Universitat Rovira i Virgili

Planning & Approximate Reasoning

Practical Exercise 1: Planner

Blocks world with 2 arms

Authors: Emer Rodriguez Formisano, Jorge Alexander
Supervisor: Dr Antonio Moreno, Dr Aïda Valls

Date: 1st November 2017

A solution to a variant of the classic blocks world problem is implemented using the Matlab programming language. The planner algorithm is a non-linear planner with goal regression. The performance is measured, analysed and improvements suggested.

Table of Contents

Introduction

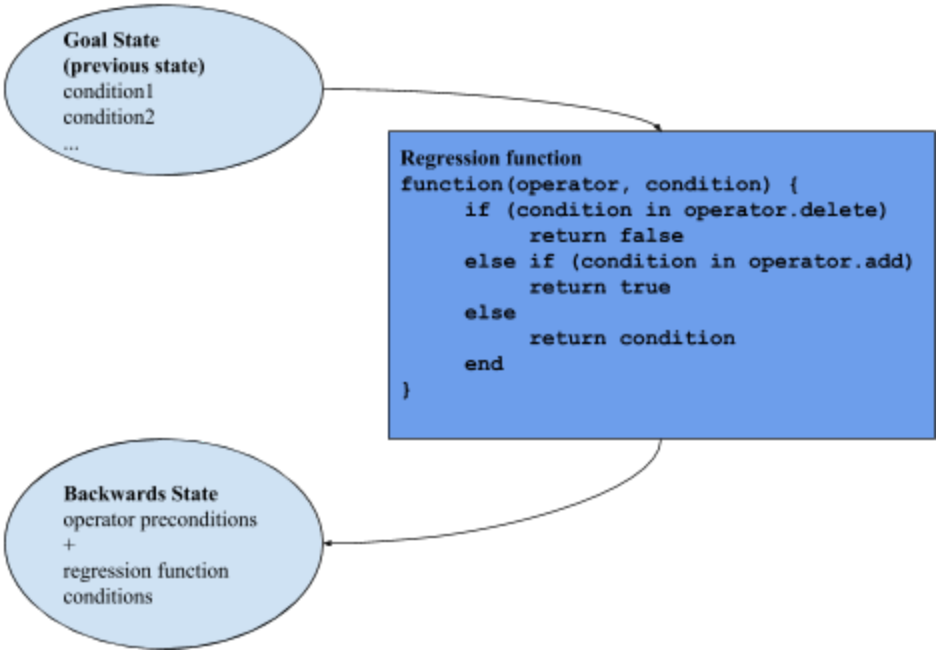
The blocks world problem is one of the most famous domains in artificial intelligence. The problem domain consists of three elements; blocks (objects) which can be operated on, arms (action doers) which can be used to operate on the blocks, and columns (positions) where the blocks can be positioned. The goal of the problem is to operate on the blocks with the arms in order to reach a certain domain configuration, known as a state. These three elements can have different rules and properties depending on the particular variant of the problem.

The problem domain discussed in this report consists of: blocks which have a weight property and that cannot support a block with a weight heavier than themselves; two arms with different weight holding capacities; a restricted integer number of columns where the blocks can be placed. The goal of the problem is to build a plan, which consists of an ordered list of operators chosen from a predefined set, that when applied in sequence on the initial state, modifies it so that a predefined and desired final state is achieved.

An operator consists of pre, addition and delete conditions. The pre-conditions are the conditions that the state must have for an operator to be applied on it, the addition conditions are conditions that are added to the state when applied on by the operator, and the delete conditions are conditions that are removed from the state when applied on.

In order to build the plan, a planner is used that implements a goal regression technique. Goal regression involves beginning the plan at the final state instead of the initial state. This requires a regression function, whose responsibility is to return the conditions of the next backwards state from the goal, given an operator and a previous state's conditions. The regression function prevents an invalid operator from being used by checking if the previous state's conditions are included in the operator's delete conditions, if it is the case, the operator is not allowed. The regression function prevents conditions from being lost, i.e. not carried backwards, by returning any conditions that are in the previous state, but not in the operator's add or delete conditions. If the operator is valid, the resulting state from the operator's application on the previous state, consists of the operator's pre-conditions, plus any conditions returned by the regression function (see Fig.1). In this report the goal-regression is used on different parameter configurations.

Fig.1 - Goal regression technique.



Analysis and problem formalisation

Domain configuration

The configuration that is investigated involves:

- blocks ‘A’, ‘B’, ‘C’, ‘D’, ‘F’ with weights of 1, 2, 2, 3, and 1 (Kg) respectively;
- arms ‘L’ and ‘R’, where ‘L’ is considered a weak-arm and has a maximum weight capacity of 1Kg, and arm ‘R’ is able to operate on any block;
- a maximum number of used columns, n, equal to 3.

Predicates

A predicate consists of a name and possible arguments. A set of predicates with their arguments set at defined values, compose a state. In order to represent the states used in this problem, the following table shows the predicates that were used, with their corresponding arguments and real-world analogies. Arguments X and Y each represent a block, argument A represents an arm, and argument N represents the number of columns. Note that a predicate argument in itself is not a defined value, only when the argument is set at one of its possible values (e.g. X set to ‘B’) does it become a component of a unique representation of the state.

Table.1 - Predicate table

Predicate name	Predicate arguments	Predicate real-world analogy
ON-TABLE	X	Block X is on the table
ON	X, Y	Block X is on block Y
CLEAR	X	Block X has no blocks on top of it
EMPTY-ARM	A	Arm A is not holding anything
HOLDING	X, A	Arm A is holding block X
USED-COLS-NUM	N	N columns are currently occupied
HEAVIER	X, Y	Block X is the same weight or heavier than block Y
LIGHT-BLOCK	X	Block X is light (weight of 1kg)

Operators

Four operator types are defined; pickup, stack, unstack and leave. Their real world analogy and the justifications of their predicates, followed by a tabular predicate representation is described.

The pickup operator type represents an action of picking up a block with either the left or right arm, therefore two variations of the pickup operator are used, “PICK-UP-LEFT” and “PICK-UP-RIGHT”.
The block must be on the table for it to be picked up, the arm picking it up must be empty, and the block must have no objects on top of it. This leads to preconditions “ON-TABLE”, “EMPTY-ARM” and “CLEAR” respectively. Also, if it the block is being picked up by a weak-arm, it must be a light-block and so the “LIGHT-BLOCK” predicate is needed as a precondition too. When the pickup operator is applied on a block, the number of columns occupied decreases by one, and the arm that picked up the block will now be holding it, this is represented with the “USED-COLS-NUM” and “HOLDING” predicates. Once the operator is applied, the block will no longer be on the table and the arm that picked it up will no longer be empty, this is represented by the “ON-TABLE” and “EMPTY-ARM” predicates.

Table.2 - Pickup operator

Name	Pre-conditions	Add-conditions	Delete-conditions
PICK-UP-LEFT(X)	ON-TABLE(X) EMPTY-ARM(‘L’) CLEAR(X) LIGHT-BLOCK(X)	HOLDING(X, ‘L’) USED-COLS-NUM(N-1)	ON-TABLE(X) EMPTY-ARM(‘L)

PICK-UP-RIGHT(X)	ON-TABLE(X) EMPTY-ARM(‘R’) CLEAR(X)	HOLDING(X, ‘R’) USED-COLS-NUM(N-1)	ON-TABLE(X) EMPTY-ARM(‘R’)

The stack operator represents an action of placing a block, already held by an arm, on top of another block, that is not held by another arm. The block must be held already by the arm, “HOLDING(X, A)”, and the block that it is going to be placed on must have no blocks on top of it, “CLEAR(Y)”. Also, the block that it is being placed upon must be equal or heavier than it in weight, “HEAVIER(Y, X)”. When the operator is applied, the block is now on top of the block that it is placed upon, represented by predicate “ON(X,Y)” and the arm that placed the block will then be empty, “EMPTY-ARM(A)”. By using the stack operator, the arm used will no longer be holding the block, and therefore the predicate “HOLDING(X,A)” is removed, and the block it placed a block upon, will no longer be clear, therefore the “CLEAR(Y)” predicate is removed.

Table.3 - Stack operator

Name	Pre-conditions	Add-conditions	Delete-conditions
STACK(X,Y)	HOLDING(X, A) CLEAR(Y) HEAVIER(Y, X)	ON(X, Y) EMPTY-ARM(A)	HOLDING(X, A) CLEAR(Y)

The unstack operator does the opposite of the stack operator, rather than placing one block on top of another, it removes a block from on top of another. As this operation can be done with arms “L” and arm “R”, this operator is represented by “UNSTACK-LEFT” and “UNSTACK-RIGHT”. In order to unstack, the block we want to unstack must be on another and it must also be the top block on the stack, hence the predicates “ON(X,Y)” and “CLEAR(X)”. In order to unstack the block, the arm performing the operation must be empty, thus pre-condition “EMPTY-ARM(A)” is needed. Also, if the arm performing the operation is the weak-arm, the block must be a light-block, “LIGHT-BLOCK(X)”. The operation will produce a new state where the unstacked block is held by the arm, “HOLDING(X,A)” and the block that it was unstacked from has nothing on top of it, “CLEAR(Y)”. The unstack will mean that the unstacked block is no longer on the block it was stacked on, leading to condition “ON(X, Y)” being removed, and the arm that did the unstacking, will no longer be empty, so the “EMPTY-ARM(A)” predicate is removed.

