AI Planning
Lecture notes

Issa Hanou

These lecture notes are inspired by the lectures on AI Planning
by Christian Muise (Queen's University, Kingston, Canada).

September 24, 2024

1 What is Planning?

Planning is the art and practice of thinking before acting.
–Patrik Haslum

Planning is a type of problem-solving. In planning, we answer the question of *how I get from where I am now to where I want to be?* by creating a plan to go from some starting point to a goal, where the plan can be represented as a sequence of actions. There are tons of examples of planning in the real world, like planning your route using Google Maps. Other, more complicated problems can also be solved using planning techniques, for example, mega-printer operations [3] or greenhouse optimization [1].

Scheduling problems, such as timetabling and personnel scheduling, are also very related to planning. The main difference is that scheduling solvers arrange known tasks over time, whereas planning solvers also have to figure out what the exact tasks are.

1.1 Types of planning

There are different types of planning, which each take into account a different set of aspects from the following list:

- **Determinism**: do we treat the problem as deterministic, non-deterministic, or probabilistic?
- **Observability**: do we assume full knowledge of the world, or is there only partial observability?
- **Multi-agent**: is there just one or multiple agents? In the second case, how do they interact, do they collaborate or are they adversaries; do we control all agents or is planning distributed?

- **Objective**: do we want any feasible plan? Do we want to minimize costs or minimize the time? Are there other preferences?

These aspects can create a range of different problem settings, which we have listed here for an overview. In this lecture, we focus on deterministic, fully observable, single-agent problems where we want to find a satisfying plan to introduce the concept of planning. However, other lectures in this course will talk about some other forms, especially the probabilistic types.

- **FOD** Fully Observable Deterministic
- **FOND** Fully Observable Non-Deterministic
- **POD** Partial Observable Deterministic
- **POND** Partial Observable Non-Deterministic
- **MDP** Markov Decision Processes
- **POMDP** Partial Observable Markov Decision Processes

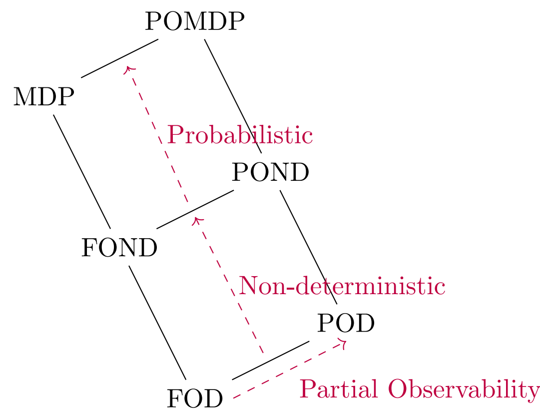


Figure 1: Comparing different types of planning.

1.2 Solving a planning problem

Planning generally has three stages. First, you create a model of your problem. Second, you solve the model. Finally, you execute the solution. All three steps are directions of research in the planning community, and they are not fully independent. The model you create, and all the fancy features that you use in your model, will determine which solvers you can use to find solutions. The problem-solving method that you use, in turn, might influence how fast you can execute and how robust you are to external changes. Moreover, consequences

from execution might influence how realistic your model is and could result in model changes.

In this lecture, we focus on the modeling of problems and show how to find solutions using planning solvers. We will look at several traditional examples that are well-known in the planning community, but all methods that we consider are not tailored to these specific domains. In other words, we consider domain-independent planning, which is the most commonly studied type. Many domains have been written in the past using the same modeling language, and many planners have been developed that solve any problem written in the same language.

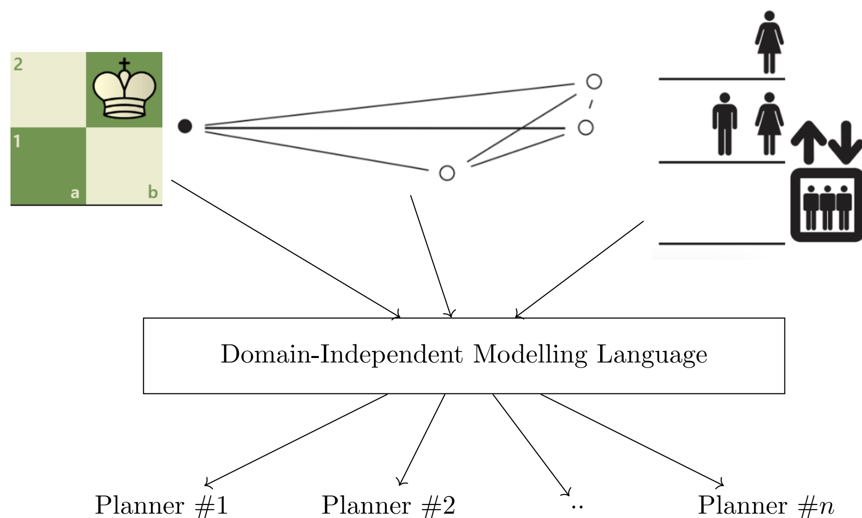


Figure 2: Domain-Independent Planning.

