Artificial Intelligence

1. Uninformed Search

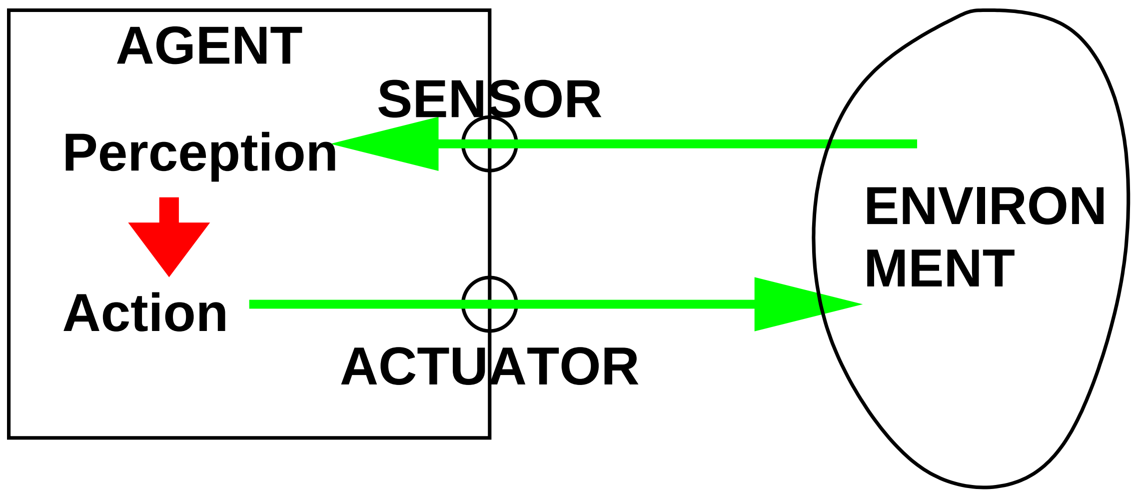
Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute of Economics and Information Systems
& Institute of Computer Science
University of Hildesheim
http://www.ismll.uni-hildesheim.de

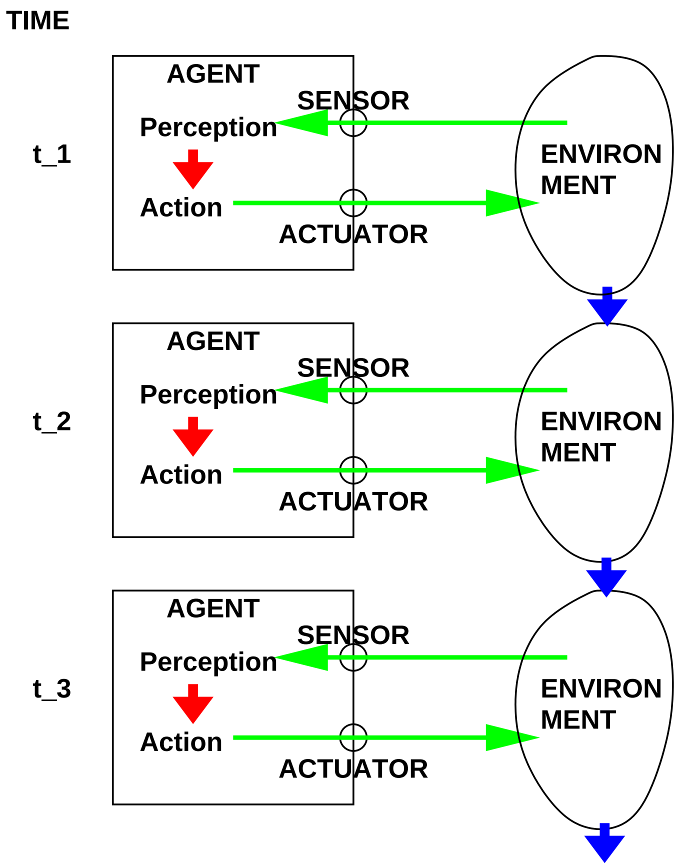
Artificial Intelligence

Lars Schmidt-Thieme, Information Systems and Machine Learning Lab (ISMLL), University of Hildesheim, Germany,
Course on Artificial Intelligence, summer term 2008 1/44

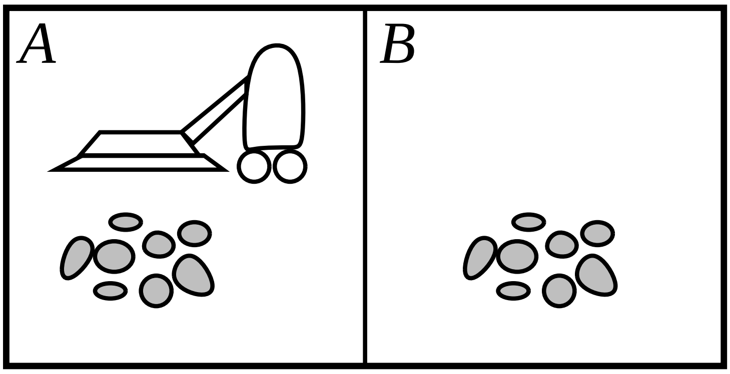
Agent, Environment, Perceptions, and Actions



Perception Sequence and Action Sequence



Silly Example: The vacuum-cleaner world



Perceptions: pairs of

- location of the vacuum-cleaner: square A or square B
- content at that location: clean or dirty

Actions: move left, move right, suck dirt, do nothing.

Silly Example: The vacuum-cleaner world

Perception sequence	action sequence
(A, clean)	right
(A, dirty)	suck
(B, clean)	left
(B, dirty)	suck
(A, clean), (A, clean)	?

Silly Example: The vacuum-cleaner world

Perception sequence	action sequence
(A, dirty)	suck
(A, clean)	right
(B, dirty)	suck
(B, clean)	left
(A, clean), (B, clean)	noop
(B, clean), (A, clean)	noop

Components of Environments

Environements consist of four components
(so-called “PEAS” model):

Performance measure:
describes successful behavior of an agent; the goal.

Environment:
describes what other entities there are to interact with.

Actuators:
describes the actions an agent can take and how they influence the environment.

Sensors:
describes the perceptions available to an agent.

Properties of Environments (1/2)

deterministic – stochastic:

deterministic: the next state is completely determined by the previous state and the action.

static – dynamic:

static: the state of the environment does not change while the agent deliberates,
e.g., a turn-based game.

fully observable – partially observable:

fully observable: all properties of the true state that are relevant to take the optimal action are perceived, e.g., in chess.

partially observable: e.g., the vacuum world with information just about the actual location.

Properties of Environments (2/2)

discrete – continuous:

discrete time: e.g., measured in steps.

discrete states: e.g., counts; locations on a grid; etc.

discrete perceptions: e.g., counts; locations on a grid; etc. (same as for states).

discrete actions: e.g., just steering left/right (but not by a continuous angle).

Episodic – sequential:

episodic: actions do only influence the next state, but not any later states.

