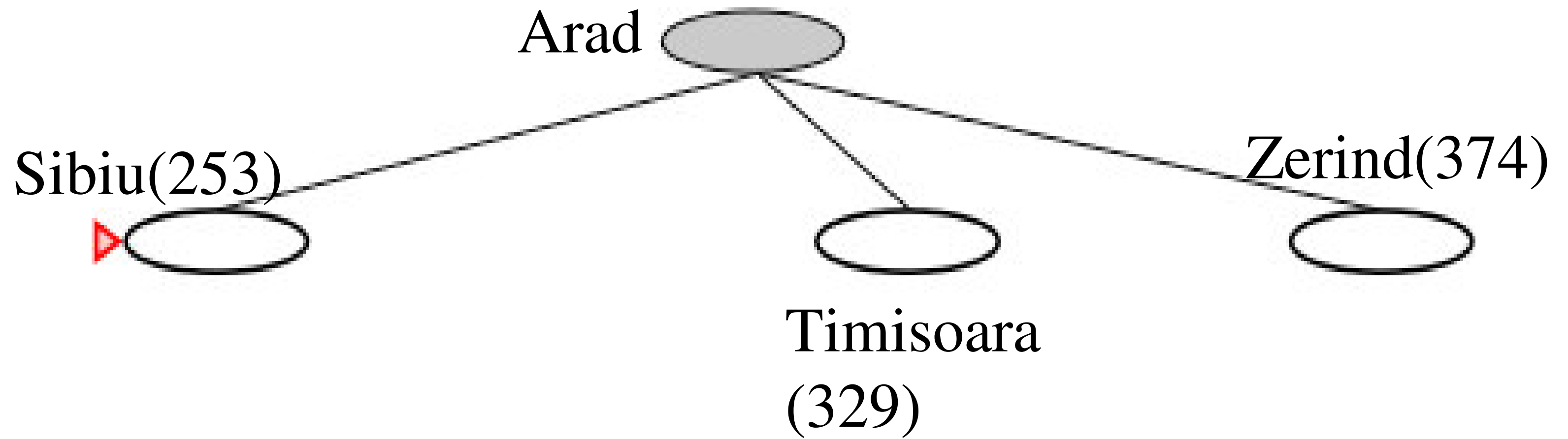
- h_{SLD}=straight-line distance heuristic
- h_{SLD} cannot be computed from the problem description itself
- In **greedy best-first search** $f(n)=h(n)$
  - Expand node that is closest to goal

Greedy search example

Arad (366)

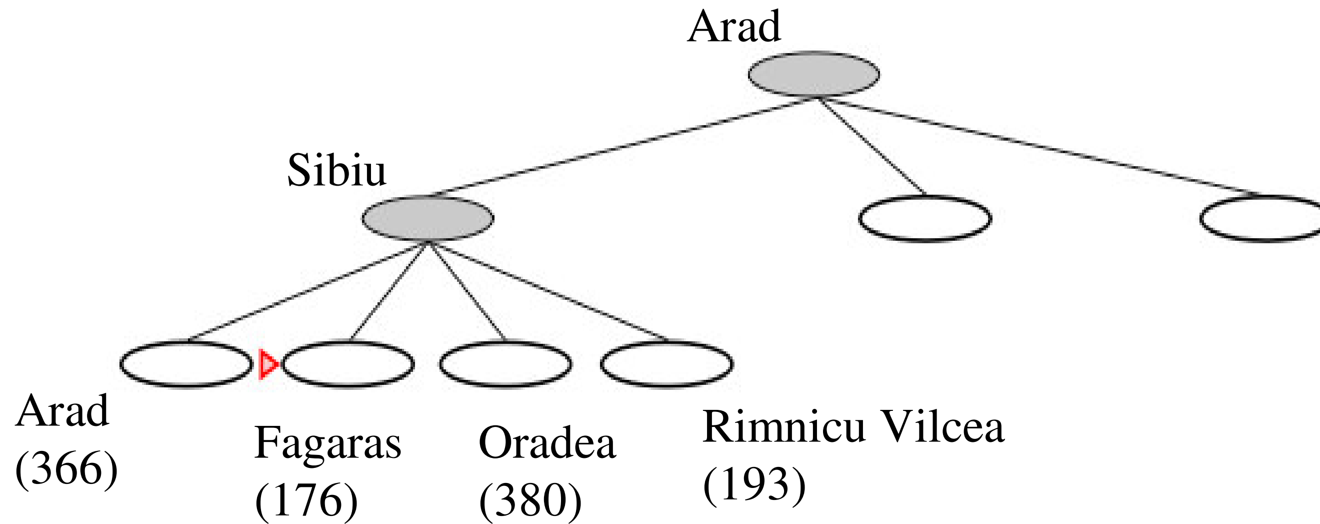
- Greedy search to solve the Arad to Bucharest problem

