Grundlagen der künstlichen Intelligenz – Uninformed Search

Matthias Althoff

TU München

October 24, 2019

Organization

The content is covered in the AI book by the section "Solving Problems by Searching", Sec. 1-4.

Learning Outcomes

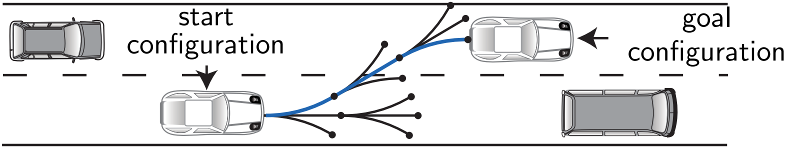
- You can create formally defined search problems.
- You understand the complexity of search problems.
- You understand how real world problems can often be posed as a pure search problem.
- You understand the difference between tree search and graph search.
- You can apply the most important uninformed search techniques: Breadth-First Search, Uniform-Cost Search, Depth-First Search, Depth-Limited Search, Iterative Deepening Search.
- You can compare the advantages and disadvantages of uninformed search strategies.

Motivation

graph-based

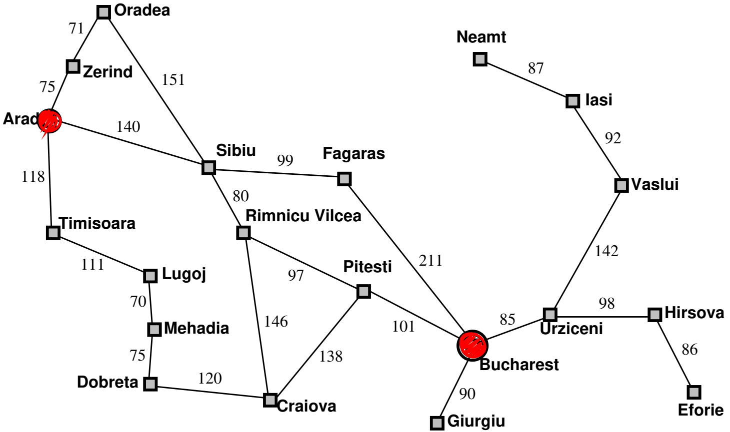
One example how search is used in my research group:

- Automated vehicles have to search a collision-free path from a start to a goal configuration.
- Searching in continuous space is difficult.
- We discretize the search problem in space and time by offering only a finite number of possible actions at discrete time steps.
- This makes it possible to use classical search techniques as introduced in this lecture.



Another Example: Holiday in Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest.

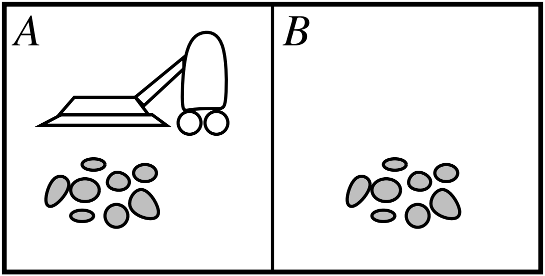


Well-defined problems

A problem can be formally defined by 5 components:

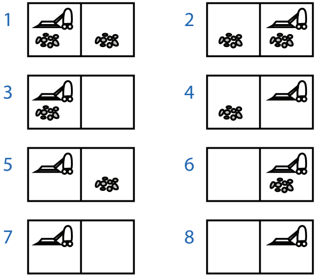
- The **initial state** the agent starts in.
  Example: initial state is `In(Arad)`.
- A description of the possible **actions**.
  Example: actions of `In(Arad)` are `Go(Sibiu)`, `Go(Timisoara)`, `Go(Zerind)`.
- A description of what each action does, which we refer to as the **transition model**.
  Example: `RESULT(In(Arad), Go(Zerind)) = In(Zerind)`.
- The **goal test**, which checks whether a given state is the goal state.
  Example: the goal state is `In(Bucharest)`.
- A **path cost** function assigning a numeric cost to each solution path.
  Example: traveled distance is a good path cost.

Vacuum-Cleaner World



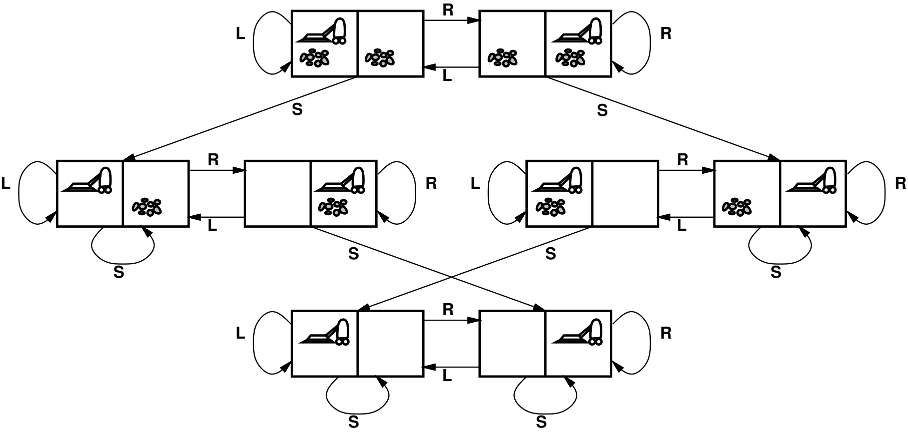
Percepts: location and contents, e.g., $[A, \mathit{Dirty}]$
Actions: *Left*, *Right*, *Suck*, *NoOp* (**No Op**eration)

Vacuum World (Toy Problem I)



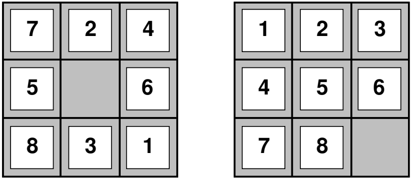
- **States**: Combination of cleaner and dirt locations: $2 \cdot 2^2 = 8$ states.
- **Initial state**: any state.
- **Actions**: `Left`, `Right`, and `Suck`.
- **Transition model**: see next slide.
- **Goal test**: checks whether all locations are clean.
- **Path cost**: Each step costs 1.

Vacuum World: Transition Model



The transition model can be stored as a directed graph, just like the Holiday-in-Romania-Problem. This is possible for all discrete problems.

8-Puzzle (Toy Problem II)

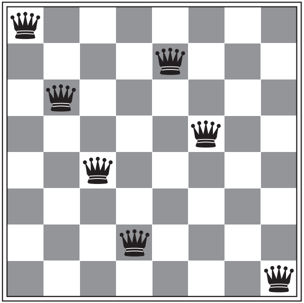


Start State   Goal State

Tiles can move to the blank space. How to reach the goal state?

- **States**: Specify the location of each tile and the blank space: $9!/2 = 181440$ states (only half of possible initial states can be moved to the goal state; see Moodle attachment).
- **Initial state**: Any state.
- **Actions**: Movement of the blank space: `Left`, `Right`, `Up`, and `Down`.
- **Transition model**: Huge, but trivial e.g., if `Left` applied to start state: '5' and 'blank' are switched.
- **Goal test**: Checks whether the goal configuration is reached.
- **Path cost**: Each step costs 1.

8-Queens Problem (Toy Problem III)



- Place 8 queens on a chessboard such that no queens attack each other (A queen attacks any piece in the same row, column or diagonal).
- Is the above figure a feasible solution?
- Two formulations:
  - **Incremental formulation**: Start with an empty chessboard and add a queen at a time.
  - **Complete-state formulation**: Start with 8 queens and move them.

8-Queens Problem Description

We try the following **incremental formulation**:

- **States**: Any arrangement of $0 - 8$ queens:
  $64 \cdot 63 \cdot \ldots \cdot 57 \approx 1.8 \cdot 10^{14}$ states.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens on the board, none attacked.
- **Path cost**: Does not apply.

