Solve a PPDDL Problem as a Simple Stochastic Game

Lorenzo Mandelli 1747091

Nicolò Paoletti 1888157

Valerio Ponzi 1760886

Giulio Recupito 1668320

October 11, 2022

I. Introduction

The aim of this work is to understand how to apply the simple stochastic games (SSGs) theory to a classic PPDDL problem, which extends PDDL toward probabilistic scenarios. SSG theory tackle problems as directed graph, considering the possibility of having probabilistic action outcome. Ideally if we had moved from a classic PPDDL to a direct graph of this kind we would have been able to also move from an exponential time to a polynomial one. Initially we will introduce PDDL and PPDDL formally than discussing how thhese kind of problems are tackled, addressing the parsing issue and the graph building. Later we will introduce how we solved the planning problem, introducing SSG theory and the consequent algorithm we end up using. Finally we conclude speaking about the complexity issue and doing some consideration.

II. PPDDL Problems

PPDDL stands for "planning domain definition language", which is an extension of its predecessor PDDL. This kind of extension is a first step towards a general language for describing probabilistic and decision theoretic planning problems.

i. PDDL

PDDL [3] is a language for specifying deterministic planning domains and problems. A PDDL planning domain consists of a set T of types, a sub-typing relation $ST \subset T \times T$, a set C of global objects (domain constants), a set P of predicates, a set F of functions, whch in our case we do not consider, and a set AS of action schemata. In PDDL, predicates are used to encode Boolean state variables. Objects and variables are terms, and in PDDL all terms have some type $\tau \in T$. All declared types are by default sub-types of the built-in PDDL type object. A type is also a sub-type of itself (the sub-typing relation ST is reflexive), and if τ_1 is a sub-type of τ_2 and τ_2 is a subtype of τ_3, then τ is a subtype of τ_3 (ST is transitive). PDDL also includes support for union types $\tau_1 \cup .. \cup \tau_n$, with the restriction that each τ_i is a simple type. It is worth noting that a domain definition alone does not, in general, determine the extent of the state space for planning problems linked to the domain, unless all predicates take no arguments, thus being domain constant. In addition to objects declared as domain constants, objects can also be declared in problem definitions. The state space can be determined only if the complete set of objects for a planning problem is known. Actions in PDDL can be thought like sets of state transitions, with a state being a particular assignment to the set of state variables of a planning problem. An action consists of a precondition, characterizing the set of states in which the action is applicable and an effect, which specifies updates to state variables that occur at the execution of the action in the given state. For a Boolean state variable b, the effect b (or an application yielding the state variable b) simply means that b should be set to true in the next state, while in order to set b to false, the notation (**not** b) is used. A planning problem consists of a set of state variables V , a set of actions A, an initial state s_0 , a goal condition θ identifying a set of goal states, and an optimization metric f that is typically a function of numeric state variables evaluated in a goal state. In PDDL, a planning problem is always associated with a domain definition, and the definition of a planning problem includes a declaration of a set of problem-specific objects O. The state variables V for the planning problem are obtained from O, C, P , and F as all type-consistent applications of predicates to objects (including domain constants). The set A of actions is obtained similarly as all type-consistent applications of action schemata in AS to objects in $O \cup C$. The process of obtaining all state variables and actions for a planning problem through the exhaustive application of predicates and action schemata to objects is referred to as *grounding*.

ii. PPDDL

In order to define probabilistic and decision theoretic planning problems, we need to add support for probabilistic effects. The syntax for probabilistic effects is

