

[Multi-agent Systems]

Project 3: STRIPS Planner

Try it out: <https://romainducrocq.github.io/MAS-project/code/index.html>


Source code: <https://github.com/romainducrocq/MAS-project>

Towers of Hanoi with STRIPS and A*

Tower of Hanoi with STRIPS and A* - Number of disks: 5

Solution:
-> 31 STACKS

- 1: STACK(D1, D2, B3)
- 2: STACK(D2, D3, B2)
- 3: STACK(D1, B3, D2)
- 4: STACK(D3, D4, B3)
- 5: STACK(D1, D2, D4)
- 6: STACK(D2, B2, D3)
- 7: STACK(D1, D4, D2)
- 8: STACK(D4, D5, B2)
- 9: STACK(D1, D2, D4)
- 10: STACK(D2, D3, D5)
- 11: STACK(D1, D4, D2)
- 12: STACK(D3, B3, D4)
- 13: STACK(D1, D2, B3)
- 14: STACK(D2, D5, D3)
- 15: STACK(D1, B3, D2)
- 16: STACK(D5, B1, B3)
- 17: STACK(D1, D2, B1)
- 18: STACK(D2, D3, D5)
- 19: STACK(D1, B1, D2)
- 20: STACK(D3, D4, B1)
- 21: STACK(D1, D2, D4)
- 22: STACK(D2, D5, D3)
- 23: STACK(D1, D4, D2)



Strips:

```

"ACTIONS": {"STACK": {"PRE": [{"CLEAR": "[0, 2]", "ON": "[0, 1]", "FITS": "[0, 2]"}, {"DEL": [{"CLEAR": "[2]", "ON": "[0, 1]"}, {"ADD": [{"CLEAR": "[1]", "ON": "[0, 2]"}]}], "S": {"INIT": [{"CLEAR": "[1, 2]", "B3": "B3"}, {"ON": [{"D1": "D2", "D2": "D3", "D3": "D4", "D4": "D5", "D5": "B1"}], "FITS": [{"D1": "D2", "D1": "D3", "D1": "D4", "D1": "D5", "D2": "D3", "D2": "D4", "D2": "D5", "D3": "D4", "D3": "D5", "D4": "D5", "D4": "B1", "D5": "B1", "D5": "B1", "D1": "B2", "D2": "B2", "D3": "B2", "D4": "B2", "D5": "B2", "D1": "B3", "D2": "B3", "D3": "B3", "D4": "B3", "D5": "B3"}], "GOAL": [{"CLEAR": "[1, 1]", "B1": "B1"}, {"ON": [{"D1": "D2", "D2": "D3", "D3": "D4", "D4": "D5", "D5": "B3"}]}]}
  
```

- Prepared by: Romain Ducrocq
- Supervised by: Dr Mahdi Zargayouna

SUMMARY

0 - INTRODUCTION

0.1 - Choice of technology

1 - PRESENTATION OF THE PROBLEM

2 - STRIPS REPRESENTATION

2.1 - Predicates

2.2 - Actions

2.3 - Initial state

2.4 - Goal

2.5 - JSON translation

2.6 - State of the world

3 - SEARCH STRATEGY

3.1 - A* algorithm

3.2 - Open set

3.3 - Nodes

3.4 - Neighbors

3.5 - Heuristic

3.6 - Goal

4 - RESULTS

5 - USER MANUAL

6 - CONCLUSION

6.1 - Difficulties encountered

6.2 - Personal note

Nota Bene: the pseudo codes presented in this project are not the exact translations of my code, but simplified versions of it aimed at presenting how it works in a readable way.

Nota Bene 2: this project was developed and tested on Firefox. Although I did my best, I am not a web developer, and can't guarantee full compatibility with other browsers.

0 - INTRODUCTION

As part of the Multi-agent Systems course, I have chosen the project 3: the STRIPS Planner.

The planner build here is designed to solve the famous Tower of Hanoi puzzle for N disks with STRIPS, using an A* search strategy with a custom heuristic.

0.1 - Choice of technology

The project has been implemented in Javascript with the p5.js library and JSON. I have chosen Javascript because I find it to be the best language for experimenting new ideas and producing proofs of concept, as long as the browser's computation speed and power are not limiting. Furthermore, I am very familiar with its p5.js library, a powerful library for creative coding and visual arts, which combines mathematical functions with graphical elements, and easily enables to beautifully render complex scientific applications. Finally, I have used JSON with Ajax as input, which is more flexible and requires less parsing than a text file.

The project is published as a web site on a github page and can be tested directly on it.