Greedy search example



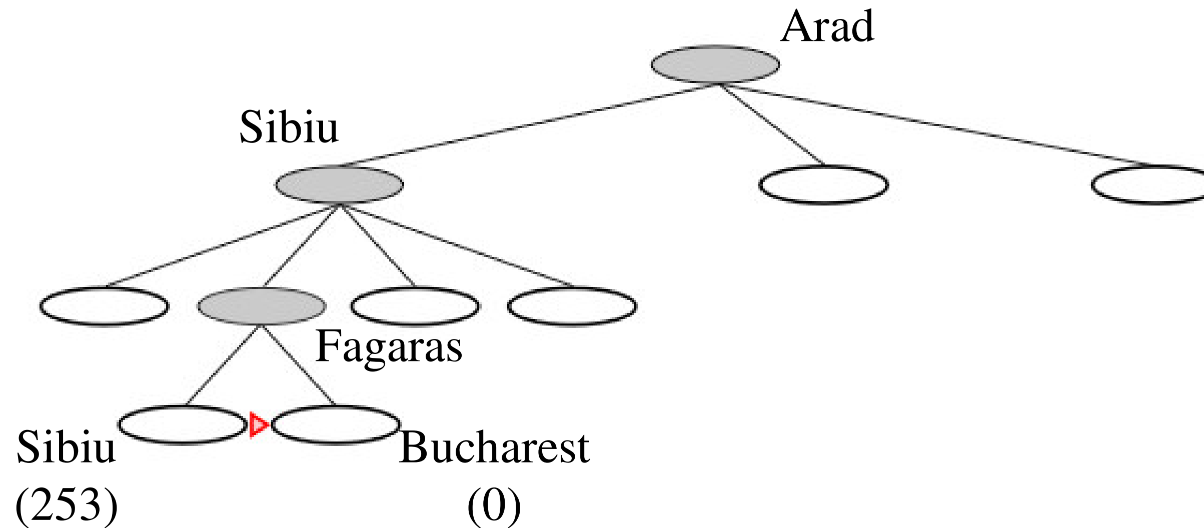
- Greedy best-first search will select Sibiu

