Artificial Intelligence
Chapter 3

Stuart RUSSEL

reorganized by L. Aszalós

March 23, 2016

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

Problem-solving agents

Restricted form of general agent:

```
function Simple-Problem-Solving-Agent(percept): action
    static: seq: an action sequence, initially empty
            state: some description of the current world state
            goal: a goal, initially null
            problem: a problem formulation

    state = Update-State(state, percept)
    if seq is empty then
        goal = Formulate-Goal(state)
        problem = Formulate-Problem(state, goal)
        seq = Search(problem)
    action = Recommendation(seq, state)
    seq = Remainder(seq, state)
    return action
```

Problem-solving agents

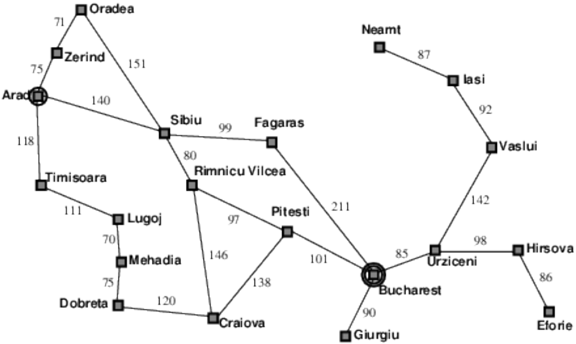
Note: this is *offline* problem solving; solution executed "eyes closed."
Online problem solving involves acting without complete knowledge.

Example: Romania

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

- *Formulate goal*: be in Bucharest
- *Formulate problem*:
  - *states*: various cities
  - *actions*: drive between cities
- *Find solution*: sequence of cities, e.g. Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem types

- *Deterministic, fully observable* $\Longrightarrow$ **single-state problem**

Problem types

- *Deterministic, fully observable* $\Longrightarrow$ **single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence

Problem types

- *Deterministic, fully observable* $\Longrightarrow$ **single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence
- *Non-observable* $\Longrightarrow$ **conformant problem**

Problem types

- *Deterministic, fully observable* $\Longrightarrow$ **single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence
- *Non-observable* $\Longrightarrow$ **conformant problem**
  - Agent may have no idea where it is; solution (if any) is a sequence

Problem types

- *Deterministic, fully observable* $\Longrightarrow$ **single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence
- *Non-observable* $\Longrightarrow$ **conformant problem**
  - Agent may have no idea where it is; solution (if any) is a sequence
- *Nondeterministic* and/or *partially observable* $\Longrightarrow$ **contingency problem**

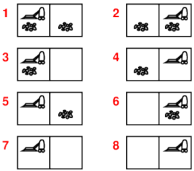
Problem types

- *Deterministic, fully observable* $\Longrightarrow$ **single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence
- *Non-observable* $\Longrightarrow$ **conformant problem**
  - Agent may have no idea where it is; solution (if any) is a sequence
- *Nondeterministic* and/or *partially observable* $\Longrightarrow$ **contingency problem**
  - percepts provide *new* information about current state solution is a **contingent plan** or a **policy** often *interleave* search, execution

Problem types

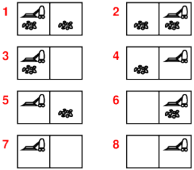
- *Deterministic, fully observable* $\Longrightarrow$ **single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence
- *Non-observable* $\Longrightarrow$ **conformant problem**
  - Agent may have no idea where it is; solution (if any) is a sequence
- *Nondeterministic* and/or *partially observable* $\Longrightarrow$ **contingency problem**
  - percepts provide *new* information about current state solution is a **contingent plan** or a **policy** often *interleave* search, execution
- *Unknown state space* $\Longrightarrow$ **exploration problem** ("online")

Example: vacuum world



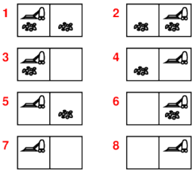
- Single-state, start in $\#5$.

Example: vacuum world



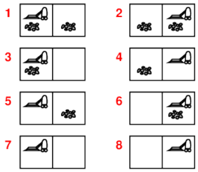
- Single-state, start in #5.
  - [Right,Suck]

Example: vacuum world



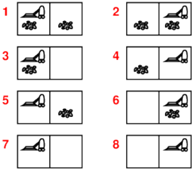
- Single-state, start in #5.
  - [Right,Suck]
- Conformant, start in {1,2,3,4,5,6,7,8}, Rigth goes to {2,4,6,8}

Example: vacuum world



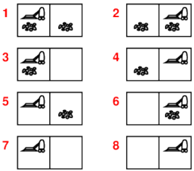
- Single-state, start in #5.
  - [Right,Suck]
- Conformant, start in {1,2,3,4,5,6,7,8}, Rigth goes to {2,4,6,8}
  - [Right,Suck,Left,Suck]

Example: vacuum world



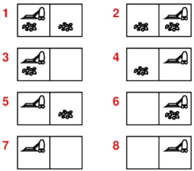
- Single-state, start in #5.
  - [Right,Suck]
- Conformant, start in {1,2,3,4,5,6,7,8}, Rigth goes to {2,4,6,8}
  - [Right,Suck,Left,Suck]
- Contingency, start in #5

Example: vacuum world



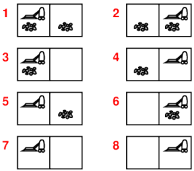
- Single-state, start in #5.
  - [Right,Suck]
- Conformant, start in {1,2,3,4,5,6,7,8}, Rigth goes to {2,4,6,8}
  - [Right,Suck,Left,Suck]
- Contingency, start in #5
  - Murphy law: Suck can dirty a clean carpet;

Example: vacuum world



- Single-state, start in #5.
  - [Right,Suck]
- Conformant, start in {1,2,3,4,5,6,7,8}, Rigth goes to {2,4,6,8}
  - [Right,Suck,Left,Suck]
- Contingency, start in #5
  - Murphy law: Suck can dirty a clean carpet;
  - local sensing: dirt, location only

