KTH Royal Institute of Technology

DD2380
Artificial Intelligence

Year 2017 - 2018

Floortile Problem

Group Project

Authors, Group 22 :

Hedström Anna	940307 – 3027	annaheds@kth.se
Legat Antoine	950311 – T297	legat@kth.se
Picó Oristrell Sandra	940531 – T268	sandrapo@kth.se
Vega Ramírez David	961202 – T031	dvr@kth.se

Professors :
Tumova Jana
Jensfelt Patric

October 5, 2017

1 Abstract

This paper puts forward a solution to a planning problem that allows for an empirical investigation using two different methodologies, namely, a heuristic search with Force Ordering Constraints (FOC) and an optimisation approach. More precisely, for this exercise we interpreted, formulated and implemented a modified version of the Floor Tile problem, which was initially proposed in PDDL language ("Planning Domain Definition Language") [1] at the International Planning Competition (IPC) in 2011. From our reported findings, we concluded that a standard forward search in this domain runs relatively slow; which in turn forced us to exploit more advanced techniques, amongst others, partial ordering constraints and the concept Goal Agenda Manager [2], in efforts to reduce the huge branching. We discuss our planning solution in comparison with the results obtained from the optimisation approach and finally conclude that in this domain, an PDDL implementation is far more efficient as such.

Contents

2 Introduction

The interestingness of the problem arguably lies in the particular configuration that makes the domain hard, namely, the existence of implicitly embedded constraints (often called Forced Goal Constraints). This domain further implies on three main sources of difficulty; one being the huge branching factor, the other the advent of deadlocks, which both are amplified by the third, that is the complexity brought by the multiple agents in the environment. Thus, it comes at no surprise that the sequential task remained unsolved for three years. It was not until 2014, exactly three years later, researchers managed to solve in a reasonable amount of time.

For this report, we will extend the initial interpretation of the problem, that is PDDL, by formulating it as an optimisation problem as well. In the light of this, our focus will be to conduct an empirical study of both approaches, investigating the performance limits of the algorithm(s) presented. To achieve this, we prepared a set of case studies, considering complexities such as different painting patterns, grid sizes and the number of robots and reported CPU running time as basis for discussion.

The report is organised as follows. We first present related work of this domain, which in effect presents a systematic literature review of existing methodologies and tools. Next, we further invite the reader for a description of methods, including formulation, implementation and case studies. Thereafter, in the Findings and Analysis section, we describe, explain and expound on the obtained results, while studying the performance limits for the proposed cases. Here, we also discuss the differences between the two approaches. Finally, for the Summary, we reflect on real world applications and future work. Further elaborations, formal formulations and code snippets can be found in the Appendices.

3 Related work

This section mirrors our efforts to conduct a systematic literature review, to help ensure theoretical underpinning throughout the entire problem solving process. As seen, the section is divided into three parts; first we explain our examination of relevant methodologies, including literature on representation, planners and heuristics, and then discuss the tools used for implementation, that are libraries, languages and design specifications.

3.1 Review of Methodology

Planning as known today entails a vast body of research. Most notably, this is evidenced by the active stream of extensions to improve what is referred to as the current state-of-the-art; may it be for expressivity in representation or efficiency in planners. At explained, the representation of the Floor Tile problem was initially proposed in PDDL, first originated in 1998 [1]. While we note that many alternative representations exist (e.g. propositional-, state-variable- and STRIPS representation [3]), PDDL is arguably considered the most recent attempt to standardise planning domain and problem description languages. Provably, many modifications have been adapted since it was first introduced, for example, object fluents [4], action costs [5] and adaptation towards multi-agent systems [6], through the many revisions that have accompanied the IPC competitions since its beginning.

For the planner, it comes at no surprise that many different algorithmic solutions exists. For state-space search, there are various progression and regression planners, and for partial-order planning, many heuristics have also been proposed. Among the previous work for planners, the most relevant methods to our approach is forward search [7], which essentially search progressively from the initial state and determines which actions apply using preconditions and removes/ adds lists to compute a new state (see implementation for details). That said, it was clear from the start that this algorithm was fit for the huge branching factor it would face in the domain. While developing an understanding of the intricacies of the domain, a set of inherent sources of difficulty emerged, as follows:

1. a large branching factor b caused by the huge search space of forward search,
2. an arrival of deadlocks caused by non-determinism in painting actions, and,
3. a complexity of n agent system, caused by the multi-agent property.

Thus, we moved on to study more advanced topics of planning, as the next section expounds, addressing each of these issues in turn. First, in this review, we came across several interesting algorithmic implementations (even a few attempts in the same domain see references [8] and [9]), proposing depth-unbounded search and multi-step forward search respectively, to address the above-listed issues. To avoid that the algorithm rechecks every available action at repeated states, we learned from [10], that hash tables could enable efficient access for repeated states. According to Russel & Norvig [11], hash tables have the potential to provide fast look-up, it can be done in constant time. Also, perhaps more relevant, is the Goal Agenda Manger (GAM) heuristic, proposed by [2], aimed at detecting reasonable goal orderings, implicitly implying on a prioritisation of actions, or cells painted, if you will. GAM is typically employed to check the ordering relationships for atomic goal pairs (which all exists in the initial state) , to then the goal set to many subsets and by this ensuring that a planner can achieve the overarching goals, in one sequence. For this case, this goal is naturally to paint *all* the tiles. Moreover, this implies, that if implemented properly, the robot would only paint tiles obeying the correct sequence, that is painting the up-most tiles first and thus help to avoid the deadlock problem. In addition, we also found from a Picat implementation on IPC'14 [9] that removing actions, more specifically the 'paint-down' actions, could be exploited to help reduce the non-determinism inherent in the painting rules.

Lastly, to adapt to multi-agent systems, that are by definition composed of multiple interacting intelligent agents within an environment, we learned from [11] that it could be solved by writing conditional action schemas, as if the agents acted fully independently (allowing for a decentralised planning). The implementation could be found in section 4.1.3.

3.2 Review of Tools

While it is tempting to shuffle between existing PDDL parsers and solvers, all easily accessible online (e.g. [12]), we preferred to complete an implementation from scratch to enable constraint modifications and implementations of search heuristics, as explained above. For this, as a starter, we reviewed relevant *github* libraries e.g. [13] that provided a parsing interface. However, after careful consideration we ended up creating our own parser, as this would allow us to more flexibly parse the set of structures we needed to work around the domain and problem files in Python.

After having reviewed off-the-shelf tools with purpose of graphical interface simulation e.g. *ROS* and *PyQt*, we soon concluded that existing graphical simulators lacked the aspect of adaptability we looked for. Thus, by using *Python turtle* library we could custom build our graphical interface, intentionally admittedly trading simplicity over library over more "aesthetically appealing" ones. *RStudio* was used for creation of plotting of results.

4 Methods and Implementation

In this section, we formulate the problem, both as a PDDL and optimisation, explain our implementation, including parsing, planning and heuristic search, and declare the case studies subject that was to testing.

4.1 PDDL

4.1.1 Formulation of Floor Tile Problem

As the PDDL formulation in section A in the Appendices explains, for each problem, we have a 2D grid of tiles T_1, ..., T_n. We have r robots, t tiles and c colors. Each robot is located at some tile T_n and holds a color c. In the initial state, all the floor tiles are clear. According to the original description, the tiles needs to be painted black and white in an alternated fashion *always*. Once a tile is painted, a robot cannot stand on it. Robots can only paint up and down. The initial placements of robots are defined in the problem file. The state space is determined by the *initial state*, including the set of tiles given at outset.

The *goal* is to paint all floor tiles according to some target state. To reach the goal, there are seven different *actions* at the robots' disposal.

1. The robot can move ($up, down, left, right$),
2. A robot can paint ($up, down$),
3. A robot can change color ($black, white$),

This means that the up-most tiles need to be painted first. The reason is that the atom Robot-at(r, t_x) is mutually exclusive with the atomic goal being Painted(c,t_x) which cannot be removed once it has been added, causing the search to arrive at a deadlock.

4.1.2 Interpretation

For our planner to understand the internal structure of our states, that is presented by the above seen first order predicate logic, we specified a parser to enable syntax parsing. Thus, we created a file (*parse.py*) that takes the information from the PDDL file and generates the 2D matrices and 1D vectors, to analyse for the main file. Parse file uses the some structures supported by the *pddlpy* library. Our parsing implementation allowed us to adapt our code to any problem file and ensure that all conditions are satisfied once a search is executed.

As seen below, by using these values we simplify our board, since now we can identify more easily the current state of our board as well as the necessities to update the board.

1	0	3
0	0	2
3	0	1
0	0	0

Figure 1: Board representation

Value	Use
0	The Cell is clear
1	Cell has been painted white
2	Cell has been painted black
3	Robot is in top of the cell

Table 1: Meaning of the values in the matrix

The robots are being represented as a 1D vector, where each position is an integer value, similar to the representation of the board.

1	1	2	100	100

Figure 2: Robot Representation

Each cell defines the current state for the robot, the first two cells define the current 'X' and 'Y' position of the robot and the third cell represents the current colour that the robot it's using. In Figure 2 the robot is currently using the colour 'black', having the last two values represent the amount of paint remaining for 'white' and 'black' paint.

The combination of these two representations (board and robots) gives us an unique overall state, which can help us differentiate every possible state during the planning. Since this combination its unique we can make use of 'memoisation' in order to avoid loops/duplication of states later on during the planning, that are hash tables.

4.1.3 Implementation

This is the algorithm that we are using in order to solve the problem of planning, which is always guaranteed to find the most optimal sequence of states that can solve the problem. In essence, we are generating a tree and traversing it by levels like a breath-first-search (BFS). By using the 'memoisation' of states we can keep the graph as a tree avoiding the loops. We keep track of what movements were done by each robot by appending the next possible moves to he ones that were before those, so at the end of the process we are able to reconstruct their paths.

The function `getSequence` generates the new edges for the tree, inside the function we find all the possible movements for robots given our current state.

Algorithm 1 Planning algorithm

procedure PLANNING(ROBOTS,INITIALSTATE,TARGETSTATE)
    state its defined as [robots,initialState,sequenceMoves]
    q ← queue of states
    memory ← dictionary of states
    q ← push [robots,initialState,[]]
    while:
    if *q is empty* **then**
        break
    current ← *top(q)*
    pop(q)
    if *memory[current]* == *true*) **then**
        continue
    if *current[1]* == *targetState* **then**
        return current
    memory[current] = *true*
    newPossibleStates ← $getSequence(current[0], 0, current[1], targetState]$.
    For possible in newPossibleStates:
    q ← *push [possible[0], possible[1], current[2].append(possible[2])*
    goto *For*
    goto *while*.
    return null.

Algorithm 2 getSequence algorithm

procedure GETSEQUENCE(ROBOTS,INDEX,STATE,TARGETSTATE,MOVEMENTS)
    if *index* == *len(robots* **then**
        resStates.append([robots,index,movements]
        return
    nextMovements ← *getPossiblesFOC(robot[index],state,targetState)*
    For next in nextMovements:
    getPossibles(robots,index+1,next[0],targetState,movements.append(next[1])
    goto *For*
    if *index* == *0* **then**
        return resStates

In this algorithm we are using recursion in order to go through all of our robots. Based on what the previous robot did we generate a new state and we try that state with the subsequent robot. Therefore obtaining all the possible combinations that could be achieved by the robots, given our initial state that was given from our previous algorithm. To put it in perspective this method generates a new tree of possibilities were the leafs are all the possible states after all the robots have completed an action.

We are making use of a global variable `resStates` in order to keep track of the leafs/ possible states.