Greedy search example



- Greedy best-first search will select Fagaras

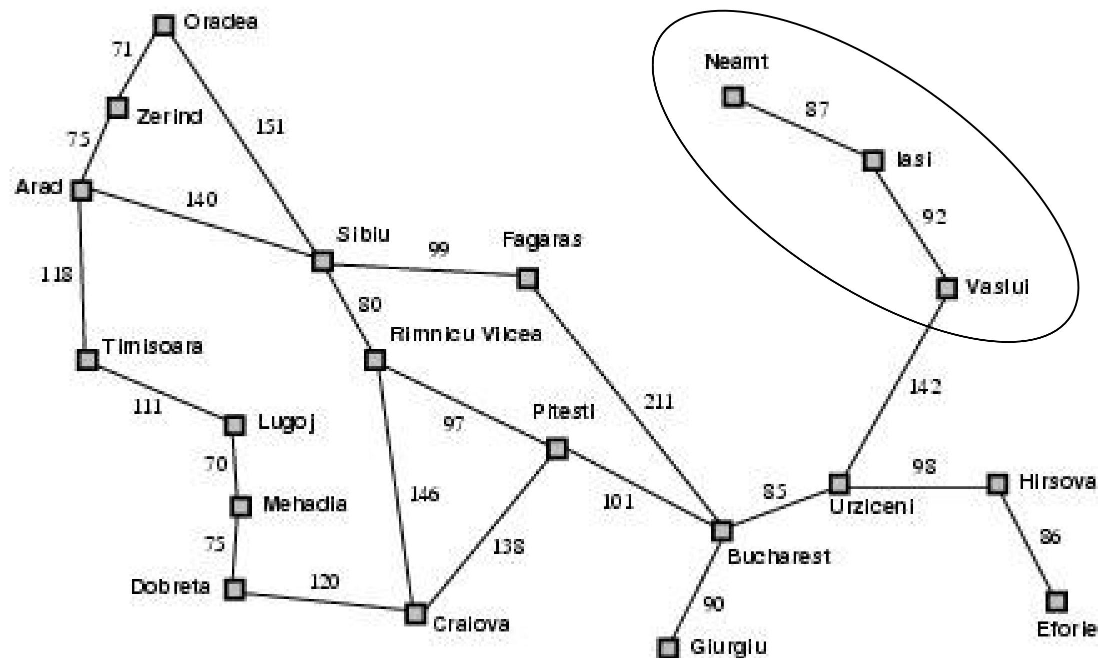
Greedy search example



- Goal reached
  - For this example no node is expanded that is not on the solution path
  - But not optimal (see Arad, Sibiu, Rimnicu Vilcea, Pitesti)

Greedy search: evaluation

- Complete or optimal: no
  - Minimizing h(n) can result in false starts, e.g. Iasi to Fagaras
  - Check on repeated states



Greedy search: evaluation

- Time and space complexity:
  - In the worst case all the nodes in the search tree are generated: $O(b^m)$