Table.4 - Unstack operator

Name	Pre-conditions	Add-conditions	Delete-conditions
UNSTACK-LEFT(X,Y)	ON(X,Y) CLEAR(X) EMPTY-ARM(LEFT) LIGHT-BLOCK(X)	HOLDING(X, LEFT) CLEAR(Y)	ON(X, Y) EMPTY-ARM(LEFT)

UNSTACK-RIGHT(X,Y)	ON(X,Y) CLEAR(X) EMPTY-ARM(right)	HOLDING(X,right) CLEAR(Y)	ON(X,Y) EMPTY-ARM(right)

Finally, the leave operator is analogous to an arm leaving a block that it is holding in an empty column. In order to do this, it must be holding a block and the number of used columns must be less than the total columns available, therefore we add precondition predicates "HOLDING(X,A)" and "USED-COLS-NUM(N < 3). When the leave operator is applied, the block that it operates on will be left on the table, the arm will be empty, and the number of used columns will increase by one. Therefore we add "ON-TABLE(X)", "EMPTY-ARM(A)" and "USED-COLS-NUM(N + 1)". However, the predicate "HOLDING(X, A)" will need to be removed as the arm will no longer be holding the block.

Table.5 - Leave operator∫

LEAVE(X)	HOLDING(X, A) USED-COLS-NUM(N < 3)	ON-TABLE(X) EMPTY-ARM(A) USED-COLS-NUM(N+1)	HOLDING(X, A)

Special Cases

One possible issue with this setup is a situation where both arms are holding a block, and an inferred "UN-STACK" operator attempts to create a backwards state with one block on-top of the other. This can happen because although a block is being held by an arm, it still has the "CLEAR(X)" predicate associated with it, allowing for the "UN-STACK" operator to be used. In order to solve this, either the "CLEAR(X)" predicate can be removed when a block is picked up by an arm, or a rule which ignores the "UN-STACK" operator when attempting to create a state with the "ON(X,Y)", "HOLDING(X)" and "HOLDING(Y)" predicates. The latter strategy is implemented, explained in the intelligent strategies section.

Planning algorithm

As briefly mentioned in the introduction section, the main objective of the *non-linear planner with goal-regression* algorithm, is to find the smallest set of actions which when applied to an initial state would lead to a desired goal state. The algorithm combines two main components: a *breadth-first* search technique and a *regression* function, however a *depth-first* search method could also theoretically be used.

In summary, the algorithm starts with the final state as the root node of a tree and generates the children nodes, each a possible state previous to the parent state, it then selects the next child node to expand by traversing the tree using a *breadth-first* approach. The function "**`inferaction`**" generates the list of all feasible previous actions that, when applied to a previous state, would generate the parent state as a result. Given the predicates of the parent state and the list of feasible actions, all previous states can be generated using the regression function.

The planning algorithm can be represented with the following pseudocode:

```
SEARCH_TREE := SET-ROOT(FINAL_STATE)
DONE := FALSE
WHILE NOT(EMPTY(SEARCH_TREE) AND NOT(DONE)DO
    N := GET-PENDING-STATE-TO-BE-EXPLORED(SEARCH_TREE)
    FEASIBLE_ACTIONS := INFER-ALL-FEASIBLE-ACTIONS-TO-THE-STATE(N)
    FOR EACH ACTION IN FEASIBLE_ACTIONS DO
        NEXT_STATE = REGRESSION-FUNCTION(N, FEASIBLE_ACTION)
        IF NEXT_STATE IS INITIAL_STATE THEN
            DONE := TRUE
            PLAN := PATH-TO-THE-STATE(NEXT_STATE, SEARCH_TREE)
        ELSE IF IS-PREVIOUSLY-VISITED-NODE(NEXT_STATE)
            CONTINUE
        ELSE
            ADD-PREVIOUSLY-VISITED-NODE(NEXT_STATE)
            ADD-NODE-SEARCH-TREE(NEXT_STATE, SEARCH_TREE)
        ENDIF
    ENDFOR
ENDWHILE
IF DONE THEN
    RETURN PLAN
ELSE
    NO PLAN WAS FOUND TO GO FROM INITIAL STATE TO FINAL STATE
ENDIF
```

The algorithm does an exhaustive search. Thus, the end result will be the optimal plan if it exists. As noted above, a key part of the effectiveness of the result is the "**inferaction"** function, it can be seen as an enhanced version of the regression function, which not only checks if a predicate of the parent state is compatible or not with a given operator, but it also checks any inconsistencies that may appear between an action's preconditions and the parent state.

Intelligent strategies

The mentioned incompatibilities can be resolved by using domain knowledge and the context or state information in which the operator is applied. The knowledge of how to resolve the inconsistencies is hard coded as a set of rules:

- If block X is **ON-TABLE** and no other blocks are **ON** top of X and an **EMPTY-ARM** is available (considering the weight of X too), then the most plausible action is **LEAVE** X on table.
- If block X is **ON** block Y and the Y block is **HEAVIER** than X and an **EMPTY-ARM** is available (considering the weight of X too) and no other blocks are **ON** X, then the most plausible action is **STACK** X on Y.

- If block Y is **CLEAR** and an arm is **HOLDING** block X and block Y is **HEAVIER** than X then, the most plausible action is **UNSTACK-LEFT** or **UNSTACK-RIGHT,** depending on the holding arm.
  Special case: Note that CLEAR Y could be present in the state when Y is being held by an arm. If this is the case, the rule is skipped. Otherwise, it could try to UNSTACK Y from X which would mean that a precedent state has ON(Y,X), in other words, two blocks stacked on one arm.
- An arm **HOLDING** block X and there are available columns **USED-COLS-NUM(N < MAX)**, the most plausible action is **PICK-UP-LEFT** or **PICK-UP-RIGHT**, depending on the arm being used.

Using this set of rules, the algorithm becomes efficient as it avoids generating unfeasible states. The algorithm also avoids considering the entire list of actions in each state. In addition, it also reduces the work of the regression function, as when an add predicate is passed to the regression function, it will always return TRUE.

Implementation design

Originally, an object oriented approach was considered. The idea was to have a Parser, Block, Predicate, Operator, State, Planner and PlanBuilder class, using a builder design pattern in order to build the plan. The advantages of this were the re-usability of the abstract classes, that could have led to an easier adaptation to different problems, as well as the separation of responsibility into clear entities. However, the disadvantages were that the size of the project grows because of the separation between classes, making it difficult to test and modify quickly, and harder to insert rules for specific cases. Also, two programming languages were considered, Matlab and Java. Matlab was chosen because it is known for its good vector computation, something that was considered as a possibility for a more advanced implementation. However, Matlab was not designed handle object-oriented programming well by default, so instead, a functional approach was chosen. An overview of the function sequence is shown in at the end of this section. The key functions are the `solver`, `inferaction`, `action` and `regression` ones, described here.

Solver function

The `solver` function is responsible for building the solver plan, which will contain the list of operators that lead to the final state from the initial state. It is an important part of the implementation as it has great control over the length of the execution of the program. Not only can it stop the execution if the max iterations are reached and no plan is found from initial and final state, but it also decides when the initial state is found from the final state. Another important role of the solver is to dictate the search of the states. It is implemented to do a breadth-first search, by pushing new states into a queue, and then selecting the oldest state in the queue that has not been operated on. However, it can be modified to do a depth-first search, by simply operating on the state as soon as it is generated, and if no more states are generated from it, the most recent state in the queue that has not been operated on instead. The breadth-first search was implemented first, as this was a project requirement, however it would have been interesting to be

able to compare, on each state creation, how different the created state was from the initial state and, depending on the difference, choose to do a breadth-first or depth-first search on each iteration.

Inferaction function

The `inferaction` function is crucial for applying the heuristics of the domain. The idea was to separate domain knowledge into the inferaction functions, such as `inferedArm` so that the problem could be applied to new domains, just by modifying them. However, in the final implementation, an `action` function was also needed, which included predicate domain knowledge in order to retrieve the preconditions, add and delete predicates of an operator.

