Unit 6: Planning

J. L. Ruiz Reina
A. Riscos Núñez
Fco. Jesús Martín Mateos
J. A. Alonso Jiménez

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Índice

1. Introduction
2. Representation of problems as state spaces
3. Basic search techniques on state spaces
4. PDDL: a specific representation for planning problems
5. Forward and backward search in the state space
6. Heuristics for planning
7. Partial Order Planning: POP

Section 1

Introduction

Planning in AI

- Planning: finding a sequence of *actions* that enable to reach a given *goal* when executed from a given *initial state*.
- Plan: sequence of actions that achieve the goal
- Real-world applications:
  - Robotics
  - Manufacturing by components assembly
  - Space missions
- *Planning* vs *Scheduling*
  - *Planning* focuses on finding a sequence of actions to achieve a goal and *Scheduling* emphasizes the efficient assignment of available resources to the actions

Planning problems in AI

- Types of problems, according to:
  - Actions: deterministic or non-deterministic
  - States: finite or infinite descriptions of states
  - Fully observable *vs* partially observable
  - Duration of actions
  - Concurrency, single agent *vs* multiagent
  - Optimization issues
- For the sake of simplicity, along the unit we assume:
  - A finite number of states
  - Fully observable
  - Deterministic actions, precisely defined by their specification
  - Implicit time: actions are performed *without duration*
  - Planning is done *a priori*

Planning problems in AI

- (Some) Issues to address:
  - *Representation of the world and the actions transforming it*
  - *Algorithms for searching plans*
  - Minimizing the resources consumed by the plan
  - Time when each action is executed
  - Monitorize the execution of the plan, including revisions in case of errors or contingencies
  - Multiagent planning

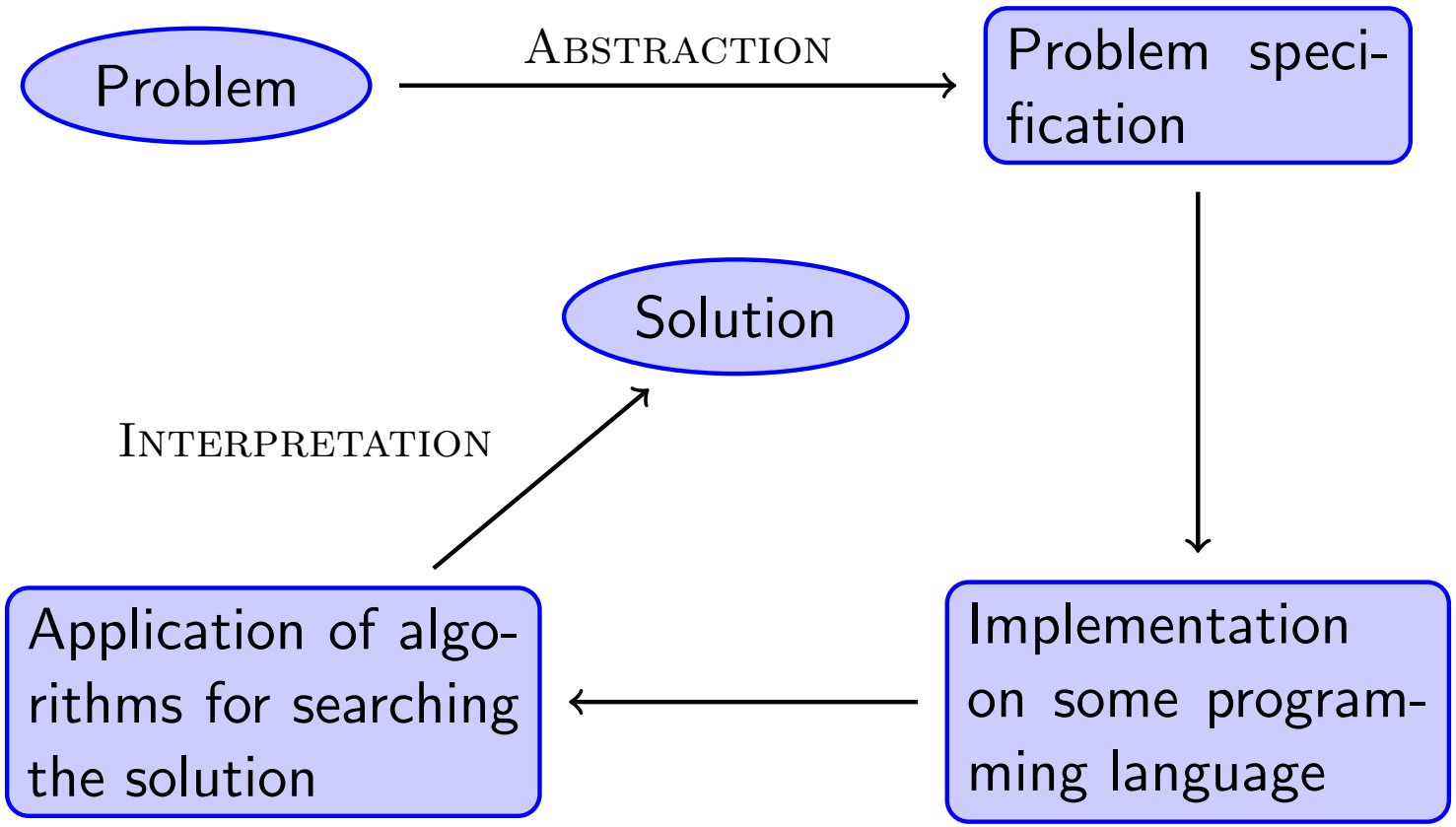
Section 2

Representation of problems as state spaces

Definition of a problem as a state space

- Planning problemas ara a special case of state spece search problems
- Before searching for the solution to a problem, we have to *specify* (describe) it
- Elements of a state space problem:
  - Which is the *initial* situation where we start?
  - Which is the *final goal*?
  - How to describe all possible intermediate situations or *states* that we could encounter?
  - Which elementary steps or *actions* are available to modify the state and how do they work?
- Specifying a problem as a *state space* consists on describing in a precise way each one of such components
  - Advantage: general procedures for searching solutions
  - *Independent* of the problem

Protocol for solving problems



Example: formulating the 8-puzzle problem

- A 3x3 board with 8 numbered tiles on it (thus leaving a blank free space of the same size of a tile). A tile adjacent to the blank space can slide into it. The game consists on transforming the initial position into the final one by means of a sequence of tile slidings. In particular, let us consider the following initial and final states:

2	8	3
1	6	4
7		5

Initial state

1	2	3
8		4
7	6	5

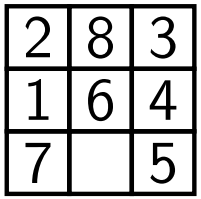
Final state

Representing states

- State: description of a possible situation of the problem
  - Abstraction of its properties
- Importance of a good representation of states
  - Only information being relevant for the problem should be considered
  - Depending on the chosen representation the number of states will vary, influencing the performance of the procedures for searching solutions
- Example of the 8-puzzle: Elements of the representation:
  - relevant: localization of each tile and the space;
  - irrelevant: the material the tiles are made of, colors,...

States Representation

- Example of the 8-puzzle: State representations



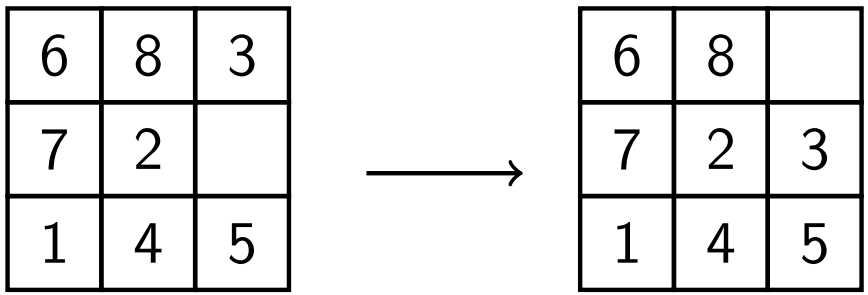
  - Description of the exact position of each one of the tiles
- Representation *vs.* implementation
  - Tuples: `(2 8 3 1 6 4 7 H 5), (2 8 3 4 5 H 7 1 6)`
  - Nested lists: `((2 8 3)(1 6 4)(7 H 5))`
  - Dictionaries: {`"firstleft":2, "firstcenter":8, ...`}
- Number of states: $9! = 362.880$.

Actions

- Actions:
  - Represent a finite set of elementary operators that transform a state into another one
- Elements describing an action
  - *Applicability:* conditions needed to apply a given action to a given state (preconditions, postconditions)
  - Resulting state after the *application* of an (applicable) action over a state
- Criterion for representing actions.
  - Depends on the representation of the states
  - Preferably representations with lower number of actions
- Example: Actions for the 8-puzzle:
  - One for each possible movement of a tile: 32.
  - One for each possible movement of the space: 4.

Actions

- Actions for the 8-puzzle
  - Move the space up
  - Move the space down
  - Move the space to the right
  - Move the space to the left
- Description of the action "Move the space up"
  - Applicability: it is applicable over states where the space is not in the first (upper) row
  - Result of applying the action: the space and the tile situated over it swap their positions



- Analogously for the remaining three actions

Initial State

- Initial State
  - A state describing the situation when the game begins
- Initial State for the 8-puzzle example

2	8	3
1	6	4
7		5

Final States

- Description of the goal
  - Usually, a set of states, which will be called *final*
  - Eventually (not always) a single final state
- Example of the 8-puzzle (only one final state)

1	2	3
8		4
7	6	5

- Approaches for defining final states:
  - Enumerative
  - Declarative

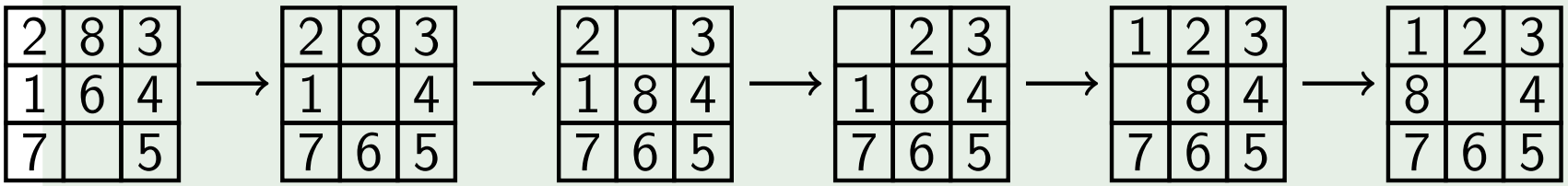
Solutions to a problem

Definition

- Sequence of actions to be performed in order to achieve a goal
- Sequence of actions s.t. a final state is reached if they are applied starting from the initial state

Example

A solution to the 8-puzzle problem

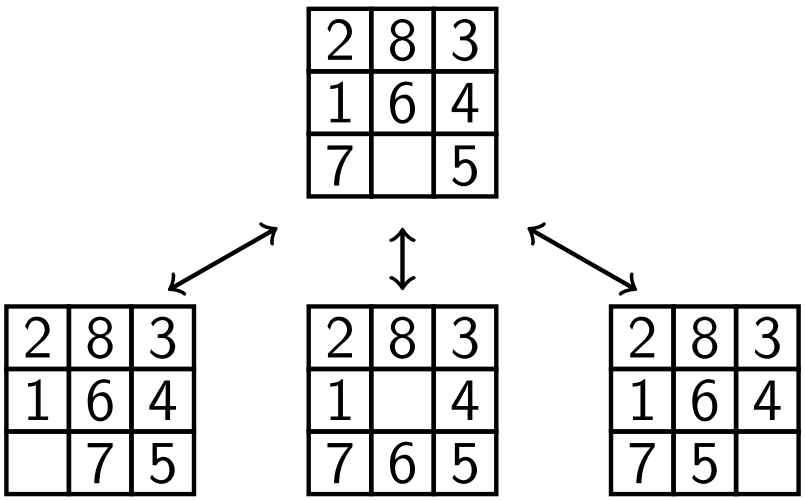


Solutions to a problem

- Types of problems:
  - Searching for a solution.
  - Determining whether there exists a solution and finding a final state.
  - Searching for any solution as fast as possible.
  - Search for all solutions.
  - Search for the shortest solution.
  - Search for the solution of "least cost".

State space as a graph

- A state space can be seen as a directed graph
  - Nodes (or vertices) $\equiv$ States
  - Two nodes are connected in the graph if there is an action applicable to the first one, that obtains the second
- 8-puzzle Example



Implementing the state space

- Usually, the graph that represents the state space is *huge*, so we cannot store it completely in the computer memory
- But we can define the elements (data and functions) that allow us to generate the *successors* of a given state
  - This will allow us to generate parts of the graph *on demmand*, when searching for a solution

Elements for implementing the state space representation

- Implementation of a problem as a state space consists on:
  - Choose a representation (*data structure*) for *states* and for *actions*
  - Define a variable **`*INITIAL-STATE*`**
    - Stores the representation of the initial state
  - Define a function **`IS-FINAL-STATE(STATE)`**
    - Checks whether a given state is final or not
  - Define a function **`ACTIONS(STATE)`**
    - Returns the applicable actions for a given state
  - Define a function **`APPLY(ACTION,STATE)`**
    - Yields the state resulting from applying a given (applicable) action over a given state
- Note: we are not talking about solving the problem yet; for the moment we are only specifying the problem

Example: the Farmer's problem

- The Farmer's problem can be stated as follows:
  - A farmer is by a river with a wolf, a goat and a cabbage.
  - He wishes all of them to cross the river.
  - There's a boat that allows only to bring one thing at a time.
  - The wolf will eat the goat if the farmer is not present.
  - The goat will eat the cabbage if the farmer is not present.
- Information about the states: river bank (left or right) where each element is
  - The river bank of the boat is redundant information

Example: the Farmer's problem

- The Farmer's problem can be stated as follows:
  - A farmer is by a river with a wolf, a goat and a cabbage.
  - He wishes all of them to cross the river.
  - There's a boat that allows only to bring one thing at a time.
  - The wolf will eat the goat if the farmer is not present.
  - The goat will eat the cabbage if the farmer is not present.
- Information about the states: river bank (left or right) where each element is
  - The river bank of the boat is redundant information

Formulating the Farmer's problem

- Representation of the states: (x y z u) in $\{l,r\}^4$.
  - Number of states: 16.
- Initial State: (l l l l).
- Final State (unique): (r r r r).
- Actions:
  - The farmer crosses alone.
  - The farmer crosses with the wolf.
  - The farmer crosses with the goat.
  - The farmer crosses with the cabbage.

Formulating the Farmer's problem

- Representation of the states: (x y z u) in $\{l,r\}^4$.
  - Number of states: 16.
- Initial State: (l l l l).
- Final State (unique): (r r r r).
- Actions:
  - The farmer crosses alone.
  - The farmer crosses with the wolf.
  - The farmer crosses with the goat.
  - The farmer crosses with the cabbage.

Formulating the Farmer's problem

- Applicability of the actions
  - Precondition (for the last three): the two of them must be in the same river bank
  - Poscondition: the resulting state should not have wolf and goat together, neither goat and cabbage together, except if the farmer is also on their river bank
- State resulting of applying the action
  - Swaping the bank of the elements that go on the boat

Example: the water jug problem

- The water jug problem can be stated as follows:
  - We have two jugs, of 3 and 4 liters respectively.
  - None of them has any measuring markers on it.
  - There is a tap that can be used to fill the jugs with water.
  - Figure out the way to get exactly 2 liters of water into the larger jug (the 4-liter one).
- Representation of the states: (xy) being x in $\{0,1,2,3,4\}$ and y in $\{0,1,2,3\}$.
- Number of states: 20.

Example: the water jug problem

- The water jug problem can be stated as follows:
  - We have two jugs, of 3 and 4 liters respectively.
  - None of them has any measuring markers on it.
  - There is a tap that can be used to fill the jugs with water.
  - Figure out the way to get exactly 2 liters of water into the larger jug (the 4-liter one).
- Representation of the states: (xy) being x in $\{0,1,2,3,4\}$ and y in $\{0,1,2,3\}$.
- Number of states: 20.

Formulation of the water jug problem

- Initial State: (0 0).
- Final States: every state of the form (2 y).
- Actions:
  - Fill the 4-liter jug at the tap.
  - Fill the 3-liter jug at the tap.
  - Empty the 4-liter jug to the drain.
  - Empty the 3-liter jug to the drain.
  - Fill the 4-liter jug by pouring the 3-liter jug in.
  - Fill the 3-liter jug by pouring the 4-liter jug in.
  - Empty the 3-liter jug by pouring it into the 4-liter jug.
  - Empty the 4-liter jug by pouring it into the 3-liter jug.

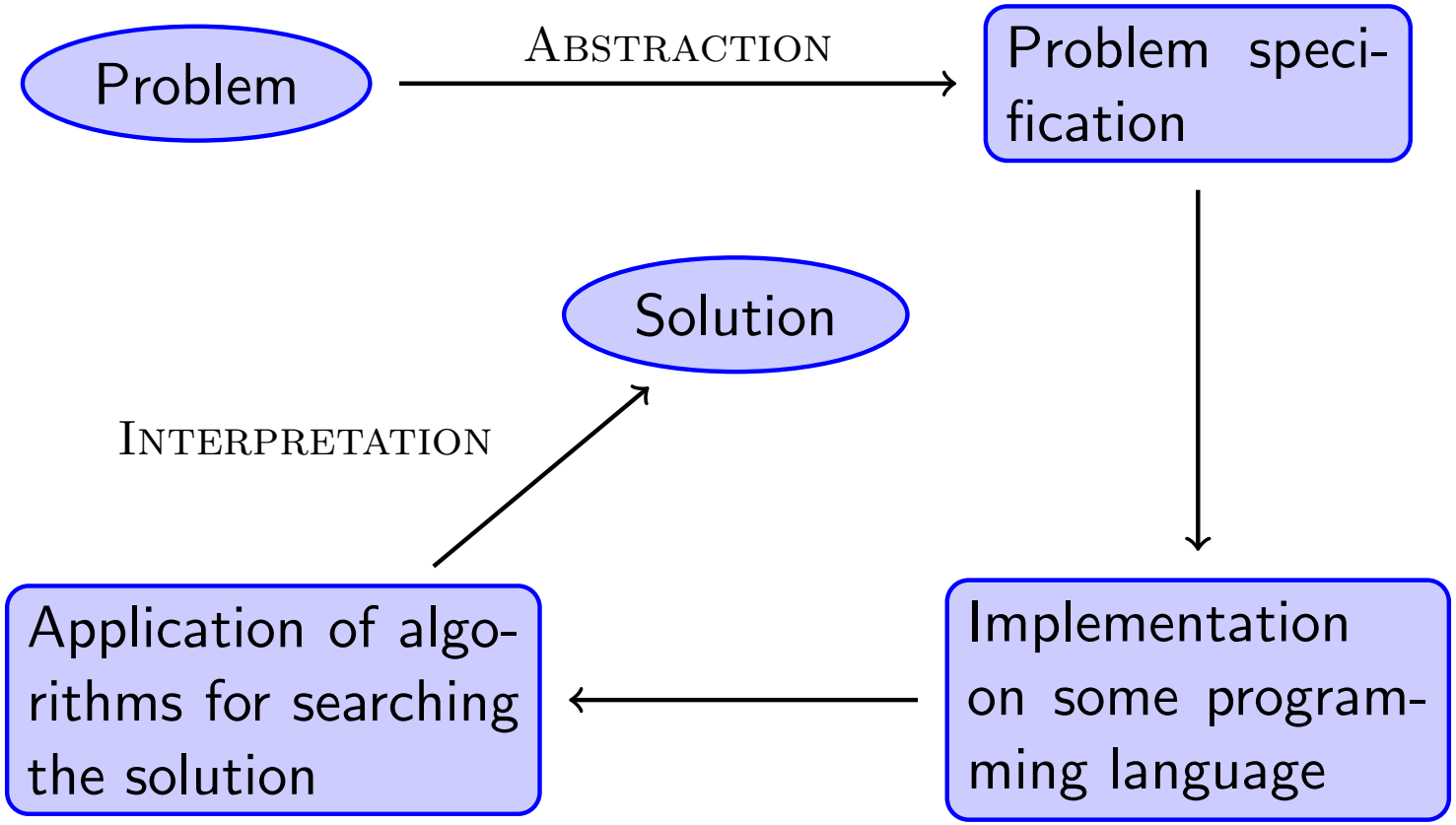
Formulation of the water jug problem

- Application of actions on a state (x y)
- Action "Fill the 3-liter jug at the tap"
  - Applicability: $y<3$ (precondition)
  - Resulting State: (x 3)
- Action "Fill the 4-liter jug by pouring the 3-liter jug in"
  - Applicability: $x<4$, $y>0$, $x+y>4$ (precondition)
  - Resulting State: (4 x+y-4)
- Action "Empty the 3-liter jug by pouring it into the 4-liter jug"
  - Applicability: $y>0$, $x+y\leq 4$ (precondition)
  - Resulting State: (x+y 0)
- Analogously for the rest of the actions

Section 3

Basic search techniques on state spaces

Recall: Protocol for solving problems

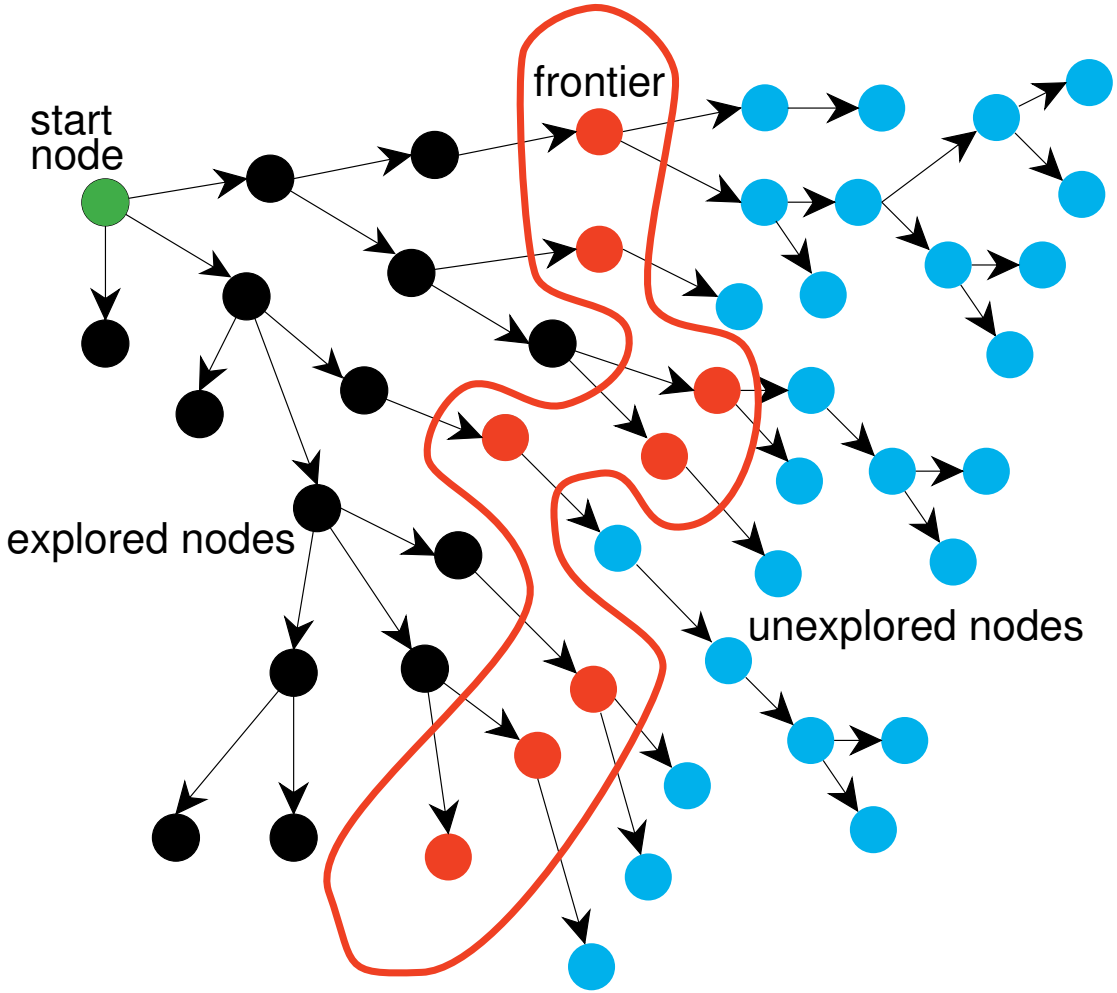


Searching solutions on state spaces

- Goal: to find a sequence of actions that, if applied from the initial state, will lead to a final state
- Basic idea: exploring the *graph* of the state space
  - At each step the current state is analyzed (starting from the initial state)
  - If the current state is final, then halt (gathering the sequence of actions leading to that final state)
  - Otherwise, get the successors of the current state (*expand*)
  - *Select* a new current state, *keeping* the remaining ones for further analysis (if needed)
  - Repeat the process while there are states to analyze
- The selection of the current state for the next step determines a searching *strategy*

