2017

# Non-Linear Planner with Goal Regression

PLANNING AND APPROXIMATE REASONING - EXERCISE 1

JOSEP FAMADAS ALSAMORA / JORDI RIU VICENTE

UNIVERSITAT ROVIRA i VIRGILI

URV | Facultat de Informàtica

# Table of content

# Table of Tables

# Table of Figures

# 1. Introduction

In this document it is faced the classical “blocks world” (BW) problem which is one of the most common AI problems in the field of planning.

In this case, there are some few modifications of the common BW domain:

- In the table there can only be a fixed number of stacks.
- Each block is characterized by a letter and a weight, which can be 1, 2, 3 or 4 kg.
- A block can only be stacked over another if its weight is lower or equal to the other’s weight.
- In order to move the blocks, we have two robotic arms. The right arm can pick up any block and the left hand only the 1kg blocks.

The objective is to design a non-linear planner with goal regression which gives a plan with a sequence of actions (from now known as *operators*) to execute in order to go from the initial state to the final one.

# 2. Domain characterization

## 2.1. Predicates

Each state (Initial, final and the intermediate ones) is characterized by the facts that are true in it. These facts will be referred in this document as *predicates*.

| *Predicate* | *Description* |
|---|---|
| *On-Table(x)* | The block *x* is on the table. |
| *On(x,y)* | The block *x* is on the block *y*. |
| *Clear(x)* | There is not any block on block *x*. |
| *Empty-Arm(a)* | The arm *a* is not holding any block. |
| *Holding(x,a)* | The block *x* is held by the arm *a*. |
| *Used-Col-Num(n)* | The number of columns with at least one block in it. |
| *Heavier(x,y)* | The block *x* is heavier than the block *y*. |
| *Light-Block(x)* | The block x weights 1kg. |

*Table 1: Predicates and its definition*

## 2.2. Operators

The operators are the actions that, when executed while being in a state, change its predicates and generates a new state which we move to.

| *Operator* | *Description* |
|---|---|
| *Pick-Up-L(x)* | Picks up the block *x* from the table with the *Left* arm |
| *Pick-Up-R(x)* | Picks up the block *x* from the table with the *Right* arm |
| *Unstack-L(x,y)* | Picks up the block *x* from the block *y* with the *Left* arm |
| *Unstack-R(x,y)* | Picks up the block *x* from the block *y* with the *Right* arm |
| *Leave(x,a)* | Leaves the block *x* on the table using the arm *a*. |
| *Stack(x,y,a)* | Leaves the block *x* on the block *y* using the arm *a*. |
| *Swap(R,L)* | Swaps the block held by the *Right* arm to the *Left* arm. |
| *Swap(L,R)* | Swaps the block held by the *Left* arm to the *Right* arm. |

*Table 2: Operators and its definition*

The way in which an operator changes the predicates of the current state is characterized with the *add list* (predicates that are not present in the current state but will be in the one generated

by the operator) and the *delete list* (predicates that are present in the current state and will not be in the next one).

Not every operator can be executed in every state, so each operator has a list of preconditions which are the predicates that must be present in a state so it can be executable.

| *OPERATORS* | *PRECONDITIONS* | *ADD LIST* | *DELETE LIST* |
|---|---|---|---|
| Pick-Up-L(x) | Empty-Arm(L)<br>On-Table(x)<br>Light-Block(x)<br>Clear(x) | Holding(x,L)<br>Used-Col-Num(n-1) | Empty-Arm(L)<br>On-Table(x) |
| Pick-Up-R(x) | Empty-Arm(R)<br>On-Table(x)<br>Clear(x) | Holding(x,R)<br>Used-Col-Num(n-1) | Empty-Arm(R)<br>On-Table(x) |
| Unstack-L(x,y) | Empty-Arm(L)<br>On(x,y)<br>Light-Block(x)<br>Clear(x) | Holding(x,L)<br>Clear(y) | Empty-Arm(L)<br>On(x,y) |
| Unstack-R(x,y) | Empty-Arm(R)<br>On(x,y)<br>Clear(x) | Holding(x,R)<br>Clear(y) | Empty-Arm(R)<br>On(x,y) |
| Leave(x,a) | Used-Col-Num(n<MAX)<br>Holding(x,a) | On-Table(x)<br>Empty-Arm(a)<br>Used-Col-Num(n+1) | Holding(x,a) |
| Stack(x,y,a) | Heavier(y,x)<br>Clear(y)<br>Holding(x,a) | On(x,y)<br>Empty-Arm(a) | Clear(y)<br>Holding(x,a) |
| Swap(R,L) | Holding(x,R)<br>Empty-Arm(L)<br>Light-Block(x) | Empty-Arm(R)<br>Holding(x,L) | Holding(x,R)<br>Empty-Arm(L) |
| Swap(L,R) | Holding(x,L)<br>Empty-Arm(R) | Empty-Arm(L)<br>Holding(x,R) | Holding(x,L)<br>Empty-Arm(R) |

*Table 3: List of the operators and its preconditions, adds and deletes*

# 3. Algorithm

As it has been said before, the algorithm used to solve this problem is the non-linear planner with goal regression.

This algorithm consists of start from the final state and go backwards by iterating a procedure:

1- Make a list of every operator that has every predicate of its *add list* in the current state and none of its *delete list*.

2- For each one of this operator generate the previous state, which is the one that, if you apply the operator in, you reach the current state. This is made by combining the preconditions list of the operator and the regression function.

3- Once all the candidates to previous state have been generated, erase the ones that cannot be true (because of having contradictory predicates) or that already exist in a previous level.

4- Finally, take each possible state one by one and repeat the algorithm from step 1 to generate the next level of states.

The algorithm ends when the initial state is reached (a plan has been found) or when at step 3 all the candidates are erased, meaning that there is no plan to go from the initial to the final state.