Algorithm 3 getPossiblesFOC algorithm

procedure GETPOSSIBLES(ROBOT,STATE,TARGETSTATE)
  res = []
  movements = [up,down,left,right]
  For mov in movements:
  aux = tryMovement(mov,robot,state)
  if *aux == NULL* **then**
    res.append(aux)
  goto *For*
  if *tryMovement(up,robot,state) and paintedColumn(robot,state) and state[robot[0:1]] == robot[2]* **then**
    res.append(paintUp(robot,state))
  if *tryMovement(downt,robot,state) and paintedColumn(robot,state) and state[robot[0:1]] == robot[2]* **then**
    res.append(paintDown(robot,state))
  res.append(changeColor(robot,state))
  res.append(still(robot,state))
  return res

Since our algorithm is operating in an environment defined by deterministic properties (such as finite, static etc) and not stochastic, our algorithm always need to explore all possible states before determining the best solution.

`getPossiblesFoc` is our satisfiability function that determines which actions are valid for a given robot at a given state. First, we tried to move the robot up, left, right, down. However, with the realisation that actions could be ordered in a priority queue (as for the assumed independence property), or example, that Painted(t_{x1}, $t_{y1}) and Painted($t$_{x2}$, t_{y1}) can be completed only after Painted(t_{x3}, t_{y1}) is painted, we decided to make use of (FOC) to make the robots paint the top-most rows first. This means that the robots are going to avoid deadlocks by painting tiles that should be painted after other tiles. By using this approach we are effectively pruning non-viable states.

Assuming its binding constraints are consistent, the ordering works as follows:

$Painted Paintedetc$,

The possible movements of the robots are being recorded in the list called `rest` which for a given index looks like `res[i] -> [robot,state]`.

In order to view and simulate the result obtained by the main file, we created a graphical simulation that represents the grid, movements of the robot and the all the steps in the path taken to achieve the target state. This simulation is executed through the `turtle Python` library.

To achieve this, we had an `Interface` file receiving information for each case as follows: the exact path, the initial state, the initial positions of the robots as well as the dimensions of the grid. Yet, to be able to simulate the graphics, the path obtained must be transformed into an appropriate array. Finally, we had the file analysing the data, drawing a simulation, taking into account that each tile has dimension of 80x80 pixels. An example of the resulting output can be visualised in 3.

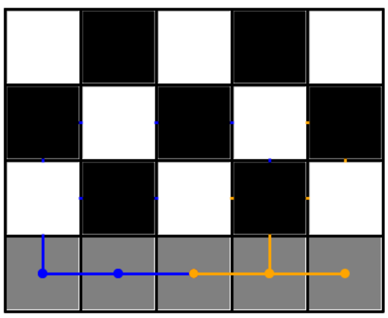

Figure 3: Graphics Representation

4.1.4 Case studies

As this section expounds, the development of case studies remains one of the most important tasks of a planning process, as it completely set the scope for the analysis and the possible applications outside of domain.

As a starter, we wanted to study the effect that the heuristic search had on the computational costs, in which we reported the difference in CPU running time between standard forward search and heuristic search, as declared above. Naturally the latter implies a lot of pruning, i.e. removing irrelevant states from the state space, thus notable efficiency gains were expected in terms of CPU running time.

Case 1: **Size of Grid**

Experimenting with the size of the grid is a common test case in planning contexts. For this exercise, we created ten different scenarios, using ten different PDDL problem files, that reported different dimensions of the grid. In this case, we were not only interested to study the effects from incrementing the grid size, but we were also curious to better understand whether shape also had an isolating effect on the CPU running time. Thus, to see whether computational costs would differ if $N > M$ or vice versa, we designed a few test cases where the number of tiles remain the same, while varying the dimensions.

Case 2: **Painted Pattern**

Next, while the original Floor Tile problem declared that the painted tiles should be colored in an alternated fashion *always*, we were interested to see what happened if we change the pattern. Naturally, this meant that we had to re-write the goal condition as such; from having all the tiles painted to only paint a pre-determined set of tiles.

Case 3: **Number of Robots**

For the third case we wanted to study the effects of multi-agent systems, more precisely, how the number of robots effect the CPU running time. We created three test cases with 2, 3 and 4 robots.

We believe that this was a very interesting one, as it forced us to generalise our algorithm to conform to n set of robots, instead of two robots, that was initially proposed in the original description of the problem. Naturally, we fixed the grid size according to the max n robots as defined in our test cases, that to ensure that each robot have an unoccupied tile in the "exit zone".

4.2 Optimisation

In order to explore another way to solve this problem, we also formulated it as an optimisation problem and implemented it using AMPL, an optimisation modeling program. The current section describes this approach, and then we ran our implementation for different cases, which we will comment at the end of this section.

4.2.1 Formulation

Since it is quite a heavy formulation, we joined it in Appendices, please find it at section B.

4.2.2 Interpretation

We implemented this problem using AMPL, an optimisation program. You can find our implementations at section C. AMPL is usually used for linear or convex optimisation problems, but recently some new solvers arose with other possibilities.

Since our problem is not linear, and contains a lot of conditional constraints, we opted for `ilogcp` solver, a solver designed by IBM for combinatorial problems. IBM describes it this way: " IBM ILOG CPLEX CP Optimizer is a necessary and important complement to the optimization specialists' toolbox for solving real-world operational planning and scheduling problems". The approach of trying to solve this problem using `ilogcp` is thus relevant.

4.2.3 Case studies

We solved this problem for 3 cases:

1D, 1 robot The board is only composed of 3 cells. ilogcp found the optimal solution after 41086 choice points and 38256 fails, in 4.6489 seconds. For 4 cells, the solver was so slow that we didn't even let it run until the end.

2D, 1 robot The board is only composed of 2 × 2 cells. ilogcp found the optimal solution after 629759 choice points and 584051 fails, in 9.2822 seconds. For more cells, the solver was too slow.

2D, 2 robots The board is only composed of 2 × 2 cells. ilogcp found the optimal solution after 1812075 choice points and 1675749 fails, in 80.091 seconds. For more cells, the solver was too slow.

5 Findings and Analysis

This section is separated into four parts, presenting and analysing the experimental results obtained.

Before we get into the results of the case studies, we need to consider the criteria in which we choose to deem the quality of the performances of the algorithms. First, one may think that completeness are sensible criteria (could be identified by reporting the minimum number of cells visited/ cells re-visited/ movements- or even time-steps taken etc). However, noting that forward search is by definition sound and complete (the latter meaning that the planner is guaranteed to find a solution if such exists), we naturally resist such measures. Thus, as the following section reports, we use CPU time (reflecting time and space complexity) as our main criteria for reporting.

5.1 Results

At first, we discovered the huge differences in terms of efficiency, comparing the heuristic search versus standard forward search. As seen in figure 4, when combining elements of GAM, removal of actions (to reduce non-determinism) and hash tables (to avoid repeatable states) with the guarantee of completeness from the property of forward search, our solution adopted a much less computationally expensive approach in solving the Floor Tile Problem. More precisely we noted in the differences in grid sizes above 4x4 as seen in figure 4.

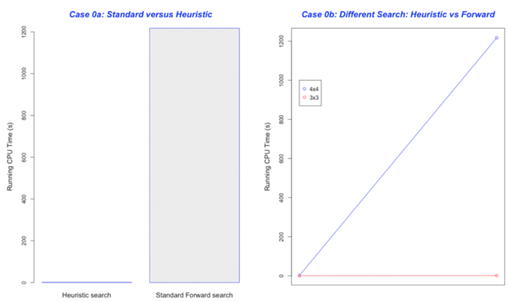

Figure 4: Case 0

As demonstrated in graphs, running time drastically decreased by implementing the heuristics, in which a decision was made to conduct the subsequent case studies using the heuristic search only. In Case 1, we tested the performance of our algorithm by varying the number of robots. As demonstrated in the graph (see figure 5), the CPU running was not linear, or proportional if you will, in respect to new robots added. We concluded that the domain suffers from the complexities that a multi-agent system implies.

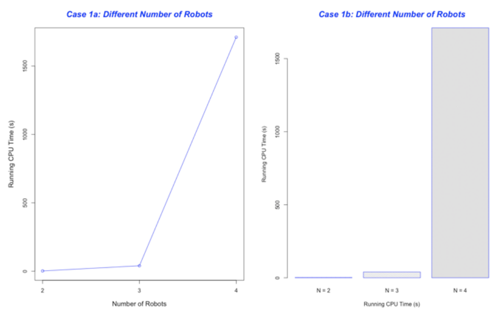

Figure 5: Case 1

For Case 2, we tested two cases with corresponding size, dimension and the number of robots, yet one which violated the initial rule of painting the tiles in an alternated fashion. As figure 6 shows, painting fewer time implied less CPU running time. That said, perhaps the takeaway from this case should not be that "a goal with fewer tiles painted is typically faster" (this is self-explanatory), rather one should view it in the context in which is was first developed; forcing additional constraints to be consider, therefore adding complexity to task.

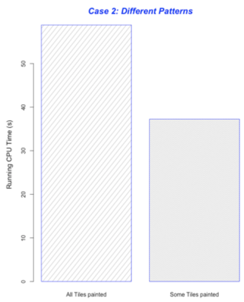

Figure 6: Case 2

Finally let us review the results for Case 3. At first glance, it might appear straight-forward that more tiles results in longer CPU running time and second, that this is not a linear relationship, as depicted in figure 7. However, as shown in figure 7, illustrating that the test cases where the number of columns were larger than the number of rows were more costly in respect to running time, although the number of tiles were exactly the same. Intuitively, this can be explained by the notion of our implemented FOC. In this heuristic, we could only restrict irrelevant vertical actions but not horisontal ones, leaving more options to move and hence also a larger state space to be evaluated.

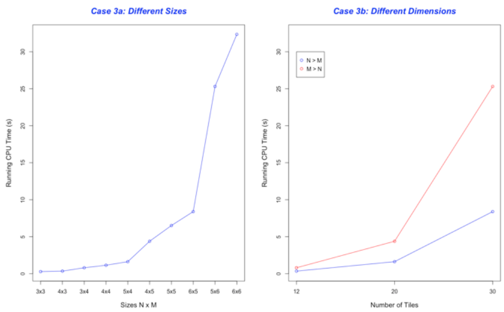

Figure 7: Case 3

5.2 Discussion

The main idea of our solution was to eliminate large portions of the search space. Most broadly, our heuristic search planner manage to do exactly this; we showed that as long as grid sizes remain larger than 4x4; dramatic efficiency gains emerge. Moreover, we admit to our surprise when we obtained the results from Case 3 - while it in hindsight appears clear that the running time would suffer more from having more columns than rows, it was a property that we did not think of as being as explicit. The FOC, that rests on that all actions are independent, would theoretically then also imply that we can execute actions in parallel, which promises larger efficiency gains if properly construed.

As Figure 8 depicts, the complexity for the worst case scenario. Each cell of the board can have 4 different states ("clear", "robot-on-top", "painted-white", "painted-black"), which turns into an exponential number of different combinations for the board. Then for each of these different combinations we are calculating the next possible combinations by trying the possible actions for the number of robots that we have.

$$O\left(\left(4^{Rows*Columns}\right)*\left(8^{No.Robots}\right)\right)$$

Figure 8: Complexity Representation

By using the hashing of states and the force ordered constraint goals we were able to downsize this complexity immensely, since we are pruning unnecessary states that would be invalid.

5.2.1 Comparison of Approach

We wanted to test the optimisation approach in order to see which one would be the most efficient. In addition, IBM ILOG CPLEX CP Optimizer is a professional solver so we wanted to compare our implementation with it. And in view of the different CPU times, we can definitely conclude that the optimisation approach is less efficient than the PDDL one. It means that the PDDL theory we saw in the course is actually useful, since a generic optimisation solver is not sufficiently efficient. Our implementation could also be optimised, we realise that our formulation is perhaps not the most efficient one.