2 Searching for solutions

In planning, we handle agents who think before they act so in other words, they look ahead. Then, finding a plan that brings you to your goal becomes a search problem. There are many ways to search for a solution, where the basic methods perform an uninformed search, like depth-first search, breadth-first search, or uniform-cost search (see subsection 2.3 for more details). However, these uninformed searches will often select the first action that comes to mind, which may not always be the best action to take.

In planning, agents generally think about what happens when they take a certain action. So, they consider what the world would be like after taking an action without having to try it out, like a simulation. However, to know the effects of an action, they need to have a model of the world. This model formally defines the search problem in finding a plan.

2.1 Search problem

A search problem consists of a state space, a successor function, a start, and a goal test. The solution to a search problem is a sequence of actions (a plan) that transforms the start state into a goal state. As we do not define the goal state, only a test referring to a (partial) description of the goal, there are often multiple goal states in any search problem. Consider, for example, the PacMan game (see Figure 4). If the goal is to eat all the dots, this could be done in different ways, and the goal state may have PacMan at different places, depending on the last dot it ate. Our search problem defines a model of the world. Usually, the model does not capture the whole world but only the specific part of the world that we are interested in for our problem.

For example, consider the problem of traveling by train in the Netherlands. Figure 3 shows a simplified illustration of the railway network, with the time it takes to travel between cities on the edge. The state space consists of the train stations in the Netherlands, and the successor function defines which stations can be reached from other stations. If our start state is Delft and the goal test is whether the state we are in is Utrecht, then a solution could use different routes through the network.

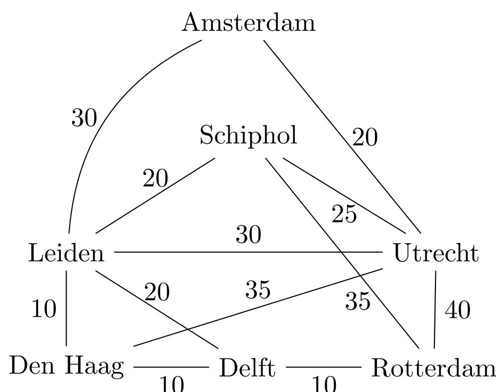


Figure 3: Example problem of finding a route through (partial) train network in the Netherlands.

2.2 State spaces

The search traverses a state space, which is related to the problem state space but stored differently. A *world state* includes every detail in the environment, but a *search state* keeps only the details needed for planning, which is an abstraction of the world state. The biggest reason for the abstraction of a world state is to maintain the size of a state space.

Consider the example in Figure 4. The world state has four parameters: 120 possible agent positions, a maximum food count of 30, 12 possible ghost

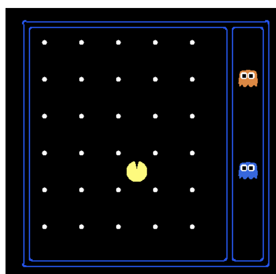
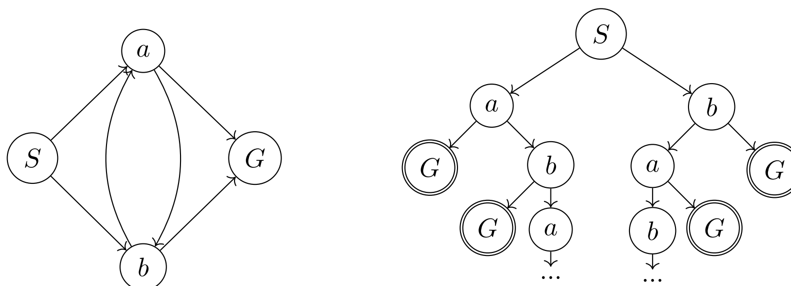


Figure 4: PacMan problem example with a 12×10 grid.

positions, and four possible directions the agent is facing. This results in $120 \times 2^{30} \times 12^2 \times 4$ possible world states. Now, consider two possible problems in this world; the first is for the agent to reach a certain location, then there are only 120 *relevant search states*. In a second problem, the agent has to eat all dots, and there are 120×2^{30} possible search states.

All the states in the state space can form a graph, which is a mathematical representation of the search problem. Different states are nodes, and the successors determine the edges. A *state space graph* does not contain any duplicates, but some states can be reached through different paths. However, a state space graph is rarely built completely due to memory limitations.

Instead, solvers often keep a search tree, with the start state at the root, and each node has its possible successors as children (which represent the what-if scenarios). Although each node represents a state, the path that you follow from the root to this node corresponds to a plan to achieve that state. In the search tree, states may be duplicated as different plans now correspond to different nodes. However, even with search trees, we can seldom build the whole tree as it grows exponentially large. Important to note is that each node in the search tree is an entire path in the state space graph. Figure 5 shows the difference between a state space graph and a search tree.



