PDDLyte
A Partial Implementation of The Planning Domain Definition Language

John Martin Jr.
jdm2213@columbia.edu

February 9, 2014

Abstract

The PDDLyte language, whose name derives from the Planning Domain Definition Language (PDDL)[1], is a specification language used to formulate and solve planning problems. Similarly to PDDL, problems are specified with an initial state, a goal description, and a domain on which to plan over. From there, PPDLyte uses causal reasoning to deduce solutions, provided they exist, as a sequence of actions that map the initial state to the goal state(s). The PDDLyte language is limited in comparison to its predecessor, in that it will only support classical planning problems for a single agent: finite, fully-observable, deterministic, static environment descriptions. Another distinguishing characteristic of PDDLyte is in the way it's compiled.

Current PDDL implementations use LISP-based interpreters to verify the solutions. For most applications, this is where the life of PDDL ends. The PDDLyte implementation will go further and be compiled to C code, then to X86 assembly. With this design, the high-level reasoning of PDDLyte solutions will be amenable to systems-level C code interfaces.

Background

A planning problem asks if a goal description can be acheived from its initial state [4]. If there's a tree of all possible actions from the initial state to every subsequent successor state, and the tree is indexed in such a way that allows us to read off the solution. This tree is of exponential size, which makes this approach impractical. A planning graph is a polynomial-size approximation to the full tree. This approach can't definitively answer if the goal is reachable from the initial state, btr it does estimate the number of steps it takes to reach thee foal. The esitmate is always correct when it reports the goal is not reachable; hence it never overestimates and is an admissible hueristic. Planning graphs work for propositional planning problems – ones with no variables.Each action at level Ai is connected to its preconditions at Si and its effects at S_{i+1}. So a literal appears because an action caused it, wut we alo want to say that a literal can persist inf no action degates it. this is represented as a persistence actoin – called a no-op.

What we build is a structure where ever action level constatins all the actions that are applicable in the ith state level along with the constraints saying that two actions cannont both be executed at the same level. every state level contains tll the literals that could result from any possible choice of actions in the previous action level along with the constraints saying which paits of literals are bot possible. it is important to not that the proces of constructing the planning graph does not require choosing among action, which woudl entail conbinatorial search. instaed, it just records the impossibility of certain choices using mutex links.

a planning fraph is polynomial in size of the planning problem. for a plannign problen with l literals and a actions, ach ith sstate has no more than l nodes and l squared mytex linkgs, and $2(al + l)$ precondition and effect links. thus an entire graph iwht n levels had size $O(n(a + l)^2)$. the time to build the fraph had the same complexity.

if any foal literal fails to appear in the final level of the graph, then tjhe problem is unsolvable.

i hope to encode the state transistion system which can be defined as a tuple $\Sigma = (S, A, E, \gamma)$. this formalism doesn't define an initial state and goal state.

planning is the explicit deliberation process that chooses and organizes actions by anticipating their outcomes – how to achieve some prestate objectives computationally. understanding intellegnt behavior involves understanding plans that were made. and how those plans were formulated and executed.

two kinds of planning: domain specific, and domain independent.
- domain-specific – perception planning, path planning, motion planning, manipulation planning, comminication planning
- domain-independent planning – generic representations and techniques to plan for commonalities in plans.
- there's no consideration of time. it is implicit. plans are assumed to be sequential list of actions, i.e. no parallelism.

objects require typinf extension. iniitial state descirbes the topology.

a olan is a sequence of action $\pi = [a_1, ..., a_k]$ with $k > 0$. not every plan is a solution. therefore, a plan is a solution to a planning problem if the state transistion funciton $\gamma(s_0, \pi) = G$. Solutions are redundant if there exists a subsequence of

i'm using the srips represestation of classical planning representation.i use first order literals to define actions.

Underlying the PPDLyte language is a graph-based planner that generates solutions to the program's problem. The algorithm which computes the solution is the GraphPlan

[?] algorithm will be used to solve problems.