Most of the computational load of this algorithm is in the step 3 because it has to check every state of a really long list of states. In order to avoid so and given the fact that we have an a priori knowledge of the domain, we have given some "intelligence" to our algorithm so in the step 1 it does not generate so many operators. Some predicates have a function that, before returning all the operators that have it in their add list, check some conditions in the current state:

| *Predicate* | *Condition checked* |
|---|---|
| *On-Table(x)* | Before giving as operators Leave(x,a) for every *a* check:<br>- In the current state, *x* is on the table and has no block on it.<br>- If *a* = Left, *x* weights 1kg. |
| *On(x,y)* | Before giving as operators Stack(x,y,a) for every *a* check:<br>- In the current state, the arm *a* is empty and *x* has no block on it.<br>- If *a* = *Left, x* weights 1 kg. |

| | |
|---|---|
| *Clear(x)* | Before giving as operators Unstack(y,x) for every *y* check:<br>- *x and y* are not the same block.<br>- In the current state, *y* is held by one of the arms.<br>- *x* is heavier than *y*. |
| *Empty-Arm(a)* | Before giving as operators Leave(x,a) for every *x* check:<br>- In the current state, *x* is on the table.<br>- If *a* = *Left*, *x* weights 1kg.<br>Before giving as operators Stack(x,y) for every *x* and *y* check:<br>- *x and y* are not the same block.<br>- In the current state, x is on *y* and *x* has no block on it.<br>- If *a* = *Left*, *x* weights 1kg.<br>Before giving as operators Swap(a,b) being *b* the other arm check:<br>- In the current state, the arm *b* is holding a block.<br>- The held block weights 1kg. |
| *Holding(x,a)* | Before giving as operators Pick-Up- "a" (x) check:<br>- In the current state, the number of used columns is less than the maximum.<br>Before giving as operators Unstack- "a" (x,y) for every *y* check:<br>- *x and y* are not the same block.<br>- In the current state, *y* is clear.<br>- *y* is heavier than *x*.<br>Before giving as operators Swap(b,a) being b the other arm check:<br>- In the current state, the arm *b* is empty.<br>- *x* weights 1kg. |

*Table 4: Conditions that are checked in the step 1 of the algorithm to improve it*

# 4. Implementation

The implementation has been performed using MATLAB and consists of a main script which contains the algorithm and 4 main classes: STATE, PREDICATE, OPERATOR and BLOCK, being the second and the third abstract classes that contain other classes inside.

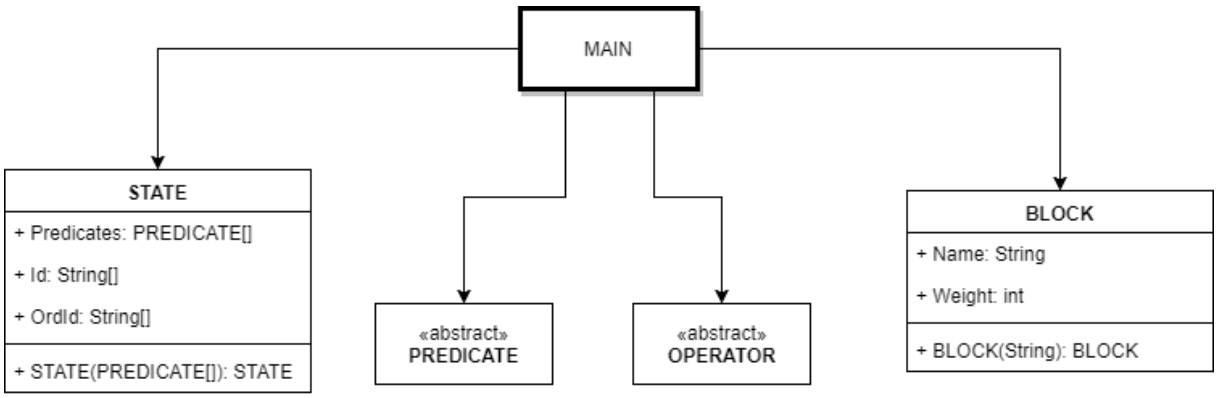


*Figure 1: Main class diagram*

## 4.1. STATE

The class consists of an attribute which is a cell of objects from any of the subclasses of PREDICATE which represents the description of the state, and another two attributes which are the transformation of the first attribute to a cell containing the Id of each predicate (Strings are much easier to compare than objects) and the same but alphabetically sorted.

The only method it has is the constructor method.

## 4.2. PREDICATE

This is an abstract class so it has not been implemented, it was just created to have an idea of the link between its subclasses. These subclasses represent the predicates described in the *Section 2.1* of this document.

They consist of some attributes that represent the parameters of the predicates (the *x*, *y*, and *a* in *Table 1*), the constructor method and two extra methods that are important for our algorithm.

The ADD method returns a cell containing the OPERATORs that have this this PREDICATE in its *add list* (after applying the "intelligence" defined in *Table 3*).

The CHECK method is used for the step 3 of the algorithm (*Section 3*).

As can be observed in *Figure 2* the predicates Heavier and Light-Block do not have their own PREDICATE subclass. This is because they did not fit in the code as classes so they have been taken into account as conditions during the execution of the algorithm.