Single agent – multiagent:

multiagent: several agents act in the environment.
(cooperative vs. competitive scenarios)

Problems

A **problem** consists of six components (here 1–4):

super state space: set $X^{\#}$
a set of entities that describe the state of the environment, i.e., the actual configuration at a given point in time.

action space: set A
a set of entities that describe the actions that an agent may perform.

initial state: element $x_0 \in X^{\#}$
the state the agent starts in.

successor function: partial function $\text{succ} : X^{\#} \times A \rightarrow X^{\#}$
triples x, a, x' consisting of
– previous state x,
– possible action a in that state and
– follow up state x'
(for deterministic environments)

Problems / State space

Initial states and successor function implicitely define the **state space** X by enumeration:

$$X := \bigcup_{n \in \mathbb{N}} \text{succ}^n(x_0) \subseteq X^{\#}$$

where succ^n denotes the n-th power of $\text{succ}(\cdot, A)$, i.e.,

$$\text{succ}^0(x) = x,$$
$$\text{succ}^1(x) = \text{succ}(x, A) = \bigcup_{a \in A} \text{succ}(x, a),$$
$$\text{succ}^2(x) = \text{succ}(\text{succ}(x, A), A) = \bigcup_{a \in A} \bigcup_{a' \in A} \text{succ}(\text{succ}(x, a'), a) \text{etc.}$$

Obviously,

$$x_0 \in X$$

and succ can be restricted to

$$\text{succ} \subseteq X \times A \times X$$

Problems

A **problem** consists of six components (here 5–6):

goal test: $g : X \rightarrow \{0, 1\}$

a function that evaluates if a given state is a goal or not.

Sometimes the set of goals $g^{-1}(1)$ is enumerated explicitely, e.g., $g^{-1}(1) = \{\text{In(Bucharest)}\}$.

path costs: $c : (A \times X)^* \rightarrow \mathbb{R}$

the cost of performing the sequence of actions $a_1, a_2, \ldots, a_n$ to move from x_0 to x_1, from x_1 to x_2, etc., and finally from x_{n-1} to x_n.

Path costs often are assumed to be just the sum of single step costs:

$$c(a_1, x_1, a_2, x_2, \ldots, a_n, x_n) = \sum_{i=1}^{n} c_{\text{step}}(x_{i-1}, a_i, x_i)$$

Problems / State graph

Problems can be represented as directed graphs with labeled edges:

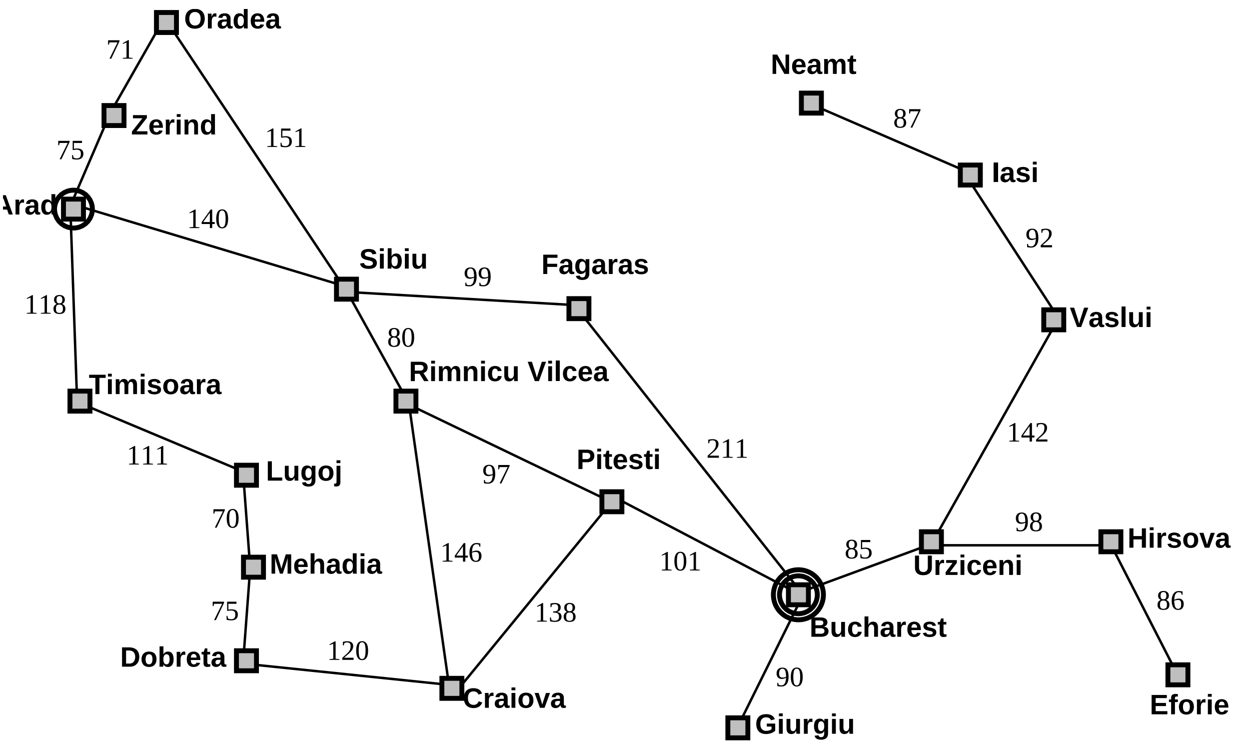
vertices: states X.

edges: there is an edge from vertex x to x' if there is an action a with succ$(x, a) = x'$.

edge labels: edges are labeled twofold:
– with the action a and
– with the step costs $c(x, a, x')$.

If from each state each successor state can be reached by at most one action, the action label often is omitted
(as it is fully determined by the two states).

Problems / State graph / Example



Solutions

A **path** in the state space can be described either by a sequence

$$(a_1, x_1, a_2, x_2, \ldots, a_n, x_n) \in (A \times X)^*, \quad \text{with succ}(x_{i-1}, a_i) = x_i, \quad i = 1, \ldots, n$$

or equivalently by a pure action sequence

$$(a_1, a_2, \ldots, a_n) \in A^*$$

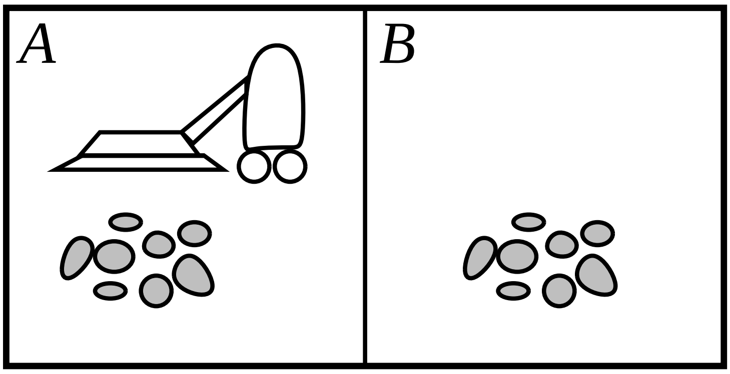
where

$$x_i := \text{succ}(x_{i-1}, a_i), \quad i = 1, \ldots, n$$

A **solution** is a path that reaches a goal, i.e., with $g(x_n) = 1$.

An **optimal solution** is a solution with smallest cost $c(a_1, x_1, a_2, x_2, \ldots, a_n, x_n)$ among all solutions.

