Major Functions

Below are the major helper functions used:

Predicate.unify(x , y , theta , tracker) - staticmethod

This function attempts to unify two lists containing variables and/or ground literals. If there are some set(s) of substitutions that will make the two lists equivalent, this function will return them as `theta`. Otherwise, the `theta` will be `None`.

For example, `unify([x1, x2], [c1, c2])` would unify to `[[(x1, c1), (x2, c2)]]` meaning that there is one valid set of substitutions that would make the two lists equivalent and that is by replacing x1 with c1 and x2 with c2.

On the other hand, `unify([c1 , x1], [c2 , c3])` would unify to `None` because no matter what we try to replace x1 with, we simply can never get the c1 to match with the c2.

This algorithm follows the pseudocode in the textbook very closely.

Action.adds(p , tracker)

This method returns a list of variable bindings that would allow an action to add the given predicate. It essentially goes through each predicate the action might add and tries to unify it with the desired predicate. If the unification succeeds, then the list of substitutions that `Predicate.unify()` returns will be returned. If no unification succeeds, then an empty list will be returned, signifying that there is no such list of variable bindings that would allow the action to add the given predicate.

Plan.calculate_threats_to_new_link(newLink)

This method calculates all potential threats to a new link in the plan and adds those threats to the unresolved threats list. In the search algorithm, whenever we create a new causal link, we have to check if any of the previous actions in the plan threaten the new link. Thus, we devote this method to accomplishing this task. The method also performs some optimizations such as checking if an ordering already resolved each potential threat before adding it. It also guards against actions threatening their own causal links.

Plan.bind_variables(subsitution , tracker)

This method bins variables to lower-level variables or ground literals. In the search algorithm, whenever we unify some predicates, we must make the variable substitutions in every action and predicate in the plan. Thus, we devote this method to accomplishing that task. It simply goes through every action, open condition, and causal link and requests that each element makes the appropriate substitutions. There are also substitute methods inside the Action and Predicate classes that this method makes good use of.

Plan.enforce_ordering(begin , end)

This method enforces some ordering constraints. Whenever we have threats to a causal link, we would like to enforce that the threat goes before the causal step, or that the threat goes after the recipient step. Whenever we satisfy an open precondition, we would like to enforce that the action satisfying the precondition comes before the action that depends on that precondition. This method is dedicated to adding these orderings to the list of ordering constraints. It creates ordering tuples and checks that they aren't redundant before adding them.

Plan.is_threat_addressed(threat)

This method determines if the current ordering constraints already address a given threat. This check is used as an optimization to avoid spawning too many children plans from resolving threats that were never threats to begin with.

estimate_cost(plan)

This is the heuristic function that estimates the remaining cost of a given plan. It uses the cost estimate

```
len( plan.steps ) + len( plan.threats) +
      len( plan.open_conditions )
```

but also employs more-sophisticated pruning methods to decrease the search space. Any inconsistent plan immediately receives a cost of `INFINITY`. Furthermore, plans marked as redundant receive a cost of `INFINITY`. Plans with cost `INFINITY` are never considered in the search and are effectively pruned.

is_redundant(plan , ordering)

Given an ordering for the steps in a plan, this function determines if there are any redundancies that would cause the plan to be pruned. The following properties are flagged as redundant:

- Picking up a block, then putting it back down
- Putting down a block, then picking it back up
- Moving a robot twice in a row
- Loading then unloading a block

The algorithm first generates a consistent plan using the topsort function and iterates through the orderings to check for redundancy. In particular, it checks more than for two redundant actions in a row – it skips over irrelevant actions and can detect redundancy between actions spaced far apart. For example, suppose this sequence of actions appeared in a plan:

```
1. load( k0 , l0 , c0 , r0 )
2. move( r1 , l0 , l1 )
3. unload( k0 , l0 , c0 , r0 )
```

This load and unload are clearly redundant, but the move action in between may throw off simpler redundancy checks. The redundancy checker in this project determines that the move action is irrelevant, skips over it, and correctly analyzes that there is the load is immediately undone by an unload and flags the plan as redundant.