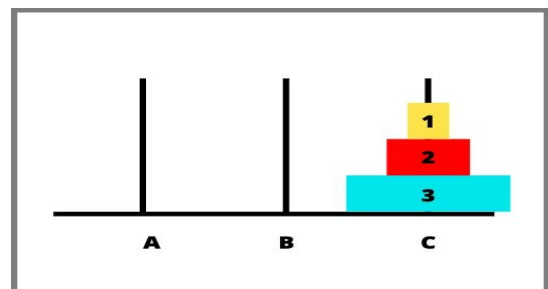
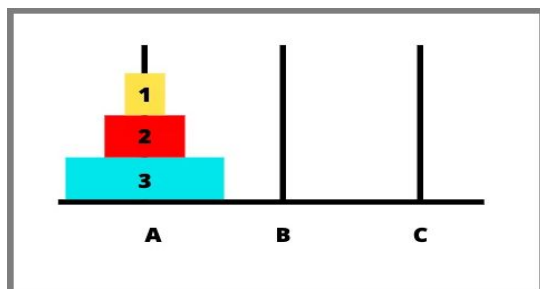
1 - PRESENTATION OF THE PROBLEM

The Tower of Hanoi is a puzzle defined by very simple rules.

It is composed of 3 bases (or rods), and a number N of disks with different sizes. At start, all disks are stacked on the first base, ordered by size, with the largest at the bottom and the smallest on top. The objective of the game is to move the entire stack from the first base to the third base, following a certain set of rules:

(1) A move consists of taking the upper disk of a stack and placing it on top of another stack, or an empty base. (2) Only one disk can be moved at a time. (3) No disk can be moved on a bigger disk. (4) All disks can be moved on all bases.

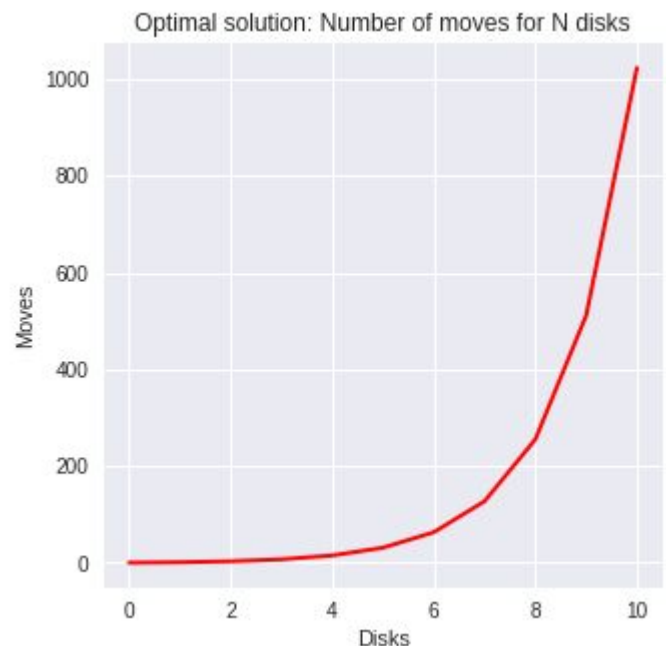
Initial and final states for 3 disks and bases A, B, C:



The problem requires 3 bases, and thus an intermediary base between the start and the goal bases, since it is a classic swap problem, in which you need a third temporary variable to exchange the values of 2 variables (swap(a, b): tmp <- a, a <- b, b <- tmp).

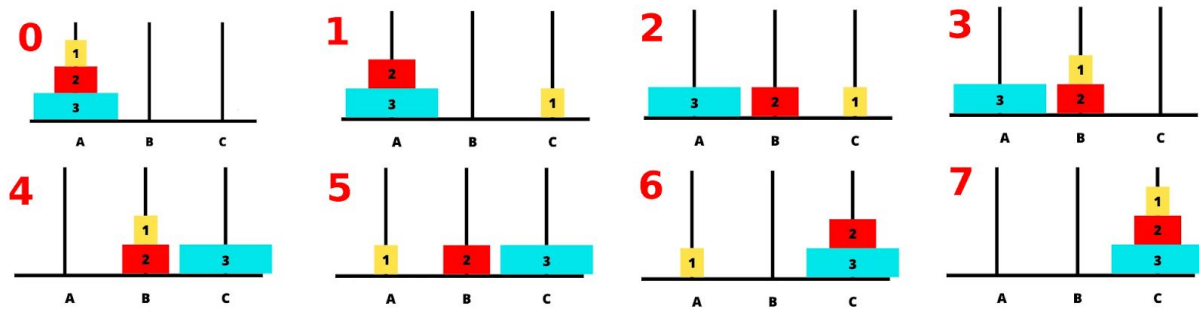
The optimal solution for N disks is $(2^N - 1)$ moves. Hence:

N disks:	Moves for optimal solution:
3	7
4	15
5	31
6	63
7	127
8	255
9	511
10	1023



We can see that the minimum number of moves is growing exponentially with the number of disks. Therefore, the problem is trivial to solve for a human with 3 disks:

DUCROCQ Romain, M2 SIA



But it becomes very hard to solve at 5 disks for a human, and virtually impossible at 8 disks. Thus, it is interesting to observe how a STRIPS planner deals with this complexity, as the problem can be decomposed solely into logical rules.