Improvement to reduce complexity: Do not place a queen on a square that is already attacked.

- **States***: Any arrangement of 0-8 queens with no queens attacking each other in the n leftmost columns (now only 2057 states).
- **Actions***: Add a queen to any empty square in the leftmost empty column such that it is not attacked by any other queen.

Examples of Real-World Problems

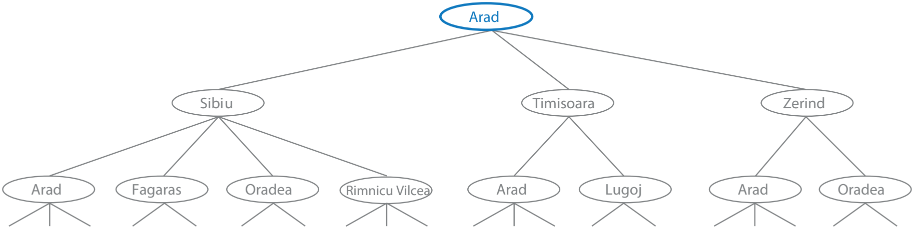
- **Route-Finding problem**: Airline travel planning, video streams in computer networks, etc.
- **Touring problem**: How to best visit a number of places, e.g., in the map of Romania?
- **Layout of digital circuits**: How to best place components and their connections on a circuit board?
- **Robot navigation**: Similar to the route-finding problem, but in a continuous space.
- **Automatic assembly sequencing**: In which order should a product be assembled?
- **Protein design**: What sequence of amino acids will fold into a three-dimensional protein?

Generating a Search Tree (1)

We are searching for an action sequence to a goal state. The possible actions from the initial state form the **search tree**:

- **Root**: Initial state.
- **Branches**: Actions.
- **Nodes**: Reached states.
- **Leafs**: Unexpanded nodes.

A search tree is expanded by applying all possible actions to each parent node. Example:

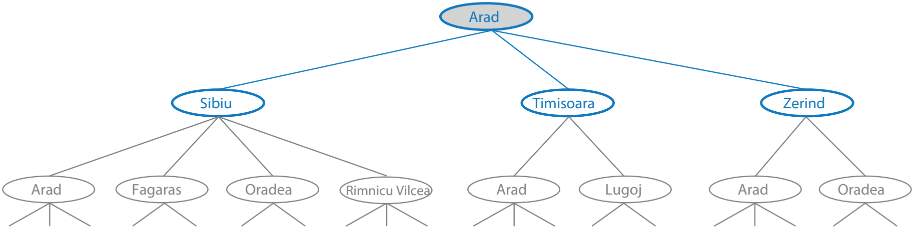


Generating a Search Tree (2)

We are searching for an action sequence to a goal state. The possible actions from the initial state form the **search tree**:

- **Root**: Initial state.
- **Branches**: Actions.
- **Nodes**: Reached states.
- **Leafs**: Unexpanded nodes.

A search tree is expanded by applying all possible actions to each parent node. Example:

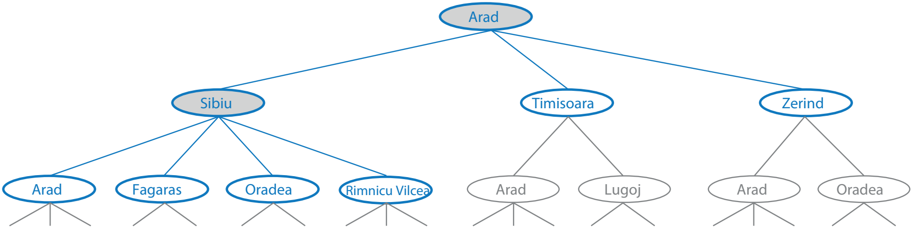


Generating a Search Tree (3)

We are searching for an action sequence to a goal state. The possible actions from the initial state form the **search tree**:

- **Root**: Initial state.
- **Branches**: Actions.
- **Nodes**: Reached states.
- **Leafs**: Unexpanded nodes.

A search tree is expanded by applying all possible actions to each parent node. Example:



Tree Search Algorithm

1

→ Blind Search e.g Breadth-first & Depth-first

→ Best first Search e.g Greedy & A*

The basic principle of expanding leafs until a goal is found can be implemented by the subsequent pseudo code.

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

Tweedback Questions

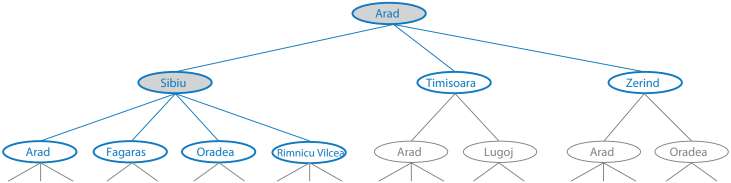
Graph Search : store where you have been

Tree Search : do not store where you have visited

- Does tree search always find a solution if one exists? NO
- Is the search tree of a finite graph also finite? NO
- Again: Does tree search always find a solution if one exists? NO

Avoiding Loops in the Search Tree

- The set of leafs is now referred to as the **frontier** or **open list**.
- In the previous example, we went back to Arad from Sibiu: Expanding from Arad only contains repetitions of previous possibilities.



- **Solution**: Only expand nodes that have not been visited before. Visited states are stored in an **explored set** or **closed list**.
- This idea is referred to as graph search (see next slide).

Graph Search Algorithm

2

Differences to the tree search are highlighted in green.

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

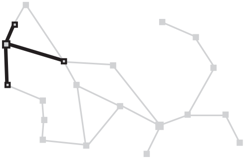
    choose a leaf node and remove it from the frontier

    if the node contains a goal state **then return** the corresponding solution

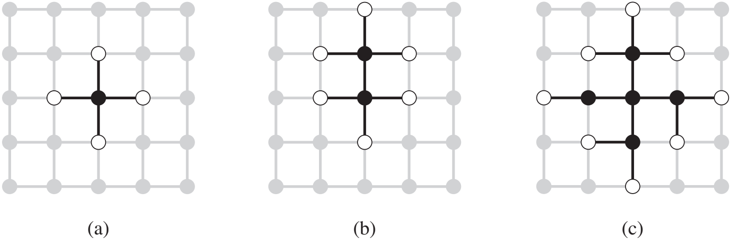
    add the node to the explored set

    expand the chosen node, adding the resulting nodes to the frontier only if not in frontier or explored set

Graph Search Algorithm: Illustrations



A sequence of search trees generated by graph search. The northernmost city (Oradea) has become a dead end; this would not have happened with tree search.



The frontier (white nodes) separates the explored nodes (black nodes) from the unexplored ones (gray nodes). Nodes are not expanded to previously visited ones.

Tweedback Questions

- Is it possible that graph search is slower than tree search? YES
- What is the maximum number of steps required in graph search (according to slide 20)?
  - A The shortest number of edges from the initial state to the goal state.
  - B The number of edges of the graph.
  - C The number of nodes of the graph minus one.

initial state

3 common variants for queue

FIFO → pops the oldest element

LIFO → pops the newest element

priority → pops the highest priority element

Measuring Problem-Solving Performance

We can evaluate the performance of a search algorithm using the following criteria:

- **Completeness**: Is it guaranteed that the algorithm finds a solution if one exists?
- **Optimality**: Does the strategy find the optimal solution (minimum costs)?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

Infrastructure for Search Algorithms

Structure of a node n

- n.STATE: The state in the state space to which the node corresponds;
- n.PARENT: The node in the search tree that generated this node;
- n.ACTION: The action that was applied to the parent to generate the node;
- n.PATH-COST: The cost, traditionally denoted by $g(n)$, of the path from the initial state to the node.