Examples / Vacuum cleaner

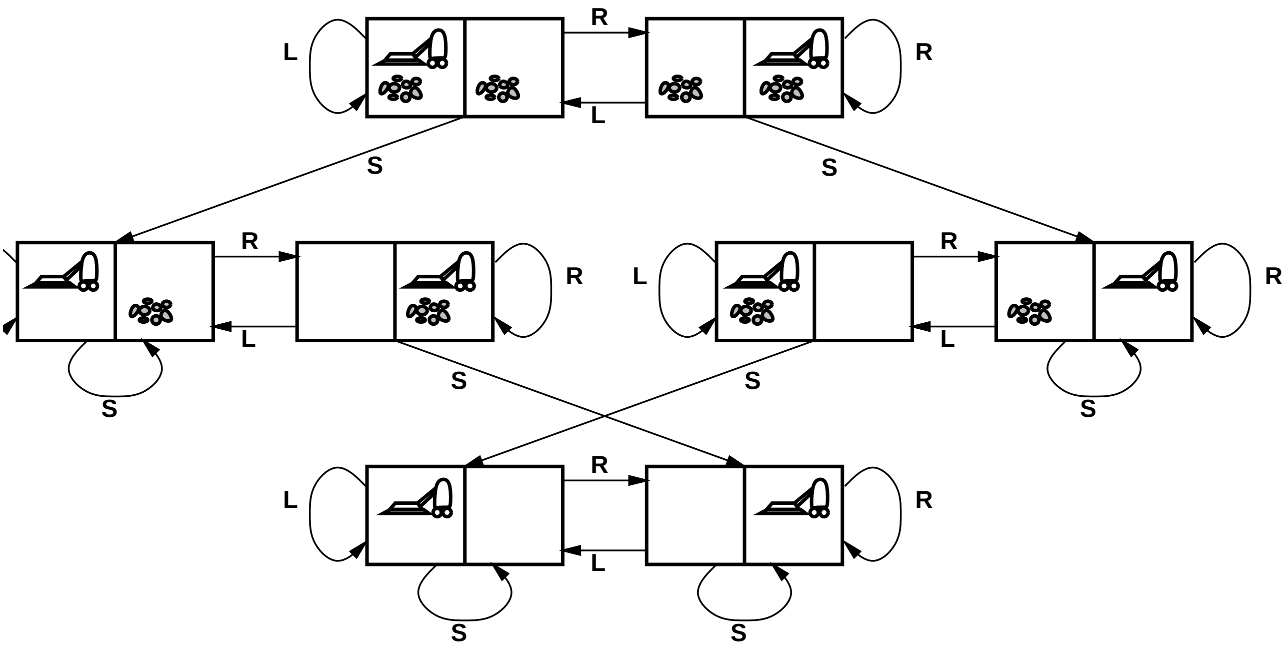


state space $X := \{A, B\} \times \{\text{dirty}, \text{clean}\}^{\{A,B\}}, \quad |X| = 8$.
initial state any.
successor function

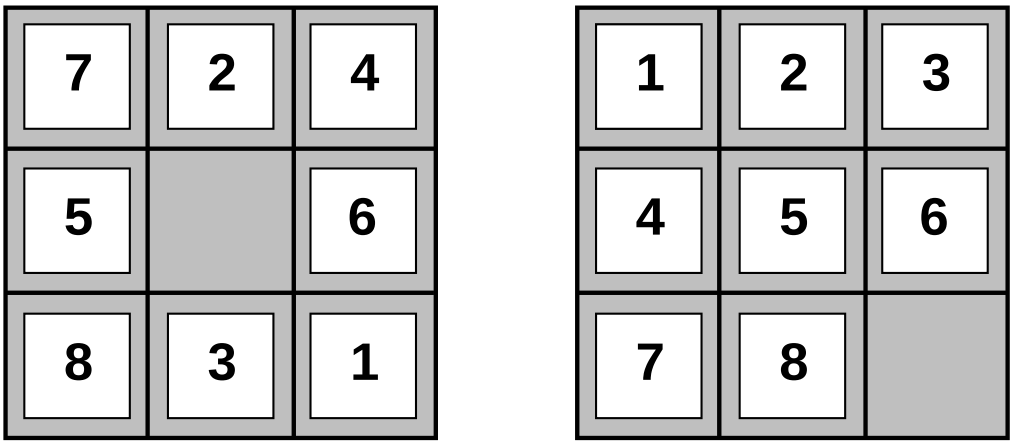
$$\text{succ}((A, \{(A, \text{dirty}), (B, \text{dirty})\}), \text{suck}) = (A, \{(A, \text{clean}), (B, \text{dirty})\})$$

etc. (see next slide).
goal function: $g((*, \{(A, \text{clean}), (B, \text{clean})\})) = 1$, else 0.
path cost: $c(x, a, x') = 1$

Examples / Vacuum cleaner



Examples / 8-puzzle



Start State **Goal State**

state space $X := \{f : \{1, 2, \ldots, 8\} \to \{1, 2, \ldots, 9\} \mid f$ injective$\}$.
initial state any.
successor function effect of moving the blank (see next slide).
goal function: $g($designated goal state$) = 1$, else 0.
path cost: $c(x, a, x') = 1$

Examples / 8-puzzle

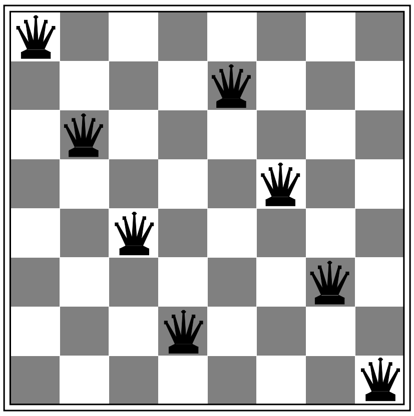
$$\text{succ}(\begin{pmatrix} 7 & 2 & 4 \\ 5 & & 6 \\ 8 & 3 & 1 \end{pmatrix}, \text{move blank left}) = \begin{pmatrix} 7 & 2 & 4 \\ & 5 & 6 \\ 8 & 3 & 1 \end{pmatrix}$$

Examples / 8-puzzle

8-puzzle is an instance of the **sliding-block puzzle** class,
a NP-complete problem class.

name	board	reachable states	difficulty
8-puzzle	3×3	$9!/2 = 181,440$	solved easily
15-puzzle	4×4	$\approx 1.3 \cdot 10^{18}$	solved in a few milliseconds
24-puzzle	5×5	$\approx 10^{25}$	difficult to solve

Examples / 8-queens problem



state space

$$X := \{x \subset \{1, \ldots, 64\} \,|\, |x| \leq 8\}, \quad |X| = \binom{64}{8} = 4.4 \cdot 10^9$$

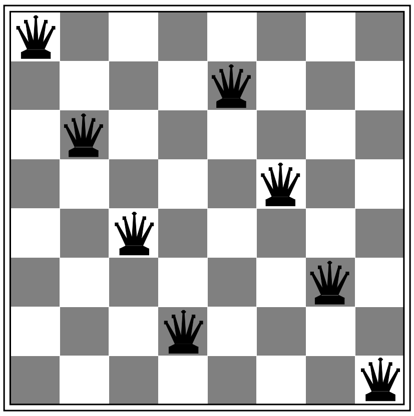
initial state $x = \emptyset$.