5.2.2 Extensions and Improvement

Naturally, there is much that could improve our solution, would more time be given. First and foremost, to really experience the performance limits of the algorithm proposed, it would have been interesting to test it against more extreme cases, for example, 100 robots instead of 4 or grid size of 1000 rows and columns instead of 10. However, as our findings expounds, this was not possible to implement given the limited processing power. Also, to increase confidence of our results the next step would naturally be to run each test case in a simulator, detecting variances and extracting averages that would be more reliable as a whole.

Also, as we ourselves experienced the power of adding dynamic move-ordering constraints (in addition to restricting actions to reduce non-determinism (by removing paint-down)), we would initially like to explore how learning of previous solutions, that is for example, trying the moves that were best in the past of same state spaces, could effect the computation time. Also, it would also be interesting to add cost to actions, by adding relative weights, and examine the effects on computation time as such.

Lastly, we believe it could be extremely fascinating to implement a FCO with conditions that are not predefined, but instead discovered in response to its environment. In the above domain, all the Force Ordering Constraints exist in the initial state, but naturally in a real-world setting, there is none. For this, we would use a large set of PDDL data, and report subsequent computation time as a measure of inefficiency and then use this information (e.g. we do not use paint down and we should always prioritise painting the top most tiles) to reduce the computation time further.

6 Summary

6.1 Conclusion

In this paper, we have presented, tested and evaluated two alternative solutions for the Floor Tile problem, the first being an modified version of the classical PDDL and the second an optimisation interpretation. With the realisation that we could partially plan our search, more explicitly construct a temporal ordering of actions in a pre-determined sequential order, we gained huge efficiencies as our findings section evidences. We personally were astonished by the efficiency gains that followed the added constraints exhibited on the CPU running time, especially while taking into account the simplicity of the algorithm itself (it is merely a forward BFS adopted for multi-agents). Although we would never claim uniqueness over our findings, we also note that to our knowledge, there has been no previous attempt to empirically study the limits of this domain, using an algorithmic implementation combining forward search, hash maps and force goal constraints in a multi-agents system,in planning literature thus far. At the same time, comparing the results obtained through the Optimisation, we also realise the importance and the efficiency of using the PDDL tool in these kind of problems.

6.2 Applications and Future Work

To reiterate, why should one care about this project? In our attempt to appropriate this question, we acknowledge the necessity to explain which parts of our solution (if any) could hold relevance for real world applications.

Perhaps needless to say, practical applications in the real world can be very different to that of toy problems i.e. our Floor Tile implementation as such. For example, in the real world one typically faces ambiguous data and inconsistent constraints. Thus, it is important to acknowledge that only parts of our implementation (those with domain-independent properties) theoretically could be utilised outside of the domain.

With this in mind, we consider our FOC implementation of most value for other practical applications. As [9] explained, in the context of air defence, (more precisely optimisation of missiles), efforts have been made to adapt it to real time, extending its theoretical applicability. We consider this example as very interesting, as this evidence that our implementation holds the promise of being general purpose. As long as the domains share the same characteristics, being that they have implicit goal constraints; that sub-goals need to be completed in a fixed order to achieve the overarching goal, applicably could be considered.

For future work, we believe that there are many areas of interest. First, we would believe it could be useful to automate the generation of PDDL files, noting the the current format of PDDL files made the development of edge cases (e.g. 100x100 grid sizes) extremely time-consuming. Second, we believe that a further investigation of more advanced algorithms to better handles larger grid sizes, action spaces and an increase of the number of robots, is useful. Also, we think that implementing the same case but taking into account that the robot behaviour may be stochastic is of great interest, at least for real world applications. This could be done by using probabilities defining the movements and then analysing and comparing the efficiencies with a deterministic case.

Future research

Appendices

A PDDL Formulation

Planning tasks specified in PDDL are separated into two files; a domain file (for predicates and action) and a problem file (for objects, initial state and goal specifications). The formulation of the Floor Tile Problem in PDDL is interpreted as follows:

Init (Tile(T$_1$)$(T_2) \wedge \ldots Tile(T_n) \wedge Tile(Rob_1)Robot(Rob_2) \wedge \ldots Robot(Rob_n) \wedge Color(White) \wedge Color(Black) \wedge Color(Color) \wedge (RobotAt(Robt_1, T_3) \wedge (RobotAt(Robt_2, T_2) \wedge (AvailableColor(White) \wedge (AvailableColor(Black) \wedge (RobotHas(Rob_1, White) \wedge (RobotHas(Rob_2, Black) \wedge (Clear(T_1)(Clear(T_2) \wedge (Clear(T_3) \wedge (Clear(T_n) \wedge (Up(T_3, T_1) \wedge (Up(T_4, T_2) \wedge (Up(T_5, T_n) \wedge (Down(T_1, T_3) \wedge (Down(T_2, T_4) \wedge (Down(T_5, T_n) \wedge (Right(T_2, T_1) \wedge (Right(T_2, T_3) \wedge (Right(T_5, T_n) \wedge (Left(T_1, T_2) \wedge (Left(T_3, T_2) \wedge (Left(T_5, T_n))$

Goal (Painted(T$_1, Black) \wedge (Painted(T_2, White)(T_1, \wedge Black) \wedge (Painted(T_2, White) \wedge (Painted(T_n, Color_n))$

Action(Change-Color)

PRECOND: (RobotHas(r$_1, c_1) \wedge AvailableColor(c_1))$

EFFECT: (RobotHas(r$_1, c_1)\neg RobotHas(r_1, c_2))$

Action(Paint-Up)

PRECOND: (RobotHas(r$_1, c_1) \wedge RobotAt(r_1, t_x) \wedge Up(t_y, t_x) \wedge Clear(t_y)$

EFFECT: Clear(t$_y)\neg Painted(t_y)$

Action(Paint-Down)

PRECOND: (RobotHas(r$_1, c_1) \wedge RobotAt(r_1, t_x) \wedge Down(t_y, t_x) \wedge Clear(t_y)$

EFFECT: Clear(t$_y)\neg Painted(t_y)$

Action(Up)

PRECOND: RobotAt(r$_1, t_x) \wedge Up(ty, t_x) \wedge Clear(t_y)$

EFFECT: RobotAt($\mathsf{r}_1, t_y)\neg RobotAt(r_1, t_x) \wedge Clear(t_x)\neg Clear(t_y)\neg Painted(t_y)$

Action(Down)

PRECOND: RobotAt($\mathsf{r}_1, t_x) \wedge Down(t_y, t_x) \wedge Clear(t_y)$

EFFECT: RobotAtr$_1 r, t_y)\neg RobotAt(r_1, t_x) \wedge Clear(t_x)\neg Clear(t_y)\neg Painted(t_y)$

Action(Right)

PRECOND: RobotAt($2_1, t_x) \wedge Right(t_y, t_x) \wedge Clear(t_y)$

EFFECT: RobotAt($\mathsf{r}_1, t_y)\neg RobotAt(r_1, t_x) \wedge Clear(t_x)\neg Clear(t_y) notPainted(t_y)$

Action(Left)

PRECOND: RobotAt($\mathsf{r}_{1,x}$) $\wedge Left(t_y, t_x) \wedge Clear(t_y)$

EFFECT: RobotAt($\mathsf{r}_1, t_y)\neg RobotAt(r_1, t_x) \wedge Clear(t_x)\neg Clear(t_y)\neg Painted(t_y)$

B Optimisation problem formulation

Parameters

- n the number of rows
- m the number of columns
- $\texttt{pattern}(i, j)$ the desired color at cell (i, j)

Variables

- t^* the time needed for complete board
- $\texttt{cell}(i, j, t)$ the state of cell (i, j) at time t. 0 = not painted, 1 = painted
- $\texttt{painting}(i, j, t, r) = 1$ if robot r is painting cell (i, j) at time t
- State of the robots
  - $y(t, r)$ the vertical position of robot r at time t
  - $x(t, r)$ the horizontal position of robot r at time t
  - $\texttt{color}(t, r)$ the current color of robot r at time t. 1 = white, 0 = black
  - $\texttt{stock0}(t, r)$ the current stock of black paint of robot r at time t
  - $\texttt{stock1}(t, r)$ the current stock of white paint of robot r at time t
- Actions of the robot
  - $\texttt{paint}(t, r) = 1$ if robot r is painting at time t
  - $\texttt{move}(t, r) = 1$ if robot r is moving at time t
  - $\texttt{switch}(t, r) = 1$ if robot r is switching color at time t

$$
\begin{aligned}
\min \quad & t^* & \\
\text{such that} \quad & \sum_{i,j} \texttt{cell}(i,j,t^*) = nm & \text{Complete board}\\
& \texttt{paint}(t,r) + \texttt{move}(t,r) + \texttt{switch}(t,r) \leq 1 \quad \forall t,r & \text{One action at a time}\\
& \texttt{cell}(x(t,r), y(t,r), t) = 0 \quad \forall t,r & \text{Not stand on paint}\\
& x(t,r) = x(t,r') \Rightarrow y(t,r) \neq y(t,r') \quad \forall r, r' \neq r & \text{One robot per cell – 1}\\
& y(t,r) = y(t,r') \Rightarrow x(t,r) \neq x(t,r') \quad \forall r, r' \neq r & \text{One robot per cell – 2}\\
& \texttt{paint}(t,r) = 1 \Rightarrow \texttt{painting}(x(t,r), y(t,r)+1, t, r) & \\
& \qquad + \texttt{painting}(x(t,r), y(t,r)-1, t, r) = 1 \quad \forall t,r & \text{Painting update}\\
& \texttt{paint}(t,r) = 1 \Rightarrow \sum_{i,j} \texttt{painting}(i,j,t,r) = 1 \quad \forall t,r & \text{Painting only one cell}\\
& \texttt{paint}(t,r) = 0 \Rightarrow \texttt{painting}(i,j,t,r) = 0 \quad \forall i,j,t,r & \text{Not painting}\\
& \texttt{cell}(i,j,t) = 0 \text{ and } \sum_{r} \texttt{painting}(i,j,t,r) \geq 1 \Rightarrow \texttt{cell}(i,j,t+1) = 1 \quad \forall i,j,t & \text{Cells update}\\
& \texttt{cell}(i,j,t) = 1 \Rightarrow \texttt{cell}(i,j,t+1) = 1 \quad \forall i,j,t & \text{Stay painted}\\
& \texttt{cell}(i,j,t) = 0 \text{ and } \sum_{r} \texttt{painting}(i,j,t,r) = 0 \Rightarrow \texttt{cell}(i,j,t+1) = 0 \quad \forall i,j,t & \text{Not painted}\\
& \texttt{move}(t,r) = 1 \Rightarrow |y(t+1,r) - y(t,r)| + |x(t+1,r) - x(t,r)| = 1 \quad \forall t,r & \text{Moving}\\
& \texttt{move}(t,r) = 0 \Rightarrow |y(t+1,r) - y(t,r)| + |x(t+1,r) - x(t,r)| = 0 \quad \forall t,r & \text{Not moving}\\
& \texttt{switch}(t,r) = 1 \Rightarrow |\texttt{color}(t+1,r) - \texttt{color}(t,r)| = 1 \quad \forall t,r & \text{Switching colors}\\
& \texttt{switch}(t,r) = 0 \Rightarrow \texttt{color}(t+1,r) = \texttt{color}(t,r) \quad \forall t,r & \text{Not switching}\\
& \texttt{paint}(t,r) = 1 \text{ and } \texttt{color}(t,r) = 0 \Rightarrow \texttt{stock0}(t+1,r) = \texttt{stock0}(t,r) - 1 & \\
& \qquad \text{and } \texttt{stock1}(t+1,r) = \texttt{stock1}(t,r) \quad \forall t,r & \text{Decrement black stock}\\
& \texttt{paint}(t,r) = 1 \text{ and } \texttt{color}(t,r) = 1 \Rightarrow \texttt{stock1}(t+1,r) = \texttt{stock1}(t,r) - 1 & \\
& \qquad \text{and } \texttt{stock0}(t+1,r) = \texttt{stock0}(t,r) \quad \forall t,r & \text{Decrement white stock}\\
& \texttt{paint}(t,r) = 0 \Rightarrow \texttt{stock0}(t+1,r) = \texttt{stock0}(t,r) & \\
& \qquad \text{and } \texttt{stock0}(t+1,r) = \texttt{stock0}(t,r) \quad \forall t,r & \text{Constant stocks}\\
& \texttt{paint}(t,r) = 1 \text{ and } \texttt{color}(t,r) = 1 \Rightarrow \texttt{pattern}(x(t,r)+1, y(t,r)) & \\
& \qquad = \texttt{color}(t,r) \quad \forall t,r & \text{Respect pattern}\\
& 1 \leq y(t,r) \leq n, \quad 1 \leq x(t,r) \leq m \quad \forall t,r & \text{Stay inside the board}\\
& t^*, y(t,r), x(t,r), \texttt{stock0}(t,r), \texttt{stock1}(t,r) \in \mathbb{Z}^+ \quad \forall t,r & \text{Integer variables}\\
& \texttt{cell}(i,j,t), \texttt{painting}(i,j,t,r), \texttt{color}(t,r), \texttt{paint}(t,r), \texttt{move}(t,r), & \\
& \qquad \texttt{switch}(t,r) \in \{0,1\} \quad \forall i,j,t,r & \text{Binary variables}
\end{aligned}
$$

Note that we also need to add the initial conditions for the robots, we did it for the implementation (see section C), but we omitted them here to avoid overloading the formulation.