(a) Example problem state space graph. (b) Example problem (partial) search tree.

Figure 5: State space graphs versus search trees.

The search tree is used to find a solution, where we start at the starting

state (root node) and expand nodes to unfold potential plans. Here, we try to expand as few nodes as possible while trying to reach a plan that finds the goal. Algorithm 1 shows the pseudocode for a tree search, where the most important notion is the search frontier (the bottom level of explored nodes) and what node from the frontier to expand next. Figure 6 shows an example of how to progress through a search tree with an arbitrary search strategy, which only returns when the goal is removed from the open list.

Algorithm 1 Pesudocode for tree search.

```
function TREE-SEARCH(problem, strategy)
    Initialize the search tree using the initial state of problem
    loop
        if there are no candidates for expansion then return failure
        end if
        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 if
    end loop
end function
```

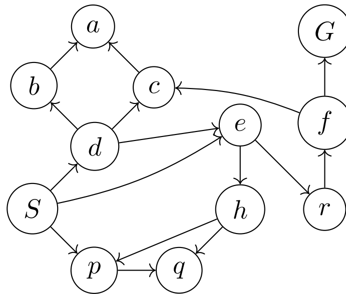
2.3 Search algorithms *(not in lecture)*

In this section, we give some more information on different search algorithms, which have been previously discussed in earlier lectures as well. However, other search algorithms also exist. First, we cover a set of basic search types using uninformed search: they do not have any knowledge of the goal.

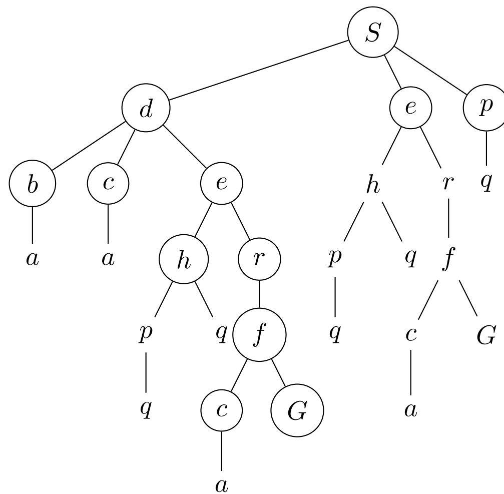
2.3.1 Uninformed search

Depth-First Search (DFS) Depth-first search is a basic search algorithm where the search tree is traversed by always expanding the deepest node in the search frontier first. In this case, the search frontier can be implemented with a Last-In-First-Out stack. When the goal is reached, the search stops, so the whole tree is not necessarily searched, which may result in sub-optimal plans. When using a branching factor b, and a search tree depth of m, the search time can go up to $O(b^m)$, but only if m is finite (if there are cycles it can be infinite). However, the space required to store the search frontier is always limited by $O(bm)$, as only the siblings on the path to the root are stored.

Breadth-First Search (BFS) Contrary to depth-first search, breadth-first search expands the shallowest node first, so the search frontier forms a First-In-First-Out queue. In this case, the time to search is expressed by the branch-



(a) Problem: find path from S to G.



(b) Search tree

S
$S \to d$
$S \to e$
$S \to p$
$S \to d \to b$
$S \to d \to c$
$S \to d \to e$
$S \to d \to e \to h$
$S \to d \to e \to r$
$S \to d \to e \to r \to f$
$S \to d \to e \to r \to f \to c$
$S \to d \to e \to r \to f \to G$

(c) Search paths explored.

Figure 6: Tree-Search example.

ing factor b and the depth of the shallowest solution as $O(b^s)$, which can still be $O(b^m)$ in the worst-case scenario. However, the algorithm needs the same amount of space $O(b^s)$. If all the edge costs are 1, it is possible to find the optimal solutions using this algorithm.

Iterative Deepening This search type is a combination of the last two. It starts by running a DFS search with depth limit 1. Then, if no solution is found, it runs a DFS search with depth 2, and so on. The advantage is that only the memory storage of DFS is used, and we avoid doing the most work in the lowest level of the search tree (if a shallow solution exists).

Uniform-Cost Search This search type is similar to BFS, but now we use edge costs. The tree expands the next cheapest nodes while using a priority

queue for the search frontier with the cumulative cost to get to a certain node. Now, only nodes that have a lower cost than the cheapest solution are expanded. The optimal solution cost is C^*, where each edge costs at least ϵ, so the effective depth to the solution is $\frac{C^*}{\epsilon}$. The search is exponential in the effective depth, so it takes $O(b^{\frac{C^*}{\epsilon}})$. Uniform-Cost Search is a special case of the Dijkstra algorithm, where nodes only get added to the queue if they are needed.

2.3.2 Informed Search

Informed search assumes knowledge of a goal state, and often uses heuristics to estimate how close a current state is to a goal state. Most heuristics are designed for a particular search problem, but there are also more general heuristics.