successor function add a queen to any empty square.

goal function: goal reached if 8 queens on the board, none attacked.

path cost: $c(x, a, x') = 1$

Examples / 8-queens problem



A better problem formulation:

state space n queens ($n = 0, \ldots, 8$) in the n left-most columns, one per column, non attacked. $|X| = 2057$.

initial state $x = \emptyset$.

successor function add a queen to the left-most empty column, not attacked.

goal function: goal reached if 8 queens on the board, none attacked.

path cost: $c(x, a, x') = 1$

1. The Agent Metaphor

2. Problem Descriptions

3. Uninformed Tree Search

4. Uninformed Graph Search

The Problem (1/3)

Algorithmics / Graph theory:

Given a directed graph $G := (V, E)$ with edge weights $w : E \rightarrow \mathbb{R}$ and two vertices $x, y \in V$, find a shortest path from x to y, i.e., a path $P \in V^*$ with $P_1 = x, P_2 = y$ and

$$w(P) := \sum_{i=1}^{n-1} w(P_i, P_{i+1})$$

minimal among all paths from x to y.

Artificial Intelligence:

If from each state any other state can be reached by at most one action and costs decompose in single step costs, then

$$\begin{aligned} V :=& X \quad \text{(the states)} \\ E :=& \{(x, y) \in X^2 \mid \exists a \in A : \text{succ}(x, a) = y\} \\ w(x, y) :=& \text{cost}(x, a, y) \quad (a \text{ unique with succ}(x, a) = y) \\ x :=& x_0 \quad \text{(initial state)} \\ y :=& \text{any } x \in X \text{ with } g(x) = 1 \end{aligned}$$

The Problem (2/3)

But:

- X often is not finite, so it cannot be stored, but relevant portions must be constructed by succ recursively.

- $g^{-1}(1)$ may not be easy to compute (although for each specific x it may be easy to check if $g(x) = 1$, e.g., check-mate).

The Problem (3/3)

For this section, assume:

Each state can be reached by at most one sequence of actions.

I.e., the search space is a tree.

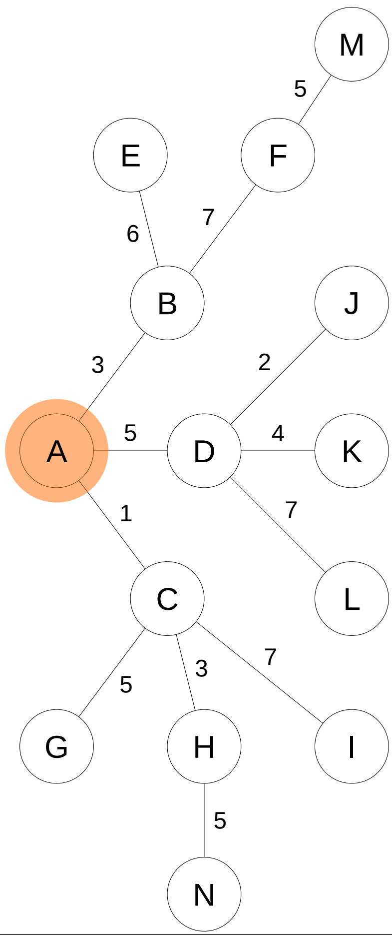
Breadth-First Search

Idea:

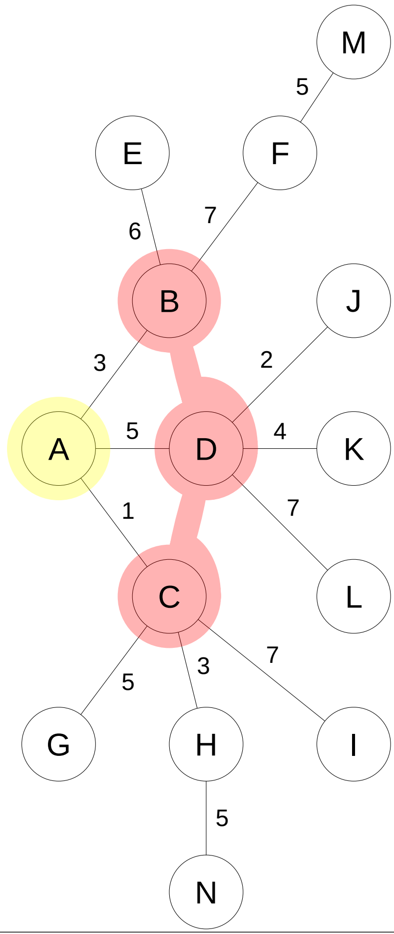
- start with initial state as border.
- iteratively replace border by all states reachable from the old border.

```
breadth-first-search(X, succ, border, g) :
newborder := ∅
for x ∈ border do
    for y ∈ succ(x, A) do
        if g(y) = 1
            return y
        else
            newborder := newborder ∪ {y}
        fi
    od
od
if newborder ≠ ∅
    return breadth-first-search(X, succ, newborder, g)
else
    return ∅
fi
```

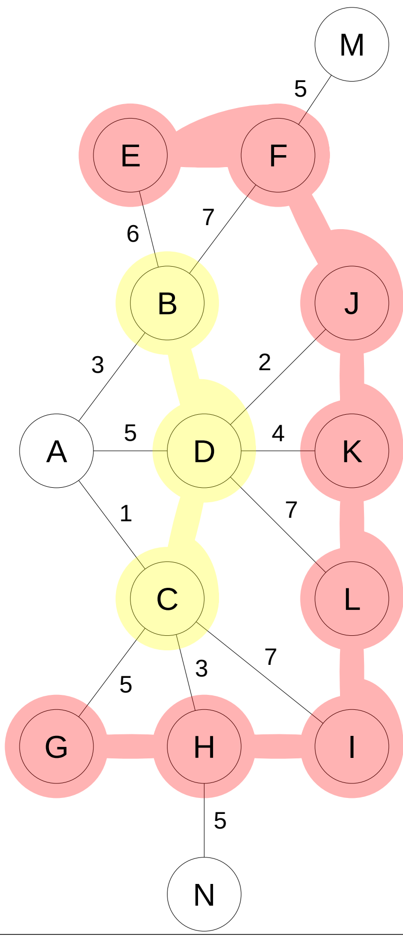

Breadth-First Search / Example



Breadth-First Search / Example



Breadth-First Search / Example



Breadth-First Search

```
breadth-first-search(X, succ, border, g) :
newborder := ∅
for x ∈ border do
    for y ∈ succ(x, A) do
        if g(y) = 1
          return y
        else
             newborder := newborder ∪ {y}
        fi
    od
od
if newborder ≠ ∅
  return breadth-first-search(X, succ, newborder, g)
else
    return ∅
fi
```

```
breadth-first-search(X, succ, x_0, g) :
border := {x_0}
while border ≠ ∅ do
        x := border[1]
        if g(x) = 1
          return x
        fi
        for y ∈ succ(x, A) do
            append(border, y);
        od
        remove(border, x)
od
return ∅
```


Characteristics of Algorithms

In algorithmics, the complexity of (shortest path) algorithms is measured in steps as function of the number of vertices and edges (big-O notation).

