Fundamentals of Artificial Intelligence – Uninformed Search

Matthias Althoff

TU München

Winter semester 2023/24

Organization

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

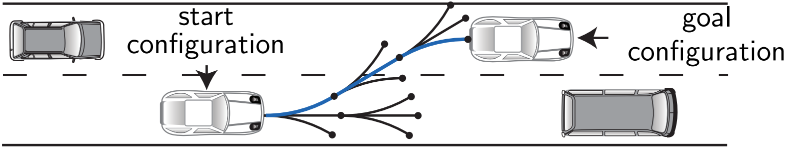
Learning Outcomes

- 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-like 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 understand why Best-First Search generalizes the above search techniques.
- You can compare the advantages and disadvantages of uninformed search strategies.

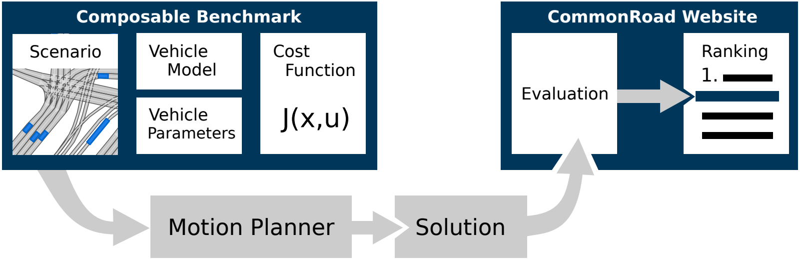
Motivation

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.



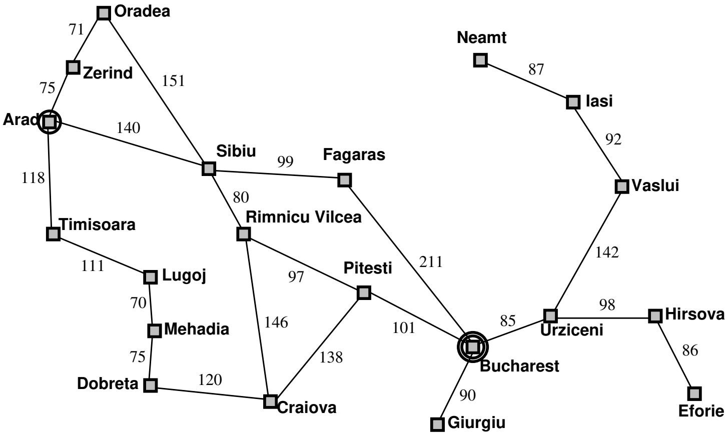
Introducing CommonRoad



Website: **https://commonroad.in.tum.de**

Another Example: Holiday in Romania

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

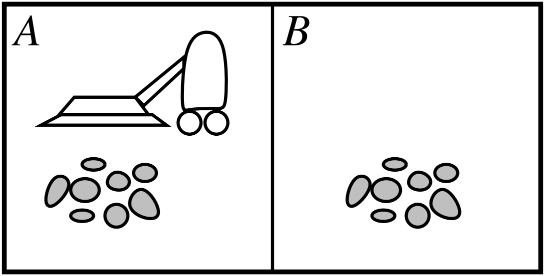


Well-Defined Search Problems

A search problem can be formally defined as follows:

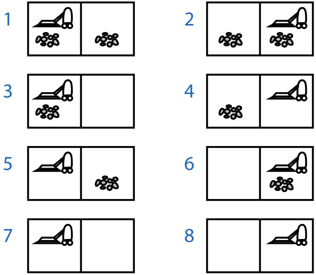
Component	Example
State Space: set of possible states	{`Arad, Zerind, ...`}
Initial State: where the agent starts in	`Arad`
Actions `Actions(s)`: possible actions that are applicable in a state `s`	`Actions(Arad)=`{`ToSibiu, ToTimisoara, ToZerind`}
Transition Model `Result(s, a)`: returns result of action `a` in state `s`	`Result(Arad, ToZerind) = Zerind`
Goal Test `Is-Goal(s)`: checks whether `s` is a goal state	`Is-Goal(Pitesti)=false, Is-Goal(Bucharest)=true`
Action Cost `c(s, a, s')`: cost of applying action `a` in state `s` to reach `s'`	`c(Arad, ToZerind, Zerind) = 75`

Vacuum-Cleaner World



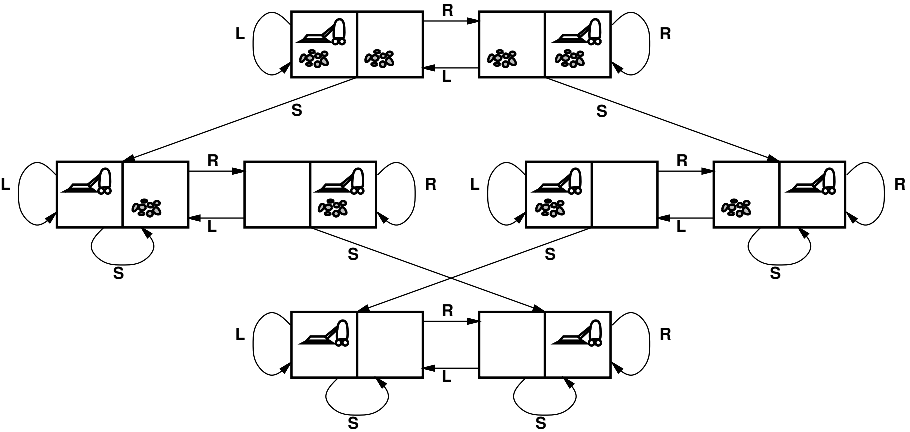
Percepts: location and contents, e.g., [*A*, *Dirty*]
Actions: *Left*, *Right*, *Suck*, *NoOp* (**No Op**eration)

Vacuum World (Toy Problem I)



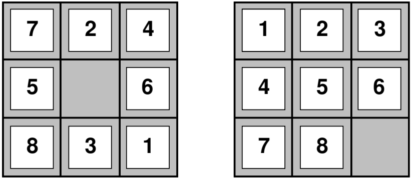
- **States**: Combination of cleaner and dirt locations: $2 \cdot 2^2 = 8$ states.
- **Initial state**: any state.
- **Actions**: `Left`, `Right`, and `Suck`.
- **Transition model**: see next slide.
- **Goal test**: checks whether all locations are clean.
- **Action cost**: Each step costs 1.

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)

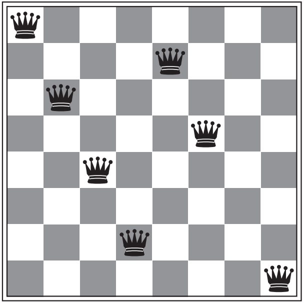


Start State **Goal State**

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.
- **Action 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.
- **Action 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

Not relevant for the exam

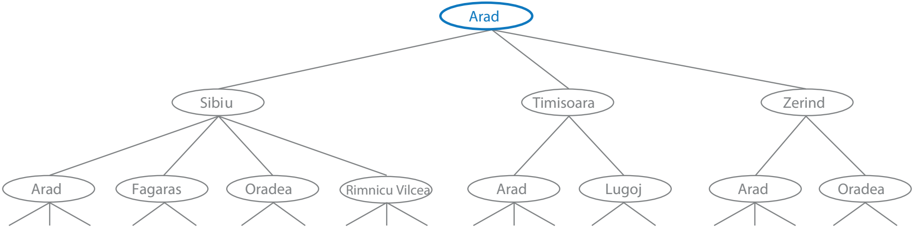
- **Route-Finding problem**: Airline travel planning, video streams in computer networks, etc.
- **Touring problem**: How to best visit a number of places, e.g., in the map of Romania?
- **Layout of digital circuits**: How to best place components and their connections on a circuit board?
- **Robot navigation**: 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.
- **Leaves**: Unexpanded nodes. We call the set of leaves the **frontier**.