C AMPL code

C.1 Floortile 1D, 1 robot

You will find below the data, model and running files that we used for the optimisation of 1 robot in a 1D board.

```
# Floortile problem
# 1 robot, 1D

param n := 3; # nb rows
param T := 12; # time max
```

```
param initialStock0 = 1;
param initialStock1 = 2;
param pattern := 1 1
                 2 0
                 3 1;
```

```
# Floortile problem
# 1 robot, 1D

### PARAMETERS ###

param n; # nb rows
param T; # time max
param pattern{i in 1..n};
param initialStock0;
param initialStock1;

### VARIABLES ###

var tstar integer ≥1; # time needed for complete board
var cell{i in 1..n, t in 1..T} binary; # 0 = not painted, 1 = painted

# State of the robot
var y{t in 1..T} integer ≥ 0, ≤ n;
var color{t in 1..T} binary; # 0 = black, 1 = white
var stock0 {t in 1..T} integer ≥ 0; # stock of black color
var stock1 {t in 1..T} integer ≥ 0; # stock of white color

# Actions of the robot, 1 = doing this action
var paint{t in 1..T-1} binary;
var move{t in 1..T-1} binary;
var switch{t in 1..T-1} binary;

### OBJECTIVE ###

minimize objective: tstar;

### CONSTRAINTS ###

subject to BoardComplete:
    exists{t in 1..T} (sum{i in 1..n} cell[i,t] = n and t = tstar);

subject to OneAction {t in 1..T-1}:
    paint[t] + move[t] + switch[t] ≤ 1;

subject to NotStandOnPaint {t in 1..T}:
    y[t] ≥ 1 ==> exists{i in 1..n} (cell[i,t] = 0 and y[t] = i);

# Initial conditions
subject to InitialY:
    y[1] = 0;
subject to InitialColor:
    color[1] = 0;
subject to InitialBoard {i in 1..n}:
    cell[i,1] = 0;
subject to InitialStock0:
    stock0[1] = initialStock0;
subject to InitialStock1:
    stock1[1] = initialStock1;

# Cells update
subject to PaintingY0 {t in 1..T-1}:
    paint[t] = 1 and y[t] = 0 ==> cell[1,t+1] = 1 + cell[1,t];
subject to PaintingY1 {t in 1..T-1}:
    paint[t] = 1 and y[t] = 1 ==> cell[2,t+1] = 1 + cell[2,t];
subject to PaintingYn {t in 1..T-1}:
    paint[t] = 1 and y[t] = n ==> cell[n-1,t+1] = 1 + cell[n-1,t];
subject to PaintingYinside {t in 1..T-1}:
    paint[t] = 1 and y[t] ≥ 2 and y[t] ≤ n-1 ==>
    exists{i in 2..n-1} (cell[i-1,t+1] + cell[i+1,t+1] = 1 + cell[i-1,t] + cell[i+1,t] and i = y[t]);
subject to PaintingOthersRemainTheSame {i in 1..n, t in 1..T-1}:
    paint[t] = 1 and y[t] <> i-1 and y[t] <> i+1 ==> cell[i,t+1] = cell[i,t];
subject to NotPainting {i in 1..n, t in 1..T-1}:
    paint[t] = 0 ==> cell[i,t+1] = cell[i,t];
```

```

# Position update
subject to Moving {t in 1..T-1}:
    move[t] = 1 ==> abs(y[t+1] - y[t]) = 1;
subject to NotMovingY {t in 1..T-1}:
    move[t] = 0 ==> y[t+1] = y[t];

# Color update
subject to Switching {t in 1..T-1}:
    switch[t] = 1 ==> abs(color[t+1] - color[t]) = 1;
subject to NotSwitching {t in 1..T-1}:
    switch[t] = 0 ==> color[t+1] = color[t];

# Stock update
subject to DecrementStock0 {t in 1..T-1}:
    paint[t] = 1 and color[t] = 0 ==> stock0[t+1] = stock0[t] - 1 and stock1[t+1] = stock1[t];
subject to DecrementStock1 {t in 1..T-1}:
    paint[t] = 1 and color[t] = 1 ==> stock1[t+1] = stock1[t] - 1 and stock0[t+1] = stock0[t];
subject to StockRemainsSame {t in 1..T-1}:
    paint[t] = 0 ==> stock0[t+1] = stock0[t] and stock1[t+1] = stock1[t];

# Respect the pattern
subject to RespectPatternUp {t in 1..T-1}:
    paint[t] = 1 and y[t] ≤ n-1 ==> exists{i in 0..n-1} (color[t] = pattern[i+1] and i = y[t]);
subject to RespectPatternDown {t in 1..T-1}:
    paint[t] = 1 and y[t] = n ==> color[t] = pattern[n-1];
```

```
# Floortile problem
# 1 robot, 1D

reset;
model floortile1D.mod;
data  floortile1D.dat;

option solver ilogcp;
printf "\n** Before solve **\n\n";
solve;

printf "\n** Results**\n\n", n;
display y;
display stock0;
display stock1;
display color;
display move;
display paint;
display switch;
display cell;
display _ampl_elapsed_time;
printf "\n ** For 1D floortile, with %g cells and 1 robot : **\n", n;
printf "==> Board complete after %g time steps\n\n", tstar;
```

C.2 Floortile 2D, 1 robot

You will find below the data, model and running files that we used for the optimisation of 1 robot in a 2D board.

```
# Floortile problem
# 1 robot, 2D

param n := 2; # nb rows
param m := 2; # nb cols
param T := 12; # time max
param initialStock0 = 2;
param initialStock1 = 2;
param pattern: 1 2 :=
             1 1 0
             2 0 1;
```

```
# Floortile problem
```

```
# 1 robot, 2D

### PARAMETERS ###

param n; # nb rows
param m; # nb cols
param T; # time max
param pattern{i in 1..n, j in 1..m};
param initialStock0;
param initialStock1;

### VARIABLES ###

var tstar integer ≥1; # time needed for complete board
var cell{i in 1..n, j in 1..m, t in 1..T} binary; # 0 = not painted, 1 = painted

# State of the robot
var y{t in 1..T} integer ≥ 0, ≤ n;
var x{t in 1..T} integer ≥ 1, ≤ m;
var color{t in 1..T} binary; # 0 = black, 1 = white
var stock0 {t in 1..T} integer ≥ 0; # stock of black color
var stock1 {t in 1..T} integer ≥ 0; # stock of white color

# Actions of the robot, 1 = doing this action
var paint{t in 1..T-1} binary;
var move{t in 1..T-1} binary;
var switch{t in 1..T-1} binary;

### OBJECTIVE ###

minimize cost: tstar;

### CONSTRAINTS ###

subject to BoardComplete:
    exists{t in 1..T} (sum{i in 1..n, j in 1..m} cell[i,j,t] = n*m and t = tstar);

subject to OneAction {t in 1..T-1}:
    paint[t] + move[t] + switch[t] ≤ 1;

subject to NotStandOnPaint {t in 1..T}:
    y[t] ≥ 1 ==> exists{i in 1..n, j in 1..m} (cell[i,j,t] = 0 and y[t] = i and x[t] = j);

# Initial conditions
subject to InitialY:
    y[1] = 0;
subject to InitialX:
    x[1] = 1;
subject to InitialColor:
    color[1] = 0;
subject to InitialBoard {i in 1..n, j in 1..m}:
    cell[i,j,1] = 0;
subject to InitialStock0:
    stock0[1] = initialStock0;
subject to InitialStock1:
    stock1[1] = initialStock1;

# Cells update
subject to Painting01 {t in 1..T-1}:
    paint[t] = 1 and y[t] = 0 and x[t] = 1 ==> cell[1,1,t+1] = 1 + cell[1,1,t];
subject to Painting02 {t in 1..T-1}:
    paint[t] = 1 and y[t] = 0 and x[t] = 2 ==> cell[1,2,t+1] = 1 + cell[1,2,t];
subject to Painting11 {t in 1..T-1}:
    paint[t] = 1 and y[t] = 1 and x[t] = 1 ==> cell[2,1,t+1] = 1 + cell[2,1,t];
subject to Painting12 {t in 1..T-1}:
    paint[t] = 1 and y[t] = 1 and x[t] = 2 ==> cell[2,2,t+1] = 1 + cell[2,2,t];
subject to Painting21 {t in 1..T-1}:
    paint[t] = 1 and y[t] = 2 and x[t] = 1 ==> cell[1,1,t+1] = 1 + cell[1,1,t];
subject to Painting22 {t in 1..T-1}:
    paint[t] = 1 and y[t] = 2 and x[t] = 2 ==> cell[1,2,t+1] = 1 + cell[1,2,t];
subject to PaintingOthersRemainTheSame {i in 1..n, j in 1..m, t in 1..T-1}:
    paint[t] = 1 and ((y[t] <> i-1 and y[t] <> i+1) or x[t] <> j) ==> cell[i,j,t+1] = cell[i,j,t];
subject to NotPainting {i in 1..n, j in 1..m, t in 1..T-1}:
    paint[t] = 0 ==> cell[i,j,t+1] = cell[i,j,t];

# Position update
```

```
subject to Moving {t in 1..T-1}:
    move[t] = 1 ==> abs(y[t+1] - y[t]) + abs(x[t+1] - x[t]) = 1;
subject to NotMoving {t in 1..T-1}:
    move[t] = 0 ==> y[t+1] = y[t] and x[t+1] = x[t];

# Color update
subject to Switching {t in 1..T-1}:
    switch[t] = 1 ==> abs(color[t+1] - color[t]) = 1;
subject to NotSwitching {t in 1..T-1}:
    switch[t] = 0 ==> color[t+1] = color[t];

# Stock update
subject to DecrementStock0 {t in 1..T-1}:
    paint[t] = 1 and color[t] = 0 ==> stock0[t+1] = stock0[t] - 1 and stock1[t+1] = stock1[t];
subject to DecrementStock1 {t in 1..T-1}:
    paint[t] = 1 and color[t] = 1 ==> stock1[t+1] = stock1[t] - 1 and stock0[t+1] = stock0[t];
subject to StockRemainsSame {t in 1..T-1}:
    paint[t] = 0 ==> stock0[t+1] = stock0[t] and stock1[t+1] = stock1[t];

# Respect the pattern
subject to RespectPatternUp {t in 1..T-1}:
    paint[t] = 1 and y[t] <= n-1 ==>
    exists{i in 0..n-1, j in 1..m} (color[t] = pattern[i+1,j] and i = y[t] and j = x[t]);
subject to RespectPatternDown {t in 1..T-1}:
    paint[t] = 1 and y[t] = n ==> exists{j in 1..m} (color[t] = pattern[n-1,j] and j = x[t]);
```

```
# Floortile problem
# 1 robot, 2D

reset;
model floortile2D.mod;
data  floortile2D.dat;

option solver ilogcp;
printf "\n** Before solve **\n\n";
solve;

printf "\n** Results**\n\n", n;
display x;
display y;
display stock0;
display stock1;
display color;
display move;
display paint;
display switch;
display cell;
display _ampl_elapsed_time;
printf "\n ** For 2D floortile, with %g x %g cells and 1 robot : **\n", n, m;
printf "==> Board complete after %g time steps\n\n", tstar;
```