(**probabilistic** p_1 e_1 .. p_k e_k)

meaning that effect e_i occurs with probability p_i. We require that the constraints $pi \geq 0 \sum_{i=1}^{k} p_i = 1$ are fulfilled: a probabilistic effect declares an exhaustive set of probability-weighted outcomes. We do, however, allow a probability-effect pair to be left out if the effect is empty. In other words, the effect (**probabilistic** p_1 e_1 .. p_k e_k) with $\sum_{i=1}^{l} p_i \leq 1$ is syntactic sugar for (**probabilistic** p_1 e_1 .. p_l e_l q (**and**)) with q = 1 - $\sum_{i=1}^{l} p_i$. PPDDL allows arbitrary nesting of conditional and probabilistic effects. It is worth noting that a single PPDDL action schema can represent a large number of actions and a single predicate can represent a large number of state variables, meaning that PPDDL often can represent planning problems more synthetically than other representations. For example, the number of actions that can be represented using m objects and n action schemata with arity c is $m \cdot n$, which is not bounded by any polynomial in the size of the original representation $m + n$. Grounding is by no means a prerequisite for PPDDL planning, so planners could conceivably take advantage of the more compact representation by working directly with action schemata.

iii. Parser

The PPDDL parser we used is an extension of the classical PDDL parser. Our implementation is based on Lax and Yacc. The parse read the source program and discover its structure.

Lex and Yacc can generate program fragments that solve the this task by decomposing it into sub tasks:

- Split the source file into tokens (Lex);
- Find the hierarchical structure of the program (Yacc);

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. Lex source is a table of regular expressions and corresponding program fragments. We extend the set of regular expression in order to include also the possibility of having probability. The introduced set new set of regular expressions handle non uniform probability density functions together with nested probability. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized, the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed following the order in which the corresponding regular expressions occur in the input stream. The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point.

On the other hand Yacc provides a general tool for imposing structure on the input to a computer program. Yacc then generates a parser function to control the input process defined by the user, by calling the the lexical analyzer to pick up the tokens from the input stream. These tokens are organized according to the input structure rules, called grammar rules. When one of these rules has been recognized, then an action is invoked. The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read. The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls a lexical analyzer function(Lex), represent-

ing the kind of token read.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the end marker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification, hence, when the input tokens seen, together with the lookahead token, cannot be followed by anything that would result in a legal input.

iv. Graph

Based on the aforementioned parser, we can represent a common PPDDL program as a directed graph. We defined all the classes needed to handle correctly each object composing the PPDDL problem, namely a domain class, a problem class and, literals, predicates and terms classes, thus having an object oriented structure underling our programs. Terms are the smaller units needed to build predicates out of we obtain literals. For instance considering an example

`vehicle-at :('?loc':  'x02y04')` is a parser.literal.Literal object
`'?loc':  'x02y04'` is a parser.predicate.Predicate object

`'x02y04'` is a parser.term.Term object

Actions' preconditions and effects are predicates, consequently states are obtained as lists of predicates. The idea is to start from the the problem and domain objects, identifying the initial state, defined inside the problem, and build a tree representation of all the possible paths as a dictionary action:state. In order to handle the probability information and carry it on in the best way, the state value has been defined their selves as a list of dictionary probability:predicates, where here the general notation 'predicates' means a list of objects describing a specific configuration, hence a state. The transition to a new state is done checking all the action preconditions, and building the resulting state accordingly, hence removing the precondition predicates and adding the effects ones. The "possible paths" dictionary is later used to build a direct graph by means of networkx library. Each node in the graph contains information about the node type, as we have seen varying among max and average and the node index, needed to define edges direction. Max nodes, differently from average nodes, contain information about the state; average nodes are like gates where the edges branch, thus there is no meaningful reasons to copy information about the state witch remains the same as the parental one. Each edge contains information about the action taking place in the node where it starts and the probability that action takes place. As we have seen edges from max nodes to average nodes as well as edges between max nodes describe deterministic action outcomes, hence happening with unitary probability while edges from average nodes to max nodes describe probabilistic action outcomes, thus having multiple branches with values summing up to one. In addition we cannot have edges from average nodes to average nodes. Next in Figure 1 we display an example of graph we obtained from a PPDDL problem. For the sake of simplicity we choosed a simple problem, and a resulting "finite and readable" graph. The red node is the initial state, black nodes are sink states, in green we coloured average nodes and in light blue max nodes.