Example: vacuum world



- Single-state, start in #5.
  - [Right,Suck]
- Conformant, start in {1,2,3,4,5,6,7,8}, Rigth goes to {2,4,6,8}
  - [Right,Suck,Left,Suck]
- Contingency, start in #5
  - Murphy law: Suck can dirty a clean carpet;
  - local sensing: dirt, location only
  - [Right, if dirt then Suck]

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"

A **solution** is a sequence of actions leading from the initial state to a goal state

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"
- **successor function** $S(x) =$ set of action–state pairs

A **solution** is a sequence of actions leading from the initial state to a goal state

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"
- **successor function** S(x) = set of action–state pairs
  - e.g. S(Arad) = {<`Arad-Zerind, Zerind`>, ... }

A **solution** is a sequence of actions leading from the initial state to a goal state

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"
- **successor function** S(x) = set of action–state pairs
  - e.g. S(Arad) = {<`Arad-Zerind, Zerind`>, ... }
- **goal test**, can be

A **solution** is a sequence of actions leading from the initial state to a goal state

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"
- **successor function** S(x) = set of action–state pairs
  - e.g. S(Arad) = {<`Arad-Zerind, Zerind`>, ... }
- **goal test**, can be
  - *explicit*, e.g. x = "at Bucharest"

A **solution** is a sequence of actions leading from the initial state to a goal state

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"
- **successor function** S(x) = set of action–state pairs
  - e.g. S(Arad) = {<`Arad-Zerind, Zerind`>, … }
- **goal test**, can be
  - *explicit*, e.g. x = "at Bucharest"
  - *implicit*, e.g. `NoDirt(x)`

A **solution** is a sequence of actions leading from the initial state to a goal state

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"
- **successor function** S(x) = set of action–state pairs
  - e.g. S(Arad) = {<`Arad-Zerind, Zerind`>, . . . }
- **goal test**, can be
  - *explicit*, e.g. x = "at Bucharest"
  - *implicit*, e.g. `NoDirt(x)`
- **path cost** (additive)

A **solution** is a sequence of actions leading from the initial state to a goal state

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"
- **successor function** S(x) = set of action–state pairs
  - e.g. S(Arad) = {<`Arad-Zerind, Zerind`>, . . . }
- **goal test**, can be
  - *explicit*, e.g. x = "at Bucharest"
  - *implicit*, e.g. `NoDirt(x)`
- **path cost** (additive)
  - e.g. sum of distances, number of actions executed, etc.

A **solution** is a sequence of actions leading from the initial state to a goal state

Single-state problem formulation

A **problem** is defined by four items:

- **initial state** e.g. "at Arad"
- **successor function** S(x) = set of action–state pairs
  - e.g. S(Arad) = {<`Arad-Zerind, Zerind`>, . . . }
- **goal test**, can be
  - *explicit*, e.g. x = "at Bucharest"
  - *implicit*, e.g. `NoDirt(x)`
- **path cost** (additive)
  - e.g. sum of distances, number of actions executed, etc.
  - c(x,a,y) is the **step cost**, assumed to be ≥ 0

A **solution** is a sequence of actions leading from the initial state to a goal state

Selecting a state space

- Real world is absurdly complex

Selecting a state space

- Real world is absurdly complex
  - ⇒ state space must be *abstracted* for problem solving

Selecting a state space

- Real world is absurdly complex
  - ⇒ state space must be *abstracted* for problem solving
- (Abstract) state = set of real states

Selecting a state space

- Real world is absurdly complex
  - $\Rightarrow$ state space must be *abstracted* for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions

Selecting a state space

- Real world is absurdly complex
  - $\Rightarrow$ state space must be *abstracted* for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g. "Arad $\rightarrow$ Zerind" represents a complex set of possible routes, detours, rest stops, etc.

Selecting a state space

- Real world is absurdly complex
  - $\Rightarrow$ state space must be *abstracted* for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g. "Arad $\rightarrow$ Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - For guaranteed realizability, *any* real state "in Arad" must get to *some* real state "in Zerind"

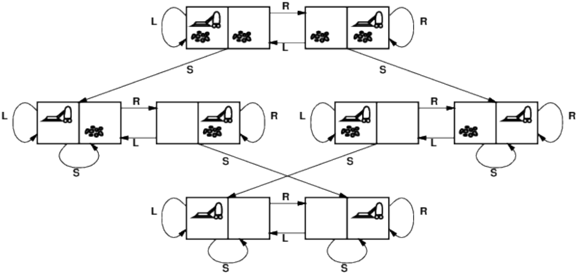
Selecting a state space

- Real world is absurdly complex
  - $\Rightarrow$ state space must be *abstracted* for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g. "Arad $\rightarrow$ Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - For guaranteed realizability, *any* real state "in Arad" must get to *some* real state "in Zerind"
- (Abstract) solution = set of real paths that are solutions in the real world

Selecting a state space

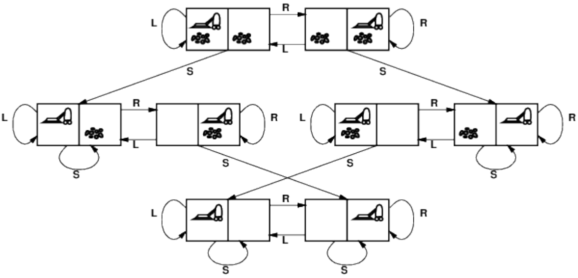
- Real world is absurdly complex
  - $\Rightarrow$ state space must be *abstracted* for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g. "Arad $\rightarrow$ Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - For guaranteed realizability, *any* real state "in Arad" must get to *some* real state "in Zerind"
- (Abstract) solution = set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem!