C.3 Floortile 2D, 2 robots

You will find below the data, model and running files that we used for the optimisation of 2 robots in a 2D board.

```
# Floortile problem
# 2 robots, 2D

param n := 2; # nb rows
param m := 2; # nb cols
param T := 7; # time max
param R := 2; # nb robots
param initialStock0 := 1 2
                       2 2;
param initialStock1 := 1 2
                       2 2;
param pattern: 1 2 :=
            1 1 0
            2 0 1;
```

```
# Floortile problem
# 2 robots, 2D

### PARAMETERS ###

param n; # nb rows
param m; # nb cols
param T; # time max
param R; # nb robots
param pattern{i in 1..n, j in 1..m};
param initialStock0{r in 1..R};
param initialStock1{r in 1..R};

### VARIABLES ###

var tstar integer ≥1; # time needed for complete board
var cell{i in 1..n, j in 1..m, t in 1..T} binary; # 0 = not painted, 1 = painted
var painting{i in 1..n, j in 1..m, t in 1..T, r in 1..R} binary; # 1 = r is painting (i,j) at time t

# State of the robot
var y{t in 1..T, r in 1..R} integer ≥ 0, ≤ n;
var x{t in 1..T, r in 1..R} integer ≥ 1, ≤ m;
var color{t in 1..T, r in 1..R} binary; # 0 = black, 1 = white
var stock0 {t in 1..T, r in 1..R} integer ≥ 0; # stock of black color
var stock1 {t in 1..T, r in 1..R} integer ≥ 0; # stock of white color

# Actions of the robot, 1 = doing this action
var paint{t in 1..T-1, r in 1..R} binary;
var move{t in 1..T-1, r in 1..R} binary;
var switch{t in 1..T-1, r in 1..R} binary;

### OBJECTIVE ###

minimize objective: tstar - sum{i in 1..n, j in 1..m, t in 1..T-1, r in 1..R} painting[i,j,t,r];

### CONSTRAINTS ###

subject to BoardComplete:
    exists{t in 1..T} (sum{i in 1..n, j in 1..m} cell[i,j,t] = n*m and t = tstar);

subject to OneAction {t in 1..T-1, r in 1..R}:
    paint[t,r] + move[t,r] + switch[t,r] ≤ 1;

subject to NotStandOnPaint {t in 1..T, r in 1..R}:
    y[t,r] ≥ 1 ==> exists{i in 1..n, j in 1..m} (cell[i,j,t] = 0 and y[t,r] = i and x[t,r] = j);

subject to OneRobotPerCellX {t in 1..T}:
    x[t,1] = x[t,2] ==> y[t,1] <> y[t,2];
subject to OneRobotPerCellY {t in 1..T}:
    y[t,1] = y[t,2] ==> x[t,1] <> x[t,2];

# Initial conditions
subject to InitialY:
    y[1,1] = 0 and y[1,2] = 0;
subject to InitialX:
    x[1,1] = 1 and x[1,2] = 2;
subject to InitialColor{r in 1..R}:
    color[1,r] = 0;
subject to InitialBoard {i in 1..n, j in 1..m}:
    cell[i,j,1] = 0;
subject to InitialStock0{r in 1..R}:
    stock0[1,r] = initialStock0[r];
subject to InitialStock1{r in 1..R}:
    stock1[1,r] = initialStock1[r];

# Paintings
subject to Painting01 {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and y[t,r] = 0 and x[t,r] = 1 ==> painting[1,1,t,r] = 1;
subject to Painting02 {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and y[t,r] = 0 and x[t,r] = 2 ==> painting[1,2,t,r] = 1;
subject to Painting11 {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and y[t,r] = 1 and x[t,r] = 1 ==> painting[2,1,t,r] = 1;
subject to Painting12 {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and y[t,r] = 1 and x[t,r] = 2 ==> painting[2,2,t,r] = 1;
```

```
subject to Painting21 {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and y[t,r] = 2 and x[t,r] = 1 ==> painting[1,1,t,r] = 1;
subject to Painting22 {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and y[t,r] = 2 and x[t,r] = 2 ==> painting[1,2,t,r] = 1;
subject to PaintOnlyOne {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 ==> sum{i in 1..n, j in 1..m} painting[i,j,t,r] = 1;
subject to NotPainting {i in 1..n, j in 1..m, t in 1..T-1, r in 1..R}:
    paint[t,r] = 0 ==> painting[i,j,t,r] = 0;

# Cells update
subject to UpdateCells {i in 1..n, j in 1..m, t in 1..T-1}:
    cell[i,j,t] = 0 and sum{r in 1..R} painting[i,j,t,r] ≥ 1 ==> cell[i,j,t+1] = 1;
subject to NotUpdateCells {i in 1..n, j in 1..m, t in 1..T-1}:
    cell[i,j,t] = 0 and sum{r in 1..R} painting[i,j,t,r] = 0 ==> cell[i,j,t+1] = 0;

# Position update
subject to Moving {t in 1..T-1, r in 1..R}:
    move[t,r] = 1 ==> abs(y[t+1,r] - y[t,r]) + abs(x[t+1,r] - x[t,r]) = 1;
subject to NotMoving {t in 1..T-1, r in 1..R}:
    move[t,r] = 0 ==> y[t+1,r] = y[t,r] and x[t+1,r] = x[t,r];

# Color update
subject to Switching {t in 1..T-1, r in 1..R}:
    switch[t,r] = 1 ==> abs(color[t+1,r] - color[t,r]) = 1;
subject to NotSwitching {t in 1..T-1, r in 1..R}:
    switch[t,r] = 0 ==> color[t+1,r] = color[t,r];

# Stock update
subject to DecrementStock0 {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and color[t,r] = 0 ==> stock0[t+1,r] = stock0[t,r] - 1 and stock1[t+1,r] = ...
        stock1[t,r];
subject to DecrementStock1 {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and color[t,r] = 1 ==> stock1[t+1,r] = stock1[t,r] - 1 and stock0[t+1,r] = ...
        stock0[t,r];
subject to StockRemainsSame {t in 1..T-1, r in 1..R}:
    paint[t,r] = 0 ==> stock0[t+1,r] = stock0[t,r] and stock1[t+1,r] = stock1[t,r];

# Respect the pattern
subject to RespectPattern {t in 1..T-1, r in 1..R}:
    paint[t,r] = 1 and y[t,r] ≤ n-1 ==>
    exists{i in 0..n-1, j in 1..m} (color[t,r] = pattern[i+1,j] and i = y[t,r] and j = x[t,r]);
```

```
# Floortile problem
# 2 robots, 2D

reset;
model floortile2D2.mod;
data  floortile2D2.dat;

option solver ilogcp;
printf "\n** Before solve **\n\n";
solve;

printf "\n** Results**\n\n", n;
display x;
display y;
display stock0;
display stock1;
display color;
display move;
display paint;
display switch;
display cell;
display _ampl_elapsed_time;
printf "\n ** For 2D floortile, with %g x %g cells and 2 robots : **\n", n, m;
printf "==> Board complete after %g time steps\n\n", tstar;
```