Operations on a queue (list of elements)

- Empty(queue): Returns true if queue is empty;
- Pop(queue): Removes the first element of the queue and returns it;
- Insert(element, queue): Inserts an element and returns the resulting queue.

Uninformed Search vs. Informed Search

Uninformed search

- No additional information besides the problem statement (states, initial state, actions, transition model, goal test) is provided.
- Uninformed search can only produce next states and check whether it is a goal state.

Informed search

additional information is provided

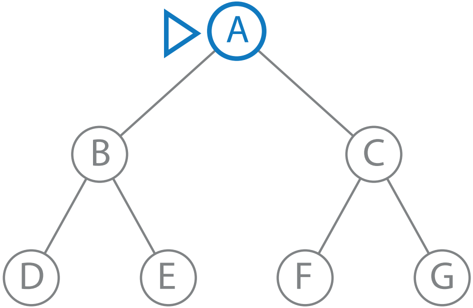
- Strategies know whether a state is more promising than another to reach a goal.
- Informed search uses measures to indicate the distance to a goal.

Breadth-First Search: Idea (1)

Shallowest nodes first

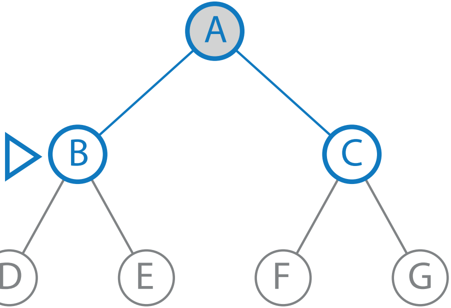
Special instance of the graph-search algorithm (slide 20): All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded:

deeper



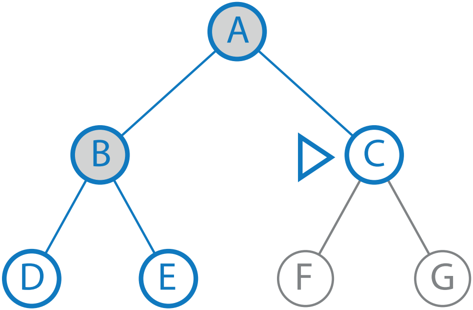
Breadth-First Search: Idea (2)

Special instance of the graph-search algorithm (slide 20): All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded:



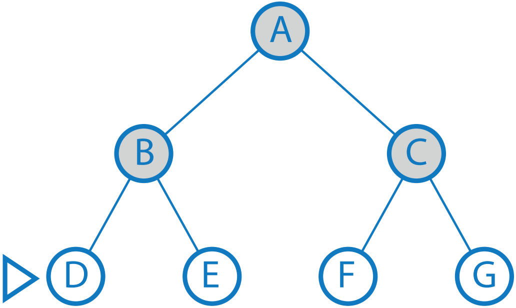
Breadth-First Search: Idea (3)

Special instance of the graph-search algorithm (slide 20): All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded:



Breadth-First Search: Idea (4)

Special instance of the graph-search algorithm (slide 20): All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded:



Breadth-First Search: Algorithm (UninformedSearch.ipynb)

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 neither in *explored* nor *frontier* **then**
            if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
            frontier ← `Insert`(*child*,*frontier*)

Auxiliary Algorithm: Child-Node

function Child-Node (problem, parent, action) **returns** a node

return a node with

State $=$ *problem*.Result(*parent*.State, *action*)

Parent $=$ *parent*

Action $=$ *action*

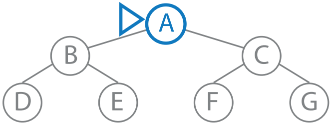
Path-Cost $=$ *parent*.Path-Cost

$+$ *problem*.Step-Cost(*parent*.State, *action*)

Breadth-First Search: Algorithm (Step 1a)

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 neither in *explored* nor *frontier* **then**
            if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
            frontier ← `Insert`(*child*,*frontier*)

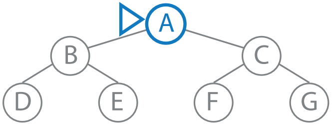


goal: F
node: A
frontier: A
explored: ∅

Breadth-First Search: Algorithm (Step 1b)

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 neither in *explored* nor *frontier* **then**
      if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
      frontier ← `Insert`(*child*,*frontier*)

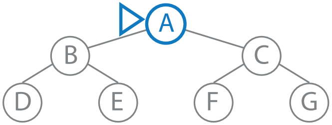


goal: F
node: A
frontier: ∅
explored: ∅

Breadth-First Search: Algorithm (Step 1c)

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 neither in *explored* nor *frontier* **then**
            if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
            frontier ← `Insert`(*child*,*frontier*)

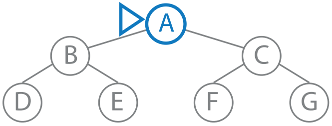


goal: F
node: A
frontier: ∅
explored: A

Breadth-First Search: Algorithm (Step 1d)

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 neither in *explored* nor *frontier* **then**
            if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
            frontier ← `Insert`(*child*,*frontier*)

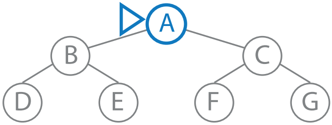


goal: F
node: A
frontier: B
explored: A

Breadth-First Search: Algorithm (Step 1e)

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 neither in *explored* nor *frontier* **then**
      if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
      frontier ← `Insert`(*child*,*frontier*)

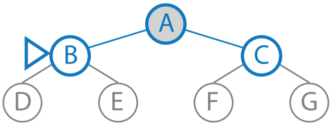


goal: F
node: A
frontier: B, C
explored: A

Breadth-First Search: Algorithm (Step 2a)

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 neither in *explored* nor *frontier* **then**
      if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
      frontier ← `Insert`(*child*,*frontier*)

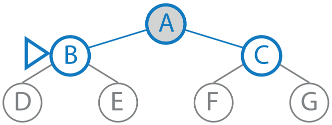


goal: F
node: B
frontier: C
explored: A

Breadth-First Search: Algorithm (Step 2b)

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 neither in *explored* nor *frontier* **then**
            if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
            frontier ← `Insert`(*child*,*frontier*)

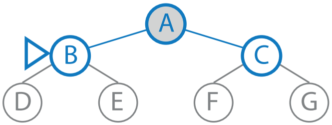


goal: F
node: B
frontier: C
explored: A, B

Breadth-First Search: Algorithm (Step 2c)

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 neither in *explored* nor *frontier* **then**
            if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
            frontier ← `Insert`(*child*,*frontier*)

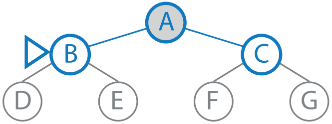


goal: F
node: B
frontier: C, D
explored: A, B

Breadth-First Search: Algorithm (Step 2d)