For problems with infinite number of vertices or edges this is not possible.

Use instead:

maximum branching factor b:
maximum number of successors of a state.

depth of least-cost solution d:
length of least cost path to a goal state.

maximum depth of state space m
length of longest path, also called diameter; evtl. ∞.

Characteristics of Algorithms

Characterize by:

Completeness
does the algorithm always find a solution if one exists?

Optimality
does the algorithm always find an optimal solution?

Time complexity
size of the visited part of the search tree

Space complexity
size of the search tree in memory

Breadth-First Search

Completeness
yes (if b is finite)

Optimality
no (unless all step costs are the same, e.g., 1)

Time complexity
$1 + b + b^2 + \cdots + b^d + b(b^d - 1) = O(b^{d+1})$

Space complexity
same as time complexity as whole search tree is kept in memory.

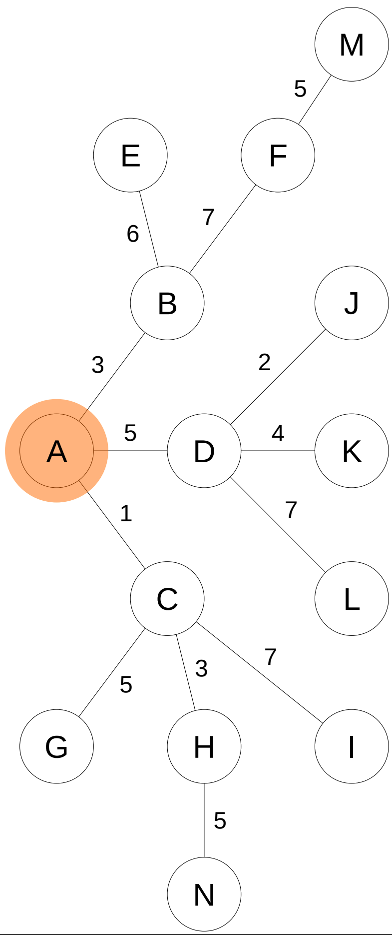
Uniform Cost Search

Idea:

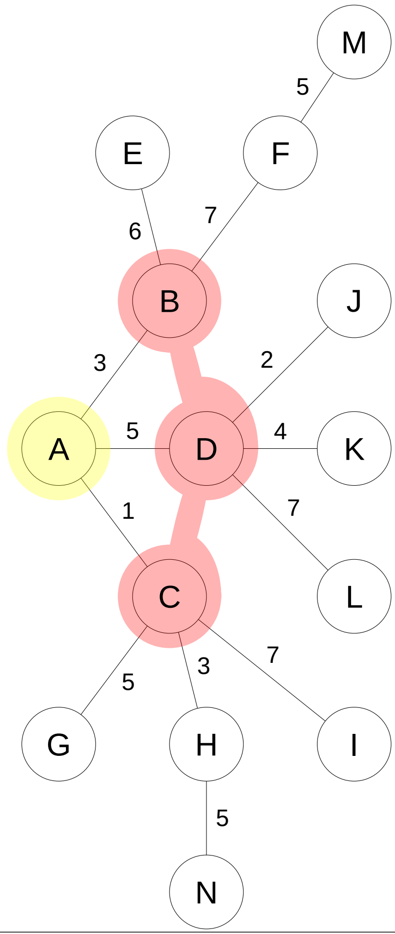
- as breadth-first search.
- but visit state with minimal path cost first.

```
uniform-cost-search(X, succ, cost, x_0, g) :
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
    x := argmin_{x∈border} c(x)
    if g(x) = 1
       return x
    fi
    for y ∈ succ(x, A) do
        border := border ∪ {y}
        c(y) := c(x) + cost(x, y)
    od
    border := border \ {x}
od
return ∅
```

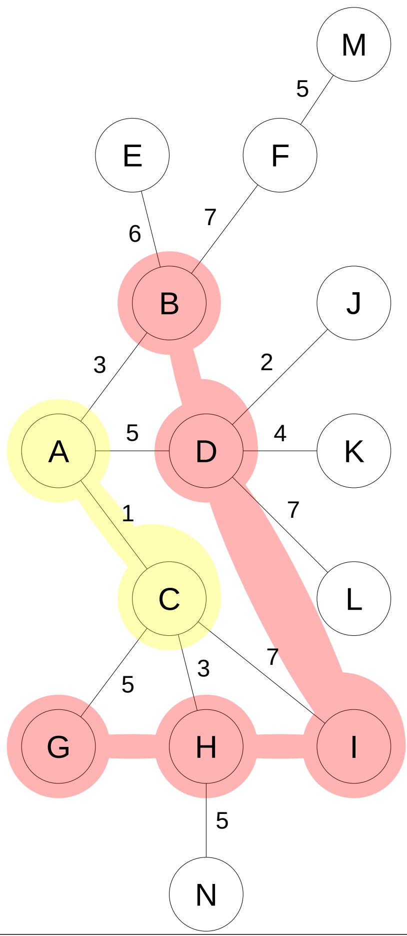

Uniform Cost Search / Example



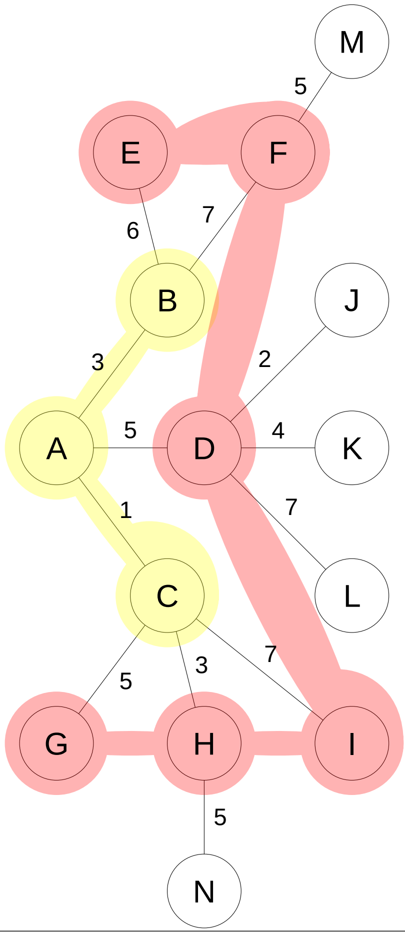
Uniform Cost Search / Example



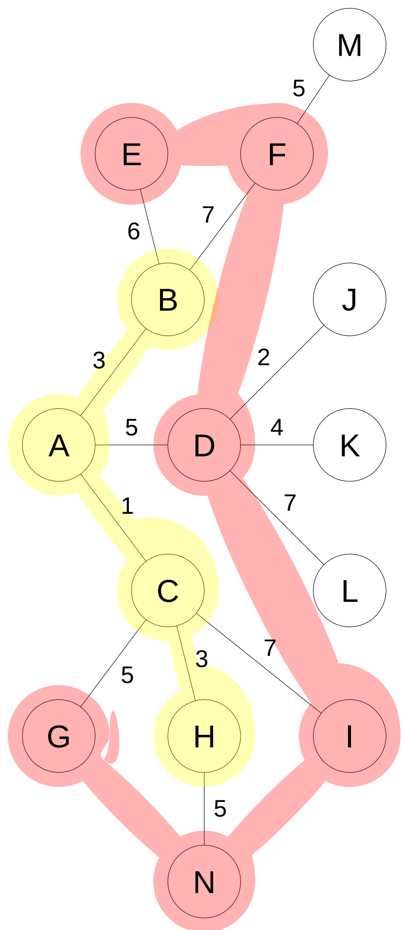
Uniform Cost Search / Example



Uniform Cost Search / Example



Uniform Cost Search / Example



Uniform Cost Search

Completeness

yes (if step costs are $\geq \epsilon > 0$).