Search trees

The searching process just described can be seen as incrementally growing a *search tree* over the state space graph

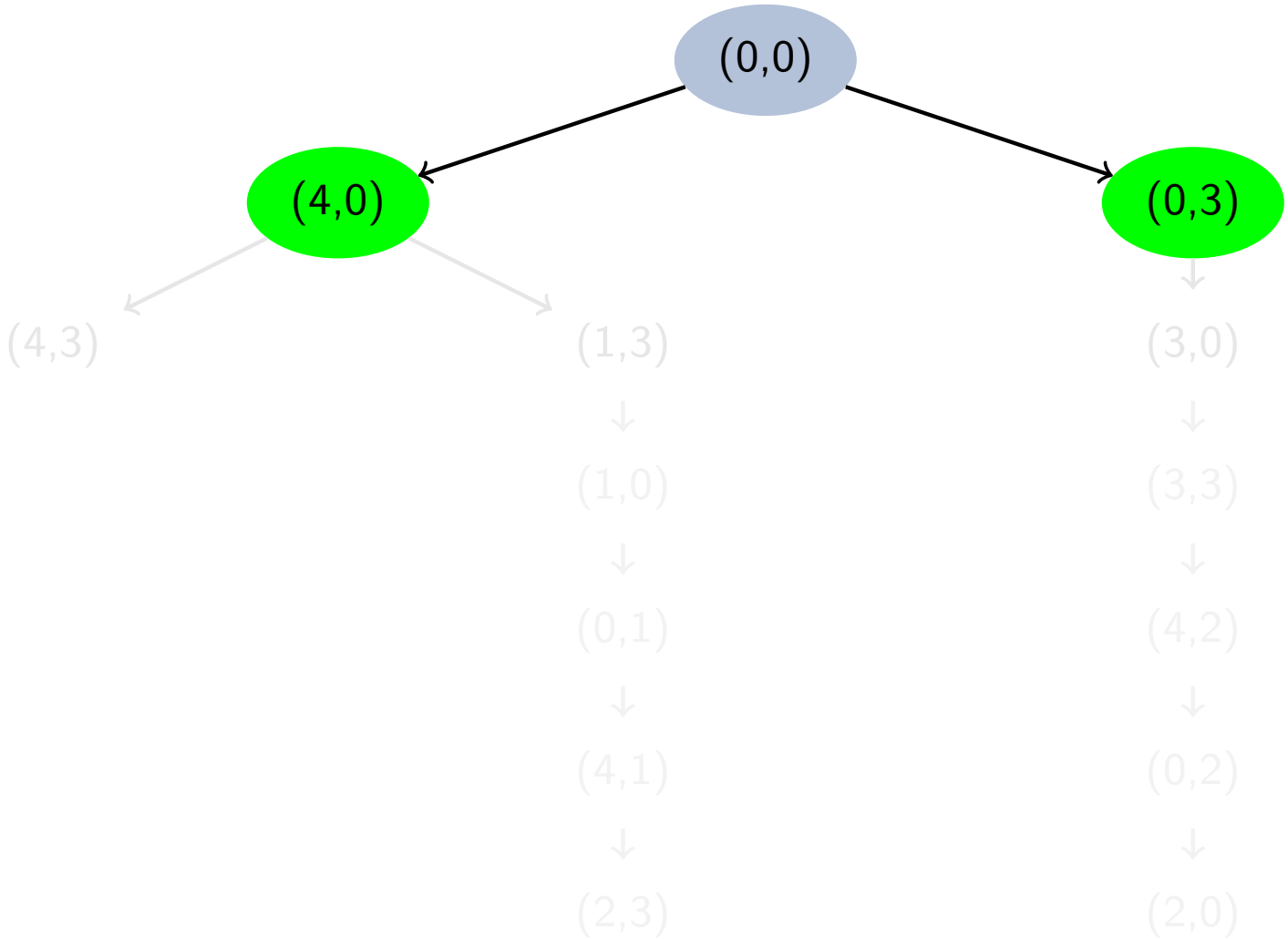


Search strategies: breadth-first vs depth-first

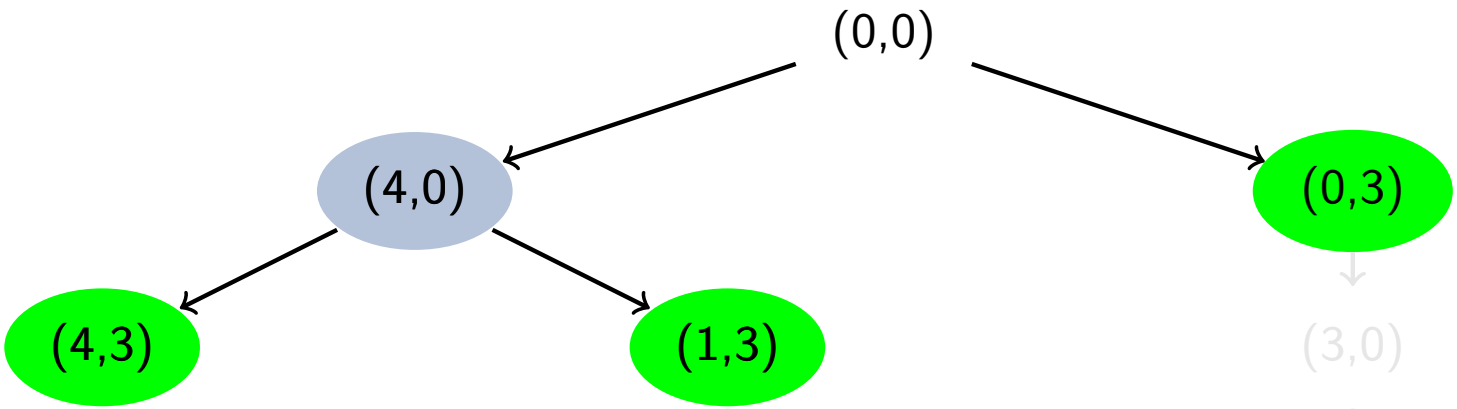
- Search strategies are determined by the way we select the next node to be expanded
- Depth of a node: number of action applications from the initial state
- Two well-known search strategies:
  - Breadth-first strategy: FIFO (first in, first out)
  - Depth-search startegy: LIFO (last in, first out)
- Breadth-first: nodes with less *depth* are expanded first
- Breadth-first: nodes with more *depth* are expanded first