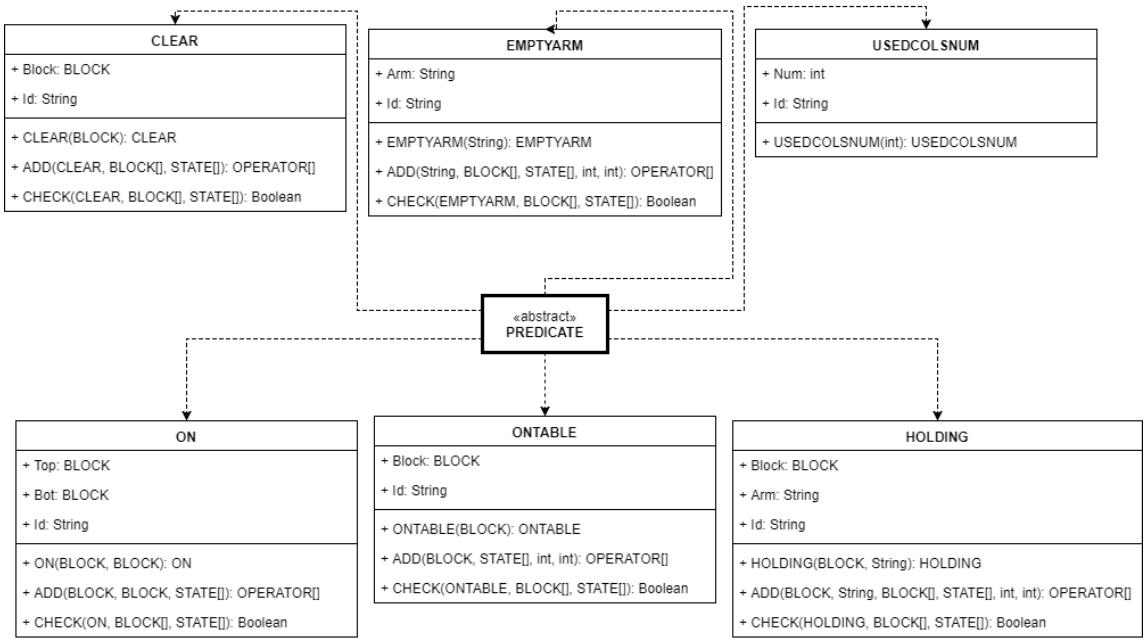


*Figure 2: Predicate class diagram*

## 4.3. OPERATOR

This is an abstract class so it has not been implemented, it was just created to have an idea of the link between its subclasses. These subclasses represent the operators described in the *Section 2.2* of this document.

They consist of some attributes that represent the parameters of the predicates (the *x*, *y*, and *a* in *Table 2*) and 3 cell of PREDICATEs which are the *add list*, the *delete list*, and the *preconditions list*.

The only method these subclasses have is the constructor method.

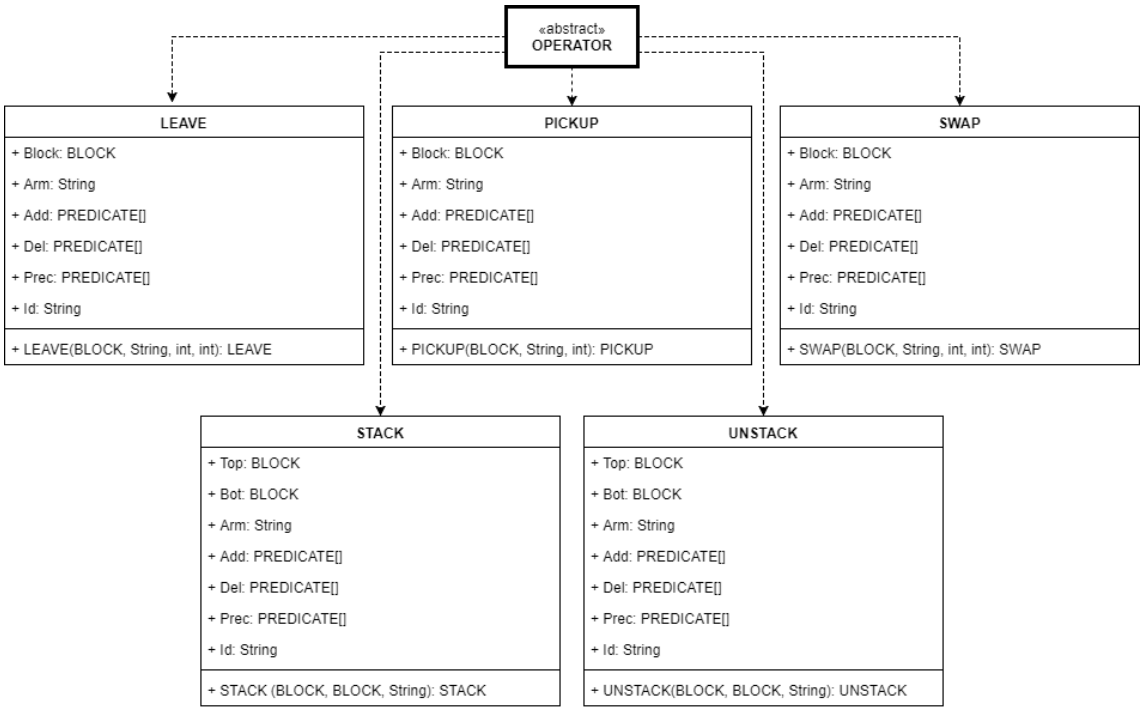


*Figure 3: Operator class diagram*

## 4.4. BLOCK

This class is only formed by a String which is the block name and its weight.

The only method it has is the constructor method which splits the letter from the weight (given as '*').

# 5. Testing and Results

## 5.1. Configuration 1

INITIAL STATE:

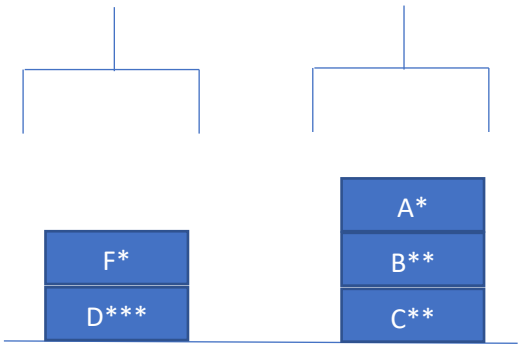


*Figure 4: Configuration 1: Initial state*

GOAL STATE:

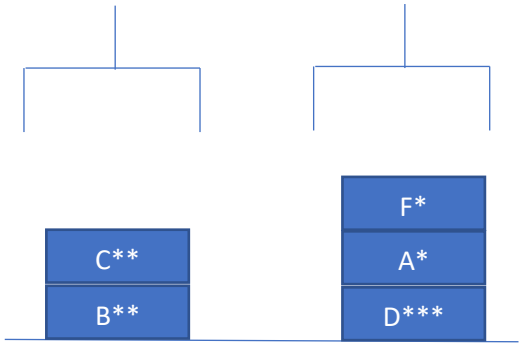


*Figure 5: Configuration 1: Goal state*

This problem has been solved under 6 different considerations: Maximum Columns (MCol) = 2,3,4,5,6 & 7.

Find in *Figure 6* the number of steps required to solve the problem as a function of the maximum number of columns allowed. Note that for MCol = 2 the problem is unsolvable.

The results on *Figure 7* and *Figure 8* show that when MCol >=4 the number of states visited by the algorithm stabilize. For MCol = 3 we find that the optimal path (which is always found by the NLP with regression) has 8 steps. Such plan remains being optimal even when we increase MCol.

