CENTRALE NANTES

M2 - CORO-EMBEDDED REAL-TIME SYSTEMS

Automated Planning Tool

Author:
Ragesh RAMACHANDRAN
Gopi Krishnan REGULAN

Supervisor:
Loïg JEZEQUEL

January 24, 2019

Contents

Chapter 1

Introduction

1.1 Objective

The goal of the practical session is to implement a automated planning tool. This tool will take as input the fragment of PDDL that is used in planning competitions.The A* algorithm will be used to solve planning problems. One heuristic has to be implemented and the efficiency of the implementation has to be evaluated.

1.2 Work to be done

1.2.1 Parsing PDDL file

As a first step you have to implement a tool that can read a PDDL file written with the subset of PDDL that involves STRIPS, action costs, negative preconditions, and conditional effects. This tool should output a grounded version of the planning problem given in input as a PDDL file.

1.2.2 A* algorithm

Implement the A* algorithm and the algorithm should run on the grounded version of a planning problem obtained with the tool developed at the first step. At the end there should be a single tool that takes as input a planning problem described as PDDL files and outputs a cost-optimal plan for this problem.

1.2.3 Heuristic

Implement one of the heuristics studied during lectures and use it with the implementation of A*.

1.2.4 Evaluation of the efficiency of your tool

Finally, evaluate the efficiency of the tool developed. Propose and justify an evaluation protocol and apply it.

Chapter 2

Automated Planning

2.1 Introduction

In automated-planning research the word planning refers specifically to plans of action. Planning is an important component of rational behavior and plans are needed in many different fields of human endeavor, and in some cases it is desirable to create these plans automatically. A conceptual model is a simple theoretical device for describing the main elements of a problem.

1. **State transition system:** A formal model of the real-world system for which we wanted to create plans. A state transition system is a 4 tuple $\sum = (S, A, E, \gamma)$ where
   - $S = \{s0, s1, s2, ...\}$ is the set of states.
   - $A = \{a1, a2, ...\}$ is the set of actions whose occurrence is controlled by the plan executor.
   - $E = \{e1, e2, ...\}$ is the set of events whose actions are not controlled by the plan executor.
   - $\gamma = S \times (A \cup E) \rightarrow 2^s$ is a state transition function.
2. **Controller:**Controller performs action that changes the state of the system. The input of the controller consists of plans and observations about the current state of the system. The controller's output consists of actions to be performed in the state-transition system.
3. **Planner:**Planner produces plans that drives the controller.The planner's input is a planning problem, which includes a description of the system, an initial situation and some objective

Classical planning must satisfy the following set of the restrictive assumptions:

- **Assumption A0:** The system $\sum$ has a finite set of states.
- **Assumption A1:** The system $\sum$ is fully observable.
- **Assumption A2:** The system $\sum$ is deterministic.
- **Assumption A3:** The system $\sum$ is static.
- **Assumption A4:** Goals are defined by final states only, not by the states traversed
- **Assumption A5:** A solution plan to a planning problem is a linearly ordered finite sequence of actions.
- **Assumption A6:** Actions and events have no duration, they are instantaneous state transitions.
- **Assumption A7:** Changes in $\sum$ while planning are not possible.

2.2 Planning Problem

A planning problem P is given by, $P = (O, s_0, g)$ where

- O is a set of operators and $O = (name(O), precond(O), effects(O))$
  - $name(O)$is the name of the operator.
  - $precond(O)$ are set of literals that must be true to use operator O
  - $effects(O)$are set of literals that O will make true.
- s_0 is the initial state - a set of atoms.
- g the goal state - a set of literals, every state that satisfies g is goal state.

A plan π is a solution for $P = (O, s_0, g)$ if

- π is executable in s_0
- The resulting state$\gamma\left(s_0, \pi\right)$ satisfies g

In this project the classical representation of the Planning tasks is used and is defined in PDDL file using STRIPS. A cost-optimal plan for this problem is determined by the A* algorithm.