Greedy search Greedy search is the simplest form of informed search. The strategy says: expand the node that you think is closest to a goal state. This is a form of best-first search, but it heavily depends on the quality of the heuristic. In case of a bad heuristic, the search acts like a badly-guided BFS. However, with a good heuristic, good solutions can be found rather quickly.

A^* **Search** A^* search is a well-known algorithm used a lot in Artificial Intelligence. It is a combination of uniform-cost search and greedy search. The uniform-cost search orders the search frontier by path cost (how much did it take to get to where the current node is), which is also called the *forward cost* or $g(n)$. On the other hand, greedy search orders the nodes by goal proximity (according to some heuristic) and estimates the cost $h(n)$ of the cheapest path from node n to the goal, sometimes called the *backward cost*. The A^* search frontier orders by the sum of $f(n) = g(n) + h(n)$. The A^* search does not yet stop when it encounters a goal, but only when a goal node is *dequeued* from the search frontier, this ensures optimality of the algorithm, but only when the heuristic is good. To have an optimal search algorithm, the heuristic must be *admissible* so that it never overestimates the cost-to-go to the goal. Formally, this is defined such that the value of an admissible heuristic $h(n)$ can never be more than the true cost $h^*(n)$: $0 \leq h(n) \leq h^*(n)$.

2.4 Search strategies

Besides the different search algorithms you can use, we also introduce different search strategies. Where most search problems use a regular search, called *forward search*, there are other options. Since planning has a notion of looking ahead to see possible consequences, we can also use this information in a *backward search*. In forward search, we use the cost of the path to get from the start node to node n, which is $g_F(n)$. Then, we expand a next node u and we can find the cost $g_F(u)$. However, with backward search, we use the cost of the path to get from some node n to the goal node, which is $g_B(n)$. Then, we can look at the options how we could have gotten to node n and use that to find the previous node v to get $g_B(v)$.

With these two search strategies, we can also introduce a third, namely *bidirectional search*. With this strategy, you generally keep two search frontiers (one forward and one backward) and alternate which one to expand next. Then, ideally, the two searches meet in the middle to complete a path. However, it may also be that the two searches completely miss each other, and then you are doing twice as much work without necessarily finding a better path. So, deciding which of the two to expand next is very important and an active area of research.

Finally, there is a different set of strategies that does not do regular search at all. You can also position your problem as a Constraint Satisfaction Problem, by encoding the state into some data structure, which often becomes somewhat of a black box. Then, you can keep a variable for the action at time t which will eventually compose your plan. The goal is encoded as a set of constraints that must hold. While this allows you to use any of the general solvers (like constraint programming, Boolean satisfiability, or mixed integer programming), the state space is often very big and increased due to the time component. So, you need a very strong formulation to get results within a meaningful computation time.

3 Modeling problems

So, how do you model problems for a planning domain? As shown in Figure 2, we use one modeling language to define a range of problem domains. First, we introduce STRIPS: the Stanford Research Institute Problem Solver, which provides a language, solver, and search procedure for planning problems. This was introduced in 1971 to program 'Shakey' the robot [2]. In STRIPS, we define a problem as a tuple of $\langle P, A, I, G \rangle$ with a set of predicates P (things that can be either true or false, remember these from the logic lecture), a set of actions A (what can the agent do), an initial state I (atoms that hold at the start of the problem setting), and a goal state G (atoms that should hold in the end). Here, we talk about *atoms* which are ground predicates: they are instantiated with a specific object from the problem instance. Usually, objects that are not grounded are named with a capital letter, while specific objects have names starting with a lower case.

For example, consider the problem shown in Figure 7, where we have to connect the right wires to turn on the power. In this case we have predicates like `(connected Link)`, `(link Link1 Link2)`, `(color Link Color)`, and more. These predicates have a type of object, and we can create atoms, like `(connected l1)`, `(link l1 r4)`, `(color l1 purple)`. Actions can be `(connect Link1 Link2)` or `(turn-on-power)`, which we can ground to get the concrete action `(connect l2 r2)`, for example. In the initial state, we know the following (and more) holds: `(power-off)` and `(color l3 red)` and the goal is `(power-on)`.