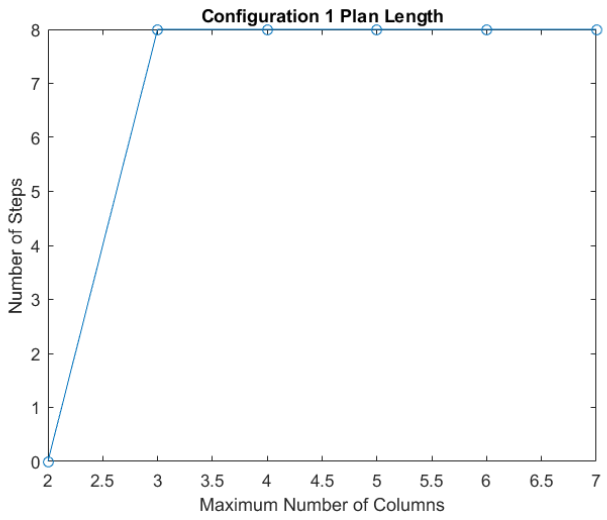


*Figure 6: Configuration 1: Plan length*

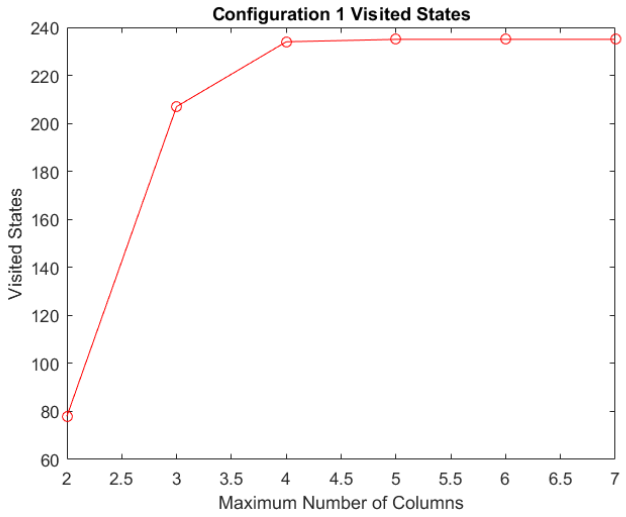


*Figure 7: Configuration 1: Visited states*

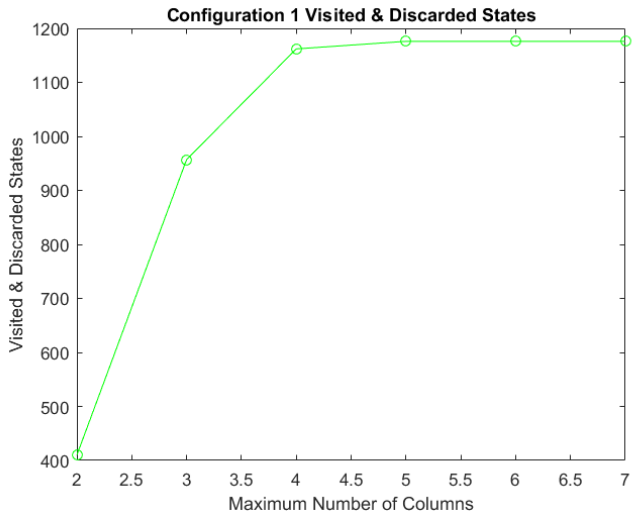


*Figure 8: Configuration 1: Visited and discarded states*

## 5.2. Configuration 2

INITIAL STATE:

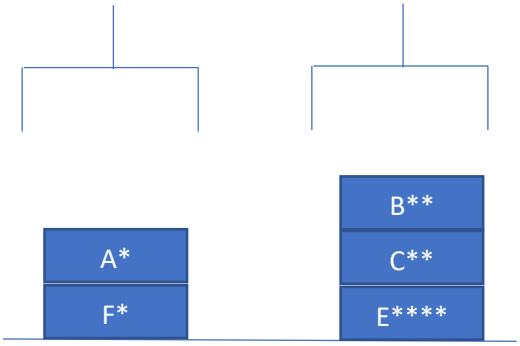


*Figure 9: Configuration 2: Initial state*

GOAL STATE:

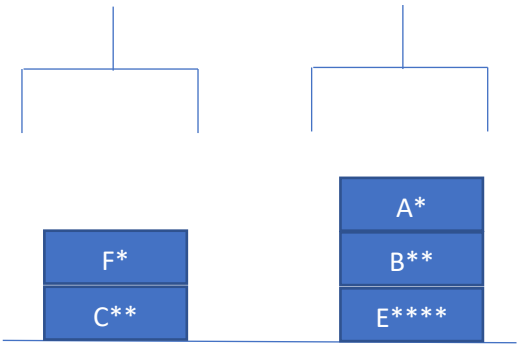


*Figure 10: Configuration 2: Goal state*

This problem has been solved under 6 different considerations: Maximum Columns (MCol) = 2,3,4,5,6 & 7.

Find in *Figure 11* the number of steps required to solve the problem as a function of the maximum number of columns allowed. Note that for MCol = 2 the problem is unsolvable. Moreover, for this configuration as in the previous one, the minimum plan length is reached for MCol >= 4. For MCol = 3 there exists a path in the states space that connects the initial and final states, but this path is suboptimal when compared with the shortest path available for MCol > = 4.

The results on *Figure 12* and *Figure 13* show that when MCol >=5 the number of states visited by the algorithm stabilize. Since the states of a lower MCol will always be contained in the State Space of a higher MCol, such stability implies that there are very few new states visited by the algorithm when increasing the value of MCol above 5. Therefore, it is unlikely to find a better path with respect to the one found for lower MCol.