2.3 PDDL

PDDL - Planning Domain Definition Language, is a recent attempt to standardize planning domain and problem description languages. It was developed mainly to make the International Planning Competition series possible. The components of a PDDL planning task are:

- **Objects**: Things in the world that interest us.
- **Predicates:** Properties of objects that we are interested in; can be true or false.
- **Initial state:** The state of the world that we start in.
- **Goal specification**: Things that we want to be true.
- **Actions/Operators:** Ways of changing the state of the world.

Planning tasks specified in PDDL are separated into two different files:

- **Domain file:** The domain definition contains the domain predicates and operators called actions
- **Problem file:** The problem definition contains the problem description, the initial state and the goal of the problem.

2.4 A* Algorithm

procedure ASTAR(O, s_0, g, c)
  $nexts \leftarrow \{s_0\}$; $pred[s_0] \leftarrow nil$; $f[s_0] \leftarrow 0$; $\forall s \neq s_0, f[s] \leftarrow +\infty$
  while $nexts \neq \emptyset$ **do**
    take s in $nexts$ which minimizes $f(s) + h(s)$
    if s satisfies g **then return** π built from $pred[s]$
    end if
    $E \leftarrow \{a \mid a$ is a ground instance of an operator in O,
        and $precond(a)$ is true in $s\}$
    for all $a \in E$ **do**
      if $f(\gamma(s,a)) > f(s) + c(a)$ **then**
        $nexts \leftarrow nexts \cup \{\gamma(s,a)\}$; $pred[\gamma(s,a)] \leftarrow (s,a)$
        $f[\gamma(s,a)] \leftarrow f(s) + c(a)$
      end if
    end for
  end while
end procedure

Figure 2.1: A* Algorithm

2.5 Heuristics

A heuristic is a function h that, for any state s, gives an estimate of the cost to reach the goal from a given state.
We studied some of the heuristics which is explained below:

- Flexible abstraction heuristics
- Planning with pattern databases
- Delete relaxation heuristics

2.5.1 Flexible abstraction heuristics

An approach to deriving consistent heuristics for automated planning, based on explicit search in abstract state spaces is described. The key to managing complexity is interleaving

composition of abstractions over different sets of state variables with abstraction of the partial composites. The approach is very general and can be instantiated in many different ways by following different abstraction strategies.

In particular, the technique subsumes planning with pattern databases as a special case. Moreover, with suitable abstraction strategies it is possible to derive perfect heuristics in a number of classical benchmark domains, thus allowing their optimal solution in polynomial time.

Atomic projections and synchronized products can fully capture the state-transition semantics of a SAS+ task. However, for all but trivial tasks we cannot explicitly compute the product of all atomic projections: At some point, the abstract transition graphs become too large to be represented in memory. In the case of PDB heuristics, the memory limit translates into an effective limit on the number of variables that can be included in any single projection.

1. Two abstractions can be merged by replacing them with their synchronized product.
2. An abstraction can be shrunk by replacing it with a homomorphism of itself.

To keep time and space requirements under control, we enforce a limit on the size of the computed abstractions, specified as an input parameter N. Each computed abstraction, including the final result, contains at most N states. If there are more than N states in the product of two abstractions A1 and A2 (i. e., if the product of their state counts exceeds N), either or both of the abstractions must be shrunk by a sufficient amount before they are merged. The general procedure is shown in Fig. Note that the procedure has two important choice points:

Merging strategy: the decision which abstractions A1 and A2 to select from the current pool of abstractions. shrinking strategy: the decision which abstractions to shrink and how to shrink them, i. e. which homomorphism to apply to them.

Shrinking strategy the decision which abstractions to shrink and how to shrink them, i. e. which homomorphism to apply to them.

We refer to the combination of a particular merging and shrinking strategy as an abstraction strategy. To obtain a concrete algorithm, we must augment the generic abstraction algorithm in Fig. with an abstraction strategy.

Assuming that N is polynomially bounded by the input size (e. g., a constant) and that the abstraction strategy is efficiently computable, computing the abstraction requires only polynomial time and space. As argued previously, the resulting abstraction heuristic is admissible and consistent. Even though these important properties hold independently of the choice of abstraction strategy, selecting a suitable strategy is of paramount importance, as it determines the quality of the resulting heuristic. In the following

section, we introduce one particular abstraction strategy, which we use for experimentally evaluating the approach.