A search tree is expanded by applying actions to leaves. Example:

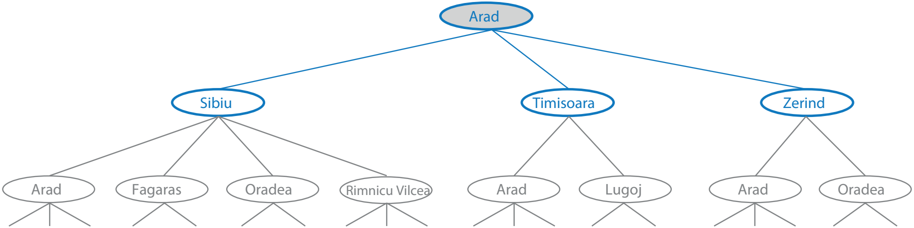


Generating a Search Tree (2)

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.
- **Leaves**: Unexpanded nodes. We call the set of leaves the **frontier**.

A search tree is expanded by applying actions to leaves. Example:

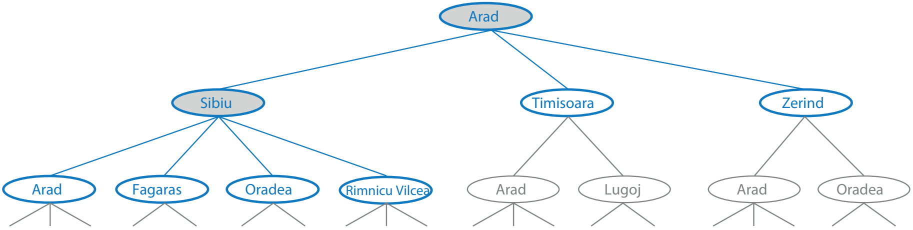


Generating a Search Tree (3)

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.
- **Leaves**: Unexpanded nodes. We call the set of leaves the **frontier**.

A search tree is expanded by applying actions to leaves. Example:



Tree-Like Search Algorithm

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

function Tree-Like-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

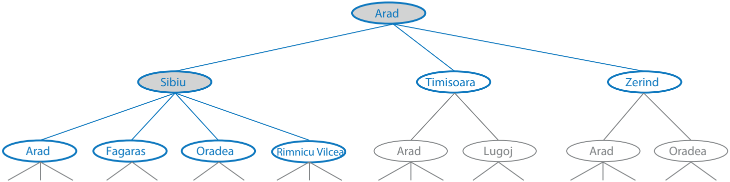
  expand the chosen node, adding the resulting nodes to the frontier

Tweedback Questions

- Does tree-like search always find a solution if one exists?
- Is the search tree of a finite graph also finite?

Avoiding Cycles in the Search Tree

Problem: 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. Reached states are stored in a **reached set**.

This idea is referred to as graph search (see next slide).

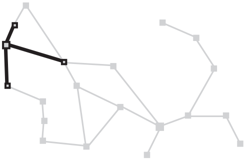
Graph Search Algorithm

Differences to the tree search are highlighted in orange.

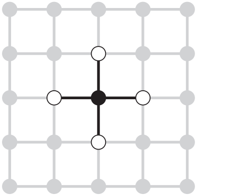
function Graph-Search (problem) **returns** a solution or failure

initialize the frontier using the initial state of *problem*
initialize the reached 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
    expand the chosen node, adding the resulting nodes to the frontier
        and the reached set,
        but only if not yet in the reached set on an equal or better path

Graph Search Algorithm: Illustrations



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



(a) (b) (c)

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

Tweedback Questions

- Is it possible that graph search is slower than tree-like search?
- What is the maximum number of steps required in graph search (according to slide 21)?
  - 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.
- You have to assemble parts and each part exists only once. What search technique would you use?
  - A Tree-like search.
  - B Graph search.

Graph Search is Not Always Required

Example: What is the best assembly of a modular robot to fulfill a task?

Graph search would only require more memory without adding any benefit.

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`

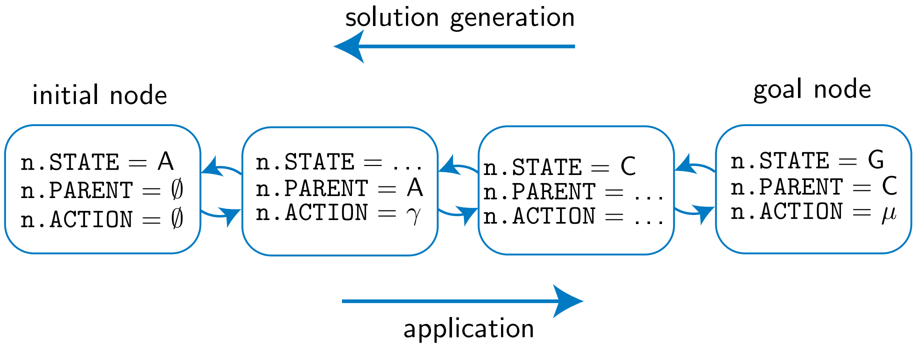
- `n.STATE`: The state in the state space to which the node corresponds;
- `n.PARENT`: The node in the search tree that generated this node;
- `n.ACTION`: The action that was applied to the parent to generate the node;
- `n.PATH-COST`: The cost, traditionally denoted by $g(n)$, of the path from the initial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

Operations on a queue (required for the frontier)

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

Obtaining the Solution

1. Obtain solution (i.e, state and action sequence) by iterating backwards over parents.
2. Apply actions forward to reach the goal.



State sequence: A, ..., C, G.

Action sequence: $\gamma, \ldots, \mu$.

Uninformed Search vs. Informed Search

Uninformed search

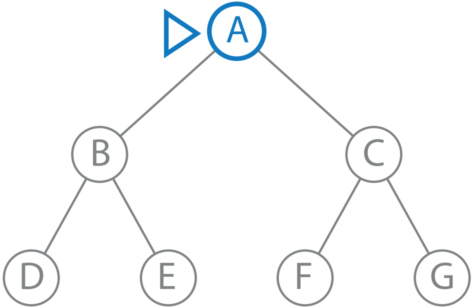
- No additional information besides the problem statement (states, initial state, actions, transition model, goal test, action cost) is provided.
- Uninformed search can only produce next states and check whether it is a goal state.

Informed search

- 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.

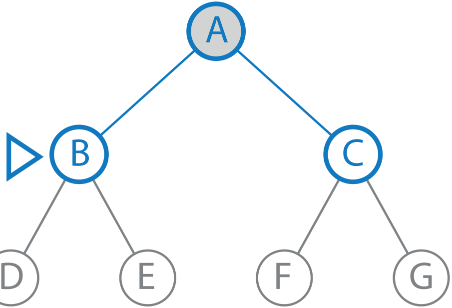
Breadth-First Search: Idea (1)

Special instance of the graph-search algorithm (slide 21): All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded:



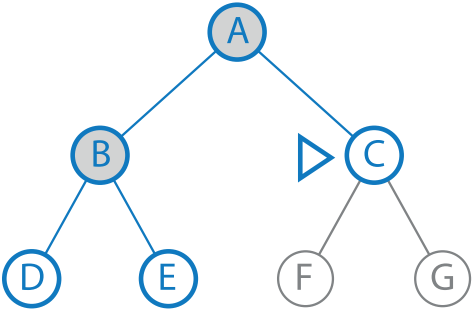
Breadth-First Search: Idea (2)

Special instance of the graph-search algorithm (slide 21): All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded:



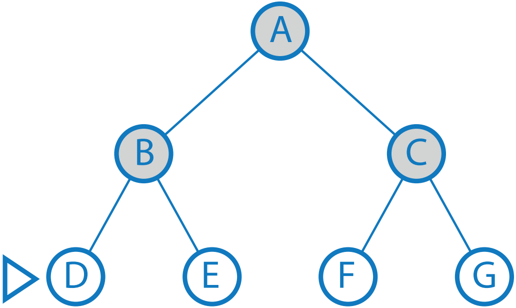
Breadth-First Search: Idea (3)

Special instance of the graph-search algorithm (slide 21): 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 21): All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded:



This will be realized by a **FIFO queue** (first-in-first-out) for the **frontier**: by first popping the first-added nodes, nodes are expanded level-by-level.