Optimality

yes

Time complexity

$O(b^{1+\lfloor \frac{\text{cost}(P^*)}{\epsilon} \rfloor})$, where P^* is an optimal solution.

Space complexity

same as time complexity as whole search tree is kept in memory.

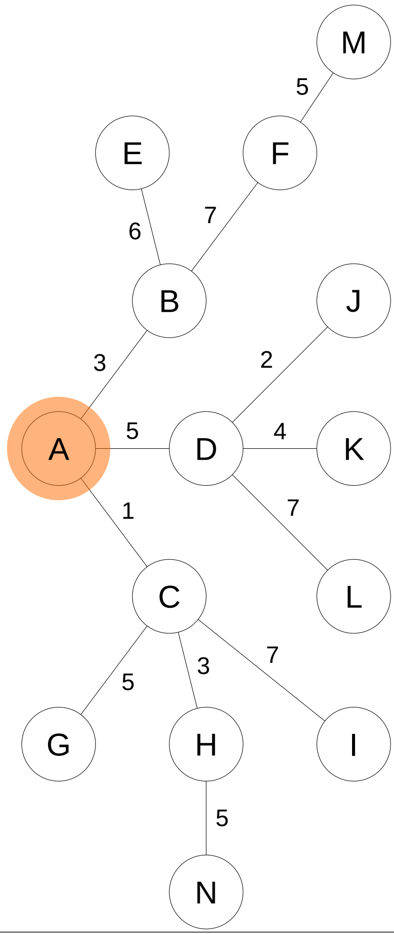
Depth-First Search

Idea:

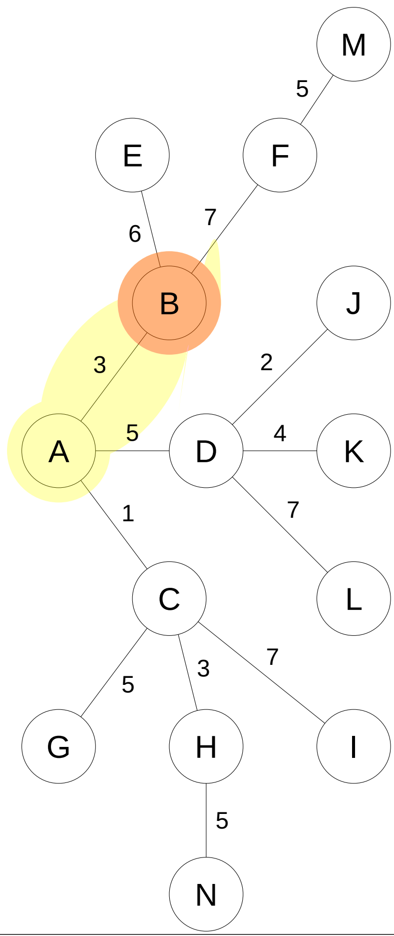
- start with initial state.
- iteratively visit successors one by one.

```
depth-first-search(X, succ, x_0, g) :
for y ∈ succ(x_0, A) do
    if g(y) = 1
        return y
    else
            depth-first-search(X, succ, y, g);
    fi
od
return ∅
```

