Sokoban: PDDL Domain planning with Single-Agent

Written by Bachelor students from King's Collage London
[1]Zhuo Ying Jiang Li - zhuo.jiang_li@kcl.ac.uk , 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 1). 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

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

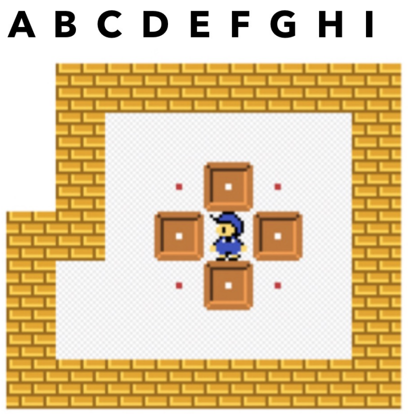

Figure 1: Example of proposed Sokoban puzzle solved with 33 steps using OPTIC planner

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 designs 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.