Searching for Solutions:
2.2. Uninformed Search Algorithms

Outline

- Recap on problem formulation
- Algorithms and Pseudocode
- Tree and graph searches
- Uninformed search algorithms
  - BFS
  - UCS
  - DFS
  - DLS
  - IDS

Recap – Problem Formulation

- States, initial/start state, goal state
- Actions, goal test, transition models (successor functions), path cost
- Toy problems – vacuum world, 8-queens, 8-puzzle
- Abstraction of real-world problems to search problems requires creativity and intuition

Computing Basics: Algorithms & Pseudocode

- An **algorithm** is a systematic logical approach used to solve problems in a computer – recipe for solving computational problems
- Algorithms are expressed using pseudo-code – statements in plain English which may be translated later into a programming language
- A **pseudocode** is not a computer program or code, but a step-by-step recipe or process flow of how something is done in a computing environment
- Can look similar to actual code but still very high level (closer to natural language)
- Alternatives are flowcharts, UML diagrams, etc.
- Why is pseudocode useful? Help non-programmers (and sometimes even developers) understand what is happening in a computer program or algorithm

Pseudo-code/Code Components

- function (independent block that does something meaningful, e.g. convert Celsius to Fahrenheit)
- loop/for/while (repeated actions with a counter or condition that controls when the iteration should stop)
- if-then-else (control structure that only runs when some condition is satisfied)
- assignment ← (assign values to variables)
  - a variable is a data item that may take on more than one value during the runtime of a program (e.g. name, grade, number of dots eaten by Pacman)
- return (exits a function and returns a value back to the main body of code)

Pseudocode cont.

Pseudocode to Calculate the Sum & Average fo 10 Numbers

```
begin
        initialize counter to 0
        initialize accumulator to 0
        loop
                read input from keyboard
                accumulate input
                increment counter
        while counter < 10
        calculate average
        print sum
        print average
end
```

n be

Algorithm 2 CDS with betweenness centrality

Require: A connected graph $G(V, E)$

1: $d \leftarrow \{v : bw(v)\}, v \in V$, sort by BW on ascending order
2: $V' \leftarrow \varnothing$, connected dominating sets
3: **for all** $v : bw(v), v \notin V'$ **do**
4: **if** $bw(v) = 0$ OR $G(V - \{v\})$ is connected **then**
5: $V' \leftarrow V' \cup MAX - BW(N(v))$
6: **else**
7: $V' \leftarrow V' \cup \{v\}$
8: **end if**
9: $V \leftarrow V - \{v\}$
10: **end for**

Pseudocode and Actual Code

Pseudocode

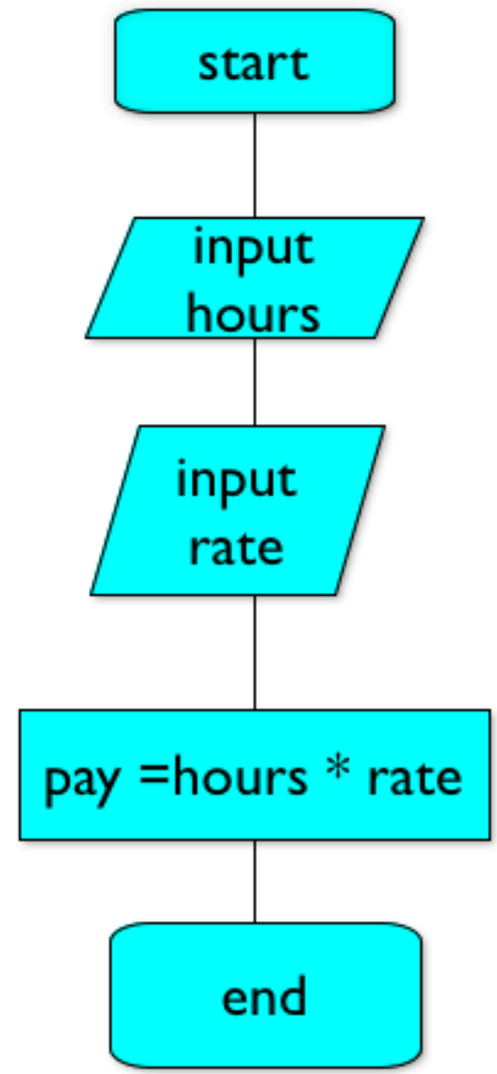
```
if the score is 90 or above
        grade is an "A"
else
        if the score is 80 or above
                grade is a "B"
        else
                if the score is 70 or above
                        grade is a "C"
                else
                        grade is an "F"
```

Actual Python code

```
if score >= 90:
    grade = "A"
else:
    if score >= 80:
        grade = "B"
    else:
        if score >= 70:
            grade = "C"
        else:
            grade = "F"
```

Pseudocode vs. Flowchart

Flowchart



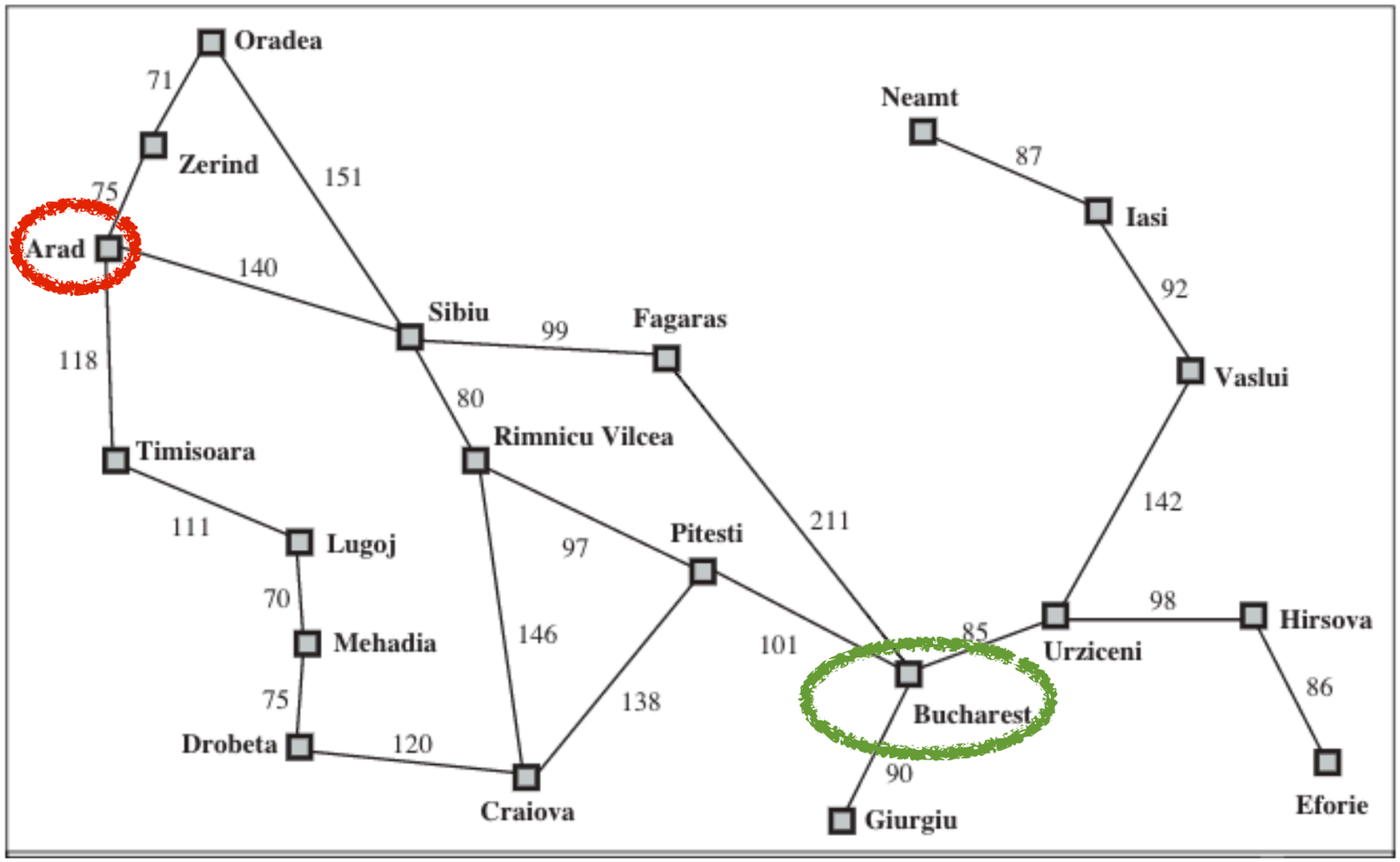
Pseudocode

```
BEGIN
    input hours
    input rate
    pay = hours * rate
    print pay
END
```

