SAPIENZA - UNIVERSITY OF ROME

Reasoning Agent Project

Davide di Ienno

February 7, 2022

1 INTRODUCTION

The main objective of this work is to translate a probabilistic planning problem to a Simple Stochastic Game (SSG). The starting point are pddl-based problems, in particular the ones from [1]; their main characteristic (that differentiate them from basic pddl problems) are probabilistic effects in their actions. In order to obtain a SSG it is made an assumption: there is only one player, that in this case is the Max player (it is not relevant the choice, it is important that there is only one player). Such choice can be interpreted as if, in the resulting game, the Max player represent the agent that is fighting with the environment, that is instead modelled with Average vertices, to reach its goal (satisfied in the sink vertix). One important choice made here is to not modify the problem while translating it: the resulting SSG does not retain information contained in the original problem, but each Max vertex correspond to a state and each edge represent the choice of doing an action. This has serious consequences that are discussed later in this work.
This report is organized in the following way: First, a section that explain how the original problems in pddl are read with the objective of retrieve key information about the domain and the problem; Then, the Section 3 is used to present how the game is generated from the imported problem and the assumptions done to make this possible, presenting also its main limitations; Finally, there is a final section dedicated to the solution of the generated SSG (in the simplified case with only Max and Average vertices).

2 PARSER

The first, needed, step consists of reading the domain and problem pddl files. In order to work on this project and to test the implementation presented here, it was really useful to have as a reference the *prob-benchmarks* in [1, 2, 3, 4]. Not all of them were used (some of the domains are buggy, as stated in the repository itself), but a couple of them functioned as a key reference while developing this implementation. Not all the features of pddl were implemented: some keywords were avoided since they are beyond the scope of this project. On the other side, it was put much effort while dealing with probabilistic effects, as will be explained in the action section. This section is organized in order to reflect the implementation: first, there is a

```
(:predicates (vehicle-at ?loc - location) (spare-in ?loc - location) (road
    ?from - location ?to - location) (not-flattire) (hasspare))
```

Listing 1: Predicates of tire_domain.pddl from [1].

```
{'vehicle-at': [('loc', 'location')],
 'spare-in': [('loc', 'location')],
 'road': [('from', 'location'), ('to', 'location')],
 'not-flattire': [],
 'hasspare': []}
```

Listing 2: Output of the method for importing predicates when given as input the ones in Listing 1.

description about how the domain file is read; then, in a similar way, it is presented how the problem file is handled.

2.1 Domain

Not all the features from pddl are implemented here, only a limited (basic) set of them. The domain file is read starting from the types, where they are present. They are not really used in this work, but a full implementation should use them to do the necessary checks in the next phases.
Then there are predicates, they can be true or false, they can involve objects (and such possibility is indicated by words starting with a question mark) or not. Consider as an example the *tire_domain.pddl*, the predicates from it are reported in Listing 1. The proposed solution is to exploit the syntax and save the predicates in a Python Dict, saving information regarding variables and related types. The output for this example is reported in Listing 2. Predicates such as *not-flattire* are really useful when checking the preconditions of actions: it is easy to verify if they are true or false in a certain state, saving time and avoiding useless computations.

2.1.1 Actions

It is important to remember that the domains considered are non-deterministic; in particular, while considering the ones in *prob-benchmarks* [1], the actions have effects that are applied with a certain probability. Actions have: parameters, preconditions and effects. In order to apply an action to a given state, the preconditions must be satisfied for the declared parameters. For more information about the applicability of actions and the satisfaction of preconditions, see Section 3.1.
Consider again an example from the tire domain, the action *move-car* in Listing 3. A class was

```
(:action move-car
:parameters (?from - location ?to - location)
:precondition (and (vehicle-at ?from) (road ?from ?to) (not-flattire))
:effect (and (vehicle-at ?to) (not (vehicle-at ?from)) (probabilistic 2/5 (
    not (not-flattire))))
)
```