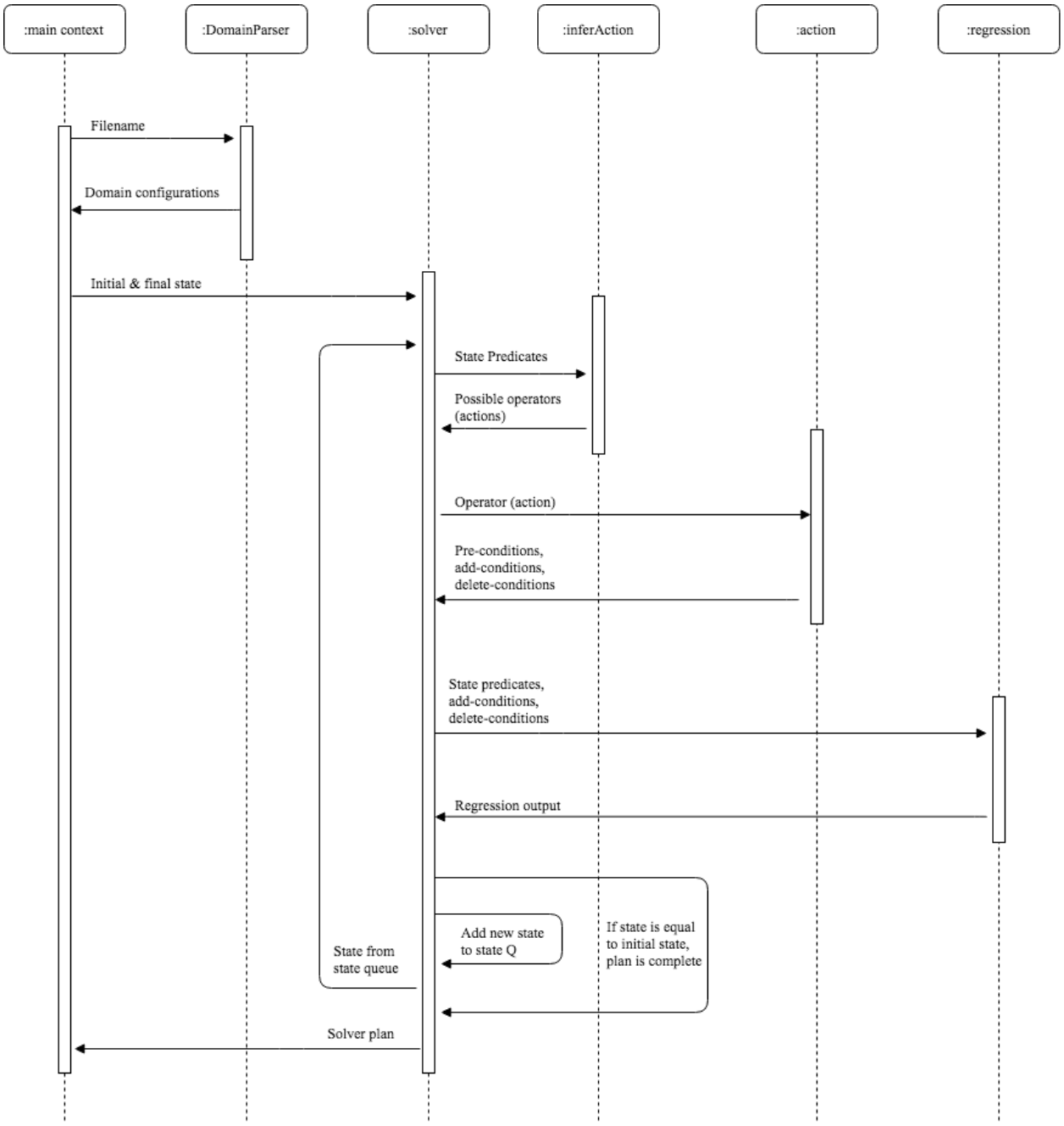
Action function

The action function fulfills the role of a lookup function. It allows the correct predicates of an operator to be returned for a given action description, state and block weight. This is crucial for the `solver` function to work correctly, as when iterating through the proposed actions to use, the solver needs to lookup the predicates for that action in order to apply the `regression` function. It was implemented as such to allow a dynamic action definition, where every possible action variation did not have to be defined. For example, instead of defining the “PICKUP-LEFT” action for every possible block combination, the `action` function builds the correct one dynamically, given an action description.

Regression function

Although small, the `regression` function is key to the goal-regression algorithm strategy. Given a state’s predicates, and an operator's add and delete predicates, it returns any conditions that need to be added to the previous state but won’t be added from the operator's preconditions. It also prevents an invalid state, where the operator would delete the state’s predicates, from being created. This is implemented simply by checking if the predicate provided is a member of the operator’s add or delete predicates; if it’s part of the delete, it is invalid; if it is not part of the delete or add predicates, it is returned and is required to be added to the state being created. In order to implement this, the Matlab `ismember` function was used. What is interesting about this function, is that it can operate on two sets of arrays at once, which allows the possibility for comparing several predicates at once in a future implementation.

Fig.2 - Implementation sequence diagram



Results & Discussion

The implementation was tested with five different cases; problem 0, 1, 2, and benchmark 1 and 2. The problem cases are for testing the algorithm functionality, the benchmark cases are for illustrating the performance in different situations.

The program output has the following important fields:

- `Iteration`, which shows the current iteration that the program is on, the goal state being iteration 1;
- `Current State`, detailing the predicates of the current state being operated on;
- `Proposed Actions`, showing the proposed operators that can be used to operate on the current state;
- `Analyzing`, which displays the current action that is being considered;
- `Next State` which shows the state that is generated by applying the action that is currently in analysis;
- `Previous Actions`, which shows the previous actions used to arrive at the current state;
- `Number of operators`, shown at the end of an execution to show the number of operators needed to reach the goal state;
- `Number of states generated`, shown at the end of an execution to show the number of states generated to reach the goal state;
- `Plan`, shown at the end of an execution to show the plan of operators needed to reach the goal state from the initial state;

Problem 0

The configuration of Problem 0 can be found in "testing0.txt". The full ouput of the program is found in Appendix 1.

This is the simplest problem considered during the implementation of the algorithm. It is easy to understand as it has a short solution. The simplicity allows the programmer to focus on the code rather than the complexity of the problem.

Fig.3 - *Problem 0* configuration

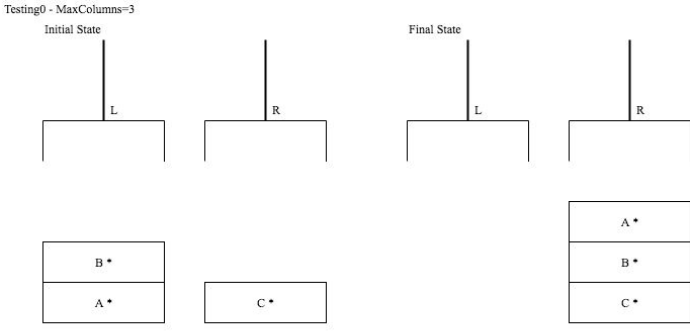


Fig.3, shows a diagram of the initial and final state of Problem 0.
The first iteration proceeds as follows:

```
Iteration: 1
Current State: CLEAR(A),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(B,C),ON-TABLE(C)
  Proposed Actions: STACK(A,B)
    Analyzing: STACK(A,B)
      Next State: CLEAR(A),CLEAR(B),EMPTY-ARM(R),HOLDING(A,L),ON(B,C),ON-TABLE(C)
...
```

Every iteration belongs to the expansion of a node. Given the current state, which in `Iteration: 1` is the goal state, the algorithm proposes actions using the "`inferaction`", its output marked in blue. As the proposed action "STACK(A,B)" is feasible, it is considered and analysed. The actions that lead to the final plan are depicted in red.

Note how Iteration 4 explored a branch irrelevant for obtaining the final solution, none of the analysed actions were included in the final plan:

```
...
Iteration: 4
Previous Actions: STACK(B,C),STACK(A,B)
Current State: CLEAR(A),CLEAR(B),CLEAR(C),HOLDING(A,L),HOLDING(B,R),ON-TABLE(C)
  Proposed Actions: PICK-UP-LEFT(A),PICK-UP-RIGHT(B),UNSTACK-RIGHT(B,C)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(L),HOLDING(B,R),ON-TABLE(A),ON-TABLE(C)
    Analyzing: PICK-UP-RIGHT(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(R),HOLDING(A,L),ON-TABLE(B),ON-TABLE(C)
    Analyzing: UNSTACK-RIGHT(B,C)
      Next State: CLEAR(A),CLEAR(B),EMPTY-ARM(R),HOLDING(A,L),ON(B,C),ON-TABLE(C)
        State already visited
...
```

The analysis of the proposed action, involves calling the regression function, which creates the children nodes seen on the `Next State` line, for future exploration. The children are then added to the search tree while others marked as "State already visited" (shown above in orange), can be safely ignored.

Below the fifth and final iteration is shown.

```
...
Iteration: 5
Previous Actions: STACK(B,C),PICK-UP-LEFT(A),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(R),HOLDING(B,L),ON-TABLE(A),ON-TABLE(C)
```

```
  Proposed Actions:
LEAVE(A),LEAVE(C),PICK-UP-LEFT(B),UNSTACK-LEFT(B,A),UNSTACK-LEFT(B,C)
    Analyzing: LEAVE(A)
      Next State: CLEAR(A),CLEAR(B),CLEAR(C),HOLDING(A,R),HOLDING(B,L),ON-TABLE(C)
    Analyzing: LEAVE(C)
      Next State: CLEAR(A),CLEAR(B),CLEAR(C),HOLDING(B,L),HOLDING(C,R),ON-TABLE(A)
    Analyzing: PICK-UP-LEFT(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(L),EMPTY-ARM(R),ON-TABLE(A),ON-TABLE(B),ON-TABL
E(C)
    Analyzing: UNSTACK-LEFT(B,A)
      Next State:
CLEAR(B),CLEAR(C),EMPTY-ARM(L),EMPTY-ARM(R),ON(B,A),ON-TABLE(A),ON-TABLE(C)
DONE!
Number of operators: 4
Number of states generated: 13
Plan: UNSTACK-LEFT(B,A),STACK(B,C),PICK-UP-LEFT(A),STACK(A,B)
```

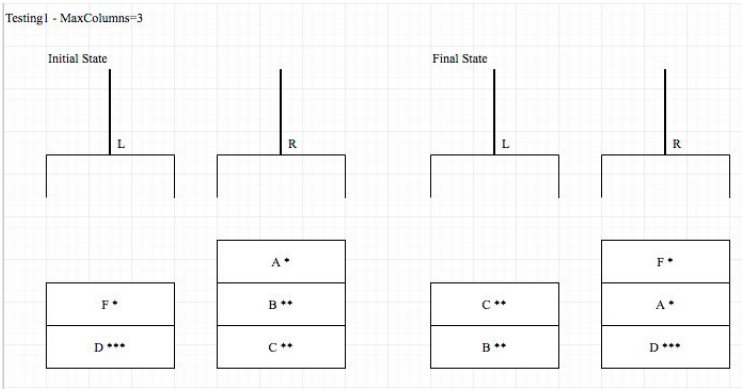
Once the initial state is found, the algorithm stops and returns the number of operators, the number of states generated and the **optimal plan** (coloured in green). It shows that the plan was built with 4 operators, generating 13 states, and in 5 iterations. After analysing the results it is seen that iteration 4 explored a branch irrelevant for the final plan. Had the algorithm not explored this branch, it would have had to analyse three actions less, and would be more efficient. However, it was not known that this branch was not necessary for the plan, until it was explored.