D PDDL code

D.1 Main.py

```
"""
DD2380 ai17 HT17-2 : (Artificial Intelligence) Floortile planning project
File: main.py (Main file)
Authors: Antonie Legat, Anna Hedstr m, Sandra P ic , David Vega
12th October 2017
"""
from copy import copy, deepcopy
import datetime as time
import numpy as np
import parse as p
import Interface as gui

robot1 = []
robot2 = []
robots =[]
state = []
target = []
columns = 0
rows = 0


#Path as a global variable
path = []

# Sequence
sequenceStates = []

# Robots
sequenceRobots =[]

# Keeping track during DP
sequenceMovements = []

## trying all the possible combinations between N robots
def getSequence(robots,index,currState,seqMov,seqRobots,target):
    if(index == len(robots)):
        global sequenceStates,sequenceRobots,sequenceMovements
        sequenceMovements.append(seqMov)
        sequenceStates.append(currState)
        sequenceRobots.append(seqRobots)
        return
    possibles,movement = getPossiblesFOC_ClearCells(robots[index],currState,target)
    for i in range(len(possibles)):
        auxMov = deepcopy(seqMov)
        auxMov.append("Robot"+str(index+1) +": " +str(movement[i]))
        auxRobots= deepcopy(seqRobots)
        auxRobots.append(possibles[i][0])
        getSequence(robots, index+1, possibles[i][1],auxMov,auxRobots,target)
    return

## this functions checks all the possible actions that we can take for a given robot and an state
## target it's our goal, we want to paint with the colors that we should
def moveRobot(robot, dx, dy):
    return [[robot[0][0] + dx, robot[0][1] + dy], robot[1], robot[2], robot[3]]

def removePaint(robot, d1, d2):
    return [robot[0], robot[1], robot[2] - d1, robot[3] - d2]

def changePaint(robot, change):
    return [robot[0], change, robot[2], robot[3]]


""" Generate the next possibles states taking into account the preconditions and also the Forced ...
    Ordering Constraints.
    In this case, this function is used when the pattern has some clear cells in the target.
    @robot - robot information defined as: [ [x,y], color, color1Remaining, color2Remaining]
    @state - matrix that defines the current state configuration
    @target - matrix that defines the target configuration
    """
```

```
def getPossiblesFOC_ClearCells(robot, state, target):

    s = []
    states = []

    #If the robot wants to move in an existing position and the position of this particular state ...
        is clear..
    if ((robot[0][1] + 1) < len(state) and state[robot[0][1] + 1][robot[0][0]] == 0):
        row = robot[0][1]
        column = robot[0][0]
        painted = True
        #Priorizing
        #Forced ordering constraints.
        if (row != 0):
            for i in range(0,row):
                for j in range(0, columns):
                    #If there is non-painted cell
                    if ((state[i][j] != 2) and (state[i][j] != 1)):
                        #If is really need to be painted...
                        if((target[i][j] == 2) or (target[i][j] == 1)):
                            painted = False
        if (painted == True):
            s.append("down")
            aux = deepcopy(state)
            aux[robot[0][1]][robot[0][0]] = 0
            aux[robot[0][1] + 1][robot[0][0]] = 3
            states.append([moveRobot(robot, 0, 1), aux])

    #If the robot wants to move in an existing position and the position of this particular state ...
        is clear..
    if ((robot[0][1] - 1) ≥ 0 and state[robot[0][1] - 1][robot[0][0]] == 0):
        s.append("up")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1] - 1][robot[0][0]] = 3
        states.append([moveRobot(robot, 0, -1), aux])

    #If the robot wants to move in an existing position and the position of this particular state ...
        is clear..
    if ((robot[0][0] + 1) < len(state[0]) and state[robot[0][1]][robot[0][0] + 1] == 0):
        s.append("right")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1]][robot[0][0] + 1] = 3
        states.append([moveRobot(robot, 1, 0), aux])

     #If the robot wants to move in an existing position and the position of this particular ...
         state is clear..
    if ((robot[0][0] - 1) ≥ 0 and state[robot[0][1]][robot[0][0] - 1] == 0):
        s.append("left")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1]][robot[0][0] - 1] = 3
        states.append([moveRobot(robot, -1, 0), aux])

    ## if its possible to move up / we have paint / we need to paint it like that / and the "up"- ...
        row is already painted..
    if (("up") in s and robot[1 + robot[1]] > 0 and target[robot[0][1] - 1][robot[0][0]] == ...
        robot[1]):
        #Where you want to paint
        row_robot = robot[0][1]-1
        column_robot = robot[0][0]
        painted = True
        if (row_robot != 0):
            for i in range(0,row_robot):
                for j in range(0,columns):
                    if ((state[i][j] != 2) and (state[i][j] != 1)):
                        #And its really needed to paint there..
                         if((target[i][j] == 2) or (target[i][j] == 1)):
                            painted = False
        if (painted == True):
            s.append("paint_up")
            aux = deepcopy(state)
            aux[robot[0][1] - 1][robot[0][0]] = robot[1]
            d1 = d2 = 0
            if (robot[1] == 1):
```

```
                d1 = 1
            else:
                d2 = 1
            states.append([removePaint(robot, d1, d2), aux])
    ## todo append SAT for painting
    if(robot[ robot[1] +  1 ] >0):
        if(robot[1] == 1):
            auxR = changePaint(robot,2)
            s.append("change_paint_black")
        else:
            auxR = changePaint(robot,1)
            s.append("change_paint_white")
        states.append([auxR, state])
    s.append("still")
    states.append( [robot, state])
    return states, s


""" Generate the next possibles states taking into account the preconditions and also the Forced ...
    Ordering Constraints.
    In this case, the algorithm is much more effected because a lot of braches are prunned. The ...
        action paint_down is not considered.
    @robot - robot information defined as: [ [x,y], color, color1Remaining, color2Remaining]
    @state - matrix that defines the current state configuration
    @target - matrix that defines the target configuration
    """
def getPossiblesFOC(robot, state, target):
    s = []
    states = []

    #If its possible to go down...
    if ((robot[0][1] + 1) < len(state) and state[robot[0][1] + 1][robot[0][0]] == 0):
        row = robot[0][1]
        column = robot[0][0]
        painted = True
        #Check if you are not in the top row and if you have tiles to paint up you.
        if (row != 0):
            for i in range(0,row):
                for j in range(0, columns):
                    if ((state[i][j] != 2) and (state[i][j] != 1)):
                        painted = False

        #You can only go down if you don't have any non-painted tile up to you. (Forced Ordering ...
            Constraints)
        if (painted == True):
            s.append("down")
            aux = deepcopy(state)
            aux[robot[0][1]][robot[0][0]] = 0
            aux[robot[0][1] + 1][robot[0][0]] = 3
            states.append([moveRobot(robot, 0, 1), aux])

    if ((robot[0][1] - 1) ≥ 0 and state[robot[0][1] - 1][robot[0][0]] == 0):
        s.append("up")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1] - 1][robot[0][0]] = 3
        states.append([moveRobot(robot, 0, -1), aux])

    if ((robot[0][0] + 1) < len(state[0]) and state[robot[0][1]][robot[0][0] + 1] == 0):
        s.append("right")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1]][robot[0][0] + 1] = 3
        states.append([moveRobot(robot, 1, 0), aux])

    if ((robot[0][0] - 1) ≥ 0 and state[robot[0][1]][robot[0][0] - 1] == 0):
        s.append("left")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1]][robot[0][0] - 1] = 3
        states.append([moveRobot(robot, -1, 0), aux])

    ## if up is clear / we have paint / we need to paint it like that / and the "up"- row is ...
        already painted..
    if (("up") in s and robot[1 + robot[1]] > 0 and target[robot[0][1] - 1][robot[0][0]] == ...
        robot[1]):
```

```
        #Where you want to paint
        row_robot = robot[0][1]-1
        column_robot = robot[0][0]
        painted = True
        #If you want to paint, check if there is any other tile non-painted up to you.
        if (row_robot != 0):
            for i in range(0,row_robot):
                for j in range(0,columns):
                    if ((state[i][j] != 2) and (state[i][j] != 1)):
                        painted = False
        if (painted == True):
            s.append("paint_up")
            aux = deepcopy(state)
            aux[robot[0][1] - 1][robot[0][0]] = robot[1]
            d1 = d2 = 0
            if (robot[1] == 1):
                d1 = 1
            else:
                d2 = 1
            states.append([removePaint(robot, d1, d2), aux])

    #Change color if available.
    if(robot[ robot[1] +  1 ] >0):
        if(robot[1] == 1):
            auxR = changePaint(robot,2)
            s.append("change_paint_black")
        else:
            auxR = changePaint(robot,1)
            s.append("change_paint_white")
        states.append([auxR, state])
    s.append("still")
    states.append( [robot, state])
    return states, s



""" Generate the next possibles states taking into account the preconditions.
    Generate all the possibles nextStates without Forced Ordering Constraints
    @robot - robot information defined as: [ [x,y], color, color1Remaining, color2Remaining]
    @state - matrix that defines the current state configuration
    @target - matrix that defines the target configuration
    """
def getPossibles(robot, state, target):

    s = []
    states = []


    """
    ## For actions: down, up, right, left, can be considered as a possible action if:
    ##   The adjacent cell exists (where you want to move) and if it's clear ( no robot, no ...
        painted )
    ##         exists -> it's in the bounds of the board
    """
    if ((robot[0][1] + 1) < len(state) and state[robot[0][1] + 1][robot[0][0]] == 0):
        s.append("down")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1] + 1][robot[0][0]] = 3
        states.append([moveRobot(robot, 0, 1), aux])

    if ((robot[0][1] - 1) ≥ 0 and state[robot[0][1] - 1][robot[0][0]] == 0):
        s.append("up")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1] - 1][robot[0][0]] = 3
        states.append([moveRobot(robot, 0, -1), aux])

    if ((robot[0][0] + 1) < len(state[0]) and state[robot[0][1]][robot[0][0] + 1] == 0):
        s.append("right")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1]][robot[0][0] + 1] = 3
        states.append([moveRobot(robot, 1, 0), aux])

    if ((robot[0][0] - 1) ≥ 0 and state[robot[0][1]][robot[0][0] - 1] == 0):
```

```
        s.append("left")
        aux = deepcopy(state)
        aux[robot[0][1]][robot[0][0]] = 0
        aux[robot[0][1]][robot[0][0] - 1] = 3
        states.append([moveRobot(robot, -1, 0), aux])


    ## To execute paint_up, the preconditions that must happen are:
    ## if up is clear( we can execute up action) / we have enough amount of paint / we need to ...
        paint it like that (as target defines)
    if (("up") in s and robot[1 + robot[1]] > 0 and target[robot[0][1] - 1][robot[0][0]] == ...
        robot[1]):
        s.append("paint_up")
        aux = deepcopy(state)
        aux[robot[0][1] - 1][robot[0][0]] = robot[1]
        d1 = d2 = 0
        if (robot[1] == 1):
            d1 = 1
        else:
            d2 = 1
        states.append([removePaint(robot, d1, d2), aux])

    if (("down") in s and robot[1 + robot[1]] > 0 and target[robot[0][1] + 1][robot[0][0]] == ...
        robot[1]):
        s.append("paint_down")
        aux = deepcopy(state)
        aux[robot[0][1] + 1][robot[0][0]] = robot[1]
        d1 = d2 = 0
        if (robot[1] == 1):
            d1 = 1
        else:
            d2 = 1
        states.append([removePaint(robot, d1, d2), aux])

    ## If the robot wants to paint , we need to check that is possible ( enough amount of paint )
    if(robot[ robot[1] +  1 ] >0):
        if(robot[1] == 1):
            auxR = changePaint(robot,2)
            s.append("change_paint_black")
        else:
            auxR = changePaint(robot,1)
            s.append("change_paint_white")
        states.append([auxR, state])

    s.append("still")
    states.append( [robot, state])
    return states, s

""" Forward Search algorithm to solve the problem
    This is the first version that we did to solve the problem. It only works with 2 robots.
    @how_many_robots - integer that define how many robots we have in the state.
    @robots - list of robots defined as [ [x,y], color, color1Remaining, color2Remaining]
    @iniState - matrix that defines the initial configuration
    @target - matrix that defines the target configuration
    """
def solve_2robots(how_many_robots,robots, iniState, target):
    # using list as queue and dictionary as hashmap
    q = []
    visited = {}
    # initializing the queue for the BFS
    # the state it's represented as current state of robots
    # current state of the board
    # and the list of movements
    q.append([robots, iniState, []])
    # Using np arrays because of comparation function between matrices
    targetCheck = np.array(target)
    # flag to check if we were able to achieve the target
    done = False;
    current = None
    robotPossibles = []
    robotMovement = []
    index  = []
    for i in range(0,how_many_robots):
        element = []
        num = 0
        robotPossibles.append(element)
```

```
        index.append(num)
        robotMovement.append(element)

    while q:
        current = q[0]
        q.pop(0)
        currentCheck = np.array(current[1])
        # converting the state to string for hashing
        keyState = str(current[1])
        KeyRobots = str(current[0])
        # checking if we have been in this state before
        value1 = visited.get(keyState)
        value2 = visited.get(KeyRobots)
        if value1!= None:
            continue
        # marking and hashing state
        visited[keyState] = True
        visited[KeyRobots] = True

        # Checking if our current board it's our target
        currentCheck[currentCheck == 3] = 0
        if (currentCheck == targetCheck).all():
            print(currentCheck)
            print(current[2])
            global path
            path = current[2]
            done = True
            break;

        # Get possibles for robot1 then try those as current States for robot2 and so on
        # Only working with 2 robots.
        robot1Possibles, movement1 = getPossiblesFOC_2(current[0][0], current[1], target)
        index1 = 0
        for possible1 in robot1Possibles:
            robot2Possibles, movement2 = getPossiblesFOC_2(current[0][1], possible1[1], target)
            index2 = 0
            for possible2 in robot2Possibles:
                sequence = deepcopy(current[2])
                sequence.append("Robot1: " + movement1[index1])
                sequence.append("Robot2: " + movement2[index2])
                index2 += 1
                q.append([[possible1[0], possible2[0]], possible2[1], sequence])
            index1 += 1
    return done


""" Forward Search algorithm to solve the problem
    @how_many_robots - integer that define how many robots we have in the state.
    @robots - list of robots defined as [ [x,y], color, color1Remaining, color2Remaining]
    @iniState - matrix that defines the initial configuration
    @target - matrix that defines the target configuration
    """
def solve_NRobots(how_many_robots, robots, iniState, target):

    # using list as queue and dictionary as hashmap
    q = []
    visited = {}
    # initializing the queue for the BFS
    # the state it's represented as current state of robots
    # current state of the board
    # and the list of movements
    q.append([robots, iniState, []])
    # Using np arrays because of comparation function between matrices
    targetCheck = np.array(target)
    # flag to check if we were able to achieve the target
    done = False;
    current = None
    robotPossibles = []
    robotMovement = []
    index  = []
    for i in range(0,how_many_robots):
        element = []
        num = 0
        robotPossibles.append(element)
        index.append(num)
        robotMovement.append(element)
```

```
    while q:
        current = q[0]
        q.pop(0)
        currentCheck = np.array(current[1])
        ### converting the state to string for hashing
        keyState = str(current[1])
        KeyRobots = str(current[0])
        ### checking if we have been in this state before
        value1 = visited.get(keyState)
        value2 = visited.get(KeyRobots)
        if value1!= None:
            continue
        # marking and hashing state
        visited[keyState] = True
        visited[KeyRobots] = True

        # Checking if our current board it's our target
        currentCheck[currentCheck == 3] = 0
        if (currentCheck == targetCheck).all():
            print(currentCheck)
            print(current[2])
            global path
            path = current[2]
            done = True
            break;

        global sequenceRobots,sequenceStates,sequenceMovements
        # Sequence
        sequenceStates = []
        # Robots
        sequenceRobots =[]
        # Keeping track during DP
        sequenceMovements = []

        # Get possibles for all the robots.
        # This algorithm works with N robots.
        getSequence(current[0],0,current[1],[],[],target)
        for i in range(len(sequenceStates)):
            aux = deepcopy(current[2])
            aux.extend(sequenceMovements[i])
            q.append( [sequenceRobots[i], sequenceStates[i], aux]     )
    return done


""" Main function of the program:
    - Take the information of the pddl.
    - Solve the planning problem.
    - Send the path into the graphics.
    - Print the cpu time needed to find the optimal path.
    """
if __name__ == '__main__':

    #Actual time
    a = time.datetime.now()
    #Information from the PDDL file using parse.py. Robots information, the initial state and the ...
        target state.
    robots,state,target = p.main()
    how_many_robots = len(robots)
    for i in range(0,how_many_robots):
        robots[i][1] +=1
        robots[i][0][0] -=1
    rows = len(state)
    columns = len(state[0])
    solution = False
    #Solve and find the proper path
    solution = solve_NRobots(how_many_robots,robots, state, target)
    #Time needed
    print("Time: ")
    print(time.datetime.now() - a)
    if (solution == True):
        #We found a path. Needed to draw it.
        print("Solved")
        #Draw and simulate the path.
        gui.Draw(robots,state,path)
    else:
        #No path found.
```

```
        print("To solve")
```