Listing 3: An example of an action from tire_domain.pddl from [1], in particular the move-car action. The example is useful to highlight the structure of preconditions and probabilisitc effects.

```
[{'name': 'from', 'type': 'location '}, {'name': 'to', 'type': 'location'}]
```

Listing 4: Output of the method for importing parameters of an action when given as input the ones in Listing 3.

```
[{'predicate': 'not-flattire', 'names': [], 'bool': True},
 {'predicate': 'vehicle-at', 'names': ['from'], 'bool': True},
 {'predicate': 'road', 'names': ['from', 'to'], 'bool': True}]
```

Listing 5: Output of the method for importing preconditions of an action when given as input the ones in Listing 3. Note how the Python list is ordered by the number of possible objects in the predicates. The 'bool' entry is used to take into account negated condition (the ones with *not* before them), which are not present in this example.

designed to properly handle the actions. The text is passed to it and all the necessary information are organized in properties as dicts, in a similar way as already seen with predicates. First of all, there are parameters; a List of Dict, such as the one in Listing 4 can be obtained by saving their name without discarding the related type. Then, there are preconditions. Only a subset of possible pddl keywords were actually implemented: *and, not, =*. So, keywords such as *or, imply, forall, when, exists*, were not implemented and passing a domain file that includes them will result in some sort of error. It is also true that adding support for such keywords should be straithforward and could be subject of future work. While importing preconditions in the class as a dict, they are ordered by the number of parameters they hold, this will be useful later on while checking for their applicability. The result for the action taken as example is presented in Listing 5. Note how, in each precondition, it is reported the predicate, the list of parameters (when there are any) and a boolean value that, basically, is triggered to be false if the precondition is preceded by *not*.
Finally, there are effects, that may be probabilistic or not; actually, they can also be partially probabilistic (i.e. only a part of the effects is probabilistic, while the the rest is deterministic). When the effects are deterministic, it is easier to handle them and actually not so different from preconditions. Regarding the probabilistic effects, if there are both deterministic and probabilistic effects, the deterministic ones are duplicated; in this way it is possible to obtain a set of probabilistic effects. If needed, the complementary probability is added (for example, when a couple of effects, x and y, are deterministic and a third, z, happens with probability 0.4, the output will be something like: x, y, z with probability 0.4 and x, y with probability 0.6). In the example considered, there are two deterministic effects (one also negated) and one probabilistic (negated) effect; the output of the described method is in Listing 6. Note how the structure is similar to the one of the preconditions and the addition of probabilities (0.4 and 0.6).

2.2 Problem

In the problem file there are information about the objects, initial state and goal condition. It is important to remark that it is valid the closed-world assumption, so if something cannot be found in the current state, it is considered to be false. Importing the objects is straithforward, since they are just a list. The initial state is really important, but the process is not so different from predicates and preconditions saw before. The state is actually embedded in a specific python class in order to have easy access to its information: there are many methods that are later used to retrieve a subset of the state based on some conditions (the predicate, the presence of certain objects and more).

```
{'prob': True,
 'effects': [(0.4,
  [{'predicate': 'not-flattire', 'names': [], 'bool': False},
   {'predicate': 'vehicle-at', 'names': ['to'], 'bool': True},
   {'predicate': 'vehicle-at', 'names': ['from'], 'bool': False}]),
  (0.6,
  [{'predicate': 'vehicle-at', 'names': ['to'], 'bool': True},
   {'predicate': 'vehicle-at', 'names': ['from'], 'bool': False}])]}
```

Listing 6: Output of the method for importing probabilistic effects of an action when given as input the ones in Listing 3. The 'bool' entry is used to take into account negated effects, i.e. the predicates that shall be eliminated from the original state while applying the action. Note how there is a List of Tuple with the structure (*probability, effects*) to handle probabilitic effects. In this example only the *not (not -flattire)* has a probability attached to it, so the other two effects are just replicated in all the cases; this is done to make things easier in a later stage.