Search Trees

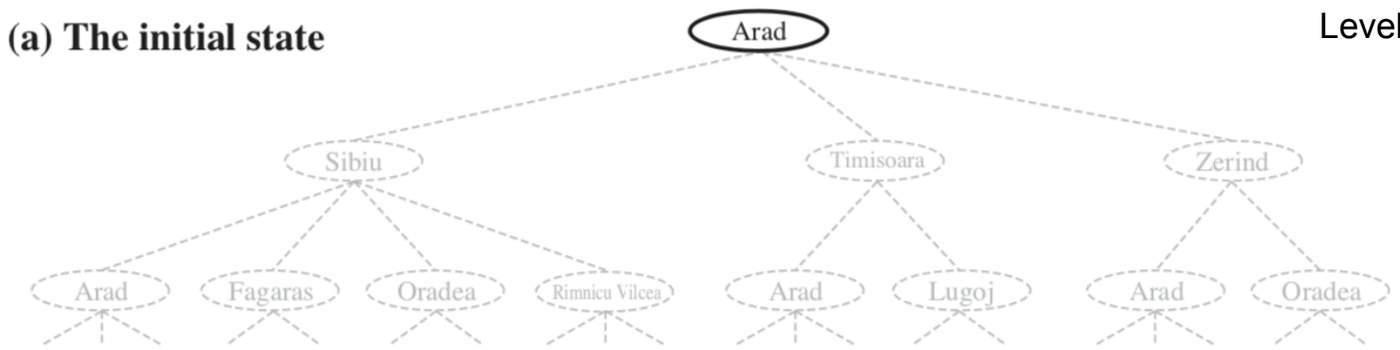
- A solution is an action sequence – the possible action sequences starting at the initial state from a search tree to a goal state
- Initial state at the root node of the tree, branches are actions and nodes correspond to states in the search space
- At each node (even at root), test if it is goal state, otherwise expand (branch out) current node (state) to new states by applying each legal action to it
- A node before expansion is called a parent node and the resulting nodes are called child nodes
- Choose one of the child nodes and perform goal test, otherwise repeat expanding nodes from this child or consider next child node
- A set of all nodes available for expansion at any given point is a frontier/ fringe
- How the algorithm chooses which state to expand next is called search strategy

Back to Romania

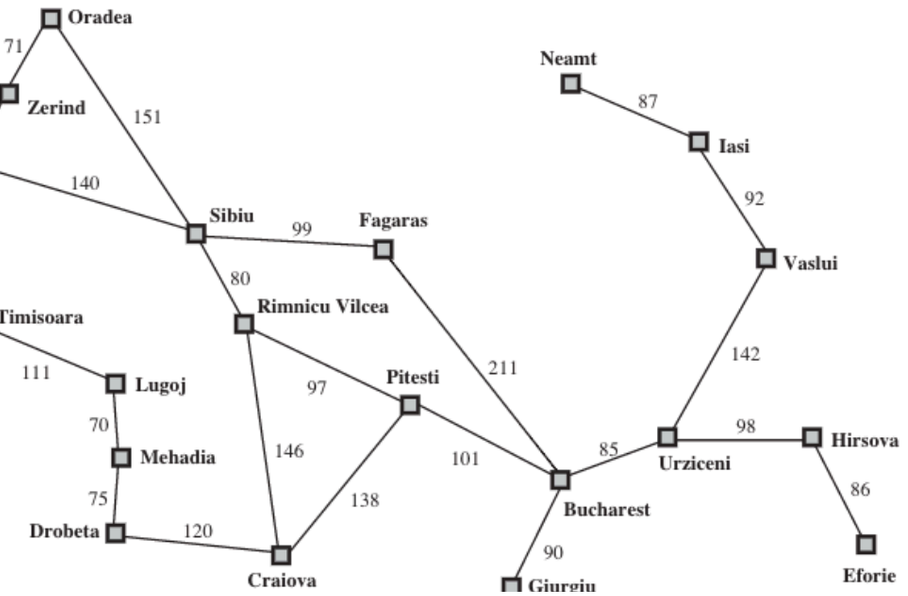


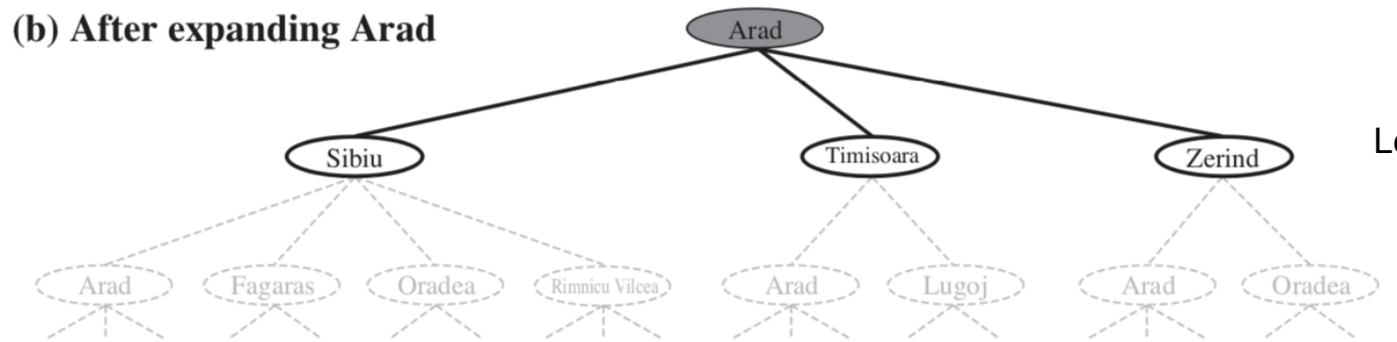
Initial state/Root node: Arad
Goal state: Bucharest
Goal test: Is current state == Bucharest?
State space: Cities
Transition model: Go to adjacent city with cost = distance
Solution? Any path from Arad to Bucharest

