# Logics and Symbolic AI
# Planning

Nils Holzenberger

September 29, 2025

# Outline

# Outline

# The task

- Description: see `https://ailab.r2.enst.fr/LKR/TP2.html`
- Famous toy problem in planning
- The world can be fully described by its *state*
- It is possible to go from one state to another using *actions*
- This vocabulary is identical in *reinforcement learning*
- *Planning* is the task of finding a sequence of actions going from initial to final state

# Stanford Research Institute Problem Solver

- Stanford Research Institute Problem Solver (STRIPS), 1971
- Planning strategy, and also formal representation for problems
- The basic principles of this language are still in use

# Knowledge representation

Describe the state of the world

- Monkey is in `a`, box is in `b`, banana is at `c`

# Knowledge representation

Describe the state of the world

- Monkey is in `a`, box is in `b`, banana is at `c`
  `At(a), BoxAt(b), BananaAt(c), Level(low)`

# Knowledge representation

Describe the state of the world

- Monkey is in a, box is in b, banana is at c
  `At(a), BoxAt(b), BananaAt(c), Level(low)`
- Monkey is in a, box is in a, monkey is on top of box, banana is at c

# Knowledge representation

Describe the state of the world

- Monkey is in a, box is in b, banana is at c
  `At(a), BoxAt(b), BananaAt(c), Level(low)`
- Monkey is in a, box is in a, monkey is on top of box, banana is at c
  `At(a), BoxAt(a), BananaAt(c), Level(high)`

# Knowledge representation

Describe the state of the world

- Monkey is in a, box is in b, banana is at c
  `At(a), BoxAt(b), BananaAt(c), Level(low)`
- Monkey is in a, box is in a, monkey is on top of box, banana is at c
  `At(a), BoxAt(a), BananaAt(c), Level(high)`
- Monkey is in b, box is in a, monkey holds banana

# Knowledge representation

Describe the state of the world

- Monkey is in a, box is in b, banana is at c
  `At(a), BoxAt(b), BananaAt(c), Level(low)`
- Monkey is in a, box is in a, monkey is on top of box, banana is at c
  `At(a), BoxAt(a), BananaAt(c), Level(high)`
- Monkey is in b, box is in a, monkey holds banana
  `At(b), BoxAt(a), Have(banana)`

# Knowledge representation

Describe the state of the world

- Monkey is in a, box is in b, banana is at c
  `At(a), BoxAt(b), BananaAt(c), Level(low)`
- Monkey is in a, box is in a, monkey is on top of box, banana is at c
  `At(a), BoxAt(a), BananaAt(c), Level(high)`
- Monkey is in b, box is in a, monkey holds banana
  `At(b), BoxAt(a), Have(banana)`
- Predicates are capitalized words, constants are lowercase words

# Knowledge representation

Initial state: monkey is at a, on the floor, box is at c, banana is at b.

# Knowledge representation

Initial state: monkey is at a, on the floor, box is at c, banana is at b.

`BananaAt(b), At(a), Level(low), BoxAt(c)`

# Knowledge representation

Initial state: monkey is at a, on the floor, box is at c, banana is at b.

`BananaAt(b), At(a), Level(low), BoxAt(c)`

Goal state: monkey is at a and has the banana.

# Knowledge representation

Initial state: monkey is at a, on the floor, box is at c, banana is at b.
`BananaAt(b), At(a), Level(low), BoxAt(c)`

Goal state: monkey is at a and has the banana.
`Have(banana), At(a)`

# Knowledge representation

Monkey can move from `X` to `Y`

```
Move(X, Y)
Preconditions:  At(X), Level(low)
Postconditions: !At(X), At(Y)
```

Predicates are capitalized words, constants are lowercase words, variables are capitalized words.

# Knowledge representation

Climbing up the box

`ClimbUp(Location)`

`Preconditions:`

# Knowledge representation

Climbing up the box

```
ClimbUp(Location)

Preconditions: At(Location), BoxAt(Location), Level(low)
```

# Knowledge representation

Climbing up the box

`ClimbUp(Location)`

`Preconditions: At(Location), BoxAt(Location), Level(low)`

`Postconditions:`

# Knowledge representation

Climbing up the box

```
ClimbUp(Location)

Preconditions: At(Location), BoxAt(Location), Level(low)

Postconditions: Level(high), !Level(low)
```