Breadth-First Search: Algorithm

(jupyter UninformedSearch.ipynb)

function Breadth-First-Search (problem) **returns** a solution or failure

```
node ← Node(State=problem.Initial-State, Path-Cost=0)
if problem.Is-Goal(node.State) then return Solution(node)
frontier ← a FIFO queue with node as the only element
reached ← {node.State}
loop do
  if Is-Empty(frontier) then return failure
  node ← Pop(frontier) // chooses a shallowest node in frontier
  for each child in Expand(problem, node) do
    s ← child.State
    if problem.Is-Goal(s) then return Solution(child)
    if s is not in reached then
      add s to reached
      frontier ← Add(child,frontier)
```

Auxiliary Algorithm: Expand

function Expand (problem, node) **yields** nodes

$s \leftarrow node.\texttt{State}$

for each *action* **in** $problem.\texttt{Actions}(s)$ **do**

$s' \leftarrow problem.\texttt{Result}(s, action)$

$cost \leftarrow node.\texttt{Path-Cost} + problem.\texttt{Action-Cost}(s, action, s')$

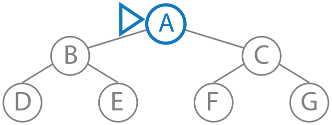
yield $\texttt{Node}(\texttt{State}{=}s', \texttt{Parent}{=}node,$

$\texttt{Action}{=}action, \texttt{Path-Cost}{=}cost, \texttt{Depth}{=}node.\texttt{Depth}{+}1)$

Breadth-First Search: Algorithm (Step 1a)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State, Path-Cost=0)`
if *problem*`.Is-Goal(`*node*`.State)` **then return** `Solution(`*node*`)`
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*`.State`}
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses a shallowest node in *frontier*
  for each *child* **in** `Expand(`*problem*, *node*`)` **do**
    s ← *child*`.State`
    if *problem*`.Is-Goal(`*s*`)` **then return** `Solution(`*child*`)`
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← `Add(`*child*,*frontier*`)`

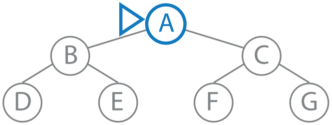


goal: F
node: A
frontier: A
reached: A

Breadth-First Search: Algorithm (Step 1b)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← `Node`(`State`=*problem*.`Initial-State`, `Path-Cost`=0)
if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*.`State`}
loop do
  if `Is-Empty`(*frontier*) **then return** failure
  node ← `Pop`(*frontier*) // chooses a shallowest node in *frontier*
  for each *child* **in** `Expand`(*problem*, *node*) **do**
    s ← *child*.`State`
    if *problem*.`Is-Goal`(*s*) **then return** `Solution`(*child*)
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← `Add`(*child*,*frontier*)

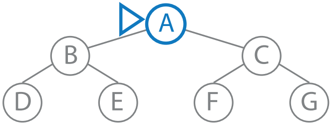


goal: F
node: A
frontier: ∅
reached: A

Breadth-First Search: Algorithm (Step 1c)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← `Node`(`State`=*problem*.`Initial-State`, `Path-Cost`=0)
if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*.`State`}
loop do
  if `Is-Empty`(*frontier*) **then return** failure
  node ← `Pop`(*frontier*) // chooses a shallowest node in *frontier*
  for each *child* **in** `Expand`(*problem*, *node*) **do**
    s ← *child*.`State`
    if *problem*.`Is-Goal`(*s*) **then return** `Solution`(*child*)
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← `Add`(*child*,*frontier*)

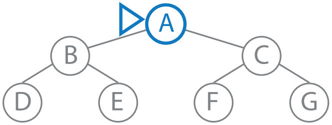


goal: F
node: A
frontier: B
reached: A, B

Breadth-First Search: Algorithm (Step 1d)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State, Path-Cost=0)`
if *problem*`.Is-Goal(`*node*`.State)` **then return** `Solution(`*node*`)`
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*`.State`}
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses a shallowest node in *frontier*
  for each *child* **in** `Expand(`*problem*, *node*`)` **do**
    s ← *child*`.State`
    if *problem*`.Is-Goal(`*s*`)` **then return** `Solution(`*child*`)`
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← `Add(`*child*,*frontier*`)`

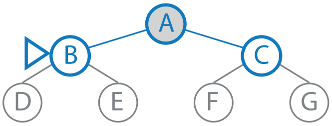


goal: F
node: A
frontier: B, C
reached: A, B, C

Breadth-First Search: Algorithm (Step 2a)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← `Node`(`State`=*problem*.`Initial-State`, `Path-Cost`=0)
if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*.`State`}
loop do
  if `Is-Empty`(*frontier*) **then return** failure
  node ← `Pop`(*frontier*) // chooses a shallowest node in *frontier*
  for each *child* **in** `Expand`(*problem*, *node*) **do**
    s ← *child*.`State`
    if *problem*.`Is-Goal`(*s*) **then return** `Solution`(*child*)
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← `Add`(*child*,*frontier*)

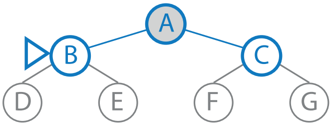


goal: F
node: B
frontier: C
reached: A, B, C

Breadth-First Search: Algorithm (Step 2b)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State, Path-Cost=0)`
if *problem*`.Is-Goal(`*node*`.State)` **then return** `Solution(`*node*`)`
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*`.State`}
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses a shallowest node in *frontier*
  for each *child* **in** `Expand(`*problem*, *node*`)` **do**
    s ← *child*`.State`
    if *problem*`.Is-Goal(`*s*`)` **then return** `Solution(`*child*`)`
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← `Add(`*child*,*frontier*`)`

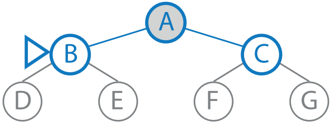


goal: F
node: B
frontier: C, D
reached: A, B, C, D

Breadth-First Search: Algorithm (Step 2c)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State, Path-Cost=0)`
if *problem*`.Is-Goal(`*node*`.State)` **then return** `Solution(`*node*`)`
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*`.State`}
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses a shallowest node in *frontier*
  for each *child* **in** `Expand(`*problem*, *node*`)` **do**
    s ← *child*`.State`
    if *problem*`.Is-Goal(`*s*`)` **then return** `Solution(`*child*`)`
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← `Add(`*child*,*frontier*`)`

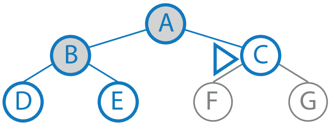


goal: F
node: B
frontier: C, D, E
reached: A, B, C, D, E

Breadth-First Search: Algorithm (Step 3a)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← Node(State=*problem*.Initial-State, Path-Cost=0)
if *problem*.Is-Goal(*node*.State) **then return** Solution(*node*)
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*.State}
loop do
  if Is-Empty(*frontier*) **then return** failure
  node ← Pop(*frontier*) // chooses a shallowest node in *frontier*
  for each *child* **in** Expand(*problem*, *node*) **do**
    s ← *child*.State
    if *problem*.Is-Goal(*s*) **then return** Solution(*child*)
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← Add(*child*,*frontier*)

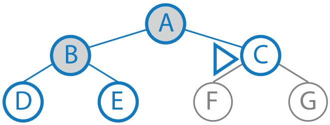


goal: F
node: C
frontier: D, E
reached: A, B, C, D, E

Breadth-First Search: Algorithm (Step 3b)

function Breadth-First-Search (problem) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State, Path-Cost=0)`
if *problem*`.Is-Goal(`*node*`.State)` **then return** `Solution(`*node*`)`
frontier ← a FIFO queue with *node* as the only element
reached ← {*node*`.State`}
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses a shallowest node in *frontier*
  for each *child* **in** `Expand(`*problem*, *node*`)` **do**
    s ← *child*`.State`
    if *problem*`.Is-Goal(`*s*`)` **then return** `Solution(`*child*`)`
    if *s* is not in *reached* **then**
      add *s* to *reached*
      frontier ← `Add(`*child*,*frontier*`)`



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

Breadth-First Search: Performance

We introduce

- the branching factor b (maximum number of successors of any node),
- the depth d (depth of the shallowest goal node),
- the maximum length m of any path in the state space,

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, b_0 for $d = 5$ are:

- $M = 2,\ b_0 = 2,$
- $M = 1.5,\ b_0 = 3,$
- $M = 1.35,\ b_0 = 4.$

Since M, b_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	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

Breadth-First Search: CommonRoad Example

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

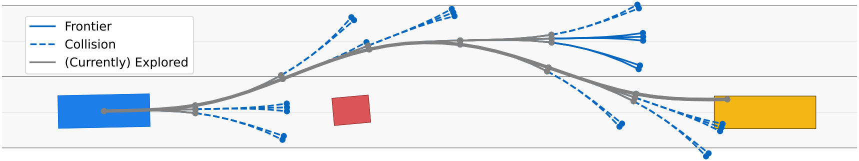
Breadth-First Search: CommonRoad Example

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Breadth-First Search: CommonRoad Example

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

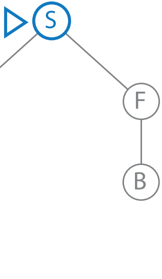
Breadth-First Search: CommonRoad Example

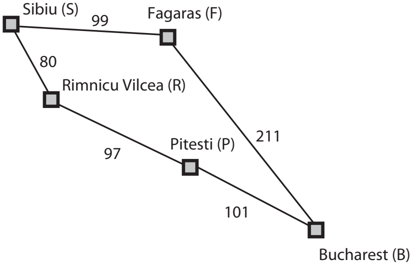


- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

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

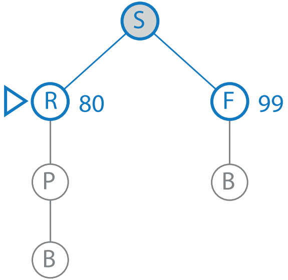
- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n) = n.\texttt{Path} - \texttt{Cost}$.
- This is realized by storing the frontier as a priority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

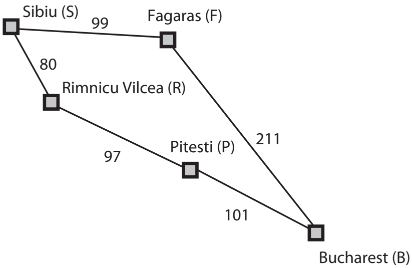




Uniform-Cost Search (aka Dijkstra's algorithm): Idea (2)

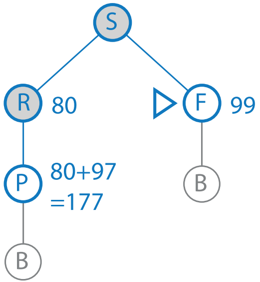
- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n) = n.\mathtt{Path} - \mathtt{Cost}$.
- This is realized by storing the frontier as a priority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

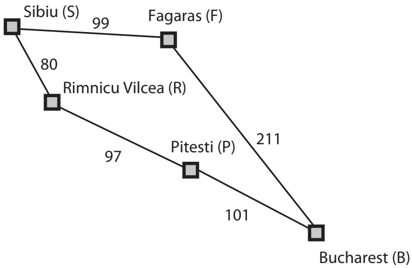




Uniform-Cost Search (aka Dijkstra's algorithm): Idea (3)

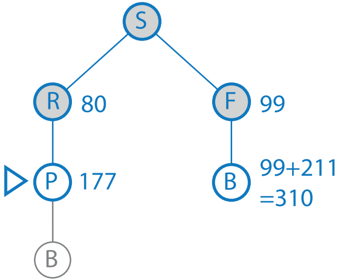
- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n) = n.\texttt{Path} - \texttt{Cost}$.
- This is realized by storing the frontier as a priority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

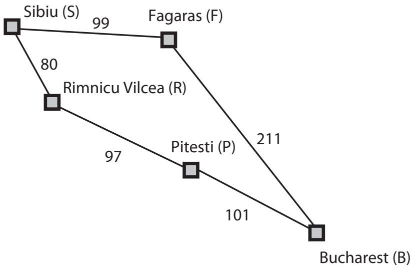




Uniform-Cost Search (aka Dijkstra's algorithm): Idea (4)

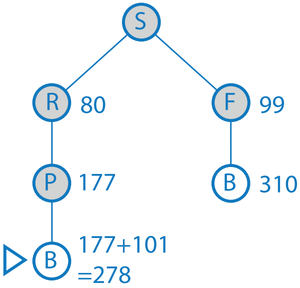
- When all step costs are equal, breadth-first is optimal because it always expands the shallowest nodes.
- Uniform-cost search is optimal for any step costs, as it expands the node with the lowest path cost $g(n) = n.\texttt{Path} - \texttt{Cost}$.
- This is realized by storing the frontier as a priority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

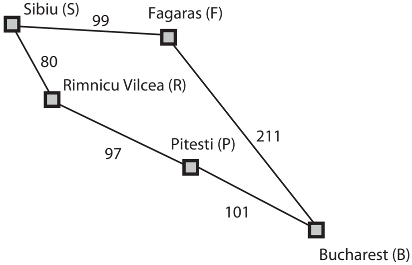




Uniform-Cost Search (aka Dijkstra's algorithm): Idea (5)

- 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 $g(n) = n.\texttt{Path} - \texttt{Cost}$.
- This is realized by storing the frontier as a priority queue ordered by g.





Best-First Search: A generalization of Uniform-Cost Search

Uniform-Cost Search is a special form of a more general search approach: **Best-First Search**.

It uses a **priority queue** for the **frontier**, which is ordered by some parametrized evaluation function $f(n)$: always the node with the minimum value of $f(n)$ is expanded first.

$\rightarrow$ different functions f result in different algorithms.

$\rightarrow$ Uniform-Cost Search is Best-First Search with $f(n) = g(n)$.

Best-First Search: Changes to Breadth-First Search

- Goal test is applied to a node when it is selected for expansion rather than when it is first generated.
  Reason: the first generated goal node might be on a suboptimal path.
- A test is added in case a better path to an already reached state is found.
  Realization: subsequent operations on the lookup table `reached`.

Operations on the lookup table `reached`

- `s in reached`: returns true if `reached` contains the state `s`.
- `reached[s]`: returns the node for state `s` in `reached`; due to the internal structure of lookup tables, this can be done in constant time[a].
- `reached[s]` $\leftarrow$ `n`: sets the value of state `s` to the node `n`.

[a] https://www.geeksforgeeks.org/introduction-to-hashing-data-structure-and-algorithm-tutorials/

Best-First: Algorithm

(UninformedSearch.ipynb)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State,Path-Cost=0)`
frontier ← a priority queue ordered by f, with *node* as the only element
reached ← a lookup table, with one entry (*node*`.State` → *node*)
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*`.Is-Goal(`*node*`.State)` **then return** `Solution(`*node*`)`
  for each *child* **in** `Expand(`*problem*,*node*`)` **do**
    s ← *child*`.State`
    if s is not in *reached* **or** *child.Path-Cost* < *reached*[s]`.Path-Cost` **then**
      reached[s] ← *child*
      frontier ← `Insert(`*child*,*frontier*`)`

By removing the lines in orange we obtain a tree-like search.

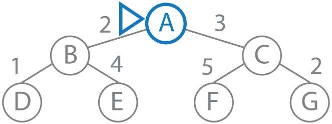
Tweedback Question

Can Breadth-First Search be implemented using Best-First Search?

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 1a)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State,Path-Cost=0)`
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.`State` → *node*)
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.`Is-Goal(`*node*.`State)` **then return** `Solution(`*node*`)`
  for each *child* **in** `Expand(`*problem*,*node*`)` **do**
    s ← *child*.`State`
    if s is not in *reached* **or** *child.Path-Cost* < *reached*[s].`Path-Cost` **then**
      reached[s] ← *child*
      frontier ← `Add(`*child*,*frontier*`)`

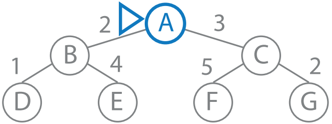


goal: G
node: A(0) (path cost in parentheses)
frontier: A(0)
reached: A

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 1b)

function Best-First-Search (problem, f) **returns** a solution or failure

```
node ← Node(State=problem.Initial-State,Path-Cost=0)
frontier ← a priority queue ordered by f, with n ode as the only element
reached ← a lookup table, with one entry (node.State → node)
loop do
  if Is-Empty(frontier) then return failure
  node ← Pop(frontier) // chooses the node n with minimum f(n) in frontier
  if problem.Is-Goal(node.State) then return Solution(node)
  for each child in Expand(problem,node) do
    s ← child.State
    if s is not in reached or child.Path-Cost < reached[s].Path-Cost then
      reached[s] ← child
      frontier ← Add(child,frontier)
```

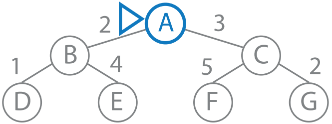


goal: G
node: A(0)
frontier: $\emptyset$
reached: A

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 1c)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State,Path-Cost=0)`
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.`State` → *node*)
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*`.Is-Goal(`*node*`.State)` **then return** `Solution(`*node*`)`
  for each *child* **in** `Expand(`*problem*`,`*node*`)` **do**
    s ← *child*`.State`
    if s is not in *reached* **or** *child.Path-Cost* < *reached*[s]`.Path-Cost` **then**
      reached[s] ← *child*
      frontier ← `Add(`*child*`,`*frontier*`)`

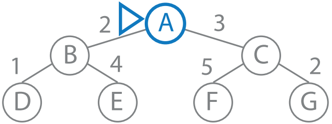


goal: G
node: A(0)
frontier: B(2)
reached: A, B

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 1d)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← Node(State=*problem*.Initial-State,Path-Cost=0)
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.State → *node*)
loop do
  if Is-Empty(*frontier*) **then return** failure
  node ← Pop(*frontier*) // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.Is-Goal(*node*.State) **then return** Solution(*node*)
  for each *child* **in** Expand(*problem*,*node*) **do**
    s ← *child*.State
    if s is not in *reached* **or** *child.Path-Cost* < *reached*[s].Path-Cost **then**
      reached[s] ← *child*
      frontier ← Add(*child*,*frontier*)

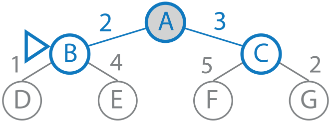


goal: G
node: A(0)
frontier: B(2), C(3)
reached: A, B, C

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 2a)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State,Path-Cost=0)`
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.`State` → *node*)
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.`Is-Goal(`*node*.`State)` **then return** `Solution(`*node*`)`
  for each *child* **in** `Expand(`*problem*,*node*`)` **do**
    s ← *child*.`State`
    if s is not in *reached* **or** *child.Path-Cost* < *reached*[s].`Path-Cost` **then**
      reached[s] ← *child*
      frontier ← `Add(`*child*,*frontier*`)`

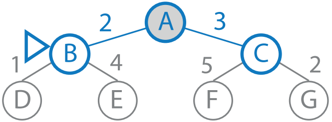


goal: G
node: B(2)
frontier: C(3)
reached: A, B, C

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 2b)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← Node(State=*problem*.Initial-State,Path-Cost=0)
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.State → *node*)
loop do
  if Is-Empty(*frontier*) **then return** failure
  node ← Pop(*frontier*) // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.Is-Goal(*node*.State) **then return** Solution(*node*)
  for each *child* **in** Expand(*problem*,*node*) **do**
    s ← *child*.State
    if s is not in *reached* **or** *child.Path-Cost* < *reached*[s].Path-Cost **then**
      reached[s] ← *child*
      frontier ← Add(*child*,*frontier*)

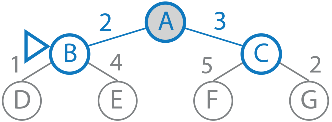


goal: G
node: B(2)
frontier: C(3), D(3)
reached: A, B, C, D

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 2c)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State,Path-Cost=0)`
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.`State` → *node*)
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.`Is-Goal(`*node*.`State)` **then return** `Solution(`*node*`)`
  for each *child* **in** `Expand(`*problem*,*node*`)` **do**
    s ← *child*.`State`
    if s is not in *reached* **or** *child.Path-Cost* < *reached*[s].`Path-Cost` **then**
      reached[s] ← *child*
      frontier ← `Add(`*child*,*frontier*`)`

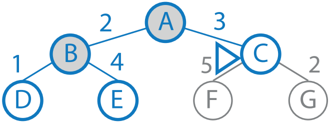


goal: G
node: B(2)
frontier: C(3), D(3), E(6)
reached: A, B, C, D, E

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 3a)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State,Path-Cost=0)`
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.`State` → *node*)
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← Pop(*frontier*) // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.`Is-Goal(`*node*.`State)` **then return** `Solution(`*node*`)`
  for each *child* **in** `Expand(`*problem*,*node*`)` **do**
    s ← *child*.`State`
    if *s* is not in *reached* **or** *child.Path-Cost* < *reached*[s].`Path-Cost` **then**
      reached[s] ← *child*
      frontier ← `Add(`*child*,*frontier*`)`

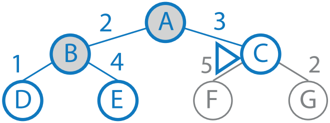


goal: G
node: C(3)
frontier: D(3), E(6)
reached: A, B, C, D, E

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 3b)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← Node(State=*problem*.Initial-State,Path-Cost=0)
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.State → *node*)
loop do
  if Is-Empty(*frontier*) **then return** failure
  node ← Pop(*frontier*) // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.Is-Goal(*node*.State) **then return** Solution(*node*)
  for each *child* **in** Expand(*problem*,*node*) **do**
    s ← *child*.State
    if *s* is not in *reached* **or** *child.Path-Cost* < *reached*[s].Path-Cost **then**
      reached[s] ← *child*
      frontier ← Add(*child*,*frontier*)

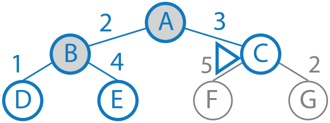


goal: G
node: C(3)
frontier: D(3), E(6), F(8)
reached: A, B, C, D, E, F

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 3c)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← Node(State=*problem*.Initial-State,Path-Cost=0)
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.State → *node*)
loop do
  if Is-Empty(*frontier*) **then return** failure
  node ← Pop(*frontier*) // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.Is-Goal(*node*.State) **then return** Solution(*node*)
  for each *child* **in** Expand(*problem*,*node*) **do**
    s ← *child*.State
    if s is not in *reached* **or** *child*.*Path-Cost* < *reached*[s].Path-Cost **then**
      reached[s] ← *child*
      frontier ← Add(*child*,*frontier*)

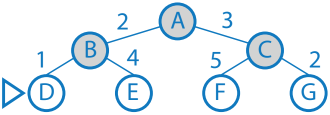


goal: G
node: C(3)
frontier: D(3), G(5), E(6), F(8)
reached: A, B, C, D, E, F, G

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 4)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← Node(State=*problem*.Initial-State,Path-Cost=0)
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.State → *node*)
loop do
  if Is-Empty(*frontier*) **then return** failure
  node ← Pop(*frontier*) // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.Is-Goal(*node*.State) **then return** Solution(*node*)
  for each *child* **in** Expand(*problem*,*node*) **do**
    s ← *child*.State
    if s is not in *reached* **or** *child*.*Path-Cost* < *reached*[s].Path-Cost **then**
      reached[s] ← *child*
      frontier ← Add(*child*,*frontier*)

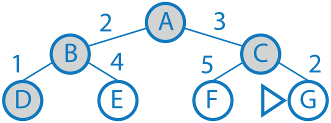


goal: G
node: D(3)
frontier: G(5), E(6), F(8)
reached: A, B, C, D, E, F, G

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 5a)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← Node(State=*problem*.Initial-State,Path-Cost=0)
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.State → *node*)
loop do
  if Is-Empty(*frontier*) **then return** failure
  node ← Pop(*frontier*) // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*.Is-Goal(*node*.State) **then return** Solution(*node*)
  for each *child* **in** Expand(*problem*,*node*) **do**
    s ← *child*.State
    if *s* is not in *reached* **or** *child.Path-Cost* < *reached*[s].Path-Cost **then**
      reached[s] ← *child*
      frontier ← Add(*child*,*frontier*)

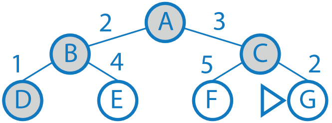


goal: G
node: G(5)
frontier: E(6), F(8)
reached: A, B, C, D, E, F, G

Best-First Search: Algorithm for $f(n) = g(n)$ (Step 5b)

function Best-First-Search (problem, f) **returns** a solution or failure

node ← `Node(State=`*problem*`.Initial-State,Path-Cost=0)`
frontier ← a priority queue ordered by f, with *n ode* as the only element
reached ← a lookup table, with one entry (*node*.State → *node*)
loop do
  if `Is-Empty(`*frontier*`)` **then return** failure
  node ← `Pop(`*frontier*`)` // chooses the node n with minimum $f(n)$ in *frontier*
  if *problem*`.Is-Goal(`*node*`.State)` **then return** `Solution(`*node*`)`
  for each *child* **in** `Expand(`*problem*,*node*`)` **do**
    s ← *child*`.State`
    if *s* is not in *reached* **or** *child.Path-Cost* < *reached*[s]`.Path-Cost` **then**
      reached[s] ← *child*
      frontier ← `Add(`*child*,*frontier*`)`



goal state G found!
node: G(5)
frontier: E(6), F(8)
reached: A, B, C, D, E, F, G

Best-First Search: Performance for $f(n) = g(n)$

We introduce

- the cost C^* of the optimal solution,
- the minimum step-cost ϵ,

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

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

  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.

Best-First Search: CommonRoad Example for $f(n) = g(n)$

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- $g(n)$: Time to reach current state.
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

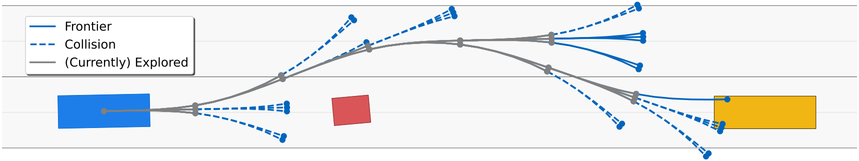
Best-First Search: CommonRoad Example for $f(n) = g(n)$

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- $g(n)$: Time to reach current state.
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Best-First Search: CommonRoad Example for $f(n) = g(n)$

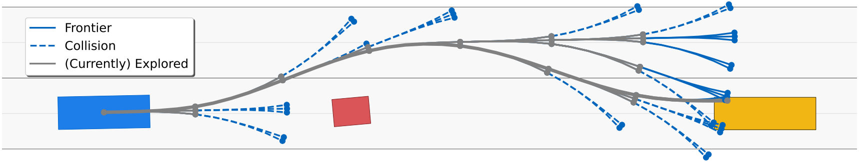
- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- $g(n)$: Time to reach current state.
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Best-First Search: CommonRoad Example for $f(n) = g(n)$



- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- $g(n)$: Time to reach current state.
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

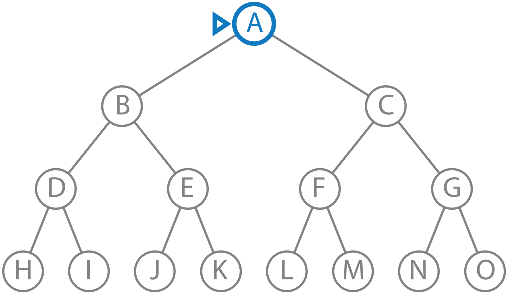
Best-First Search: CommonRoad Example for $f(n) = g(n)$



- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- $g(n)$: Time to reach current state.
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

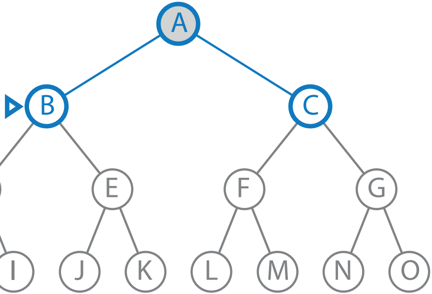
Depth-First Search: Idea (1)

We always expand the deepest node in the frontier:



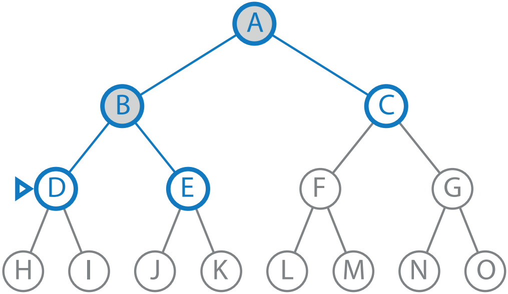
Depth-First Search: Idea (2)

We always expand the deepest node in the frontier:



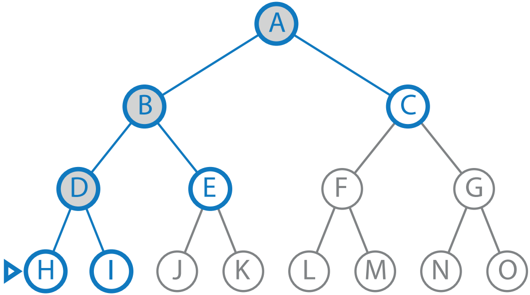
Depth-First Search: Idea (3)

We always expand the deepest node in the frontier:



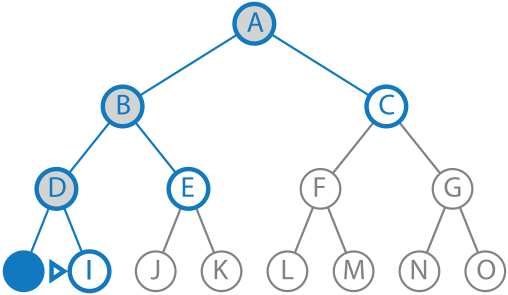
Depth-First Search: Idea (4)

We always expand the deepest node in the frontier:



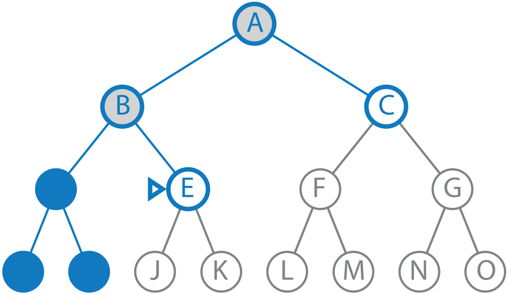
Depth-First Search: Idea (5)

We always expand the deepest node in the frontier:



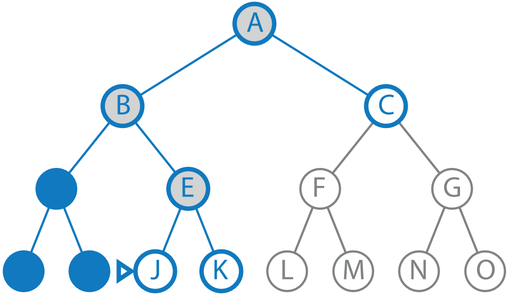
Depth-First Search: Idea (6)

We always expand the deepest node in the frontier:



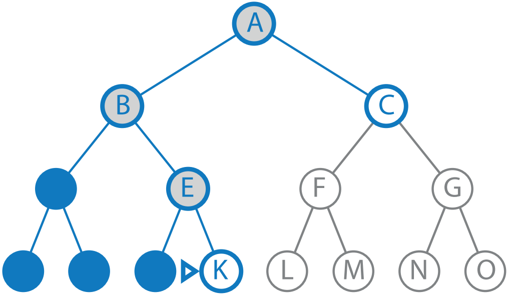
Depth-First Search: Idea (7)

We always expand the deepest node in the frontier:



Depth-First Search: Idea (8)

We always expand the deepest node in the frontier:



Tweedback Question

Can Depth-First Search be implemented using Best-First Search?

Solution: Best-First Search with $f(n) = -n.\texttt{Depth}$.

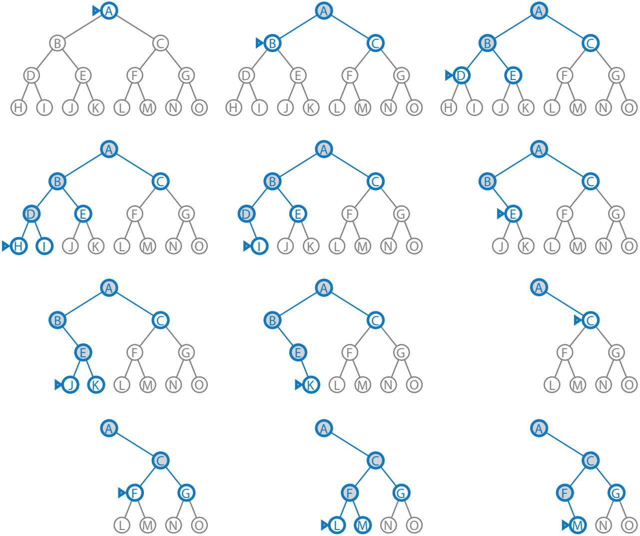
But: usually implemented as a tree-like search (for pseudo-code, please see Depth-Limited Search on slide 86 with $limit = \infty$).

Depth-First Search: Performance

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

- **Completeness**: No. But: for finite state spaces, completeness can be achieved by checking for cycles.
- **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.

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
$\approx 7 \cdot 10^{13}$

Depth-First Search: CommonRoad Example

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

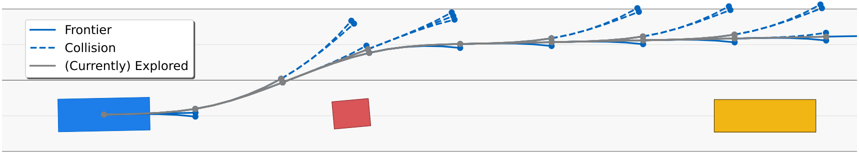
Depth-First Search: CommonRoad Example

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Depth-First Search: CommonRoad Example

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Depth-First Search: CommonRoad Example



- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

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.

New issue:
How to choose the depth-limit?

Realization:
LIFO queue (last-in-first-out, also known as *stack*) for the frontier: it first pops the most recently added node.

Depth-Limited Search: Algorithm (UninformedSearch.ipynb)

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

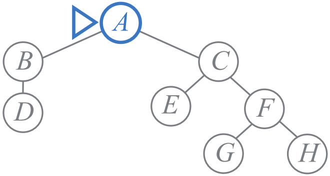
frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

Here: the function `Is-Cycle`(*node*) iterates the chain of `Parent`-pointers upwards to check whether the state *node*.`State` already exists in a previous node. The maximum number of iterated `Parent`-pointers is generally user-defined.

Depth-Limited Search: Algorithm (Step 1)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

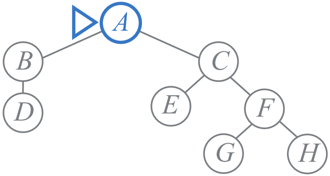


goal: D
limit: 1
node: -
frontier: A
result: failure

Depth-Limited Search: Algorithm (Step 2a)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

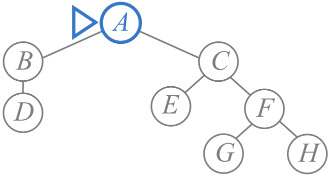


goal: D
limit: 1
node: A
frontier: ∅
result: failure

Depth-Limited Search: Algorithm (Step 2b)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

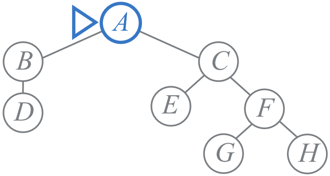


goal: D
limit: 1
node: A
frontier: B
result: failure

Depth-Limited Search: Algorithm (Step 2c)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

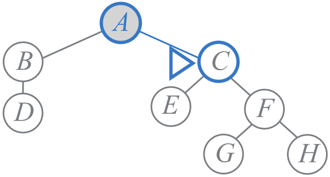


goal: D
limit: 1
node: A
frontier: B, C
result: failure

Depth-Limited Search: Algorithm (Step 3a)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

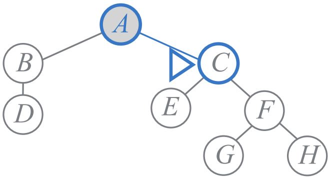


goal: D
limit: 1
node: C
frontier: B
result: failure

Depth-Limited Search: Algorithm (Step 3b)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

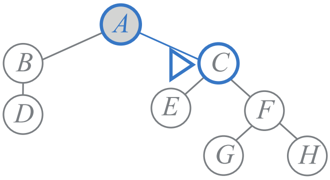


goal: D
limit: 1
node: C
frontier: B, E
result: failure

Depth-Limited Search: Algorithm (Step 3c)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

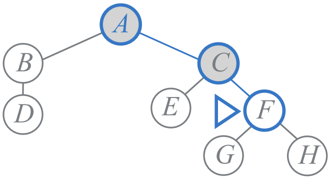


goal: D
limit: 1
node: C
frontier: B, E, F
result: failure

Depth-Limited Search: Algorithm (Step 4a)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

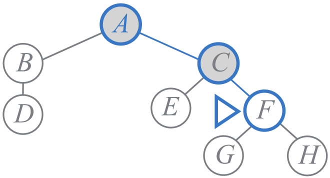


goal: D
limit: 1
node: F
frontier: B, E
result: failure

Depth-Limited Search: Algorithm (Step 4b)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

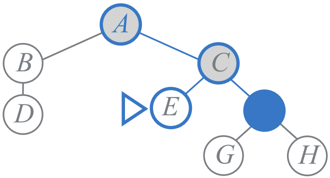


goal: D
limit: 1
node: F
frontier: B, E
result: cutoff

Depth-Limited Search: Algorithm (Step 5a)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

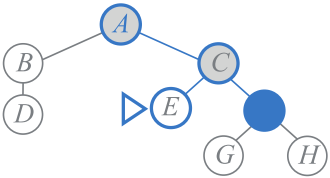


goal: D
limit: 1
node: E
frontier: B
result: cutoff

Depth-Limited Search: Algorithm (Step 5b)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

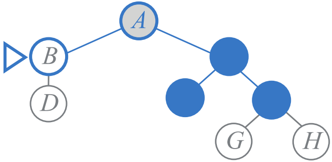


goal: D
limit: 1
node: E
frontier: B
result: cutoff

Depth-Limited Search: Algorithm (Step 6a)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