D.2 Parse.py

```
"""
DD2380 ai17 HT17-2 : (Artificial Intelligence) Floortile planning project
File: parse.py (Parsing information between pddl file and main file)
Authors: Antonie Legat, Anna Hedstr m, Sandra Pic , David Vega
12th October 2017
"""

import pddlpy

#All the pddl files that you can choose
Algorithms_Problems = ['algo_case_1.pddl','algo_case_2.pddl']
Size_Problems = ['size_case_1.pddl', 'size_case_2.pddl','size_case_3.pddl','size_case_4.pddl', ...
    'size_case_5.pddl','size_case_6.pddl', 'size_case_7.pddl', 'size_case_8.pddl', ...
    'size_case_9.pddl','size_case_10.pddl']
Pattern_Problems = ['pattern_case_1.pddl', 'pattern_case_2.pddl']
Robots_Problems = ['robots_case_1.pddl', 'robots_case_2.pddl', 'robots_case_3.pddl']


""" Parsing the information of the pddl file using the pddlpy library.
    Return the information of the pddl file (robots information, initial state and target state)
    """
def main():

    #All the robots will have 1000 availability of black color and white color.
    amount = 1000
    #Reading from the Domain and problem file.
    domprob = pddlpy.DomainProblem('Domain.pddl', Size_Problems[5])
    initRob = []
    initState = []
    targetState = []
    max_robot = -300
    #Checking how many robots we have in the pddl file
    for o in domprob.initialstate():
        if ((str(o)).split(',')[0] == "('robot-at'"):
            robot = ((str(o)).split(',')[1])[7]
            Row =(((str(o)).split(',')[2]).split('-')[0]).split('_')[1]
            Column = (((str(o)).split(',')[2]).split('-')[1]).split("'")[0]
            element = [(int(Column)),(int(Row))]

            #In order to know how many robots we have
            if (int(robot) > max_robot):
                max_robot = int(robot)

    #Create a list with all the information from the robots.
    for i in range(0,max_robot):
        element = []
        initRob.append(None)

    #Look at the initial positions of the robot.
    #Update the robots list.
    for o in domprob.initialstate():
        if ((str(o)).split(',')[0] == "('robot-at'"):
            robot = ((str(o)).split(',')[1])[7]
            Row =(((str(o)).split(',')[2]).split('-')[0]).split('_')[1]
            Column = (((str(o)).split(',')[2]).split('-')[1]).split("'")[0]
            element = [(int(Column)),(int(Row))]
            element2 = []
            element2.append(element)
            element2.append(0)
            element2.append(amount)
            element2.append(amount)
            initRob[int(robot)-1] = element2

    #Look at which color has the robot.
    for o in domprob.initialstate():
        if ((str(o)).split(',')[0] == "('robot-has'"):
            robot = ((str(o)).split(',')[1])[7]
```

```
            color = ((str(o)).split(',')[2])
            if(str(color) == " 'black')"):
                initRob[int(robot)-1][1]= 0
            else:
                initRob[int(robot)-1][1]= 1

    max_column= -3000
    max_rows= -3000
    #How many columns and how many rows we have in this pddl file.
    for o in domprob.worldobjects():
        if ('tile') in o:
            x,column =  (str(o)).split('-')
            if (int(column)≥ int(max_column)):
                max_column = column
            z,row = (str(x)).split('_')
            if(int(row)≥ int(max_rows)+1):
                max_rows = row

    # Create the initial and the target state.
    for i in range((int(max_rows)) +1):
        list = []
        list2= []
        for j in range(int(max_column)):
            list.append(0)
            list2.append(0)
        initState.append(list)
        targetState.append(list2)

    #Update the initState with the init robot positions.
    for i in range(0,max_robot):
        row_robot, column_robot = initRob[i][0]
        initState[column_robot][row_robot-1] = 3

    #Update the target state using the goals defined in the pddl file.
    for g in domprob.goals():
        goal_row = (((str(g)).split(',')[1]).split('-')[0]).split('_')[1]
        goal_column = (((str(g)).split(',')[1]).split('-')[1]).split("'")[0]
        goal_color = ((str(g)).split(',')[2])
        if (str(goal_color) == " 'black')"):
            num_color = 2
        else:
            num_color = 1

        targetState[int(goal_row)-1][int(goal_column)-1] = int(num_color)

    #return the robot information, the init state information and also the target State information
    return initRob, initState, targetState
```