In the STRIPS formalism, a state is a conjunction of atoms that currently hold. In other words, planning uses a *factored representation* of the world, where a state consists of a vector of (Boolean) attribute values. In a complete state, all other predicate instantiations are assumed to be false, while in a partial state,

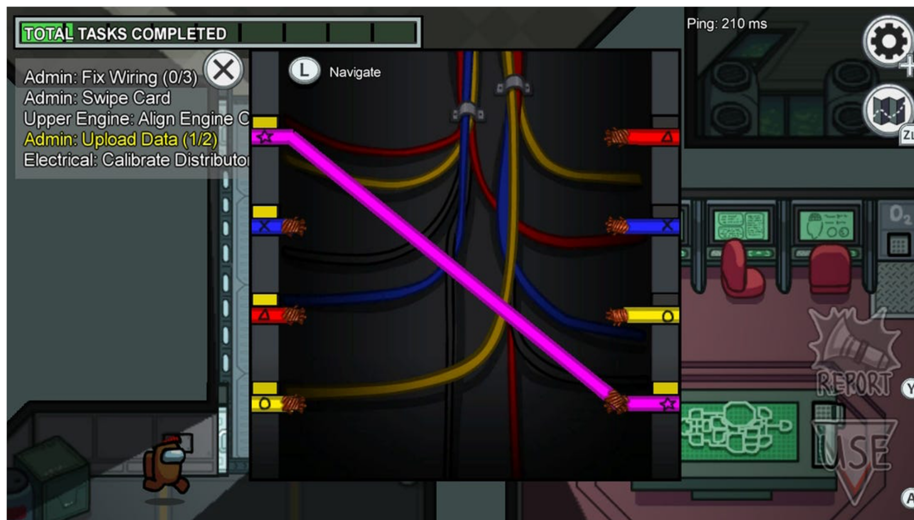


Figure 7: Example problem of linking the correct wires to turn on the power.

the other predicates' values do not matter. Every action a is defined using the preconditions $pre(a)$ (set of predicates that must hold to execute a), the negative effects $del(a)$ (set of predicates that are removed from the state), the positive effects $add(a)$ (set of predicates that are added to the state). You can optionally also define a cost $c(a)$ of executing the action, which allows you to control the definition of an optimal plan more specifically.

An action is applicable if the preconditions hold true in the current state. After applying action a to state s, we progress the state as follows: $\textsc{progress}(s, a) = (s - \textsc{del}(a)) + \textsc{add}(a)$. In our example from Figure 7, the action `(turn-on-power)` has preconditions $\textsc{pre}(a) = \{$`(connected r1)`, `(connected r2)`, `(connected r3)`, `(connected r4)`$\}$, and effects $\textsc{del}(a) = \{$`(power-off)`$\}$ and $\textsc{add}(a) = \{$`(power-on)`$\}$. Finally, we can stop searching for a plan once we encounter the goal, when $G \subseteq s$.

3.1 How to model a problem?

The study of problem modeling is often referred to as Knowledge Engineering for planning and scheduling. In most cases, you start with a short natural language description of the problem and you are asked to create a model that can be solved using a domain-independent solver (a planner in this case). For example, we get the following description:

> In a grid world, there is a truck agent, which needs to pick up packages and deliver them.

This is pretty vague, and you might ask questions like 'Do all packages have the same target cell?' or 'Can there be multiple packages at the same cell?'. In

many cases, these things are not further specified, so you have to make choices as the modeler. You could implement different choices and compare the results: can one model be solved considerably faster than the other or can one model solve more problem instances? Let's walk through the steps of creating a model for this action. We want to derive a set of predicates that define a state as well as a set of actions (with pre and post-conditions).

We already know that the state space is defined by a grid world, so we can only move to adjacent cells. So, we need a `cell` type of object, and we can define adjacency by creating a predicate like `(adjacent Cell1 Cell2)`. Then, for a move action we can check in the preconditions whether the cell we want to move to is adjacent to the current cell. This also requires that we know what cell we are currently in, so we define an `(at Agent Cell)` predicate to say the agent is at a particular cell. So, we conclude we also need a `Agent` type of object (or maybe `Truck`). If we assume that multiple agents and objects can be at the same cell, then this is all we need for a move action (otherwise we need an additional check for an empty cell in the precondition).

We are expected to deliver a package, so we need to know where they are at, which could use the same predicate `(at Package Cell)` but now for another object type `Package`. This is something that can be specified in the initial and goal state: where packages are located and where they need to go? So, we can define a `pick-up` action to pick up a package, with the precondition that both the agent and the package are currently at the same cell. The post-condition (or effect) will be that the package is no longer at that cell, but it is now picked up by the agent, so we need a predicate `(carrying Agent Package)` that says whether an agent is carrying a certain package. Finally, we need an action `put-down` to drop packages at the desired cell.

If we assume that no agent can carry two packages at the same time, then we also need to check this before we can pick up a package. Then, we could something like `(not (exists Package) (carrying Agent Package))` which would say that there is currently no package for which it is true that the agent is holding p. However, this is often not very efficient to check during solving. Instead, we can create a predicate `(empty Agent)` to say that agent is empty, and not carrying a package. Then, we simply add the updates of this predicate to the postconditions of pickup and putdown, and we can quickly check whether this holds before we pick up any package. Constraints like this are very common in many planning problems, so it is good to think about how you can model this in a way that a solver can process it efficiently.