3 Game Generator

Once that the domain and problem have been correctly imported in the designed structure, it is needed to generate the corresponding game. The main idea is to generate all the possible states, starting from the initial one; then exporting all of them as a SSG. This is not viable in all the cases, as will be explained later in this section.

3.1 Applicable actions

Before talking about the states generation, it is useful to clarify how the possible actions are considered. Given a state, an agent/player shall decide what to do; but, in order to do that, a set of possible actions (and in some cases a combinations of them involving different objects) shall be available. So, considering a given state (it is not important what state here, the process is general), it was implemented a method that is able to return the set of available (to the agent) actions and their corresponding objects used as parameters. This is done by searching for verified preconditions in the state; in other words, checking if all the preconditions are verified and in which combinations (as said before, there may be different options to satisfy the preconditions of an action, leading to a more rich scenario of choices for the agent). In order to speed up such process, the preconditions are checked following the order exposed in the previous section. In fact, in the action *move-car* considered before, the first condition that is checked to be true is the *not-flattire*; then, if it is found in the considered state, the next condition, in ascending order of 'dimension', is checked as well, in this case it is *vehicle -at ?from*. The process continue until one condition is verified to be false or all che conditions are met. Checking the conditions using such criteria allow the algorithm to be pretty fast: often the first, easy to check, condition is not met and the action is discarded; otherwise, it is helpful to restrict the pool of valid objects with a condition that invole only one of them, before checking the ones that involve multiple objects. Overall, this process is fast, at least while compared to the process that will be discussed in the following section, and require fractions of a second, also thanks to the criteria used.

3.2 States expansion

This is the heart of the whole process of translating the imported problem to a SSG. First of all, in the resulting game there is only one player (in this work the Max player, but it is not im-

```
for state in list of states:
  if goal is verified in state:
    add edge from state to sink
  else:
    if there are admissible actions for state:
      if action is deterministic:
        if state not in list of states:
          append resulting state as Max node and add edge between them
        else:
        add edge to existing state
      elif action is probabilisitc:
        append placeholder state as Average node
        for each resulting state (after applying action):
          if state not in list of states:
            append resulting state as Max node and add edge between them
          else:
            add edge to existing state
```

Listing 7: Pseudocode designed to generate all the states and, so, to generate the SSG. Ideally, the process is completed when all the possible states have been generated and there are no further actions to be applied to them. In many cases, this is not possible, due to the high computational requirements of checking for duplicates between states; in the actual implementation there is the possiblity to limit the generation with a time constraint (with a default value of 300 seconds).

portant, it does not change much if it is replaced with the Min player); thus the output of the algorithm presented here is a SSG with only Max and Average vertices. The algortihm used to expand the states is summerized in Listing 7. The pseudocode above is just a compact representation and it will be explained more in depth in this section. The schema followed to obtain a SSG is based on a breadth-first algorithm, where duplicates of already known states are not admitted (when a duplicate is discovered, it is used as the end of the link). One assumption made here is that states are Max nodes. This means that, given a state (and so a Max node) and an applicable actions, the effects of such action are handled in the following way:

- If the effects are deterministic, apply the action, generate a new state (that will be represented as a new Max node) and link it through an edge (from the old one to the new one).
- If the effects are probabilistic, the choice of the agent to do such action is represented as an Average node connected to the starting node. This means that Average nodes do not represent a particular state, they are just placed to show the possibility of the player to do an action where he does not know which will be the outcome. Then, from the Average node there are weighted edges that are connected to new Max nodes representing the possible outcomes of the action.

It is worth noting that:

- The Sink for the Max player is only one, whenever the goal is verified, the node used to represent such achievement is always the same.
- Every time a new state is generated, it is compared to the already known states; if there is a match, the already existing state is used as a destination, instead of generating a new node.
- If the effects of an action are limited to one probabilistic case (for example, if there is only a probability p that something occour, with a 1-p probability that nothing happens), a link in added considering that the state is not changed with a certain probability (that is calculated to reach 1).