D.3 Interface.py

```
"""
DD2380 ai17 HT17-2 : (Artificial Intelligence) Floortile planning project
File: Interface.py (Create the graphics )
Authors: Antonie Legat, Anna Hedstr m, Sandra P i c , David Vega
12th October 2017
"""
import turtle

#Definition of robot's path color
#The maximum case that we tried is with 5 robots.
robot_path = ["orange","blue","pink","red","yellow"]
#Which color has de robot
robot_has_color = ["black","white","black","white","black"]

#Coordinates offset.
#(-200,300)
offsetX = -200
offsetY = -300


""" Update the information in order to be able to print the main grid.
```

```
    @robots - robot information defined as: [ [x,y], color, color1Remaining, color2Remaining]
    @initialState - matrix that defines the initial state configuration
    @how_many_robots - matrix that defines the target configuration
    """
def setInitialData(robots,initialState,how_many_robots):

    #How many rows and columns we have
    total_rows = len(initialState)
    total_columns = len(initialState[0])

    robot_column = []
    robot_row = []

    for i in range(0,how_many_robots):
        robot_row.append(0)
        robot_column.append(0)

    #Update the information for the robots
    for i in range(0,how_many_robots):
        robot_row[i] = robots[i][0][1]
        robot_column[i] = robots[i][0][0]
        color = robots[i][1]
        if (color == 1):
            robot_has_color[i] = "white"
        else:
            robot_has_color[i] = "black"

    return robot_row, robot_column,total_rows,total_columns


""" Adapt the path received from the main file to a list that the graphs can interpret.
    @data - path received from the solver algorithm.
    @robot_row , robot_column, robot_has_color: initial configurations of the robots.
    @total rows, total columns : to know how many data we need.
    """
def GenerateData(data,robot_row,robot_column,robot_has_color,total_rows,total_columns):

    """ This function returns the information path defined as:
        [Action, parameter1, parameter2, parameter3, parameter4, parameter5]
        Action {0 = Change-color, 1 = Paint-up, 2 = Paint-down, 3 = Up, 4 = Down, 5 = Right, 6 = ...
            Left, 7 = Stop}"
        Parameter1: {Number of the robot: 1, 2, 3,...N}
        The rest of the parameters depend on the action:

        Change-Color: [0,robot,color,0,0,0] - Color{0 = black, 1 = white}
        Paint-up = [1,robot,row,column, 0,0,0]
        Paint-down = [2,robot,row,column,0,0,0]
        Up = [3,robot,row_initial,column_initial,row_final,column_final]
        Down = [4,robot,row_initial,column_initial,row_final,column_final]
        Right = [5,robot,row_initial,column_initial,row_final,column_final]
        Left = [6,robot,row_initial,column_initial,row_final,column_final]
        Stop = [7,robot,row,column,0,0,0]
    """

    list = []
    #Analyzing all the information.
    for i in data:
        aux = []
        robot, action = str(i).split(':')
        r = str(robot).split(' ')
        a= str(action).split()
        n_robot = 0
        if (str(robot) == 'Robot1'):
            n_robot = 1
        elif (str(robot) == 'Robot2'):
            n_robot = 2
        elif (str(robot) == 'Robot3'):
            n_robot = 3
        elif (str(robot) == 'Robot4'):
            n_robot = 4
        elif (str(robot) == 'Robot5'):
            n_robot = 5
        if (str(action) == ' paint_up'):
            aux.append(1)
            aux.append(n_robot)
            row = robot_row[n_robot-1] -1
```

```
            column = robot_column[n_robot-1]
            aux.append(row)
            aux.append(column)
            aux.append(0)
            aux.append(0)
            aux.append(0)
        elif (str(action) == ' paint_down'):
            aux.append(2)
            aux.append(n_robot)
            row = robot_row[n_robot] + 1
            column = robot_column[n_robot]
            aux.append(row)
            aux.append(column)
            aux.append(0)
            aux.append(0)
            aux.append(0)
        elif (str(action) == ' up'):
            aux.append(3)
            aux.append(n_robot)
            row = robot_row[n_robot-1]
            column = robot_column[n_robot-1]
            aux.append(row)
            aux.append(column)
            robot_row[n_robot-1] = robot_row[n_robot-1] -1
            row = robot_row[n_robot-1]
            aux.append(row)
            aux.append(column)
        elif (str(action) == ' down'):
            aux.append(4)
            aux.append(n_robot)
            aux.append(robot_row[n_robot-1])
            aux.append(robot_column[n_robot-1])
            robot_row[n_robot-1] = robot_row[n_robot-1] + 1
            aux.append(robot_row[n_robot-1])
            aux.append(robot_column[n_robot-1])
        elif (str(action) == ' right'):
            aux.append(5)
            aux.append(n_robot)
            aux.append(robot_row[n_robot-1])
            aux.append(robot_column[n_robot-1])
            robot_column[n_robot-1] = robot_column[n_robot-1] +1
            aux.append(robot_row[n_robot-1])
            aux.append(robot_column[n_robot-1])
        elif (str(action) == ' left'):
            aux.append(6)
            aux.append(n_robot)
            aux.append(robot_row[n_robot-1])
            aux.append(robot_column[n_robot-1])
            robot_column[n_robot-1] =  robot_column[n_robot-1] -1
            aux.append(robot_row[n_robot-1])
            aux.append(robot_column[n_robot-1])
        elif (str(action) == ' change_paint_black'):
            aux.append(0)
            aux.append(n_robot)
            aux.append(2)
        elif (str(action) == ' change_paint_white'):
            aux.append(0)
            aux.append(n_robot)
            aux.append(1)
        list.append(aux)

    #return the information path.
    return list


""" Draw the main board ( edge )
    Each tile has 80 pixels.
    """
def DrawBoard(rows, columns,newOffsetY,total_rows,total_columns):

    #Square definition
    turtle.speed(0)
    turtle.pensize(3)
    turtle.penup()
    turtle.goto(offsetX,newOffsetY)
    turtle.pendown()
```

```
    turtle.color("black")
    turtle.forward(columns*80)
    turtle.right(90)
    turtle.forward(rows*80)
    turtle.right(90)
    turtle.forward(columns*80)
    turtle.right(90)
    turtle.forward(rows*80)
    turtle.right(90)

    #Lets fill the board using columns and rows
    turtle.penup()
    turtle.goto(offsetX, newOffsetY)
    actualx = offsetX
    actualy = newOffsetY
    for i in range(0,rows):
        turtle.goto(actualx,actualy)
        turtle.pendown()
        turtle.forward(columns*80)
        turtle.penup()
        actualy = actualy - 80
    turtle.penup()
    actualy = newOffsetY
    actualx = offsetX
    for j in range(0,columns):
        turtle.goto(actualx,actualy)
        turtle.pendown()
        turtle.right(90)
        turtle.forward(rows*80)
        turtle.left(90)
        turtle.penup()
        actualx = actualx + 80




""" Paint the main board with grey color as initialization.
    Each tile has 80x80
    """
def setInitialBoard(rows,columns,robot_row, ...
    robot_column,newOffsetY,total_rows,total_columns,how_many_robots):

    global offsetX
    global offsetY

    newOffsetY = offsetY + (total_rows*80)
    rob_x = []
    rob_y = []
    for i in range(0,how_many_robots):
        rob_x.append(0)
        rob_y.append(0)

    for i in range(0,how_many_robots):
        rob_x[i]=(robot_column[i]*80) + 40 + offsetX
        rob_y[i] = newOffsetY - 40 - (robot_row[i]*80)

    for i in range(0,rows):
        for j in range(0,columns):
           paintPosition(i,j,"grey",newOffsetY)
    turtle.penup()
    for i in range(0,how_many_robots):
        turtle.penup()
        turtle.goto(rob_x[i],rob_y[i])
        turtle.color(robot_has_color[i])
        turtle.stamp()


""" Draw a line in order to represent the robot movement.
    Each robot will have a different color
    The information needed is the initial position and the target position, plus which robot is ...
        moving.
    """
def WriteLine(robot,initial_row, initial_column, target_row, target_column,action,newOffsetY):

    global offsetX
    global offsetY
```

```

    x_initial = (initial_column*80) + 40 + offsetX
    y_initial = newOffsetY - 40 - ((initial_row)*80)
    x_target = (target_column*80) + 40 + offsetX
    y_target = newOffsetY - 40 - (target_row*80)
    n_robot = robot
    turtle.color(robot_path[n_robot-1])
    turtle.penup()
    turtle.goto(x_initial,y_initial)
    turtle.pendown()
    if (action == 0):
        turtle.left(90)
        turtle.forward(80)
        turtle.right(90)
    elif(action == 1):
        turtle.right(90)
        turtle.forward(80)
        turtle.left(90)
    elif(action == 2):
        turtle.forward(80)
    elif(action == 3):
        turtle.backward(80)


""" Function called to paint a specific tile of the grid.
    row, column: which tile do you want to paint.
    color : which color (black,white)
    """
def paintPosition(row,column,color,newOffsetY):

    global offsetX
    global offsetY

    turtle.penup()
    x_position = (column*80) + offsetX +2
    y_position = newOffsetY - (row*80) +2 -80

    turtle.goto(x_position,y_position)
    turtle.color(color)
    turtle.begin_fill()
    for k in range(4):
        turtle.forward(76)
        turtle.left(90)
    turtle.end_fill()


""" This function draws a point where the robot stands.
    """
def DrawEndPoint(robot,row,column,newOffsetY):

    global offsetY
    global offsetX
    turtle.penup()
    if (robot == 1):
        color = robot_path[0]
    elif(robot ==2):
        color = robot_path[1]
    elif(robot == 3):
        color = robot_path[2]
    elif(robot == 4):
        color = robot_path[3]
    elif (robot == 5):
        color = robot_path[4]
    x = (column*80) + 40 + offsetX
    y = newOffsetY - 40 - (row*80)
    turtle.goto(x,y)
    turtle.dot(10, color)




""" Analyzing and drawing the data.
    The data is analized following the criteria previously used in the GenerateData function.
    The main parameter needed is the list with the path information.
    """
def AnalyzingData(list,newOffsetY):
```

```
    global robot_has_color

    for e in list:
        n_robot = 0
        if (e[1] == 1):
            n_robot = 1
        elif(e[1] ==2):
            n_robot = 2
        elif(e[1] == 3):
            n_robot = 3
        elif(e[1] == 4):
            n_robot = 4
        elif(e[1] == 5):
            n_robot = 5
        if (e[0] == 0):
            if(e[2] == 1):
                robot_has_color[n_robot-1] = "white"
            elif(e[2] ==2):
                robot_has_color[n_robot-1] = "black"
        elif(e[0] == 1):
            color = robot_has_color[n_robot-1]
            paintPosition(e[2],e[3],color,newOffsetY)
        elif(e[0] == 2):
            color = robot_has_color[n_robot-1]
            paintPosition(e[2],e[3],color,newOffsetY)
        elif(e[0] == 3):
            WriteLine(e[1],e[2],e[3],e[4],e[5],0,newOffsetY)
            DrawEndPoint(e[1],e[4],e[5],newOffsetY)
        elif(e[0] == 4):
            WriteLine(e[1],e[2],e[3],e[4],e[5],1,newOffsetY)
            DrawEndPoint(e[1],e[4],e[5],newOffsetY)
        elif(e[0] == 5):
            WriteLine(e[1],e[2],e[3],e[4],e[5],2,newOffsetY)
            DrawEndPoint(e[1],e[4],e[5],newOffsetY)
        elif(e[0] == 6):
            WriteLine(e[1],e[2],e[3],e[4],e[5],3,newOffsetY)
            DrawEndPoint(e[1],e[4],e[5],newOffsetY)
        elif(e[0] == 7):
            DrawEndPoint(e[1],e[2],e[3],newOffsetY)

""" Main function of the Graphs.
    It draws all the graphics. The information is received from main.py file.
    """
def Draw(robots, state, path):

    turtle.setup(800,800)
    turtle.speed(0)
    how_many_robots = len(robots)
    robot_row,robot_column,total_rows, total_columns = setInitialData(robots,state,how_many_robots)
    newOffsetY = offsetY + (total_rows*80)
    DrawBoard(total_rows, total_columns,newOffsetY,total_rows,total_columns)
    setInitialBoard(total_rows,total_columns,robot_row,robot_column,newOffsetY,total_rows,total_columns,how_many
    data = GenerateData(path,robot_row, robot_column, robot_has_color,total_rows,total_columns)
    turtle.speed(0)
    AnalyzingData(data, newOffsetY)
    turtle.hideturtle()
    turtle.done()
```

E Result PDDL

CASES:

0. Algorithm without prunning vs algorithm using prunning:

Super simple PDDL file:

1. BFS(without FOC) Time: 0:00:01.001279

2. BFS (with FOC) Time: 0:00:00.302438

Little bigger size:

1. BFS(without FOC) Time: 0:20:17.625262

2. BFS (with FOC) Time: 0:00:01.151655

1. Number of robots

-> We have 3 PDDL files using the same grid. In robots$_case_1.pddl- > 2robotsInrobots_case_2.pddl- > 3robotsInrobots_case_3.pdd$
$4robots$

-> The cpu time achieved in all the three cases is:

Time: 0:00:01.969610 Time: 0:00:39.900257 Time: 0:28:30.818512

2. Pattern

-> We have 2 PDDL files using the same grid but changing the pattern. Basically, we have 1 pattern with all the tiles painted and another one with some tiles painted and some others not.

In pattern$_case_1.pddl- > AllthetilesarepaintedInpattern_case_2.pddl- > Onlysometilesarepainted.$

-> The cpu time achieved in the two cases is:

Time: 0:00:58.846066

Time:

0:00:37.260438

3. Size case

-> We have 10 PDDL files changing its size and using only 2 robots. -> The goal in this case is basically study how the complexity increase in terms of number of tiles but also in terms of the shape/structure of the grid.

In size$_case_1.pddl- > 3rows, 3columns$

In size$_case_2.pddl- > 4rows, 3columns$

In size$_case_3.pddl- > 3rows, 4columns$

In size$_case_4.pddl- > 4rows, 4columns$

In size$_case_5.pddl- > 5rows, 4columns$

In size$_case_6.pddl- > 4rows, 5columns$

In size$_case_7.pddl- > 5rows, 5columns$

In size$_case_8.pddl- > 6rows, 5columns$

In size$_case_9.pddl- > 5rows, 6columns$

In size$_case_10.pddl- > 6rows, 6columns$

-> The cpu results in this case are:

1. Time: 0:00:00.279385

2. Time: 0:00:00.345064

3. Time:

0:00:00.798212 4. Time:

0:00:01.140241

5. Time: 0:00:01.622258

6. Time: 0:00:04.383790

7. Time: 0:00:06.509273

8. Time: 0:00:08.390610

9. Time: 0:00:25.309673

10.Time: 0:00:32.354215

4. Available colors case (FUTURE RESEARCH)

-> In that case we only have one PDDL file. What we want to test here is basically the fact that the amount of color can be reduced or limited.

4.1 Two robots with a lot of amount-color. 1000, 1000

4.2 Two robots with the limited amount of color and only for one color. Robot 1 -> Can only paint black and has the "just enough" Robot 2 -> Can only paint white and has the "just enough"

4.3 Two robots with the limited amount of color but they can paint with the two colors (black and white)

-> Cpu time for the three cases: (6 cells in black, 6 cells in white)

4.1 Time: 0:00:00.650595

References

[1] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl-the planning domain definition language," 1998.

[2] J. Koehler and J. Hoffmann, "On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm," *Journal of Artificial Intelligence Research*, vol. 12, pp. 338–386, 2000.

[3] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial intelligence*, vol. 90, no. 1, pp. 281–300, 1997.

[4] D. L. Kovacs, "Bnf definition of pddl 3.1," *Unpublished manuscript from the IPC-2011 website*, 2011.

[5] A. Coles, A. Coles, A. G. Olaya, S. Jiménez, C. L. López, S. Sanner, and S. Yoon, "A survey of the seventh international planning competition," *AI Magazine*, vol. 33, no. 1, pp. 83–88, 2012.

[6] D. L. Kovacs, "A multi-agent extension of pddl3. 1," in *Proceedings of the 3rd Workshop on the International Planning Competition (IPC), ICAPS-2012, Atibaia, Brazil*, 2012, pp. 19–27.

[7] B. Bonet and H. Geffner, "Hsp: Heuristic search planner, entry at 4th international conf. on artificial intelligence planning systems (aips-98) planning competition," 1998.

[8] J. Luo, W. Zhang, J. Cui, C. Zhu, J. Huang, and Z. Liu, "Heuristic search for planning with different forced goal-ordering constraints," *The Scientific World Journal*, vol. 2013, 2013.

[9] N.-F. Zhou, R. Barták, and A. Dovier, "Planning as tabled logic programming," *Theory and Practice of Logic Programming*, vol. 15, no. 4-5, pp. 543–558, 2015.

[10] D. P. Mehta and S. Sahni, *Handbook of data structures and applications*. CRC Press, 2004.

[11] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, p. 27, 1995.

[12] C. Muise, "Planning.domains," https://bitbucket.org/planning-researchers/, 2017.

[13] F. Hernán, "pddl-lib," https://github.com/hfoffani/pddl-li,, 2015.