Search Tree – Arad to Bucharest

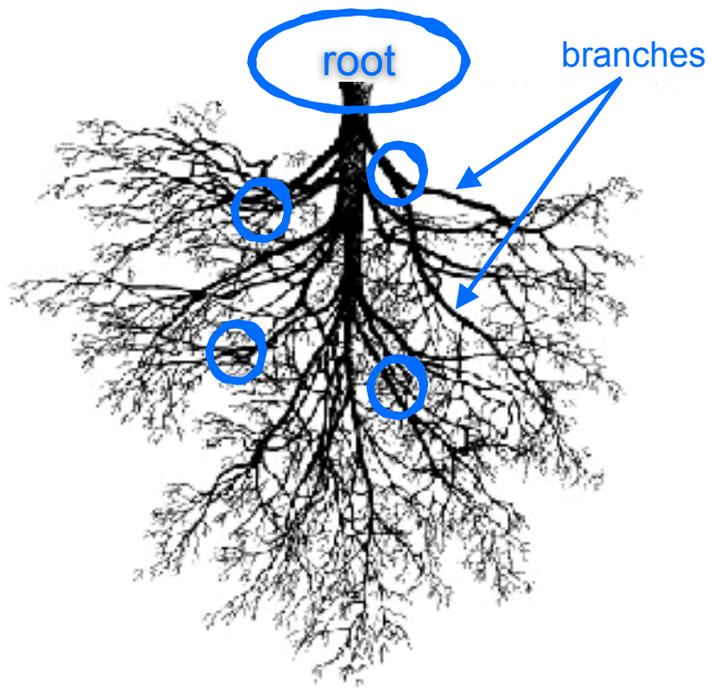


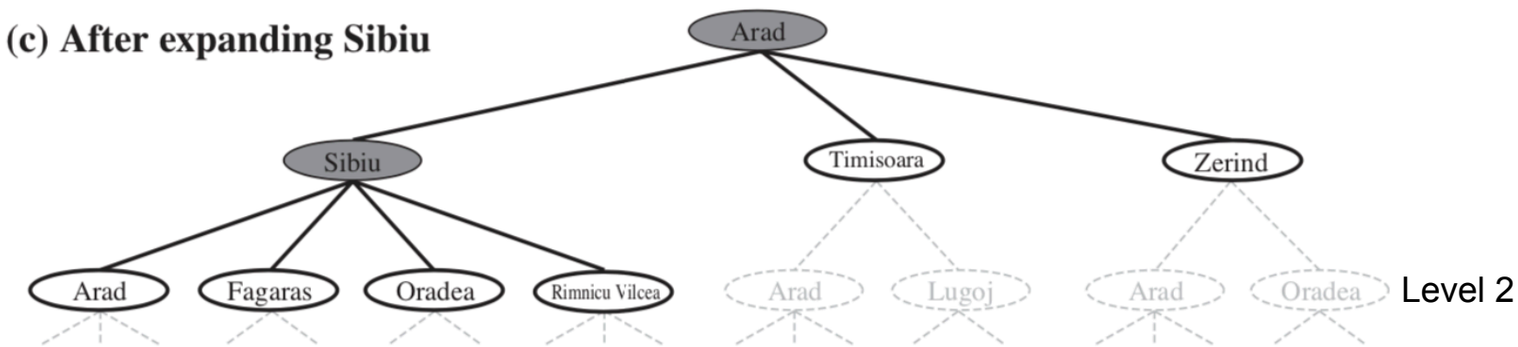
Level 0





Level 1

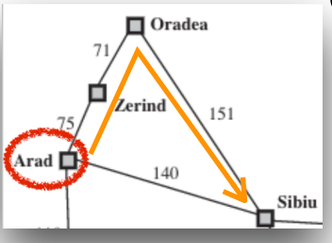




Level 2

Loopy Paths & Redundancies

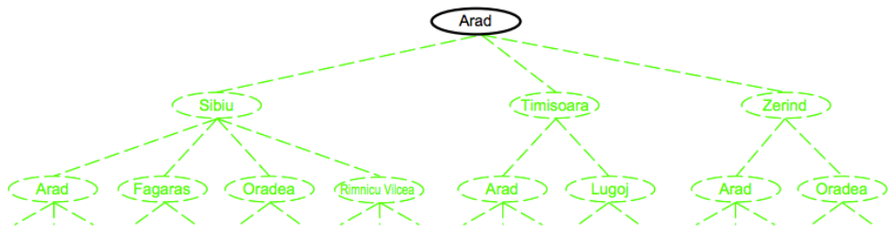
- Search tree includes the path from Arad to Sibiu to Arad again (tree search)
- Such paths can cause infinite loops, because you can keep going back to the repeated state, i.e. *In(Arad)*
- Loopy paths are redundant because it would just add to the path cost



- Redundant paths are those less optimal paths when there is more than one way to get from one state to another
  - Arad–Sibiu *vs.* Arad–Zerind–Oradia–Sibiu (second is redundant)
- In some cases, the problem is defined in such a way that redundant paths are removed (i.e. a graph search)
  - 8-queens: Instead of placing queen on any column, place it on the leftmost empty column

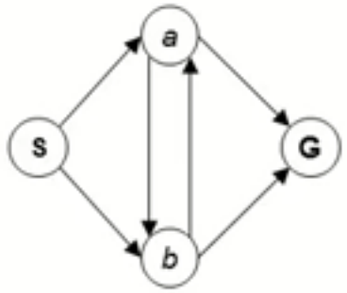
Tree and Graph Search Algorithms

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  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
    expand the chosen node, adding the resulting nodes to the frontier



---

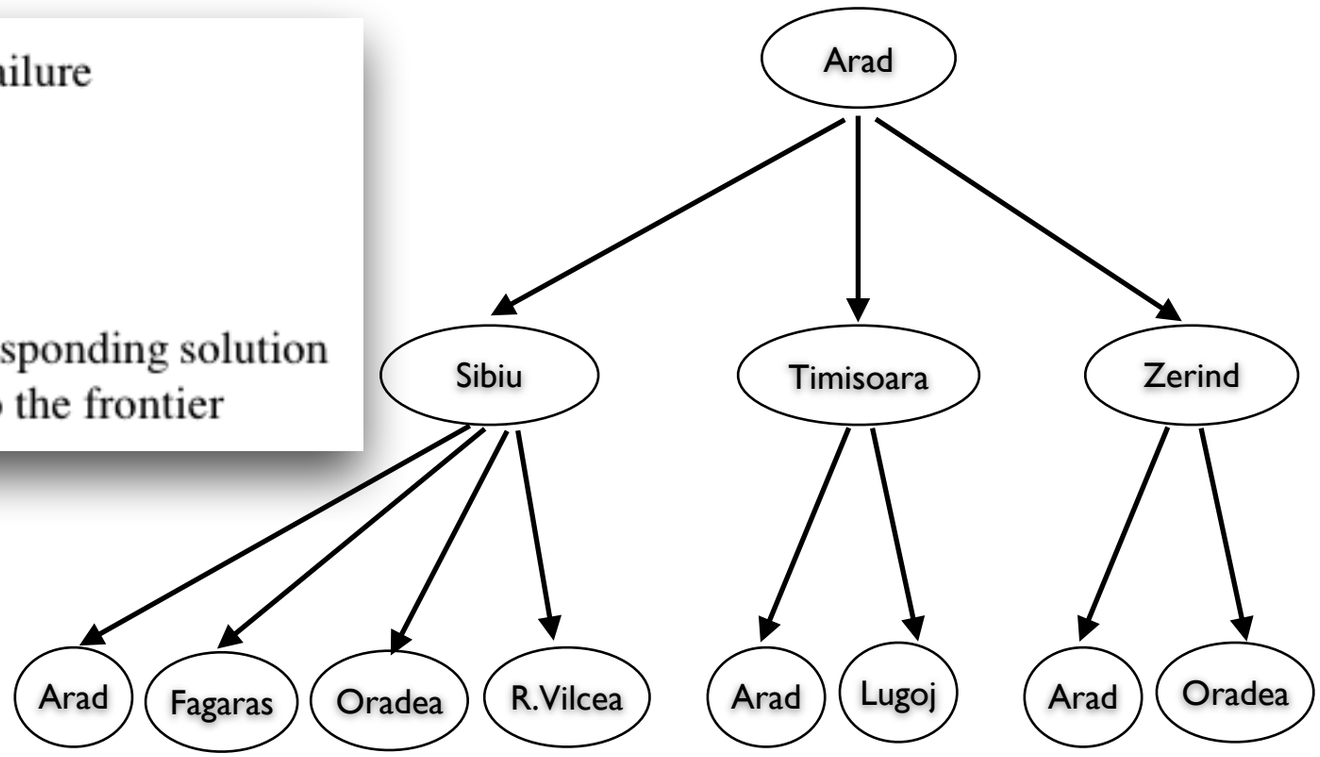
function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the 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