Search tree for Breadth-first-search

Water jug problem

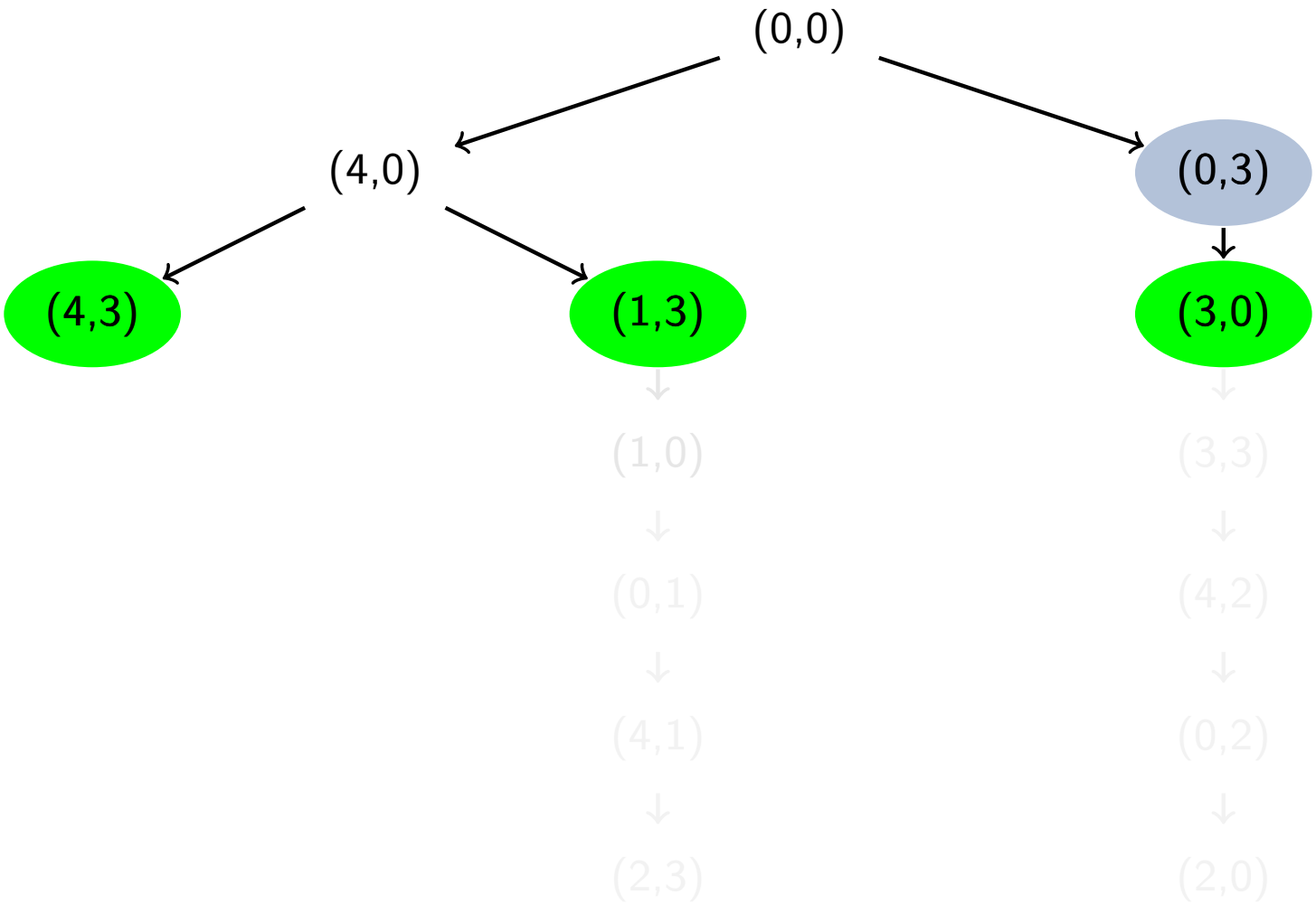


Search tree for Breadth-first-search
Water jug problem



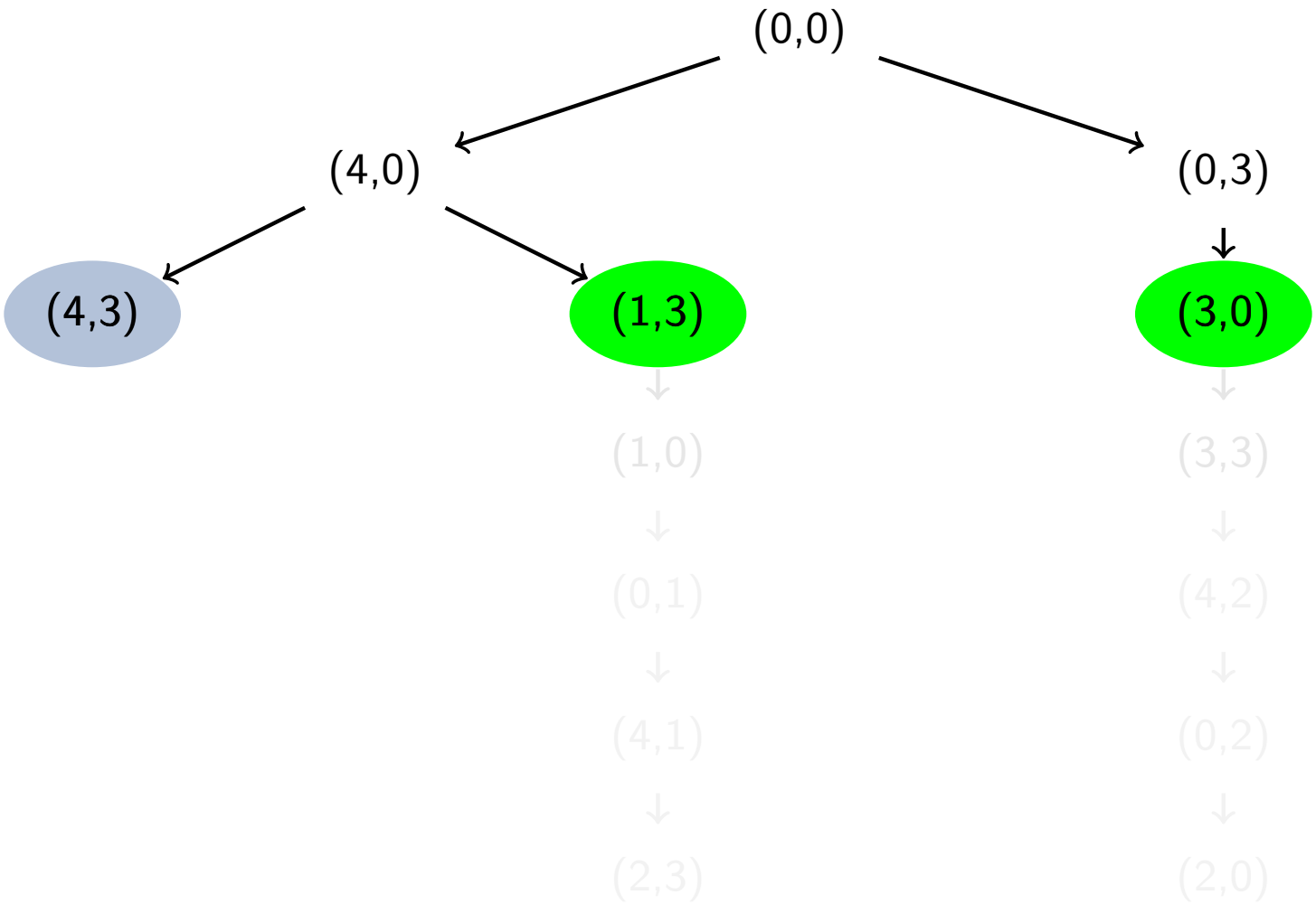
Search tree for Breadth-first-search

Water jug problem



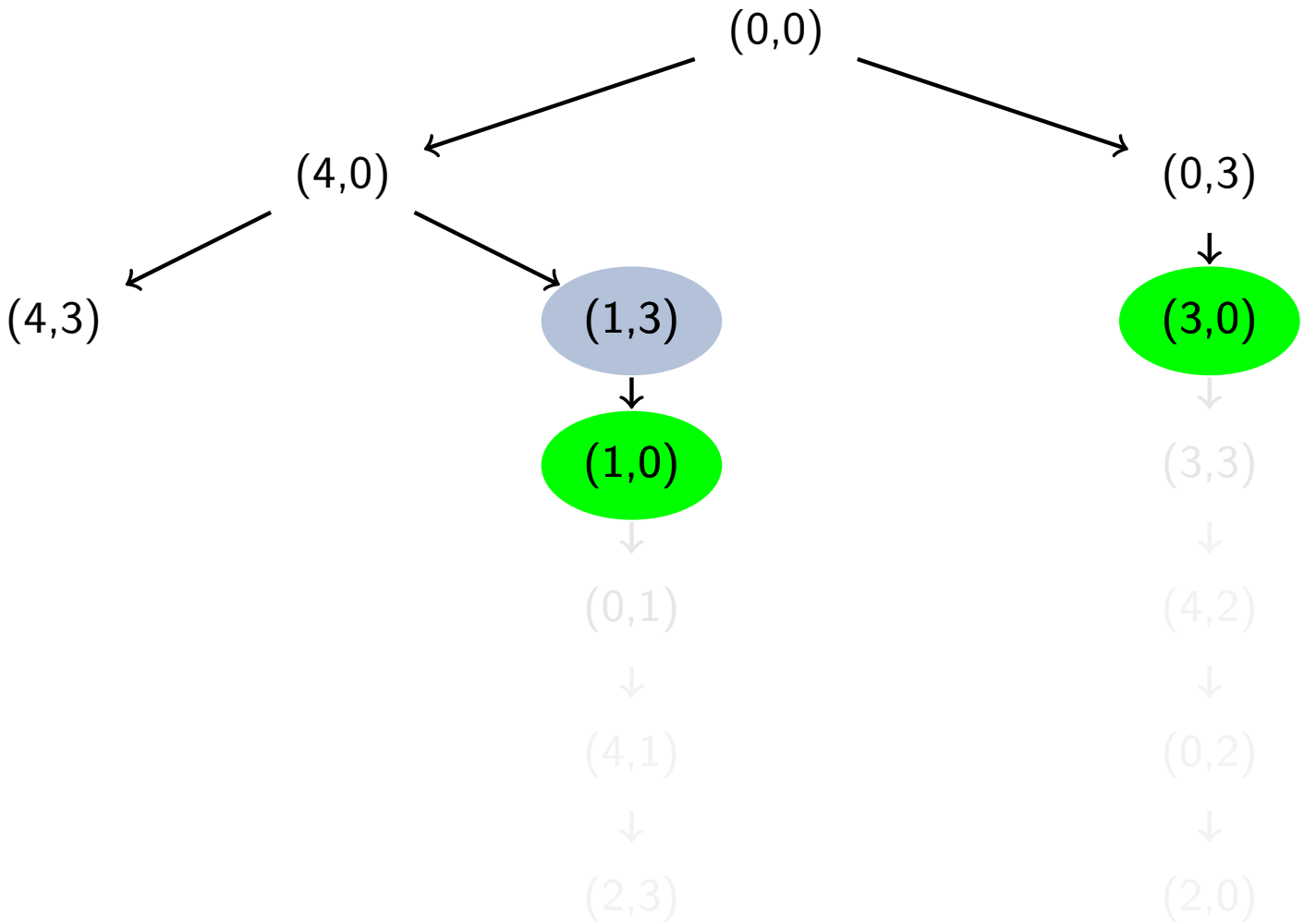
Search tree for Breadth-first-search

Water jug problem



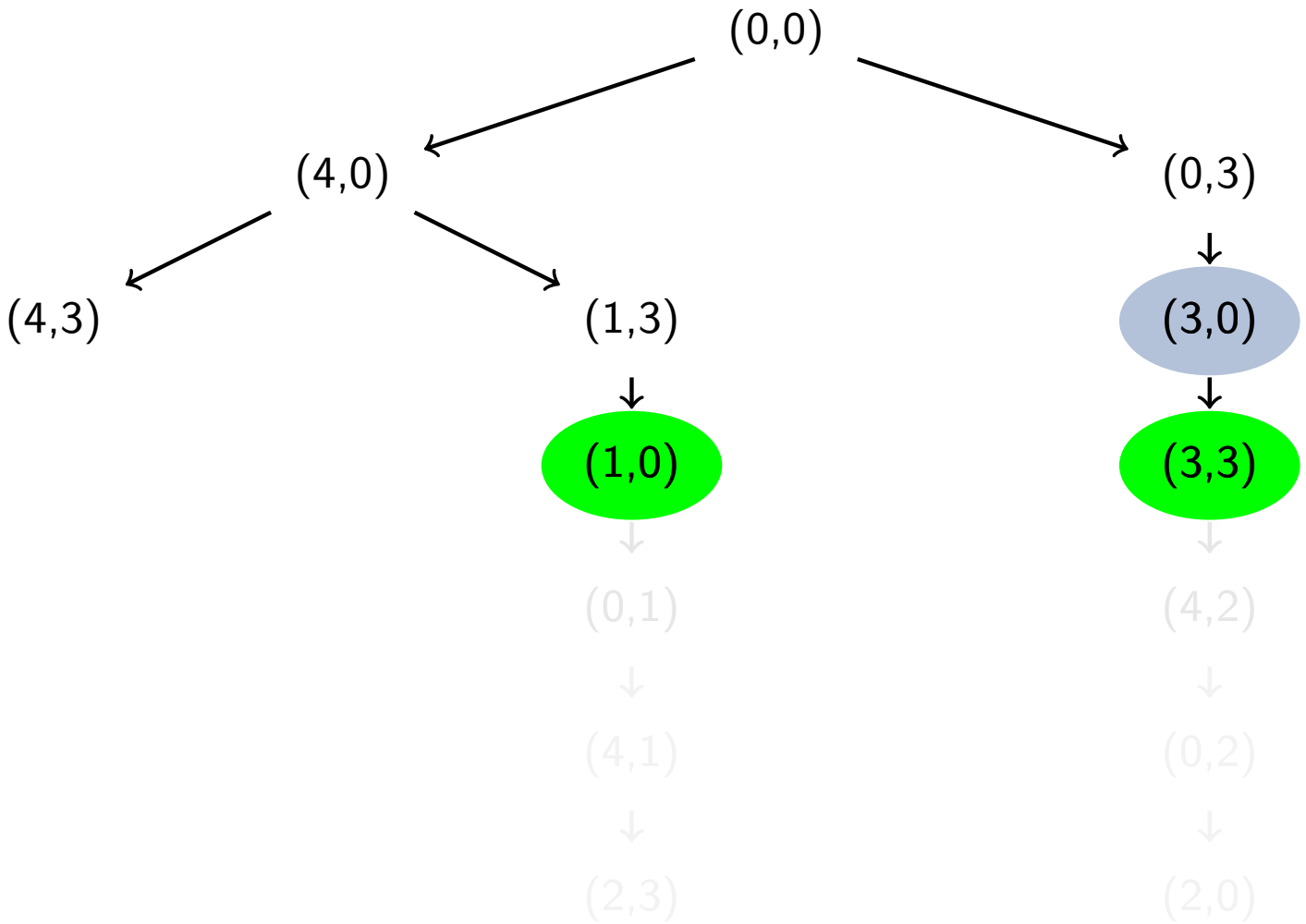
Search tree for Breadth-first-search

Water jug problem



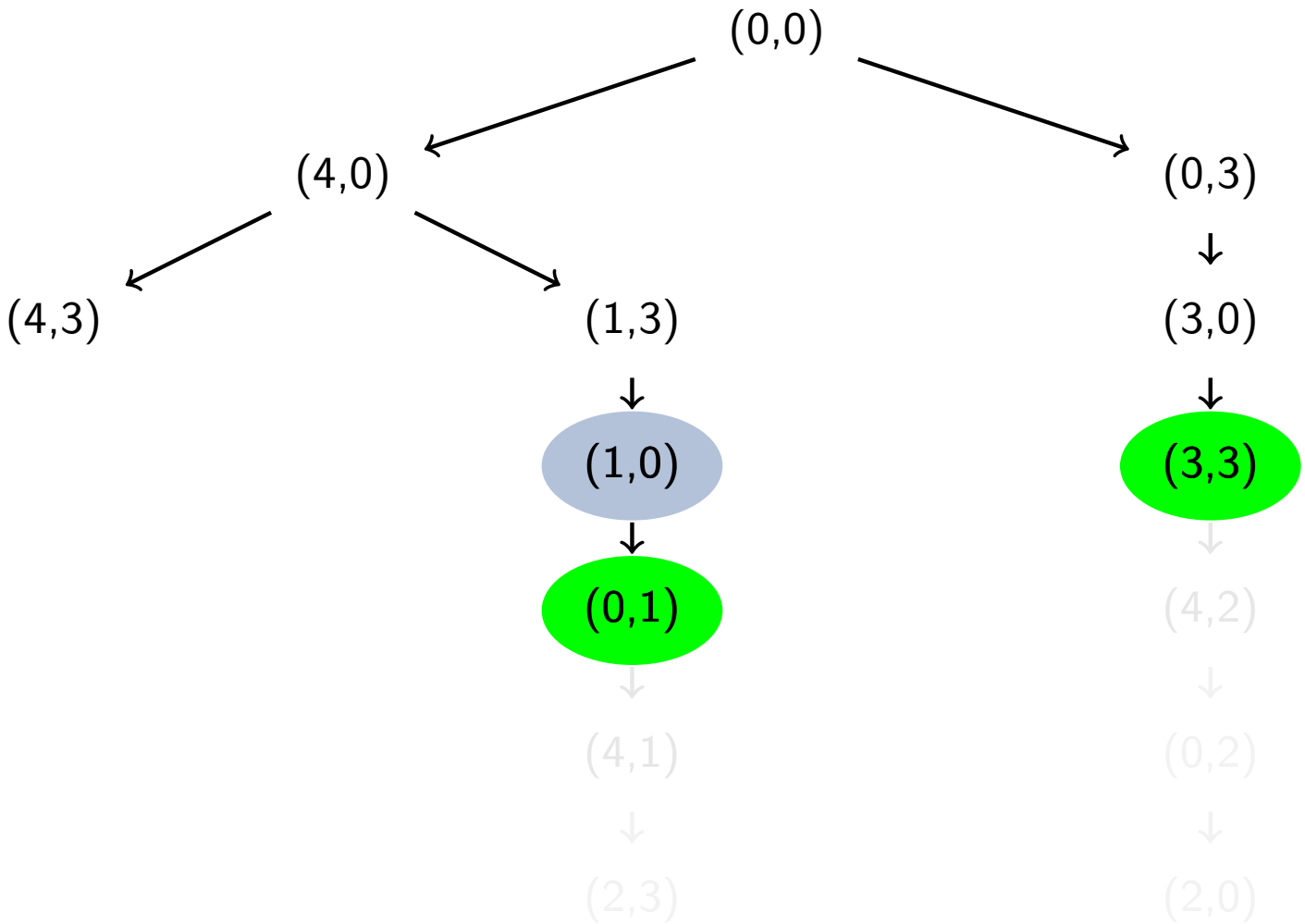
Search tree for Breadth-first-search

Water jug problem



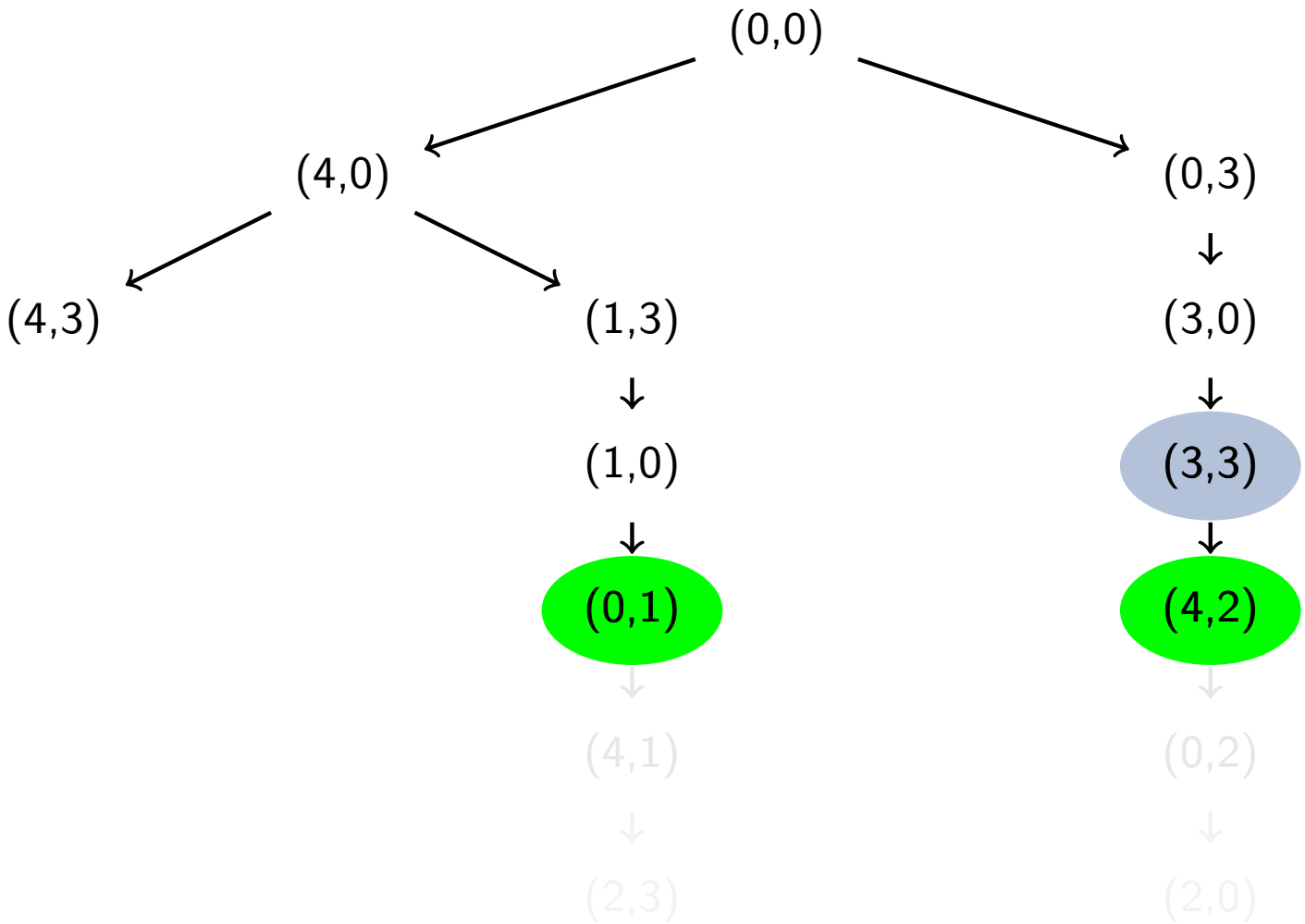
Search tree for Breadth-first-search

Water jug problem



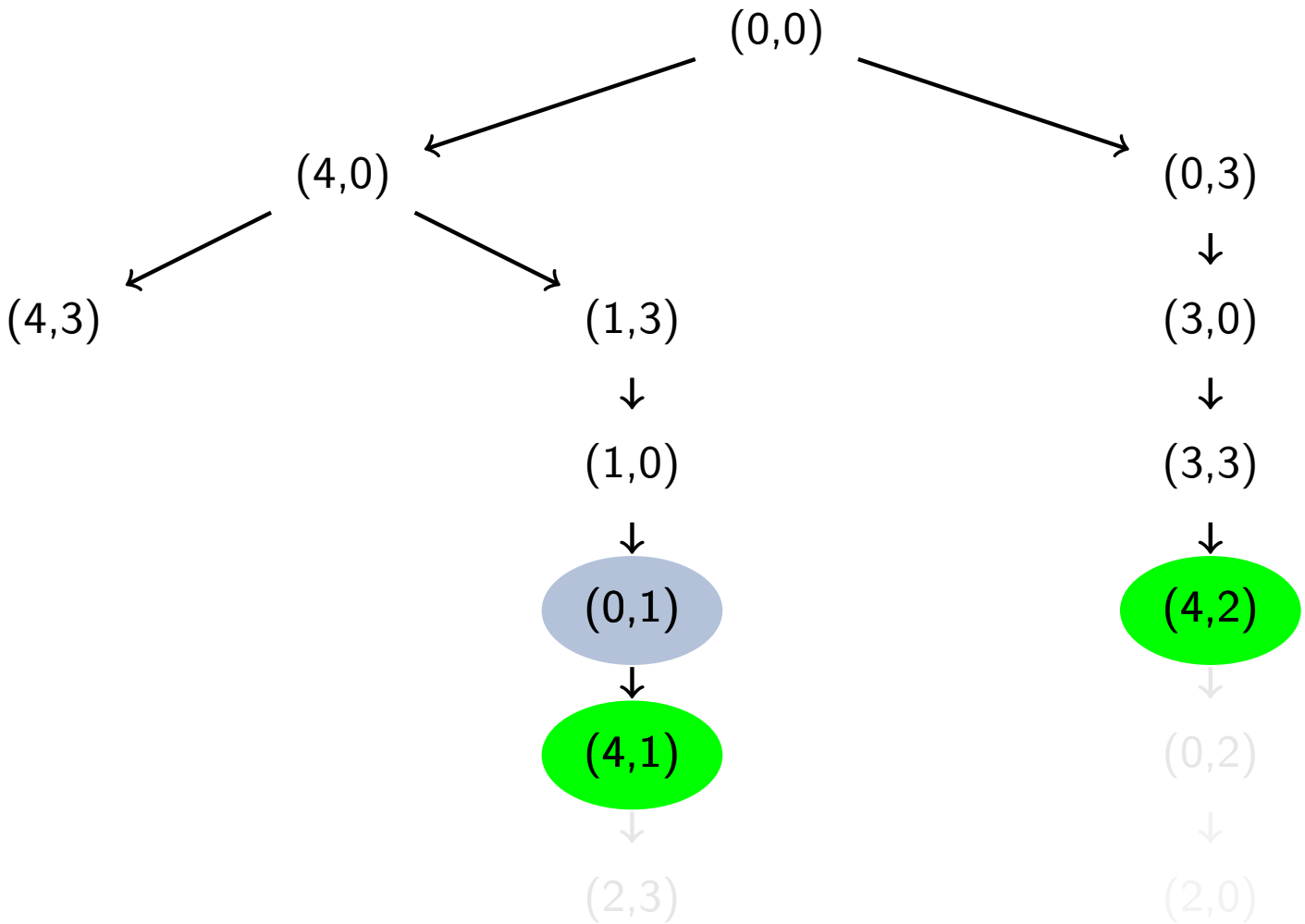
Search tree for Breadth-first-search

Water jug problem



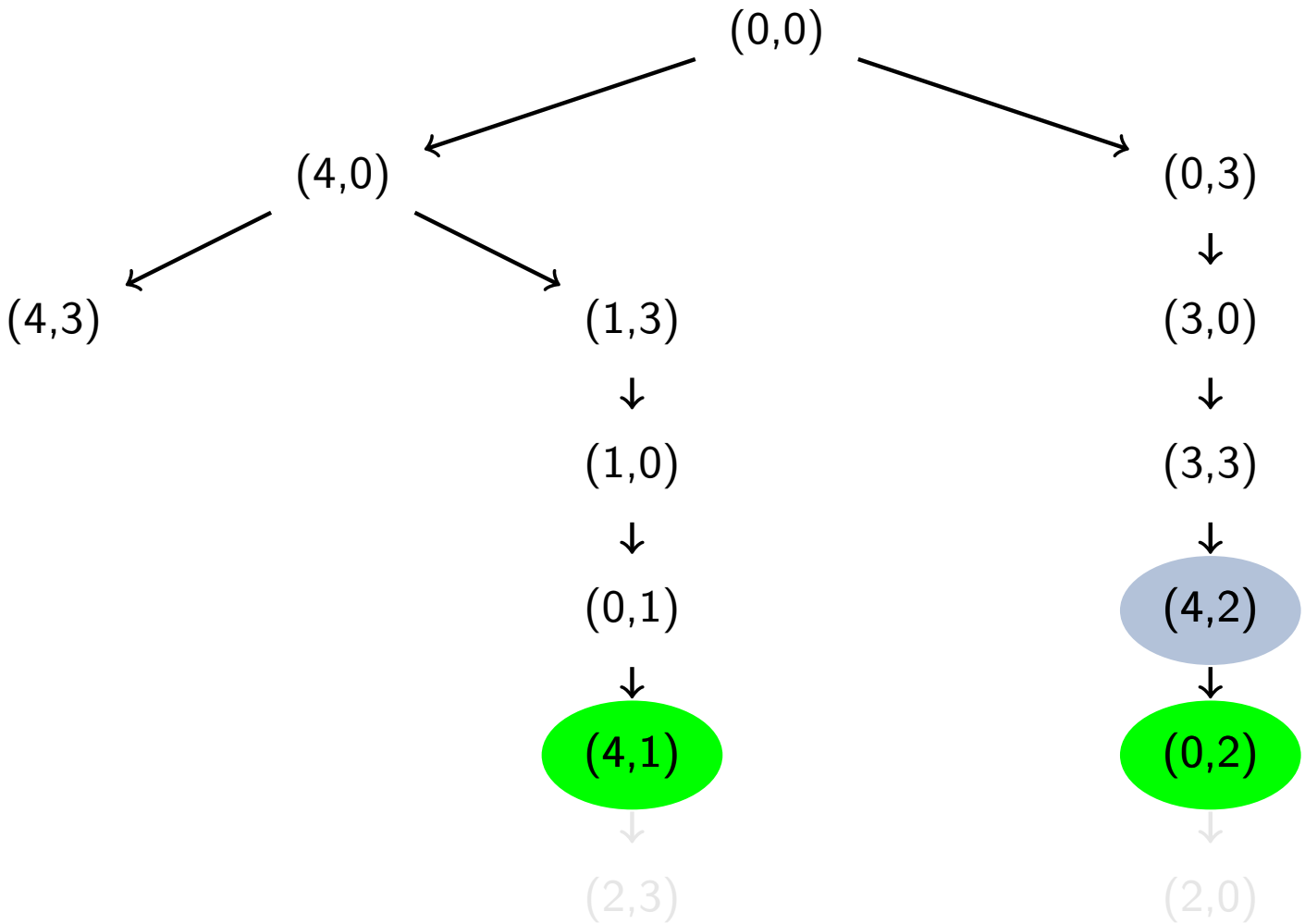
Search tree for Breadth-first-search

Water jug problem



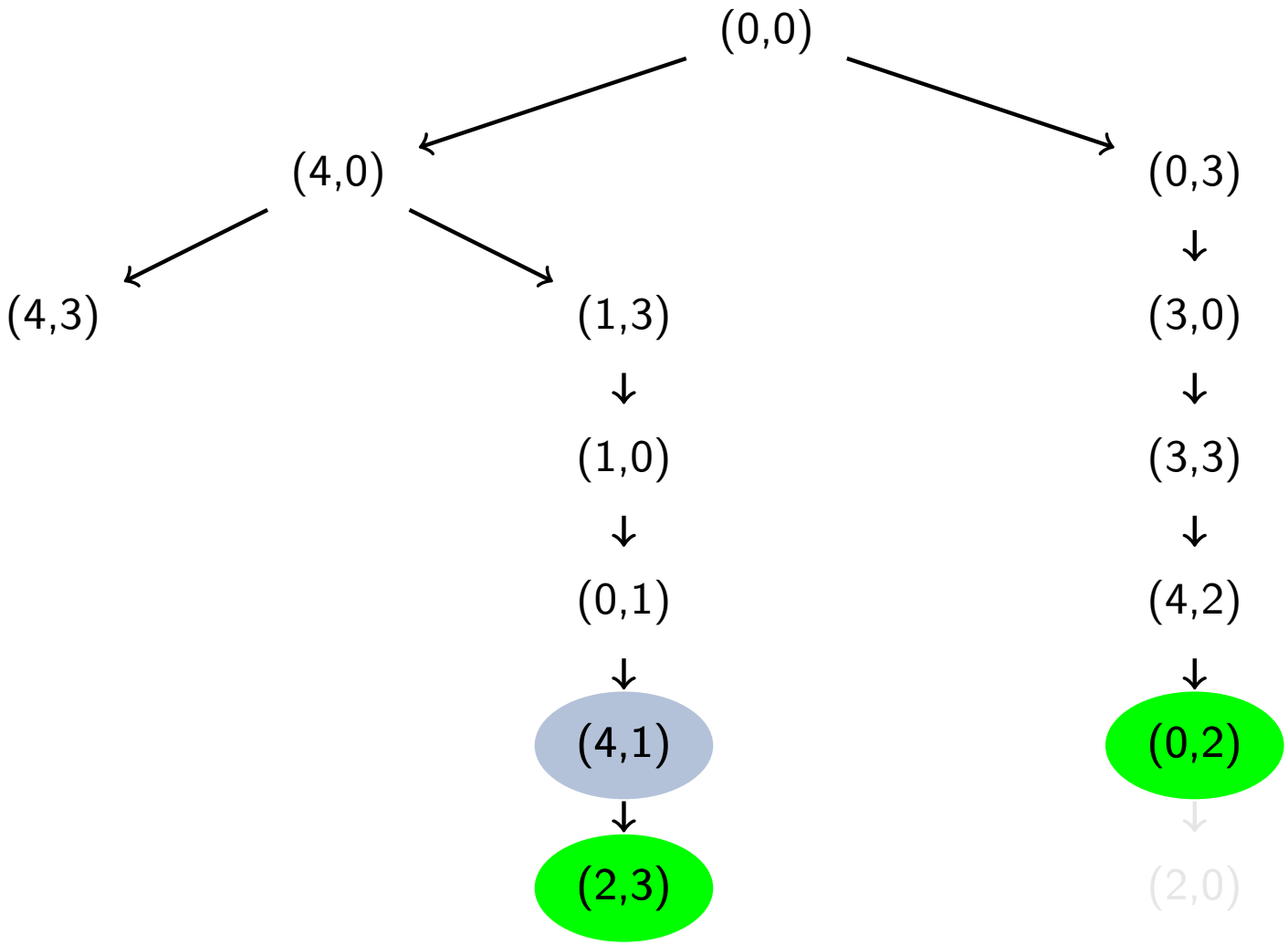
Search tree for Breadth-first-search

Water jug problem



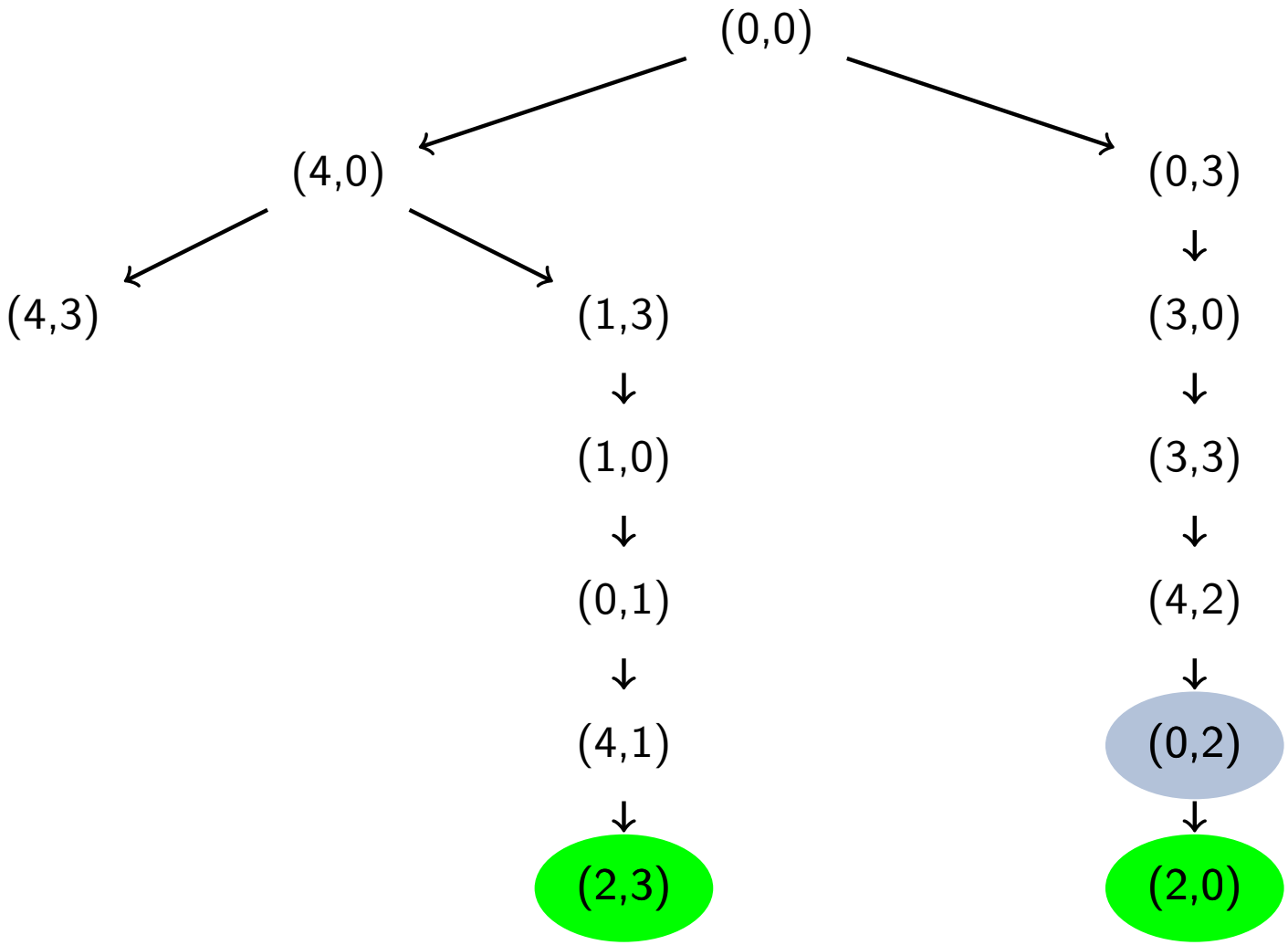
Search tree for Breadth-first-search

Water jug problem



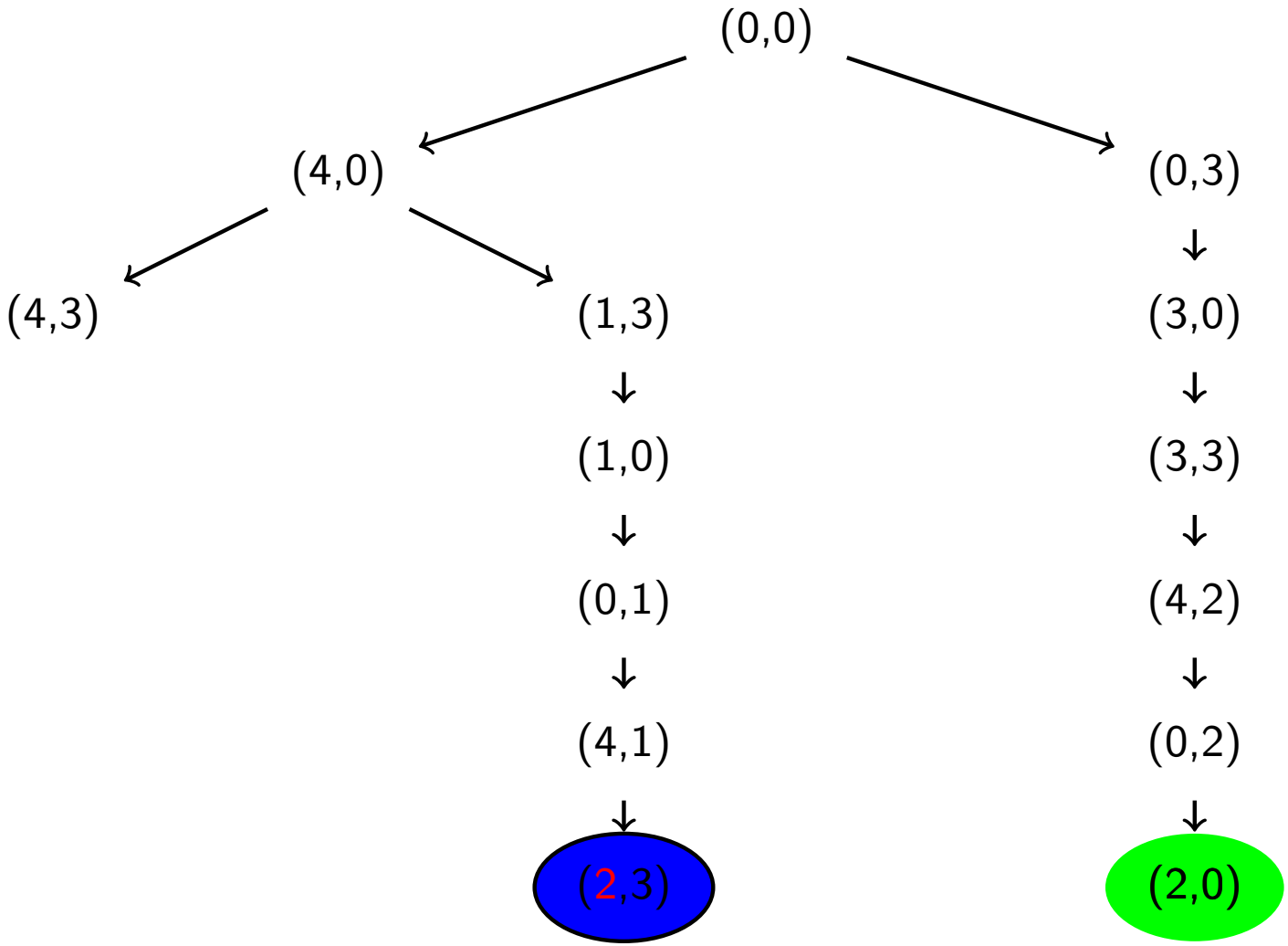
Search tree for Breadth-first-search

Water jug problem

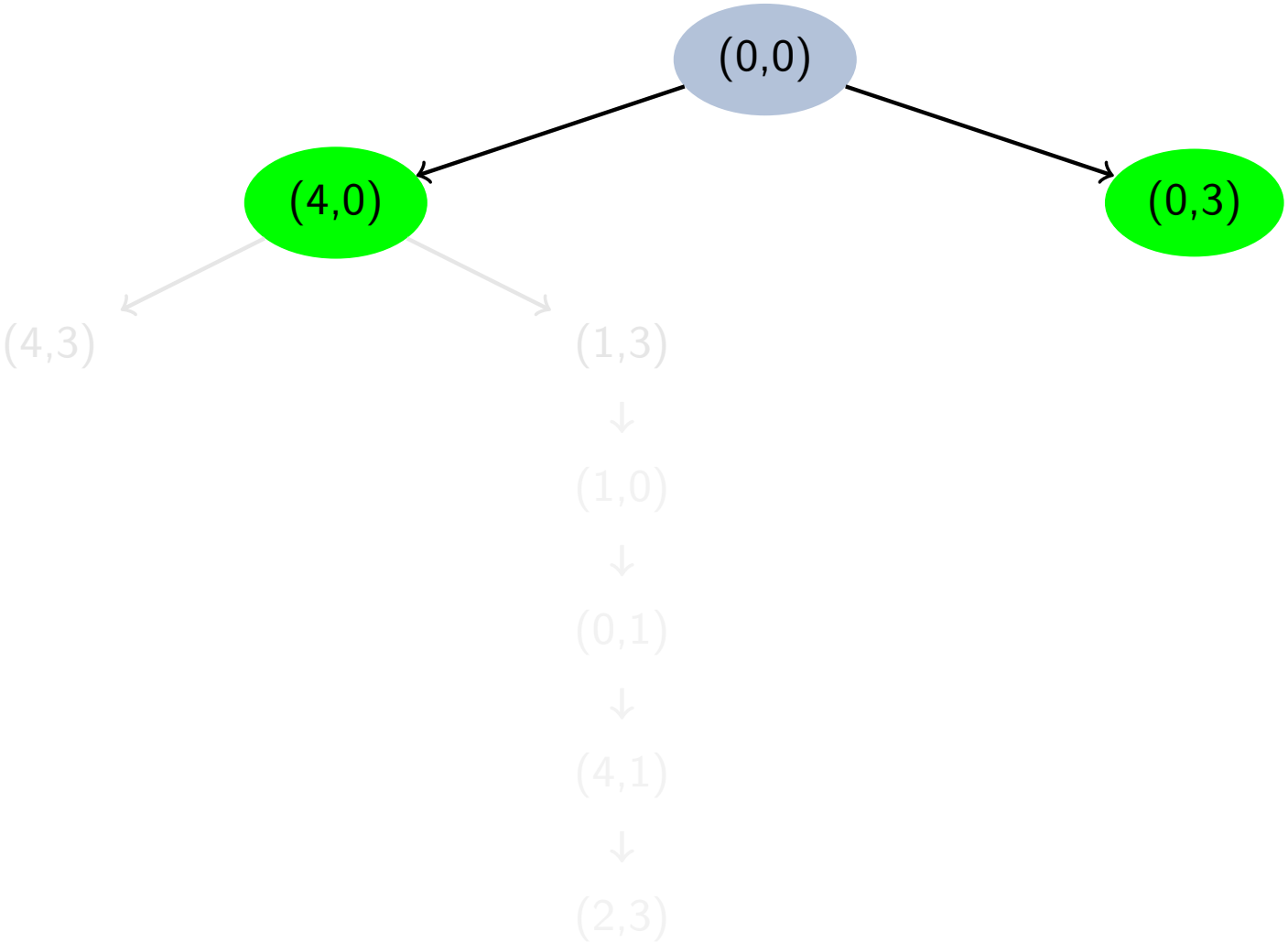


Search tree for Breadth-first-search

Water jug problem

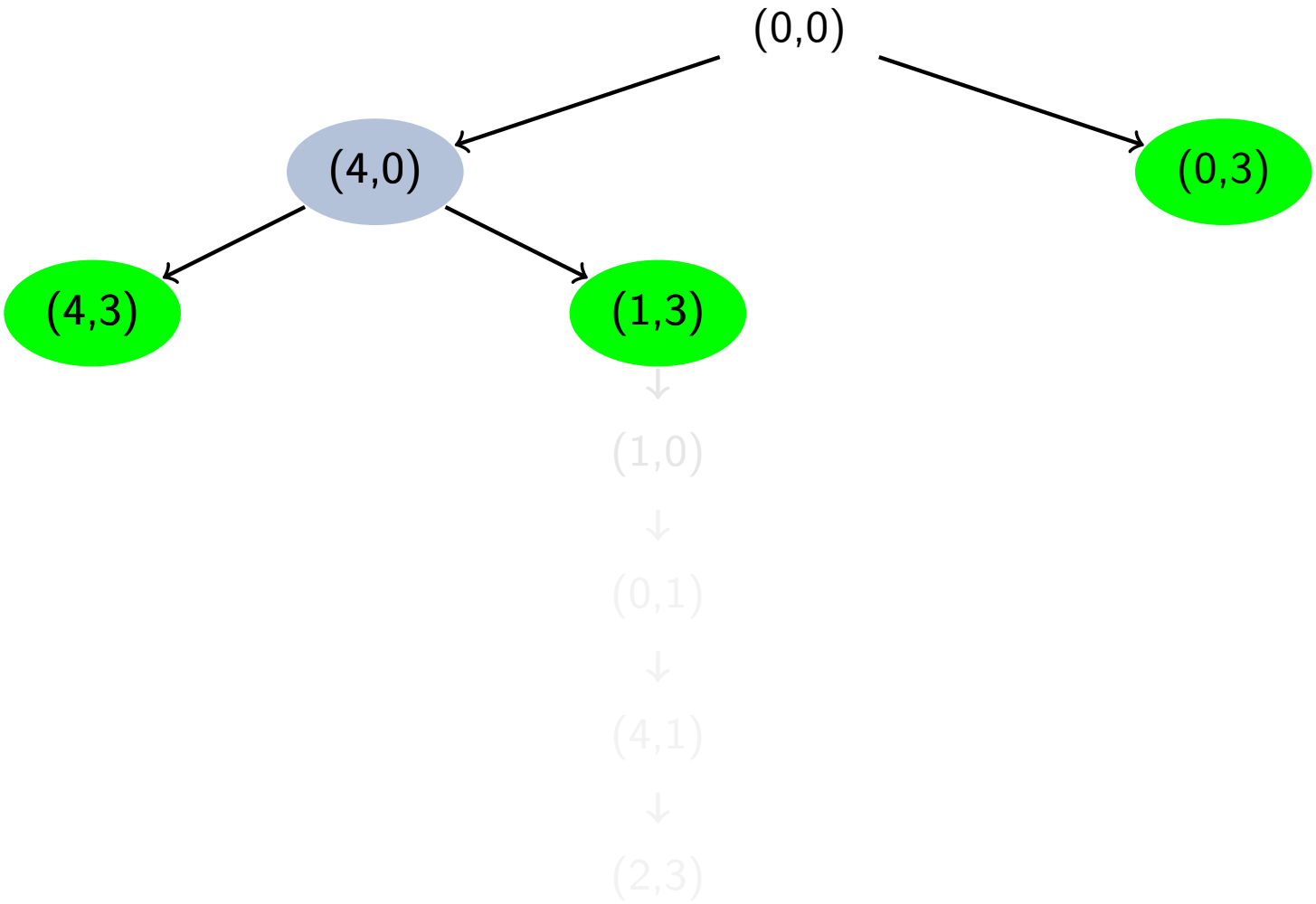


Search tree for Depth-first-search
Water jug problem



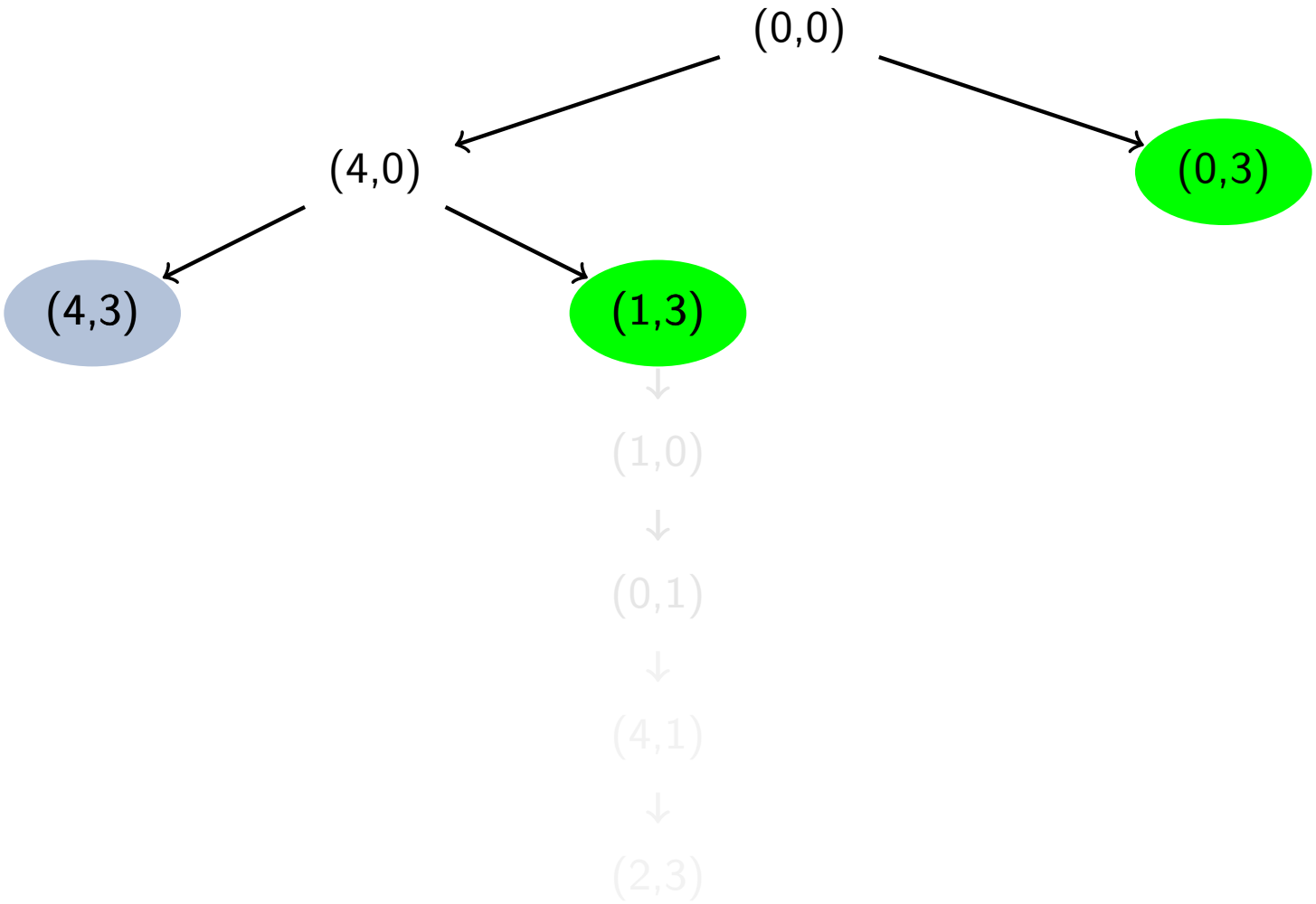
Search tree for Depth-first-search

Water jug problem



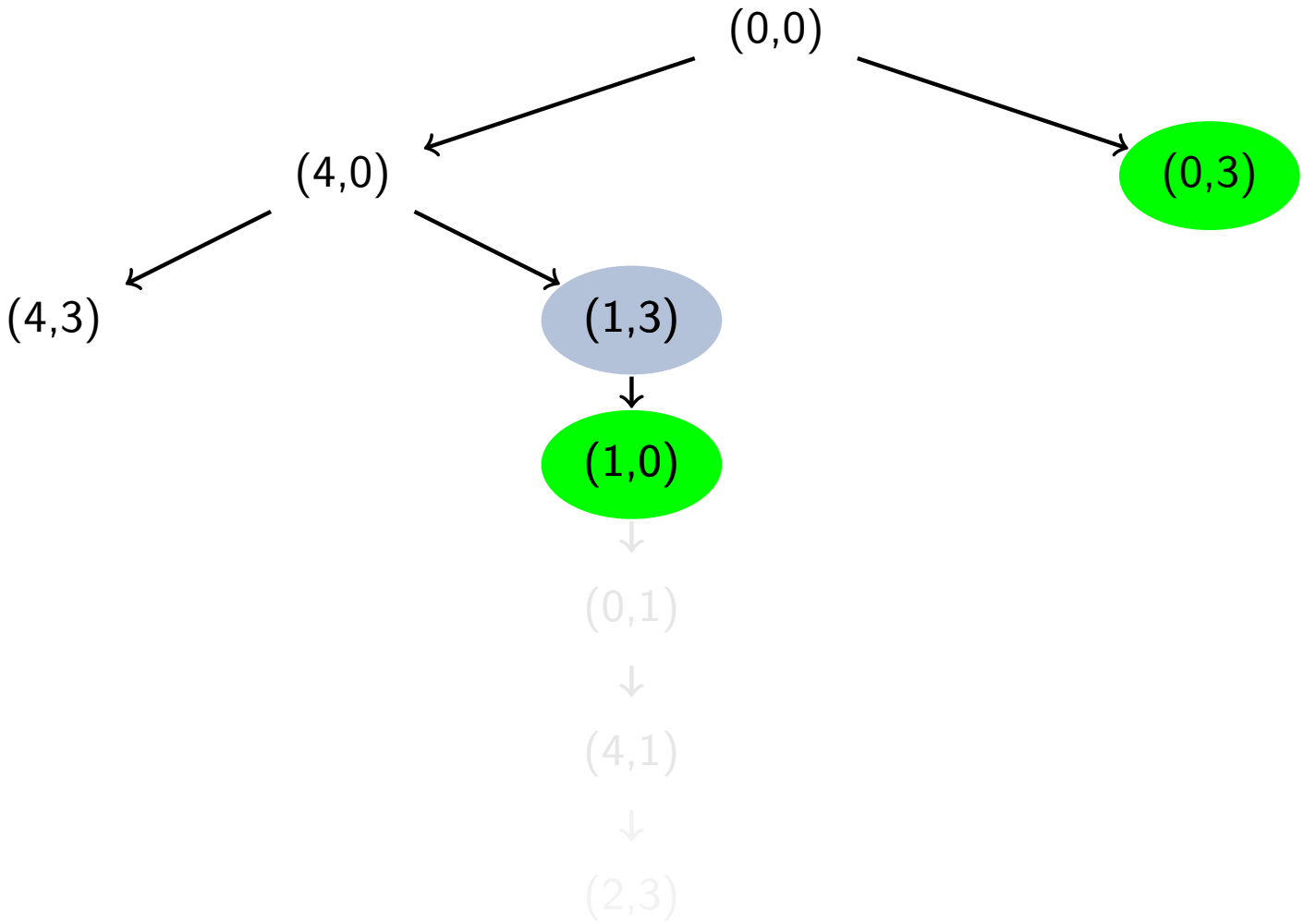
Search tree for Depth-first-search

Water jug problem



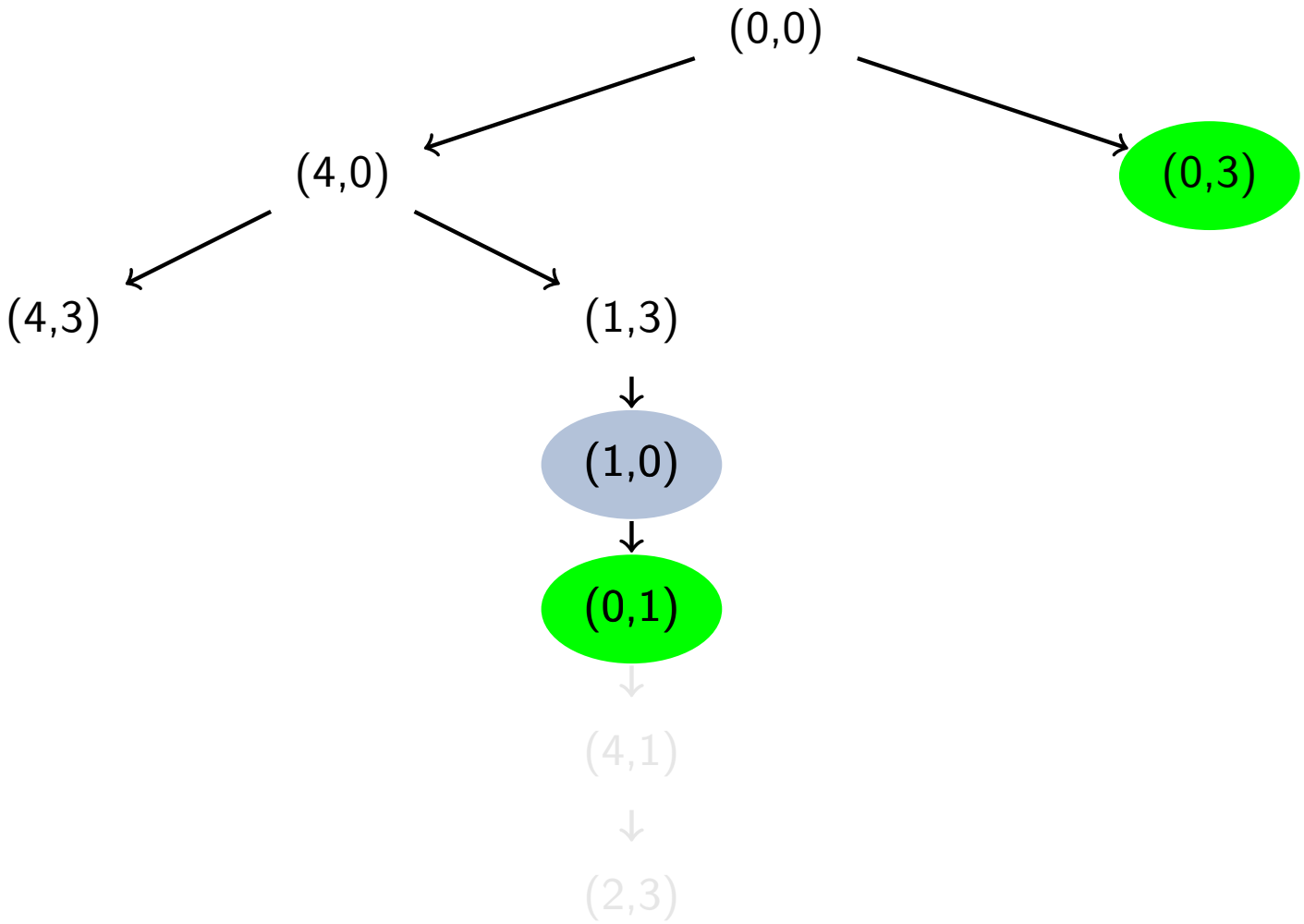
Search tree for Depth-first-search

Water jug problem



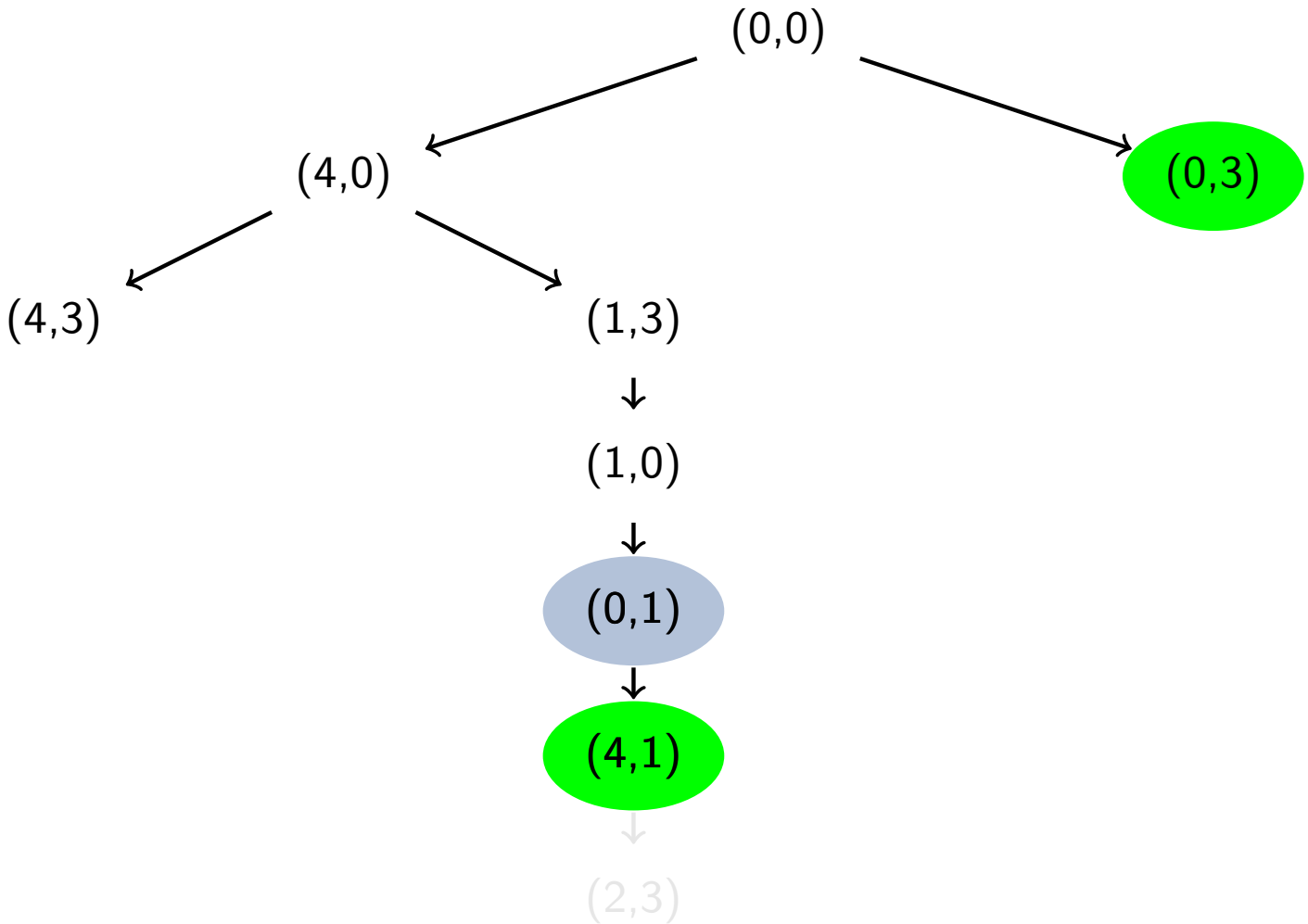
Search tree for Depth-first-search

Water jug problem



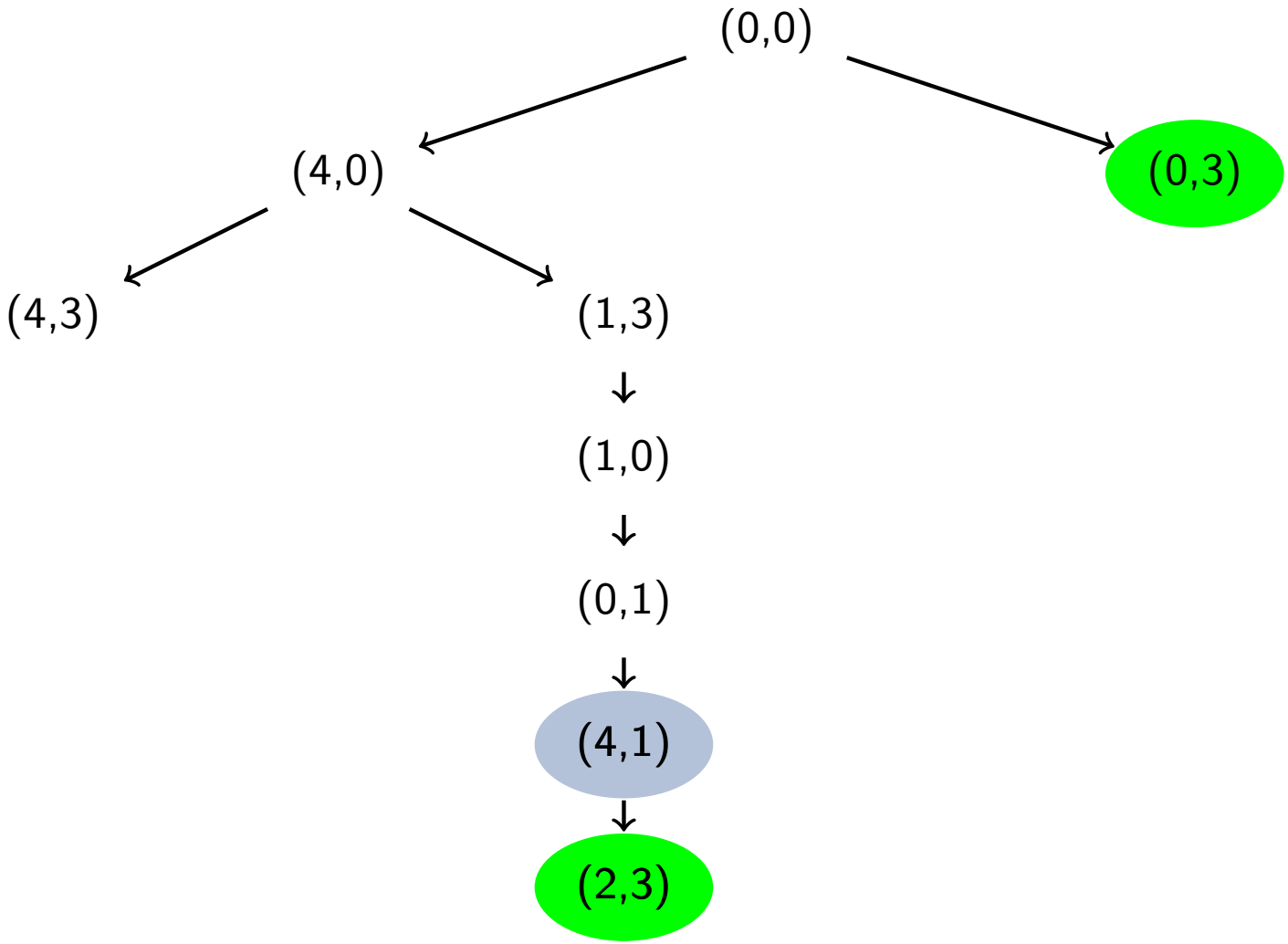
Search tree for Depth-first-search

Water jug problem



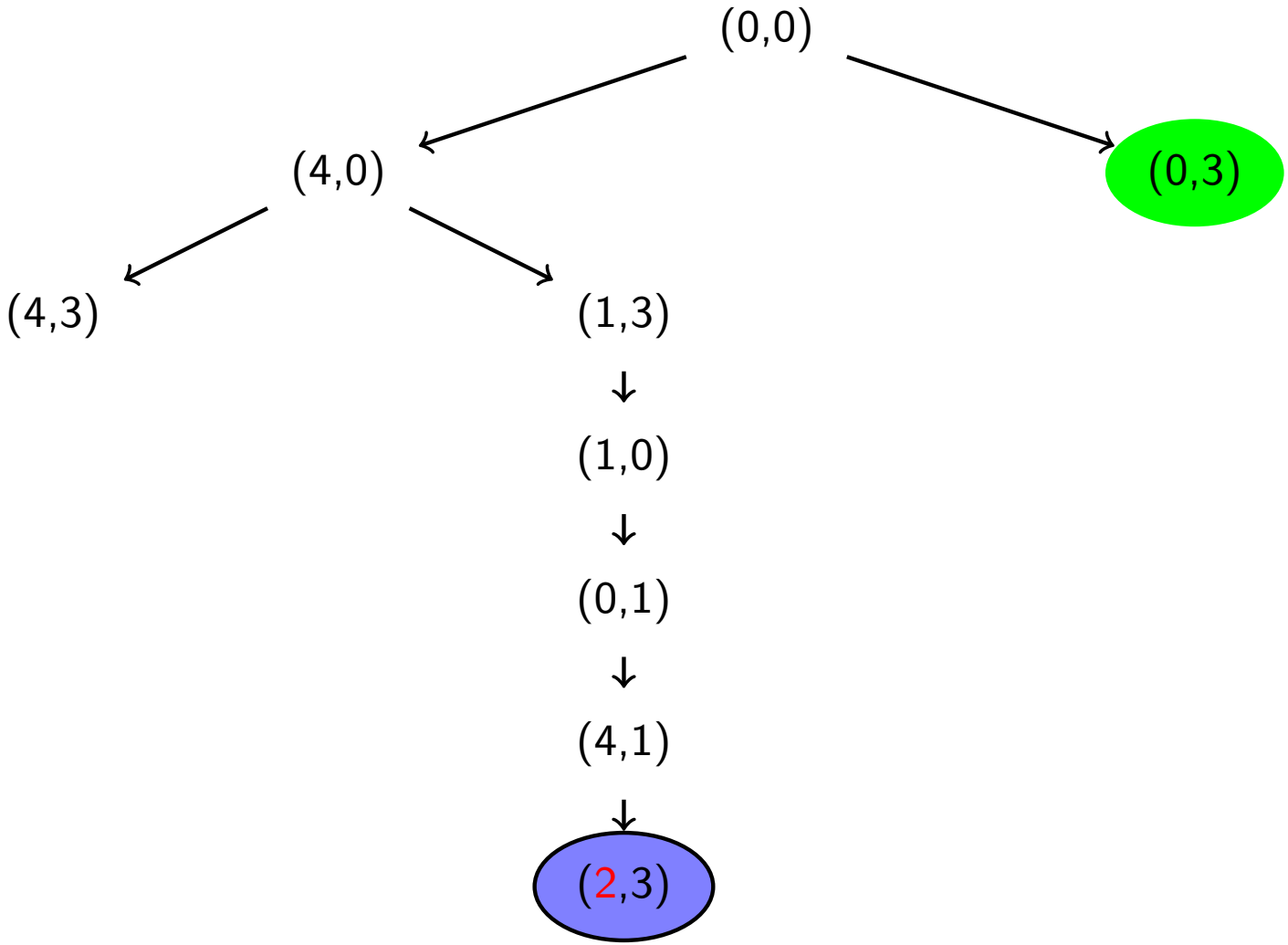
Search tree for Depth-first-search

Water jug problem



Search tree for Depth-first-search

Water jug problem



Recursive depth-first search (backtracking)

- Input: a state space representation, as previously described
  - We assume we have defined states, actions, ***INITIAL-STATE***, **IS-FINAL-STATE**, **ACTIONS**, **APPLY**
- Output: a sequence of actions that is a solution (or **FAIL** if there is no solution)
- Pseudocode:

```
FUNCTION DEPTH-FIRST-SEARCH()
 Return DFS-REC({},{*INITIAL-STATE*},*INITIAL-STATE*)

FUNCTION DFS-REC(SEQ,VISITED,CURRENT)
1. If IS-FINAL-STATE(CURRENT) then return SEQ
2. For each ACT in ACTIONS(CURRENT)
   3.1 Let S'=APPLY(ACT,CURRENT)
   3.2 If S' is not in VISITED
       3.2.1 Let RES equal to
             DFS-REC(SEQ·ACT,VISITED U {S'},S')
       3.2.2 If RES is not FAIL, return RES and halt
3. Return FAIL
```

Complexity issues

- Breadth-first:
  - Time complexity $O(b^d)$, where:
    - b: branching factor (maximum number of succesors of a state)
    - d: depth of the shortest solution.
  - Space complexity $O(b^d)$
- Depth-first:
  - Time complexity $O(b^m)$, where:
    - b: branching factor
    - m: maximum depth of any node.
  - Space complexity $O(b \cdot m)$ (needs a modified implementation).
- Due to a better space complexity, depth-first is usually preferred to breadth-first search
  - Although breadth-fist search guarantees the *shortest* solution
- Anyway, exponential time complexity (in the worst case) is a serious drawback

Limitations of uninformed (blind) search

- Exponential cost in time, makes it often not feasible in practice
- For example, assuming branching factor $r = 10$ and 10^6 nodes per sec.:

Depth	Nodes$\sim$	Time
2	110	0.11 ms.
4	11110	11 ms.
6	10^6	1.1 s.
8	10^8	2 min.
10	10^{10}	3 hours
12	10^{12}	13 days
14	10^{14}	3.5 years
16	10^{16}	350 years

- Problem: the search algorithms presented are *blind*
  - We are not using any information about how close is a state to a final state

Informed search

- Blind or uninformed search (Breadth-first, Depth-first,...): have no knowledge concerning how to reach the goal.
- Informed search: apply *knowledge* to the search process to make it more efficient.
- The knowledge will be given through a function *estimating* the "kindness" of states:
  - Give preference to better states.
  - Purpose: to reduce the search tree, gaining *efficiency* in practice.
  - Although that will depend on the "quality" of the estimation
- Those estimations are called *heuristics*

Concept of heuristic

- Heuristic:
  - from Greek *heuriskein*, to discover: *Eureka!*
  - Merriam-Webster Dictionary: "involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods".
  - Informally: method for solving problems that may not guarantee the solution, but works fine in general.
  - In our case, a heuristic will be a numerical function defined over states.
- Heuristic function, **`heuristic(state)`**:
  - Estimates the "distance" to the goal.
  - Always greater or equal than 0.