3.2 PDDL

The more general planning formalism is called the Planning Domain Definition Language (PDDL)[1], which is the standard planning language in the research community. PDDL contains the STRIPS formalism but can express more than STRIPS[2], and is thus also a factored world representation. Various planners

[1] `https://planning.wiki/`

[2] `https://users.cecs.anu.edu.au/~patrik/pddlman/writing.html`

have been developed that support PDDL, driven by the International Planning Competition, which is held bi-annually (roughly). Sometimes, predicates are referred to as fluents, while sometimes fluents are used to express numeric fluents, which is an extension of PDDL that allows users to define numerical predicates instead of only boolean predicates. Other PDDL extensions have been developed too, like the possibility of defining conditional effects or durative actions, to express more dynamic and temporal problem features.

In PDDL, users define a domain file, and can then specify one or more problem instance files. The domain specifies the predicates that can be true or false as well as the possible actions. Actions in PDDL only have preconditions and effects, so there is no explicit difference between positive and negative effects. Domains can further define requirements, which are special features that are used in the domain. An example that is used a lot is `typing`, which allows you to specify object types. The problem file instantiates objects (of a type if you use typing), the atoms for the initial state and the atoms for the goal state. Finally, domains can also specify constants, which are object instantiations that occur in all problem instances. However, you can also encode the types of objects as predicates, the difference is shown in Appendix A. Another extension of PDDL to define numeric fluents is shown in Appendix B.

An example of the syntax is given in Listing 1 (domain) and Listing 2 (problem) for a well-known problem in the planning community called Blocksworld (example situation shown in Figure 8) and subsection 3.3 gives further explanation on this example. PDDL uses a LISP-like syntax, so there are a lot of brackets used. Other things to note are using `?var` for denoting variables in the domain and using `?var - type` to specify the type of a domain (only possible when using the `typing` requirement).

Many planners are publicly available. You can have a look at `https://editor.planning.domains` where you can edit problems online and choose from a selection of solvers. This will then output a plan like in Listing 3. However, a lot of planners use their own parsing, so not all planner will support the same files, and some may fail on a feasible problem. Most solvers will also output some statistics, like the number of nodes expanded or the time it needed to search for the plan.

In the labs, you will use a Python library for PDDL called unified-planning[3]. You can write PDDL yourself and parse it using the library, but you can also use the functions to construct the PDDL as Python objects. As an example, Listing 4 shows the code for defining this Blocksworld problem in Python using the library. Note, that here the problem and domain are specified together instead of separate files. More information on modeling can be found in this tutorial[4].

[3] `https://unified-planning.readthedocs.io/en/latest/`

[4] `https://github.com/aiplan4eu/unified-planning/blob/master/docs/notebooks/01-basic-example.ipynb`

```
(define (domain blockworld)
  (:requirements :strips :typing)
  (:types block)
  (:predicates
    (on ?x - block ?y - block)
    (ontable ?x - block)
    (clear ?x - block)
    (handempty)
    (holding ?x - block)
  )
  (:action pick-up
    :parameters (?x - block)
    :precondition (and (clear ?x) (ontable ?x) (handempty))
    :effect (and
      (not (ontable ?x))
      (not (clear ?x))
      (not (handempty))
      (holding ?x)))
  (:action put-down
    :parameters (?x - block)
    :precondition (holding ?x)
    :effect (and
      (not (holding ?x))
      (clear ?x)
      (handempty)
      (ontable ?x)))
  (:action stack
    :parameters (?x - block ?y - block)
    :precondition (and (holding ?x) (clear ?y))
    :effect (and
      (not (holding ?x))
      (not (clear ?y))
      (clear ?x)
      (handempty)
      (on ?x ?y)))
  (:action unstack
    :parameters (?x - block ?y - block)
    :precondition (and (on ?x ?y) (clear ?x) (handempty))
    :effect (and
      (holding ?x)
      (clear ?y)
      (not (clear ?x))
      (not (handempty))
      (not (on ?x ?y))))
)
```

Listing 1: Domain syntax Blocksworld

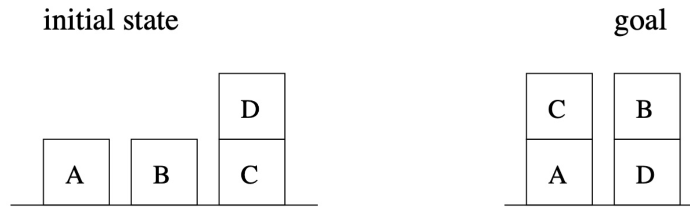


Figure 8: Blockworld example problem.

```
(define (problem blocksworld-4-2)
      (:domain blocksworld)
(:objects D B A C - block)

(:init
      (clear A)
      (clear B)
      (clear D)
      (ontable C)
      (ontable A)
      (ontable B)
      (on D C)
      (handempty)
)
(:goal (and (on C A) (on B D)))
)
```

Listing 2: Problem syntax Blocksworld

```
(unstack d c)
(put-down d)
(pick-up b)
(stack b d)
(pick-up c)
(stack c a)
```

Listing 3: Feasible plan for problem above using the ENHSP (2020) planner.