- Highlighted text in GRAPH-SEARCH avoids repeated states and redundant paths by *not* adding any newly generated nodes that match previously generated nodes (contained in explored list). Therefore, contains at most one copy of each state. Most algorithms that follow in this section will adopt this approach.

Tree Search Algorithm

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  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
    expand the chosen node, adding the resulting nodes to the frontier

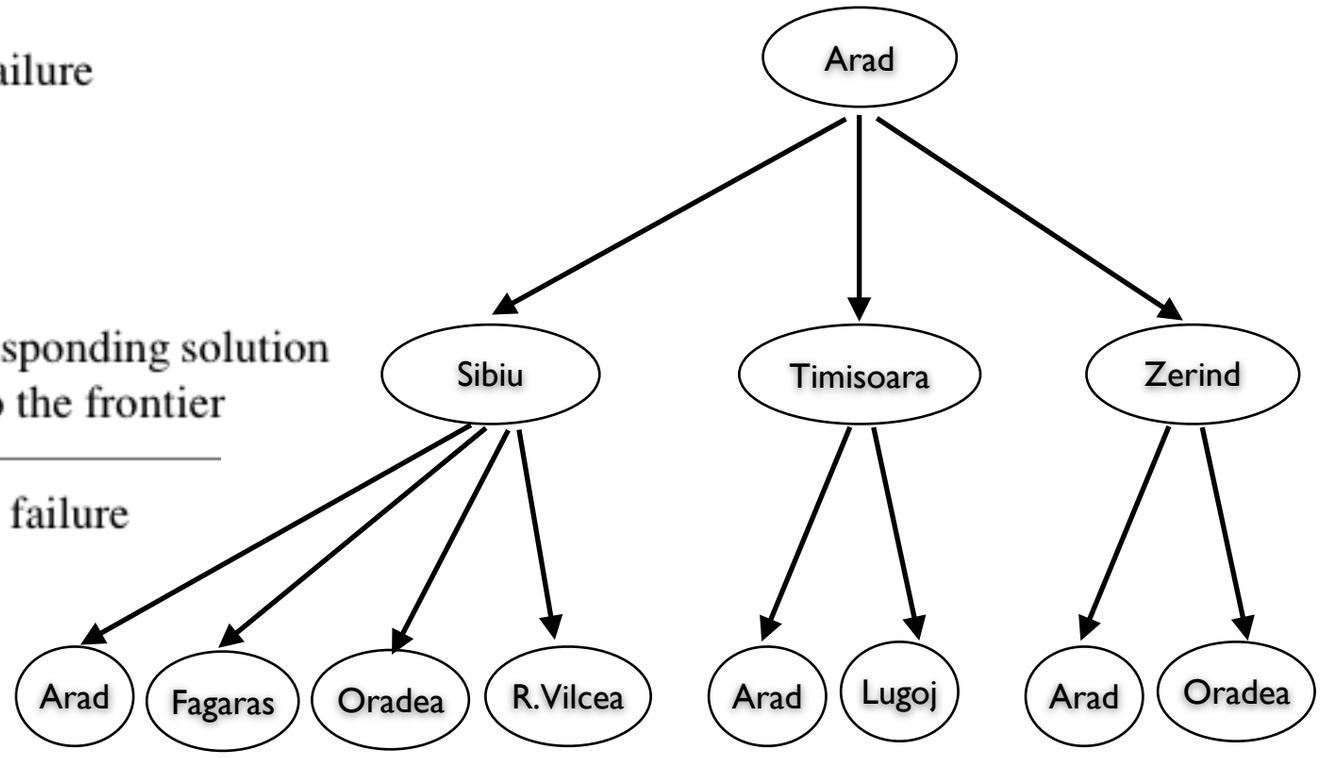


Graph Search Algorithm

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  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
    expand the chosen node, adding the resulting nodes to the frontier

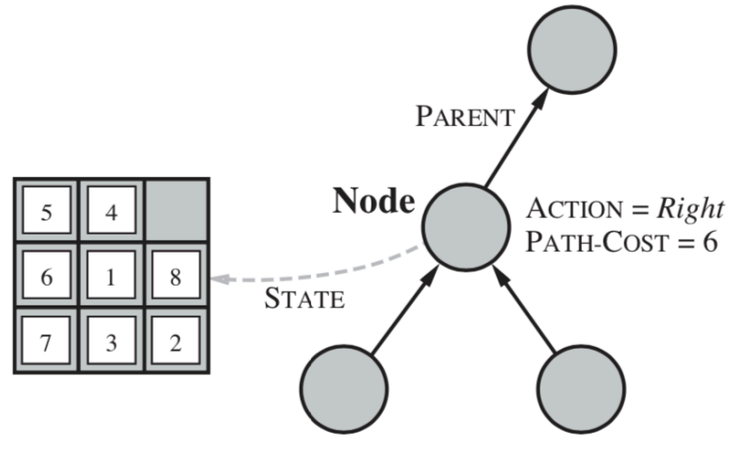
---

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



- Highlighted text in GRAPH-SEARCH avoids repeated states and redundant paths by *not* adding any newly generated nodes that match previously generated nodes (contained in explored list). Therefore, contains at most one copy of each state. Most algorithms that follow in this section will adopt this approach.