2 - STRIPS REPRESENTATION

In the Tower of Hanoi STRIPS representation, both bases and disks are represented by the same logical rules, and are equivalent objects in the eyes of the planner. Their properties are only implicitly described by the initial state to maintain a consistent state of the world.

The state of the world of the Tower of Hanoi is defined by 3 predicates and 1 action.

2.1 - Predicates

- **CLEAR(X)**: X is clear.
- **ON(X, Y)**: X is on Y.
- **FITS(X, Y)**: X fits on Y.

Semantic interpretation in a consistent world:

- **CLEAR(X)**: X is clear.
 - X is on top of a stack. (= Nothing is on top of X.)
- **ON(X, Y)**: X is on Y.
 - Y is directly under X in the same stack. (= X is directly on Y.)
 - X is a disk. (= X is not a base.)
 - X is smaller than Y. (= Y is larger than X.)
 - X is not Y.
- **FITS(X, Y)**: X fits on Y.
 - X is a disk. (= X is not a base.)
 - X is smaller than Y. (= Y is larger than X.)
 - X is not Y.

2.2 - Actions

- **STACK(A, B, C)**: Stack A from B to C.
 - PRE: CLEAR(A), CLEAR(C), ON(A, B), FITS(A, C)
 - DEL: \neg CLEAR(C), \neg ON(A, B)
 - ADD: CLEAR(B), ON(A, C)

DUCROCQ Romain, M2 SIA

Semantic interpretation in a consistent world:

Before (Stack A from B to C):

- PRE: (A is clear) AND (C is clear) AND (A is on B) AND (A fits on C).
 - A is on top of a stack.
 - C is on top of a stack.
 - B is directly under A.
 - A is smaller than B.
 - A is smaller than C.
 - A is a disk.
 - A is not B.
 - A is not C.
 - B is not C.

After (Stack A from B to C):

- DEL: (C is not clear) AND (A is on not on B)
 - C is not on top of a stack.
 - B is not directly under A.
- ADD: (B is clear) AND (A is on C)
 - B is on top of a stack.
 - C is directly under A.

Further implications:

1. There are always and only 3 CLEAR formulas in the world: each stack has one top.
2. There are always and only N ON formulas for N disks in the world: all disks are always and only on one another object, and no bases are ever on another object.
3. The set of FITS formulas is fixed and doesn't change over time.

2.3 - Initial state

The initial state of the world with N disks is defined such that:

1. The smallest disk and all bases which are not the start base are clear.
2. All the disks except the biggest one are on the disk which is directly bigger than themselves, and the biggest disk is on the start base.
3. All the disks fit on all the disks bigger than themselves, and on all the bases.

Example for 3 disks:

1. CLEAR(D1), CLEAR(B2), CLEAR(B3)
2. ON(D1, D2), ON(D2, D3), ON(D3, B1)
3. FITS(D1, D2), FITS(D1, D3), FITS(D2, D3), FITS(D1, B1), FITS(D2, B1), FITS(D3, B1), FITS(D1, B2), FITS(D2, B2), FITS(D3, B2), FITS(D1, B3), FITS(D2, B3), FITS(D3, B3)

Since the actions and the initial state are consistent, we are guaranteed to maintain a consistent state of the world, and we can accept the semantic interpretation seen previously.

2.4 - Goal

Since the set of FITS formulas doesn't change over time, the state must meet the following conditions to reach the goal:

1. The smallest disk and all bases which are not the goal base are clear.
2. All the disks except the biggest one are on the disk which is directly bigger than themselves, and the biggest disk is on the goal base.

Example for 3 disks:

1. CLEAR(D1), CLEAR(B1), CLEAR(B2)
2. ON(D1, D2), ON(D2, D3), ON(D3, B3)

2.5 - JSON translation

Using Javascript for this project, I decided to pass the possible actions, the initial state of the world and the goal to be achieved as a JSON file rather than as a plain text file. Indeed, JSON provides a flexible data-interchange format, easy to read and write, and requires minimal parsing. Thus, I could freely add as many initial states and goals as desired, for different numbers of disks, without worrying about complex parsing and the resulting errors.