Example: vacuum world state space graph



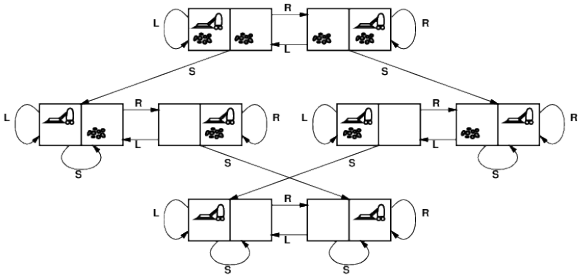
- states

Example: vacuum world state space graph



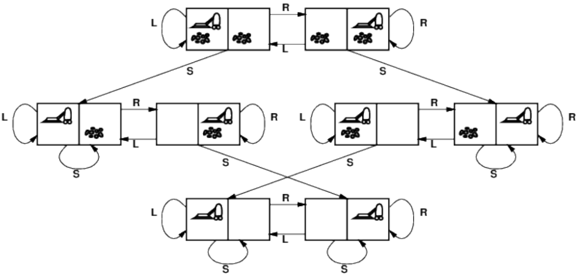
- states
  - integer dirt and robot locations (ignore dirt etc.)

Example: vacuum world state space graph



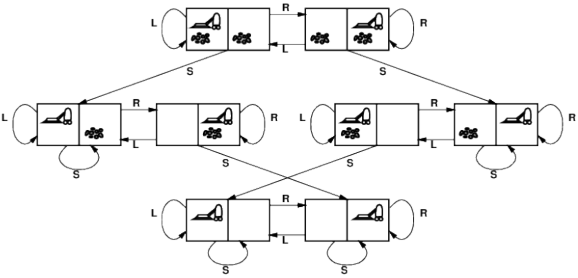
- states
  - integer dirt and robot locations (ignore dirt etc.)
- actions

Example: vacuum world state space graph



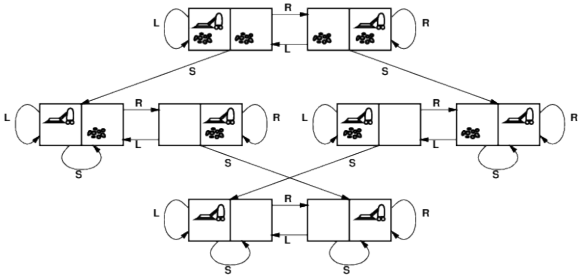
- states
  - integer dirt and robot locations (ignore dirt etc.)
- actions
  - Left, Right, Suck, NoOp

Example: vacuum world state space graph



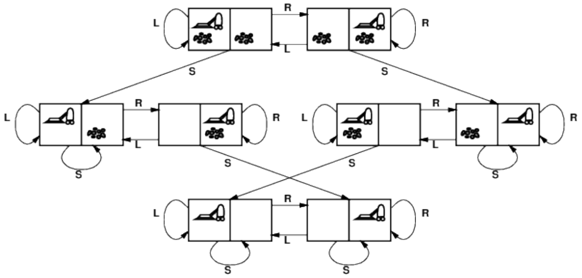
- states
  - integer dirt and robot locations (ignore dirt etc.)
- actions
  - Left, Right, Suck, NoOp
- goal test

Example: vacuum world state space graph



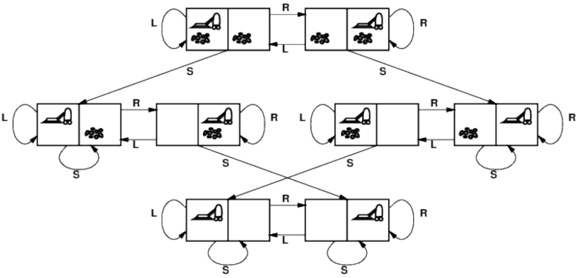
- states
  - integer dirt and robot locations (ignore dirt etc.)
- actions
  - Left, Right, Suck, NoOp
- goal test
  - no dirt

Example: vacuum world state space graph



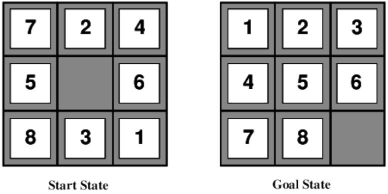
- states
  - integer dirt and robot locations (ignore dirt etc.)
- actions
  - Left, Right, Suck, NoOp
- goal test
  - no dirt
- path cost

Example: vacuum world state space graph



- states
  - integer dirt and robot locations (ignore dirt etc.)
- actions
  - Left, Right, Suck, NoOp
- goal test
  - no dirt
- path cost
  - 1 per action (0 for NoOp)

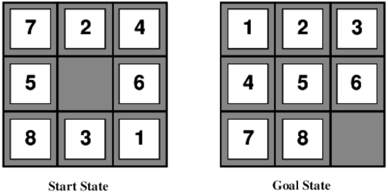
Example: The 8-puzzle



- states

Note: optimal solution of *n*-Puzzle family is NP-hard

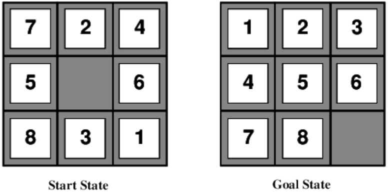
Example: The 8-puzzle



- states
  - integer locations of tiles (ignore intermediate positions)

Note: optimal solution of *n*-Puzzle family is NP-hard

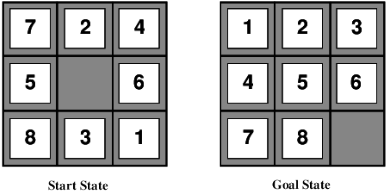
Example: The 8-puzzle



- states
  - integer locations of tiles (ignore intermediate positions)
- actions

Note: optimal solution of n-Puzzle family is NP-hard

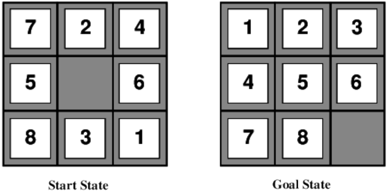
Example: The 8-puzzle



- states
  - integer locations of tiles (ignore intermediate positions)
- actions
  - move blank left, right, up, down (ignore unjamming etc.)