  - Value on final states: 0.
  - Value ∞ is allowed.
- All the specific knowledge about the problem that will be used is encoded in the heuristic function.

Example: heuristic for the 8-puzzle problem

- 8-puzzle problem:
  - **`heuristic(state)`**: number of pieces out of their correct positions (w.r.t. final state)
- Example:

2	8	3
1	6	4
7		5

H = 4

1	2	3
8		4
7	6	5

H = 0

Example: another heuristic for the 8-puzzle problem

- 8-puzzle problem (second heuristic):
  - **`heuristic(state)`**: sum of the Manhattan distances of each tile to its position in the final state
- Example:

2	8	3
1	6	4
7		5

H = 5

1	2	3
8		4
7	6	5

H = 0

Searching with heuristics

- We can use a heuristic to improve depth-first search:
  - Just explore the succesors of a give state in the order suggested by the heuristics
  - Prioritize states with low heuristic (that is, those *estimated* to be closer to the goal).
- Pseudocode:

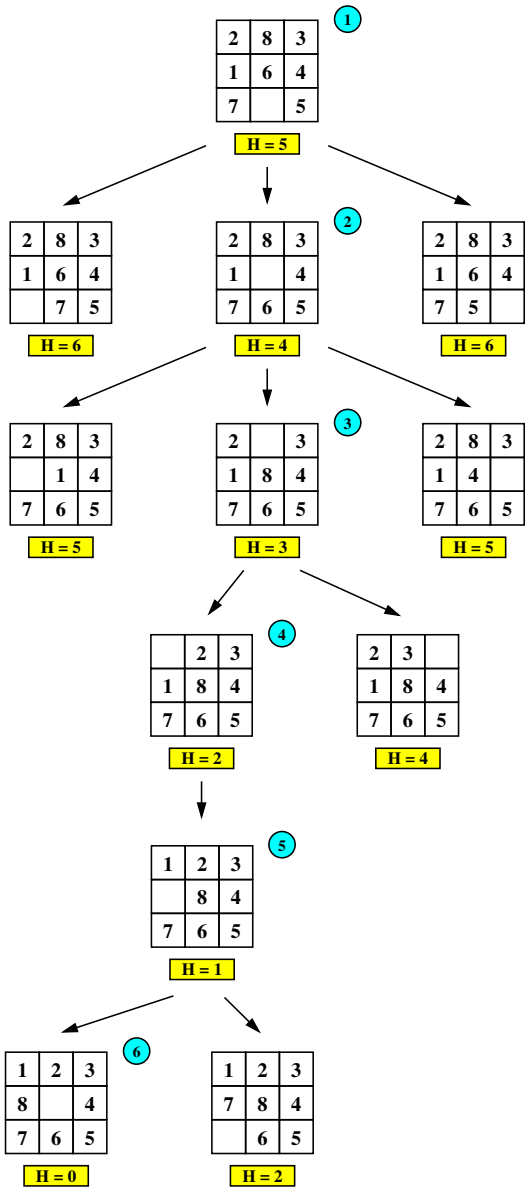
```
FUNCTION DEPTH-FIRST-SEARCH-H()
 Return DFS-REC-H({},{*INITIAL-STATE*},*INITIAL-STATE*)

FUNCTION DFS-REC-H(SEQ,VISITED,CURRENT)
1. If IS-FINAL-STATE(CURRENT) then return SEQ
2. For each ACT in HEURISTIC-SORT(ACTIONS(CURRENT))
   3.1 Let S'=APPLY(ACT,CURRENT)
   3.2 If S' is not in VISITED
       3.2.1 Let RES equal to
             DFS-REC(SEQ·ACT,VISITED U {S'},S')
       3.2.2 If RES is not FAIL, return RES and halt
3. Return FAIL
```

  - Where **HEURISTIC-SORT** is a function that sorts (in ascending order) the list of applicable actions according to the heuristic of the corresponding states

Search tree for Depth-first search with heuristic

8-puzzle problem (2nd heuristic)



Inventing heuristics

- Recall: estimation of the "steps" until reaching a final state
- Techniques to find heuristics
  - Relaxation of the problem
  - Combination of heuristics
  - Use of statistical information, experimentation
- Efficient evaluation of the heuristic function

Section 4

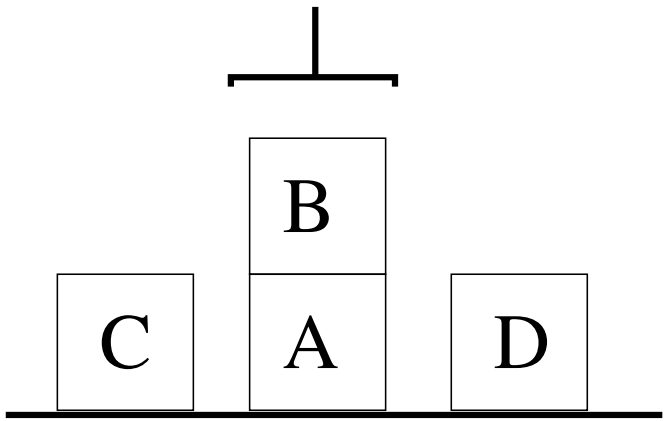
PDDL: a specific representation for planning problems

Practical planning in states spaces

- Some issues in real planning problems:
  - Description of the real-world states is extremely complex
  - Huge variety of possible actions, most of them irrelevant for achieving the final goal
  - Actions only have effect on a small piece of the world (*frame problem*)
  - Heuristics *independent* of the domain are needed
  - An action can be judged necessary without having decided yet all the previous actions (*minimum commitment*)
  - It is often advisable to *decompose* into simpler subproblems
- Idea: use logic to represent states, actions and goals; and use algorithms handling this representation

Example: the blocks world

- Classical example of planning domain.
- Elements involved:
  - A table.
  - A set of cube-shaped blocks.
  - A robot arm, able to hold one block at a time.
  - Each block can be either sitting on the table or piled on top of another block.

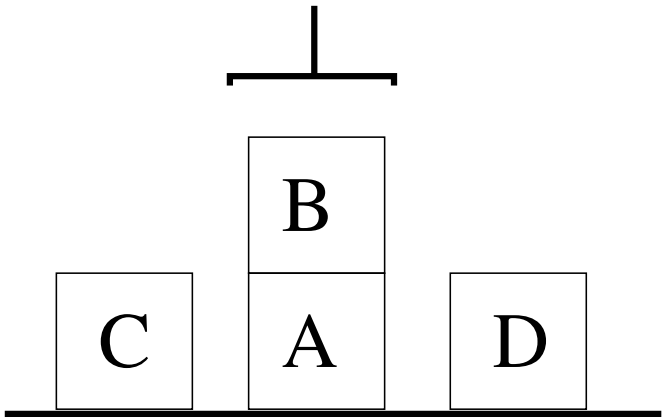


Logic formalism: PDDL language

- A language to represent planning problems:
  - Constants: objects in the world (capitalized)
  - Variables that represent any object (lower case)
  - Predicate symbols (to express properties of objects)
  - Symbols to represent the actions
- Terminology:
  - Atoms: formulas of the form $P(o_1, \ldots, o_n)$, where P is a predicate symbol and each o_i is either a constant or a variable (no function symbols)
  - Literals: atoms or negation of atoms (symbol – will be used as negation)
  - Closed atoms and literals: no variables on them
- States: conjunction of closed atoms
  - Closed world hypothesis: all atoms not explicitly stated is assumed to be false

Representation of states in the blocks world

- Description of a state:



CLEAR(B), CLEAR(C), CLEAR(D), FREEARM(),
ON(B,A), ONTHETABLE(C), ONTHETABLE(D), ONTHETABLE(A)

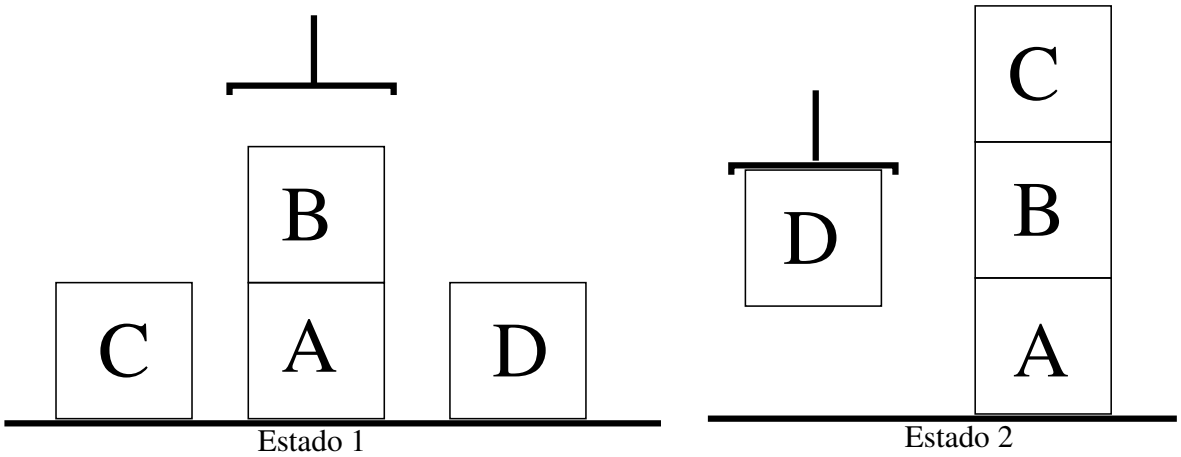
- Logic predicates used in this representation:

CLEAR(x), block x is clear (nothing on top of it).
FREEARM(), the arm is not holding any block.
ONTHETABLE(x), block x is sitting on the table.
ON(x,y), block x is sitting on top of y.
HOLD(x), block x is being hold by the arm.

Representation of goals

- Goals: description of final states.
- Goals are represented as conjunction of literals (use of variables is allowed)
  - Variables appearing in goals are interpreted as existentially quantified
- Satisfying (achieving) a goal:
  - A state satisfies a goal if it is possible to replace the variables in the goal by constants in such a way that the state includes all positive literals of the *instantiated goal* and none of the negative
  - A state is a final state if it satisfies the goal
- Important: checking if a state satisfies a goal is merely a symbolic calculation

Example of goals in the blocks world



- Examples of goals:
  - **ON(B,A), ONTHETABLE(A), -ON(C,B)**