For the initial states and goals, I represent my sets of formulas as arrays of arrays of string. Each possible predicate forms an array, which contains each formula as an array of parameters, which are the string identifiers of the objects of the world.

Therefore, the initial state for 3 disks is translated from plain text:

CLEAR(D1), CLEAR(B2), CLEAR(B3)

ON(D1, D2), ON(D2, D3), ON(D3, B1)

FITS(D1, D2), FITS(D1, D3), FITS(D2, D3), FITS(D1, B1), FITS(D2, B1), FITS(D3, B1),
FITS(D1, B2), FITS(D2, B2), FITS(D3, B2), FITS(D1, B3), FITS(D2, B3), FITS(D3, B3)

to JSON:

```
"INIT": {  
  "CLEAR": [  
    ["D1"], ["B2"], ["B3"]  
  ],  
  "ON": [  
    ["D1", "D2"], ["D2", "D3"], ["D3", "B1"]  
  ],  
  "FITS": [  
    ["D1", "D2"], ["D1", "D3"], ["D2", "D3"], ["D1", "B1"], ["D2", "B1"],  
    ["D3", "B1"], ["D1", "B2"], ["D2", "B2"], ["D3", "B2"], ["D1", "B3"],  
    ["D2", "B3"], ["D3", "B3"]  
  ]  
}
```

DUCROCQ Romain, M2 SIA

And the goal for 3 disks is translated from plain text:

CLEAR(D1), CLEAR(B1), CLEAR(B2)
ON(D1, D2), ON(D2, D3), ON(D3, B3)

to JSON:

```
"GOAL": {  
  "CLEAR": [  
    ["D1"], ["B1"], ["B2"]  
  ],  
  "ON": [  
    ["D1", "D2"], ["D2", "D3"], ["D3", "B3"]  
  ]  
}
```

The STACK(A, B, C) action is represented as an object with each PRE, DEL and ADD rule as sub-objects, which themselves contain the predicates as arrays of arrays of indexes. These are 0, 1, 2 and refer to the A, B, C parameters of the stack action for each formula.

Therefore, the STACK(A,B,C) action is translated from plain text:

PRE: CLEAR(A), CLEAR(C), ON(A, B), FITS(A, C)
DEL: ¬CLEAR(C), ¬ON(A, B)
ADD: CLEAR(B), ON(A, C)

To JSON:

```
"ACTIONS": {"STACK": {  
  "PRE": {  
    "CLEAR": [[0], [2]], "ON": [[0, 1]], "FITS": [[0, 2]]  
  },  
  "DEL": {  
    "CLEAR": [[2]], "ON": [[0, 1]]  
  },  
  "ADD": {  
    "CLEAR": [[1]], "ON": [[0, 2]]  
  }  
}}
```

Finally, I have implemented the function makestrips(n) to generate the initial state and the goal to achieve for a given N number of disks.

For 6 disks, the output is as follow and can be directly inserted in the JSON file:

```
"6": {"INIT": {"CLEAR": [["D1"], ["B2"], ["B3"]], "ON": [["D1", "D2"], ["D2", "D3"], ["D3", "D4"], ["D4", "D5"], ["D5", "D6"], ["D6", "B1"]], "FITS": [["D1", "D2"], ["D1", "D3"], ["D1", "D4"], ["D1", "D5"], ["D1", "D6"], ["D2", "D3"], ["D2", "D4"], ["D2", "D5"], ["D2", "D6"], ["D3", "D4"], ["D3", "D5"], ["D3", "D6"], ["D4", "D5"], ["D4", "D6"], ["D5", "D6"], ["D1", "B1"], ["D2", "B1"], ["D3", "B1"], ["D4", "B1"], ["D5", "B1"], ["D6", "B1"], ["D1", "B2"], ["D2", "B2"], ["D3", "B2"], ["D4", "B2"], ["D5", "B2"], ["D6", "B2"], ["D1", "B3"], ["D2", "B3"], ["D3", "B3"], ["D4", "B3"], ["D5", "B3"], ["D6", "B3"]]}, "GOAL": {"CLEAR": [["D1"], ["B1"], ["B2"]], "ON": [["D1", "D2"], ["D2", "D3"], ["D3", "D4"], ["D4", "D5"], ["D5", "D6"], ["D6", "B3"]]}},
```

2.6 - State of the world

In the main program, the state of the world is defined by a state object composed of 3 sets: one for each type of predicate (CLEAR Set, ON Set, FITS Set).

The sets contain the formulas describing the actual state of the world, initialized by the JSON and modified by the successive actions. The use of sets is suited here, as the only operations performed on the state of the world are `has(formula)` for PRE, `delete(formula)` for DEL and `add(formula)` for ADD, which all have $O(1)$ complexity in modern Javascript Set. Furthermore, the sets contain the formulas as their stringifications rather than as arrays, since Javascript storage and comparison are done by reference, and not by value, for non-primitive objects. With the stringification, the comparison by value becomes possible.