Search algorithm's data structure



- A node *n* has the following components:
  - State: the state in the state space to which the node corresponds;
  - Parent: the node in the search tree that generated this node;
  - Action: the action which was applied to the parent to generate the node;
  - Path-Cost: the cost, *g(n)* of the path from the initial state to the node
- Node vs. state: a node is a bookkeeping data structure used to represent the search tree while a state corresponds to a configuration (snapshot) of the world
- A path in a state space is a sequence of states connected by a sequence of actions

Algorithm's Performance Evaluation

- Completeness – is the algorithm guaranteed to find a solution when there is one?
- 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
  - *b* branching factor of the search tree or the maximum number of successors/children of any node.
  - *d* depth of the least cost solution; number of steps from the root (shallowest goal node)
  - *m* maximum depth of the state space (maximum length of any path in the state space, may be infinite)
- *Time* is often measured in terms of the number of nodes *generated* during the search and *space* in terms of the maximum number of nodes *stored* in memory (size of frontier at any given time)

Uninformed Search Strategies

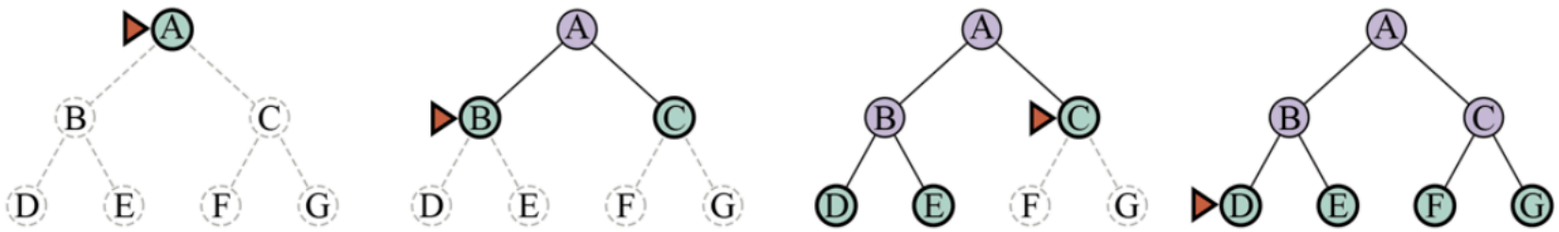
Also called blind search strategies because they only use the information available in the problem definition

All search strategies are distinguished by the *order* in which the nodes are expanded

- Breadth-first search (BFS)
- Uniform-cost search (UCS)
- Depth-first search (DFS)
- Depth-limited search (DLS)
- Iterative deepening search (IDS)

Breadth-first Search

- Shallowest unexpanded node is selected for expansion
- Implementation: frontier is a FIFO (first-in-first-out) queue, i.e. new successors go to the end of the queue and old nodes, which are shallower than the new nodes, get expanded first
- Frontier/fringe: set of nodes being considered for expansion; border of all the nodes at the end of each path in a search tree
- Returns shallowest solution (path to goal with minimal number of steps from the start state) if there are more than 1



depth 0 = 1 node

depth 1 = b nodes

depth 2 = b^2 nodes

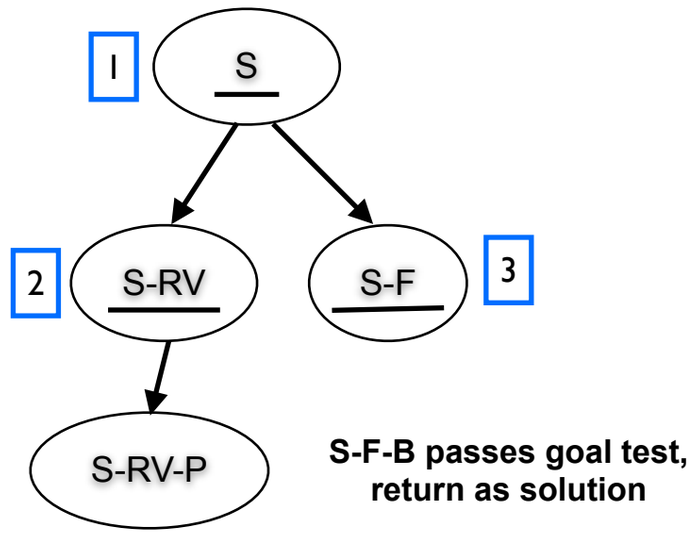
...

depth m = b^m nodes

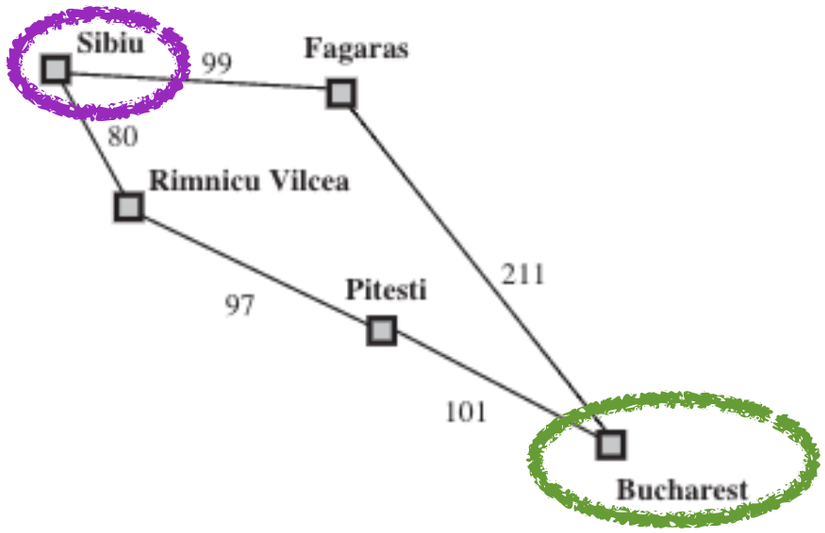
BFS Method 1 – Search Tree

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
frontier ← a FIFO queue with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*) /* chooses the shallowest node in *frontier* */
  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**
      if *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)
      frontier ← INSERT(*child*, *frontier*)



S-F-B passes goal test, return as solution



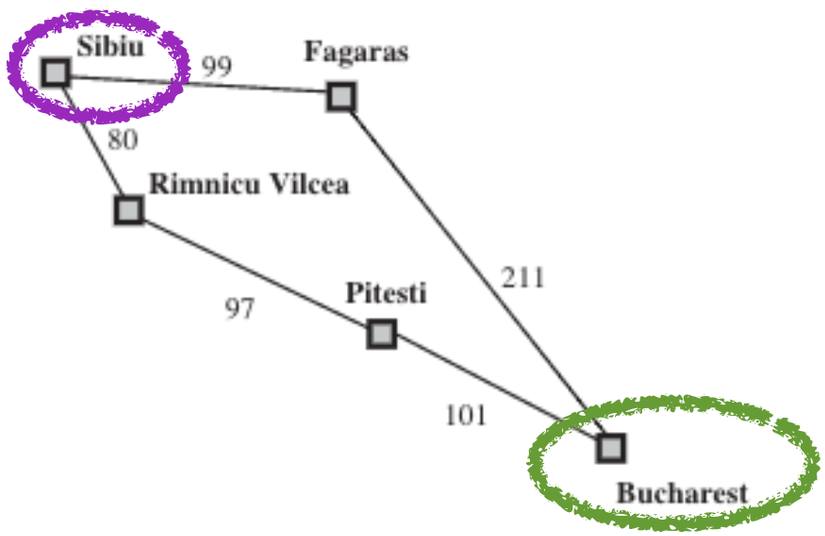
- Node selected for expansion is underlined
- Order of nodes selected for expansion is also indicated by numbers
- Goal test is applied to each node when it is generated rather than when it is selected for expansion (i.e. when it is in frontier)
- Discards any new path to a state already in the frontier or explored set

BFS Method 2 – List

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        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
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Frontier (Queue)	***Explored***
[S]	[ ]
[S-RV, S-F]	[S]
[S-F, S-RV-P]	[S, RV]
[S-RV-P]	[S, RV, F]

S-F-B passes goal test, return as solution



- Node selected for expansion is underlined
- Numbering not required because list is already in order
- Goal test is applied to each node when it is generated rather than when it is selected for expansion (i.e. when it is in the frontier)
- Discards any new path to a state already in the frontier or explored set
- Frontier and Explored lists are shown after the last node is popped