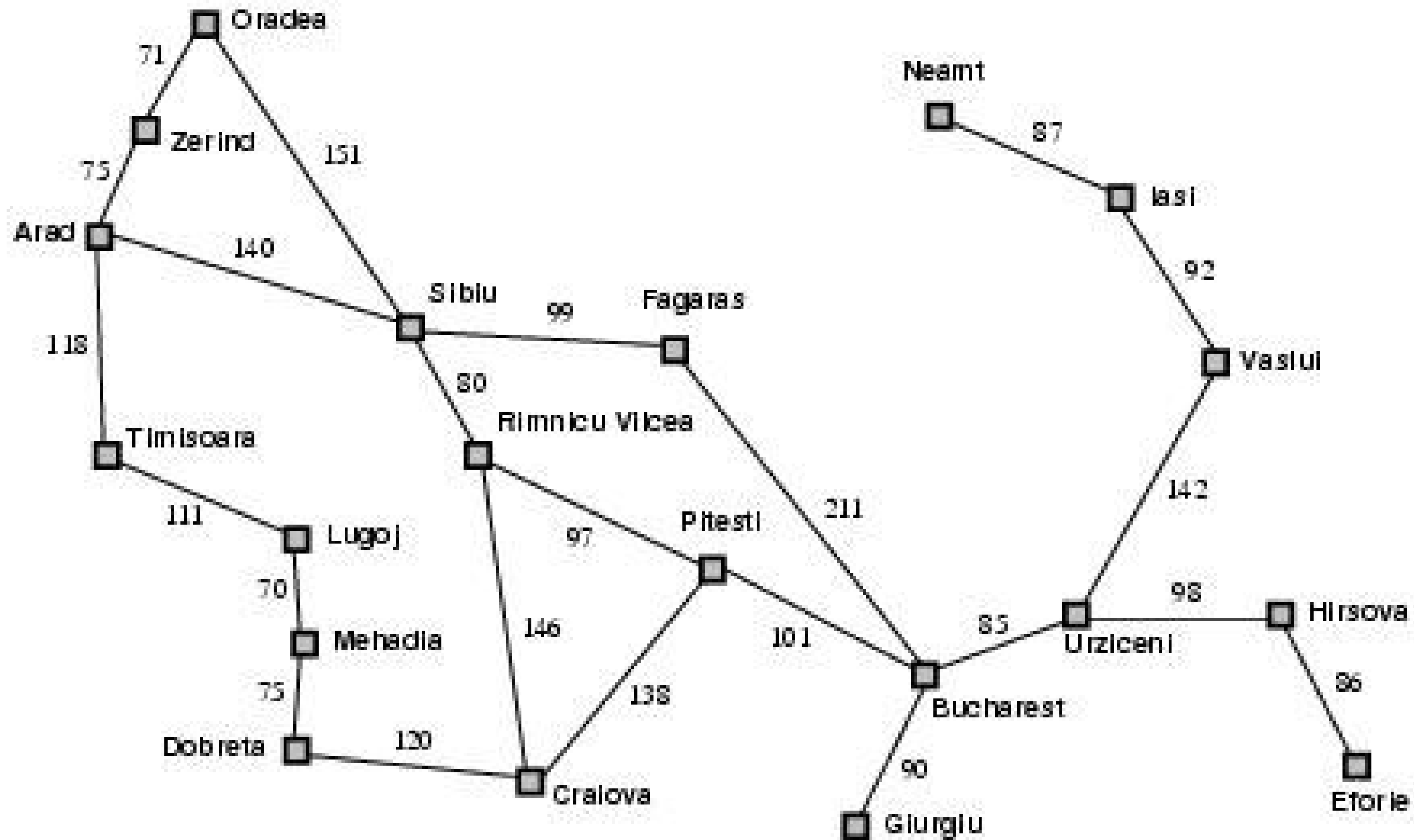
    (m is maximum depth of search tree and b is branching factor)
  - But: choice of a good heuristic can give dramatic improvement

A* search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n)=g(n) + h(n)$
  - $g(n)$: the cost (so far) to reach the node
  - $h(n)$: estimated cost to get from the node to the goal
  - $f(n)$: estimated total cost of path through n to goal
- A* search is both complete and optimal if $h(n)$ satisfies certain conditions

- A* search is optimal if h(n) is an **admissible heuristic**
- A heuristic is admissible if it *never overestimates* the cost to reach the goal
  - $h(n) <= h^*(n)$ where $h^*(n)$ is the true cost from n
- Admissible heuristics are optimistic about the cost of solving the problem
- e.g. $h_{SLD}(n)$ never overestimates the actual road distance