3 - SEARCH STRATEGY

The STRIPS planner performs a forward chaining search strategy using the A* algorithm.

3.1 - A* algorithm

A* is a path search algorithm and an extension of Dijkstra guided by a heuristic. It explores the graph by exploring the node it deems the closest to the goal, and by expanding the neighbors of this current node. To do so, it computes the f score of the node, a guess of the total distance from the initial state to the goal at this node, by summing its g score, the distance from the initial state, and its h score, the distance to the goal guessed by the heuristic. In doing so, it updates the so-called open set, storing the nodes which have yet been expanded but not explored, while each node keeps a reference to the node it comes from and the neighbors it expands to.

At each iteration, the node in the open set with the best f score is explored and removed, and its neighbors are expanded and added to the open set, if not already stored. If a neighbor is a node previously explored, its f score is improved within the new cheaper path.

The algorithm stops when either the goal is reached or the open set is empty.

If the goal is reached, the path is reconstructed by backward chaining the nodes from the last one to the first one, using the reference to the previous node stored in each node.

3.2 - Open set

In the original A* algorithm, the open set is intended to be a min heap or a priority queue. In fact, the operation of exploring the node from the open set with the lowest f score at each iteration can be performed in $O(1)$ with these types of collections, compared to $O(n)$ with other collections. However, such types do not exist in Javascript.

I therefore replaced the open set by an array with a sorting mechanism simulating a priority queue. The nodes, stored by references, are sorted with swaps based on a pivot similar to gray code, and ordered by their f score. Hence, the search for the node with lowest f score is $O(1)$, as its reference is always the first element in the array. However, checking that a node is already in the open set is still $O(n)$, as it still has to traverse the array. Moreover, we come back to the fact that Javascript objects are compared by references, and not by value.

To compensate for this, I have added a second collection, a classical Javascript Set, to check if a node is in the open set in $O(1)$. This Set doesn't store the reference to the node, but a stringification of its associated state of the world, to perform a search by value.

Thus, both the simulated priority queue and the Set store the nodes in different ways, the former to search the lowest f score and the latter to check if a node is already stored.

3.3 - Nodes

Here, the nodes represent the states of the world. They define a unique state of the world in the search graph, and are implemented by a Node class.

The Node class has the following attributes:

- Its g, h and f scores, as opposed to the original A* algorithm where they are stored separately. As the edges of the graph are not weighted, the g score is the number of states between the initial state and the current state.
- The current state of the world, as the collection of sets of formulas described previously in part 2.6.
- A stringification of the state of the world for comparison by value in the open set. This stringification is enhanced by logical operations to result in strictly identical keys for states of the world that are identical but have their formulas in a different order.
- A reference to the previous node, from which the current node was expanded, to reconstruct the search path after reaching the goal.
- A stringification of the last performed stack, to reconstruct the solution of the problem after reaching the goal.
- An array with the references to all the neighbors expanded from the current node.

The Node class has also the following methods, except from simple setters:

- setH(state), to compute the h scores with the heuristic and the current state.
- AddNeighbors(), to compute the next possible states and expand the neighbors.

3.4 - Neighbors

To expand the neighbors from the current state, we need to find all the valid next possible states of the world accessible from the current state of the world.

Therefore, we have to find all the valid Stacks(A, B, C) that can be performed in the current state of the world, i.e. the stacks meeting the PRE rule.

Simplified pseudo-code:

```
possibleStacks := empty array
```

```
foreach on in state.onSet
```

```
    if currentState.clearSet has on[0]
```

```
        foreach clear in currentState.clearSet
```

```
            if clear is not on[0]
```

```
                if currentState.fitsSet has [on[0], clear]
```

```
                    possibleStacks.push([on[0], on[1], clear])
```

DUCROCQ Romain, M2 SIA

With all the valid A, B, C combinations, we can perform DEL and ADD for each Stack.

Simplified pseudo-code:

```
nextStates := empty array
foreach stack in possibleStacks:
    nextState := currentState
    del stack[2] from nextState.clearSet
    del [stack[0], stack[1]] from nextState.onSet
    add stack[1] to nextState.clearSet
    add [stack[0], stack[2]] to nextState.onSet
    nextStates.push(nextState)
```

Now, we have all the next valid states of the world, and can expand the neighbors of the current node. The next step is to compute their heuristic to choose the best one to explore.

3.5 - Heuristic

Definition of the heuristic

A* defines the best node to explore with a heuristic function, which is a guess of the distance of a node to the goal. Here, it is a guess of the number of moves needed to finish the puzzle.