BFS Steps – Simplified

1. If Start node is goal the RETURN it as the solution (STOP)
2. Otherwise
   2.1 Initialise frontier to be a (first-in-first-out) queue with the Start node in it, frontier = [S]
   2.2 Initialise explored list to empty [ ]
3. While frontier is not empty do the following (LOOP)
   3.1 Remove the first node from the frontier
   3.2 Add this node to explored
   3.3 Look at this node's children one by one and do the following (LOOP). When done with all its children or if it has no children, go back to 3
      3.3.1 If the child is not in explored or frontier, check further if it is a goal. If it is, RETURN it as the Solution (STOP)
      3.3.2 Otherwise if child is not a goal, put the child in the frontier (Do nothing if the child is in explored or frontier, move on to the next child or to 3 if no more children left)
4. If frontier is empty (no more nodes to process) RETURN Failure because there is no solution (STOP)

BFS: Properties

- Complete? Yes (if b is finite)
- Optimal? Yes (if cost is equal for every step); not optimal in general
- Time? $b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- Space? $O(b^d)$ (keeps every node in memory)
- ✱ Space is the big problem. Anything in exponential complexity is scary – can generate 10MB nodes per second which means 860GB per day! Time is also an issue, although we can wait a few days, we may not want to wait a few years for a solution

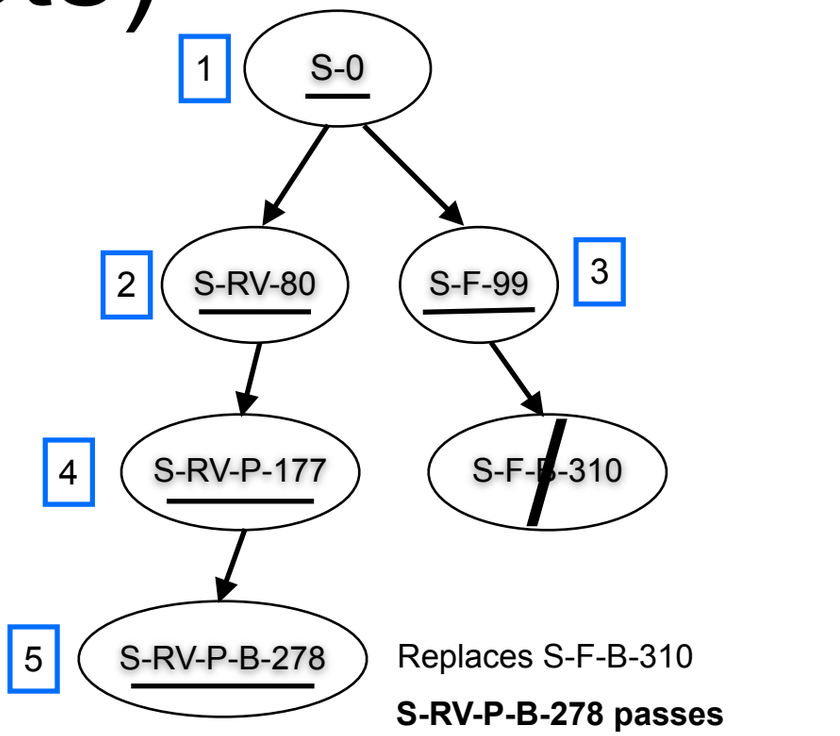
Uniform-cost Search (Djikstra's algorithm)

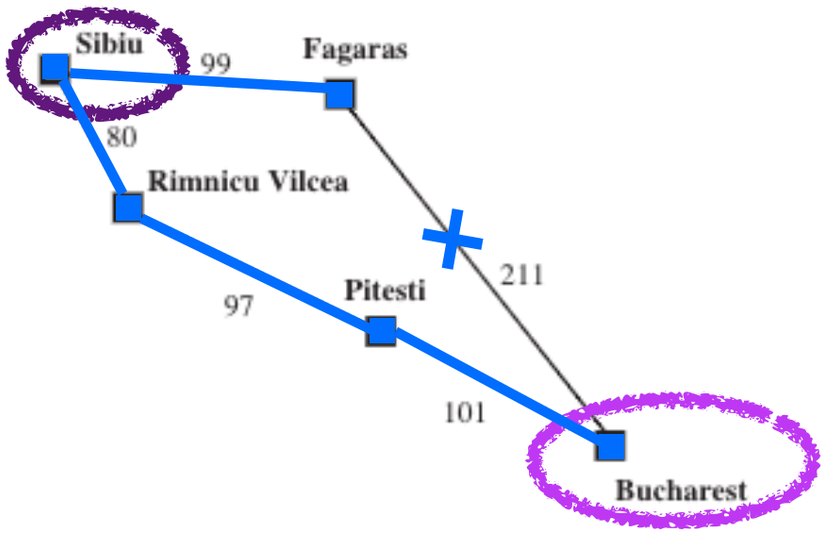
- Least-cost unexpanded node is selected for expansion, i.e. with the lowest path cost, *g(n)*
- Implementation: frontier is a (priority) queue ordered by path cost, lowest first
- Equivalent to BFS if step costs are all equal
- Does not care about the *number* of steps a path has, but only about their total cost
- Versus BFS:
  - In UCS, goal test is applied to each node when it is selected for expansion rather than when it is first generated – why? Because the first goal node that is generated may be suboptimal
  - A test is added in case a better path is found to a node currently on the frontier

UCS Working (Search Tree & Lists)

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*





Frontier	*Explored*	*Remarks*
[S-0]	[ ]	
[S-RV-80, S-F-99]	[S]	
[S-F-99, S-RV-P-177]	[S, RV]	
[S-RV-P-177, S-F-B-310]	[S, RV, F]	
[S-RV-P-B-278]	[S, RV, F, P]	S-RV-P-B-278 replaces S-F-B-310
[ ]	[S, RV, F, P]	

S-RV-P-B-278 passes goal test
Solution: **S-RV-P-B-278**

UCS Steps – Simplified

1. Initialise frontier to be a priority queue with the Start node in it, frontier = [S]
2. Intialise explored list to empty [ ]
3. While frontier is not empty do the following (LOOP)
   - 3.1 Remove the cheapest cost node from the frontier
   - 3.2 Check if the node is a goal, if so RETURN the solution (STOP)
   - 3.3 Otherwise, add this node to explored
   - 3.4 Look at this node's children one by one and do the following (LOOP). When done with all children or it doesn't have any children, go back to 3
     - 3.4.1 If the child is not in explored or frontier, add it to frontier
     - 3.4.2 Otherwise if the child is in frontier with a higher cost, replace that child with this child (with the lower cost)
4. If frontier is empty (no more nodes to process) RETURN Failure because there is no solution (STOP)

