Sokoban: PDDL Domain planning with Single-Agent

Written by Bachelor students from King's College London
Zhuo Ying Jiang Li - zhuo.jiang_li@kcl.ac.uk,[1] Shumeng Liu - shumeng.liu@kcl.ac.uk[2]
Jiaqing Nie - jiaqing.nie@kcl.ac.uk, [3] Wenyu Zheng - wenyu.zheng@kcl.ac.uk [4]
Xin Fan Guo - xinfan.guo@kcl.ac.uk [5]

Abstract

Planning in artificial intelligence has been instrumental in a variety of applications. Game serves as good testing ground for various model planning in PDDL. This paper focuses on PDDL domain planning and problem analysis of a puzzle-based video game Sokoban. The reason for this choice is simple, Sokoban is a puzzle-based game which offers intricate dynamics and great complexity for testing planners. Secondly Sokoban also has some flexibility and offers room for expansion and creativity. As we will see in the following section, we will add our own proposed ideas i.e., tools based on traditional model. Sokoban problem file ranges with different level of complicity, thus offers complexity which can be used as benchmark for evaluating overall plan quality (Sokoban is a PSPACE-complete problem). Proposed domain will focus on deterministic model with single-agent and known initial states where state changes are based on the actions proposed by plan.

Introduction

This model simulates a well-known grid-based puzzle video game *"Sokoban"*. The original game comes with an objective for a player to push all boxes into predefined storage locations or units. Based on this construct we have added some creativity and enabled tools to be used. The domain represents 2D $N * N$ grid with helper tools such as *"trampoline"* which enables player to jump across the wall or *"bomb"* which enables player to break walls. Helper tools are placed on grids to be picked up by user. Because the domain world is represented by a grid with composite squares as single units. The convention used in the problem files are grids being vertically marked with ascending numbers and horizontally with alphabetical characters growing with lexical order (i.e., first most upper left square is represented as *"sq-a1"*, please see Figure **??**). For sake of scalability and variations of problem files, every player has been supplied with number of pre-set *"pliers"* which enables player to pull box in any direction. Any of such tools can be used only once and decreases with the use. Except of trampoline, player can pick up any number of tools bigger than one. The interesting aspect of this model is that instead of having a concrete goal

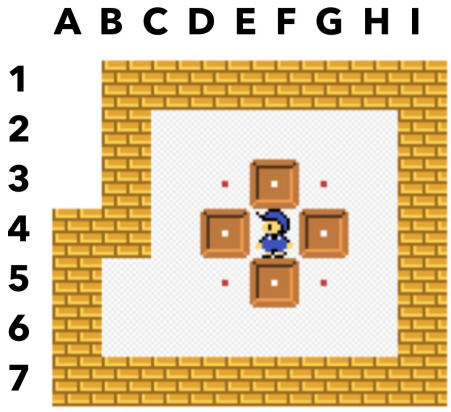


Figure 1: Example of proposed Sokoban puzzle solved with 33 steps using OPTIC planner

which defines a box to be at specific positions (i.e., box at square a3). We let planner freely choose to which storage location (represented by a hole in our domain world) player will push the box as longs as we allocate all boxes to storage locations(marked as collected box if box is pushed on the hole - storage location). In addition, player can re-allocate boxes even if it is already on storage location. Those features adds to flexibility and increments the possibility of finding a plan. For creativity considerations, we have randomly added coins on squares to be picked up. The overall aim of modified version *"Sokoban 2.0"* is to push all the boxes on storage units with minimum total cost and maximum number of coins collected on the way.

Part I: Designing a Planning Domain

In the following section we will pick the most interesting parts of our domain and introduce reasoning behind the design choices and declarations in our domain.pddl file.

Domain constants To reinforce the idea of single-agent problem we have declared a player *"p1"* as constant to avoid

ambiguity of multi-players. Throughout our problem and domain file this player will be the only actor in our proposed world.

Domain predicates Probably the most important predicate from our domain is `(collected-box ?b -box)`. As being mentioned, the proposed domain file does restrict itself on specific goals (placing boxes on predefined squares). Instead our aim is to have all declared boxes *"collected"* without explicitly mentioning where to position different boxes. We have in essence given a *free-hand* and left the decision on the planner to find the *"best-optimal solution"*.