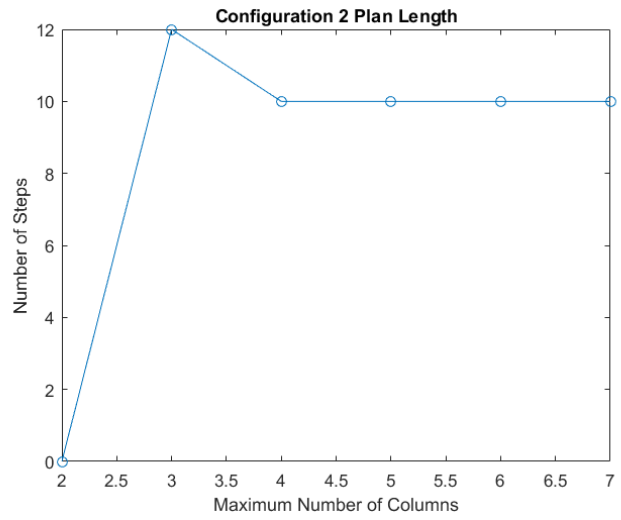


*Figure 11: Configuration 2: Plan length*

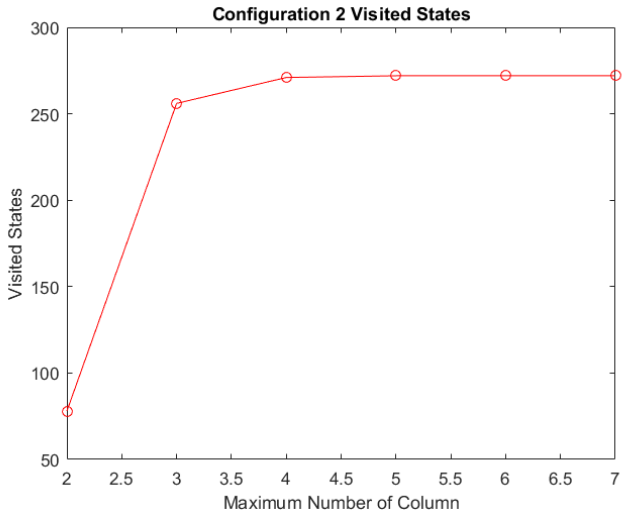


*Figure 12: Configuration 2: Visited states*

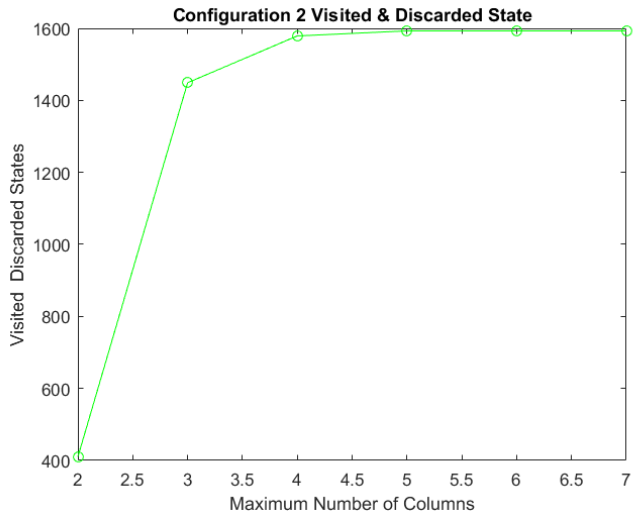


*Figure 13: Configuration 2: Visited & discarded states*

## 5.3. Configuration 3

INITIAL STATE:

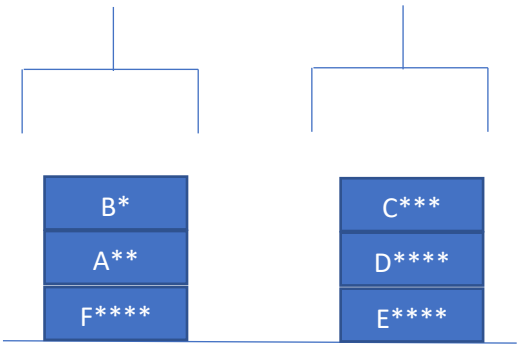


*Figure 14: Configuration 3: Initial state*

GOAL STATE:

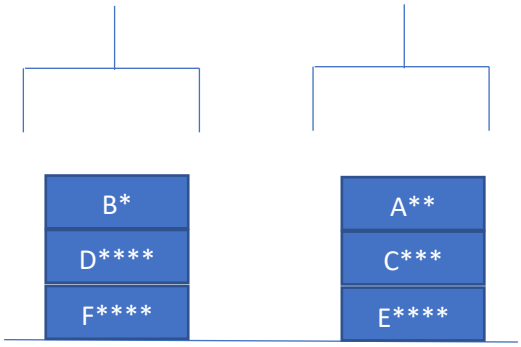


*Figure 15: Configuration 3: Goal state*

This problem has been solved under 6 different considerations: Maximum Columns (MCol) = 2,3,4,5,6 & 7.

Find in *Figure 18* the number of steps required to solve the problem as a function of the maximum number of columns allowed. Note that for MCol = 2 the problem is unsolvable. Moreover, for this configuration as in the previous one, the minimum plan length is reached for MCol >= 4. For MCol = 3 there exists a path in the states space that connects the initial and final states, but this path is suboptimal when compared with the shortest path available for MCol > = 4. A simple representation of this would be the one shown in *Figure 16* and *Figure 17*

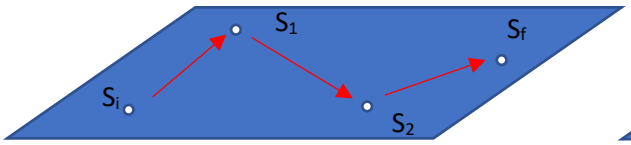


*Figure 16: States space for MCol = 3*

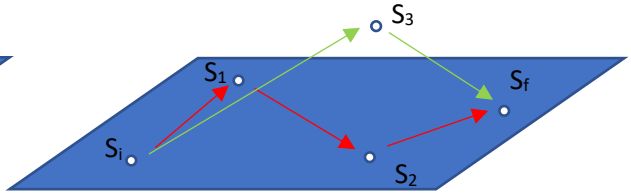


*Figure 17: States space for MCol ≥ 4*

*Figure 19* and *Figure 20* show the number of visited states and total candidate states to find the plan as a function of the maximum number of columns allowed. Although the problem is unsolvable for MCol = 2, the number of visited states is > 0. This implies that the region of the state space that is explorable from the initial configuration does not contain the desired goal state.