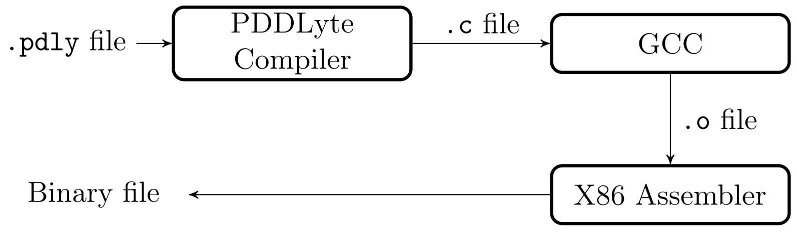
- Phase one — plan graph expansion: creates graph encoding pair-wise consistency and reachability of actions and propositions from initial state. Graph includes, as a subset, all plans that are complete and consistent.
- Phase two — Solution extraction: graph is treated as kind of a constraint satisfaction problem. Selects whether or not to perform each action at each time point by assigning CSP variables and testing consistency.

the solver works under the premise that an action can be executed in a given state, provided the preconditions are satisfied. That is $a \in A_s \leftrightarrow s \models P_s$

Motivation

Motivation for creating PDDLyte.

PDDLyte Pipeline



Lexicon

The syntax in PDDLyte derives from PDDL. The components of each language focus around the classical-planning representation of STRIPS, which itself is a restricted, state-transition system $\Sigma = (S, A, \gamma)$ over a function-free, first-order language $\mathcal{L}$. For more information on the formalism of PDDL, see [2].

In essence, PDDLyte will interpret conjunctions of predicates, themselves comprised of grounded atoms and sets literals, in order to solve problems. Therefore, the language must represent atoms, literals, and booleans. Additionally, the PDDLyte language will support integer arithmetic for various quantifications.

Primitive types

Boolean — True or false values.
Integers — Any element of the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
Objects — Generic datatypes identified with symbols, consisting of one or more character elements from the ASCII set. Fluent objects must be prefixed with a question mark: `?<var>`.

Structured types

Lists — A set of components separated with spaces and enclosed with parentheses: the component's types can be dissimilar.
Atoms — An atom is a predicate with a specified number of object arguments. An atom is said to be grounded when they relate to specific objects with values – not variables.

Domains

Domains can be thought of as universe. What describes these universes are states, actions and means to use actions at states to make transitions. Only a single domain may be defined per file.

```
(define (domain <name>)
           (<types_def>)
         (<actions_def>)
     (<predicates_def>))
```

Types — Types are symbols that specify objects of the domain. This attribute is an extension of PDDL, but will be inherently supported with PDDLyte.
Actions — Actions are the operators that transition the system between states. These are represented as triples, $a = (name, precondition, effect)$, and are the basic elements of a solution [3]. In PDDL, an action's name is considered both the unique symbol and a set of parameters that define the operation. The pre-conditions must be satisfied for the transition to take place. Furthermore, the effects must be valid according to an active problem description. If no pre-conditions are specified, then an action is always valid.

```
(:action   <name>
           :parameters      (<param_def>)
           :precondition (<precond_def>)
           :effect        (<effect_def>))
```

Parameters are a list of atoms used in the action's precondition and effect conjunctions.

```
:parameters (?<name> - <type> ... ?<name_n> - <type_n>)
```

Preconditions are propositions that must be true for an operator to be applied. This is expressed as a logical conjunction of literals. *Literals* are positive or negative atoms.

```
:precondition (and (<literal>) ...)
```

Effects describe changes that occur when an action is completed at the successor state. This is expressed as a logical conjunction of literals. Furthermore, developers should be mindful to balance the preconditions with the effects; as predicate states are not automatically negated by transitions.

```
:effect (and (<literal>) ...)
```

Predicates — Predicates define relationships between object variables. These can be static relations that hold from state to state, or fluent relations. Each predicate is defined with a symbolic name and one or more object name-type groups; where the object name is separated from the type with a dash:

```
(:predicates (pred ?<name> - <type> t...) ...)
```

Problems

Problems are triples defined as $P = (\Sigma, s_0, G)$. This inlcudes a domain Σ an initial state s_0, and a set of ground literals describing the goal condition G.

```
(define (problem <name>)
              (<domain>)
          (<objects_def>)
       (<init_state_def>)
    (<goal_descrip_def>))
```

Objects — Objects refer to those used in the problem configuration. Each object is declared with a symbolic name and one or more object name-type groups; where the object name is separated from the type with a dash:

```
(:objects ?<name> - <type> ... ?<name_n> - <type_n>))
```

Initial State — The initial state defines the predicates that are true in the system's starting configuration. This can be any valid state within the domain and is written as a conjunction of ground literals.

```
(:init <atom_list>)
```

Goal Description — The goal description defines the predicates that are true in the system's final configuration. This is a conjunction of grounded literals.

```
(:goal <literal>)
```

Operators

Operators will be specified using prefix notation; where the operator is placed to the left of its arguments. This convention is adopted from PDDL, which inherits its syntax from LISP to simplify parsing.