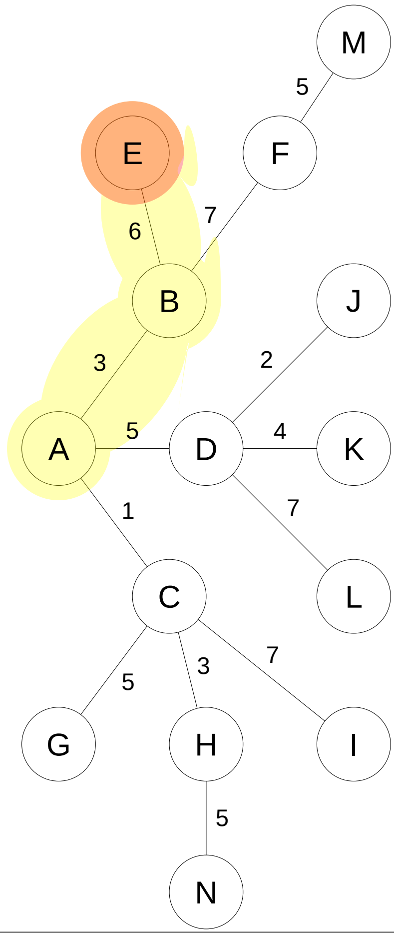
Depth First Search / Example



Depth First Search / Example

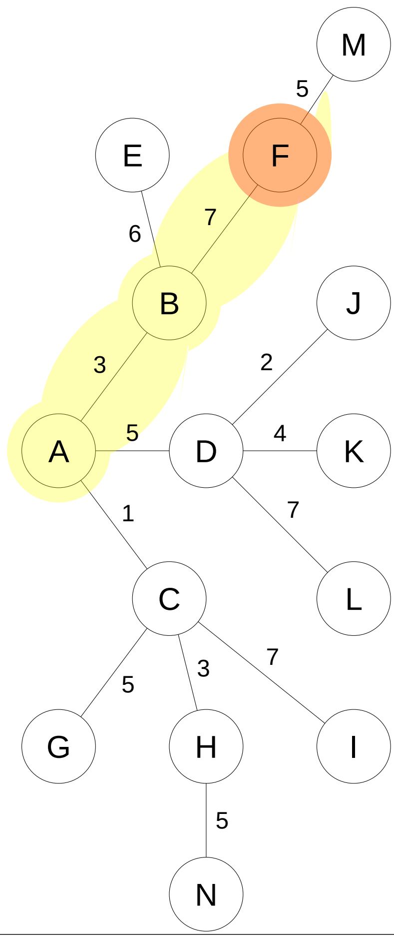


Depth First Search / Example

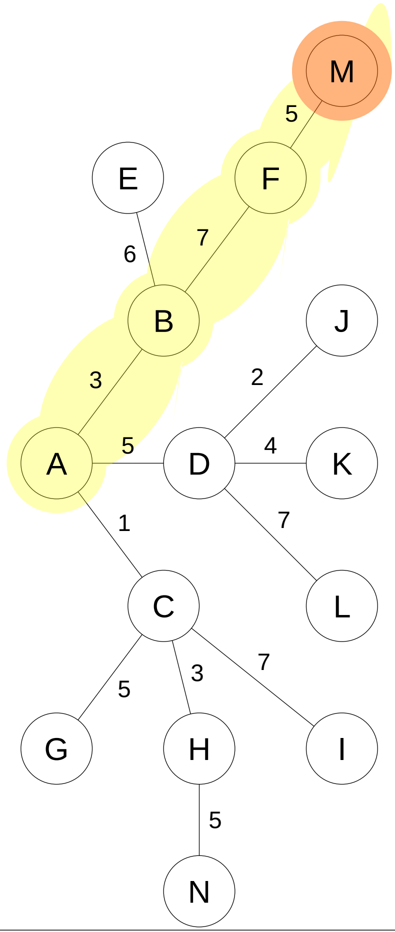


Artificial Intelligence / 3. Uninformed Tree Search

Depth First Search / Example

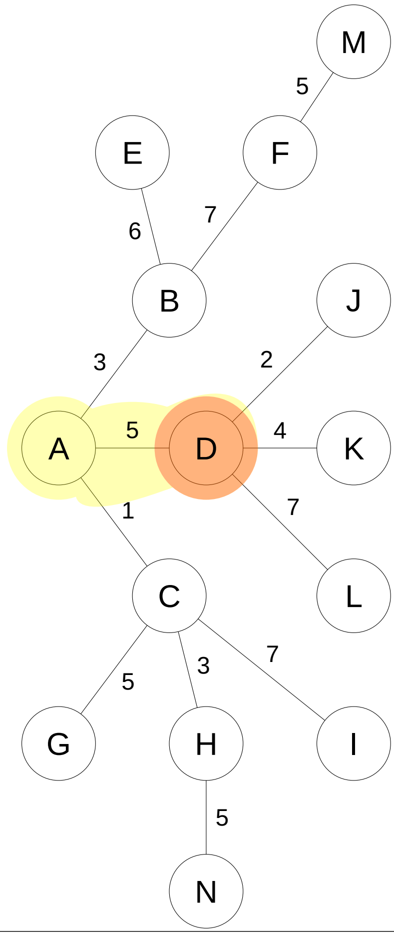


Depth First Search / Example

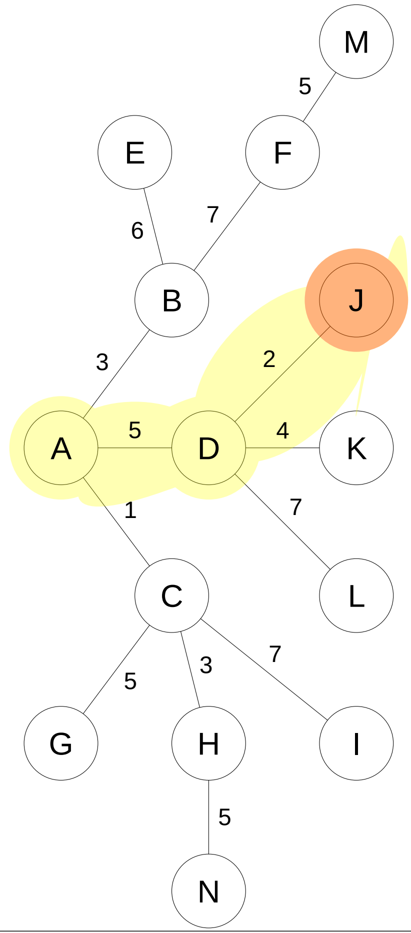


Artificial Intelligence / 3. Uninformed Tree Search

Depth First Search / Example



Depth First Search / Example



Depth-First Search

depth-first-search(X, succ, x_0, g) :
for $y \in \text{succ}(x_0, A)$ **do**
  if $g(y) = 1$
    return y
  else
    depth-first-search(X, succ, y, g);
  fi
od
return $\emptyset$

depth-first-search(X, succ, x_0, g) :
border $:= \{x_0\}$
while border $\neq \emptyset$ **do**
  $x :=$ border$[1]$
  if $g(x) = 1$
    return x
  fi
  for $y \in \text{succ}(x, A)$ **do**
    insert-at-beginning(border, y);
  od
  remove(border, x)
od
return $\emptyset$

Depth First Search

Completeness
no (if $m = \infty$, e.g., due to loops).

Optimality
no

Time complexity
$O(b^m)$ — bad, if $m >> d$, but great for dense solutions.

Space complexity
$O(bm)$.

Depth-Limited Search

Idea:

- as depth-first search.
- stop at given maximum depth maxdepth.

```
depth-limited-search(X, succ, x_0, g, maxdepth) :
limit-active := false
for y ∈ succ(x_0, A) do
    if g(y) = 1
        return y
    elsif maxdepth > 0
        depth-limited-search(X, succ, y, g, maxdepth − 1);
    else
        limit-active := true
    fi
od
if limit-active
    return "cutoff"
else
    return ∅
fi
```

Depth-Limited Search

Completeness
no (if $m = \infty$, e.g., due to loops or if $d >$ maxdepth).

Optimality
no

Time complexity
$O(b^{\text{maxdepth}})$.

Space complexity
$O(b \cdot \text{maxdepth})$.

Iterative Deepening Search

Idea:

- as depth-limited search.
- but repeat for increasing maximal depth maxdepth.

```
iterative-deepening-search(X, succ, x_0, g, maxdepth) :
for d = 1 ... maxdepth do
    P := depth-limited-search(X, succ, x_0, g, d);
    if P ≠ "cutoff"
        return P
    fi
od
return "cutoff"
```

Iterative Deepening Search

Completeness
yes

Optimality
no (unless all step costs are equal, e.g., 1; but can be modified).

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

Space complexity
$O(bd)$

Uniform Cost Search / Explicit branch bookkeeping

```
uniform-cost-search(X, succ, cost, x_0, g) :
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
      x := argmin_{x∈border} c(x)
      if g(x) = 1
        return x
      fi
      for y ∈ succ(x, A) do
          border := border ∪ {y}
          c(y) := c(x) + cost(x, y)
      od
      border := border \ {x}
od
return ∅
```

If succ is expensive to invert (or not possible to invert, because the search space is not a tree), branches must be stored explicitely.

```
uniform-cost-search(X, succ, cost, x_0, g) :
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
      x := argmin_{x∈border} c(x)
      if g(x) = 1
        return branch(x, previous)
      fi
      for y ∈ succ(x, A) do
          border := border ∪ {y}
          c(y) := c(x) + cost(x, y)
          previous(y) := x
      od
      border := border \ {x}
od
return ∅

branch(x, previous) :
P := ∅
while x ≠ ∅ do
      insert-at-beginning(P, x)
      x := previous(x)
od
return P
```


Uniform Cost Search / Duplicate states

If duplicate states can occur
(i.e., there are several paths to the same state,
i.e., the search space is not a tree),
and if still a tree search should be applied,
states cannot be used as index anymore,
but have to be wrapped in “nodes”.

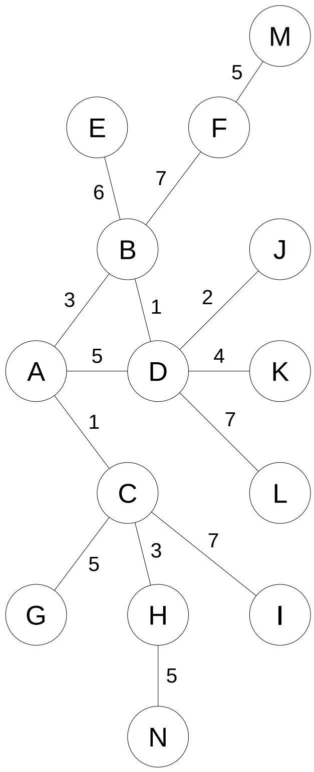
The same modifications have to be applied to all other search algorithms.