# Knowledge representation

Climbing down from the box

`ClimbDown(Location)`

`Preconditions:`

# Knowledge representation

Climbing down from the box

`ClimbDown(Location)`

`Preconditions: At(Location), BoxAt(Location), Level(high)`

# Knowledge representation

Climbing down from the box

`ClimbDown(Location)`

`Preconditions: At(Location), BoxAt(Location), Level(high)`

`Postconditions:`

# Knowledge representation

Climbing down from the box

`ClimbDown(Location)`

`Preconditions: At(Location), BoxAt(Location), Level(high)`

`Postconditions: Level(low), !Level(high)`

# Knowledge representation

Moving the box around

`MoveBox(X, Y)`

`Preconditions:`

# Knowledge representation

Moving the box around

`MoveBox(X, Y)`

`Preconditions: At(X), BoxAt(X), Level(low)`

# Knowledge representation

Moving the box around

```
MoveBox(X, Y)

Preconditions: At(X), BoxAt(X), Level(low)

Postconditions:
```

# Knowledge representation

Moving the box around

`MoveBox(X, Y)`

`Preconditions: At(X), BoxAt(X), Level(low)`

`Postconditions: BoxAt(Y), !BoxAt(X), At(Y), !At(X)`

# Knowledge representation

Taking the banana

`TakeBanana(Location)`

`Preconditions:`

# Knowledge representation

Taking the banana

`TakeBanana(Location)`

`Preconditions: BananaAt(Location), At(Location), Level(high)`

# Knowledge representation

Taking the banana

`TakeBanana(Location)`

`Preconditions: BananaAt(Location), At(Location), Level(high)`

`Postconditions:`

# Knowledge representation

Taking the banana

`TakeBanana(Location)`

`Preconditions: BananaAt(Location), At(Location), Level(high)`

`Postconditions: Have(banana)`

# Finding the solution

Run the program

Run the trace

Backtracking:

- it can solve any problem
- it can spend time exploring pointless strategies
- it's up to programmer to encode knowledge about useless strategies to avoir exploring them

# Planning with STRIPS

- Encode relevant knowledge
- Run the solver long enough to find a solution or realize there is none
- To speed things up, write better program, exploit corner cases, or build better hardware

# Outline

# Blocks

## Blocks world

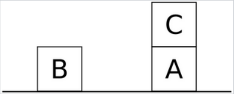
文A 3 languages

Article Talk    Read Edit View history Tools

From Wikipedia, the free encyclopedia

*This article is about the general concept in computer science research. For the sandbox video game, see Blocksworld.*

The **blocks world** is a planning domain in artificial intelligence. It consists of a set of wooden blocks of various shapes and colors sitting on a table. The goal is to build one or more vertical stacks of blocks. Only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved. Moreover, some kinds of blocks cannot have other blocks stacked on top of them.[1]



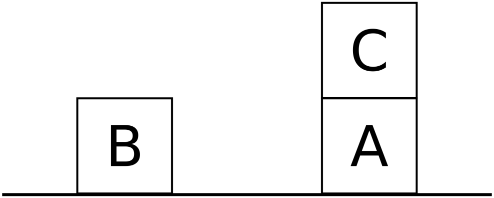

Step 1 of the Sussman anomaly, a problem in which an agent must recognise the blocks and arrange them into a stack with A at the top and C at the bottom

The simplicity of this toy world lends itself readily to classical symbolic artificial intelligence approaches, in which the world is modeled as a set of abstract symbols which may be reasoned about.

# Blocks

Blocks can be stacked on top of one another

There is a special block called `t` (for *table*)




`Table(t), On(b,t), Clear(b), On(a,t), On(c,a), Clear(c)`

# Blocks actions

Move block from atop one block to another block

```
MoveBlocks(BlockMoved, BlockFrom, BlockTo)
Pre: !Table(BlockMoved), Clear(BlockMoved), Clear(BlockTo),
     On(BlockMoved, BlockFrom)
Post: On(BlockMoved, BlockTo), !On(BlockMoved, BlockFrom),
      !Clear(BlockTo), Clear(BlockFrom)
```

# Blocks actions

Move block from table to another block

```
MoveFromTable(BlockMoved, BlockFrom, BlockTo)
Pre: !Table(BlockMoved), Table(BlockFrom), Clear(BlockMoved),
     Clear(BlockTo), On(BlockMoved, BlockFrom)
Post: On(BlockMoved, BlockTo), !On(BlockMoved, BlockFrom),
      !Clear(BlockTo)
```

# Blocks actions

Move block from block to table

```
MoveToTable(BlockMoved, BlockFrom, BlockTo)
Pre: !Table(BlockMoved), Table(BlockTo), Clear(BlockMoved),
     On(BlockMoved, BlockFrom)
Post: On(BlockMoved, BlockTo), !On(BlockMoved, BlockFrom),
      Clear(BlockFrom)
```