Note: optimal solution of *n*-Puzzle family is NP-hard

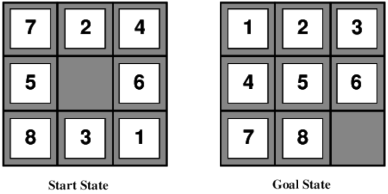
Example: The 8-puzzle



- states
  - integer locations of tiles (ignore intermediate positions)
- actions
  - move blank left, right, up, down (ignore unjamming etc.)
- goal test

Note: optimal solution of *n*-Puzzle family is NP-hard

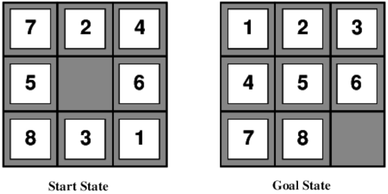
Example: The 8-puzzle



- states
  - integer locations of tiles (ignore intermediate positions)
- actions
  - move blank left, right, up, down (ignore unjamming etc.)
- goal test
  - $=$ goal state (given)

Note: optimal solution of *n*-Puzzle family is NP-hard

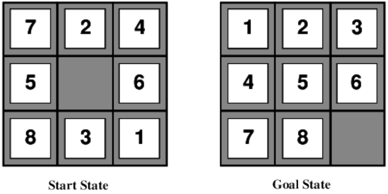
Example: The 8-puzzle



- states
  - integer locations of tiles (ignore intermediate positions)
- actions
  - move blank left, right, up, down (ignore unjamming etc.)
- goal test
  - = goal state (given)
- path cost

Note: optimal solution of *n*-Puzzle family is NP-hard

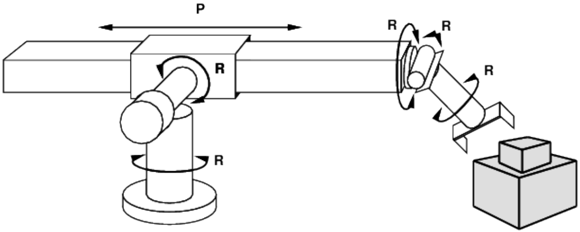
Example: The 8-puzzle



- states
  - integer locations of tiles (ignore intermediate positions)
- actions
  - move blank left, right, up, down (ignore unjamming etc.)
- goal test
  - $=$ goal state (given)
- path cost
  - 1 per move

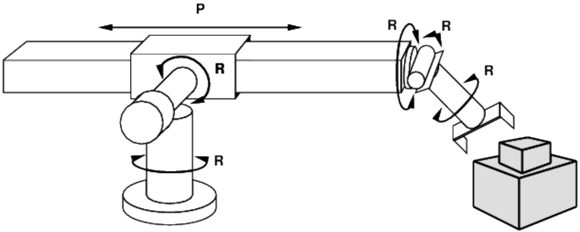
Note: optimal solution of n-Puzzle family is NP-hard

Example: robotic assembly



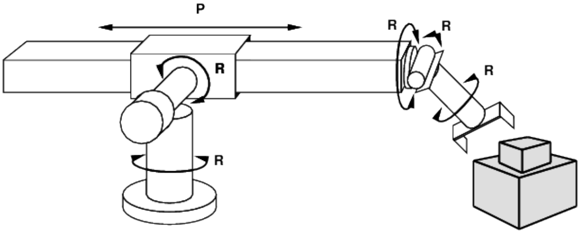
- states

Example: robotic assembly



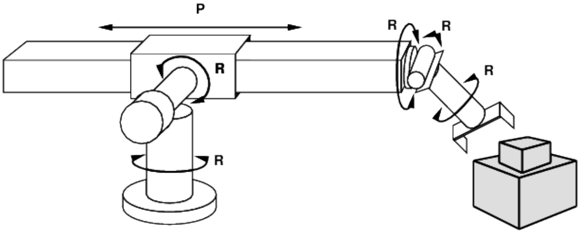
- states
  - real-valued coordinates of robot joint angles

Example: robotic assembly



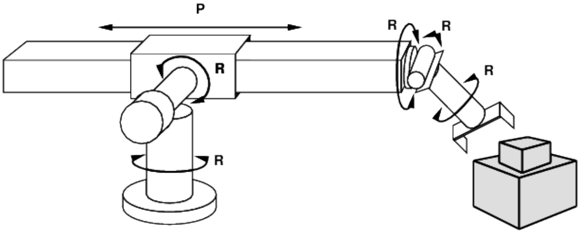
- states
  - real-valued coordinates of robot joint angles
  - parts of the object to be assembled

Example: robotic assembly



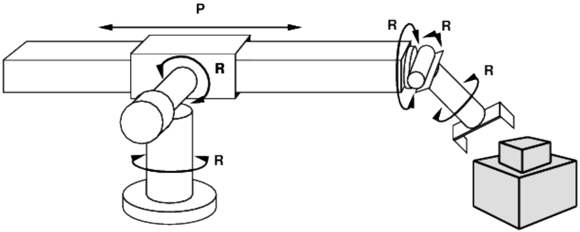
- states
  - real-valued coordinates of robot joint angles
  - parts of the object to be assembled
- actions

Example: robotic assembly



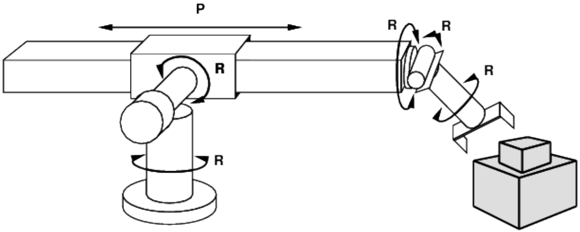
- states
  - real-valued coordinates of robot joint angles
  - parts of the object to be assembled
- actions
  - continuous motions of robot joints