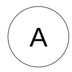
Uniform Cost Search / Duplicate states

```
uniform-cost-search(X, succ, cost, x_0, g) :
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
      x := argmin_{x∈border} c(x)
      if g(x) = 1
         return branch(x, previous)
      fi
      for y ∈ succ(x, A) do
          border := border ∪ {y}
          c(y) := c(x) + cost(x, y)
          previous(y) := x
      od
      border := border \ {x}
od
return ∅

branch(x, previous) :
P := ∅
while x ≠ ∅ do
      insert-at-beginning(P, x)
      x := previous(x)
od
return P
```

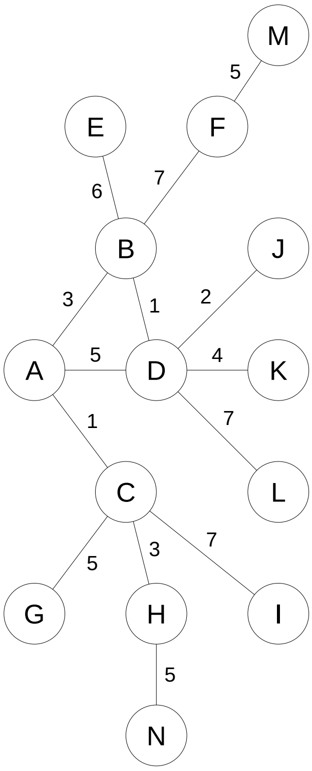
```
uniform-cost-search(X, succ, cost, x_0, g) :
N := new node(state = x_0, c = 0, previous = ∅)
border := {N}
while border ≠ ∅ do
      N := argmin_{N∈border} N.c
      if g(N.state) = 1
         return branch(N)
      fi
      for y ∈ succ(N.state, A) do
          N' := new node(state = y,
                         c = N.c + cost(N.state, y),
                         previous := N)
          border := border ∪ {N'}
      od
      border := border \ {N}
od
return ∅

branch(N) :
P := ∅
while N! = ∅ do
      insert-at-beginning(P, N.state)
      N := N.previous
od
return P
```

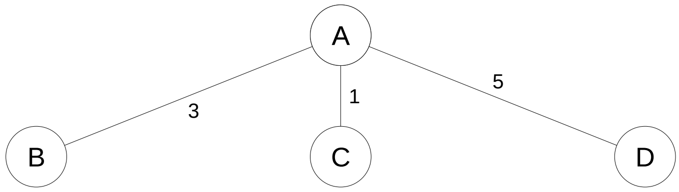

Several paths blow up the search tree



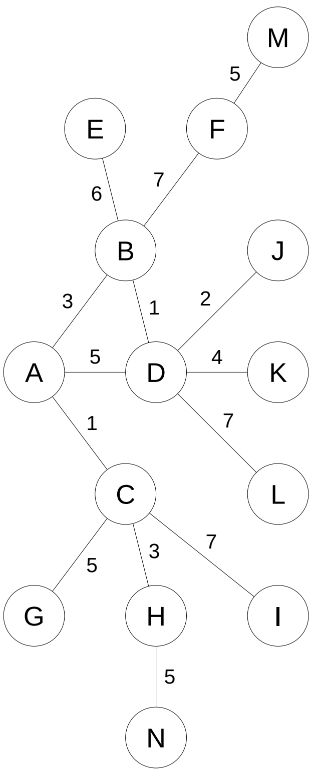


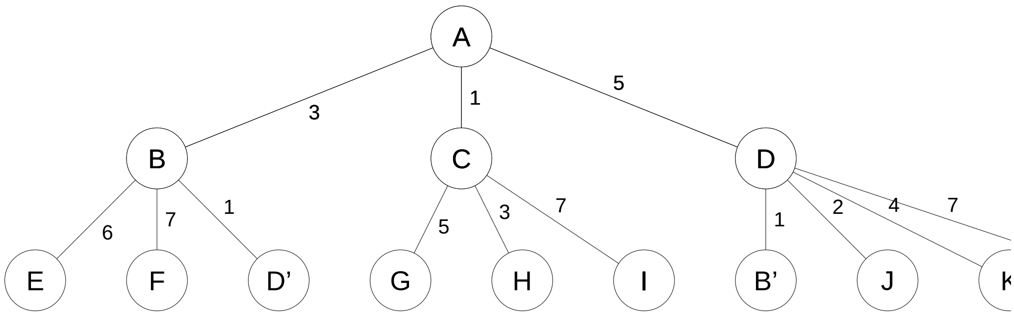
Several paths blow up the search tree





Several paths blow up the search tree





Closed list

The tree search algorithms must be modified s.t. they keep track of all the nodes visited so far (so-called **closed list**).

If the current state is already in the closed list, it is discarded instead of expanded.

This means that all algorithms have to keep the whole visited part of the state space in memory, i.e., the space complexity always is the one of breadth first search..

Uniform Cost Search in Graph State Spaces (1/2)

```
uniform-cost-search(X, succ, cost, x_0, g) :
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
        x := argmin_{x∈border} c(x)
        if g(x) = 1
          return x
        fi
        for y ∈ succ(x, A) do
            border := border ∪ {y}
            c(y) := c(x) + cost(x, y)
        od
        border := border \ {x}
od
return ∅
```

```
uniform-cost-search-graph(X, succ, cost, x_0, g) :
visited := ∅
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
        x := argmin_{x∈border} c(x)
        if g(x) = 1
          return x
        fi
        for y ∈ succ(x, A) do
            if y ∉ visited
              border := border ∪ {y}
              if c(x) + cost(x, y) < c(y)
                c(y) := c(x) + cost(x, y)
                previous(y) := x
              fi
            fi
        od
        border := border \ {x}
        visited := visited ∪ {x}
od
return ∅
```

Uniform Cost Search in Graph State Spaces (2/2)

```
uniform-cost-search-graph(X, succ, cost, x_0, g) :
visited := ∅
border := {x_0}
c(x_0) := 0
while border ≠ ∅ do
    x := argmin_{x∈border} c(x)
    if g(x) = 1
      return x
    fi
    for y ∈ succ(x, A) do
        if y ∉ visited
          border := border ∪ {y}
          if c(x) + cost(x, y) < c(y)
            c(y) := c(x) + cost(x, y)
            previous(y) := x
          fi
        fi
    od
    border := border \ {x}
    visited := visited ∪ {x}
od
return ∅
```

```
uniform-cost-search-graph(X, succ, cost, x_0, g) :
notvisited := X
c(x) := { 0,  if x = x_0
        { ∞,  else
while notvisited ≠ ∅ do
    x := argmin_{x∈notvisited} c(x)
    if g(x) = 1
      return x
    fi
    for y ∈ succ(x, A) do
        if c(x) + cost(x, y) < c(y)
          c(y) := c(x) + cost(x, y)
          previous(y) := x
        fi
    od
    notvisited := notvisited \ {x}
od
return ∅
```