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 neither in explored nor frontier then
            if problem.Goal-Test(child.State) then return Solution(child)
            frontier ← Insert(child,frontier)
```

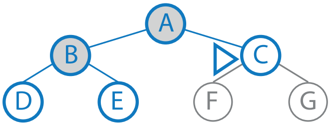


goal: F
node: B
frontier: C, D, E
explored: A, B

Breadth-First Search: Algorithm (Step 3a)

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 neither in *explored* nor *frontier* **then**
            if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
            frontier ← `Insert`(*child*,*frontier*)

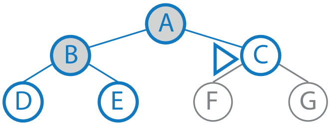


goal: F
node: C
frontier: D, E
explored: A, B

Breadth-First Search: Algorithm (Step 3b)

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 neither in *explored* nor *frontier* **then**
            if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
            frontier ← `Insert`(*child*,*frontier*)

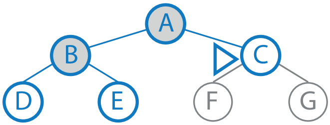


goal: F
node: C
frontier: D, E
explored: A, B, C

Breadth-First Search: Algorithm (Step 3c)

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 neither in *explored* nor *frontier* **then**
      if *problem*.`Goal-Test`(*child*.`State`) **then return** `Solution`(*child*)
      frontier ← `Insert`(*child*,*frontier*)



goal state F found!
node: C
frontier: D, E
explored: A, B, C

We stopped when goal F was generated, not reached

Breadth-First Search: Performance

We introduce think of them as dim

children

- the branching factor **b** (maximum number of successors of any node),
- the depth **d** (depth of the shallowest goal node), distance from start node to goal
- the maximum length **m** of any path in the state space, horizontal level

and use the previously introduced criteria:

- **Completeness**: Yes, if depth d and branching factor b are finite.
- **Optimality**: Yes, if cost is equal per step; not optimal in general.
- **Time complexity**: The worst case is that each node has b successors. The number of explored nodes sums up to

$$b + b^2 + b^3 + \cdots + b^d = \mathcal{O}(b^d)$$

- **Space complexity**: All explored nodes are $\mathcal{O}(b^{d-1})$ and all nodes in the frontier are $\mathcal{O}(b^d)$

Landau Notation aka Big O Notation (for students from other disciplines)

Describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Shall $f(x)$ be the actual function for parameter x, then there exist positive constants M, x_0, such that

$$|f(x)| \leq M|g(x)| \text{ for all } x > x_0.$$

Here, $f(b) = b + b^2 + b^3 + \cdots + b^d$ and $g(b) = b^d$. Possible combinations of M, d_0 for $b = 10$ are:

- $M = 2,\ d_0 = 2,$
- $M = 1.2,\ d_0 = 3,$
- $M = 1.12,\ d_0 = 4.$

Since M, x_0 only have to exist and their concrete values do not matter, we just write $\mathcal{O}(b^d)$.

Tweedback Question

Assume: branching factor is $b = 10$

Up to what depth is a breadth-first-search problem solvable on your laptop?

- A $d = 8$
- B $d = 16$
- C $d = 32$

Breadth-First Search: Complexity Issue

An exponential complexity, such as $\mathcal{O}(b^d)$, is a big problem. The following table lists example time and memory requirements for a branching factor of $b = 10$ on a modern computer:

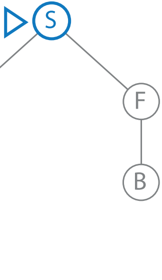
Depth	Nodes	Time	Memory
2	110	0.11 ms	107 kilobytes
4	$11,110$	11 ms	10.6 megabytes
6	10^6	1.1 s	1 gigabyte
8	10^8	2 min	103 gigabytes
10 feasible	10^{10}	3 h	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

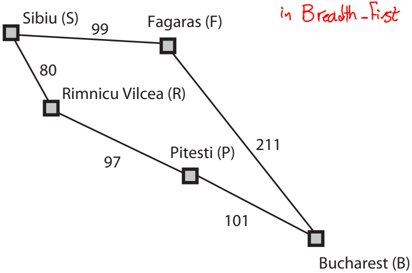
Uniform-Cost Search (aka Dijkstra's algorithm): Idea (1)

2

- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n)$.
- This is realized by storing the frontier as a priority queue ordered by g.

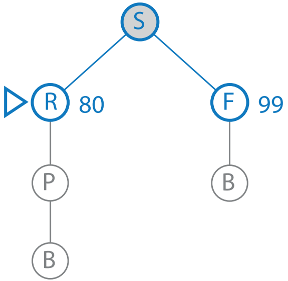
instead of FIFO in Breadth-First

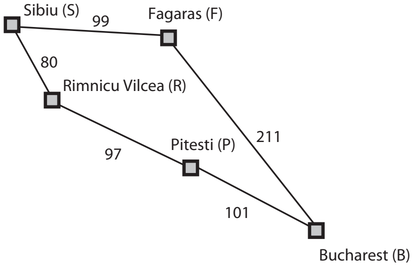




Uniform-Cost Search (aka Dijkstra's algorithm): Idea (2)

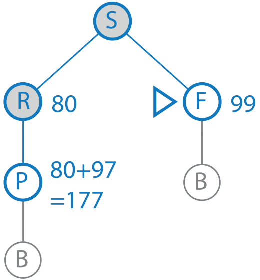
- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n)$.
- This is realized by storing the frontier as a priority queue ordered by g

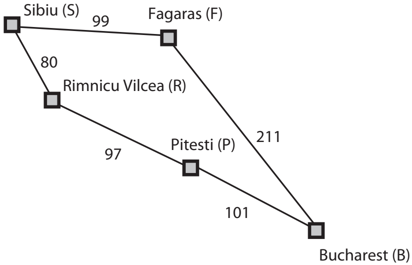




Uniform-Cost Search (aka Dijkstra's algorithm): Idea (3)

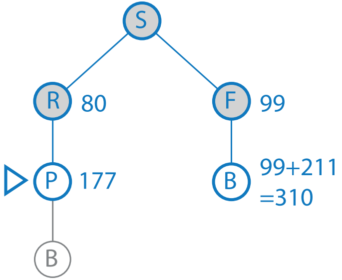
- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n)$.
- This is realized by storing the frontier as a priority queue ordered by g

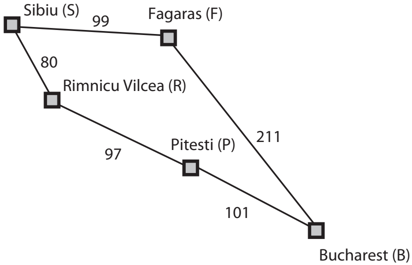




Uniform-Cost Search (aka Dijkstra's algorithm): Idea (4)

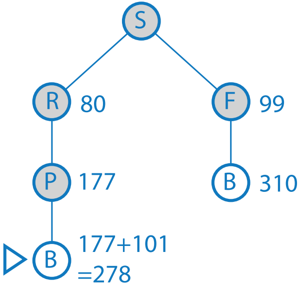
- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n)$.
- This is realized by storing the frontier as a priority queue ordered by g

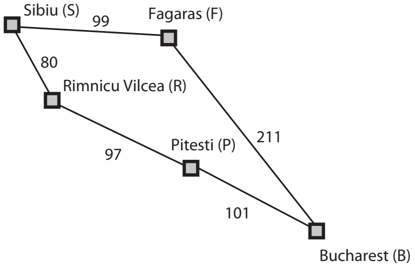




Uniform-Cost Search (aka Dijkstra's algorithm): Idea (5)

- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n)$.
- This is realized by storing the frontier as a priority queue ordered by g





Uniform-Cost Search: Changes to Breadth-First Algorithm

would find a non-optimal soln

- Ordering of the queue according to path costs.
- Goal test is applied to a node when it is selected for expansion rather than when it is first generated. This is because the first generated goal node might be on a suboptimal path.
- A test is added in case a better path to a frontier node is found.

Uniform-Cost Search: Algorithm (UninformedSearch.ipynb)

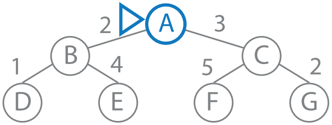
function Uniform-Cost-Search (problem) **returns** a solution or failure

node ← a node with `State` = *problem*.`Initial-State`, `Path-Cost`=0
frontier ← a priority queue 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 neither in *explored* nor *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: Algorithm (Step 1a)

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 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 neither in *explored* nor *frontier* **then**
            frontier ← `Insert`(*child*,*frontier*)
        else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
            replace that *frontier* node with *child*

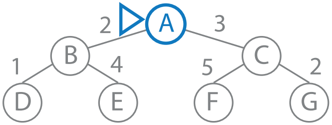


goal: G
node: A
frontier: A
explored: ∅

Uniform-Cost Search: Algorithm (Step 1b)

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 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 neither in explored nor frontier then
            frontier ← Insert(child,frontier)
        else if child.State is in frontier with higher Path-Cost then
            replace that frontier node with child
```