# Planning

- Context: yearly competitions on planning
- Henry Kautz and Bart Selman, *Planning as satisfiability*, European conference on Artificial intelligence 1992
- Express a *planning* problem as a *SAT* problem, such that a valid plan corresponds to a solution to the SAT problem
- The corresponding SAT problem is much larger than original planning problem, but SAT solvers can be more efficient than theorem provers

# Planning as satisfiability

- Add a time variable `Move(a,b)` → `Move(a,b,1)`
- Ground all clauses, i.e. enumerate all possible values for a given predicate's arguments
  e.g. in the monkey problem, there are 9 possible `Move` actions at time step 5:
  - `Move(a,a,5)`
  - `Move(a,b,5)`
  - ...
  - `Move(c,c,5)`
- Turn a *fully grounded* clause into a boolean variable
  `Move(a,b,12)` → `move_a_b_12`
- Express the relationships between all possible clauses as logical constraints (← this is the hard part)

# Planning as satisfiability

- Express the initial state and the goal state
- Describe constraints of the problem
  - no block is the table except T
  - no block is on top of itself
  - the table is never on top of anything
  - the table is always clear
  - exactly one action per time step
- Describe explanatory frame axioms
- Describe actions

# Express the initial state

STRIPS:

```
Table(t), On(b,t), Clear(b), On(a, t), On(c, a), Clear(c)
```

SAT:

```
NOT clear-A-0 AND clear-B-0 AND clear-C-0
AND NOT on-A-A-0 AND NOT on-A-B-0 AND NOT on-A-C-0
  AND on-A-T-0
AND NOT on-B-A-0 AND NOT on-B-B-0 AND NOT on-B-C-0
  AND on-B-T-0
AND on-C-A-0 AND NOT on-C-B-0 AND NOT on-C-C-0
  AND NOT on-C-T-0
```

There is only one possible model for this SAT problem and it corresponds to the initial state

# Express the goal state

Need to fix the number of steps in the plan ahead of time. Here say 10.

STRIPS:

```
Table(t), On(c,t), On(b,c)
```

SAT:

```
on-C-T-10 AND on-B-C-10
```

There are many possible models for this SAT problem. Each corresponding state satifies the goal.

# Constraints of the problem

See lab session.

# Explanatory frame axioms

If some effect is observed, it means some action was taken

E.g. if a is on top of b at time 5 and a is not on top of b at time 4 then it means a was moved onto b at time 4.

```
(on-A-B-5 AND NOT on-A-B-4)
  => (moveFromTable-A-T-B-4 OR moveBlocks-A-C-B-4)
```

In general: a change in a state variable between time $t$ and $t+1$ implies the disjunction of actions that change this state variable at time $t$.[1]

[1] It's hard to give a general definition because it depends on the underlying problem. Approximate definition on slide 19 of https://www.cs.toronto.edu/~sheila/2542/s14/material/CSC2542s14_SATPlan.pdf

# Actions

Action $a$ has preconditions $p$ and postconditions $c$. $a$ can only be taken if $p$ is true, and if $a$ is done then $c$ must be true. This constraint corresponds to

$$a_t \Rightarrow (p_t \land c_{t+1})$$

| $a_t$ | $p_t$ | $c_{t+1}$ | $a_t \Rightarrow (p_t \land c_{t+1})$ |
|---|---|---|---|
| T | T | T | T |
| T | T | F | F |
| T | F | T | F |
| T | F | F | F |
| F | T | T | T |
| F | T | F | T |
| F | F | T | T |
| F | F | F | T |

The final model will satisfy this contraint.

- If the action was taken at time $t$, then the preconditions $p_t$ must be satisfied and the postconditions $c_{t+1}$ must hold
- If the action was not taken at time $t$, $p_t$ and $c_{t+1}$ can be true or false

# One last thing

The SAT planner assumes a fixed number of steps ahead of time

To solve the planning problem, gradually increase the maximum number of steps until the problem becomes satisfiable

# Lab session

In the lab session, you will put translate the blocks world into a SAT problem using Z3

You get a STRIPS representation of the blocks world, and a SATPLAN model

# Lab session

Two changes concerning how actions are represented:

`Move(X, Y, Z, T)` (moving block X from block Y to block Z at time T) becomes `Object(X, T)`, `Source(Y, T)`, `Destination(Z, T)`

This reduces the combinatorial explosion and makes some constraints easier to encode.

We also drop the distinction between `MoveBlocks`, `MoveFromTable` and `MoveToTable`. They are all represented using `Object`, `Source` and `Destination`, and the predicate `Table`.

`MoveFromTable` = move object from source to destination where object is not the table, source is the table, and destination is not the table