Problem 1

The configuration of Problem 1 can be found in "testing1.txt". The full ouput of the program is found in Appendix 2.

Fig.4 shows the initial and final states of *Problem1*. It is more complex than *Problem0* as it has more blocks to rearrange in the same amount of space, and the blocks now have more than one weight, which means that the algorithm should not use the left arm (the weak arm) to lift any blocks heavier than 1Kg.

Fig.4 - *Problem1* initial and final configuration.



The final iteration of the algorithm to show *Problem1* is shown below:

```
Iteration: 100
Previous Actions:
UNSTACK-LEFT(F,D),STACK(A,D),STACK(F,A),UNSTACK-RIGHT(B,C),LEAVE(B),PICK-UP-RIGHT(C)
,STACK(C,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(L),HOLDING(A,R),ON(B,C),ON(F,D),ON-TABLE(C),ON-
TABLE(D)
  Proposed Actions:
PICK-UP-RIGHT(A),STACK(F,D),UNSTACK-RIGHT(A,B),UNSTACK-RIGHT(A,F)
    Analyzing: PICK-UP-RIGHT(A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(B,C),ON(F,D),ON-TABLE(A),ON-
TABLE(C),ON-TABLE(D)
        State already visited
    Analyzing: STACK(F,D)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(D),CLEAR(F),HOLDING(A,R),HOLDING(F,L),ON(B,C),ON-TABLE(C),ON
-TABLE(D)
        State already visited
    Analyzing: UNSTACK-RIGHT(A,B)
      Next State:
CLEAR(A),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(B,C),ON(F,D),ON-TABLE(C),ON-T
ABLE(D)
DONE!
Number of operators: 8
Number of states generated: 350
Plan:
UNSTACK-RIGHT(A,B),UNSTACK-LEFT(F,D),STACK(A,D),STACK(F,A),UNSTACK-RIGHT(B,C),LEAVE(
B),PICK-UP-RIGHT(C),STACK(C,B)
```

After exploring 350 states, over the course of 100 iterations, the algorithm finds the optimal plan for solving Problem 1. The plan is made of the following 8 operators:
**UNSTACK-RIGHT(A,B),UNSTACK-LEFT(F,D),STACK(A,D),STACK(F,A),UNSTACK-RIG
HT(B,C),LEAVE(B),PICK-UP-RIGHT(C),STACK(C,B)**

Compared to *Problem0*, which took 5 iterations and generated 13 states, *Problem1* is more complex. Although it took 20 times as many iterations to reach a solution, this is not representative of it being 20 times as complex. This is because on average more states were generated per iteration in *Problem1*:
$\frac{350}{100} = 3.5$ *states per iteration*
compared to *Problem0*:
$\frac{13}{5} = 2.6$ *states per iteration*
A rough measure is to compare the number of states generated (350 vs 13), which approximates *Problem1* as being roughly 27 times as complex. The higher number of states generated per iteration may be explained due to there being more actions to consider, because of there being more blocks to operate on in *Problem1* than *Problem0* (see the conclusion for more detail).

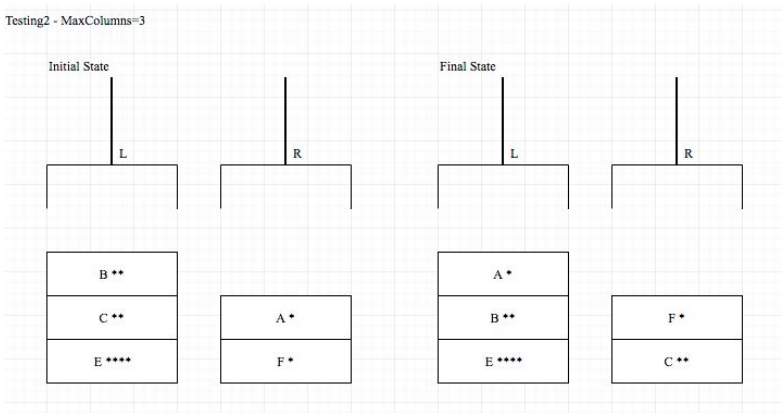
Note although the console output in the Appendix 2 is a bit dense for human consumption, it shows details about the options considered in each iteration. The output has been truncated in order to display only the iterations relevant to the final solution.

Problem 2

The results of Problem 2 can be found in "testing2.txt". The full output of the program is found in Appendix 3.

The initial configuration of *Problem2* is shown in Fig 5.2. It has the same number of blocks as *Problem1*, with a similar block positioning and a similar block weight distribution, however instead of a block D with a weight of 3Kg, it has a block E with a weight of 4Kg. This difference in weight should not affect the arm weight condition, so it should be interesting to see if *Problem2* has a different complexity to *Problem1*.

Fig.5 - *Problem2* configuration



After exploring 735 states over 221 iterations, it was found that the algorithm finds the optimal plan for solving *Problem2*. The plan is made of the following 12 operators:

UNSTACK-RIGHT(B,C),UNSTACK-LEFT(A,F),LEAVE(B),PICK-UP-RIGHT(F),STACK(F,B),UNSTACK-RIGHT(C,E),LEAVE(C),UNSTACK-RIGHT(F,B),STACK(F,C),PICK-UP-RIGHT(B),STACK(B,E),STACK(A,B)

Compared to the *Problem1*, *Problem2* has a longer plan by 4 operators, and more than double the states and iterations.

The state to iteration ratio of *Problem2* (3.33), is not that dissimilar to *Problem1* (3.5). This could be because the number of combinations to consider between each iteration stays fairly constant between both problems, because they have a similar number of operators that can be considered on each iteration, and therefore a similar number of states generated per iteration.

However, the state to operator ratio for *Problem2*

$\frac{735}{12}$ = 61.25 *states per operator*

vs Problem1:

$\frac{350}{8}$ = 43.75 *states per operator*

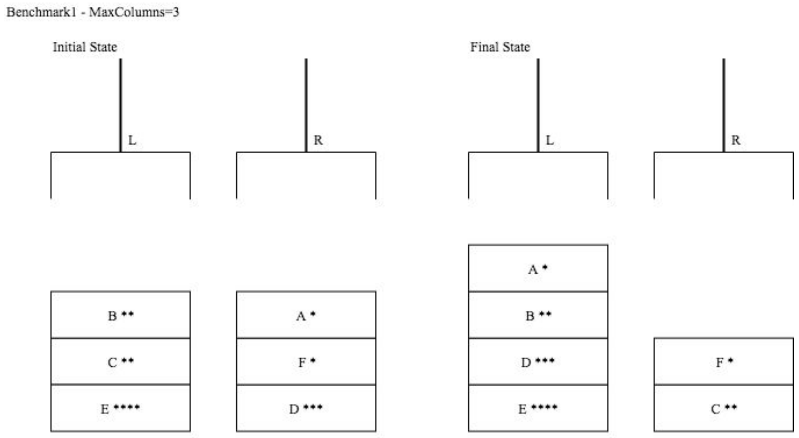
shows that more states had to be explored on average, for each operator that made the final plan in *Problem2* than in *Problem1*. Why is this the case? *Benchmark1* and *Benchmark2* test the planner with more states and operators to see if conclusions can be drawn.

Benchmark 1

The results of Benchmark 1 can be found in file "benchmark1.txt".

Benchmark 1 is the first case proposed for evaluating the impact of increasing the number of available columns versus the resources used (time, number of operators and generated states). The definition of this problem is slightly harder than *Problem 2* as it just adds an extra block D***. The figure below shows the initial and final states.