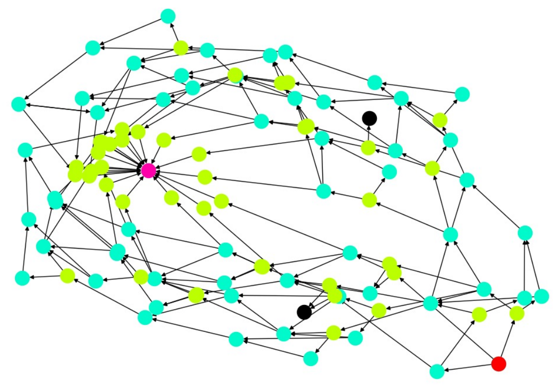

Figure 1: *Example of a graph obtained by a simple problem.*

III. Solver

Considering the given obtained representation of our problem, which is a direct graph with three types of vertices respectively, max, average, and sink plus a start vertex, we are now able to treat it like a simple stochastic game (briefly SSG).

i. SSG

Based on [1] a simple stochastic game is a direct graph G = (V,E). The vertex set V is the union of disjoint sets $V_{max}, V_{average}$, together with two special vertices, the 0-sink and the 1-sink. One vertex of V is called the *start* vertex. Assuming each node, different from the sink ones, having one or multiple neighbours, depending on which kind of node it is (max or average). We considered an extended version of the initial problem, taking in account the possibility that an average vertex could have more than two neighbours, thus being associated to a non-uniform probability density function; moreover we modeled a player versus nature environment, not as the player versus player setting proposed in [1]. A *strategy* τ in this scenario is than a set of edges of E, each with its left hand at a max vertex, such that for each max vertex i there is exactly one edge (i,j) in τ. In the game theory literature, this kind of strategies are called *pure stationary* strategies, since the player do not use probabilistic choices and the player always chose the same move from a vertex every time the same vertex is reached. In the giving settings it is prove that the SSG can be solved in polynomial time. As sated in [**?**] the solution of G = (V,E) be an SSG with n vertices, is the optimal solution of a simple constrained minimization linear programming problem, which is: minimize $\sum_{l=1}^{n}$ subject to the constraints:

$v(i) \geq v(j), (i,j) \in E$ if i is a max vertex

$v(i) \geq \sum_{j=1}^{k} p_j(v(J)), (i,j) \in E$ if i is an average vertex, where k is the number of neighbours of i and p_j is the probability to end up in the j^{th} neighbour

$v(i) = 0$ if i is a sink node

$v(i) = 1$ if i is a goal node

$v(i) \geq 0, \forall i$

At this point we know that, having the graph representing such a problem, we can move the task to a simple optimization problem, solvable, for instance by means of the **Simplex Method**.

ii. Simplex Method

In the following lines we will briefly describe how the Simplex method works. The simplex algorithm operates on linear programs in the canonical form

$$\begin{aligned} \min_{x} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b, \\ & x \geq 0 \end{aligned} \tag{1}$$

with $c = (c_1, ..., c_n)$ the coefficients of the objective function, x = (x_1, ..., x_n) are the variable of the problem, A a pxn matrix and b=(b_1, ..., b_p). In geometric terms, the feasible region defined by all the values of x such that $Ax \leq b$ and $\forall i, x_i \geq 0$ is a (possibly unbounded) convex polytope. An extreme point or vertex of this polytope is known as a basic feasible solution (BFS). The solution of a linear program is accomplished in two steps. In the first step, known as Phase 1, a starting extreme point is found, resulting in a BFS or in an empty region. If the latter is the case then the program is infeasible, if not the simplex algorithm pass to Phase 2, where the algorithm is applied using the BFS found in Phase 1 as a starting point. The possible result from Phase 2 are either an optimum basic feasible solution or an infinite edge on which the objective function is unbounded above. We can also express a linear program in standard form, so to say when we have a linear program in the form **Ax = b** $\forall i, x_i \geq 0$ as a tableau

$$\begin{bmatrix} 1 & -c^T & 0 \\ 0 & A & b \end{bmatrix}$$

The first row defines the objective function and the remaining rows specify the constraints. The zero in the first column represent the zero vector of the same dimension as,vector b. If the linear program is in given in the canonical tableau, the simplex algorithm proceeds by performing successive pivot operations each of which give an improved basic feasible solution; the choice of pivot element at each step is largely determined by the requirement that this pivot improves solution.