- The whole process starts from the initial state and proceed until no more states can be generated or a time limit is reached.

Indeed, the main problem of this implementation is that many problems have too many possible states and checking for duplicates is computationally expensive, to the point that it is a serious bottleneck.

3.3 Result

The time and computational requirements of the method proposed are, in general, too high to make a comparison with the reference planner [1, 2, 3, 4]. Checking for duplicate states is the real bottleneck. In order to discuss this performance issue, the *blocksworld* is a good benchmark, because it provides problems with a different number of blocks involved (and so, a different number of states to be generated). The proposed method require 15 seconds to generate the game of problems with 5 blocks[1]; but problems with 10 blocks are too large to be generated: after 5 minutes it even fail to find a state that satisfy the goal[1]. On the other side, the planner [1, 2, 3, 4] find a solution for both of them in less than 1 second, respectively 0.03 and 0.18 seconds[1].
There are also domains that do not require to generate so many states; in particular, the climber domain can be used as an example to show the output of the proposed approach. The output of the generator is, in fact, a list of nodes and edges that characterize the generated game; that are initially handled with Pandas [5] and later saved as a csv. In the current implementation, information about states, variables, actions and so on are completely discarded. The output of the presented method is just a SSG with only Max and Average nodes. The generated game is represented in Figure 3.1, using NetworkX [6], at least for the climber domain where the nodes are limited.

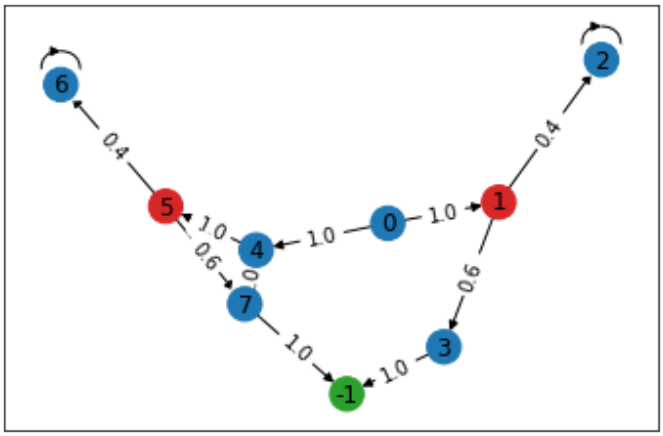

Figure 3.1: SSG generated from the climber problem. Numbers inside the nodes are their unique id, zero represent the starting node (state) and -1 is the (only) sink. Max nodes are in blue, Average nodes are in red and finally the sink (of the Max player) is in green. Number on the edges present the probability; in fact, they are different from one only when starting from a red node; note also how the sum of the two edges starting from an Average node is always equal to one.

[1] Hardware configuration of the benchmarks: ThinkPad P51 with i7 7820hq (RAM and GPU are not relevant here). Software configuration: Arch linux with Plasma DE, planner tested on a Docker container with Ubuntu 20.04

4 Linear Solver

Once the SSG is available, it is necessary to solve the game. Solving SSG in their original definition (with Max, Min and Average) is not trivial and it is computationally expensive. As explained in the previous section, the game generated in this work is a subset of the more general problem. In particular, the games generated in Section 3, as the one represented in Figure 3.1, have only Average and Max vertices; thus it is possible to use a Linear Programming algorithm to solve the game in an efficient way and without heavy computational requirements. The usual algorithm to solve such games is represented in Figure 4.1 and it is taken from [7]. The input is a SSG as just described; the algorithm consists in a minimization that is subject to a set of constraints. The solver used in this work require a small variation: The algorithm in Figure 4.1 is designed for a SSG where the options from an Average vertix are limited to two vertices with the same probability attached to them (that is 0.5 per option). This is not true, in general, for the games generated with the proposed method, so the constraint is generalized by substituting it with $v(x) \geq \sum_y p(x,y)v(y)$, where $p(x,y)$ is the probability attached to the edge from node x (the Average vertex) to node y and $v(y)$ is the value of the destination node; of course the sum is between all the possible outcome of the Average node. The updated algorithm was implemented using a Python library dedidcated to Linear Programming, PuLP.
It is worth noting that the choice of using Max nodes instead of Min ones is not relevant: the algorithm is basically the same, with a maximization instead of a minimization, but the idea behind it is the same and, so, there are no differences from a computational point of view.

