CS 161 Fundamentals of Artificial Intelligence
Lecture 3

Uninformed Search

Quanquan Gu

Department of Computer Science
UCLA

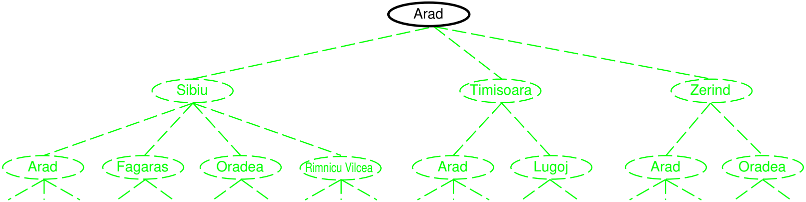
Jan 17, 2023

Tree search algorithms

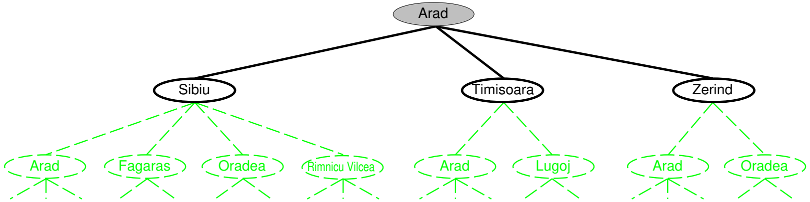
Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. **expanding** states)

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
end

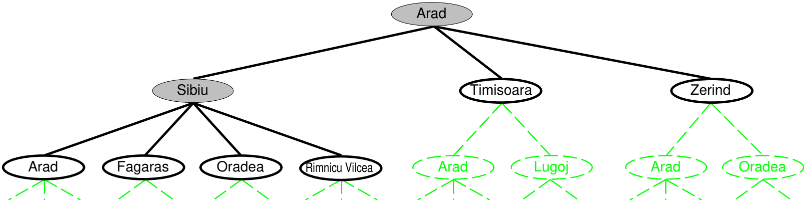
Tree search example



Tree search example

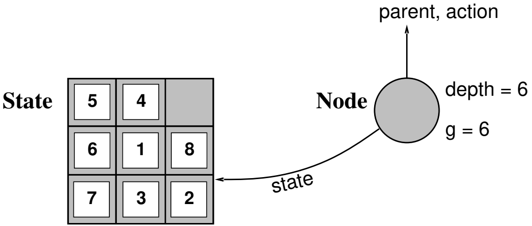


Tree search example



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 parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.

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 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(STATE[node], action, result)
DEPTH[s] ← DEPTH[node] + 1
add s to successors
return successors
```

Search strategies

A strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

- **completeness**—does it always find a solution if one exists?
- **time complexity**—number of nodes generated/expanded
- **space complexity**—maximum number of nodes in memory
- **optimality**—does it always find a least-cost 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 strategies use only the information available in the problem definition

Breadth-first search (BFS)

Depth-first search (DFS)

Depth-limited search

Iterative deepening search

Uniform-cost search

Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

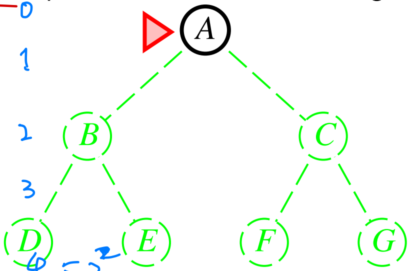
Breadth-first search

queue

Expand shallowest unexpanded node

Implementation:

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



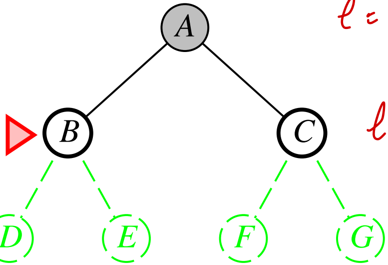
1. ∅ 0
2. A 1
3. B C 2
4. C D E 3
5. D E F G 4 ≤ 2
6. E F G 3
7. F G 2
8. G 1
9. ∅ 0

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



$\ell = 0 \quad 2^0 = 1$

$\ell = 1 \quad 2^1 = 2$

$\ell = 2 \quad 2^2 = 4$

$b = 2$

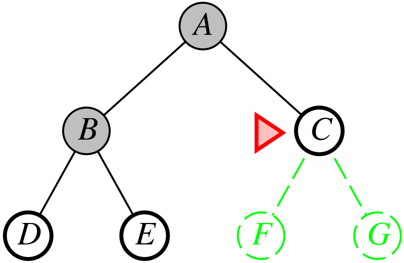
$$2^0 + 2^1 + 2^2 + \cdots + 2^d = O(2^d)$$

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

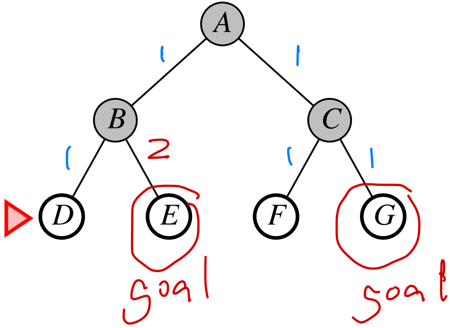


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Properties of breadth-first search

Complete?? Yes (if b is finite)

Properties of breadth-first search

Complete?? Yes (if b is finite)
Time?? $b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in d

$b^0 +$

Properties of breadth-first search

Complete?? Yes (if b is finite)
Time?? $b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in d
Space?? $O(b^d)$ (keeps every node in memory)

Properties of breadth-first search

Complete?? Yes (if b is finite)
Time?? $b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in d
Space?? $O(b^d)$ (keeps every node in memory)
Optimal?? Yes (if cost $=$ 1 per step); not optimal in general

Properties of breadth-first search

Complete?? Yes (if b is finite)
Time?? $b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in d
Space?? $O(b^d)$ (keeps every node in memory)
Optimal?? Yes (if cost = 1 per step); not optimal in general
Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Depth-first search

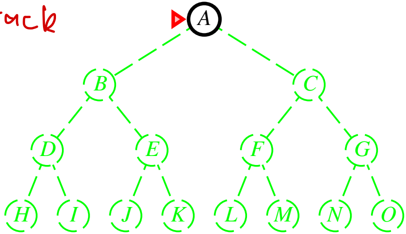
Stack

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

Stack

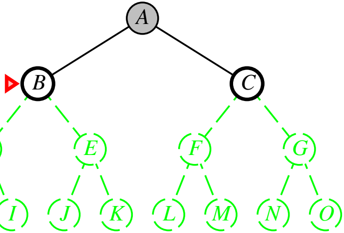


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

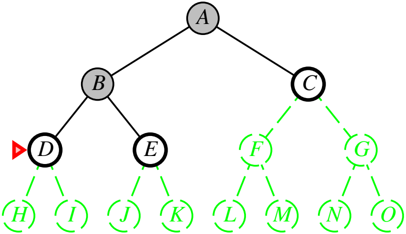


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

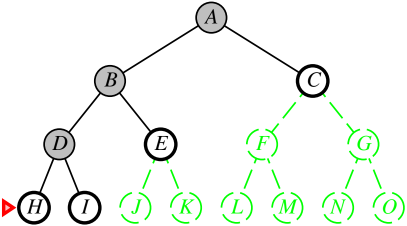


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

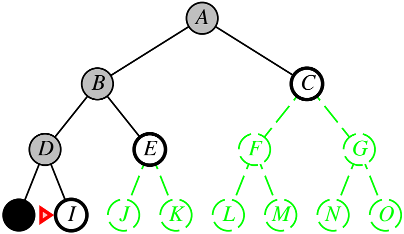


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

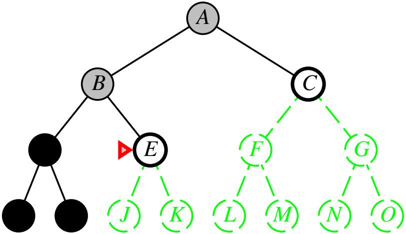


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

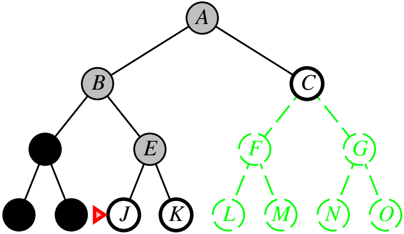


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

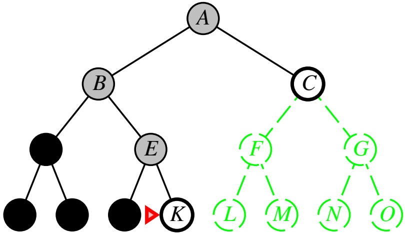


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

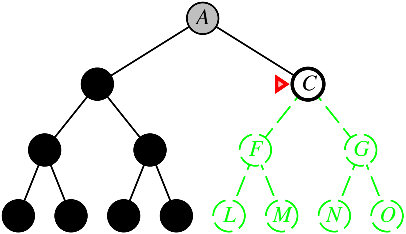


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

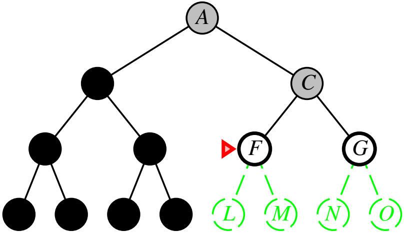


Depth-first search

Expand deepest unexpanded node

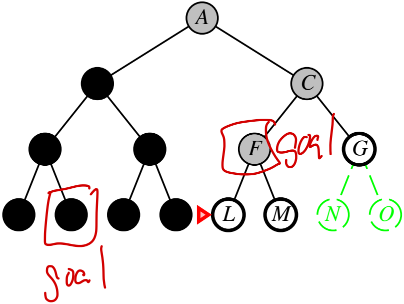
Implementation:

fringe = LIFO queue, i.e., put successors at front



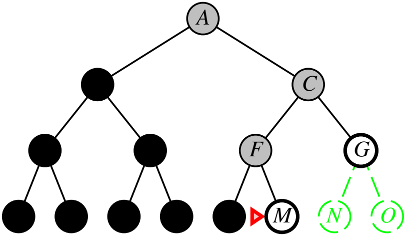
Depth-first search

Expand deepest unexpanded node
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node
Implementation:
fringe = LIFO queue, i.e., put successors at front



4=2x2

1. ϕ
2. A
3. B, C
4. D, E, C
5. H, I, E, C
6. I, E, C
7. E, C
8. J, K, C
9. K, C
10. C
11. F, G
12. L, M, G
13. M, G
14. G
15. N, O
16. O
17. ϕ

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
    Modify to avoid repeated states along path
    $\Rightarrow$ complete in finite spaces

V.S. b^d for BFS

Time?? $O(b^m)$: terrible if m is much larger than d, where m is the maximum depth of any node
    but if solutions are dense, may be much faster than breadth-first

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
    $\Rightarrow$ complete in finite spaces
Time?? $O(b^m)$: terrible if m is much larger than d, where m is the maximum depth of any node
  but if solutions are dense, may be much faster than breadth-first
Space?? $O(bm)$, i.e., linear space! V.S. $O(b^d)$ in BFS