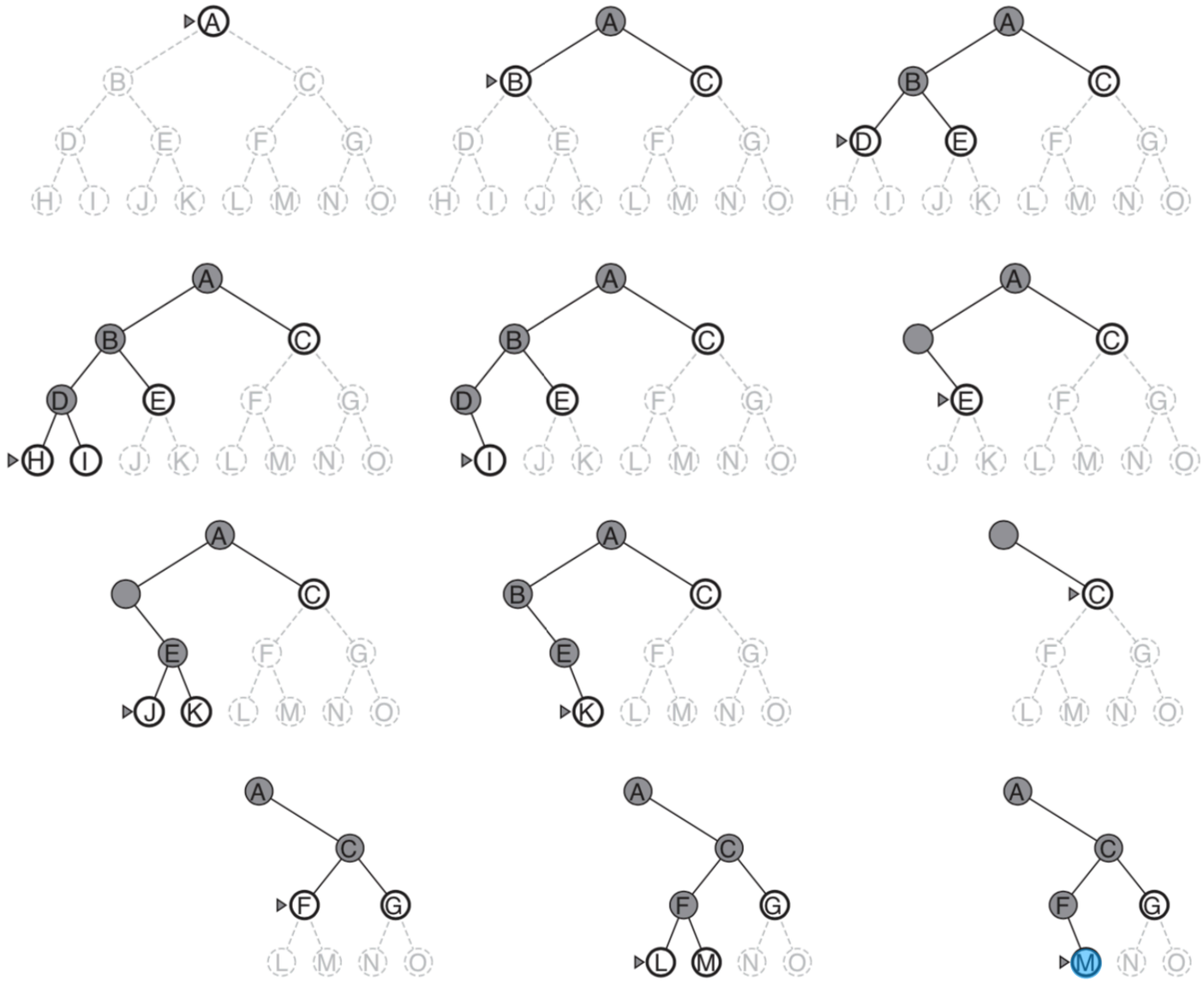
UCS: Properties

- Complete? Yes (if step cost ≥ ε, minimum action cost)
- Optimal? Yes (nodes expanded in in increasing order of *g(n)*
- Time? #nodes with *g*≤cost of optimal solution, $O(b^{1+[C^*/\varepsilon]})$ where C* is the cost of the optimal (cheapest) solution
- Space? $O(b^{1+[C^*/\varepsilon]})$ (keeps every node in memory)

✱ $O(b^{1+[C^*/\varepsilon]})$ can be much greater than b^d. When all step costs are the same, UCS is similar to BFS except that BFS stops as soon as it generates a goal whereas UCS examines all nodes at the goal's depth to see if one has a lower cost

Depth-first Search

- Deepest unexpanded node is selected for expansion
- Implementation: frontier is a LIFO (last-in-first-out) queue, also known as a stack, i.e. new successors go to the front
- Properties of DFS depend strongly on whether the graph-search or tree-search version is used
- Graph-search version is complete and finite because it will eventually expand every node
- Tree-search version is *not* complete, e.g. Arad–Sibiu–Arad loop forever
- Both versions are non optimal (see next page)



- Unexplored regions are shown in light grey
- Explored nodes with no descendants in the frontier are removed from memory
- *M* is the only goal node
- Non-optimal case: if *C* and *J* were goal nodes, DFS would return *J* instead of *C*

DFS Recursive Algorithm

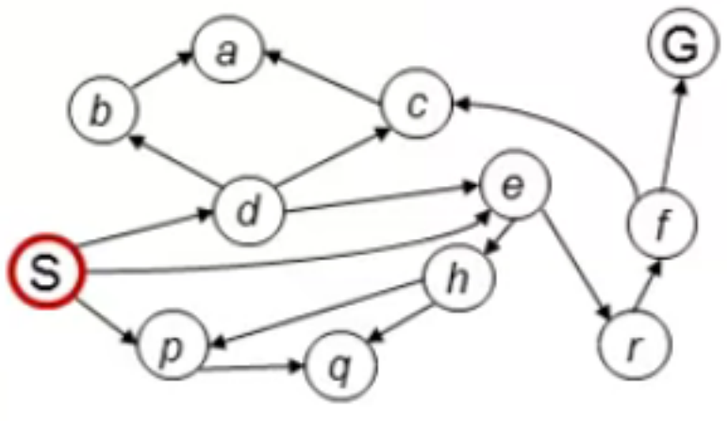
```
dfs(n):

For each child of n:

    If ParentOf(child) is None: #child not in
    explored:
```

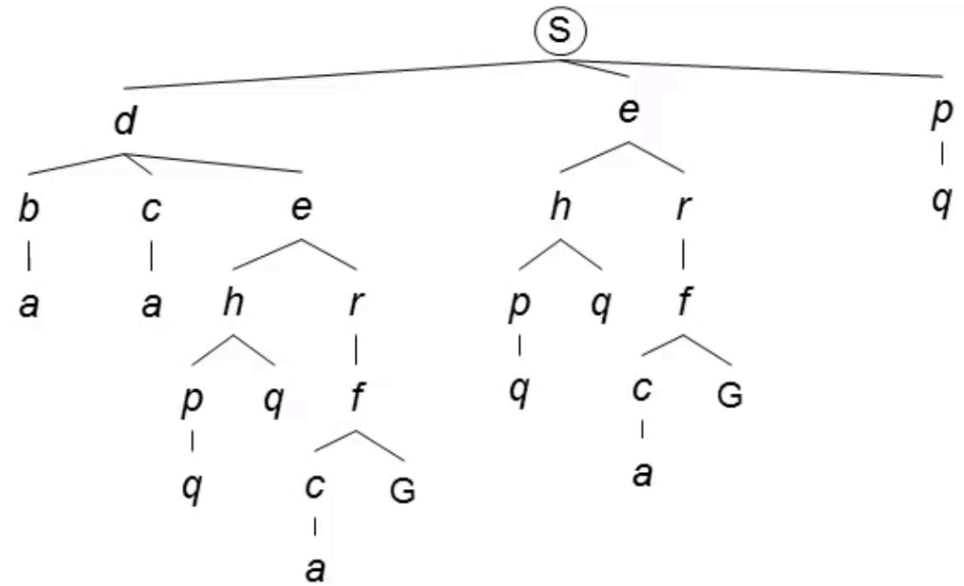
- `ParentOf(child) = n #put child in explored`
- `If child passes Goal-test`