Domain requirements Requirement declaration includes `:constraints`, `:universal-preconditions`, `:typing`, `:fluents`, `:preferences` and `:action-costs`. The reason behind declaration of above stated requirements is given as follows:

- `:typing` - reflect different object types in the domain i.e., player, box, square as single object declaration can't represent all different entities in the domain.
- `:universal-preconditions` - The use of universal quantifier spared us writing lengthy and tedious preconditions or goals. For instance, instead of listing all collected boxes (boxes on storage units) in conjunction in the form of `(:goal (and (collected-box` $b_1..b_n, b_{n+1}$`)))` where $n >= 1$ we could simply write `(:goal (forall (?b - box) (collected box)))` indicating that our *hard goal* is to place all existing boxes on some storage units.
- `:fluents` - We also exploit the feature of using complete numeric subset of PDDL and use numeric modelling throughout our domain file. Declaring numeric state variables through `:fluents` allowed numeric formulation of our actions i.e., break-noth-wall with precondition `(>(bomb-available ?p) 0)` which restrict the occurrence of the action unless we have more than $x >= 1$ bombs. Since Sokoban is PSPACE-complete problem, thus problem file evaluation can increase exponentially with size of the search space. For problem file simplification reasons, we have introduced pull action as oppose to traditional Sokoban game which only allows push . Expressed as pull-*direction* with condition `(>(pliers-available) 0)` indicating that player is able to pull if it has the tool *"plier"*. The number of pre-set pliers initialized in the ground function increases with difficulty of the problem file.
- `:preferences` - Declaration of preferences in the domain file enable us to achieve soft goals and define properties that we would like to achieve but are not compulsory. Thus, we have exploited this feature and defined that it will be preferable for the player to collect as much coins as possible. Implemented through `(:goal (preference (> (collected-coins) 0))`.
- `:action-cost` - For optimization reasons, every action in our action schema comes with a cost which means that every move will incur penalty. Our goal is to minimize incurred action cost in planning metric so minimum number of steps are taken to reach the objective. This was realized through declaration of `:function (total-cost)`,`:effect (increase (total-cost) 1)` and `:metric (minimize (total-cost))`.

Domain actions The introduced action schema tries to mimic all possible moves taken by player. Every single action in the domain file is named in self-explanatory way i.e., pull-*direction*, push-*direction*, jump-*direction*, break-*direction* and pick-up-*tool*(bomb, coin, trampoline). Implementing domain file using this kind of architecture allows better trackability of generated plan. It is also important to mentioned here that push and pull actions come in two different version i.e., push-*direction*, push-*direction*-to-hole. Former indicating a push to free location , latter indicating a push to hole or storage location (See following section for reason).

Planner constraints and proposed solutions **OPTIC** planner provided lacks several important functionalities such as support for the **ADL**, conditional effects and negative preconditions. For such reasons we remodelled the action schema to avoid those features. As some action's preconditions are made up of a single or clauses of *k* terms, we have remodelled the problem to create *k* copies of an action and used one term for each as a regular precondition i.e., instead of writing in the `:effect (when(hole-at ?box-after-move))` to represent condition when a box is moved on a hole after push or pull. We have re-written this into two push and pull actions i.e., push-up with `:precondition (no-hole-at ?box-after-move)` and push-up-to-hole with `:precondition (hole-at ?box-after-move)` and `:effect (collected-box ?box)`. As being mentioned **OPTIC** also does not support negative precondition, therefore all predicates which needs to be negative in the precondition have been represented by two static predicates i.e., has- trampoline and has-no-trampoline to indicate that a player has this tool. While in principle this does allows **OPTIC** to work with this problem, but as a result it reduces efficiency and come at expense given increased number of actions required.

Possible domain improvements Although axioms do not increase language expressiveness, but it will be great to include *derived predicates* as it decreases the domain description size and plan length, compiling away from axiom can result in an exponential growth in number of actions. In our case instead of listing all squares which are above one another in the problem file i.e., `(above sq-a1 sq-a2)` indicating square *a1* being above *a2*. We could have written `((: derived (above? a ?b - squares) (and (left-to b? ?c) (above c? ?a))` that recursively defines relationship between pairs of squares.