  - **ON(x,A), CLEAR(x), FREEARM()**
  - **ON(x,A), -ON(C,x)**
  - **ON(x,A), ON(y,x)**

  The first three goals are satisfied by state 1 but not by state 2. The last one is satisfied by state 2 but not by state 1.

Description of actions schemes

- To deal with the *frame problem*, we only specify the *changes* produced by the action.
- An action is described by means of:
  - Its name and *all* the variables involved: $O(x_1, \ldots, x_n)$
  - Precondition: list of literals that need to be True in order for the action to be applicable.
  - Effects: list of literals indicating the changes to be produced when the action is executed.

Description of actions schemes

- In the effects list we distinguish:
  - Positive effects (or *addition* list): atoms that will start being True
  - Negative effects (or *erasing* list): atoms that will stop being True
- The use of variables implies that an action usually represents an *actions scheme*:
  - For each possible way of substituting variables by constants, we get a precise *action* (an *instance* of the scheme)

Example: actions in the blocks world

- Put a block on top of another:

```
PILE(x,y)
  Precond: CLEAR(y), HOLD(x)
  Effects: -CLEAR(y), -HOLD(x),
           FREEARM(), ON(x,y), CLEAR(x)
```

- Lift a block which was on top of another:

```
UNPILE(x,y)
  Precond: ON(x,y), CLEAR(x), FREEARM()
  Effects: -ON(x,y), -CLEAR(x), -FREEARM(),
           HOLD(x), CLEAR(y),
```

- Take a block from the table:

```
GRAB(x)
  Precond: CLEAR(x), ONTHETABLE(x), FREEARM()
  Effects: -CLEAR(x), -ONTHETABLE(x), -FREEARM(),
         HOLD(x)
```

- Release a block on the table:

```
RELEASE(x)
  Precond: HOLD(x)
  Effects: -HOLD(x),
         ONTHETABLE(x), FREEARM(), CLEAR(x)
```

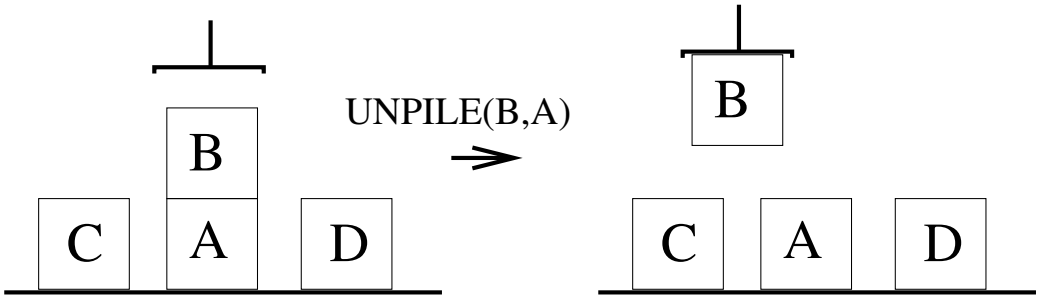
Applicability of an action

- An action is *applicable* on a state if the latter satisfies its precondition
  - If there are variables in the precondition, then the applicability is defined *with respect to the substitution* θ used to satisfy the precondition
  - Sometimes, to simplify, the substitution will appear implicitly when refering to the action
  - For example, we write **`UNPILE(A,B)`** to refer to the action **`UNPILE(x,y)`** using the substitution **`[x/A,y/B]`**
  - On a given state, there could be several applicable actions coming from the same actions scheme

Result of applying an action

- The result of applying an applicable action (w.r.t. a substitution θ) over a state E is the state obtained by:
  - Eliminate from E the atoms, instantiated by θ, corresponding to the list of negative effects (if they were present)
  - Add to E the atoms, instantiated by θ, corresponding to the list of positive effects (if they were not present)

Example of application of an action (I)



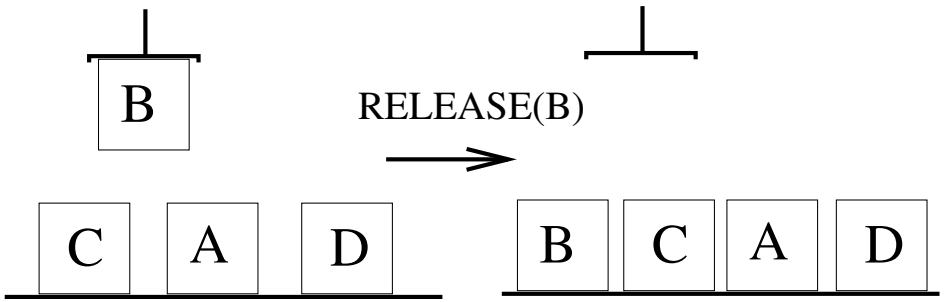

```
* State before applying UNPILE(B,A):
E = {CLEAR(B), CLEAR(C), CLEAR(D), FREEARM(),
     ON(B,A), ONTHETABLE(C), ONTHETABLE(D), ONTHETABLE(A)}

* Preconditions of UNPILE(B,A):
Preconds = {ON(B,A), CLEAR(B), FREEARM()}
------ Conditions satisfied by the state -------
------    (thus action is applicable)    -------

* Effects of UNPILE(B,A):
Effec- = {ON(B,A), CLEAR(B), FREEARM()}
Effec+ = {HOLD(B), CLEAR(A)}

* State after applying UNPILE(B,A):
E' = (E - Efec-) U Efec+ =
{CLEAR(C), CLEAR(D), ONTHETABLE(C),
 ONTHETABLE(D), ONTHETABLE(A), HOLD(B), CLEAR(A)}
```

Example of application of an action (II)




```
* State befor applying RELEASE(B):
E = {CLEAR(C), CLEAR(A), CLEAR(D), ONTHETABLE(C),
     ONTHETABLE(A), ONTHETABLE(D), HOLD(B)}

* Preconditions of RELEASE(B):
Preconds = {HOLD(B)}
------ Conditions satisfied by the state -------
------    (thus action is applicable)    -------

* Effects of RELEASE(B):
Effec- = {HOLD(B)}
Effec+ = {ONTHETABLE(B), FREEARM(), CLEAR(B)}


* State after applying RELEASE(B):
E' = (E - Efec-) U Efec+ =
{CLEAR(C), CLEAR(A), CLEAR(D), ONTHETABLE(C),
 ONTHETABLE(A), ONTHETABLE(D),ONTHETABLE(B), FREEARM(),
 CLEAR(B)}
```

Plans and solutions

- Plan: (ordered) sequence of actions
  - First one applicable on the initial state and each one of the rest applicable on the result of the previous one
- Solution: plan that starting from the initial state obtains a state which satisfies the goal

Example: changing a flat tire

- Language:
  - Objects: **SPARE-TIRE**, **FLAT-TIRE**, **AXLE**, **TRUNK**, **GROUND**
  - Predicate: **AT(-,-)**
- Initial state:

  AT(FLAT-TIRE,AXLE),AT(SPARE-TIRE,TRUNK)
- Final state:

  AT(SPARE-TIRE,AXLE)

Actions for changing a flat tire

- Take the spare tire out of the trunk:

```
REMOVE(SPARE-TIRE,TRUNK)
 Precond: AT(SPARE-TIRE,TRUNK)
 Effects: AT(SPARE-TIRE,GROUND), -AT(SPARE-TIRE,TRUNK)
```

- Remove the flat tire from the axle:

```
REMOVE(FLAT-TIRE,AXLE)
 Precond: AT(FLAT-TIRE,AXLE)
 Effects: -AT(FLAT-TIRE,AXLE), AT(FLAT-TIRE,GROUND)
```

- Put the spare tire on the axle:

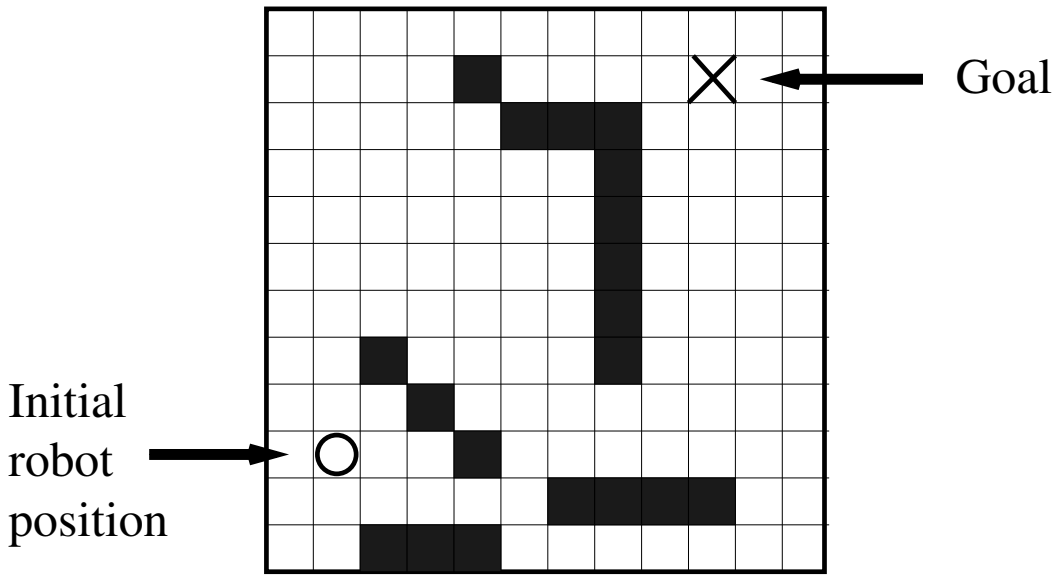
```
PUT-ON(SPARE-TIRE,AXLE)
 Precond: -AT(FLAT-TIRE,AXLE), AT(SPARE-TIRE,GROUND)
 Effects: -AT(SPARE-TIRE,GROUND), AT(SPARE-TIRE,AXLE)
```

- Leave the car alone until next morning:

```
LEAVEALONE()
 Precond: {}
 Effects: -AT(SPARE-TIRE,GROUND), -AT(SPARE-TIRE,AXLE),
         -AT(SPARE-TIRE,TRUNK), -AT(FLAT-TIRE,AXLE),
         -AT(FLAT-TIRE,GROUND)
```

Example: robot moving along a grid

- A robot is asked to move from an initial position to final position, being able to move along a grid
  - 8 possible moves: N, S,E,W,NW,NE,SW,SE
  - Some of the cells of the grid contain unavoidable obstacles



Representation of the problem of the robot in the grid

- Language:
  - Constants: numbers indicating horizontal and vertical coordinates
  - Predicates: **`ROBOT-AT(-,-)`** and **`FREE(-,-)`**
- Initial state (empty cells having no obstacles, and current position of the robot):

```
FREE(1,1),...,FREE(6,2),FREE(11,2),...,FREE(12,12),
ROBOT-AT(2,3).
```

- Goal: **`ROBOT-AT(10,11)`**
- Actions (the remaining seven are analogous to this one):

```
MOVE-SW(x,y)
  Precond: ROBOT-AT(x,y), FREE(x+1,y-1)
  Effects: -ROBOT-AT(x,y), -FREE(x+1,y-1),
           ROBOT-AT(x+1,y-1), FREE(x,y)
```

  - For this problem, a small extension of the semantics is required, since we need function symbols ($+$ and $-$), and we need the satisfiability test to sum or substract

Extensions to the PDDL formalism

- There exist planning systems that use a richer representation language
- For example:
  - Variables having types (this extension does not provide higher expressive power)
  - Use of function symbols
  - Use of the equal predicate
  - Evaluation of functions when applying actions
  - Disjunctions on the preconditions
- In general there exists a compromise between language expresiveness and simplicity of the algorithms handling the representation

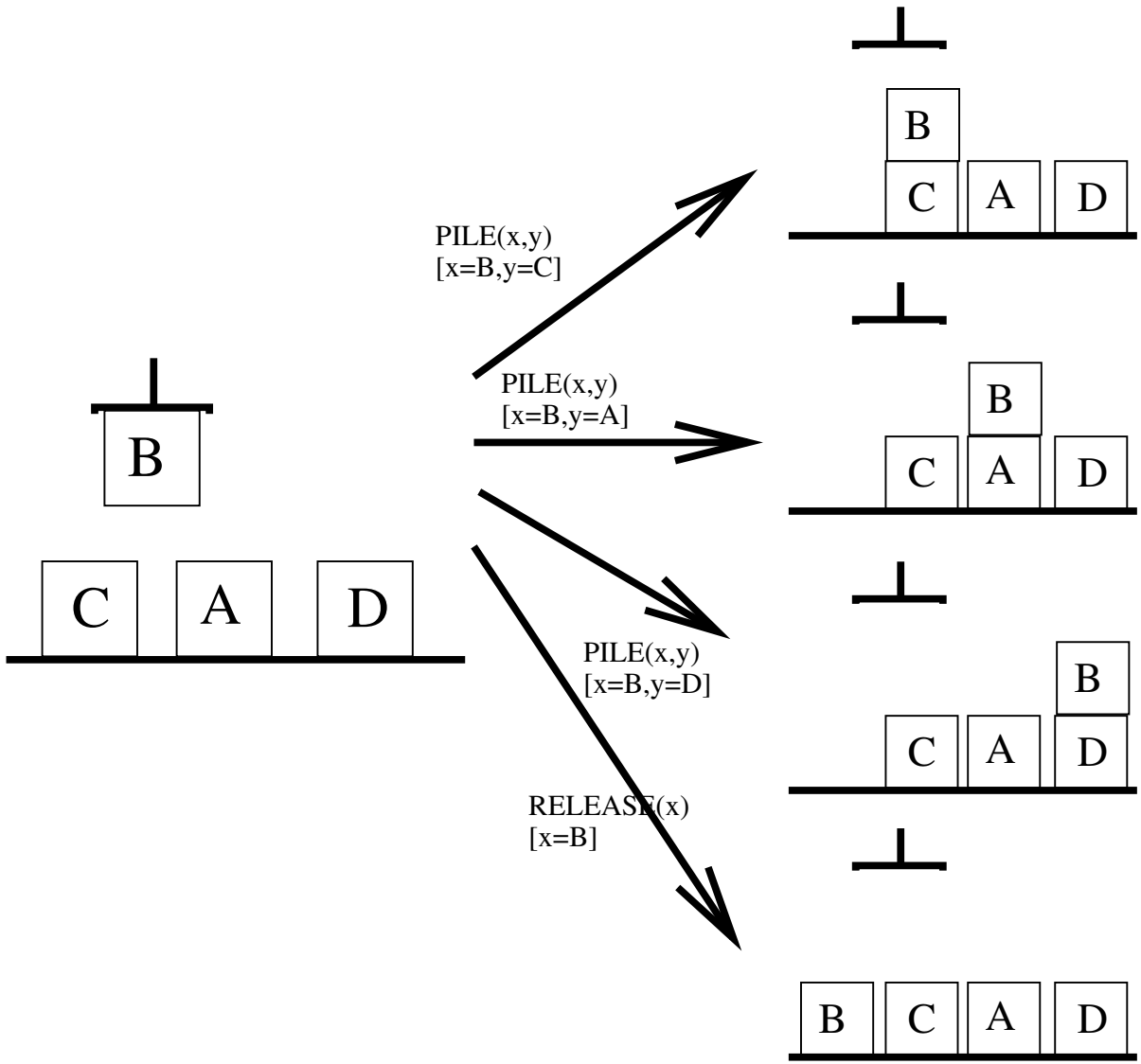
Section 5

Forward and Backward search in the state space

Search in states space

- A planning problem can be formalized in the framework of states space search:
  - States described by means of lists of closed atoms.
  - Actions as lists of preconditions and effects.
  - Function **`is-final-state`** described as a goal.
- The search for plans could be carried out by the search algorithms discusse: breadth-first, depth-first,. . .
- The use of the logic formalism allows to define *domain-independent* heuristics
  - For example, number of literals in the goal that still need to be satisfied by a state

Obtaining successors



Obtaining successors

- Example:

```
S = {CLEAR(C), CLEAR(A), CLEAR(D), ONTHETABLE(C),
     ONTHETABLE(A), ONTHETABLE(D), HOLD(B)}
Four possible applicable actions:
 1), 2) and 3): A=PILE(x,y) with THETA=[x=B,y=C],
                THETA=[x=B,y=A] and THETA=[x=B,y=D], resp.
 4): A=RELEASE(x) with substitution THETA=[x=B].

Successors:

S1= { CLEAR(A), CLEAR(D), ONTHETABLE(C), ONTHETABLE(A),
      ONTHETABLE(D), FREEARM(), CLEAR(B), ON(B,C)}

S2= { CLEAR(C), CLEAR(D), ONTHETABLE(C), ONTHETABLE(A),
      ONTHETABLE(D), FREEARM(), CLEAR(B), ON(B,A)}

S3= { CLEAR(A), CLEAR(C), ONTHETABLE(C), ONTHETABLE(A),
      ONTHETABLE(D), FREEARM(), CLEAR(B), ON(B,D)}

S4= { CLEAR(A), CLEAR(C), CLEAR(D), ONTHETABLE(C),
      ONTHETABLE(A), ONTHETABLE(D), FREEARM(), CLEAR(B),
      ONTHETABLE(B)}
```

(Forward) Depth-first search with heuristic

```
FUNCTION DEPTH-FIRST-SEARCH-H(INITIAL-STATE,GOAL,ACTIONS)
 Return DFS-H-REC({},{INITIAL-STATE},INITIAL-STATE,GOAL,ACTIONS)

FUNCTION DFS-H-REC(PLAN,VISITED,CURRENT,GOAL,ACTIONS)
1. If CURRENT satisfies GOAL, then return PLAN
2. Let APPLICABLE be the list of instantiated actions from ACTIONS,
   that are applicable to CURRENT and whose application yields a
   new state not present in VISITED
3. Let ORD-APPLICABLE be equal to HEURISTIC-SORT(APPLICABLE)
4. For each ACT in ORD-APPLICABLE
   4.1 Let S' be the result of applying ACT to CURRENT
   4.2 Let RES equal to
         DFS-H-REC(PLAN·ACT,VISITED U {S'},S',GOAL,ACTIONS)
   4.3 If RES is not FAIL, return RES and halt
5. Return FAIL
```

- This algorithm has already been defined
  - But now in the PDDL context
- In this algorithm the function **HEURISTIC-SORT** have to be defined
  - In general, it is defined by using an heuristic **H** over the states, that estimates the number of steps to reach the goal

Backward Search

- With a bad heuristic, forward search has the inconvenient of too much branching
  - Many applicable actions, but most of them being irrelevant for the goal
- Alternative approach: to proceed backwards, using the goal as a guide
  - Starts on the goal, actions are inversely applied and we try to reach the initial state
  - Nodes of the search tree: goals
- Key: only actions **relevant** for the goal are inversely applied

Actions relevant for the goal

Definition

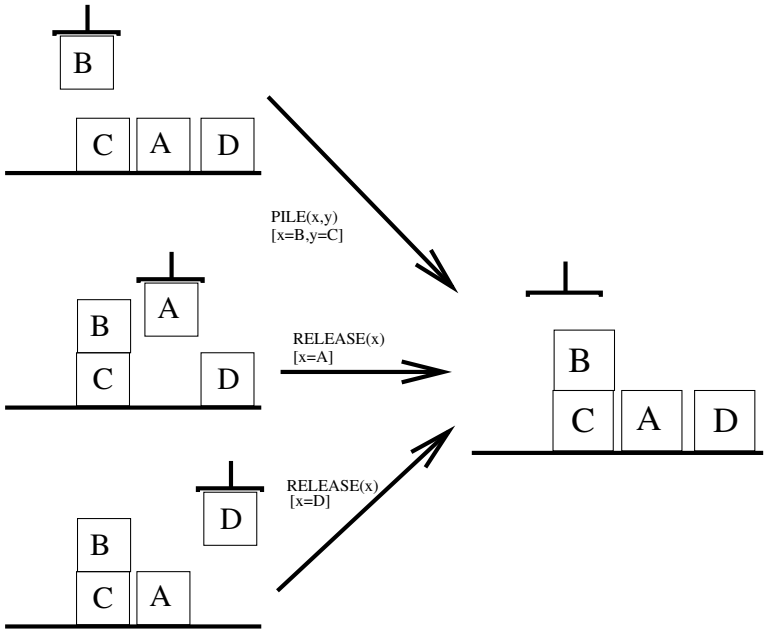
- Let G be a goal *without variables* (the case with variables will be addressed later).
  An action A is said to be *relevant* for G if:
  - At least one of the effects (positive or negative) of A appears in G with the same "sign"
  - None of the negative effects of A appears in G as positive
  - None of the positive effects of A appears in G as negative

Intuitive idea:

- An action is relevant if it could have been the last action applied from a plan leading to the goal

Relevant actions: example

- Goal: **`CLEAR(A)`**, **`CLEAR(B)`**, **`CLEAR(D)`**, **`ON(B,C)`**
  - Relevant actions: **`PILE(B,C)`**, **`RELEASE(A)`** and **`RELEASE(D)`**



  - Example of action **not relevant** for the same goal: **`PILE(A,D)`** (although has **`CLEAR(A)`** as positive effect, it also has as negative effect **`-CLEAR(D)`**)

Obtaining predecessors

- If A is an action without variables relevant for a goal G, then the *predecessor goal* of G wrt A is $(G - \mathit{effects}(A)) \cup \mathit{precond}(A)$

Example:

```
G = {CLEAR(A), CLEAR(B), CLEAR(D), ON(B,C)}

Predecessor with respect to the relevant action PILE(B,C)

G' = (G - {-CLEAR(C), -HOLD(B), FREEARM(),
           ON(B,C), CLEAR(B)})
       U {CLEAR(C), HOLD(B)}
   = {CLEAR(A), CLEAR(D), CLEAR(C), HOLD(B)}
```

- Note that the logic formalism allows to easily calculate predecessors

Backward search with heuristic (no variables)

```
FUNCTION BACKWARD-SEARCH-H(INITIAL-STATE,GOAL,ACTIONS)
  Return BS-H-REC({},{},INITIAL-STATE,GOAL,ACTIONS)

FUNCTION BS-H-REC(PLAN,VISITED,INITIAL-STATE,G-CURRENT,ACTIONS)
1. If INITIAL-STATE satisfies G-CURRENT, return PLAN
2. Let RELEVANT be the list of instantiated actions from ACTIONS,
    that are relevant for G-CURRENT
    and such that the predecessor of G-CURRENT wrt the action
    is a goal that do not contain any of the VISITED
3. Let ORDERED-RELEVANT be equal to HEURISTIC-SORT(RELEVANT)
4. For each ACT in ORDERED-RELEVANT
   4.1 Let G' the predecessor goal of G-CURRENT wrt ACT
   4.2 Let RES be equal to
        BS-H-REC(ACT·PLAN,VISITED U {G'},INITIAL-STATE,G',ACTIONS)
   4.4 If RES is not FAIL, return RES and halt
5. Return FAIL
```

- In this algorithm the function **HEURISTIC-SORT** remains to be defined
  - In general, it is defined by using an heuristic **H** over the states, that estimates the number of steps to reach the goal

Backward search with heuristic: properties

- The algorithm performs a search in a space where the states are the goals
  - Includes an heuristic ordering of goals

Properties:

- Sound, complete and always halts.
  Shortest solution is not guaranteed
- Theoretical complexity: same as forward search

- In practice, again due to high branching, its efficiency depends on the quality of the heuristic
  - We will comment further on heuristics later on
- Besides, in backward search we have a way of reducing branching, by minimum instantiation of actions
  - To this aim, we will use unification (similar to Prolog)

Section 6

Heuristics for planning

Heuristics for planning based on states space

- A foundamental component of the practical efficiency of the previous algorithms is the heuristic used for ordering states or goals
- This heuristic should estimate distance between states and goals (number of actions that we need to apply)
  - In forward search: for each state, distance to the goal
  - In backward search: for each goal, distance from the initial state
- If this estimation is below the actual minimum number of actions, we say that the heuristic is *admissible*
- Wanted: domain-independent heuristics
  - Based on the logic representation of states, goals and actions

Heuristics based on relaxing the problem

- Heuristics obtained by relaxing some of the restrictions of the problem and calculating the number of actions necessary for such relaxed problem
- Some ideas for relaxing a problem:
  - Ignore the negative preconditions and/or negative effects of the actions
  - Assume that each literal in a goal is achieved independently: estimate the number of actions necessary to achieve a goal as the *sum* of the number of steps necessary to achieve each of the literals of the goal
  - Assume that the number of actions necessary to achieve a goal is the *maximum* number of steps necessary to achieve *one* of its literals
  - Ignore all preconditions of the actions
  - Ignore certain predicates

The Δ_0 heuristic

- Let us define the Δ_0 heuristic based on the first and the second ideas just mentioned
- Given a state s, an atom p and a goal g that only has positive literals without variables, we define recursively $\Delta_0(s, p)$ and $\Delta_0(s, g)$ as follows:
  - If p appears in s, then $\Delta_0(s, p) = 0$
  - If p does not appear in s nor in the positive effects of any action, then $\Delta_0(s, p) = +\infty$
  - Otherwise, $\Delta_0(s, p) = min_A\{1 + \sum_{q \in precond^+(A)} \Delta_0(s, q) | p \in effects^+(A)\}$
  - $\Delta_0(s, g) = \sum_{p \in g} \Delta_0(s, p)$
- Notation: $precond^+(A)$ and $effects^+(A)$ denote the positive literals of the precondition and of the effects of an action A, respectively

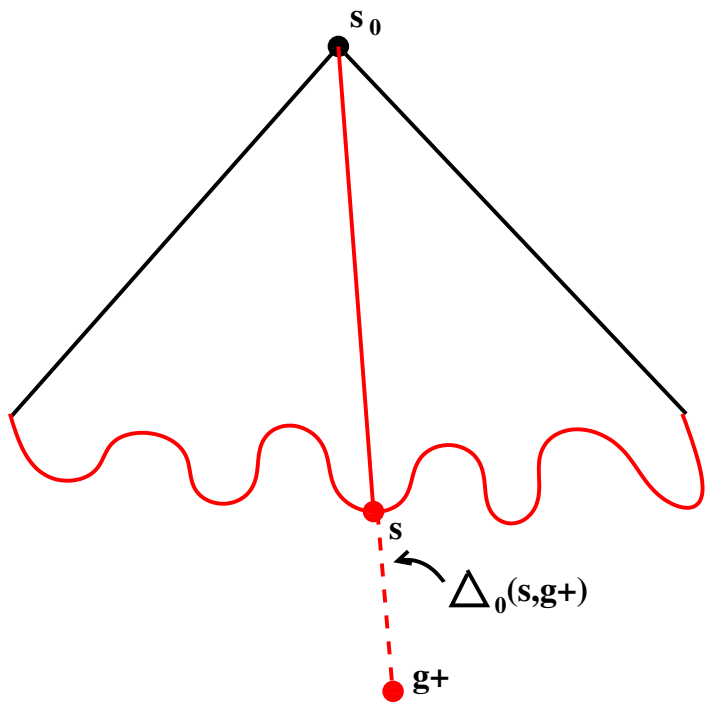
The Δ_0 heuristic (properties)

- Intuitively:
  - $\Delta_0(s, p)$ counts the least number of steps necessary starting from s in order to satisfy p, assuming that the actions do not have any negative preconditions nor negative effects
  - $\Delta_0(s, g)$ is the sum of such estimations for each atom $p \in g$
- Note that, in general, the estimation made by Δ_0 is not admissible
  - Anyway, it might perform quite well in practice
  - Besides, it is not intended to be used with A*, but with depth-first search instead
- Given a s, the values $\Delta_0(s, p)$ for each atom p can be calculated via an algorithm similar to Dijkstra's shortest path algorithm

Using Δ_0 heuristic in search (I)

In forward search:

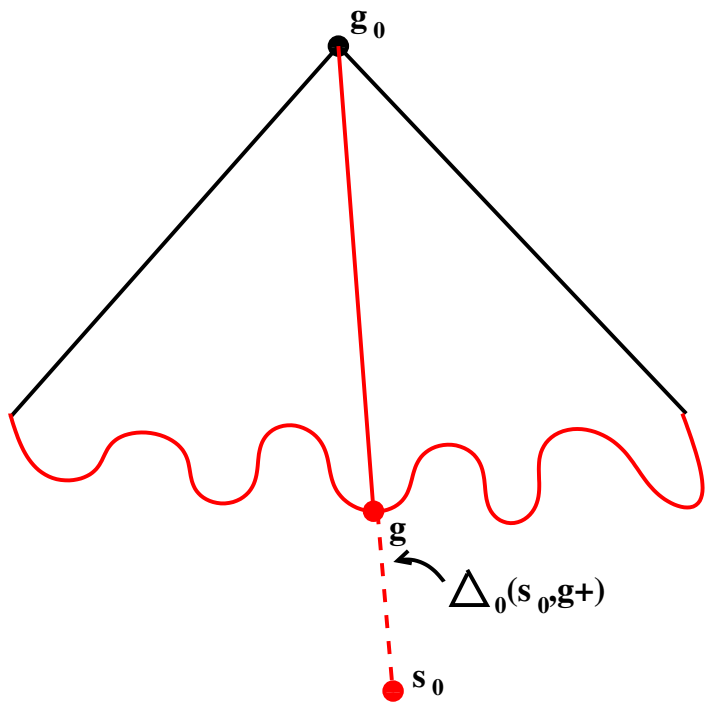
- To each action A applicable on the current state, we assign the heuristic value $\Delta_0(s, g+)$, where $g+$ is the set of the positive literals of the goal, and s is the state resulting after applying the action A



Using Δ_0 heuristic in search (II)

In backward search:

- To each action A being relevant wrt the current goal, we assign the heuristic value $\Delta_0(s_0, g+)$, where s_0 is the initial state, g is the predecessor corresponding to the action A, and $g+$ is the set of positive literals of g



Using Δ_0 heuristic in search (III)

In the case of goals with variables:

- If p is an atom with variables, $\Delta_0(s, p)$ is defined as the minimum of all $\Delta_0(s, \sigma(p))$, being $\sigma(p)$ instances without variables of p
- If g is a goal with variables, $\Delta_0(s, g)$ is the sum of $\Delta_0(s, p)$ for each positive literal p of g

Section 7

Partial Order Planning: POP

Partially ordered plans

- New approach for searching plans, different from the states-space-based search:
  - Instead of searching through a space of states or goals, the search is conducted in a space of non-fully-specified plans (partially ordered plans)
- *Partially ordered plan:* a plan where only part of the precedences between actions are specified

Example: how to pass IAING

- Language:
  - Objects: **IAING**, **ETSII**, **HOME**
  - Predicates: **AT(-)**, **STUDIED(-)**, **PASSED(-)**
- Initial state:

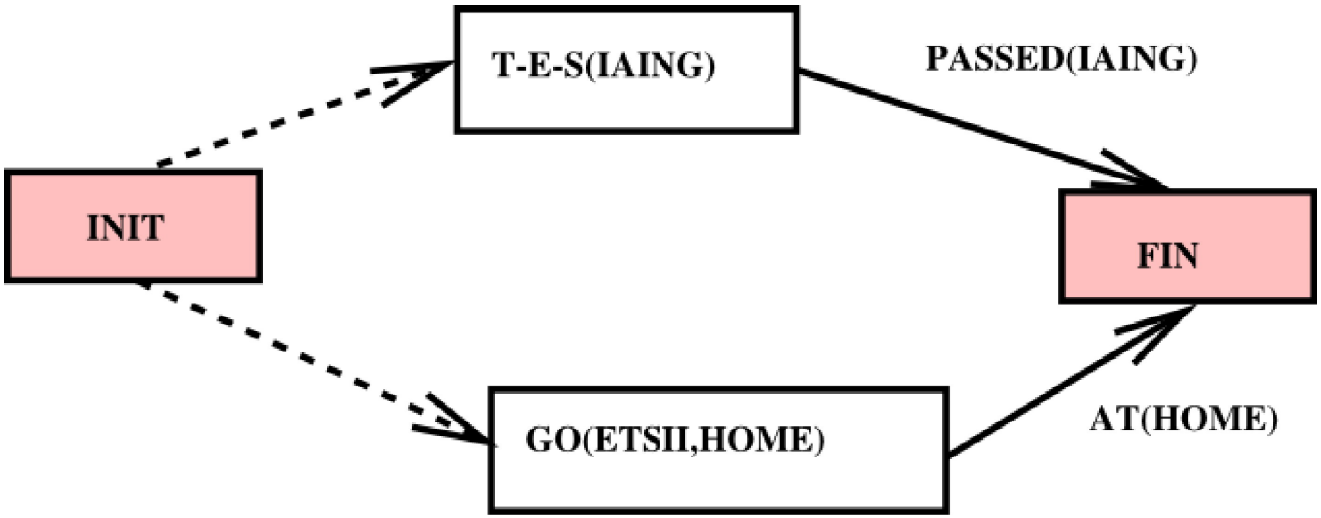
  AT(HOME)
- Goal:

  AT(HOME), PASSED(IAING)
- Actions:

```
GO(x,y)                STUDY(x)              TAKE-EXAM-SUCCESS(x)
P: AT(x)               P: {}                 P: AT(ETSII), STUDIED(x)
E: -AT(x),AT(y)        E: STUDIED(x)         E: PASSED(x)
```

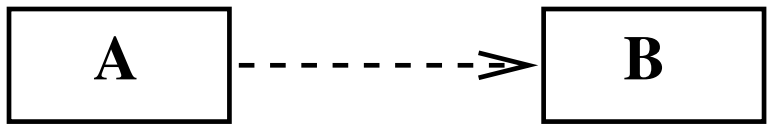
  - Actions of the type **GO(x,x)** will be discarded

Example of partial plan



Partially ordered plans: graph components (I)

- Nodes: *Actions* (from the actions of the problem, having their preconditions and effects), that constitute the steps that should be performed when executing the plan
  - Two special actions: INIT (no preconditions and its effect is the initial state) and FINISH (no effects and its precondition is the final goal)
- Arcs: *restrictions of order* $A \prec B$ between actions of the plan



(continues...)