In classical graphs, if the heuristic is admissible and consistent, we are guaranteed to find the optimal path for a problem. Here, we are performing a tree search, for which A* is optimal with only an admissible heuristic function, as the edges of the graph are unweighted.

Therefore, if the distance to the goal guessed by the heuristic function is never higher than the real lowest possible distance from the node, we are sure to find the optimal number of moves for N disks in the Tower of Hanoi problem. Hence, we aim to find the best possible admissible heuristic, in order to find the optimal solution in the shortest time.

The heuristic I propose is as follow:

The number of disks not in the goal tower plus two times the number of disks in the goal tower for which there is a bigger disk not in the goal tower.

The idea here is that:

- For every disk not in the goal tower, at least one move is needed:
 1. Move the disk to the goal tower.
- For every disk in the goal tower for which there is a bigger disk not in the goal tower, at least two moves are needed:
 1. Move the smaller disk in the goal tower out of the way.
 2. Move the bigger disk to the goal tower.

Therefore, the heuristic is at best as small as the lowest number of moves, and is admissible. The solution found will be optimal.

Reconstructing the goal tower with STRIPS formulas

To compute the heuristic with the STRIPS formulas of the state of the world, we only have to recreate the goal tower, from which we will deduce all the information.

First, we need to find the goal base, as the planner doesn't know the existence of a goal base, but only of a global goal to achieve. This step is very simple, as it only requires to compare the initial state and the goal to achieve. In fact, the goal base is the only object that is clear in the initial state and is not in the goal to achieve.

With B3 as goal base: CLEAR(B3) is in the initial state, but not in the goal to achieve.

In order to reduce the number of unnecessary operations, the goal base is found only once at the beginning and stored, rather than in every heuristic computation.

Secondly, we need to reconstruct the goal tower in the state of the world. To do so, we initialize an array with the goal base, and loop through the ON(X, Y) Set, starting with the goal base as Y. For each X found, we store X in the array and assign its value to Y, and continue to loop until an X that is also in the CLEAR(X) Set is found.

For example, we can have this array in a 6 disks world:

B3	D6	D4	D2	D1
----	----	----	----	----

Computing the heuristic with the goal tower

As a reminder, the size of the ON Set is always equal to the total number of disks, as each disk is always and only on one another object, and the bases are never on another object. Thus, we know N the total number of disks.

Therefore, the number of disks not in the goal tower is simply given by:

- (Size ON Set) - (Size goal tower) + 1

The example gives: $6 - 5 + 1 = 2$ disks not in the goal tower (D5, D3).

For the number of disks in the goal tower for which there is a bigger disk not in the goal tower, we have to picture the goal tower as in the final state of the puzzle with holes:

The example can be pictured as:

B3	D6		D4		D2	D1
----	----	--	----	--	----	----

We first compare the size of the disk at the bottom with the size of the ON Set, i.e. the total number of disks. Then we go through the array comparing the size of each disk with the size of the next one. If the difference is greater than 0 for the bottom disk, and greater than one for the others, we know that a bigger disk is missing, and is therefore not in the goal tower.

By doing so and for each of these gaps, we count the number of disks that are directly following the disk after the gap, i.e. have a size difference of exactly one.

In our example:

- (Size ON Set) - (Size D6) = $6 - 6 = 0$ \Rightarrow There is no bigger disk not in the goal tower.
- (Size D6) - (Size D4) = $2 > 1$ \Rightarrow D5 is not in the goal tower, +1 disk.
- (Size D4) - (Size D2) = $2 > 1$ \Rightarrow D3 is not in the goal tower, +1 disk.
- (Size D2) - (Size D1) = 1 \Rightarrow D1 is directly following D2, +1 disk.

DUCROCQ Romain, M2 SIA

Thus, we know that for 3 disks in the goal tower (D4, D2, D1), there is a bigger disk not in the goal tower (D5, D3).

In our example, the heuristic is: $2 + 2 * 3 = 8$.

3.6 - Goal

To know if the goal is achieved, we check at every iteration of the A* search if the state of the world of the current node contains every formula of the goal to achieve.

Simplified pseudo-code:

```
goalReached := True
foreach predicate in goalSet:
    foreach formula in goalSet.predicate:
        if state.predicate has not formula
            goalReached := False
            break
```

When the goal is reached, we reconstruct the solution and pass it to the animation program. This program reconstructs the whole world for animation, and applies the moves one by one.