Example: robotic assembly



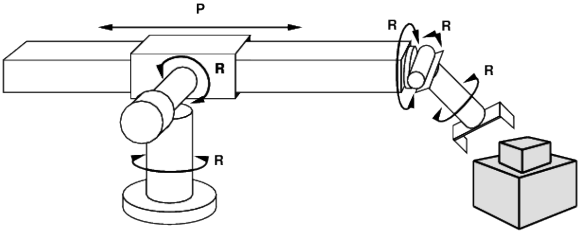
- states
  - real-valued coordinates of robot joint angles
  - parts of the object to be assembled
- actions
  - continuous motions of robot joints
- goal test

Example: robotic assembly



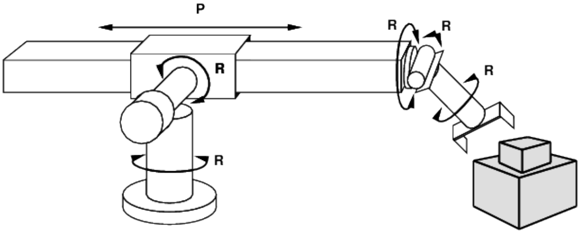
- states
  - real-valued coordinates of robot joint angles
  - parts of the object to be assembled
- actions
  - continuous motions of robot joints
- goal test
  - complete assembly *with no robot included!*

Example: robotic assembly



- states
  - real-valued coordinates of robot joint angles
  - parts of the object to be assembled
- actions
  - continuous motions of robot joints
- goal test
  - complete assembly *with no robot included!*
- path cost

Example: robotic assembly



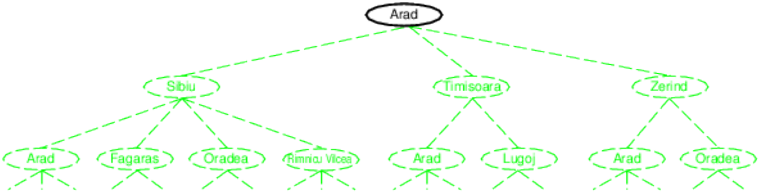
- states
  - real-valued coordinates of robot joint angles
  - parts of the object to be assembled
- actions
  - continuous motions of robot joints
- goal test
  - complete assembly *with no robot included!*
- path cost
  - time to execute

Tree search algorithms

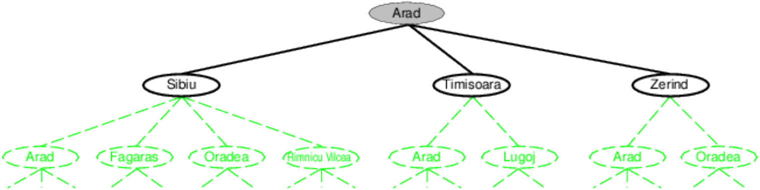
- Basic idea:
  - offline, simulated exploration of state space
  - by generating successors of already-explored states
    - (a.k.a. **expanding** states)

```
function Tree-Search(problem, strategy): a solution or failure
  initialize the search tree with --the initial state of probl
  loop do
    if there are no candidates for expansion
      then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state
      then return the corresponding solution
    else expand the node and add the resulting nodes
                                   to the search tree
    end
```

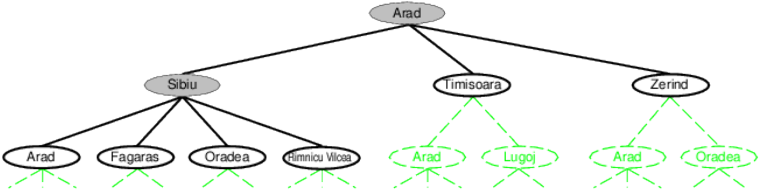
Tree search example



Tree search example

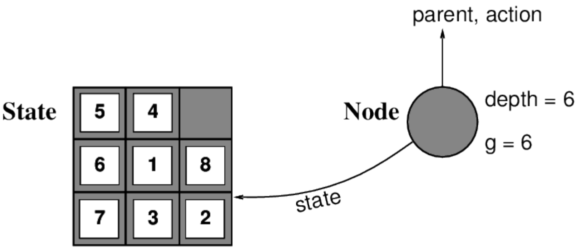


Tree search example



Implementation: states vs.nodes

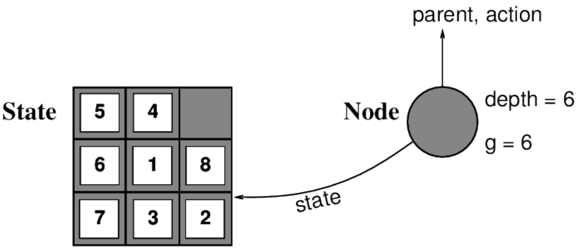
- A **state** is a (representation of) a physical configuration



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

Implementation: states vs.nodes

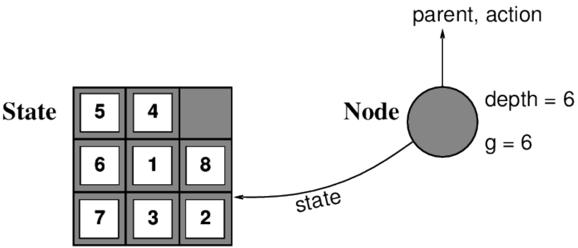
- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree



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

Implementation: states vs.nodes

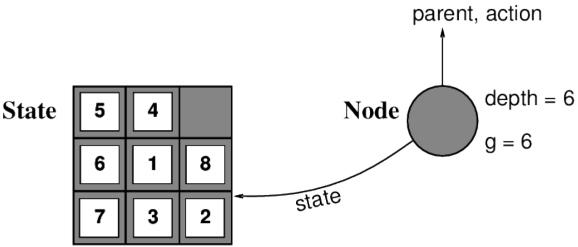
- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree
  - includes *parent*, *children*, *depth*, *path cost* g(x)



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

Implementation: states vs.nodes

- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree
  - includes *parent*, *children*, *depth*, *path cost* g(x)
- States do **not** have parents, children, depth, or path cost!