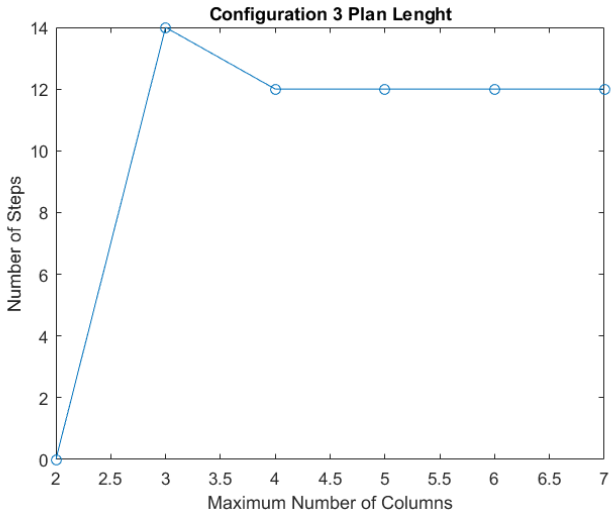


*Figure 18: Configuration 3: Plan length*

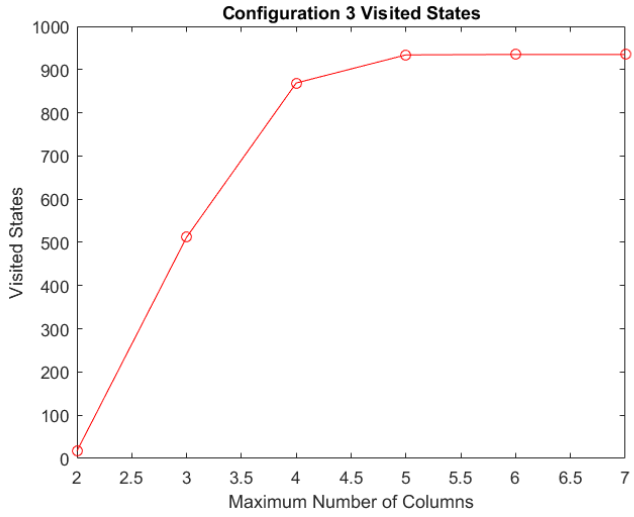


*Figure 19: Configuration 3: Visited states*

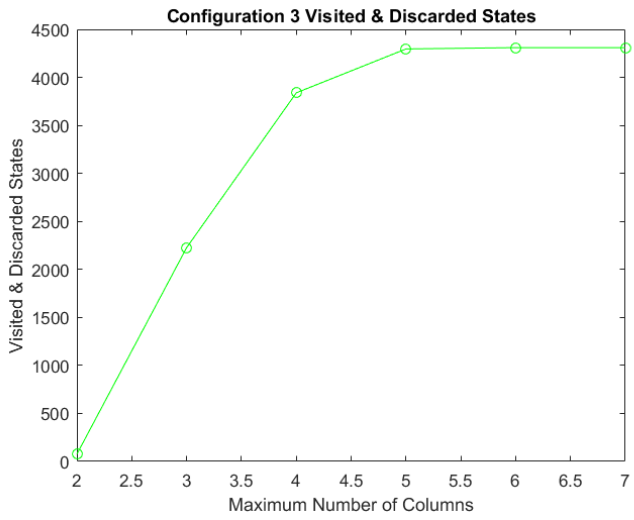


*Figure 20: Configuration 3: Visited & discarded states*

## 5.4. Configuration 4

INITIAL STATE:

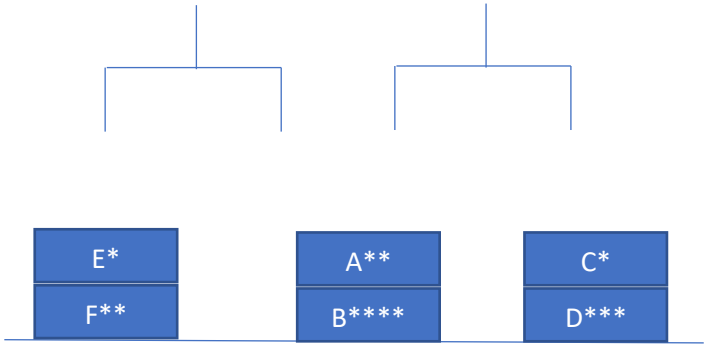


*Figure 21: Configuration 4: Initial state*

GOAL STATE

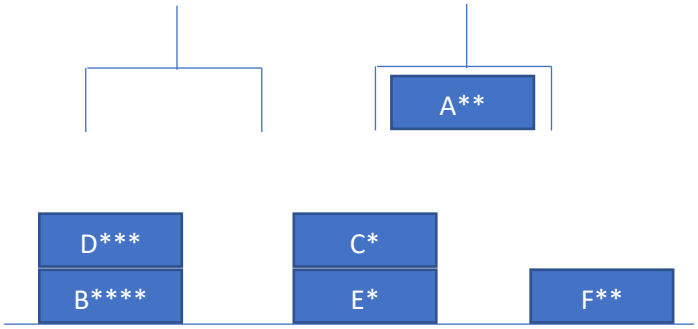


*Figure 22: Configuration 4: Goal state*

This problem has been solved under 6 different considerations: Maximum Columns (MCol) = 2,3,4,5,6 & 7.

Find in the *Figure 23* the number of steps required to solve the problem as a function of the maximum number of columns allowed. Note that for MCol = 2 the problem is unsolvable. Moreover, for this configuration as in the previous one, the minimum plan length is reached for MCol >= 4.

*Figure 24* and *Figure 25* show the number of visited states and total candidate states to find the plan as a function of the maximum number of columns allowed. For MCol = 2 the number of visited states is 0 since the number of columns of the final state is greater than the maximum number of columns allowed. For MCol >=5 the number of visited & candidate states reaches stability. For MCol=3 we observe that the number of visited & candidate states is smaller (which is to be expected since there are less possible configurations) but that the algorithm needs more steps to find a successful plan. Therefore, for this MCol value, the density of paths between states in the state space is smaller.