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d, where m is the maximum depth of any node
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors

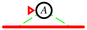
Recursive implementation:

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution node or *failure*
  for ~~*depth*~~ ℓ = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, ~~*depth*~~ ℓ)
    if *result* ≠ *cutoff* **then return** *result*

function DEPTH-LIMITED-SEARCH(*problem*, ℓ) **returns** a node or *failure* or *cutoff*
  frontier ← a LIFO queue (stack) with NODE(*problem*.INITIAL) as an element
  result ← *failure*
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    if DEPTH(*node*) > ℓ **then**
      result ← *cutoff*
    else if not IS-CYCLE(*node*) **do**
      for each *child* **in** EXPAND(*problem*, *node*) **do**
        add *child* to *frontier*
  return *result*

Iterative deepening search

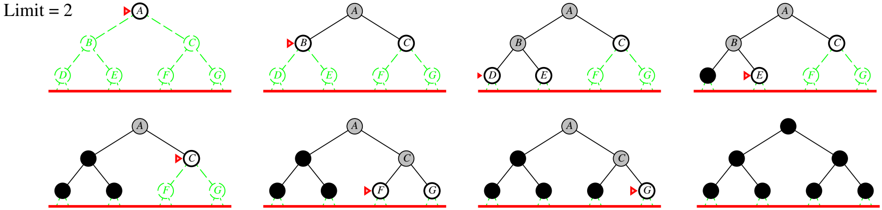
Limit = 0



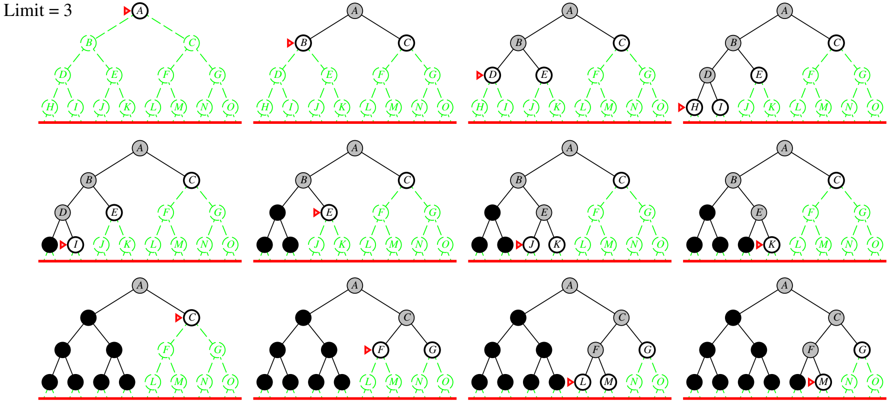
Iterative deepening search

Limit = 1

Iterative deepening search



Iterative deepening search



Properties of iterative deepening search

Complete?? Yes

Properties of iterative deepening search

Time complexity of BFS

$$(b^0) + b^1 + b^2 + \dots + b^d = O(b^d)$$

Complete?? Yes

Time?? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

$l = 1 \quad b^1$

$l = 2 \quad b^1 + b^2$

$\vdots$

$l = d \quad b^1 + b^2 + \dots + b^d$

$$d \cdot b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$$

Properties of iterative deepening search

Complete?? Yes
Time?? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
Space?? $O(bd)$

Properties of iterative deepening search

Complete?? Yes

Time?? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1

There is some extra cost for generating the upper levels multiple times, but it is not large. E.g., numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$d \cdot b^1$ $(d-1)b^2$ b^d

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

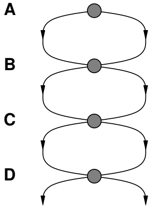
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000$$
$$= 111,110$$

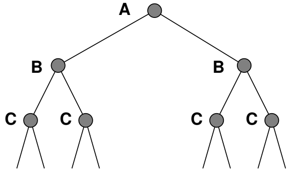
IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is **generated**

Repeated states

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





Graph search