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

Implementation: general tree search

```
function Tree-Search(problem, fringe): a solution, or failure
  fringe = Insert(Make-Node(Initial-State[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node = Remove-Front(fringe)
    if Goal-Test(problem, State(node)) then return node
    fringe = InsertAll(Expand(node, problem), fringe)
```

Implementation: general tree search

```
function Expand(node, problem)): a set of nodes
  successors = the empty set
  for each action, result in
         Successor-Fn(problem, State[node]) do
    s = a new Node
    Parent-Node[s] = node
    Action[s]     = action
    State[s]      = result
    Path-Cost[s] = Path-Cost[node] +
                   Step-Cost(State[node],action,result)
    Depth[s]      = Depth[node] + 1
    add s to successors
  return successors
```

Search strategies

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

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded
- **space complexity**

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded
- **space complexity**
  - maximum number of nodes in memory

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded
- **space complexity**
  - maximum number of nodes in memory
- **optimality**

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded
- **space complexity**
  - maximum number of nodes in memory
- **optimality**
  - does it always find a least-cost solution?

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded
- **space complexity**
  - maximum number of nodes in memory
- **optimality**
  - does it always find a least-cost solution?
- Time and space complexity are measured in terms of

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded
- **space complexity**
  - maximum number of nodes in memory
- **optimality**
  - does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded
- **space complexity**
  - maximum number of nodes in memory
- **optimality**
  - does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution

Search strategies

- A strategy is defined by picking the *order* of node expansion
- Strategies are evaluated along the following dimensions:
- **completeness**
  - does it always find a solution if one exists?
- **time complexity**
  - number of nodes generated/expanded
- **space complexity**
  - maximum number of nodes in memory
- **optimality**
  - does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be ∞)

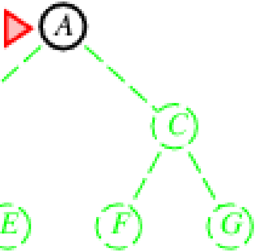
Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

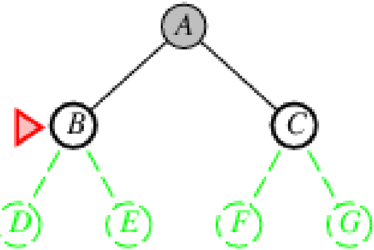
Breadth-first search

Expand **shallowest** unexpanded node
Implementation: *fringe* is a FIFO queue, i.e. new successors go at end



Breadth-first search

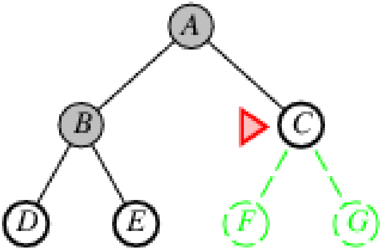
Expand **shallowest** unexpanded node
Implementation: *fringe* is a FIFO queue, i.e. new successors go at end



Breadth-first search

Expand **shallowest** unexpanded node

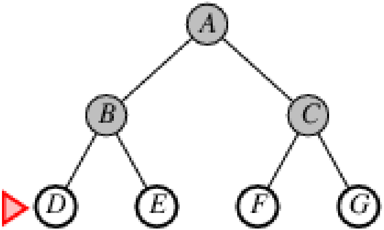
Implementation: *fringe* is a FIFO queue, i.e. new successors go at end



Breadth-first search

Expand **shallowest** unexpanded node

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



Properties of breadth-first search

- Complete?

Properties of breadth-first search

- Complete?
  - Yes (if b is finite)

Properties of breadth-first search

- Complete?
  - Yes (if b is finite)
- Time?

Properties of breadth-first search

- Complete?
  - Yes (if b is finite)
- Time?
  - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e. exp. in d

Properties of breadth-first search

- Complete?
  - Yes (if b is finite)
- Time?
  - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e. exp. in d
- Space?

Properties of breadth-first search

- Complete?
  - Yes (if b is finite)
- Time?
  - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e. exp. in d
- Space?
  - $O(b^{d+1})$ (keeps every node in memory)

Properties of breadth-first search

- Complete?
  - Yes (if b is finite)
- Time?
  - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e. exp. in d
- Space?
  - $O(b^{d+1})$ (keeps every node in memory)
- Optimal?

Properties of breadth-first search

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

Properties of breadth-first search

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

Uniform-cost search

- Expand **least-cost** unexpanded node

Uniform-cost search

- Expand **least-cost** unexpanded node
- *Implementation*: *fringe* = queue ordered by path cost, lowest first

Uniform-cost search

- Expand **least-cost** unexpanded node
- *Implementation*: *fringe* = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal

Uniform-cost search

- Expand **least-cost** unexpanded node
- *Implementation*: *fringe* = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal
- Complete?

Uniform-cost search

- Expand **least-cost** unexpanded node
- *Implementation*: *fringe* = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal
- Complete?
  - Yes, if step cost $\geq \epsilon$

Uniform-cost search

- Expand **least-cost** unexpanded node
- *Implementation*: *fringe* = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal
- Complete?
  - Yes, if step cost $\geq \epsilon$
- Time?

Uniform-cost search

- Expand **least-cost** unexpanded node
- *Implementation*: *fringe* = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal
- Complete?
  - Yes, if step cost $\geq \epsilon$
- Time?
  - Number of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* is the cost of the optimal solution

Uniform-cost search

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

Uniform-cost search

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

Uniform-cost search

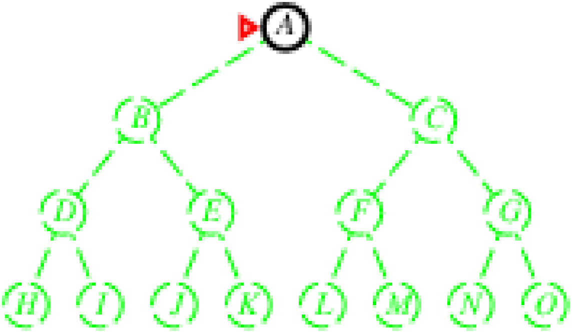
- Expand **least-cost** unexpanded node
- *Implementation*: *fringe* = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal
- Complete?
  - Yes, if step cost $\geq \epsilon$
- Time?
  - Number of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon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