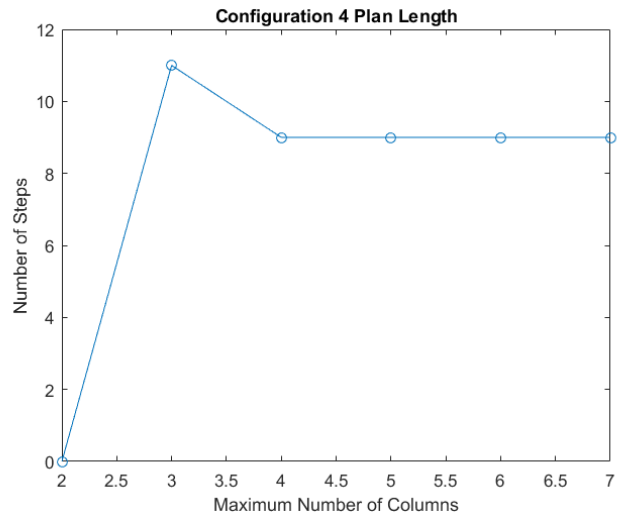


*Figure 23: Configuration 4: Plan length*

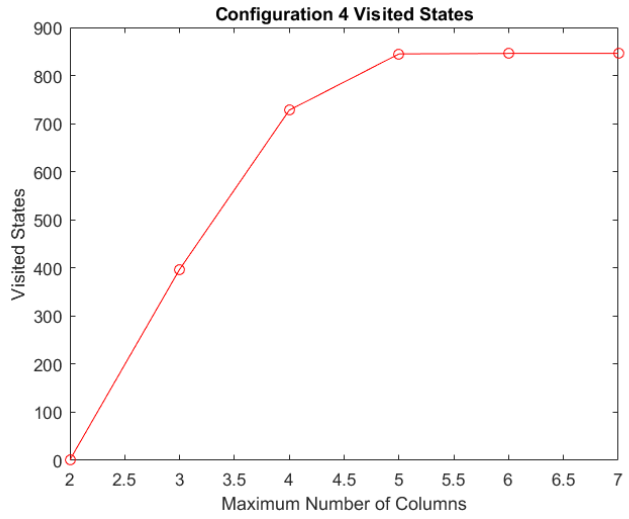


*Figure 24: Configuration 4: Visited states*

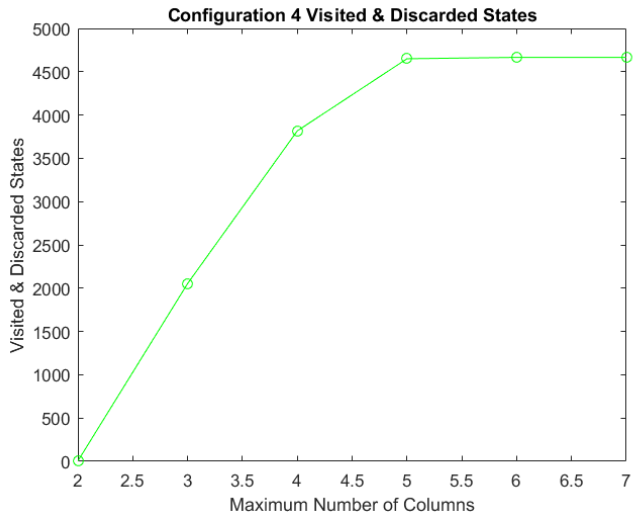


*Figure 25: Configuration 4: Visited and discarded states*

## 5.5. Configuration 5

INITIAL STATE:

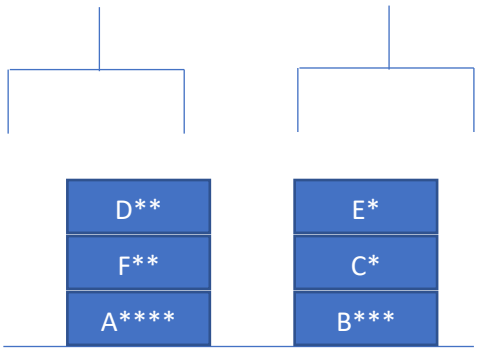


*Figure 26: Configuration 5: Initial state*

GOAL STATE

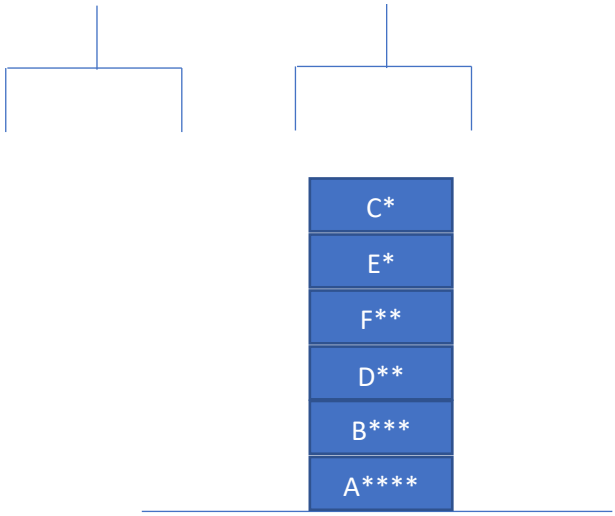


*Figure 27: Configuration 5: Goal state*

This problem has been solved under 6 different considerations: Maximum Columns (MCol) = 2,3,4,5,6 & 7.

Find in the *Figure 28* the number of steps required to solve the problem as a function of the maximum number of columns allowed. Note that for MCol = 2 the problem is unsolvable. Moreover, for this configuration and unlike the previous studied, the shortest plan length is achieved for MCol >= 5.

*Figure 29* shows the number of visited states to find the plan as a function of the maximum number of columns allowed. Note that for MCol >= 4 the number of visited states is stable but for MCol = 4 the algorithm moves through the state space with more difficulty as it takes more steps (longer plan) to reach the final state. This tendency is confirmed in *Figure 30*, as the number of total state candidates is also stable for MCol >= 4 which implies that the number of candidates per step is lower for MCol = 4.