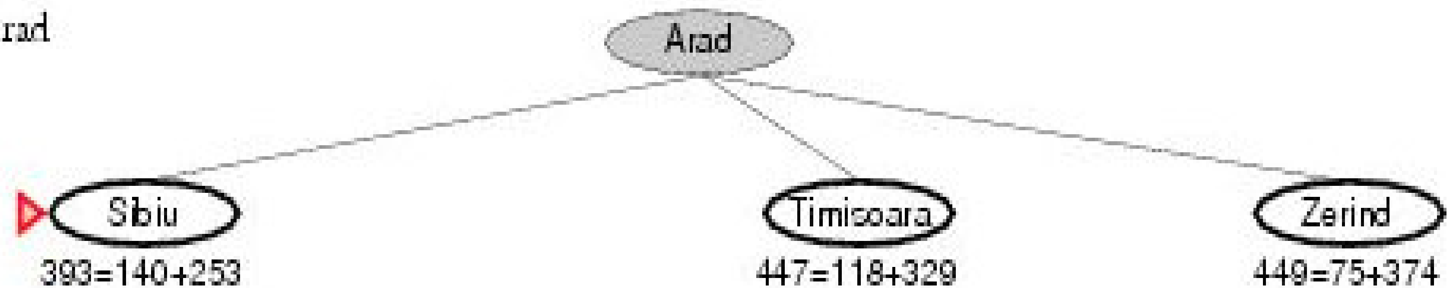
Romania example



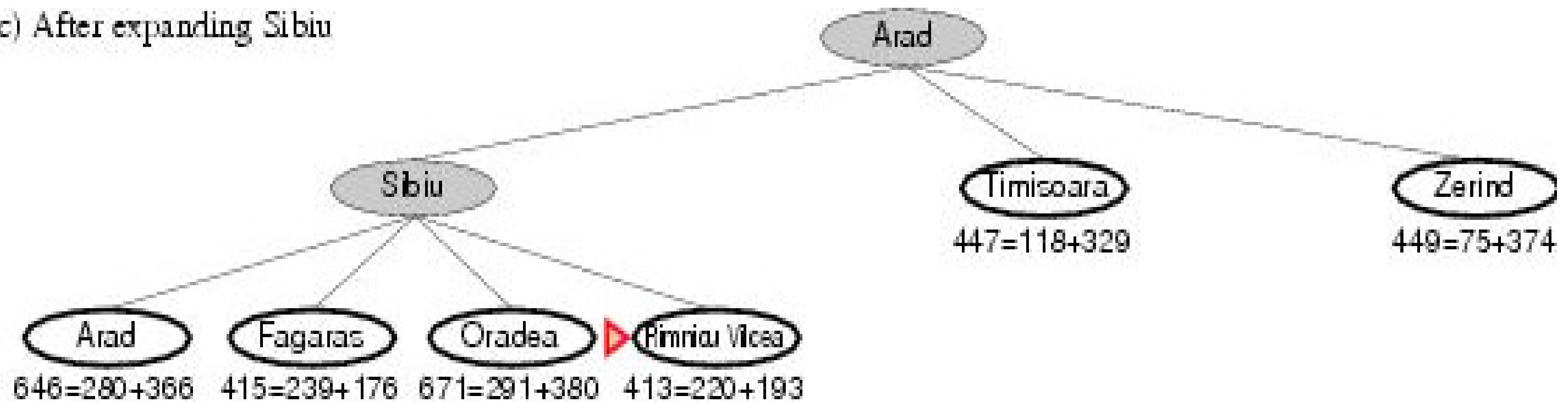
A search example*

(a) The initial state

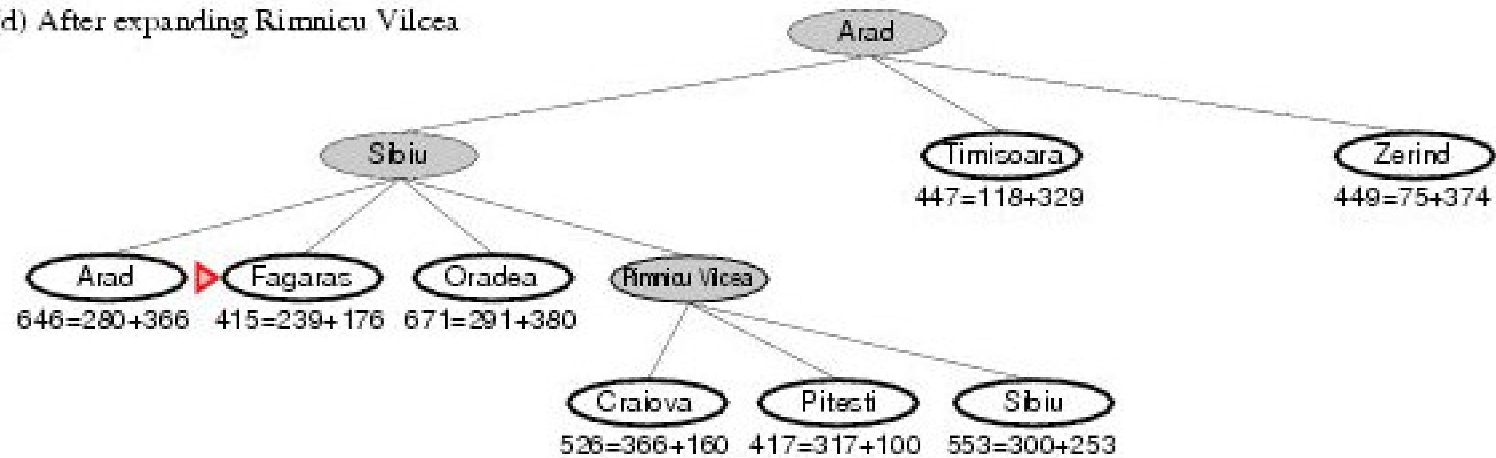
A* search example



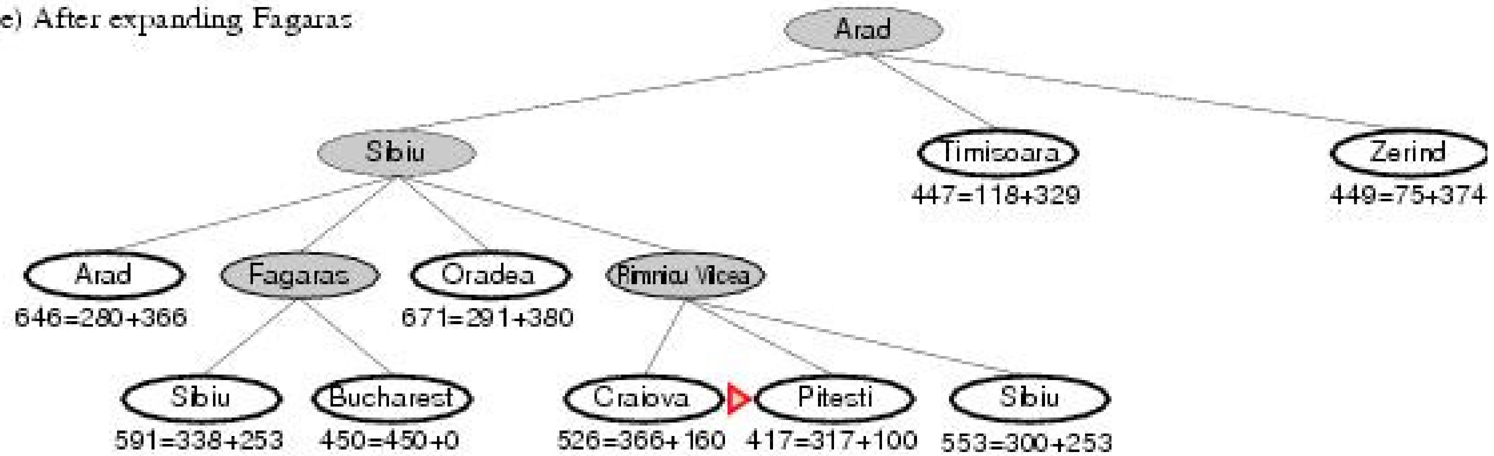
A search example*



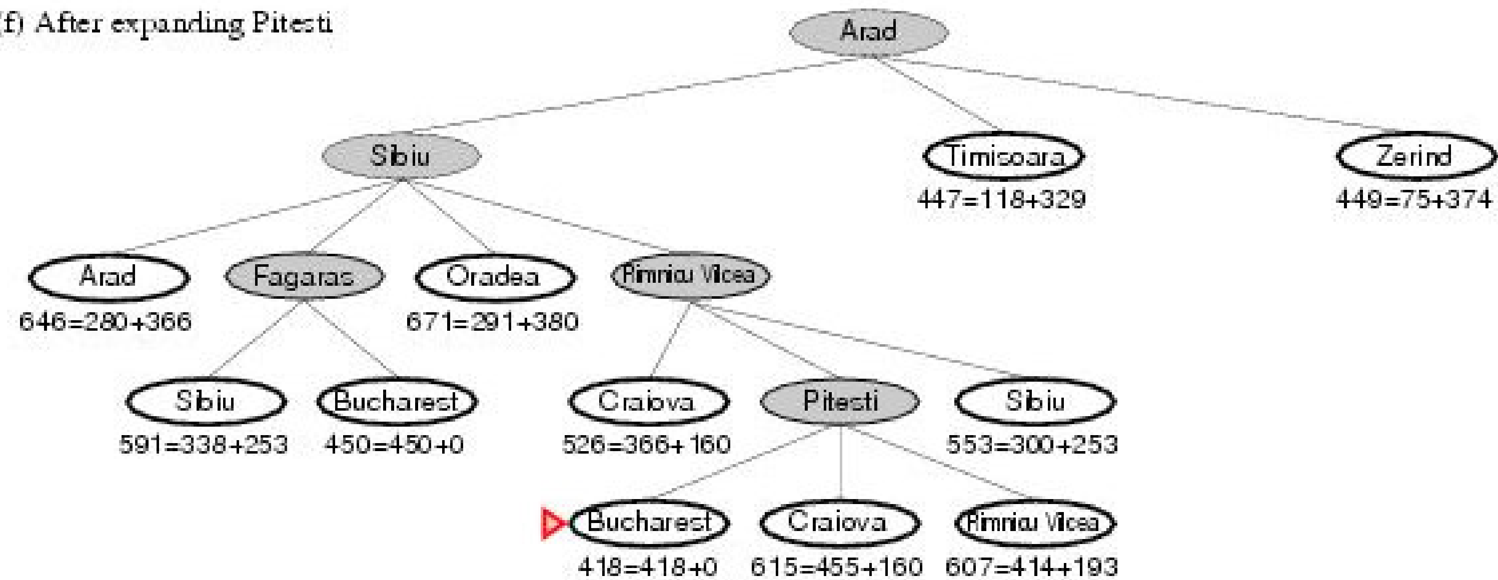
A* search example



A search example*



A* search example



A and GRAPH-SEARCH*

- GRAPH-SEARCH discards new paths to a repeated state
  - So may discard the optimal path
- Solutions:
  - Remove more expensive of the two paths