Fig.6 - *Benchmark1* configuration



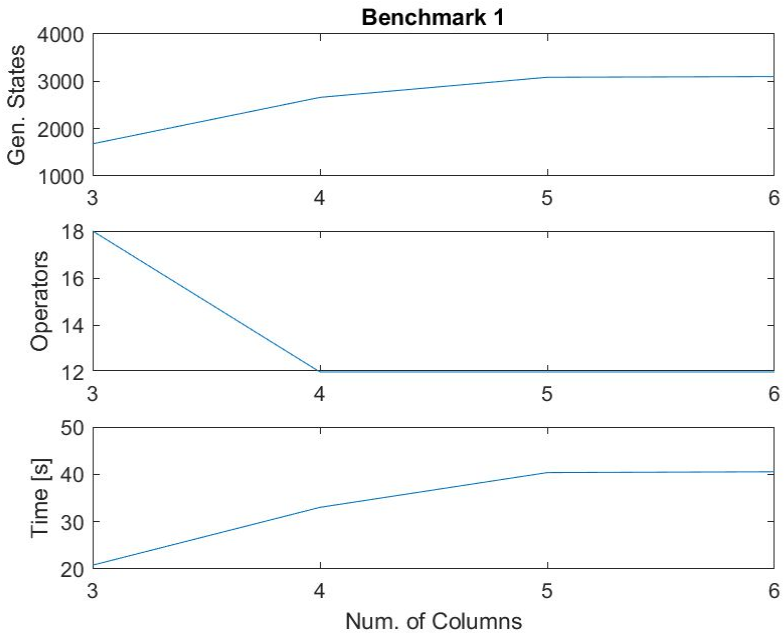
After exploring 1680 states, the algorithm finds the optimal plan for solving *Benchmark 1*. The plan is made of the following 18 operators:

UNSTACK-RIGHT(B,C),LEAVE(B),UNSTACK-RIGHT(C,E),UNSTACK-LEFT(A,F),STACK(C,B),UNSTACK-RIGHT(F,D),STACK(F,C),PICK-UP-RIGHT(D),STACK(D,E),UNSTACK-RIGHT(F,C),STACK(F,D),UNSTACK-RIGHT(C,B),LEAVE(C),UNSTACK-RIGHT(F,D),STACK(F,C),PICK-UP-RIGHT(B),STACK(B,D),STACK(A,B)

The following figure shows that increasing the number of columns to any number higher than 3, decreases the number of operators to 12. However, the number of generated nodes is significantly increased and stabilised at around 3000. Note how the time is proportional to the number of generated states. It is also

stands out how slow the implementation is as it takes over 20 seconds to solve the problem. Full details of the execution can be found at “benchmark1.log” file.

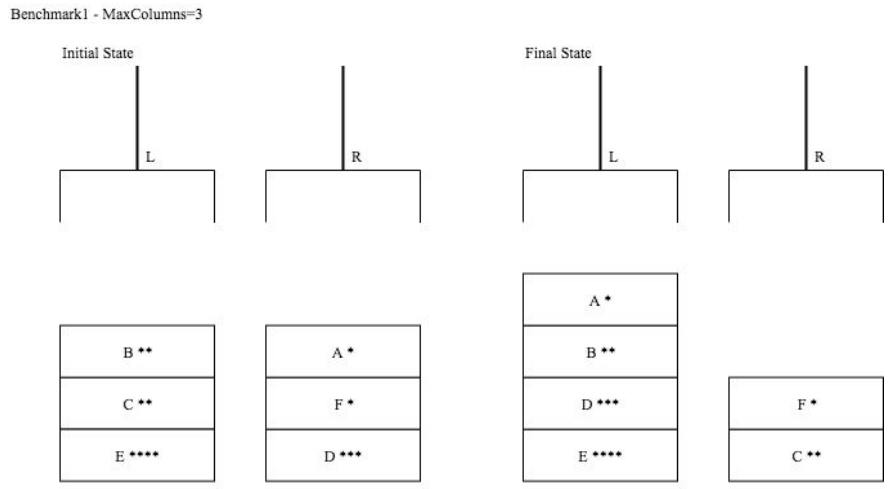
Fig.7 - *Benchmark1* results



Benchmark 2

Taking *Benchmark 1* as a reference, the complexity of the case can be easily increased by adding an extra block G*** to the scene. The figure below shows the initial and final state of this case.

Fig.8 - *Benchmark2* configuration

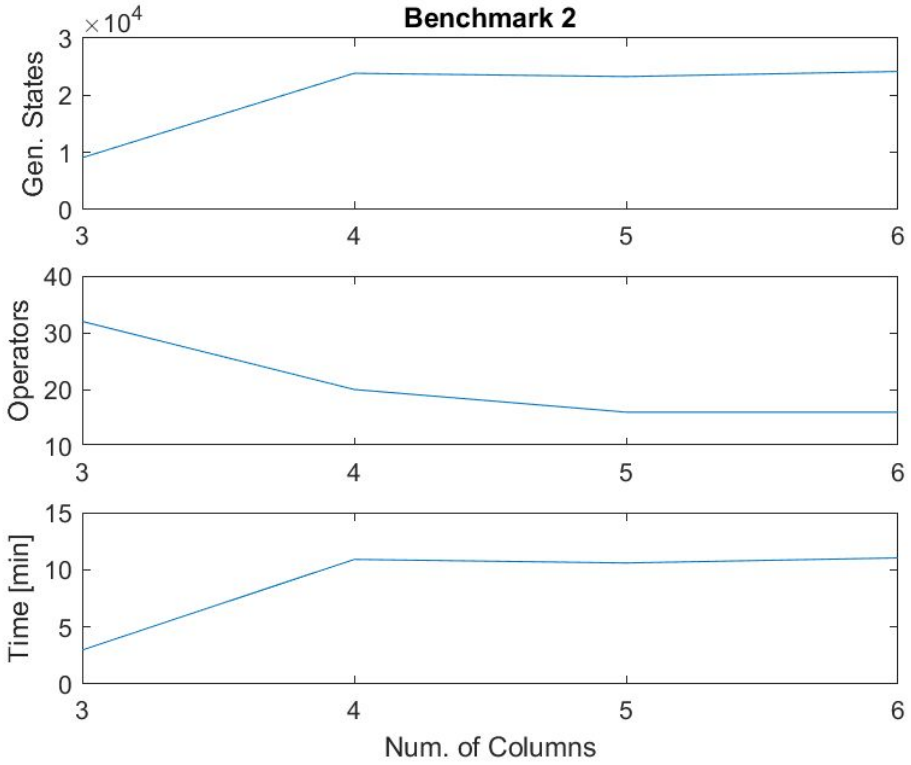


After exploring 9078 states, the algorithms finds the optimal plan for solving Benchmark 2. The plan is made of the following 32 operators:

UNSTACK-LEFT(A,F),UNSTACK-RIGHT(F,D),STACK(F,B),UNSTACK-RIGHT(D,G),LEAVE(D),UNSTACK-RIGHT(F,B),STACK(F,G),UNSTACK-RIGHT(B,C),STACK(B,D),UNSTACK-RIGHT(C,E),STACK(C,B),PICK-UP-RIGHT(E),LEAVE(A),UNSTACK-LEFT(F,G),STACK(E,G),STACK(F,E),UNSTACK-RIGHT(C,B),PICK-UP-LEFT(A),LEAVE(C),UNSTACK-RIGHT(B,D),STACK(B,C),PICK-UP-RIGHT(D),LEAVE(A),UNSTACK-LEFT(F,E),STACK(F,A),STACK(D,E),UNSTACK-RIGHT(B,C),UNSTACK-LEFT(F,A),STACK(F,C),STACK(B,D),PICK-UP-LEFT(A),STACK(A,B)

Similar to Benchmark 1, the number of generated nodes increase and stabilise when the maximum available columns is increased. However the number of generated nodes is much larger (around 24000) together with the time required to compute (around 11 minutes). The case is complex enough such that the number of operators required to solve the problem decreases when the number of available columns increases. It goes from 32 to 16 operators.

Fig.9 - *Benchmark2* results



At this point we can also analise the impact of adding a block. As mentioned, benchmark 1 and 2 cases are problem 2 modified by adding 1 and 2 blocks respectively. Although, the configuration is not exactly the same in each case, adding an extra block increases the complexity of the solution significantly. Table 6 illustrates the metrics of the resources used for each case, having the number of available columns fixed to 3.

Table.6 - *Problem2, Benchmark1 & 2,* comparison table.

Case *MaxColumns = 3*	**Iterations**	**Number of Generated states**	**Number of operators**	**Elapsed Time [s]**
Problem 2	221	735	12	7.8
Benchmark 1 (Prob. 2 + 1 block)	501	1680	18	22.4
Benchmark 2 (Prob. 2 + 2 blocks)	2702	9078	32	178.8

Table.7 - All problems results comparison table.

Case *MaxColumns = 3*	**Number of blocks**	**Generated states**	**Iterations**	**States per iteration**	**Number of operators in plan**	**States per operator**	**Elapsed Time [s]**
Problem 0	3	13	5	2.60	4	3.25	0.15
Problem 1	5	350	100	3.50	8	43.75	3.83
Problem 2	5	735	221	3.33	12	61.25	7.87
Benchmark 1 (Prob. 2 + 1 block)	6	1680	501	3.35	18	93.33	22.47
Benchmark 2 (Prob. 2 + 2 blocks)	7	9078	2702	3.36	32	283.70	178.84

Conclusion

During the analysis phase, 6 operators were defined (Pick-Up-Left, Pick-Up-Right, Stack, Unstack-Left, Unstack-Right and Leave). This decision helped to reduce the number of operators considered in each state making the algorithm more efficient.

The implementation done can be seen as a prototype. The aim was to understand the algorithm and implement its mechanics. However, the code is not production ready and it requires refactoring in order to increase performance. One of the main bottlenecks is that it doesn't take advantage of the vectorization capabilities of Matlab. Many of the vectors change size over loop iterations and preallocation should be considered. Although it is good for understanding the process, the verbosity of the output slowed the execution. Better performing data structures could be considered for representing predicates or states.