Partially ordered plans: graph components (II)

- Another special type of arcs: *causal links*, $A \xrightarrow{p} B$, specifying that one of the preconditions of B (p) is achieved by applying action A (an ordering constraint is implicit)

- A set of *open preconditions*: those that do not have yet causal links that achieve them

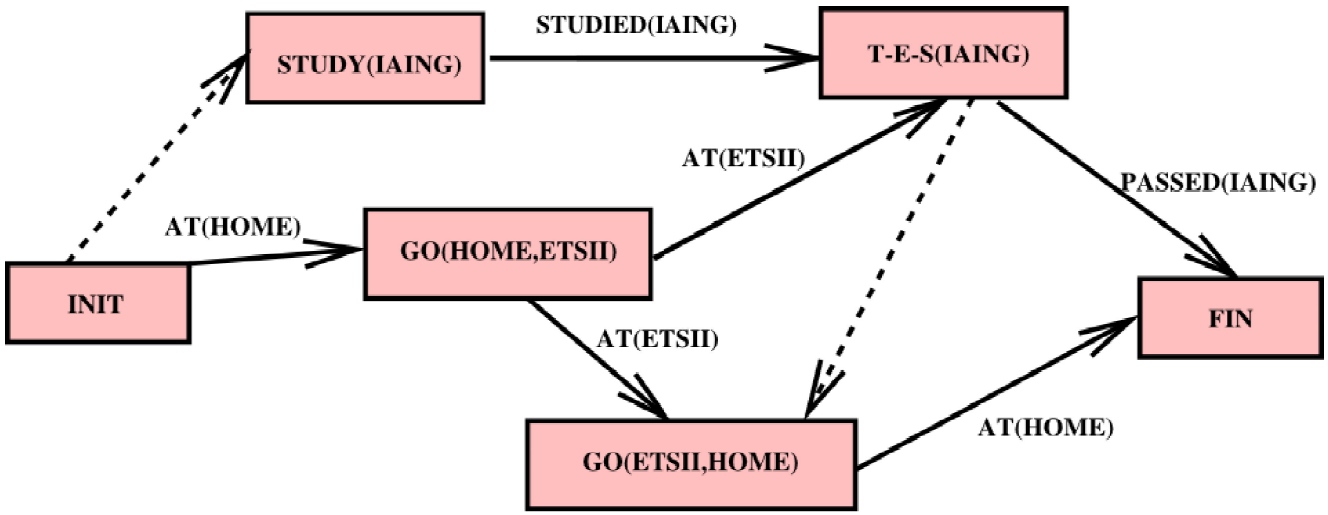
Initial and final partial plans

- Initial partial plan: plan whose only actions are INIT and FINISH, with the restriction INIT $\prec$ FINISH, no causal links, and all preconditions of FINISH being open
- Final partial plans: partial plans without conflicts between causal links, without cycles between ordering constraints and without open preconditions
  - An action C has a *conflict with* (or *threatens*) a causal link $A \xrightarrow{p} B$, if C has $\neg p$ among its effects and, according to ordering constraints, C *might be* applied after A and before B

Examples of partial plans

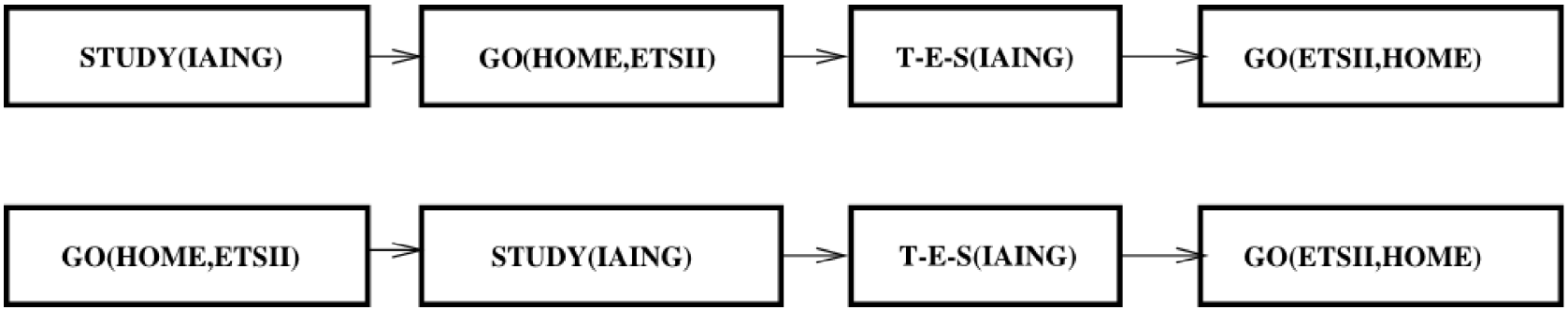
- Example of initial partial plan:

- Example of final partial plan:



Linearizations of final partial plans

- Linearization of a partial plan: line up all the actions in a row (sequence), without contradicting any of the ordering constraint deduced from the partial plan
- Key point: any linearization of a final partial plan constitutes a solution to the original problem
- In the example, two possibilities (both are solutions to the original problem):



Planification as search in the space of plans

- How to design an algorithm to find final partial plans?
  - Idea POP: start from the plan initial and apply transformations to the partial plans, *refining* them (fixing the flaws)
  - Basically, such transformations are either to achieve an open precondition or to neutralize a threat
- At each point there will be several refinement options, and not all of them lead to a final partial plan
- Problem: find the sequence of refinement that starting from the initial partial plan, yields a final partial plan (that is, with no cycles, no conflicts and no open preconditions)
  - It's again a states space search!
- But now *the states are partial plans* and *actions are refinement steps*

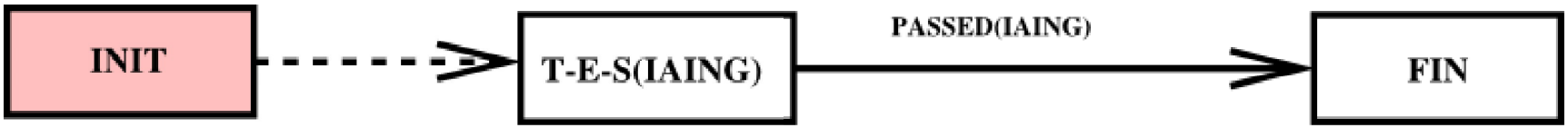
POP Algorithm for the example of how to pass IA

- For the sake of a clear graphical representation, we will agree on the following conventions:
  - Some ordering constraints will not be drawn, in particular those associated with causal links
  - Preconditions and effects will not be displayed. In particular, open preconditions will not be drawn
  - When an action does not have any open precondition, it will appear as coloured box
- Step 1: Initial plan

  - Open preconditions: **PASSED(IAING)**, **AT(HOME)**; let us achieve first **PASSED(IAING)**
  - Only one possibility: use a new action **TAKE-EXAM-SUCCESS(IAING)**

POP Algorithm for the example of how to pass IA

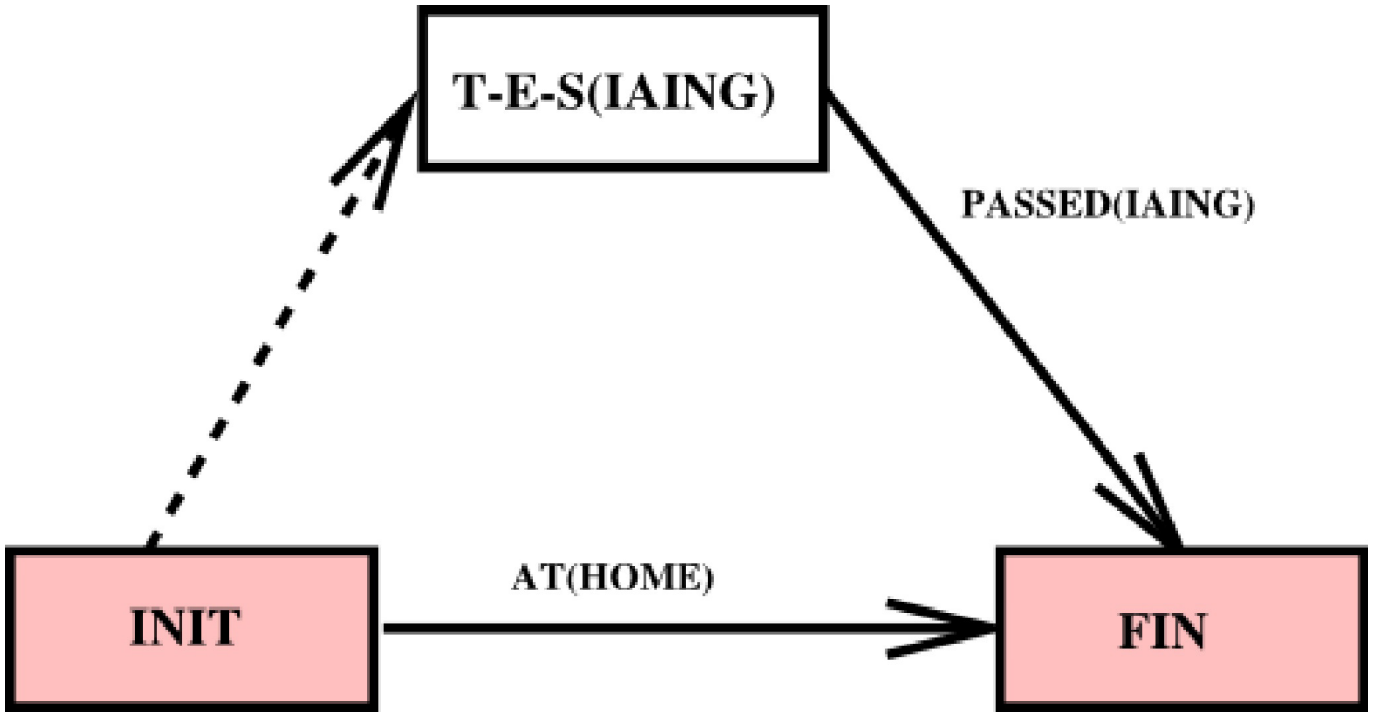
- Step 2:



  - We achieve now the precondition **`AT(HOME)`** from **`FINISH`**
  - Two possibilities: **`INIT`** or new action **`GO(ETSII,HOME)`**
  - Consider first **`INIT`**, and try the second one in case of Fail
  - When a precondition is achieved by adding a causal link from an action already in the plan, this refinement is called *simple establishment*
  - If we need to include a new action in the plan, this refinement is called *step addition*

POP Algorithm for the example of how to pass IA

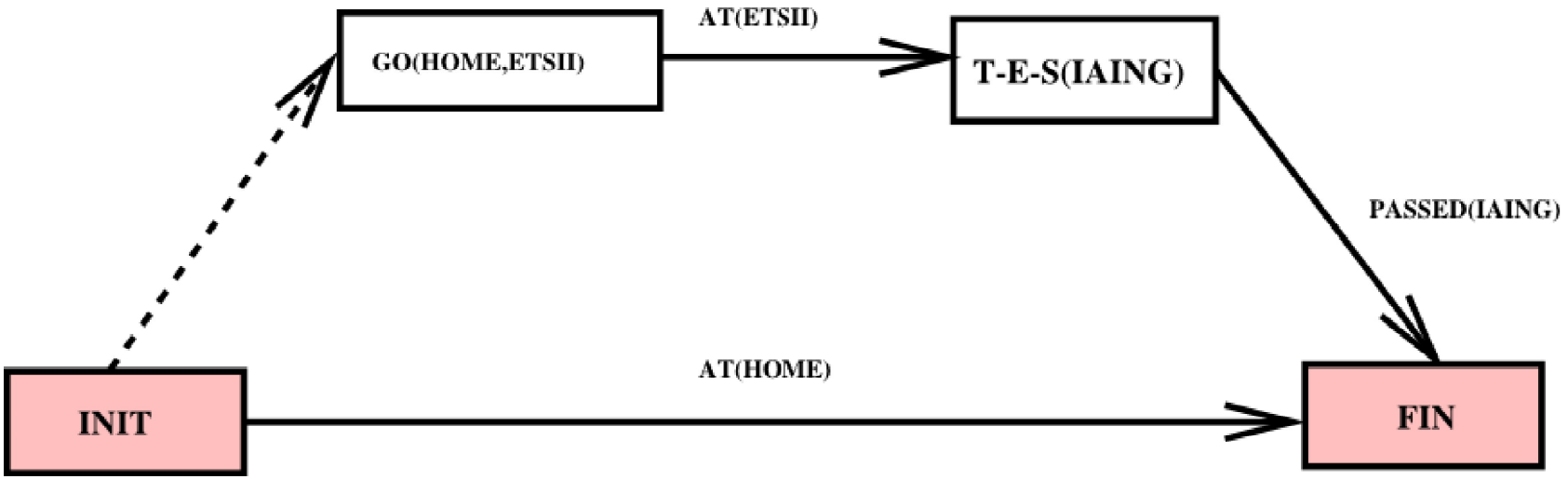
- Step 3:



  - We achieve now the precondition **`AT(ETSII)`** from **`TAKE-EXAM-SUCCESS(IAING)`**
  - Only one possibility: use a new action GO(HOME,ETSII)

POP Algorithm for the example of how to pass IA

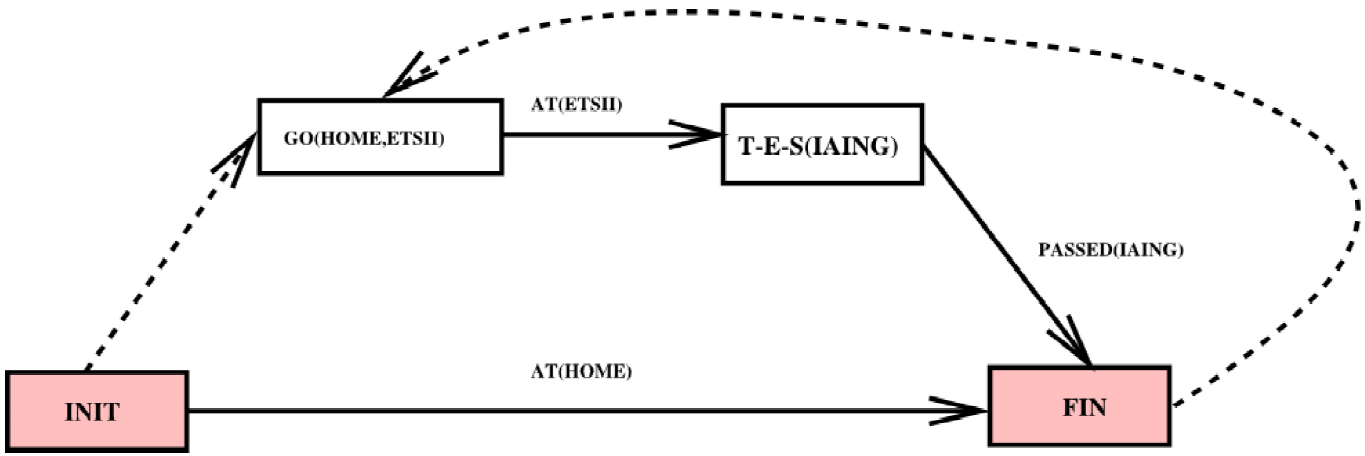
- Step 4:



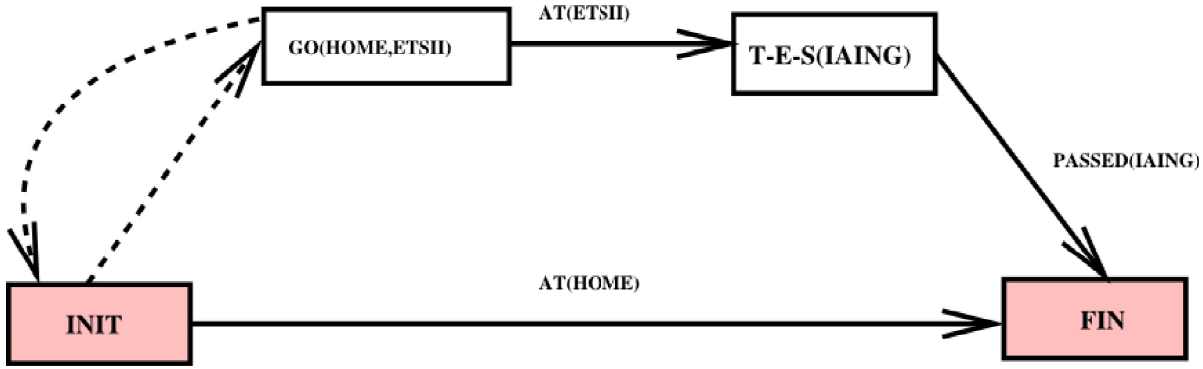
- A threat shows up: from GO(HOME,ETSII) into the causal link between INIT and FINISH
- We try to fix the threat
  - forcing GO(HOME,ETSII) to be executed after FINISH (*promotion*), or else
  - forcing GO(HOME,ETSII) to be executed before INIT (*demotion*)

POP Algorithm for the example of how to pass IA

- But in both cases, a cycle would be created:
  - Promotion:

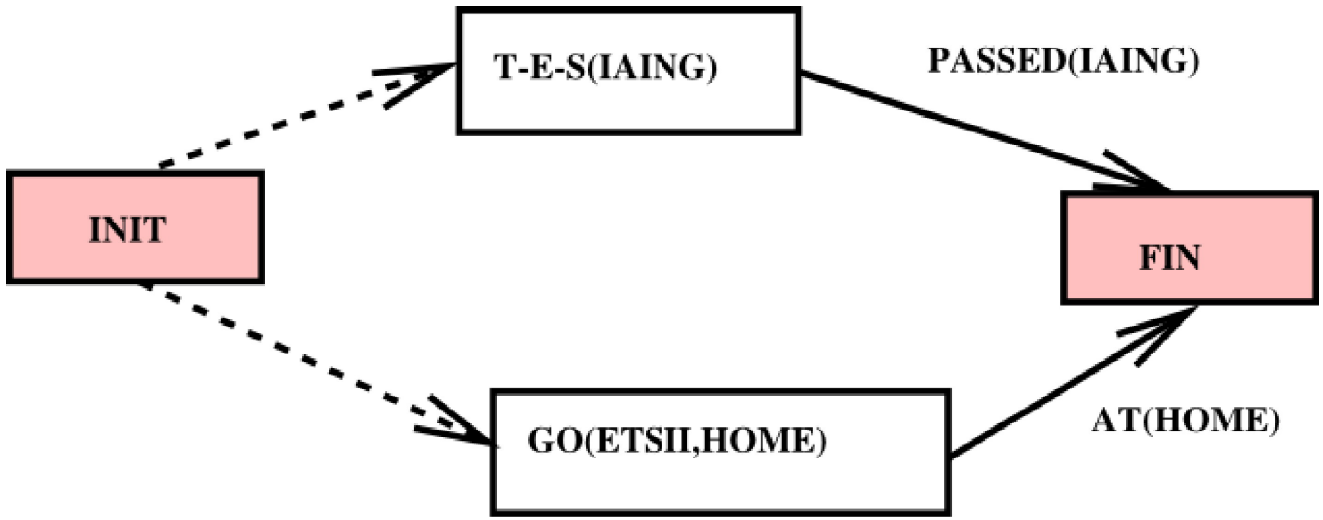


  - Demotion:



POP Algorithm for the example of how to pass IA

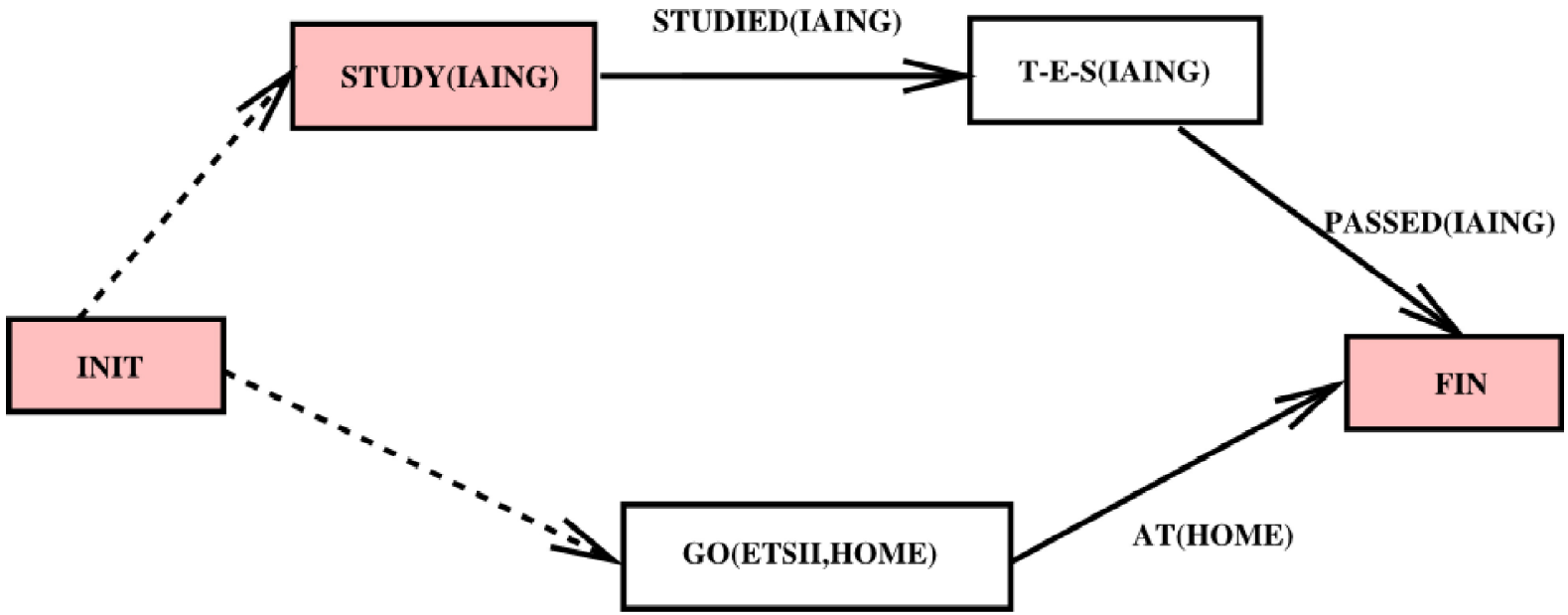
- Therefore, we get Fail and we must go back to the last branching point, which was in step 2
  - And choose the other alternative: new action GO(ETSII,HOME) to achieve the precondition **`AT(HOME)`** in FINISH
- Step 5:



- We select now the precondition **`STUDIED(IAING)`** in **`TAKE-EXAM-SUCCESS(IAING)`**
  - The only possibility to achieve it is a new action **`STUDY(IAING)`**

POP Algorithm for the example of how to pass IA

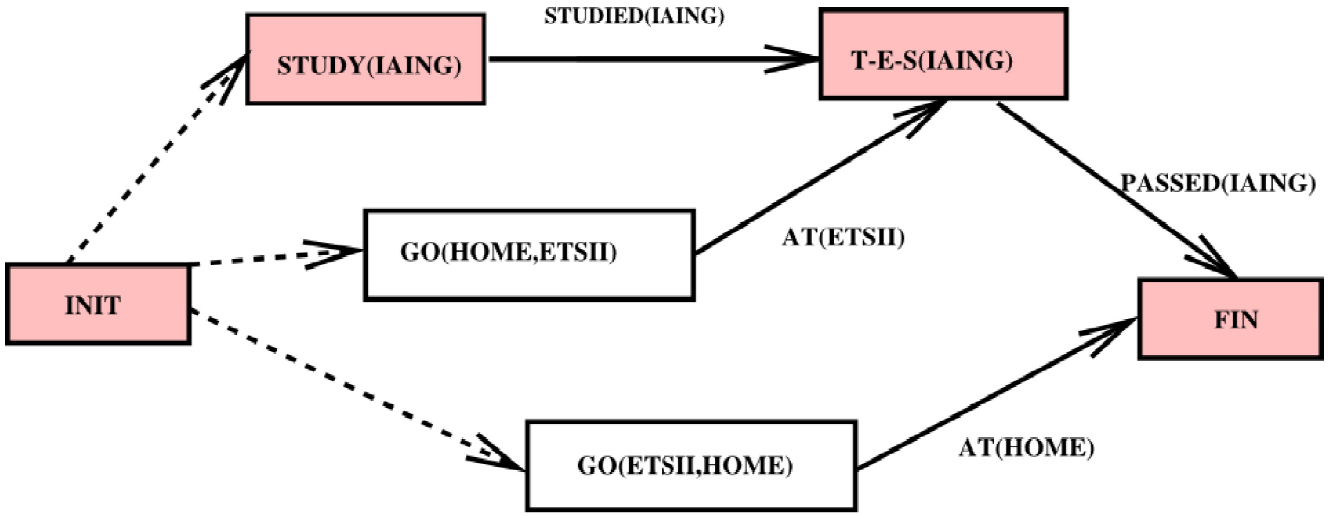
- Step 6:



- We select now the precondition **`AT(ETSII)`** in **`TAKE-EXAM-SUCCESS(IAING)`**
  - The only possibility to achieve it is a new action **`GO(HOME,ETSII)`**

POP Algorithm for the example of how to pass IA

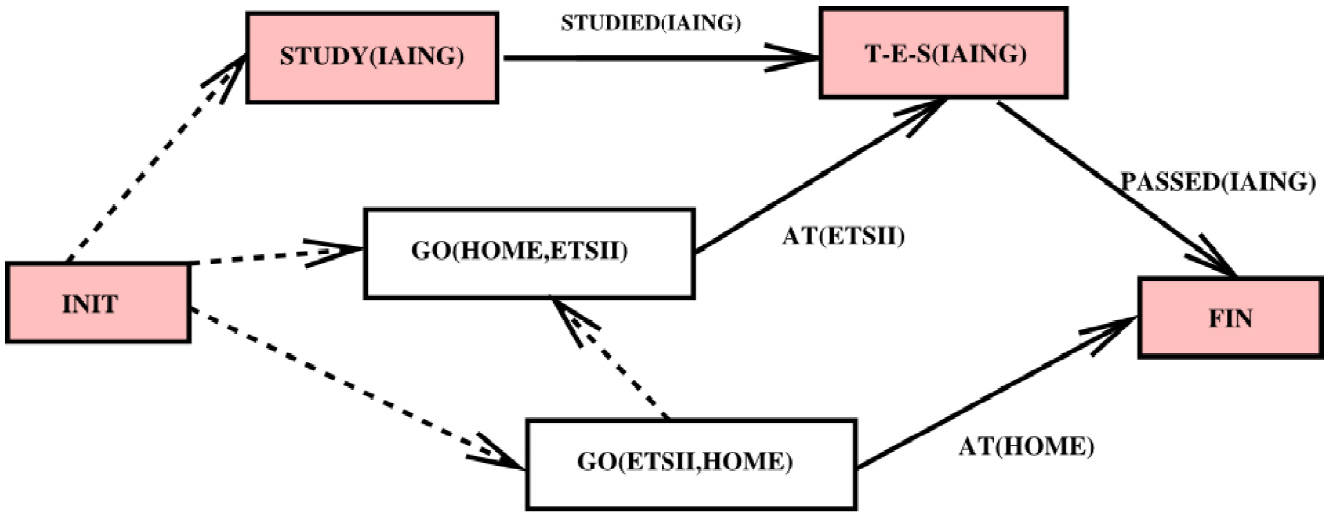
- Step 7:



  - At this point there are threats in the plan: for example, from GO(ETSII,HOME) into the causal link between GO(HOME,ETSII) and TAKE-EXAM-SUCCESS(IAING)
  - We try to fix the threat using demotion; if this doesn't work, then we try promotion

POP Algorithm for the example of how to pass IA

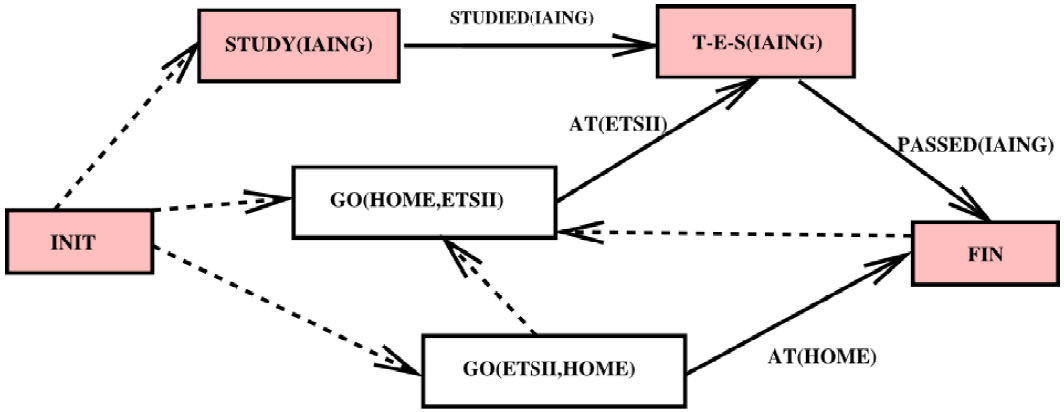
- Step 8 (demotion):



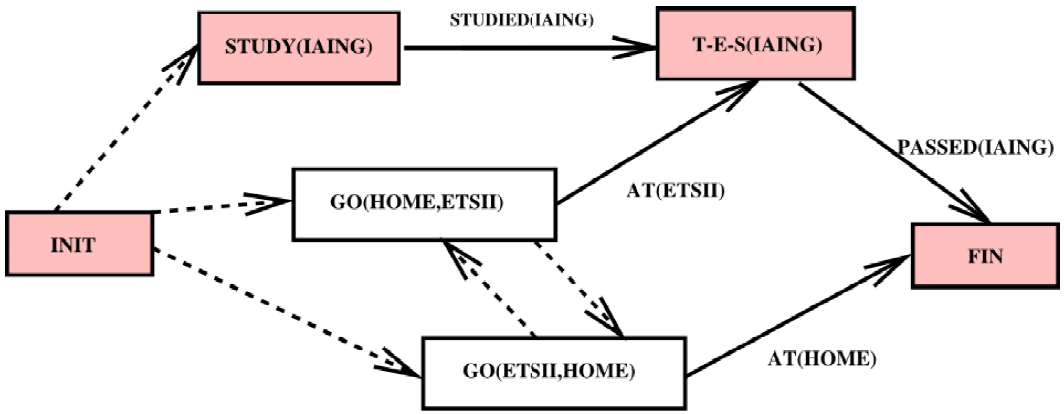
  - New threat: from GO(HOME,ETSII) into the causal link between GO(ETSII,HOME) and FINISH
  - We try to fix the threat by promotion or demotion

POP Algorithm for the example of how to pass IA

- But in both cases, a cycle would be created:
  - Promotion:

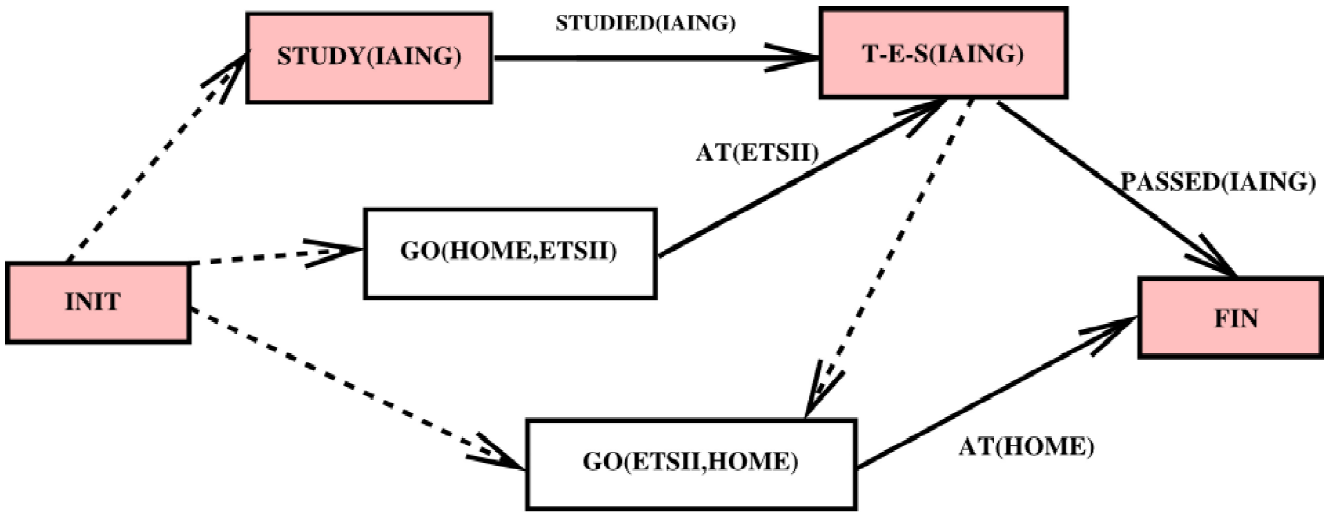


  - Demotion:



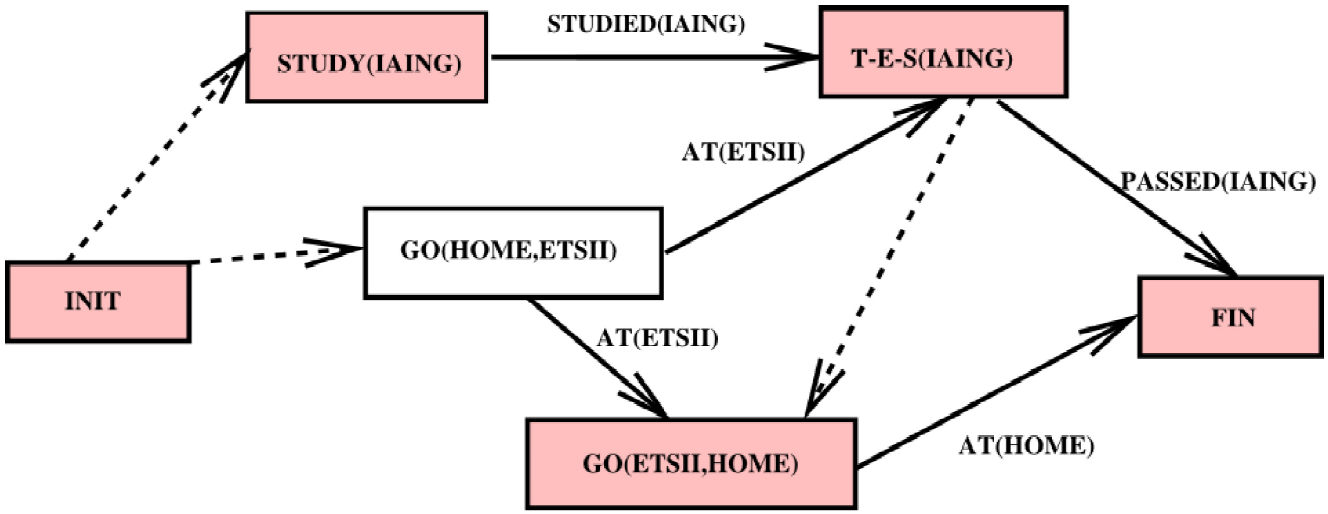
POP Algorithm for the example of how to pass IA

- Therefore, we get Fail
  - We must go back to the last branching point (step 7), and pick the other alternative: fix the threat from GO(ETSII,HOME) into the causal link between GO(HOME,ETSII) and TAKE-EXAM-SUCCESS(IAING), by promotion
- Step 9:



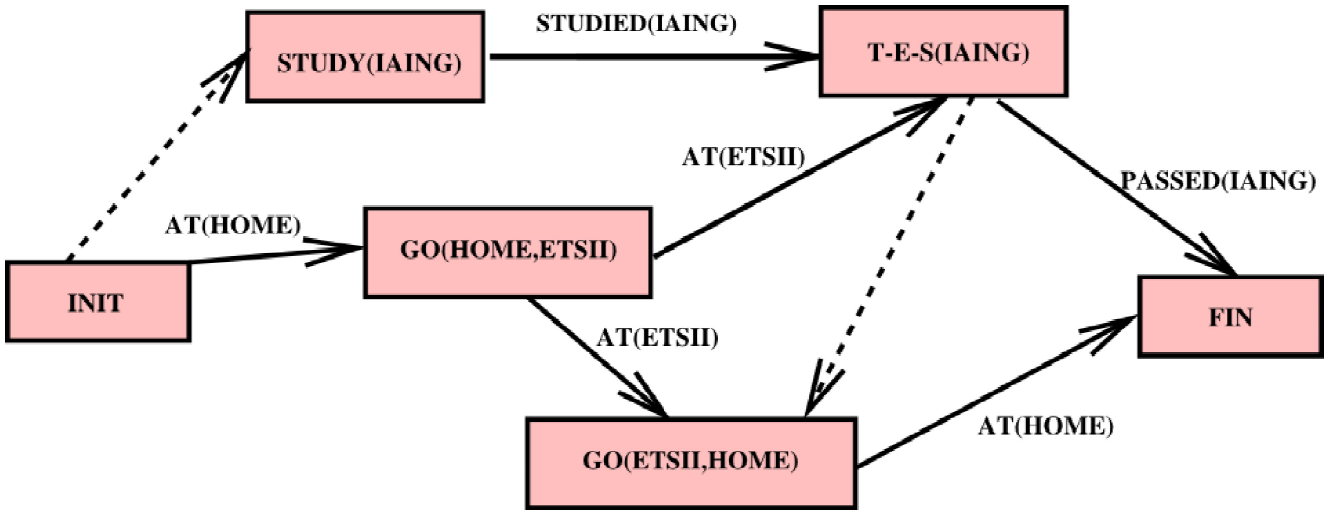
POP Algorithm for the example of how to pass IA

- We achieve now the precondition **`AT(ETSII)`** from **`GO(ETSII,HOME)`**
  - Two possibilities: simple establishment with **`GO(HOME,ETSII)`** or step addition with **`GO(HOME,ETSII)`**.
  - We select the first option, but will reconsider if necessary
- Step 10:



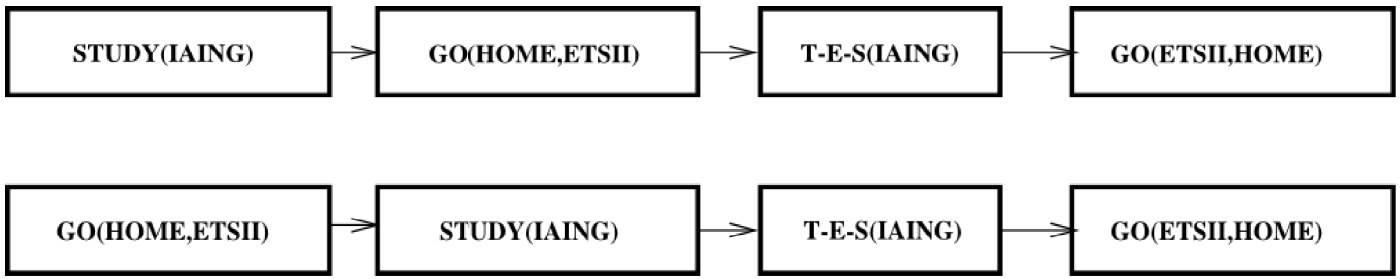
POP Algorithm for the example of how to pass IA

- We achieve now the precondition **AT(HOME)** from **GO(HOME,ETSII)**
  - Three possibilities: simple establishment with **INIT**, simple establishment with **GO(ETSII,HOME)** or step addition with **GO(ETSII,HOME)**.
  - We select the first option, but will reconsider if necessary
- Step 11 (final partial plan):



POP Algorithm for the example of how to pass IA

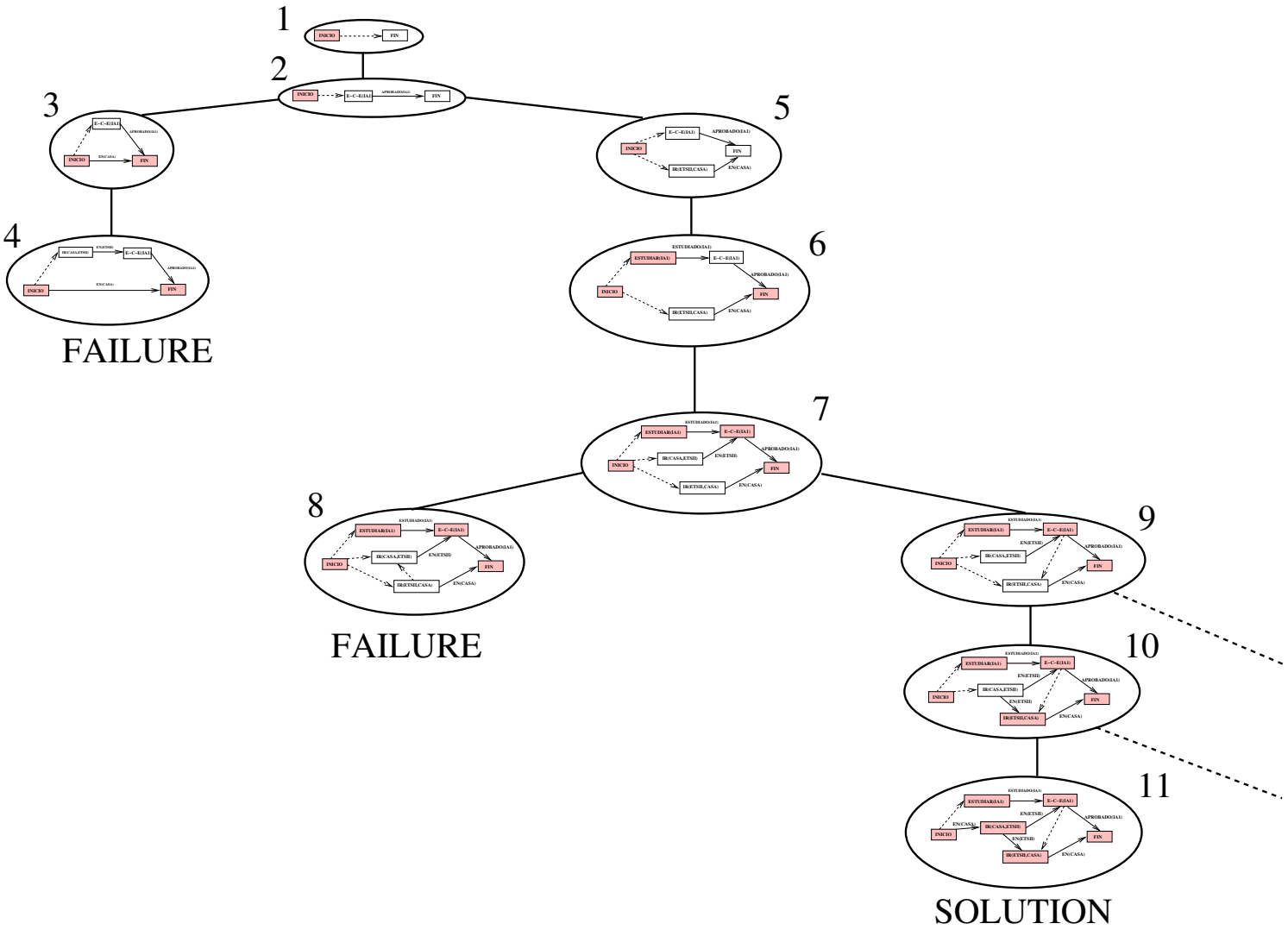
- Finally, we perform linearization (two possibilities):



Partial order planning: search

- As seen in the previous example, it is possible to implement a planning algorithm as a search
  - More precisely, as a search of a sequence of refinement transformations that starting from the initial plan obtains a final partial plan
- A POP algorithm is just an algorithm that implements a search algorithm (e.g. depth-first) in the space of partial plans

POP Search tree for the example of how to pass IA



Transformations (refinements of partial plans)

- Achieving open preconditions: given an open precondition p from an action B of the plan, for each action A having p as effect, we can obtain a successor (*refined*) plan by applying some of the following steps (provided that the resulting plan has no cycles):
  - *simple establishment*: if action A is already in the plan, add the restriction $A \prec B$ and the causal link $A \xrightarrow{p} B$
  - *Step addition*: add the new action A to the plan, the restrictions $A \prec B$, INIT $\prec A$ and $A \prec$ FINISH, and the causal link $A \xrightarrow{p} B$ (it is even possible to add as new an action equal to an existing one, assuming that it represents a different *step* in the final linearized solution)

Transformations (refinements of partial plans)

- Fixing conflicts: given a threat from the action C into the causal link $A \xrightarrow{p} B$, we can fix it and obtain a new successor plan by choosing one of the following options (provided that the new plan does not have any cycle):
  - Add the constraint $B \prec C$ (*promotion*)
  - Or add the constraint $C \prec A$ (*demotion*)

POP as a search process

- Search tree:
  - Nodes: partial plans
  - Branching due to election among actions that achieve open preconditions
  - Branching due to election among the two ways of fixing threats (promotion and demotion)
- Note: it is not necessary to consider branching due to the election of the next open precondition to achieve, neither for the election of which one of the current threats to fix. They will be dealed with in future steps, following a given order
  - Although such an order does make a difference wrt the efficiency of the search
- Depth-first search might get into an infinite branch
  - Therefore, we perform a bounded search (having a maximum number of actions in the plan)

Algorithm POP (recursive version)

```
FUNCTION POP(INITIAL-STATE,GOAL,ACTIONS,BOUND)
1. Return POP-REC(INITIAL-PLAN(INITIAL-STATE,GOAL),ACTIONS,BOUND)

FUNCTION POP-REC(PLAN,ACTIONS,BOUND)
1. If PLAN is final, return LINEARIZATION(PLAN) and halt.
2. If there exists a threat in PLAN from action C into A --p-->B,
   then
     2.1 Let PLANPR obtained from PLAN by promotion
     2.2 If PLANPR has any cycle, let RESULT equal to FAIL;
         otherwise:
            let RESULT equal to POP-REC(PLANPR,ACTIONS,BOUND).
     2.3 If RESULT is not FAIL, return RESULT and halt;
         otherwise:
         2.3.1 Let PLANDEG obtained from PLAN by demotion
         2.3.2 If PLANDEG has any cycle, let RESULT equal to FAIL;
               otherwise:
                  let RESULT equal to POP-REC(PLANDEG,ACTIONS,BOUND).
         2.3.3 Return RESULT and halt.
```

(continues...)

Algorithm POP (recursive version, cont.)

```
3. Let P an open precondition in PLAN
4. For each action A in ACTIONS (either new or already in PLAN)
    having P as an effect, do:
   4.1 Let PLANEXT the result of achieving the precondition P
        in PLAN by a causal link from A
       (in the case of a new action, add the action to the PLAN)
   4.2 If PLANEXT has no cycles and its number of actions is
        lower or equal to BOUND, then:
       4.2.1 Let RESULT equal to POP-REC(PLANEXT,ACTIONS,BOUND)
       4.2.2 If RESULT is not FAIL, return RESULT and halt.
5. Return FAIL
```

POP: some considerations

- We have presented a simplified version of the algorithm.
  To be added:
  - Use of unification
  - Some variables might remain uninstantiated (principle of minimum-commitment)
  - Inequality restrictions
- The algorithm is sound and complete (provided that the bound is large enough)
- The presented pseudocode deals first with conflicts and then with open preconditions, but this is not mandatory
  - It is also not necessary to try always promotion and then demotion

POP: use of heuristics

- It is hard to estimate how "close" a partial plan is from a solution
- There are some elections in the algorithm that, if taken correctly, might produce an efficiency improvement:
  - At each step, which threat or which open precondition is refined?
  - When dealing with an open precondition, in which order do we try the actions that achieve it?
  - When fixing a conflict, in which order do we try promotion and demotion?

POP: some heuristics

Heuristic to select the next refinement step

- Select the threat or open precondition that has the *least* number of alternatives to fix it
  - Recall that threats always have two refinement alternatives
  - For open preconditions, as many alternatives as actions (new or existing) have it in their effects

Heuristics for the order of actions that achieve open precondition

- Sort them by the size of their preconditions lists (not very good)
- Use the *planning graph*

Other approaches for planning

- SATPLAN, use of techniques based on propositional logic
- GRAPHPLAN
- Planning as PSR
- Planning with limited resources
- Planning with explicit time
- Planning by hierarchical decomposition
- Planning on environments with uncertainty
- Execution of the plan: watching and replanning
- Continuous planning
- Multiagent planning

Bibliography

- Russell, S. and Norvig, P. *Artificial Intelligence (A Modern Approach) 3rd edition* (Prentice–Hall, 2010).
  - Ch. 3 "Solving problemas by searching".
  - Ch. 10: "Planning".
- Ghallab M., Nau D. and Traverso P. *Automated Planning: theory and practice* (Elsevier, 2004)
  - Sec. 2.3 "Classical representation".
  - Ch. 4 "State space planning".
  - Ch. 5 "Plan space planning".
  - Ch. 9 "Heuristics in planning".