The simplex method output an array of n elements, one for each node of the graph, containing the probability that starting from that node, we end up in the goal node. Next we display the output of the simplex method applied to the problem depicted in 1

```
[
1.06303393 0.73251663 0.34279100 0.12182052 0.19780839 0.17698863
0.22902882 0.26717919 0.36483302 0.5427862 0.56153425 0.62315621
0.69628452 0.79620168 0.63011896 0.79745855 0.74690075 0.87246654
0.35224907 0.48704109 0.43839591 0.56774975 0.66077537 0.76635253
0.59957841 0.75022523 0.71881281 0.85675294 0.50159683 0.67995671
0.69046894 0.72185974 0.85750786 0.49879535 0.00000000 0.61405444
0.81658037 0.58944646 0.42829789 0.48527385 0.29958237 0.31527231
0.39745978 0.56788973 0.57447170 0.62934697 0.69919646 0.79720542
0.64111624 0.80097843 0.75385140 0.87439825 0.44734644 0.52695710
0.44551900 0.57446107 0.66226406 0.76459594 0.61255951 0.74642802
0.72514508 0.85730757 0.57575163 0.74203877 0.71939429 0.74868643
0.86540956 0.59833805 0.00000000 0.62128621 0.75672717 0.59490724
0.55472726 0.50273828 0.43419941 0.58461589 0.67791959 0.61991466
0.68700579 0.83147315 0.60590508 0.47843292 0.71308211 0.87906198
0.72539078 0.67305875 0.69691321 0.56625850 0.69360311 0.74068538
0.64445998 0.70908954 0.83727118 0.69728012 0.48540678 0.74042209
0.87478646 0.76250429 0.75514588 0.68293597 0.30313191 0.01502747
0.70229479 1.  ]
```

We notice how we have 0 as value only for max nodes which terminate in a sink node, and 1 only for the max node which terminate in the goal node. Starting from this array we have defined a greedy strategy; for each max node, the only node we can control, we check all its neighbours, and select the one maximizing the probability of ending up in the goal node. Thus the strategy is defied as a dictionary state:action which link the index of each max node with the best action we can choose in it. Next we display the strategy for the same problem depicted in 1

```
0 :  strategy:  move-car(’?to’:  ’x01y03’, ’?from’:  ’x01y01’)
2 :  strategy:  change-tire()
3 :  strategy:  change-tire()
4 :  strategy:  move-car(’?to’:  ’x01y03’, ’?from’:  ’x02y02’)
6 :  strategy:  load-tire(’?loc’:  ’x03y03’)
7 :  strategy:  change-tire()
8 :  strategy:  move-car(’?to’:  ’x02y04’, ’?from’:  ’x03y03’)
10 :  strategy:  load-tire(’?loc’:  ’x02y04’)
11 :  strategy:  change-tire()
12 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
14 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
16 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
18 :  strategy:  load-tire(’?loc’:  ’x03y03’)
20 :  strategy:  load-tire(’?loc’:  ’x02y04’)
21 :  strategy:  change-tire()
22 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
24 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
26 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
28 :  strategy:  move-car(’?to’:  ’x02y04’, ’?from’:  ’x03y03’)
30 :  strategy:  change-tire()
31 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
35 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x01y03’)
38 :  strategy:  move-car(’?to’:  ’x01y03’, ’?from’:  ’x02y02’)
40 :  strategy:  load-tire(’?loc’:  ’x03y03’)
41 :  strategy:  change-tire()
42 :  strategy:  move-car(’?to’:  ’x02y04’, ’?from’:  ’x03y03’)
44 :  strategy:  load-tire(’?loc’:  ’x02y04’)
45 :  strategy:  change-tire()
46 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
48 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
50 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
52 :  strategy:  load-tire(’?loc’:  ’x03y03’)
54 :  strategy:  load-tire(’?loc’:  ’x02y04’)
55 :  strategy:  change-tire()
56 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
58 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
60 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
62 :  strategy:  move-car(’?to’:  ’x02y04’, ’?from’:  ’x03y03’)
64 :  strategy:  change-tire()
65 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
69 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x01y03’)
72 :  strategy:  move-car(’?to’:  ’x01y03’, ’?from’:  ’x02y02’)
74 :  strategy:  change-tire()
75 :  strategy:  move-car(’?to’:  ’x02y04’, ’?from’:  ’x03y03’)
77 :  strategy:  change-tire()
78 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
81 :  strategy:  change-tire()
82 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x01y03’)
85 :  strategy:  move-car(’?to’:  ’x01y03’, ’?from’:  ’x02y02’)
87 :  strategy:  change-tire()
88 :  strategy:  move-car(’?to’:  ’x02y04’, ’?from’:  ’x03y03’)
90 :  strategy:  change-tire()
91 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x02y04’)
94 :  strategy:  change-tire()
95 :  strategy:  move-car(’?to’:  ’x01y05’, ’?from’:  ’x01y03’)
99 :  strategy:  move-car(’?to’:  ’x01y03’, ’?from’:  ’x01y01’)
101 :  strategy:  load-tire(’?loc’:  ’x02y02’)
```