    - But requires extra book-keeping
  - Ensure that the optimal path to any repeated state is always the first one followed
    - Requires extra condition on $h(n)$: **consistency** (or **monotonicity**)

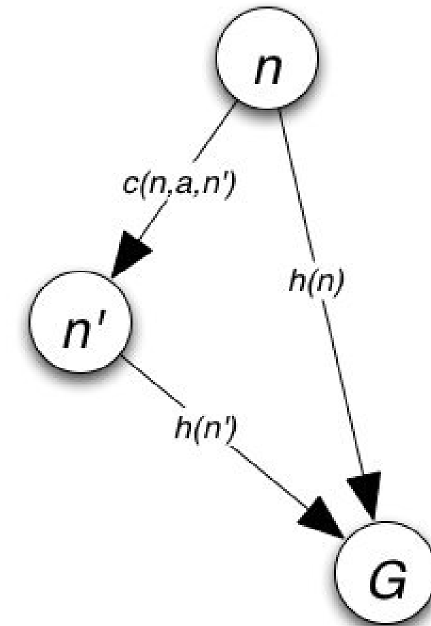
Consistency

- A heuristic is consistent if

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

- If h is consistent, we have

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



- i.e. f(n) is nondecreasing along any path
  - and so A* using GRAPH-SEARCH expands nodes in non-decreasing order of f(n)

A search: evaluation*

- Complete: yes
  - Unless there are infinitely many nodes with $f<f(G)$
  - Since bands of increasing *f* are added
- Optimal: yes
  - A* is **optimally efficient** for any given h(n): no other optimal algorithm is guaranteed to expand fewer nodes

A search: evaluation*

- Time complexity:
  - number of nodes expanded is still exponential in length of solution
- Space complexity:
  - All generated nodes are kept in memory
  - A* usually runs out of space before running out of time