Search Function - planSearch(p , tracker)

This is the main search function. It takes an initial plan `p` and returns a final plan that solves the inputted planning problem if one was found, or throws a `plan_not_found` exception. It performs an A* search and uses the `estimate_cost` function with redundancy checks as a heuristic (please refer to the description of `estimate_cost` on the previous page). The search closely follows the algorithm outlined in the McAllester paper in adding child nodes. The algorithm works as follows:

1. If a given plan has inconsistent ordering, then discard it. (The inconsistency check occurs before we add any plans to the queue of plans to search)
2. If a given plan satisfies all the desired postconditions and has no unresolved threats and has consistent ordering, then return that as a successful plan.
3. Otherwise, we should improve upon our current plan:
    a. If the plan has an unresolved threat, `T threatens A -> B` resolve it by enforcing that T is ordered before A or T is ordered after B – that is `T < A` or `B < T`. For each possible resolution, we spawn off a new child plan.
    b. If the plan has no unresolved threats, then pick an open precondition to satisfy. Try satisfying this open precondition by trying to create a causal link with current actions – that is, find all existing actions that add the given precondition and create causal links with them. For each action, we spawn off a new child plan. We can also try satisfying this open precondition by trying all potential subsequent actions. For each potential subsequent action that can add the given precondition, we create the causal link and spawn off a new child plan.
4. Keep searching while there are more child plans. If there are no more child plans, then return failure.

Real World Planning Problem

A planning problem I have been interested in is course scheduling. At a given school, all students have some set of courses they would like to take; however, there are a fixed number of class periods during which classes can take place. To make things even more complicated, only some teachers can teach specific classes and no student or teacher can be in more than one class at a time. We might realize that there are also maximum class capacities, but that would make the problem too complicated and we might ignore that for now. Below is a proposed PDDL abstraction.

Objects

- A set of class periods {p1, p2, p3, …}
- A set of courses {c0, c1, c2, …}
- A set of students {s0, s1, s2, …}
- A set of teachers {t0, t1, t2, …}

Unmodifiable Predicates

- teaches(t , c) : teacher t can teach course c
- wantsToTake(s, c): student s wants to take course c. We specify this to help the search go faster – there's no point in having a student take course c if s/he doesn't want to take it.

Modifiable Predicates

- assigned(c , p) : course c has been assigned to take place during period p
- unassigned(c) : course c has not been assigned to take place during any period
- taking(s, c, p) : student s is taking course c during period p
- notTaking(s , p) : student s has no course during period p
- teaching(t, c, p) : teacher t is teaching course c during period p
- notTeaching(t , p) : teacher t is not teaching any course during period p
- isTaught(c) : course c is being taught by some teacher
- notTaught(c) : course c is not being taught by some teacher

Actions

- Assign(c , p): assign course c to take place during period p
  - precond: unassigned(c)
  - add: assigned(c, p)
  - delete: unassigned(c)
- Unassign(c, p): unassigns course c so it can be scheduled for another period
  - precond: assigned(c, p)
  - add: unassigned(c)
  - delete: assigned(c, p)
- Register(s, c, p): registers student s to take course c during period p
  - precond: wantsToTake(s, c), notTaking(s, p), assigned(c, p)
  - add: taking(s, c, p)
  - delete: notTaking(s, p)
- Unregister(s, c, p): unregisters a student from a course
  - precond: taking(s, c, p)
  - add: notTaking(s, p)
  - delete: taking(s, c, p)
- Teach(t, c, p): directs teacher t to teach course c during period p
  - precond: notTeaching(t, p), notTaught(c), teaches(t, c), assigned(c, p)
  - add: isTaught(c), teaching(t, c, p)
  - delete: notTeaching(t, p), notTaught(c)
- RemoveTeach(t, c, p): unassigns teacher t from teaching a course c during period p
  - precond: isTaught(c), teaching(t, c, p)
  - add: notTeaching(t, p), notTaught(c)
  - delete: isTaught(c), teaching(t, c, p)

Notice that the actions, assign, register, and teach, don't necessarily need to have inverses. If we specify that a teacher is teaching a course, we don't have to "undo" that. This makes sense because there's no point in assigning a teacher to teach a course and then un-assigning that teacher later in the search.

Initial State

In the initial state, we would have a list of courses that every student wants to take, and a list of courses that every teacher can teach.

Goal State

The goal state would be that every student is assigned to the courses s/he wants to take. Since scheduling is difficult, partial solutions should also be acceptable; in that case, the student would simply have to choose a different course.

Simplifications

We've made many simplifications, though. For example, most teachers probably don't want to teach more than x number of courses. Some teachers might only want to teach in the afternoon. Also, courses were never assigned classrooms, and classrooms have maximum capacities. Maybe this is why the university has us all go through the arduous process of registering through the awful WolverineAccess website…

Algorithm

Given the amount of knowledge I know, I would probably solve this problem using a backward search – just like what I did in project 2. However, from the readings, we've learned that there are sophisticated forward search algorithms that perform much better than backward searches on larger problems like this scheduling scenario. A sophisticated forward search algorithm might be more appropriate for this kind of problem.