Part II: Problem analysis

In this section we will analyse the planner's ability to solve problems in this domain and whether some domain features will make it harder to solve for the planner. We will also compare the plan quality of the plans to a selected benchmark.

Difficulty measure as problem selection criteria

To select a robust and more complete range of problems in the puzzle domain, we chose to classify the problems according to a difficulty measure, and then select a problem suite which covers low difficulty to high difficulty problems. However, finding which problem parameter is the most representative difficulty measure is not an easy task. There are many plausible measures, such as the number of steps taken to push all the boxes to the holes, the time taken by a solver (human or artificial intelligence planner) to solve the puzzle or the percentage of successful plays. We will use the first one as our difficulty measure heuristics for this research project for the sake of simplicity.

Selected problem suite

Specifically, we will order the problems according to the minimum number of steps taken by a human player to solve it, starting with a problem with best score 25 steps up to a problem with best score 429 steps. The data of the best scores of human players are collected from the Microban website. **Please cite the Microban website!!**. This is more representative than trying to solve the problems ourselves and using our best score to compare to that of the OPTIC planner. Given that the Microban community is an active community with global Sokoban grandmasters, it is likely that the best score data from the website is the global minimum number of steps to solve each problem.

These problems are, however, just testing a part of the expressiveness of our domain, because our PDDL domain can also encode the use of tools — bombs and pliers. Thus, we run the same set of problems but with added tools (each initially with 10 available pliers) to observe how planner would handle those additional actions.

Anomalies in difficulty rating

Before diving into the analysis of the planner's plans, note that there may be anomalies in our difficulty assessment and the planner's behavior. There may be some problems which are easier to solve according to our difficulty measure but the planner is not able to find a solution. While we expect that the plan quality compared to a benchmark decreases as the difficulty increases, it is possible to find counterexamples, because the difficulty measure (smallest number of steps required to solve by human players in Microban) is merely a heuristics.

Plan quality measure

In a Sokoban game, players are rated according to the number of steps they use to solve the game, and the lowest number is marked as their best score. Therefore, using the number of steps as plan quality measure fits well our purpose.

In some online games, players are also ranked according to the time taken to solve the level. However, the major problem with this quality measure is its strong dependence on hardware in which the planner is running on. It is thus not worth comparing the planner's solve time and human players' average time.

Another possible quality measure is the number of coins the planner collects during its plan trajectory. However, we tested the OPTIC planner with a trivial game instance where there is a coin in the initial position of the player and the player only has to push the box forward to solve the game. It turns out that the planner's plan does not collect the immediately available coin.

From what is stated above, we will choose the number of steps as the plan quality measure for this reasearch. The benchmark is the best player score in Microban website (**citeme!!!**).

Plan quality analysis

The diagram in **Figure 1** illustrates how the solution given by the planner (without tools) differs from the benchmark in terms of number of steps taken. The trend from problem 1 to problem 15 suggests that number of steps is effectively a good estimation of problem difficulty. The planner can sometimes find the optimal solution for problems with under 33 moves. Between 40 to 200 steps, even if the problem is solvable by the planner, the difference between the benchmark and the plan's number of steps is increasing. Beyond that, the problem size becomes unsolvable. However, the planner managed to solve the problem 16, when it requires 429 steps for a human player to solve it. Problem 16 is distinctive because it involves only one box, while other problems with difficulty heuristics equal to 200-plus steps consist of at least 3 boxes.

This discovery implies that, the number of steps required is not the mere factor that makes a problem difficult, the number of boxes or other underlying characteristics of the problem also play a role. A similar point is also suggested in a research study: there is no single difficulty measure which gives consistent evaluation of the actual difficulty for all problem instances (**?**).

Moreover, for a human player, providing additional tools, such as pliers, would only make the problems easier to solve. However, when it comes to planners, it seems not the case. When experimenting with the problem files, we noticed that, despite it would solve the problem without any tools, the planner is unable to give any solution for problems 10 and 11 when provided with 10 pliers. A plausible explanation is illustrated in in **Figure 2** and **Figure 3**.