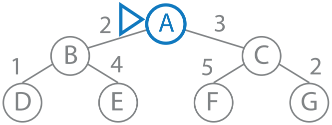


goal: G
node: A
frontier: ∅
explored: ∅

Uniform-Cost Search: Algorithm (Step 1c)

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 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 neither in *explored* nor *frontier* **then**
            frontier ← `Insert`(*child*,*frontier*)
        else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
            replace that *frontier* node with *child*

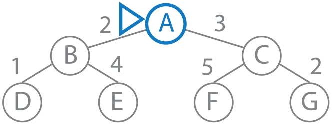


goal: G
node: A
frontier: ∅
explored: A

Uniform-Cost Search: Algorithm (Step 1d)

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 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 neither in *explored* nor *frontier* **then**
      frontier ← `Insert`(*child*,*frontier*)
    else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
      replace that *frontier* node with *child*

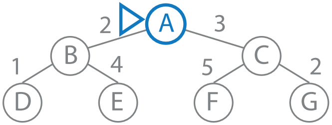


goal: G
node: A
frontier: B
explored: A

Uniform-Cost Search: Algorithm (Step 1e)

function Uniform-Cost-Search (problem) **returns** a solution or failure

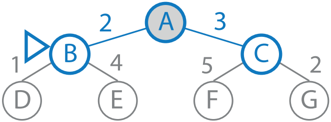
node ← a node with `State` = *problem*.`Initial-State`, `Path-Cost`=0
frontier ← a priority queue by `Path-Cost` with *node* as the only element
explored ← an empty set
loop do
    if `Empty`(*frontier*) **then return** failure
    node ← `Pop`(*frontier*) /* chooses the lowest-cost node in *frontier* */
    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 neither in *explored* nor *frontier* **then**
            frontier ← `Insert`(*child*,*frontier*)
        else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
            replace that *frontier* node with *child*



goal: G
node: A
frontier: B, C
explored: A

Uniform-Cost Search: Algorithm (Step 2a)

```
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 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 neither in explored nor frontier then
            frontier ← Insert(child,frontier)
        else if child.State is in frontier with higher Path-Cost then
            replace that frontier node with child
```



goal: G

node: B lower cost

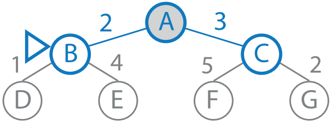
frontier: C

explored: A

Uniform-Cost Search: Algorithm (Step 2b)

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 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 neither in *explored* nor *frontier* **then**
            frontier ← `Insert`(*child*,*frontier*)
        else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
            replace that *frontier* node with *child*

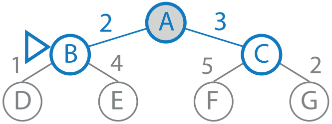


goal: G
node: B
frontier: C
explored: A, B

Uniform-Cost Search: Algorithm (Step 2c)

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 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 neither in *explored* nor *frontier* **then**
            frontier ← `Insert`(*child*,*frontier*)
        else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
            replace that *frontier* node with *child*

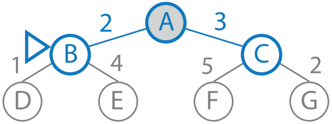


goal: G
node: B
frontier: C, D
explored: A, B

Uniform-Cost Search: Algorithm (Step 2d)

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 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 neither in *explored* nor *frontier* **then**
            frontier ← `Insert`(*child*,*frontier*)
        else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
            replace that *frontier* node with *child*

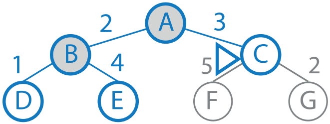


goal: G
node: B
frontier: C, D, E
explored: A, B

Uniform-Cost Search: Algorithm (Step 3a)

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 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 neither in *explored* nor *frontier* **then**
      frontier ← `Insert`(*child*,*frontier*)
    else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
      replace that *frontier* node with *child*

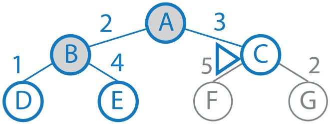


goal: G
node: C or D : both got lowest cost
frontier: D, E
explored: A, B

Uniform-Cost Search: Algorithm (Step 3b)

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 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 neither in *explored* nor *frontier* **then**
            frontier ← `Insert`(*child*,*frontier*)
        else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
            replace that *frontier* node with *child*

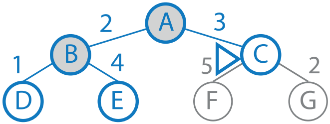


goal: G
node: C
frontier: D, E
explored: A, B, C

Uniform-Cost Search: Algorithm (Step 3c)

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 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 neither in *explored* nor *frontier* **then**
      frontier ← `Insert`(*child*,*frontier*)
    else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
      replace that *frontier* node with *child*

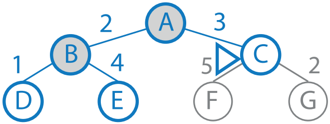


goal: G
node: C
frontier: D, E, F
explored: A, B, C

Uniform-Cost Search: Algorithm (Step 3d)

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 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 neither in *explored* nor *frontier* **then**
      frontier ← Insert(*child*,*frontier*)
    else if *child*.State is in *frontier* with higher Path-Cost **then**
      replace that *frontier* node with *child*



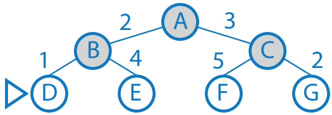
goal: G
node: C
frontier: D, E, F, G
explored: A, B, C

We would have stopped, when G is generated, in case of Breadth-first

Uniform-Cost Search: Algorithm (Step 4a)

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 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 neither in *explored* nor *frontier* **then**
            frontier ← Insert(*child*,*frontier*)
        else if *child*.State is in *frontier* with higher Path-Cost **then**
            replace that *frontier* node with *child*

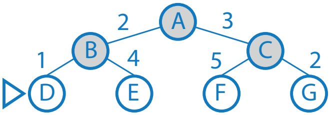


goal: G
node: D
frontier: G, E, F
explored: A, B, C

Uniform-Cost Search: Algorithm (Step 4b)

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 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 neither in *explored* nor *frontier* **then**
      frontier ← `Insert`(*child*,*frontier*)
    else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
      replace that *frontier* node with *child*

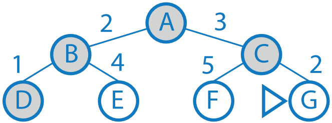


goal: G
node: D
frontier: G, E, F
explored: A, B, C, D

Uniform-Cost Search: Algorithm (Step 5a)

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 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 neither in *explored* nor *frontier* **then**
      frontier ← `Insert`(*child*,*frontier*)
    else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
      replace that *frontier* node with *child*



goal: G
node: G
frontier: E, F
explored: A, B, C, D

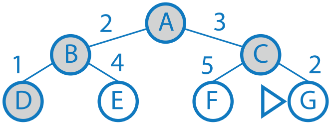
G as anode

Uniform-Cost Search: Algorithm (Step 5b)

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 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 neither in *explored* nor *frontier* **then**
      frontier ← `Insert`(*child*,*frontier*)
    else if *child*.`State` is in *frontier* with higher `Path-Cost` **then**
      replace that *frontier* node with *child*

we stopped since there is no other shorter path



goal state G found! node: D
frontier: E, F
explored: A, B, C, D

Uniform-Cost Search: Performance

We introduce 5

- the cost $\mathbf{C}^*$ of the optimal solution,
- the minimum step-cost ϵ, 1

and use the previously introduced criteria:

- **Completeness**: Yes, if costs are greater than 0 (otherwise infinite optimal paths of zero cost exist).
- **Optimality**: Yes (if cost $\geq \epsilon$ for positive ϵ).
- **Time complexity**: The worst case is that the goal branches of a node with huge costs and all other step costs are ϵ. The number of explored nodes (for e.g. $d = 1$) sums up to

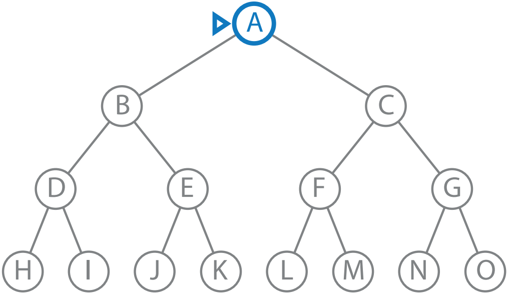
goal is the last node

$$(b - 1) + (b - 1)b + (b - 1)b^2 + \cdots + (b - 1)b^{\lfloor C^*/\epsilon \rfloor} = \mathcal{O}(b^{1+\lfloor C^*/\epsilon \rfloor}),$$

where $\lfloor a \rfloor$ returns the next lower integer of a. We require '+1' since the goal test is performed after the expansion.
- **Space complexity**: Equals time complexity since all nodes are stored.

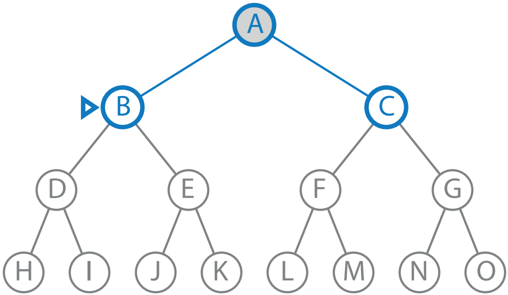
Depth-First Search: Idea (1) (UninformedSearch.ipynb)

The deepest node in the current frontier of the search tree is expanded:



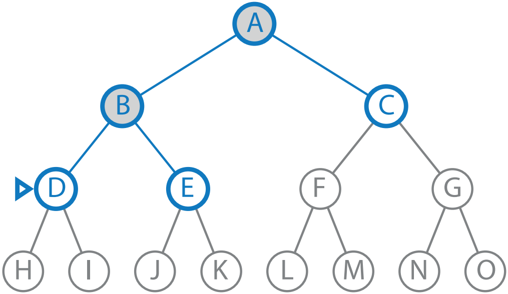
Depth-First Search: Idea (2)

The deepest node in the current frontier of the search tree is expanded:



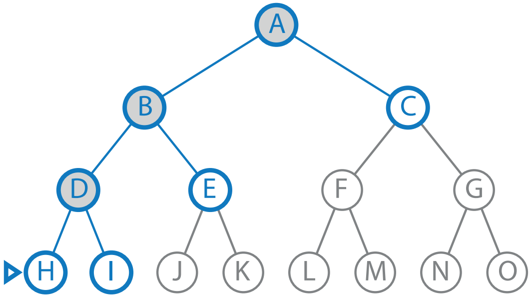
Depth-First Search: Idea (3)

The deepest node in the current frontier of the search tree is expanded:



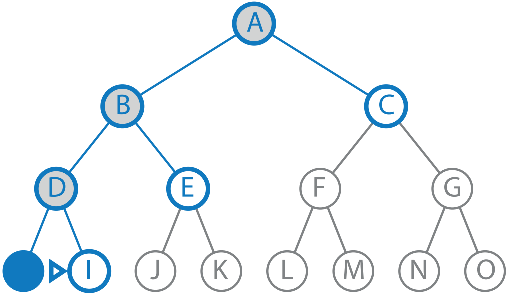
Depth-First Search: Idea (4)

The deepest node in the current frontier of the search tree is expanded:



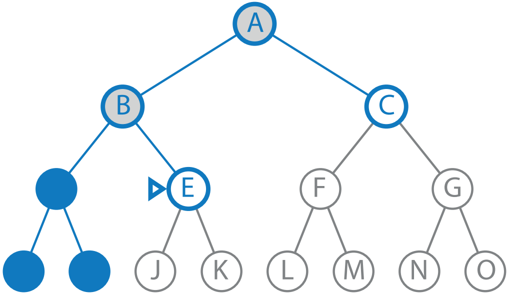
Depth-First Search: Idea (5)

The deepest node in the current frontier of the search tree is expanded:



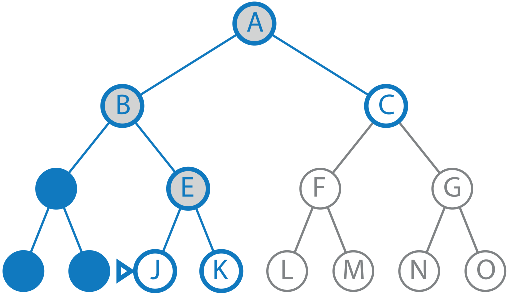
Depth-First Search: Idea (6)

The deepest node in the current frontier of the search tree is expanded:



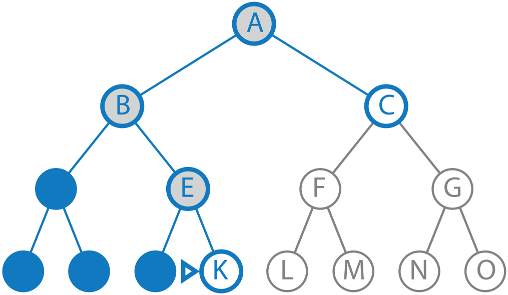
Depth-First Search: Idea (7)

The deepest node in the current frontier of the search tree is expanded:



Depth-First Search: Idea (8)

The deepest node in the current frontier of the search tree is expanded:



Depth-First Search: Performance

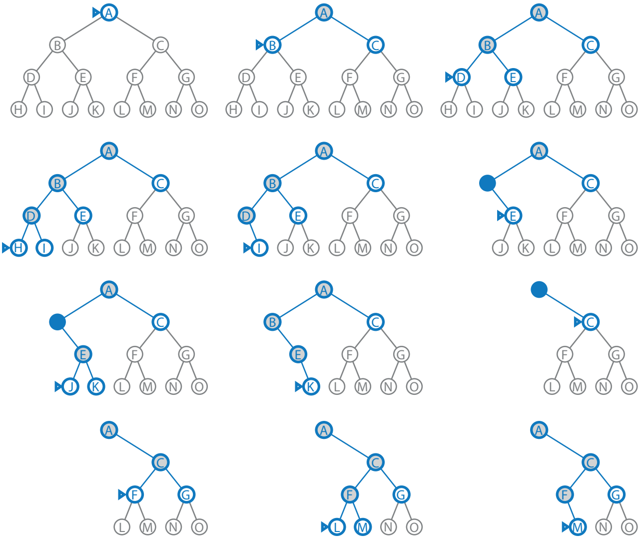
implemented in tree search

Reminder: Branching factor **b**, depth **d**, maximum length **m** of any path.