4 - RESULTS

Results after testing the STRIPS planner up to 8 disks:

Number of disks	Number of moves	Optimal solution	Computation time
1	1	YES	< 1s
2	3	YES	< 1s
3	7	YES	< 1s
4	15	YES	< 1s
5	31	YES	< 1s
6	63	YES	1s
7	127	YES	5s
8	255	YES	35s

The STRIPS planner finds the optimal solution regardless of the number of disks, as expected from the heuristic. We notice that the computation time is almost instantaneous up to 5 disks, and increases drastically from that point on. At 6 disks, we can notice it for the first time at a human scale, as we have to wait 1 second for the result. Same thing for 7 disks, as the algorithm takes 5 seconds to solve the problem. However, at 8 disks, we start to reach the limits of the browser, with 35 seconds of computation time.

DUCROCQ Romain, M2 SIA

As this project is mainly a proof of concept, I will not test it for more than 8 disks in the browser. To go further, and now that we know the STRIPS planner to be correctly and efficiently working, we could implement the algorithm in C++ to see up to how many disks it could perform before reaching the limits of the machine.

5 - USER MANUAL

!\\ The project is developed for Firefox, I do not guarantee compatibility with all browsers !

Launch project

I strongly recommend testing the project online on the dedicated Github page, but it can also be tested locally if desired.

1. Online:

Go to <https://romainducrocq.github.io/MAS-project/code/index.html>.

2. Locally:

You can also test the project on localhost by cloning the source from the repo or downloading the zip archive at <https://github.com/romainducrocq/MAS-project>.

The project has to be launched on localhost, since the JSON containing the STRIPS is loaded with an Ajax HTTP request, which is blocked by the browser with a CORS error if performed on the local file system, handled like an unidentified source.

1. Set up a local Python3 HTTP server at the root of the project:

```
python3 -m http.server
```

2. Test the project on localhost:

<http://localhost:8000/index.html>

Select the number of disks

The default number of disks is set to 5, and the computation is fast enough for the solution to be displayed with the page at start. The number of disks can be changed with the dropdown list at the top of the page, from 1 to 8.

The search of the solution starts immediately when selecting a disk number. When the solution has been found, the sequence of actions is displayed on the left of the page, the STRIPS representation is displayed at the bottom of the page, and the initial state and a run button appear in the animation panel.

Beware that nothing visible happens during the search of the solution, and the page will refresh only after the solution is found. Therefore, for 7 and 8 disks, it can feel like the program is stuck, while it is actually not.

Furthermore, the number of disks can be changed only when nothing is happening, i.e. the program is not currently searching the solution, nor displaying the animation. This was done in order to not mess with global variables.

DUCROCQ Romain, M2 SIA

Run the animation

When the solution is found, click on the run button to display the animation of the solution.

One move is performed every half second, and a counter displays the index of the current move in the move list. Therefore, the animation for 8 disks, with 255 moves, will last for a bit more than 2 minutes. The number of disks can't be changed during the animation other than by manually refreshing the page.

Compute the initial state and goal to achieve for a given number of disks

Using the function `makestrips(n)` in the browser's console will display the initial state and goal to achieve in STRIPS JSON language for n disks at the bottom of the page.

The browser's console can be accessed with CTRL + SHIFT + K.

Overview

Tower of Hanoi with STRIPS and A* - Number of disks: 7

Solution:
-> 127 STACKS

- 1: STACK(D1, D2, B3)
- 2: STACK(D2, D3, B2)
- 3: STACK(D1, B3, D2)
- 4: STACK(D3, D4, B3)
- 5: STACK(D1, D2, D4)
- 6: STACK(D2, B2, D3)
- 7: STACK(D1, D4, D2)
- 8: STACK(D4, D5, B2)
- 9: STACK(D1, D2, D4)
- 10: STACK(D2, D3, D5)
- 11: STACK(D1, D4, D2)
- 12: STACK(D3, B3, D4)
- 13: STACK(D1, D2, B3)
- 14: STACK(D2, D5, D3)
- 15: STACK(D1, B3, D2)
- 16: STACK(D5, D6, B3)
- 17: STACK(D1, D2, D6)
- 18: STACK(D2, D3, D5)
- 19: STACK(D1, D6, D2)
- 20: STACK(D3, D4, D6)
- 21: STACK(D1, D2, D4)
- 22: STACK(D2, D5, D3)
- 23: STACK(D1, D4, D2)
- 24: STACK(D4, B2, D5)
- 25: STACK(D1, D2, D4)
- 26: STACK(D2, D3, B2)