For example, the state could be represented as a single matrix, that has one dimension for all the possible predicate names, another for all the possible arguments that a predicate can take, and a last dimension for all the possible values that the argument of the predicate can be. Then, operator matrices which are represented in a similar way, can be added or subtracted to the state matrix (according to a vectorised regression function) in order to generate another state. It would be interesting to implement this design in Matlab, and compare its speed to the current implementation.

All of the proposed cases are successfully solved with a reasonable amount of generated states. Problem 0 helped during the early developing stage while Benchmark 2 clearly highlighted the limitations of the prototype with the performance issues by requiring up to 3 minutes of computational time. Problem 1 and 2 were solved by using 8 and 12 operations respectively with the cost of discovering 735 and 1680 states respectively. The plots and the tables discussed in the results section showed a couple of patterns. First, the number of resources required increases significantly with the number of blocks defined. Secondly, the

constraint of available columns is only relevant when solving high complex cases as the number of operators decreases.

Also, it is interesting that the states per iteration remains fairly stable throughout the problems, even though the domain size increases (with the exception of *Problem0* that only has 5 iterations and so not enough to converge to a significant average). This is because the number of states that can be generated per iteration is limited by the number of operators that has been defined and remains constant through the problems. It would be interesting to measure how the states per iteration changes as the number of operators available to the problem is changed.

It is also interesting to note that the states generated per operator that is added to the final plan, increases as the domain size increases. This means that as the problem becomes more complex, more states are required to be generated in order to find an operator that will be part of the final plan. This makes sense, and could be used as a measure of the complexity of the domain. It would be interesting to compare how varying the column size, block weights, number of blocks and number of arms affects this measure.

Instructions to execute the code

Once the repository is cloned or the zip file extracted, you can run the script from the terminal with the following command.

```
matlab main
```

From the Matlab interface, just place the working directory to the repository or to the extracted folder and open the main.m script. Click on the RUN button.

The folder contains the following files:

- **testingX**: cases used for testing the algorithm
- **benchmarkX**: cases used for performance analysis
- **main.m** and **loadconstant.m**: scripts containing the code
- **REDME.md**: file containing the instructions

For each file type there are:

- **{filetype}.txt**: input file with the definition of the world (MaxColumns, Blocks, InitialState and GoalState)
- **output_{filetype}.txt**: output file showing number of operators, number of states generated, the plan and the cancelled exploratory states
- **output_{filetype}.log**: verbose console output with all the steps evaluated in each iteration
- **{BenchmarkX}.png**: plots generated in the performance analysis

Bibliography

1. En.wikipedia.org. (2017). Blocks world. [online] Available at: https://en.wikipedia.org/wiki/Blocks_world [Accessed 1 Nov. 2017].
2. Russell, Stuart J., and Peter Norvig.*Artificial Intelligence: a Modern Approach*. Pearson, 2016.
3. Moreno, A. (2017). *MODULO3 PARTE4*. [online] YouTube. Available at: https://www.youtube.com/watch?v=yOunULLOxu0 [Accessed 1 Nov. 2017].
4. Moreno, A. (2017). *MODULO3 PARTE5*. [online] YouTube. Available at: https://www.youtube.com/watch?v=KsedWHIPHu8 [Accessed 1 Nov. 2017].

Appendix

1 - Problem 0 output

```
Iteration: 1
Current State: CLEAR(A),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(B,C),ON-TABLE(C)
  Proposed Actions: STACK(A,B)
    Analyzing: STACK(A,B)
      Next State: CLEAR(A),CLEAR(B),EMPTY-ARM(R),HOLDING(A,L),ON(B,C),ON-TABLE(C)
Iteration: 2
Previous Actions: STACK(A,B)
Current State: CLEAR(A),CLEAR(B),EMPTY-ARM(R),HOLDING(A,L),ON(B,C),ON-TABLE(C)
  Proposed Actions: PICK-UP-LEFT(A),STACK(B,C),UNSTACK-LEFT(A,B)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(B),EMPTY-ARM(L),EMPTY-ARM(R),ON(B,C),ON-TABLE(A),ON-TABLE(C)
    Analyzing: STACK(B,C)
      Next State: CLEAR(A),CLEAR(B),CLEAR(C),HOLDING(A,L),HOLDING(B,R),ON-TABLE(C)
    Analyzing: UNSTACK-LEFT(A,B)
      Next State: CLEAR(A),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(B,C),ON-TABLE(C)
        State already visited
Iteration: 3
Previous Actions: PICK-UP-LEFT(A),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),EMPTY-ARM(L),EMPTY-ARM(R),ON(B,C),ON-TABLE(A),ON-TABLE(C)
  Proposed Actions: LEAVE(A),STACK(B,C)
    Analyzing: LEAVE(A)
      Next State: CLEAR(A),CLEAR(B),EMPTY-ARM(R),HOLDING(A,L),ON(B,C),ON-TABLE(C)
        State already visited
    Analyzing: STACK(B,C)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(R),HOLDING(B,L),ON-TABLE(A),ON-TABLE(C)
Iteration: 4
Previous Actions: STACK(B,C),STACK(A,B)
```

```
Current State: CLEAR(A),CLEAR(B),CLEAR(C),HOLDING(A,L),HOLDING(B,R),ON-TABLE(C)
  Proposed Actions: PICK-UP-LEFT(A),PICK-UP-RIGHT(B),UNSTACK-RIGHT(B,C)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(L),HOLDING(B,R),ON-TABLE(A),ON-TABLE(C)
    Analyzing: PICK-UP-RIGHT(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(R),HOLDING(A,L),ON-TABLE(B),ON-TABLE(C)
    Analyzing: UNSTACK-RIGHT(B,C)
      Next State: CLEAR(A),CLEAR(B),EMPTY-ARM(R),HOLDING(A,L),ON(B,C),ON-TABLE(C)
        State already visited
Iteration: 5
Previous Actions: STACK(B,C),PICK-UP-LEFT(A),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(R),HOLDING(B,L),ON-TABLE(A),ON-TABLE(C)
  Proposed Actions:
LEAVE(A),LEAVE(C),PICK-UP-LEFT(B),UNSTACK-LEFT(B,A),UNSTACK-LEFT(B,C)
    Analyzing: LEAVE(A)
      Next State: CLEAR(A),CLEAR(B),CLEAR(C),HOLDING(A,R),HOLDING(B,L),ON-TABLE(C)
    Analyzing: LEAVE(C)
      Next State: CLEAR(A),CLEAR(B),CLEAR(C),HOLDING(B,L),HOLDING(C,R),ON-TABLE(A)
    Analyzing: PICK-UP-LEFT(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(L),EMPTY-ARM(R),ON-TABLE(A),ON-TABLE(B),ON-TABL
E(C)
    Analyzing: UNSTACK-LEFT(B,A)
      Next State:
CLEAR(B),CLEAR(C),EMPTY-ARM(L),EMPTY-ARM(R),ON(B,A),ON-TABLE(A),ON-TABLE(C)
DONE!
Number of operators: 4
Number of states generated: 13
Plan: UNSTACK-LEFT(B,A),STACK(B,C),PICK-UP-LEFT(A),STACK(A,B)
```