* - **Completeness**: No, if recursively implemented (see later); Yes if repeated states are avoided and the state space is finite.
- **Optimality**: No. Why?
- **Time complexity**: The worst case is that the goal path is tested last, resulting in $\mathcal{O}(b^m)$.
  Reminder: Breadth-first has $\mathcal{O}(b^d)$ and $d \leq m$.
- **Space complexity**: The advantage of depth-first when recursively implemented is a good space complexity: One only needs to store a single path from the root to the leaf plus unexplored sibling nodes (see next slide). There are at most **m** nodes to a leaf and **b** nodes branching off from each node, resulting in $\mathcal{O}(bm)$ nodes.

which is way better than $O(b^m)$

Space Requirement for Depth-First Search



Example:
$b = 10$, $d = m = 16$:
breadth-first: 10 exabytes
depth-first: 156 kilobytes

better by a factor of
$7 \cdot 10^{16}$

4

Depth-Limited Search: Idea

Shortcoming in depth-first search:
Depth-first search does not terminate in infinite state spaces. Why?

Solution:
Introduce depth limit l. in order to make sure that the goal is included

New issue:
How to choose the depth-limit?

Comment:
The algorithm on the next slide can also be used for depth-first by removing the limit.

Depth-Limited Search: Algorithm (UninformedSearch.ipynb)

Several implementations exist; we use a recursive form.

function Depth-Limited-S. (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* — no soln within depth limit

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

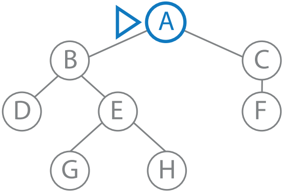
    else if *result* ≠ *failure* **then return** *result*

  if *cutoff_occurred* **then return** *cutoff* **else return** *failure* — no soln

Depth-Limited Search: Algorithm (Step 1)

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*

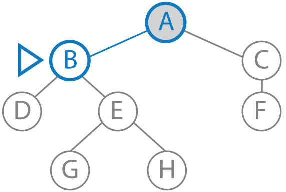


goal: F
limit: 2
current path: A
node: A
child: B
result: to be computed

Depth-Limited Search: Algorithm (Step 1)

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*

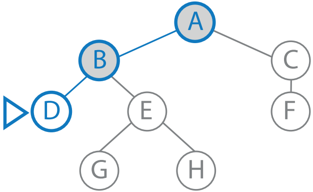


goal: F
limit: 1
current path: A, B
node: B
child: D
result: to be computed

Depth-Limited Search: Algorithm (Step 2b)

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
```

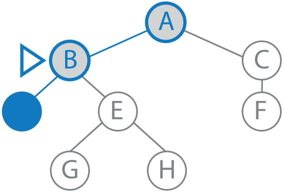


goal: F
limit: 0
current path: A, B, D
node: D
child: -
return: cutoff

Depth-Limited Search: Algorithm (Step 2c)

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*

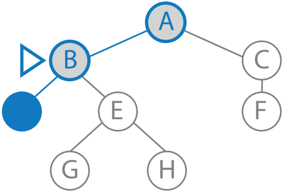


goal: F
limit: 1
current path: A, B
node: B
child: D
result: cutoff

Depth-Limited Search: Algorithm (Step 3a)

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*

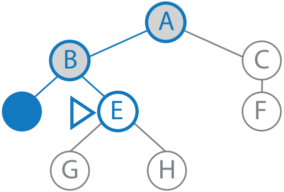


goal: F
limit: 1
current path: A, B
node: B
child: E
result: to be computed

Depth-Limited Search: Algorithm (Step 3b)

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*



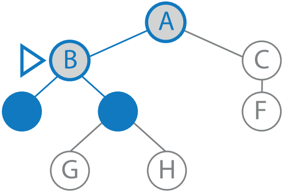
goal: F
limit: 0
current path: A, B, E
node: E
child: -
return: cutoff

No further expansion

Depth-Limited Search: Algorithm (Step 3c)

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
```

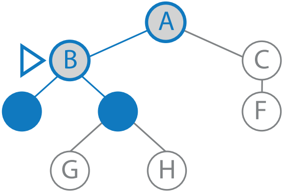


goal: F
limit: 1
current path: A, B
node: B
child: E
result: cutoff

Depth-Limited Search: Algorithm (Step 3d)