Uniform-cost search

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

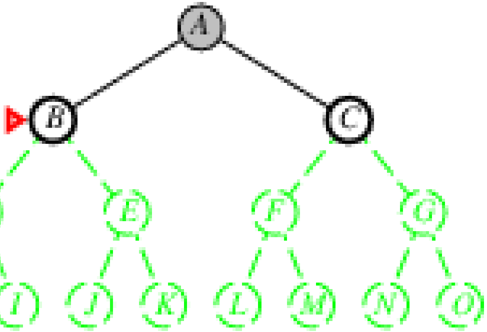
Depth-first search

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



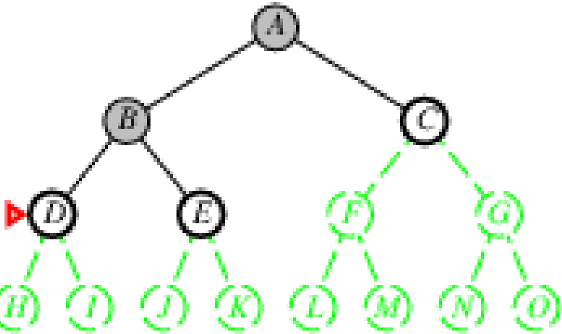
Depth-first search

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



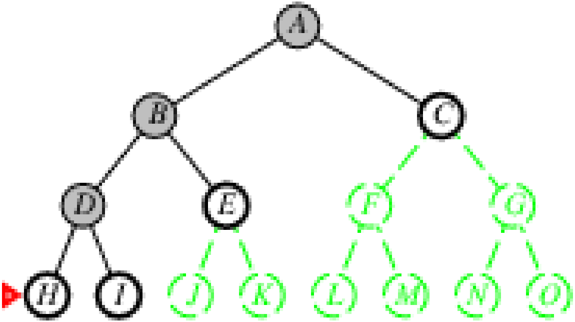
Depth-first search

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



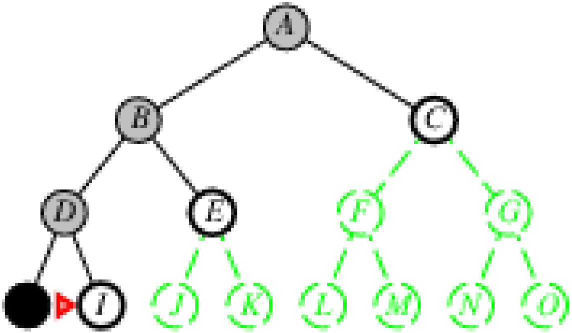
Depth-first search

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



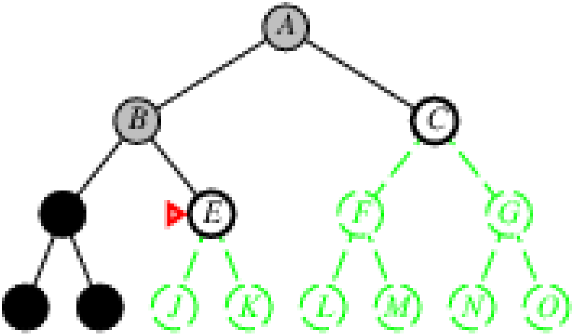
Depth-first search

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



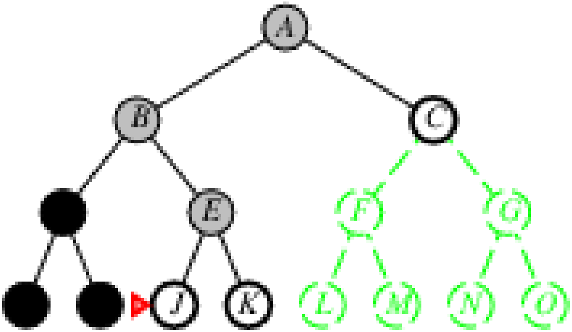
Depth-first search

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



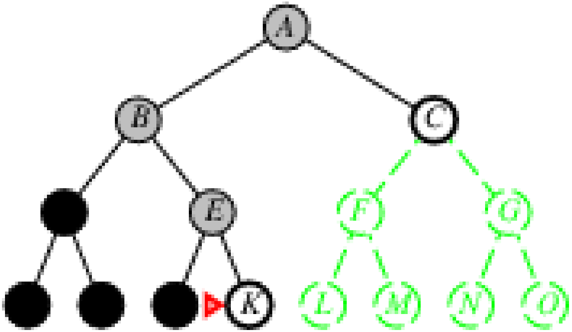
Depth-first search

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



Depth-first search

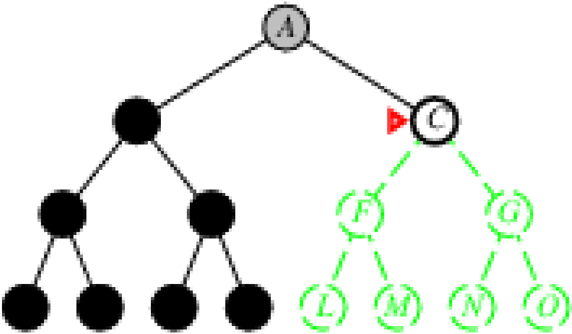
Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front



Depth-first search

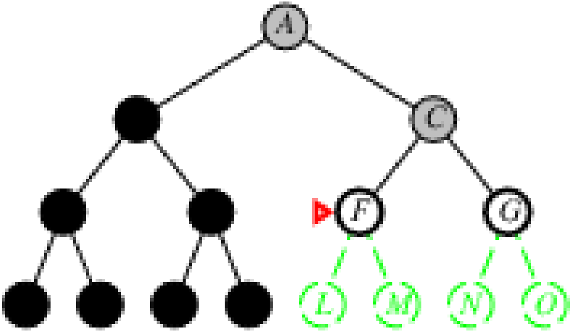
Expand **deepest** unexpanded node

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



Depth-first search

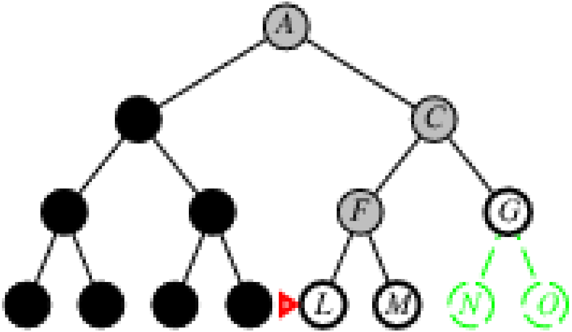
Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front