2 - Problem 1 output

```
Iteration: 1
Current State:
CLEAR(C),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(C,B),ON(F,A),ON-TABLE(B),ON-T
ABLE(D)
  Proposed Actions: STACK(C,B),STACK(F,A)
    Analyzing: STACK(C,B)
      Next State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),HOLDING(C,R),ON(A,D),ON(F,A),ON-TABLE(B),ON-
TABLE(D)
    Analyzing: STACK(F,A)
```

```
      Next State:
CLEAR(A),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(F,L),ON(A,D),ON(C,B),ON-TABLE(B),ON-
TABLE(D)
Iteration: 2
Previous Actions: STACK(C,B)
Current State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),HOLDING(C,R),ON(A,D),ON(F,A),ON-TABLE(B),ON-
TABLE(D)
  Proposed Actions: PICK-UP-RIGHT(C),STACK(F,A),UNSTACK-RIGHT(C,B)
    Analyzing: PICK-UP-RIGHT(C)
      Next State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(F,A),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(D)
    Analyzing: STACK(F,A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(C,R),HOLDING(F,L),ON(A,D),ON-TABLE(B),ON
-TABLE(D)
    Analyzing: UNSTACK-RIGHT(C,B)
      Next State:
CLEAR(C),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(C,B),ON(F,A),ON-TABLE(B),ON-T
ABLE(D)
        State already visited
...
Iteration: 4
Previous Actions: PICK-UP-RIGHT(C),STACK(C,B)
Current State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(F,A),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(D)
  Proposed Actions: LEAVE(B),LEAVE(C),STACK(F,A)
    Analyzing: LEAVE(B)
      Next State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),HOLDING(B,R),ON(A,D),ON(F,A),ON-TABLE(C),ON-
TABLE(D)
    Analyzing: LEAVE(C)
      Next State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),HOLDING(C,R),ON(A,D),ON(F,A),ON-TABLE(B),ON-
TABLE(D)
        State already visited
    Analyzing: STACK(F,A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(F,L),ON(A,D),ON-TABLE(B),ON
-TABLE(C),ON-TABLE(D)
...
Iteration: 9
Previous Actions: LEAVE(B),PICK-UP-RIGHT(C),STACK(C,B)
Current State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),HOLDING(B,R),ON(A,D),ON(F,A),ON-TABLE(C),ON-
TABLE(D)
  Proposed Actions: PICK-UP-RIGHT(B),STACK(F,A),UNSTACK-RIGHT(B,C)
```

```
    Analyzing: PICK-UP-RIGHT(B)
      Next State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(F,A),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(D)
        State already visited
    Analyzing: STACK(F,A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(B,R),HOLDING(F,L),ON(A,D),ON-TABLE(C),ON
-TABLE(D)
    Analyzing: UNSTACK-RIGHT(B,C)
      Next State:
CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(B,C),ON(F,A),ON-TABLE(C),ON-T
ABLE(D)
...
Iteration: 20
Previous Actions: UNSTACK-RIGHT(B,C),LEAVE(B),PICK-UP-RIGHT(C),STACK(C,B)
Current State:
CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(B,C),ON(F,A),ON-TABLE(C),ON-T
ABLE(D)
  Proposed Actions: STACK(B,C),STACK(F,A)
    Analyzing: STACK(B,C)
      Next State:
CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),HOLDING(B,R),ON(A,D),ON(F,A),ON-TABLE(C),ON-
TABLE(D)
        State already visited
    Analyzing: STACK(F,A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(R),HOLDING(F,L),ON(A,D),ON(B,C),ON-TABLE(C),ON-
TABLE(D)
...
Iteration: 41
Previous Actions: STACK(F,A),UNSTACK-RIGHT(B,C),LEAVE(B),PICK-UP-RIGHT(C),STACK(C,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(R),HOLDING(F,L),ON(A,D),ON(B,C),ON-TABLE(C),ON-
TABLE(D)
  Proposed Actions:
PICK-UP-LEFT(F),STACK(A,D),STACK(B,C),UNSTACK-LEFT(F,A),UNSTACK-LEFT(F,B)
    Analyzing: PICK-UP-LEFT(F)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(B,C),ON-TABLE(C),ON-
TABLE(D),ON-TABLE(F)
        State already visited
    Analyzing: STACK(A,D)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(D),CLEAR(F),HOLDING(A,R),HOLDING(F,L),ON(B,C),ON-TABLE(C),ON
-TABLE(D)
    Analyzing: STACK(B,C)
```

```
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(B,R),HOLDING(F,L),ON(A,D),ON-TABLE(C),ON
-TABLE(D)
        State already visited
    Analyzing: UNSTACK-LEFT(F,A)
      Next State:
CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(B,C),ON(F,A),ON-TABLE(C),ON-T
ABLE(D)
        State already visited
    Analyzing: UNSTACK-LEFT(F,B)
      Next State:
CLEAR(A),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,D),ON(B,C),ON(F,B),ON-TABLE(C),ON-T
ABLE(D)
...
Iteration: 100
Previous Actions:
UNSTACK-LEFT(F,D),STACK(A,D),STACK(F,A),UNSTACK-RIGHT(B,C),LEAVE(B),PICK-UP-RIGHT(C)
,STACK(C,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(L),HOLDING(A,R),ON(B,C),ON(F,D),ON-TABLE(C),ON-
TABLE(D)
  Proposed Actions:
PICK-UP-RIGHT(A),STACK(F,D),UNSTACK-RIGHT(A,B),UNSTACK-RIGHT(A,F)
    Analyzing: PICK-UP-RIGHT(A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(B,C),ON(F,D),ON-TABLE(A),ON-
TABLE(C),ON-TABLE(D)
        State already visited
    Analyzing: STACK(F,D)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(D),CLEAR(F),HOLDING(A,R),HOLDING(F,L),ON(B,C),ON-TABLE(C),ON
-TABLE(D)
        State already visited
    Analyzing: UNSTACK-RIGHT(A,B)
      Next State:
CLEAR(A),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(B,C),ON(F,D),ON-TABLE(C),ON-T
ABLE(D)
DONE!
Number of operators: 8
Number of states generated: 350
Plan:
UNSTACK-RIGHT(A,B),UNSTACK-LEFT(F,D),STACK(A,D),STACK(F,A),UNSTACK-RIGHT(B,C),LEAVE(
B),PICK-UP-RIGHT(C),STACK(C,B)
```