Input : A simple stochastic game $G = (V, E)$ with only AVE and MAX vertices
Output: The optimal value vector v_{opt}

1 **begin**

Minimize $\sum_{x \in V} v(x)$,

2

3 subject to the constraints:

$$\begin{array}{ll} v(x) \geq v(y) & \text{if } x \in V_{\text{MAX}} \text{ and } (x,y) \in E \\ v(x) \geq \frac{1}{2} \cdot v(y) + \frac{1}{2} \cdot v(z) & \text{if } x \in V_{\text{AVE}} \text{ and } (x,y),(x,z) \in E \\ v(x) = p(x) & x \in \text{SINK} \\ v(x) \geq 0 & x \in V \end{array}$$

4 Solve this linear program to obtain an optimal value vector v. Output v.

5 **end**

Figure 4.1: Linear Programming solver for SSG with only Average and Max vertices, from [7, 8]

5 Conclusion

This project suffered about limited resources and delayed development. The main consequences of this difficulties are: The missing implementation of more pddl keywords (as discussed in Section 2); but also an optimization of the states generation process. The main drawback of the proposed method is that it requires the generation of all the possible states. This is required because one key choice taken here was to not modify the problem (and in particular its solution) by using any type of criteria. But this choice make also the method so slow that is often not useful at all. Future work should be focused on the design of a generalized criteria (something based on pruning or heuristics) in order to reduce the number of states generated.

The method proposed with this work should be a solid base for future development, as demonstrated with problems such as the climber, that has a clear and nice output seen in Figure 3.1.

REFERENCES

[1] Qumulab/planner-for-relevant-policies. `https://github.com/QuMuLab/planner-for-relevant-policies`.

[2] Christian Muise, Sheila A McIlraith, and J Christopher Beck. Improved non-deterministic planning by exploiting state relevance. In *The 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

[3] Christian Muise, Vaishak Belle, and Sheila A. McIlraith. Computing contingent plans via fully observable non-deterministic planning. In *The 28th AAAI Conference on Artificial Intelligence*, 2014.

[4] Christian Muise, Sheila A. McIlraith, and Vaishak Belle. Non-deterministic planning with conditional effects. In *The 24th International Conference on Automated Planning and Scheduling*, 2014.

[5] Jeff Reback, jbrockmendel, Wes McKinney, Joris Van den Bossche, Tom Augspurger, Phillip Cloud, Simon Hawkins, Matthew Roeschke, gfyoung, Sinhrks, Adam Klein, Patrick Hoefler, Terji Petersen, Jeff Tratner, Chang She, William Ayd, Shahar Naveh, Marc Garcia, JHM Darbyshire, Jeremy Schendel, Richard Shadrach, Andy Hayden, Daniel Saxton, Marco Edward Gorelli, Fangchen Li, Matthew Zeitlin, Vytautas Jancauskas, Ali McMaster, Pietro Battiston, and Skipper Seabold. pandas-dev/pandas: Pandas 1.4.0, January 2022.

[6] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[7] Elena Valkanova. Algorithms for simple stochastic games. Master's thesis, University of South Florida, May 2009.

[8] Cyrus Derman. *Finite State Markovian Decision Processes*. Academic Press, Inc., USA, 1970.