Depth-first search

Expand **deepest** unexpanded node

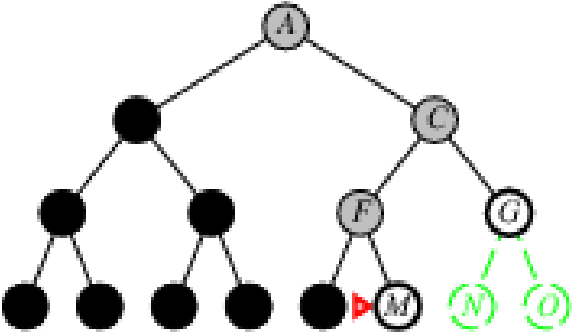
Implementation: *fringe* = LIFO queue, i.e. put successors at front



Depth-first search

Expand **deepest** unexpanded node

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



Properties of depth-first search

- Complete?

Properties of depth-first search

- Complete?
  - No: fails in infinite-depth spaces, spaces with loops,

Properties of depth-first search

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

Properties of depth-first search

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

Properties of depth-first search

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

Properties of depth-first search

- Complete?
  - No: fails in infinite-depth spaces, spaces with loops,
  - Modify to avoid repeated states along path,
  - $\Rightarrow$ complete in finite spaces
- Time?
  - $O(b^m)$: terrible if m is much larger than d, but if solutions are dense, may be much faster than breadth-first

Properties of depth-first search

- Complete?
  - No: fails in infinite-depth spaces, spaces with loops,
  - Modify to avoid repeated states along path,
  - $\Rightarrow$ complete in finite spaces
- Time?
  - $O(b^m)$: terrible if m is much larger than d, but if solutions are dense, may be much faster than breadth-first
- Space?

Properties of depth-first search

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

Properties of depth-first search

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

Properties of depth-first search

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

Depth-limited search

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

Recursive implementation:

```
function Depth-Limited-Search(problem, limit):
                                 soln/fail/cutoff
  Recursive-DLS(Make-Node(Initial-State[problem]),
                problem, limit)
```

Depth-limited search

```
function Recursive-DLS(node, problem, limit):
                                soln/fail/cutoff
  cutoff-occurred? = false
  if Goal-Test(problem, State[node]) then return node
  else if Depth[node] = limit then return cutoff
  else
    for each successor in Expand(node, problem) do
      result = Recursive-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? = true
      else if result != failure then return result
  if cutoff-occurred? then return cutoff
  else return failure
```

Iterative deepening search

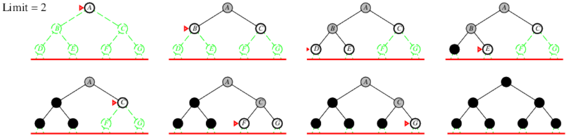
```
function Iterative-Deepening-Search(problem): a solution
  for depth = 0 to infinity do
    result = Depth-Limited-Search(problem, depth)
    if result != cutoff then return result
  end
```

Iterative deepening search $l = 0$

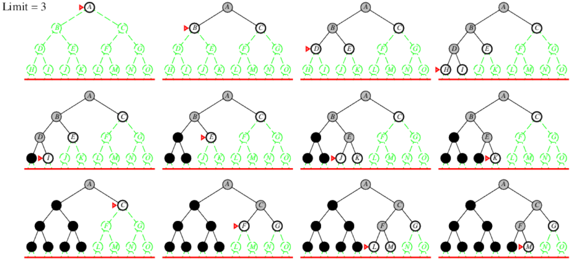
Limit = 0

Iterative deepening search $l = 1$

Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



Properties of iterative deepening search

- Complete?

Properties of iterative deepening search

- Complete?
  - Yes

Properties of iterative deepening search

- Complete?
  - Yes
- Time?

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost $= 1$

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost $= 1$
  - Can be modified to explore uniform-cost tree

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost $= 1$
  - Can be modified to explore uniform-cost tree
- Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost = 1
  - Can be modified to explore uniform-cost tree
- Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  - N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost = 1
  - Can be modified to explore uniform-cost tree
- Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  - N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
  - N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100

Properties of iterative deepening search

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost = 1
  - Can be modified to explore uniform-cost tree
- Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  - N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
  - N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
- IDS does better because other nodes at depth d are not expanded

Properties of iterative deepening search

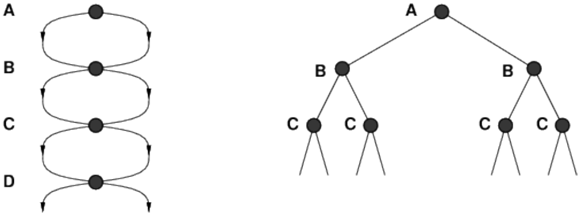
- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost = 1
  - Can be modified to explore uniform-cost tree
- Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  - N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
  - N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
- IDS does better because other nodes at depth d are not expanded
- BFS can be modified to apply goal test when a node is *generated*

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Repeated states

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



Graph search

```
function Graph-Search(problem, fringe): a solution, or failure
  closed = an empty set
  fringe = Insert(Make-Node(Initial-State[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node = Remove-Front(fringe)
    if Goal-Test(problem, State[node]) then return node
    if State[node] is not in closed then
       add State[node] to closed
           fringe = InsertAll(Expand(node, problem), fringe)
  end
```

Summary

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