3.3 Blocksworld in PDDL

The Blocksworld problem [4] defines a world with blocks on a table and a hand that can move blocks, one at a time. The state of the world is defined by the predicates (given in lines 5-9 of Listing 1)

- `(on ?a ?b)` to say that a block a is directly on top of a block b,
- `(on-table ?a)` to say that a block a is a placed on the table,
- `(clear ?a)` to say that there is no other block on top of a,
- `(holding ?a)` to say that the hand is holding block a, and
- `(handempty)` to say that the hand is not holding any block.

There are four actions to manipulate the world state. First, we consider `pick-up` action (Listing 1, line 11-18), which allows blocks that are currently `on-table` to be picked up. The preconditions clarify that the hand can only pick up something if it is not already holding another bock, and there cannot be another block on top of the block (because we can only pick up single blocks. The result of this action is to be holding the block, which means that all the preconditions no longer hold.

The `unstack` action (Listing 1, line 27-35) is very similar to the `pick-up`, but in this case the hand is not picking up a block from the table, but the hand picks up a block that is currently on top of another block. We need a different action because we have to update the `clear` predicate for the block below the one that the hand is picking up.

Similarly, we have to actions for dropping a block that the hand is currently holding, you can either use `put-down` to place it on the table, or `stack` to put it on top of another block.

The problem file specifies that in the example shown in Figure 8, there are. four blocks: A, B, C, D (line 3, Listing 2. We can see that in the initial state, three blocks are on the table, while D is stacked on top of C. Blocksworld instances generally start in a state where the hand is not holding anything. The goal state is only partially defined: C must be on top of A, and B must be on top of D, but there is no specification saying that A and D must be on the table. Many Blocksworld instances will have goal state with just one big tower.

Finally, we can see that the plan given in Listing 3 is optimal, we cannot find a plan that is shorter to achieve to achieve this goal.

References

[1] Malte Helmert and Hauke Lasinger. "The Scanalyzer domain: Greenhouse logistics as a planning problem". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 20. 2010, pp. 234–237.

[2] Bertram Raphael. *Robot Research At Stanford Research Institute*. Tech. rep. 64. Artificial Intelligence Centre, 1971.

[3] Wheeler Ruml et al. "On-line planning and scheduling: An application to controlling modular printers". In: *Journal of Artificial Intelligence Research* 40 (2011), pp. 415–468.