3 - Problem 2 output

```
Iteration: 1
Current State:
```

```
CLEAR(A),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(B,E),ON(F,C),ON-TABLE(C),ON-T
ABLE(E)
  Proposed Actions: STACK(A,B),STACK(F,C)
    Analyzing: STACK(A,B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(B,E),ON(F,C),ON-TABLE(C),ON-
TABLE(E)
    Analyzing: STACK(F,C)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(F,L),ON(A,B),ON(B,E),ON-TABLE(C),ON-
TABLE(E)
Iteration: 2
Previous Actions: STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(B,E),ON(F,C),ON-TABLE(C),ON-
TABLE(E)
  Proposed Actions:
PICK-UP-LEFT(A),STACK(B,E),STACK(F,C),UNSTACK-LEFT(A,B),UNSTACK-LEFT(A,F)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(B,E),ON(F,C),ON-TABLE(A),ON-
TABLE(C),ON-TABLE(E)
    Analyzing: STACK(B,E)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(B,R),ON(F,C),ON-TABLE(C),ON
-TABLE(E)
    Analyzing: STACK(F,C)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(A,L),HOLDING(F,R),ON(B,E),ON-TABLE(C),ON
-TABLE(E)
    Analyzing: UNSTACK-LEFT(A,B)
      Next State:
CLEAR(A),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(B,E),ON(F,C),ON-TABLE(C),ON-T
ABLE(E)
        State already visited
    Analyzing: UNSTACK-LEFT(A,F)
      Next State:
CLEAR(A),CLEAR(B),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,F),ON(B,E),ON(F,C),ON-TABLE(C),ON-T
ABLE(E)
...
Iteration: 5
Previous Actions: STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(B,R),ON(F,C),ON-TABLE(C),ON
-TABLE(E)
  Proposed Actions:
PICK-UP-LEFT(A),PICK-UP-RIGHT(B),UNSTACK-LEFT(A,F),UNSTACK-RIGHT(B,E)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(E),CLEAR(F),EMPTY-ARM(L),HOLDING(B,R),ON(F,C),ON-TABLE(A),ON
-TABLE(C),ON-TABLE(E)
        State already visited
    Analyzing: PICK-UP-RIGHT(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(E),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(F,C),ON-TABLE(B),ON
-TABLE(C),ON-TABLE(E)
    Analyzing: UNSTACK-LEFT(A,F)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(E),EMPTY-ARM(L),HOLDING(B,R),ON(A,F),ON(F,C),ON-TABLE(C),ON-
TABLE(E)
```

```
    Analyzing: UNSTACK-RIGHT(B,E)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(B,E),ON(F,C),ON-TABLE(C),ON-
TABLE(E)
        State already visited
...
Iteration: 14
Previous Actions: PICK-UP-RIGHT(B),STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(E),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(F,C),ON-TABLE(B),ON
-TABLE(C),ON-TABLE(E)
  Proposed Actions:
LEAVE(B),LEAVE(E),STACK(F,C),UNSTACK-LEFT(A,B),UNSTACK-LEFT(A,E),UNSTACK-LEFT(A,F)
    Analyzing: LEAVE(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(B,R),ON(F,C),ON-TABLE(C),ON
-TABLE(E)
        State already visited
    Analyzing: LEAVE(E)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(E,R),ON(F,C),ON-TABLE(B),ON
-TABLE(C)
    Analyzing: STACK(F,C)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(F,R),ON-TABLE(B),O
N-TABLE(C),ON-TABLE(E)
    Analyzing: UNSTACK-LEFT(A,B)
      Next State:
CLEAR(A),CLEAR(E),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(F,C),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(E)
    Analyzing: UNSTACK-LEFT(A,E)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,E),ON(F,C),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(E)
    Analyzing: UNSTACK-LEFT(A,F)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(E),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,F),ON(F,C),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(E)
...
Iteration: 29
Previous Actions: STACK(F,C),PICK-UP-RIGHT(B),STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(F,R),ON-TABLE(B),O
N-TABLE(C),ON-TABLE(E)
  Proposed Actions: UNSTACK-RIGHT(F,B),UNSTACK-RIGHT(F,C),UNSTACK-RIGHT(F,E)
    Analyzing: UNSTACK-RIGHT(F,B)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(E),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(F,B),ON-TABLE(B),ON
-TABLE(C),ON-TABLE(E)
    Analyzing: UNSTACK-RIGHT(F,C)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(E),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(F,C),ON-TABLE(B),ON
-TABLE(C),ON-TABLE(E)
        State already visited
    Analyzing: UNSTACK-RIGHT(F,E)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(F,E),ON-TABLE(B),ON
-TABLE(C),ON-TABLE(E)
...
Iteration: 48
```

```
Previous Actions:
UNSTACK-RIGHT(F,B),STACK(F,C),PICK-UP-RIGHT(B),STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(C),CLEAR(E),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(F,B),ON-TABLE(B),ON
-TABLE(C),ON-TABLE(E)
  Proposed Actions:
LEAVE(C),LEAVE(E),STACK(F,B),UNSTACK-LEFT(A,C),UNSTACK-LEFT(A,E),UNSTACK-LEFT(A,F)
    Analyzing: LEAVE(C)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(C,R),ON(F,B),ON-TABLE(B),ON
-TABLE(E)
    Analyzing: LEAVE(E)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(E,R),ON(F,B),ON-TABLE(B),ON
-TABLE(C)
    Analyzing: STACK(F,B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(F,R),ON-TABLE(B),O
N-TABLE(C),ON-TABLE(E)
        State already visited
    Analyzing: UNSTACK-LEFT(A,C)
      Next State:
CLEAR(A),CLEAR(E),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,C),ON(F,B),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(E)
    Analyzing: UNSTACK-LEFT(A,E)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,E),ON(F,B),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(E)
    Analyzing: UNSTACK-LEFT(A,F)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(E),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,F),ON(F,B),ON-TABLE(B),ON-
TABLE(C),ON-TABLE(E)
...
Iteration: 67
Previous Actions:
LEAVE(C),UNSTACK-RIGHT(F,B),STACK(F,C),PICK-UP-RIGHT(B),STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(C),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(C,R),ON(F,B),ON-TABLE(B),ON
-TABLE(E)
  Proposed Actions:
PICK-UP-LEFT(A),PICK-UP-RIGHT(C),UNSTACK-LEFT(A,F),UNSTACK-RIGHT(C,E)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(E),CLEAR(F),EMPTY-ARM(L),HOLDING(C,R),ON(F,B),ON-TABLE(A),ON
-TABLE(B),ON-TABLE(E)
    Analyzing: PICK-UP-RIGHT(C)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(E),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(F,B),ON-TABLE(B),ON
-TABLE(C),ON-TABLE(E)
        State already visited
    Analyzing: UNSTACK-LEFT(A,F)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(E),EMPTY-ARM(L),HOLDING(C,R),ON(A,F),ON(F,B),ON-TABLE(B),ON-
TABLE(E)
    Analyzing: UNSTACK-RIGHT(C,E)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(C,E),ON(F,B),ON-TABLE(B),ON-
TABLE(E)
...
Iteration: 103
```

```
Previous Actions:
UNSTACK-RIGHT(C,E),LEAVE(C),UNSTACK-RIGHT(F,B),STACK(F,C),PICK-UP-RIGHT(B),STACK(B,E
),STACK(A,B)
Current State:
CLEAR(A),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(C,E),ON(F,B),ON-TABLE(B),ON-
TABLE(E)
  Proposed Actions:
PICK-UP-LEFT(A),STACK(C,E),STACK(F,B),UNSTACK-LEFT(A,C),UNSTACK-LEFT(A,F)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(C,E),ON(F,B),ON-TABLE(A),ON-
TABLE(B),ON-TABLE(E)
        State already visited
    Analyzing: STACK(C,E)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(C,R),ON(F,B),ON-TABLE(B),ON
-TABLE(E)
        State already visited
    Analyzing: STACK(F,B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(A,L),HOLDING(F,R),ON(C,E),ON-TABLE(B),ON
-TABLE(E)
    Analyzing: UNSTACK-LEFT(A,C)
      Next State:
CLEAR(A),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,C),ON(C,E),ON(F,B),ON-TABLE(B),ON-T
ABLE(E)
    Analyzing: UNSTACK-LEFT(A,F)
      Next State:
CLEAR(A),CLEAR(C),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,F),ON(C,E),ON(F,B),ON-TABLE(B),ON-T
ABLE(E)
        State already visited
...
Iteration: 144
Previous Actions:
STACK(F,B),UNSTACK-RIGHT(C,E),LEAVE(C),UNSTACK-RIGHT(F,B),STACK(F,C),PICK-UP-RIGHT(B
),STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(A,L),HOLDING(F,R),ON(C,E),ON-TABLE(B),ON
-TABLE(E)
  Proposed Actions:
PICK-UP-LEFT(A),PICK-UP-RIGHT(F),UNSTACK-RIGHT(F,B),UNSTACK-RIGHT(F,C)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),HOLDING(F,R),ON(C,E),ON-TABLE(A),ON
-TABLE(B),ON-TABLE(E)
    Analyzing: PICK-UP-RIGHT(F)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(C,E),ON-TABLE(B),ON
-TABLE(E),ON-TABLE(F)
    Analyzing: UNSTACK-RIGHT(F,B)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(C,E),ON(F,B),ON-TABLE(B),ON-
TABLE(E)
        State already visited
    Analyzing: UNSTACK-RIGHT(F,C)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(C,E),ON(F,C),ON-TABLE(B),ON-
TABLE(E)
...
Iteration: 176
```

```
Previous Actions:
PICK-UP-RIGHT(F),STACK(F,B),UNSTACK-RIGHT(C,E),LEAVE(C),UNSTACK-RIGHT(F,B),STACK(F,C
),PICK-UP-RIGHT(B),STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(C,E),ON-TABLE(B),ON
-TABLE(E),ON-TABLE(F)
  Proposed Actions:
LEAVE(B),LEAVE(F),STACK(C,E),UNSTACK-LEFT(A,B),UNSTACK-LEFT(A,C),UNSTACK-LEFT(A,F)
    Analyzing: LEAVE(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(A,L),HOLDING(B,R),ON(C,E),ON-TABLE(E),ON
-TABLE(F)
    Analyzing: LEAVE(F)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(A,L),HOLDING(F,R),ON(C,E),ON-TABLE(B),ON
-TABLE(E)
        State already visited
    Analyzing: STACK(C,E)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(E),CLEAR(F),HOLDING(A,L),HOLDING(C,R),ON-TABLE(B),O
N-TABLE(E),ON-TABLE(F)
        State already visited
    Analyzing: UNSTACK-LEFT(A,B)
      Next State:
CLEAR(A),CLEAR(C),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,B),ON(C,E),ON-TABLE(B),ON-
TABLE(E),ON-TABLE(F)
        State already visited
    Analyzing: UNSTACK-LEFT(A,C)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,C),ON(C,E),ON-TABLE(B),ON-
TABLE(E),ON-TABLE(F)
    Analyzing: UNSTACK-LEFT(A,F)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,F),ON(C,E),ON-TABLE(B),ON-
TABLE(E),ON-TABLE(F)
...
Iteration: 195
Previous Actions:
LEAVE(B),PICK-UP-RIGHT(F),STACK(F,B),UNSTACK-RIGHT(C,E),LEAVE(C),UNSTACK-RIGHT(F,B),
STACK(F,C),PICK-UP-RIGHT(B),STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(A,L),HOLDING(B,R),ON(C,E),ON-TABLE(E),ON
-TABLE(F)
  Proposed Actions:
PICK-UP-LEFT(A),PICK-UP-RIGHT(B),UNSTACK-LEFT(A,F),UNSTACK-RIGHT(B,C)
    Analyzing: PICK-UP-LEFT(A)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(L),HOLDING(B,R),ON(C,E),ON-TABLE(A),ON
-TABLE(E),ON-TABLE(F)
        State already visited
    Analyzing: PICK-UP-RIGHT(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(C,E),ON-TABLE(B),ON
-TABLE(E),ON-TABLE(F)
        State already visited
    Analyzing: UNSTACK-LEFT(A,F)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(L),HOLDING(B,R),ON(A,F),ON(C,E),ON-TABLE(E),ON-
TABLE(F)
    Analyzing: UNSTACK-RIGHT(B,C)
```

```
      Next State:
CLEAR(A),CLEAR(B),CLEAR(F),EMPTY-ARM(R),HOLDING(A,L),ON(B,C),ON(C,E),ON-TABLE(E),ON-
TABLE(F)
...
Iteration: 221
Previous Actions:
UNSTACK-LEFT(A,F),LEAVE(B),PICK-UP-RIGHT(F),STACK(F,B),UNSTACK-RIGHT(C,E),LEAVE(C),U
NSTACK-RIGHT(F,B),STACK(F,C),PICK-UP-RIGHT(B),STACK(B,E),STACK(A,B)
Current State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(L),HOLDING(B,R),ON(A,F),ON(C,E),ON-TABLE(E),ON-
TABLE(F)
  Proposed Actions: PICK-UP-RIGHT(B),STACK(A,F),UNSTACK-RIGHT(B,C)
    Analyzing: PICK-UP-RIGHT(B)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,F),ON(C,E),ON-TABLE(B),ON-
TABLE(E),ON-TABLE(F)
        State already visited
    Analyzing: STACK(A,F)
      Next State:
CLEAR(A),CLEAR(B),CLEAR(C),CLEAR(F),HOLDING(A,L),HOLDING(B,R),ON(C,E),ON-TABLE(E),ON
-TABLE(F)
        State already visited
    Analyzing: UNSTACK-RIGHT(B,C)
      Next State:
CLEAR(A),CLEAR(B),EMPTY-ARM(L),EMPTY-ARM(R),ON(A,F),ON(B,C),ON(C,E),ON-TABLE(E),ON-T
ABLE(F)
DONE!
Number of operators: 12
Number of states generated: 735
Plan:
UNSTACK-RIGHT(B,C),UNSTACK-LEFT(A,F),LEAVE(B),PICK-UP-RIGHT(F),STACK(F,B),UNSTACK-RI
GHT(C,E),LEAVE(C),UNSTACK-RIGHT(F,B),STACK(F,C),PICK-UP-RIGHT(B),STACK(B,E),STACK(A,
B)
```