  ➡ `RETURN solution`

- `dfs(child)`

DFS & BFS Exercise





- Draw the search tree for the state graph above (tree search style) – DONE
- Perform DFS (tree search) to find a solution from S to G, taking alphabetical ordering when two or more nodes are tied
- Number the nodes 1,2,3 to show ordering when traversing the search tree with DFS
- Extra: Repeat with BFS (tree search)

DFS: Properties

- Complete? No (fails in infinite depth spaces, or spaces with loops in tree-search, modify to avoid repeated spaces along path in graph-search)
  - DFS is complete if the search tree is finite
- Optimal? No (in this example, finds the “leftmost” solution, regardless of cost and depth)
- Time? $O(b^m)$ where m is the maximum depth of any node. Terrible if m is much larger than d (depth of the shallowest solution)
- Space? $O(bm)$ – linear!
- ✱ **Space** is a big advantage. Depth first tree search needs to store only a single path from root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. E.g. 156KB (DFS) vs. 10 exabyte (BFS) at depth d = 16, a factor of 7 trillion times less space

Depth-limited Search

- DFS with depth limit ℓ, i.e nodes at depth ℓ have no successors, do not look beyond this depth
- Solves the infinite path problem that DFS has but incomplete if $\ell < d$ (shallowest goal is beyond the depth limit)
- Non-optimal if $\ell > d$, but usually d is not known
- E.g. for Romania – there are 20 cities, if there is a solution its length will be 19 at the longest, so ℓ = 19 is a possible choice. Finding a good depth limit leads to a more efficient DLS

DLS: Pseudocode

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

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
  if cutoff_occurred? then return cutoff else return failure
```

- Recursive, i.e. calls itself
- Terminates with two kinds of failures
  - No solution found (*standard*)
  - *Cutoff* value indicates no solution found within the depth limit
- Time complexity: $O(b^{\ell})$
- Space complexity: $O(b\ell)$

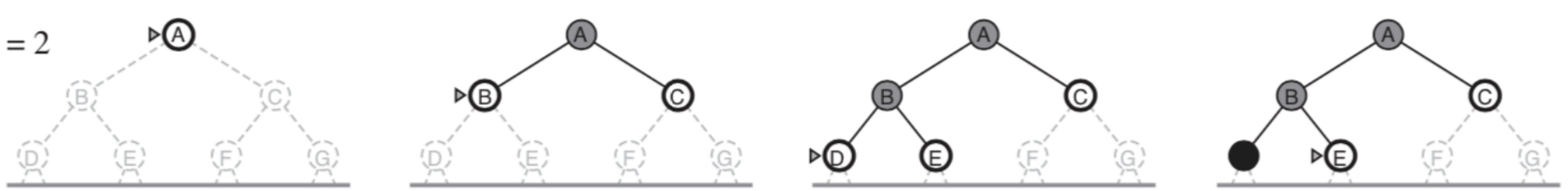
Iterative-deepening Search (IDS)

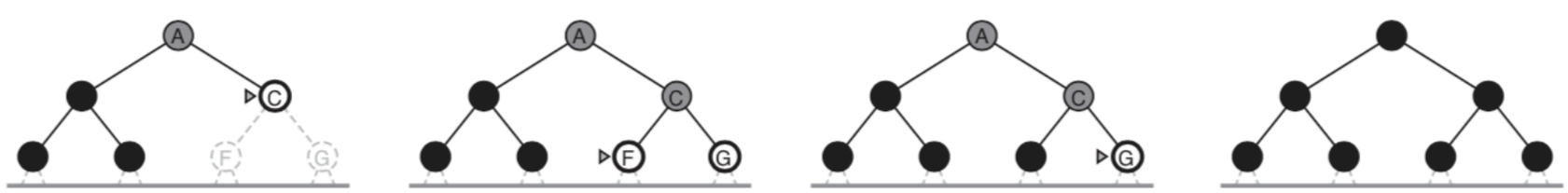
- Iterative-deepening is DLS search run repeatedly to find the best depth limit
- How? By gradually increasing the depth limit from 0, then 1, then 2 and so on until a goal is found
- Goal will be found at *d*, depth of shallowest goal
- Combines the benefits of DFS and BFS
  - like DFS its memory requirements are good: *O(bd)*
  - like BFS it's complete when *b* is finite and optimal when path cost is a nondecreasing function of the depth of the node

Limit = 0

Limit = 1

Limit = 2

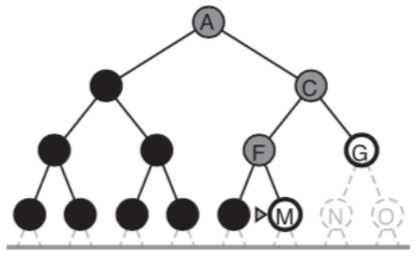




Limit = 3

IDS: Pseudocode

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



- May seem wasteful because states are generated multiple times – but it's not too costly as most of the nodes are at the bottom so multiple generations of upper level nodes are relatively cheap

- Nodes at bottom level (depth *d*) are generated once, the nodes above the bottom level twice, and so on until the children of the root which are generated *d* times

- Terminates when
  - A solution is found
  - No solution found (*standard*)
  - *Cutoff* value indicates no solution found within the depth limit

IDS: Properties

- Complete? Yes
- Optimal? Yes, if step cost = 1
- Time? $(d)b + (d-1)b^2 + ... + (1)b^d = O(b^d)$
  - Same as BFS, with a small extra cost for the multiple generations of upper level nodes, e.g. for b = 10, d = 5 and solution is at far right of tree, *N(IDS)* = 123,450 and *N(BFS)* = 111,110
- Space? $O(bd)$
  - Same as DFS, as it is doing DFS at every iteration

* In general, IDS is the preferred uninformed search method when the search space is large and the depth of the solution is unknown

Comparison of Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

b branching factor

d depth of the shallowest goal

m maximum depth of tree

ℓ is the depth limit

[1] complete if *b* is finite

[2] complete if step costs $\geq \epsilon \geq 0$

[3] optimal if step costs are all identical

[4] if both directions are breadth-first or uniform-cost

Why is Speed Important?

- Speed of computation comes from a number of areas such as computational complexity (of algorithms), vectorisation, using parallel processing (multiple cores) and locality (speed to access data)
- Using slower ways to access data (latency), e.g. over the internet, can cause a huge difference in performance, sometimes a billion times slower!
- Here is a timeline of how speed of data access has changed over the years
- Many modern algorithms especially in the context of big data are optimised in these areas

Summary

- Search algorithm
  - systematically builds the search tree
  - chooses an ordering of the frontier (unexplored nodes)
  - optimal – finds least-cost plans according to its strategy
- Tree search can cause redundancies and inifite loops
- Graph search avoids repetition of states already seen ∴ more efficient than tree search
- Uninformed search strategies are blind in that they only use the information available in the problem definition
- BFS, UCS, DFS, DLS, IDS – traversals on a search tree
- IDS uses only linear space and not much more time than other uninformed searches

References

- Russel and Norvig, Chapter 3.3–3.4.5
- Basics of nodes
- Computational complexity [Big O] [Video]
- DFS vs BFS vs UCS Pacman maze demo [Link]
- UCS animation [Link]
- UCS explanation by John Levine [Link]
- DFS BFS explanation using Stacks, Queues and Recursion in Korean (first 8 minutes) [Link]