jraffplan:

a Java implementation of graphplan

By:

Daniel Buettner

Overview

- Graphplan review
- Parsing the input
- Representing the problem
- Finding plans
- Conclusion

Graphplan review

- First described by Avrim Blum and Merrick Furst in their 1995 paper entitled *Fast Planning Through Planning Graph Analysis*. An excerpt of this paper appeared in the Proceedings of IJCAI-95.

- Represented a radical improvement over existing planning systems.

- Not only was graphplan faster, but it guaranteed termination in cases where no solution could be found. It also guaranteed that any plan returned would be the shortest possible parallel plan.

Graphplan review...

- graphplan works by building a planning graph consisting of alternating layers of propositions and actions. At problem initialization, the set of initial conditions forms the initial layer of propositions.

- To extend the planning graph to level i, graphplan determines the complete set of actions whose preconditions exist at proposition level $i-1$ and creates an action layer made up of this set. It also adds "noop" actions for each proposition that exists in level $i-1$. It then creates proposition level i, which is made up of the effects of the applicable actions, where the effect of a noop is to simply keep its precondition true.

Graphplan review...

- graphplan waits to start search until all of the goals are present in the highest proposition level.

- Once it believes the goals are reachable, it starts a sort of breadth first search from the goals back towards the initial conditions. If a solution is found, it is returned.

- When graphplan fails to find a solution, it simply extends the planning graph by a level and searches again. This is how graphplan guarantees the shortest possible plan.

Graphplan review...

- graphplan also watches to see if a proposition level is identical to the one before it. If this occurs, the planning graph is said to have "leveled off" and no solution exists.

- To ensure correctness and to reduce the search space, graphplan must find and enforce mutual exclusions between actions. If the delete list of one action contains a proposition that is in another action's preconditions or add list, these two actions cannot occur at the same action level in the planning graph. Additionally, two actions are mutually exclusive if each has a precondition mutually exclusive of a precondition of the other action.

- Propositions are marked as mutually exclusive if all ways of creating one are mutually exclusive of all ways of creating the other.

Parsing the input

- graphplan was written to parse STRIPS style input.

- Since graphplan's introduction, the planning community has introduced new formalisms for problem description.

- jraffplan parses PDDL 1.2 domain and problem descriptions, where PDDL stands for "Planning Domain Definition Language".

- graphplan's input was very simple and was parsed by a hand-written parser. PDDL is much more complex, and demands a more robust parser.

Parsing...

```
(define (domain blocksworld)
(:predicates (clear ?x)
             (on-table ?x)
             (arm-empty)
             (holding ?x)
             (on ?x ?y))

(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
               (not (arm-empty))))

(:action putdown
  :parameters  (?ob)
  :precondition (holding ?ob)
  :effect (and (clear ?ob) (arm-empty) (on-table ?ob)
               (not (holding ?ob))))

(:action stack
  :parameters  (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob)
               (not (clear ?underob)) (not (holding ?ob))))

(:action unstack
  :parameters  (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
  :effect (and (holding ?ob) (clear ?underob)
               (not (on ?ob ?underob)) (not (clear ?ob))
               (not (arm-empty)))))
```

Parsing...

```
(define (problem tower10)
   (:domain blocksworld)
   (:objects a b c d e f g h i j )
   (:init (on-table a) (on-table b) (on-table c) (on-table d)
          (on-table e) (on-table f) (on-table g) (on-table h)
          (on-table i) (on-table j) (clear a)  (clear b)
          (clear c) (clear d) (clear e) (clear j) (clear f)
          (clear g) (clear h) (clear i) (arm-empty))
   (:goal (and (on a b) (on b c) (on c d) (on d e) (on e f) (on f g)
               (on g h) (on h i) (on i j) )))
```

- These examples contain a very small subset of the PDDL language.

- Many aspects of problem and domain definition are optional, so the parser cannot assume that it will see the same tokens in the same sequence for every definition it encounters.

Parsing...

- As a C programmer, one would use lex and yacc to generate such a parser. However, I used Java.

- A third party tool has been developed for just such an occasion. JavaCC is a parser generator written in Java which takes a language description and creates a parser for it.

- The generated parser does not do much on its own other than reject invalid input.

- JJTree is an add-on which allows the generated parsers to produce syntax trees.

Parsing...

```
SimpleNode Start() : {}
{
  "(" <DEFINE> ( LOOKAHEAD(2) Domain() | LOOKAHEAD(2) Problem() ) ")"
  { return jjtThis; }
}

void Domain() : {}
{
  "(" <DOMAIN> Identifier() ")"  ( "(" ( ExtensionDef() |
         RequireDef() | ConstantsDef() | PredicatesDef() |
         TimelessDef() | StructureDef() ) ")" )*
}

void ExtensionDef() : {}
{
  <EXTENDS> (Identifier())+
}

void RequireDef() : {}
{
  <REQUIREMENTS> (RequireKey())+
}

void RequireKey() : {
  Token t;
}
{
  t=<REQUIREMENTFIELD>
  {
    jjtThis.setName(t.image);
  }
}
```

Representing the problem

- Each node of the syntax tree generated by JJTree is derived from the same base class. By altering the base class, the programmer can control what information is stored by each node.

- It is thus fairly straight forward to construct an object to represent all of the data contained in the syntax tree.

- In fact, it is possible to write a generic `load()` function which can read either a domain definition or a problem definition and create the appropriate object to represent it, without forcing the user to specify which file contains what.

Representing...

- Propositions are represented as linked lists of tokens.

- Action definitions are likewise represented as linked lists of tokens.

- Action instantiation requires placing symbols from the argument list into the appropriate propositions in both the preconditions and the effects of the action. This is handled by simply using a hash table to represent variable bindings.

- Instantiated actions thus have the same form as action definitions, but they have their arguments bound to particular symbols.

Finding plans

- Just as in the original graphplan, jraffplan begins by placing the initial conditions of the problem into proposition layer zero.

- To extend the planning graph, each applicable action is generated.

- These actions are compared against one another to find which have a mutually exclusive relationship.

- Noop actions are created for each fact in the previous proposition level.

- The effects of these actions are generated to form the next proposition level.

Finding plans...

- The graph is extended until all of the goals are present in the highest proposition level.

- A backward search is launched.
  .
  .
  .

- And unfortunately, this is where it breaks. If you ignore mutual exclusions, jraffplan will return "plans" that have simultaneous actions that are clearly mutex. But when using mutual exclusions, it always reports that no valid plan exists.

- But! It seems close to working!

Conclusion

Graphplan was an extraordinarily novel idea at its introduction. Attempting to duplicate the work of Blum and Furst certainly makes one appreciate their contribution much more.

I had never used a parser generator until this project. They are very cool. I am fairly certain that despite having to take time to learn to use the tools, the parser was working much more quickly than it would have had I attempted to build an ad hoc parser.

The fact that jraffplan is not quite working reminds us that even though something is simple in concept, it is not necessarily easy to implement.