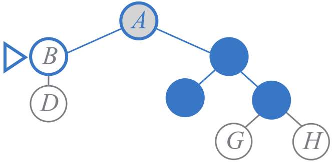


goal: D
limit: 1
node: B
frontier: ∅
result: cutoff

Depth-Limited Search: Algorithm (Step 6b)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

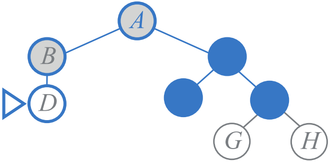


goal: D
limit: 1
node: B
frontier: D
result: cutoff

Depth-Limited Search: Algorithm (Step 7a)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*

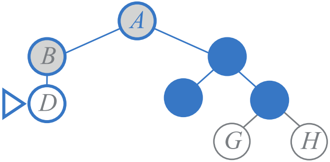


goal: D
limit: 1
node: D
frontier: ∅
result: cutoff

Depth-Limited Search: Algorithm (Step 7b)

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

frontier ← a LIFO queue (stack) with `Node`(*problem*.`Initial-State`) as the only element
result ← *failure*
while not `Is-Empty`(*frontier*) **do**
  node ← `Pop`(*frontier*)
  if *problem*.`Is-Goal`(*node*.`State`) **then return** `Solution`(*node*)
  if *node*.`Depth` > *limit* **then** *result* ← *cutoff*
  else if not `Is-Cycle`(*node*) **then**
    for each *child* **in** `Expand`(*problem*,*node*) **do**
      frontier ← `Add`(*child*,*frontier*)
return *result*



goal state D found!
limit: 1
node: D
frontier: ∅
result: cutoff

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?
- **Optimality**: No, if $l > d$. Why?
- **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: $\mathcal{O}(bl)$.

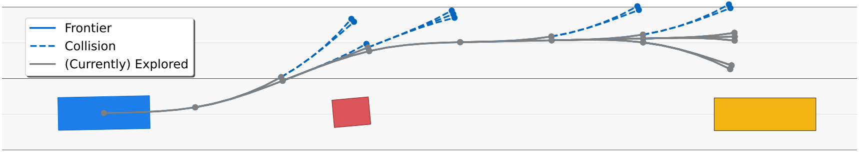
Depth-Limited Search: CommonRoad Example

- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Depth limit: 7
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Depth-Limited Search: CommonRoad Example

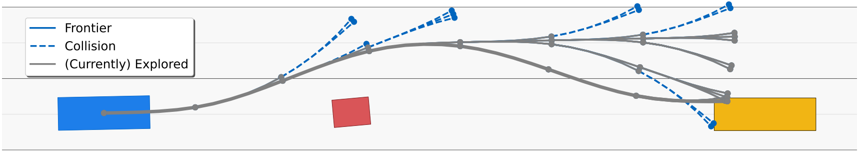
- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Depth limit: 7
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Depth-Limited Search: CommonRoad Example



- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Depth limit: 7
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Depth-Limited Search: CommonRoad Example



- Concatenation of motion primitives.
- Note: Colliding states are not further considered (collision with obstacle or out of road boundary).
- Depth limit: 7
- Link to tutorial: cr_uninformed_search_tutorial.ipynb.

Iterative Deepening Search: Idea and Algorithm

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

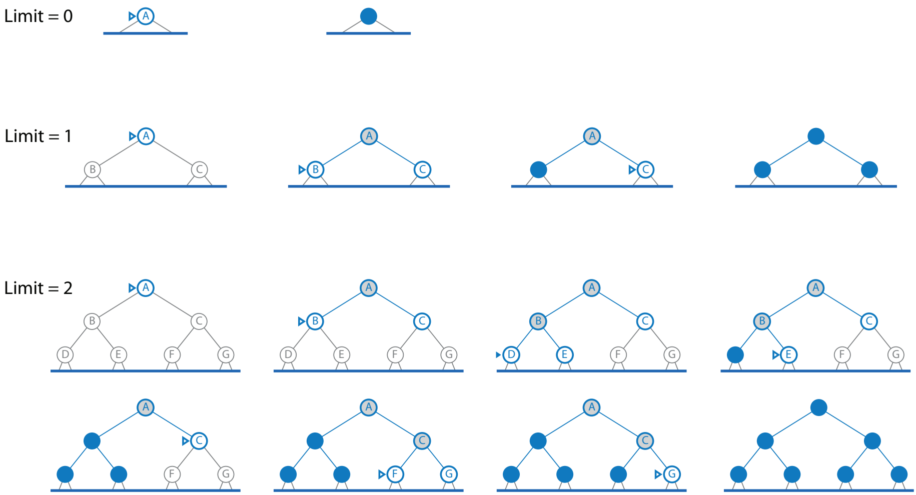
Solution:
Use depth-limited search and iteratively increase the depth limit l.

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

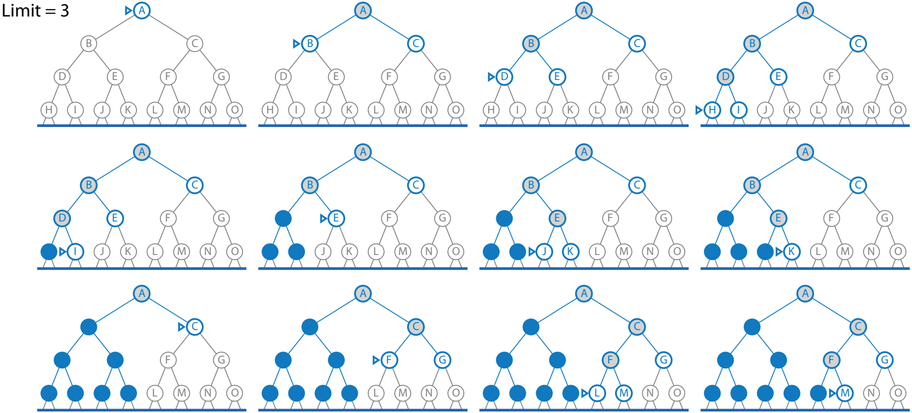
for *depth*$= 0$ **to** ∞ **do**
  result $\leftarrow$ `Depth-Limited-Search`(*problem*, *depth*)
  if *result* $\neq$ cutoff **then return** *result*

Iterative Deepening Search: Example (1)

(jupyter UninformedSearch.ipynb)



Iterative Deepening Search: Example (2)



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?

Comparison of Computational Effort

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**:

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

- **Iterative deepening search**:

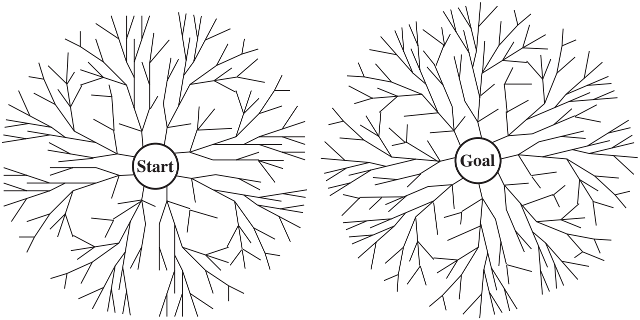
$$\begin{aligned} & (d)b + (d-1)b^2 + \ldots + (1)b^d \\ =& 5 \cdot 10 + 4 \cdot 100 + 3 \cdot 1000 + 2 \cdot 10,000 + 100,000 = 123,450 \end{aligned}$$

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

Is it always possible to use bidirectional search?

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, if $l \geq d$	Yes^a
Optimal?	Yes^c	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

Summary

- A well-defined search problem consists of: **states**, the **initial state**, **actions**, a **transition model**, a **goal test**, and an **action cost** function.
- Search algorithms are typically judged by **completeness**, **optimality**, **time complexity**, and **space complexity**.
- **Breadth-first search**: expands the shallowest nodes first; 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; special instance of **Best-First Search**.
- **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.