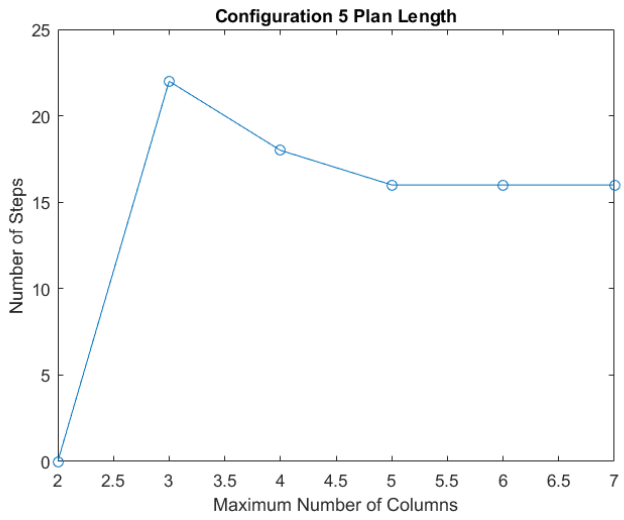


*Figure 28: Configuration 5: Plan length*

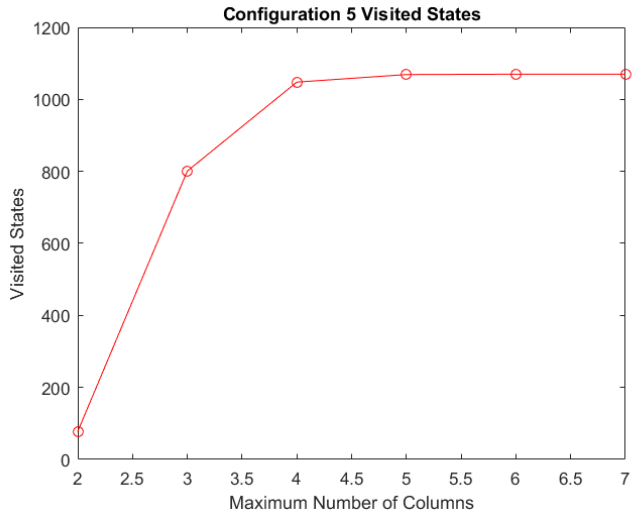


*Figure 29: Configuration 5: Visited states*

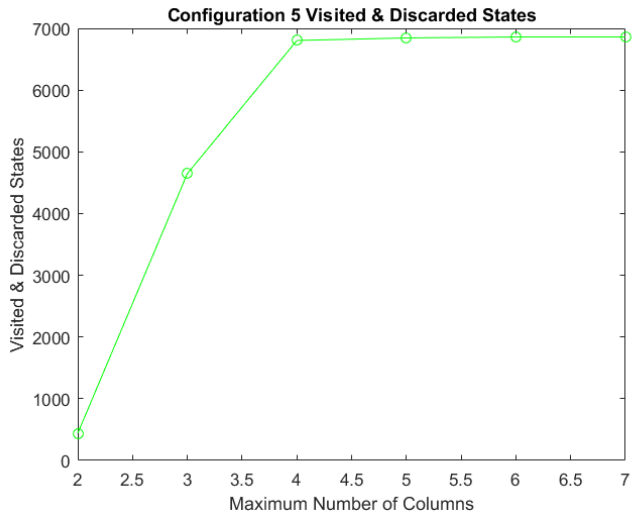


*Figure 30: Configuration 5: Visited and discarded states*

By comparing all configurations and the obtained results more conclusions are drawn:

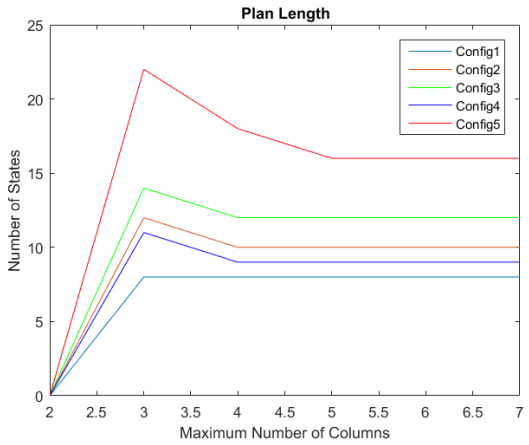


*Figure 31: All configurations: Plan length*

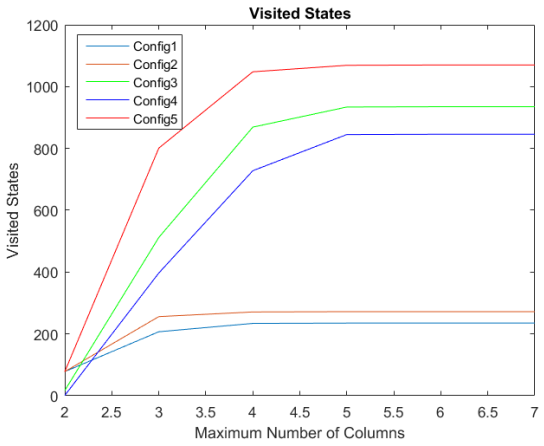


*Figure 32: All configurations: Visited states*

Firstly, a configuration with a higher number of Blocks will usually have a higher number of visited and candidate states (as there are higher number of configurations). Similarly, a higher MCol will always imply that the visited states increase (as at least all states visited for a lower MCol will also be visited by the algorithm when using a higher MCol.

Furthermore, it is also clear that to make sure to find the optimal path for any configuration and value of MCol, MCol should be greater or equal to the number of blocks of the configuration. This last statement might seem contradictory with the fact that the NLP with regression always find the optimal plan, but it isn't. For each MCol value the algorithm will find the optimal (shortest) plan, but such plan will be longer for smaller MCol since the state space of the problem is different.

# 6. Program execution

The algorithm has to be executed through the script *Main.m* located in the working directory.

It loads the input data from the file *test.txt*. This file contains 5 different *InitialState-GoalState* configurations and each of them repeated 6 times with different maximum number of columns, from 2 to 7.

The *test.txt* file that has been used for this document can be used as a template if there is the wish to substitute any of the existing configurations for another one. If done, it is important to substitute each of the 6 copies of the old configuration by the new one.

During the program execution it will generate 30 output files with the name *outputConfigXMaxColumnsY.txt*, being X one of the 5 configurations and Y one of the 6 copies of it.

These output files contain the number of steps that has the plan (or UNSOLVABLE PROBLEM if there is no plan to reach the goal state from the initial stat), the number of visited states, the plan (if there is one), and each visited state and the reason it has been discarded.

Another way to try a new configuration is to do it from scratch in another *file.txt*. Use the function NLPRegression(inputfile, outputfile, readlines) specifying the input file, the output file where the results will be saved and the line of the input file at which the parsing has to start reading.

*IMPORTANT: The input and output files must be read using MATLAB, it has been seen that with notepad can be opened but are unreadable.*