```
function GRAPH-SEARCH( problem) returns a solution, or failure

initialize the frontier using the initial state of problem
initialize the explored set to be empty
loop do
if the frontier is empty then return failure
choose a leaf node and remove it from the frontier
if the node contains a goal state then return the corresponding solution
add the node to the explored set
expand the chosen node, adding the resulting nodes to the frontier
only if not in the frontier or explored set
end
```

Uniform-cost search

- When all step costs are equal, breadth-first search is optimal

Uniform-cost search

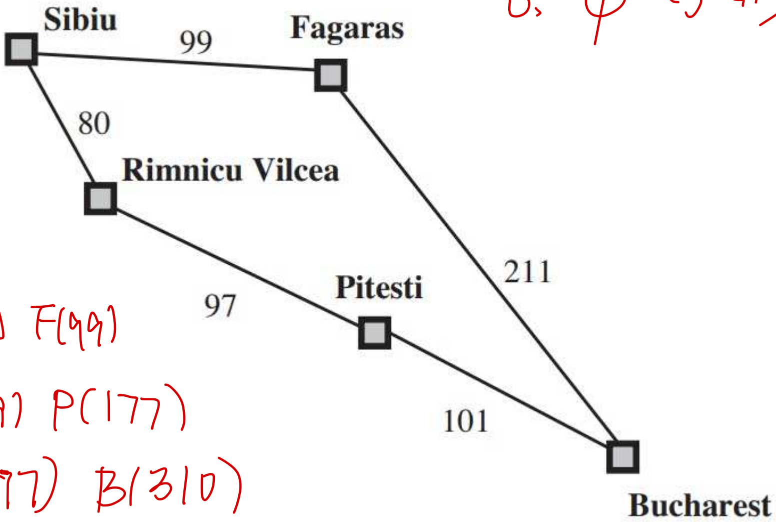
- When all step costs are equal, breadth-first search is optimal
- What if all step costs are not equal?

Uniform-cost search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Uniform-cost search

6. ∅ (goal))



1. S(0)

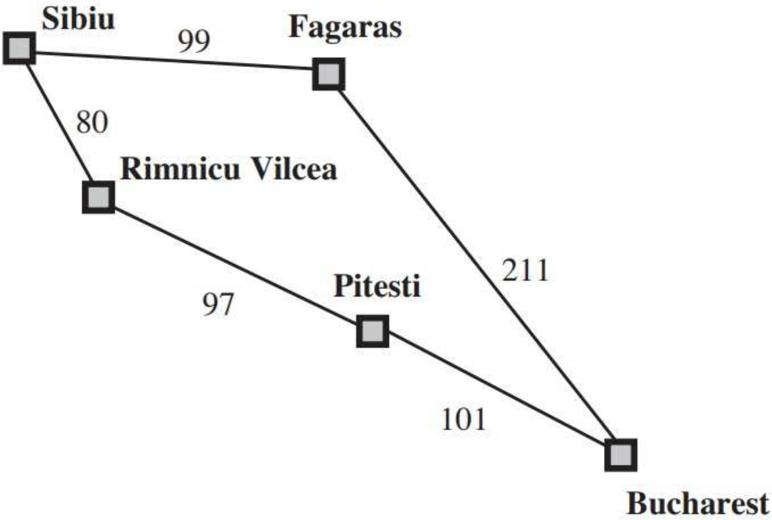
2. R(80) F(99)

3. F(99) P(177)

4. P(177) B(310)

5. B(278)

Uniform-cost search



Sibiu→ Rimnicu Vilcea→ Fagaras → Pitesti → Bucharest

Uniform-cost search

Expand least-cost unexpanded node
Let $g(n)$ be the sum of the cost (path cost) from start to node n
Implementation:
fringe = queue ordered by path cost, lowest first
Equivalent to breadth-first if step costs all equal
Complete?? Yes, if step cost $\geq \epsilon$
Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution
Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^d	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^d	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- Graph search can be exponentially more efficient than tree search

Acknowledgment

The slides are adapted from Stuart Russell et al.