[4] John Slaney and Sylvie Thiébaux. "Blocks World revisited". In: *Artificial Intelligence* 125.1–2 (Jan. 2001), pp. 119–153. URL: https://www.sciencedirect.com/science/article/pii/S0004370200000795.

```python
import os
from unified_planning.io import PDDLReader, PDDLWriter
from unified_planning.shortcuts import *
from unified_planning.model import *

#### First we define all the components of the domain


# With unified_planning, you do not have to specify requirements explicitly.

# Types can be expressed as follows, where the argument is the string name.
Block = UserType("Block")

# You specify predicates using hte Fluent object with the following
     parameters: name in string form, the type of predicate (in this course
     we stick to boolean predicates only), and then as many more parameters
     as the predicate has, in this case 2 arguments. For each parameter, you
     can specify the name and then the type (using the defined UserTypes),
     the name you use will be used in the PDDL domain as ?var
on = Fluent('on', BoolType(), b1=Block, b2=Block)
ontable = Fluent('ontable', BoolType(), b=Block)
clear = Fluent('clear', BoolType(), b=Block)
holding = Fluent('holding', BoolType(), b=Block)
handempty = Fluent('handempty', BoolType()) # this doesn't need any object
     parameters

# Now we define the actions. We do not use any special type of actions, so we
      only consider InstantaneousActions.
pickup = InstantaneousAction('pick-up', b=Block)
b = pickup.parameter('b') # Specify the parameters for use in preconditions
     and effects
pickup.add_precondition(ontable(b)) # Define preconditions
pickup.add_precondition(clear(b))
pickup.add_precondition(handempty())
pickup.add_effect(holding(b), True) # Define effect values
pickup.add_effect(handempty(), False)
pickup.add_effect(clear(b), False)
pickup.add_effect(ontable(b), False)

putdown = InstantaneousAction('put-down', b=Block)
b = putdown.parameter('b')
putdown.add_precondition(holding(b))
putdown.add_effect(holding(b), False)
putdown.add_effect(ontable(b), True)
putdown.add_effect(clear(b), True)
putdown.add_effect(handempty(), True)

stack = InstantaneousAction('stack', b1=Block, b2=Block)
b1 = stack.parameter('b1')
b2 = stack.parameter('b2')
stack.add_precondition(holding(b1))
stack.add_precondition(clear(b2))
stack.add_effect(on(b1, b2), True)
stack.add_effect(clear(b1), True)
stack.add_effect(clear(b2), False)
stack.add_effect(holding(b1), False)
stack.add_effect(handempty(), True)

unstack = InstantaneousAction('unstack', b1=Block, b2=Block)
b1 = unstack.parameter('b1')
b2 = unstack.parameter('b2')
unstack.add_precondition(on(b1, b2))
unstack.add_precondition(clear(b1))
unstack.add_precondition(handempty())
unstack.add_effect(on(b1, b2), False)
unstack.add_effect(clear(b1), False)
unstack.add_effect(clear(b2), True)
```

```
unstack.add_effect(handempty(), False)
unstack.add_effect(holding(b1), True)


#### Now define the problem adding all parts
problem = Problem('blocksworld')
# You can add multiple fluents at once by adding the object
problem.add_fluents([handempty, clear])
# Or one at a time if you want to define the initial value
problem.add_fluent(on, default_initial_value=False)
problem.add_fluent(ontable, default_initial_value=False)
problem.add_fluent(holding, default_initial_value=False)
# Similarly for actions
problem.add_action(pickup)
problem.add_actions([putdown, stack, unstack])

### We can now add the problem instance information.
# We can add objects, by creating an object of a certain UserType
a = Object('a', Block)
b = Object('b', Block)
c = Object('c', Block)
d = Object('d', Block)
problem.add_object(a)
problem.add_objects([b, c, d])
# We also have to define the initial values for all predicates. Because we
     used the default_initial_value when adding fluents to the domain, we
     only need to specify true values
problem.set_initial_value(clear(a), True)
problem.set_initial_value(clear(b), True)
problem.set_initial_value(clear(c), False)
problem.set_initial_value(clear(d), True)
problem.set_initial_value(ontable(a), True)
problem.set_initial_value(ontable(b), True)
problem.set_initial_value(ontable(c), True)
problem.set_initial_value(on(d, c), True)
problem.set_initial_value(handempty(), True)
# And we have to define the goal
problem.add_goal(on(c, a))
problem.add_goal(on(b, d))


# # We can now solve the problem
with OneshotPlanner(name='fast-downward') as planner: # also `pip install
     unified_planning[engines]`
    result = planner.solve(problem)
    if result.status == up.engines.PlanGenerationResultStatus.
     SOLVED_SATISFICING:
        print("Fast Downward returned: %s" % result.plan)
    else:
        print("No plan found.")

# We can also write the problem to a file
writer = PDDLWriter(problem)
writer.write_domain(os.getcwd() + '/blocksworld_domain.pddl')
writer.write_problem(os.getcwd() + '/blocksworld_problem.pddl')

# And if you define your model manually in PDDL you can read in the problem
     as follows.
reader = PDDLReader()
problem_read = reader.parse_problem(os.getcwd() + '/blocksworld_domain.pddl',
      os.getcwd() + '/blocksworld_problem.pddl')
with OneshotPlanner(name='fast-downward') as planner: # also `pip install
     unified_planning[engines]`
    result = planner.solve(problem_read)
    if result.status == up.engines.PlanGenerationResultStatus.
     SOLVED_SATISFICING:
        print("Fast Downward returned: %s" % result.plan)
    else:
```

```
        print("No plan found.")
```

Listing 4: Defining a PDDL model in the Unified Planning Python library.

A Typing

A.1 Using a requirement

With the requirement, you can specify the use of object types. These are defined with a '-' and work horizontally: all names before a dash are of the same type. This is true for both the type definition and the use in preconditions for example.

```
(:requirements :typing)
(:types
    vehicle location package
    car truck - vehicle)
(:objects
    truck1 - truck
    package1 - package)
(:predicates
    (at ?v - vehicle ?l - location)
    (carry ?t - truck ?p - package)
    (move ?l1 ?l2 - location))
```

A.2 Using predicates

When you don't want to use the requirement for typing, for example because a planner may not support it (although most do), then you can also use predicates. These are then defined in both the `init` and you also use them in the precondition to make sure the correct objects are inferred.

```
(:predicates
    (at ?v ?l)
    (carry ?t ?p)
    (package ?p)
    (truck ?c)
    (vehicle ?v))
(:objects truck1 package1 loc1 loc2)
(:init
    (truck truck1)
    (package package1))
(:precondition (and (carry ?t ?p)
    (truck ?t) (package ?p)
))
```

B Numeric Fluents

Numeric fluents allow users to specify values. They can be global (like `battery` below, or use parameters (like `distances`). They can be used both in preconditions and effects, and need to be initialized with a value in the `init`.

```
(:predicates (at ?loc) ... )
(:functions
    (distance ?loc1 ?loc2)
    (battery))
(:init
    (distance l1 l2 50)
    (battery 100))
(:precondition
    (> (battery) 0)
    (at ?l2))
(:effect
    (decrease (battery) (distance ?l1 ?l2))
)
```