Strips:

```
"ACTIONS": [{"STACK": {"PRE": {"CLEAR": [{"0, 2}], "ON": [{"0, 1}], "FITS": [{"0, 2}], "DEL": {"CLEAR": [{"2}], "ON": [{"0, 1}], "ADD": {"CLEAR": [{"1}], "ON": [{"0, 2]}}}}, {"INIT": {"CLEAR": [{"D1", "B2", "B3", "ON": [{"D1", "D2", "D3", "D4", "D5", "D6", "D7", "D7", "B1"}], "FITS": [{"D1", "D2", "D1", "D3", "D1", "D4", "D1", "D5", "D1", "D6", "D1", "D7", "D2", "D3", "D2", "D4", "D2", "D5", "D2", "D6", "D2", "D7", "D3", "D4", "D3", "D5", "D3", "D6", "D3", "D7", "D4", "D5", "D4", "D6", "D4", "D7", "D5", "D6", "D5", "D7", "D6", "D7", "D1", "B1", "D2", "B1", "D3", "B1", "D4", "B1", "D5", "B1", "D6", "B1", "D7", "B1", "D1", "B2", "D2", "B2", "D3", "B2", "D4", "B2", "D5", "B2", "D6", "B2", "D7", "B2", "D1", "B3", "D2", "B3", "D3", "B3", "D4", "B3", "D5", "B3", "D6", "B3", "D7", "B3"}], "GOAL": {"CLEAR": [{"D1", "B1", "B2}], "ON": [{"D1", "D2", "D3", "D4", "D5", "D6", "D7", "D7", "B3"}]}}, {"4: {"INIT": {"CLEAR": [{"D1", "B2", "B3}], "ON": [{"D1", "D2", "D3", "D4", "D5", "D6", "D7", "D7", "B1}], "FITS": [{"D1", "D2", "D1", "D3", "D1", "D4", "D1", "D5", "D1", "D6", "D1", "D7", "D2", "D3", "D2", "D4", "D2", "D5", "D2", "D6", "D2", "D7", "D3", "D4", "D3", "D5", "D3", "D6", "D3", "D7", "D4", "D5", "D4", "D6", "D4", "D7", "D5", "D6", "D5", "D7", "D6", "D7", "D1", "B1", "D2", "B1", "D3", "B1", "D4", "B1", "D5", "B1", "D6", "B1", "D7", "B1", "D1", "B2", "D2", "B2", "D3", "B2", "D4", "B2", "D5", "B2", "D6", "B2", "D7", "B2", "D1", "B3", "D2", "B3", "D3", "B3", "D4", "B3", "D5", "B3", "D6", "B3", "D7", "B3"}], "GOAL": {"CLEAR": [{"D1", "B1", "B2}], "ON": [{"D1", "D2", "D3", "D4", "D5", "D6", "D7", "D7", "B3}]}}}]
```

6 - CONCLUSION

6.1 - Difficulties encountered

Firstly, I had a hard time choosing a good problem that would suit a STRIPS planner well. After some thinking, I came to the idea of the Tower of Hanoi because I found it visually very similar to the world of cubes, with piles of objects that had to be stacked by an agent following specific logical rules. I was attracted by the non-linear difficulty scale, seeing that it was trivial for an adult to find the solution with 3 disks, but already very hard with 5, and I wanted to see how the planner would compete. However, I found the STRIPS representation fairly quickly, and was surprised by its simplicity, almost simpler as for the world of cube.

Secondly, I struggled a lot to correctly implement the A* search algorithm, mainly due to some specificities of the Javascript language that I hadn't taken into account: the comparison by reference for non-primitive objects and the absence of sorted collections, such as min heaps, priority queues or sorted sets. I had to redefine the structure of the nodes and how my open set worked multiple times before arriving at a satisfactory solution.

Finally, the most difficult part was for me to define a good admissible heuristic, and moreover implement it with knowing only the STRIPS representation of the world. In a first version, I was reconstructing the whole world every time I computed the heuristic of a node, and counting the disks tower by tower. However, even if it was working thanks to the world being so small, I was not happy with this solution, which felt like cheating and overkill. I could not imagine that a more complex planner, like a robot, should reconstruct the whole world at every action. It took me a while, but I finally arrived at a more elegant solution where the heuristic could be determined with only the goal tower.

6.2 - Personal note

I really enjoyed this project, which was among my personal favorites of the semester. It was challenging but not overly difficult, and led to a lot of exploration, which is something I really enjoy. And with the subject being so wide, it also required a certain amount of creativity.

I was first heading for the Covid 19 simulation, but I changed my mind right after the STRIPS lecture, as I found the topic to be hugely interesting. This change was also influenced by the fact that I have already done several multi-agent projects in the past, and I wanted to discover something new. In fact, I had never heard of STRIPS planners before, and I indeed learned something really cool with this project.

Overall, I am really happy with my production, as I have gotten the exact result I had in mind, which is far from always being the case with computer science projects.

I hope you will enjoy testing the planner as much as I enjoyed programming it.