By number of steps

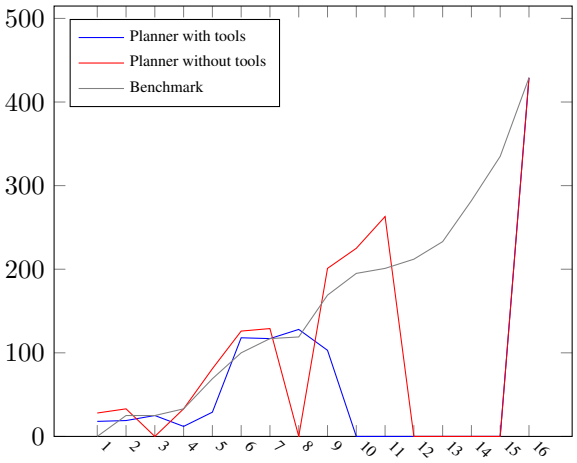
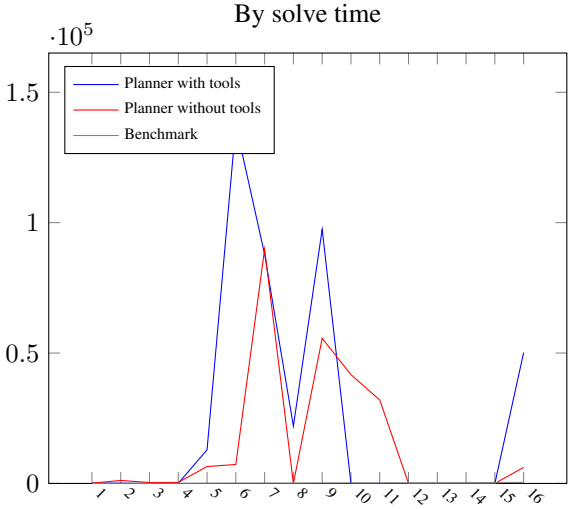
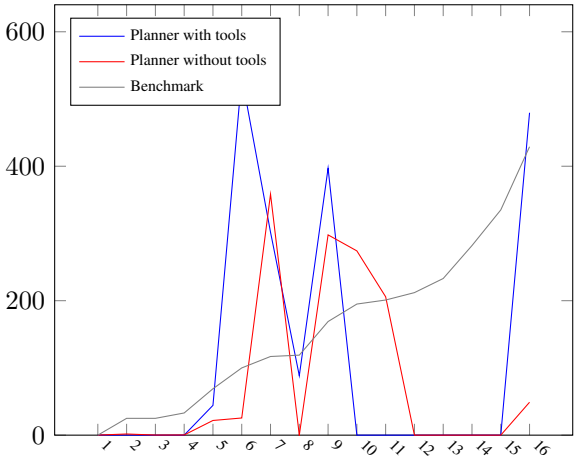


Figure 2: By number of steps

By solve time



By number of states evaluated



These two diagrams show that, due to more actions become possible, when provided with additional tools, planner would evaluate more states and take a longer time to derive a solution. This implies the planner fails to effectively prune out bad uses of the tool and wasted a lot of resources exploring unsolvable states.

Plan quality analysis improvements

The only measure of the plan quality was whether the designed plan solves the problem and how many steps was taken to achieve the goal. Perhaps a better evolution strategy is to define a set of preferable soft goals which is preferable to achieve but not compulsory. As the plan results in a higher number of correctly allocated boxes, the higher the quality of the plan. This is helpful to analyze problems of higher difficulty in more detail rather than treating them all as unsolvable. We can observe how the plan quality over the benchmark increases or deteriorates, based on the number of achieved soft goals.

If OPTIC supports `:preference` and `forall` PDDL features we could write

```
(preference allBoxesAllocated
  (forall (?b - box) (collected-box ?b))
)
```

or

```
(forall (?b - box)
  preference allocated (collected-box ?b)
)
```

, the former incurring a penalty if any box is not allocated and the latter incurring a penalty for each non-allocated box. The overall quality measure is given by the metric

```
(:metric minimize
  (* 2 (is-violated allocated)
  (*100 (is-violated allBoxesAllocated)
)
```

giving a linear combination of the preference of plan. A plan that reaches a goal state satisfying all preferences will have penalty of 0.