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
```



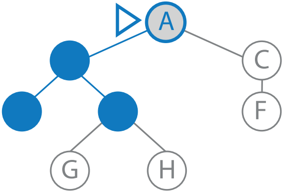
goal: F
limit: 1
current path: A, B
node: B
child: E
return: cutoff

We didn't expand E, since the original limit is 2

Depth-Limited Search: Algorithm (Step 4)

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*

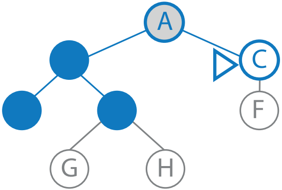


goal: F
limit: 2
current path: A
node: A
child: C
return: to be computed

Depth-Limited Search: Algorithm (Step 4)

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*

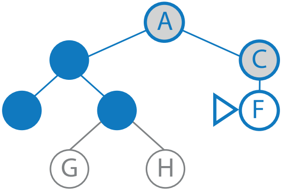


goal: F
limit: 1
current path: A, C
node: C
child: F
return: to be computed

Depth-Limited Search: Algorithm (Step 5b)

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*

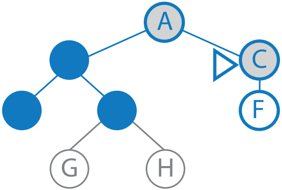


goal: F
limit: 0
current path: A, C, F
node: F
child: -
return: F

Depth-Limited Search: Algorithm (Step 5c)

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 $\leftarrow$ false
  for each *action* **in** *problem*.`Actions`(*node*.`State`) **do**
    child $\leftarrow$ `Child-Node`(*problem*, *node*, *action*)
    result $\leftarrow$ `Recursive-DLS`(*child*, *problem*, *limit* - 1)
    if *result* $=$ *cutoff* **then** *cutoff_occurred* $\leftarrow$ true
    else if *result* $\neq$ *failure* **then return** *result*
  if *cutoff_occurred* **then return** *cutoff* **else return** *failure*

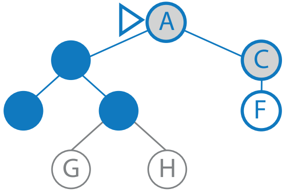


goal: F
limit: 1
found path: A, C, F
node: C
child: F
return: F

Depth-Limited Search: Algorithm (Step 5d)

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 $\leftarrow$ false
    for each *action* **in** *problem*.`Actions`(*node*.`State`) **do**
        child $\leftarrow$ `Child-Node`(*problem*, *node*, *action*)
        result $\leftarrow$ `Recursive-DLS`(*child*, *problem*, *limit* - 1)
        if *result* $=$ *cutoff* **then** *cutoff_occurred* $\leftarrow$ true
        else if *result* $\neq$ *failure* **then return** *result*
    if *cutoff_occurred* **then return** *cutoff* **else return** *failure*

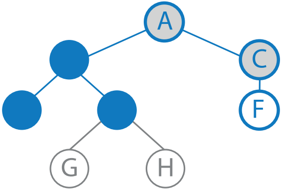


goal: F
limit: 2
found path: A, C, F
node: A
child: C
return: F

Depth-Limited Search: Algorithm (Step 5e)

function Depth-Limited-S. (problem,limit) **returns** a solution or failure/cutoff

return `Recursive-DLS(Make-Node(`*problem*`.Initial-State),`*problem*, *limit*`)`



goal: F
limit: 2
return: F

Depth-Limited Search: Performance

Reminder: Branching factor **b**, depth **d**, maximum length **m** of any path, and depth limit **l**.

- **Completeness**: No, if $l < d$. Why? if l is set wrong, you would not reach
- **Optimality**: No, if $l > d$. Why? For the same reason
- **Time complexity**: Same as for depth-first search, but with **l** instead of **m**: $\mathcal{O}(b^l)$.
- **Space complexity**: Same as for depth-first search, but with **l** instead of **m**: $\mathcal{O}(bl)$.

Iterative Deepening Search: Idea and Algorithm

5

'Optimization to Depth-limited Search'

Shortcoming in depth-limited search:
One typically does not know the depth **d** of the goal state.

Solution:
Use depth limit search and iteratively increase the depth limit **l**.

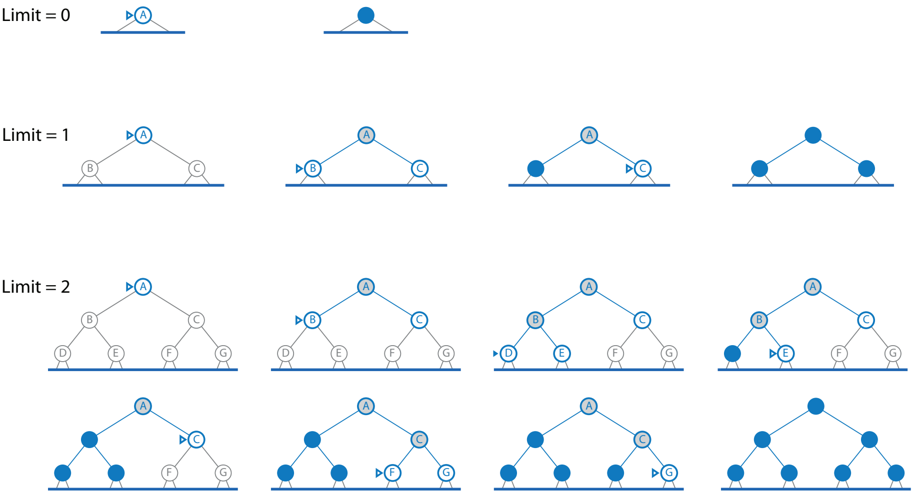
the cumulative order in which nodes are first visited is effectively breadth-first

function Iterative-Deepening-Search (problem) **returns** a solution or failure

for *depth*$= 0$ **to** ∞ **do**
  result $\leftarrow$ `Depth-Limited-Search`(*problem*, *depth*)
  if *result* $\neq$ cutoff **then return** *result*

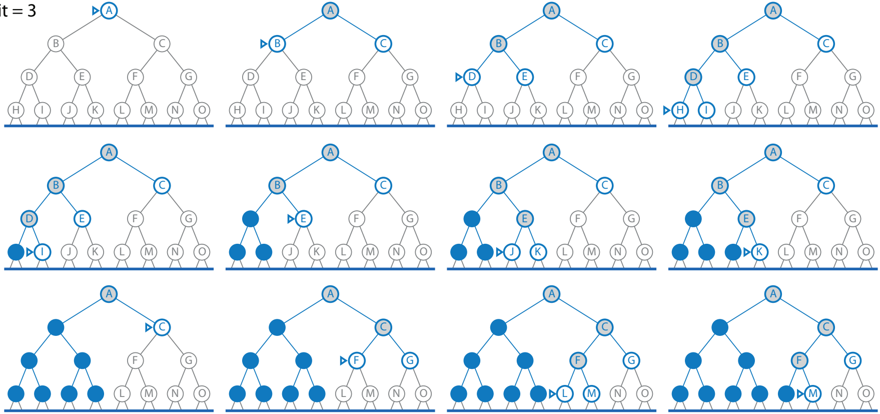
Iterative Deepening Search: Example (1)

(UninformedSearch.ipynb)



Iterative Deepening Search: Example (2)

Limit = 3



Iterative Deepening Search: Performance

Reminder: Branching factor **b**, depth **d**, maximum length **m** of any path, and depth limit **l**.

- **Completeness**: Yes, if depth d of the goal state is finite.
- **Optimality**: Yes (if cost = 1 per step); not optimal in general.
- **Time complexity**: The nodes at the bottom level are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times:

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

  which equals the one of breadth-first search.
- **Space complexity**: Same as for depth-first search, but with **d** instead of **m**: $\mathcal{O}(bd)$. Why? better than Breadth_first

Comparison of Computational Effort

Remember

The intuition that the iterative deepening search requires a lot of time is wrong. The search within the highest level is dominating.

Example:
$b = 10$, $d = 5$, solution at far right leaf:

- **Breadth-first search**:

$$\begin{aligned} & b + b^2 + b^3 + \cdots + b^d \\ =& 10 + 100 + 1000 + 10,000 + 100,000 = 111,110 \end{aligned}$$

- **Iterative deepening search**:

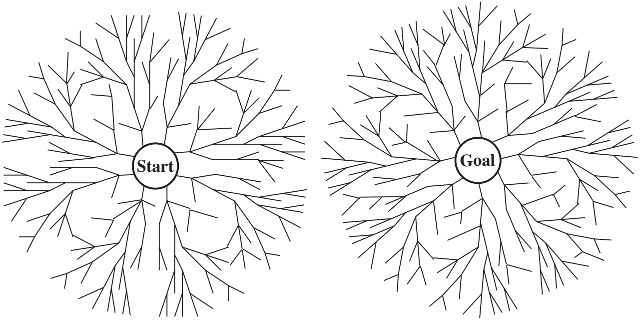
$$\begin{aligned} & (d)b + (d-1)b^2 + \ldots + (1)b^d \\ =& 5 \cdot 10 + 4 \cdot 100 + 3 \cdot 1000 + 2 \cdot 10,000 + 100,000 = 123,450 \end{aligned}$$

The difference is almost negligible and becomes relatively smaller, the larger the problem is.

Bidirectional Search: Idea

The main idea is to run two searches: One from the initial state and one backward from the goal, hoping that both searches meet in the middle.

Motivation:
$b^{\frac{d}{2}} + b^{\frac{d}{2}} < b^d$. This is also visualized in the figure, where the area from both search trees together is smaller than from one tree reaching the goal:



Tweedback Question

not always applicable, however can enormously reduce time complexity

Is it always possible to use bidirectional search?

Yes, if goal is known

Bidirectional Search: Comments

Bidirectional search requires one to “search backwards”.

- **Easy:** When all actions are reversible and there is only one goal, e.g., 8-puzzle, or finding a route in Romania
- **Difficult:** When the goal is an abstract description and there exist many goal states,
  e.g., 8-queens: “No queen attacks another queen”. What are the goal states? This would already be the solution...

Comparing Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes[a]	Yes[a,b]	No	~~Yes~~, NO if $l \geq d$	Yes[a]
Optimal?	Yes[c]/NO	Yes	No	No	Yes[c]
Time	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lfloor C^*/\epsilon \rfloor})$	$\mathcal{O}(b^m)$	$\mathcal{O}(b^l)$	$\mathcal{O}(b^d)$
Space	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lfloor C^*/\epsilon \rfloor})$	$\mathcal{O}(bm)$	$\mathcal{O}(bl)$	$\mathcal{O}(bd)$

a: complete if b is finite

b: complete if step costs $\geq \epsilon$ for positive ϵ

c: optimal if all step costs are identical

$\downarrow$

Can be very attractive

Summary

- A well-defined search problem consists of: the **initial state**, **actions**, a **transition model**, a **goal test** function, and a **path cost** function.
- Search algorithms are typically judged by **completeness**, **optimality**, **time complexity**, and **space complexity**.
- **Breadth-first search**: expands the shallowest nodes first before deeper ones; it is complete, optimal for unit step costs, but has exponential space complexity.
- **Uniform-cost search**: expands the node with the lowest path cost and is optimal for general step costs.
- **Depth-first search** and **Depth-limited search**: expands the deepest unexpended node first. It is neither complete nor optimal, but has linear space complexity.
- **Iterative deepening search**: calls depth-first search with increasing depth limits. It is complete, optimal for unit step costs, has time complexity like breadth-first search and linear space complexity.
- **Bidirectional search**: can enormously reduce time complexity, but is not always applicable.