IV. Complexity

In order to analyze the computational complexity of our solution, we have to take into account:

- The computational complexity of the graph creation
- The computational complexity of the planner

The creation of the graph requires the expansion of each possible action feasible in each possible node, except the goal node and sink nodes. Thus, the computational complexity is $O(|A|^n)$ where $|A|$ is the set of actions and n is the maximum distance (depth) from the starting node. As stated in [1], since our SSG involves just max and avg nodes, the solution of our problem can be found in a time that is polynomial in the number of nodes. In conclusion, the resulting time complexity is then exponential due to the creation of the graph.

V. Conclusions

We compared the time results with a very recent planner called Safe-Planner [SP] [2], using two different instances the triangle-tireworld problem [P01] and [P02]. As reported in the table ??, SP exponentially faster in finding a solution, since planner algorithms used do not require to expand and create all the search space exploiting various heuristics, unlike our solution that need this procedure in order to create the graph. Also the time to find the solution is then affected, since the simplex has to work with a number of node which is exponential in the depth of the solution.

Time comparisons		
P01 phase	Our planner	Safe planner
Parsing	0.0030524730682373047 s	0.003856182098388672 s
Graph creation	0.5154554843902588 s	/
Solution finding	0.56896877288 s	/
Total time to solve the problem	0.5726768970489502 s	0.01676321029663086 s
Complete process time	0.5726768970489502 s	0.02066659927368164 s
P02 phase	Our planner	Safe planner
Parsing	0.0029532909393310547 s	0.0022017955780029297 s
Graph creation	194.8837218284607 s	/
Solution finding	139.53481912612915 s	/
Total time to solve the problem	334.418540955 s	0.35431551933288574 s
Complete process time	334.46128702163696 s	0.3565833568572998 s

Table 1: *Comparison between our planner and SP execution time.*

References

[1] *The Complexity of Stochastic Games* ANNE CONDON, 1992

[2] *Safe-Planner: A Single-Outcome Replanner for Computing Strong Cyclic Policies in Fully Observable Non-Deterministic Domains* Vahid Mokhtari, Ajay Suresha Sathya, Nikolaos Tsiogkas, Wilm Decre´, 2021

[3] *PDDL - The Planning Domain Definition Language* Ghallab, Malik & Knoblock, Craig & Wilkins, David & Barrett, Anthony & Christianson, Dave & Friedman, Marc & Kwok, Chung & Golden, Keith & Penberthy, Scott & Smith, David & Sun, Ying & Weld, Daniel. (1998).