2.5.2 Planning with pattern databases

Heuristic search planning effectively finds solutions for large planning problems, but since the estimates are either not admissible or too weak, optimal solutions are found in rare cases only. In contrast, heuristic pattern databases are known to significantly improve lower bound estimates for optimally solving challenging single-agent problems like the 24-Puzzle or Rubik's Cube.

Heuristic search is currently the most promising approach to tackle huge problem spaces but usually does not yield optimal solutions. The aim is to apply recent progress of heuristic search in finding optimal solutions to planning problems by devising an automatic abstraction scheme to construct pre-compiled pattern databases. Our experiments show that pattern database heuristics are very good admissible estimators. Once calculated, our new estimate will be available in constant time since the estimate of a state is simply retrieved in a perfect hash table by projecting the state encoding. We will investigate different pruning techniques to reduce the large branching factors in planning. There are some known general pruning techniques such as FSM pruning, Relevance Cuts and Pattern Searches that should be addressed in the near future.

Although PDB heuristics are admissible and calculated beforehand, their quality can compete with the inadmissible FF-heuristic that solves a relaxed planning problem for every expanded state. The estimates are available in a simple table look-up, and, in contrast to the FF-heuristic, A* finds optimal solutions. Weighting the estimate helps to cope with difficult instances for approximate solutions. Moreover, PDB heuristics in A* can handle directed problem spaces and prove unsolvability results.

One further important advantage of PDB heuristics is the possibility of a symbolic implementation. Due to the representational expressiveness of BDDs, a breadth-first search (BFS) construction can be completed with respect to larger parts of the planning space for a better quality of the estimate.

The exploration yields a relation H(estimate,state) represented with a set of Boolean variables encoding the BFS-level and a set of variables encoding the state. Algorithm BDDA*, a symbolic version of A*, integrates the symbolic representation of the estimate. Since PDBs lead to consistent heuristics the number of iterations in the BDDA* algorithms is bounded by the square of the solution length. Moreover, symbolic PDBs can also be applied to explicit search. The heuristic estimate for a state can be determined in time linear to the encoding length.

2.5.3 Delete relaxation heuristics

Heuristic functions based on the delete relaxation compute upper and lower bounds on the optimal delete-relaxation heuristic h+, and are of paramount importance in both optimal and satisfying planning. Here we introduce a principled and flexible technique for improving h+, by augmenting delete-relaxed planning tasks with a limited amount of delete information. This is done by introducing special fluents that explicitly represent conjunctions of fluents in the original planning task, rendering h+ the perfect heuristic h+ in the limit.

Chapter 3

Implementation

3.1 PDDL Parsing

The tool is developed in python 3.0 without using any external libraries for PDDL parsing and the parsing of PDDL files are implemented in the file *pddl_parser.py*. In order to make the work more efficient object oriented approach is followed through out the program. Five classes were developed and they are:

1. **class PDDL_Parser** The PDDL parsing is implemented in the *class PDDL_Parser* with the following member functions
    - *scan_tokens(self, filename)*: The Planning problem is read from the PDDL file and the tokens are stored in the form of a *list* for ease of operation.
    - *parse_domain(self, domain_filename)*: The planning domain file is parsed in this function. The input file is scanned using the function *scan_tokens* and the tokens are stored in a list. If the token name is *action* then the function *parse_action* is used to parse the actions.
    - *parse_action(self, group)*: The action name, action parameters, positive and negative preconditions, positive effects, negative effects and the cost is parsed from the input file.
    - *parse_problem(self, problem_filename)*: The objects, initial states, positive and negative goals are parsed from the planning problem.
    - *split_propositions(self, group, pos, neg, name, part)*: The goal state is separated into positive and negative goal states and this is useful while implementing relaxation heuristics.

2. **class Domain**: The grounding of domain is performed by the function *ground(self, objects)*.
3. **class Problem**:The problem is parsed from the PDDL file and the *initial* and the *final* states are determined.
4. **class Action**: The actions that are parsed from the PDDL file are initialized as objects.
5. **class State**: States that are necessary for the Planning of problem from initial to goal location is determined. This is then used for A star algorithm to estimate the plan.
6. **class GroundedAction**: The preconditions and the effects of the action are grounded.

3.2 A* Algorithm

The algorithm is implemented in the file *planner.py* and the planner takes the problem, initial state, final state and heuristics which is already determined by the *PDDL_Parser*.

3.3 Heuristics

For the A star algorithm implemented for the planning problem we used the admissible heuristic which is implemented in the *planner.py*. The cost of each path is assumed to be 1 and the cost to reach the goal from the present state is determined.

Chapter 4

Testing the Tool

The directory of Planning tool is as follows:

```
Automated-Planning-Tool
├── PDDL
│   ├── domain_1.pddl
│   ├── domain_2.pddl
│   ├── problem_1.pddl
│   └── problem_2.pddl
├── main.py
├── pddl_parser.py
├── planner.py
└── README.md
```

4.1 Planning Problem

The *problem_2.pddl* is used as input to the tool

```
(define (problem BLOCKS-10-0)
(:domain BLOCKS)
(:objects D A H G B J E I F C )
```

```
(:init (clear C)
      (clear F)
      (ontable I)
      (ontable  F)
      (on C E)
      (on E J)
      (on J B)
      (on B G)
      (on G H)
      (on H A)
      (on A D)
      (on D I)
      (HANDEMPTY))

(:goal (and(on D C)
         (on C F)
         (on F J)
         (on J E)
         (on E H)
         (on H B)
         (on B A)
         (on A G)
         (on G I))))
```

4.2 Domain of the Problem

The *domain_2.pddl* is used as input to the tool

```
(define (domain BLOCKS)
(:requirements :strips)
(:predicates (on ?x ?y)
         (ontable ?x)
         (clear ?x)
         (handempty)
         (holding ?x))

(:action pick-up
```

```
   :parameters (?x)
   :precondition (and (clear ?x) (ontable ?x) (handempty))
   :effect (and (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)))

(:action put-down
   :parameters (?x)
   :precondition (holding ?x)
   :effect (and (not (holding ?x))
            (clear ?x)
            (handempty)
            (ontable ?x)))

(:action stack
   :parameters (?x ?y)
   :precondition (and (holding ?x) (clear ?y))
   :effect (and (not (holding ?x))
            (not (clear ?y))
            (clear ?x)
            (handempty)
            (on ?x ?y)))

(:action unstack
   :parameters (?x ?y)
   :precondition (and (on ?x ?y) (clear ?x) (handempty))
   :effect (and (holding ?x)
            (clear ?y)
            (not (clear ?x))
            (not (handempty))
            (not (on ?x ?y)))))
```

4.3 Execution of Tool

To Run the tool first run the parser for the problem and domain *pddl* files execute the python file *main.py*.

```
python3 main.py
```

```
#!/usr/bin/env python3
import sys
import pprint
from pddl_parser import PDDL_Parser
from planner import *

if __name__ == "__main__":
    '''
    PDDL parser
    '''
    domain = "./PDDL/domain_2.pddl"   #contains init and goal states
    problem = "./PDDL/problem2.pddl"       #contains problme description
    parser = PDDL_Parser(domain_file = domain, problem_file = problem)
    '''
    PDDL planner
    '''
# To generate the plan for the planning problem uncomment the section below.
    # plan = planner(parser,verbose=True)
    # for action in plan:
    #     print(action)
```

The is done because the A star algorithm takes very long to finish execution so parsing the files ensures that the input for the planner is error free. Now remove the comments of the Planner section to generate plan for the planning problem and run the tool again.

```
python3 main.py
```

```
#!/usr/bin/env python3
import sys
import pprint
from pddl_parser import PDDL_Parser
from planner import *
```

```
if __name__ == "__main__":
    '''
    PDDL parser
    '''
    domain = "./PDDL/domain_2.pddl"   #contains init and goal states
    problem = "./PDDL/problem2.pddl"       #contains problme description
    parser = PDDL_Parser(domain_file = domain, problem_file = problem)
    '''
    PDDL planner
    '''
# To generate the plan for the planning problem uncomment the section below.
    plan = planner(parser,verbose=True)
    for action in plan:
        print(action)
```