Comments — Comments begin with a semicolon (`;`) and terminate at the next new line. Furthermore, they do not nest and may not be composed within comments.

```
; commentary ends when the line breaks
```

Arithmetic operators

A minimal level of integer arithmetic will be supported in PDDLyte.

Assignment — Explicit assignments are only permitted with integer values:

```
(set <atom> <int>)
```

Modulus — The modulus of two integers can be computed:

```
(mod <int> <int>)
```

Addition — Integer addition is supported for one or more numbers:

```
(+ <int> ... <int_n>)
```

Subtraction — Integer subtraction is supported for one or more numbers:

```
(- <int> ... <int_n>)
```

Conditional Operators

Equal — Comparing the value of one or more integers is accomplished with the = predicate. The result evaluates to true if every argument is equal to the others. Otherwise, the result is nil.

```
(= <int> ... <int_n>)
```

Less than — The < predicate is used to compare if the arguments are in monotonically increasing order; if so, the result is true. Otherwise, the result is nil.

```
(< <int> ... <int\_n>)
```

Greater than — The > predicate is used to compare if the arguments are in monotonically decreasing order; if so, the result is true. Otherwise, the result is nil.

```
(> <int> ... <int\_n>)
```

Less than or equal to — The <= predicate is used to compare if the arguments are in monotonically non-decreasing order; if so, the result is true. Otherwise, the result is nil.

```
(<= <int> ... <int\_n>)
```

Greater than or equal to — The >= predicate is used to compare if the arguments are in monotonically non-increasing order; if so, the result is true. Otherwise, the result is nil.

```
(>= <int> ... <int\_n>)
```

Boolean Operators

Conjunction — Logical conjunctions are formed with the `and` predicate:

```
(and <literal>)
```

Disjunction — Logical disjunctions are formed with the `or` predicate:

```
(or <literal>)
```

Negation — The value of a logical conjunction of literals is inverted with the `not` predicate. The function returns true if its argument is nil, otherwise false.

```
(not <literal>)
```

Keywords

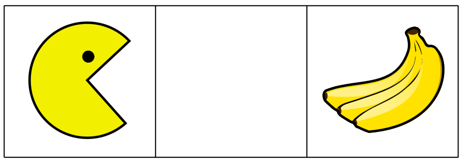
Keyword	Description
`define`	instantiates a domain or problem specification
`domain`	domain specification
`problem`	problem specification
`:types`	specifies a list of objects
`:action`	specifies an action
`:precondition`	specifies an action's preconditions
`:effect`	specifies an action's effects
`:parameters`	specifies an action's parameters
`:predicates`	specifies the domain predicates
`:objects`	specifies a problem's objects
`:init`	specifies a problem's initial state
`:goal`	specifies a problem's goal description

Example program

A program in PPDLyte consists of a domain and problem statement. Executing the program solves the problem and generates output.

Pacman

This program is simple to conceptualize, and therefore should be a good first step in generating plans. Pacman starts in one side of the world, with the goal of eating the bananas in another square. The solution is to simply move right twice.



Example PDDLyte code specifies the problem.

```
(define (domain pman)
  (:predicates
   (adj ?square-1 ?square-2)
   (at ?what ?square)
  )

  (:action move
    :parameters (?who ?from ?to)
    :precondition (and (adj ?from ?to)
                       (at ?who ?from))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to))
  )
)

(define (problem pman_prob)
  (:domain pman)
  (:objects
   sq_11 sq_12 sq_13
   pacman
   banana
  )

  (:init
   (adj sq_11 sq_12) (adj sq_12 sq_11)
   (adj sq_12 sq_13) (adj sq_13 sq_12)

   (at banana sq_13)
   (at pacman sq_11)
  )

  (:goal (and (at pacman sq_13)
              (at banana sq_13))
  )
)

; the optimal plan consists of two moves
;---------------------------------------
;  (move pacman sq_11 sq_12)
;  (move pacman sq_12 sq_13)
```

Bibliography

[1] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl — the planning domain definition language. Technical report, AIPS Planning Competition Committee.

[2] Malik Ghallab, Dana Nau, and Palo Traverso. *Automated Planning, Theory and Practice.* Morgan Kaufmann.

[3] Nils J. Nilsson and Richard E. Fikes. Strips: A new approach to the application of theorm proving to problem solving. Technical report, Artificial Intelligence Group, Stanford Research Institute.

[4] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall.