Contents

1 Domain

Let the set of objects currently known be O. The pickplace domain consists of nine operators:

- Scan
  - Result: `ScannedTable`
  - Side Effects: None
  - Refinement: ScanPrimitive
  - Suggestors Required: None
  - Pre-Conditions: None
- LookAt(X)
  - Result: `LookingAt`(X)
  - Side Effects: None
  - Refinement: LookPrimitive
  - Suggestors Required: None
  - Pre-Conditions:
    C_0: [`ScannedTable`]
- ConfirmPosition(X) (Clearly this should really be the same subtask as LookAt but I couldn't get my entails or something correct to make it so.)

- Result: `LookedAt`(X)
- SideEffects: None
- Refinement: LookPrimitive
- Suggestors Required: None
- Pre-Conditions:

  C_0: [`ScannedTable`]

- PlaceK(X, R)
  - Result: `KIn`(X, R)
  - Side Effects: None
  - Refinement: None
  - Suggestors Required: None
  - Pre-Conditions:

    C_0: [`In`(X, R), `LookedAt`(X)]

- Place(X, R)
  - Result: `In`(X, R)
  - Side Effects: [`Holding`(Nothing), $\neg$`LookedAt`(X), $\forall o \in O$ $\neg$`LookingAt`(o)]
  - Refinement: PlacePrimitive
  - Suggestors Required: SuggestPlaceMotion
  - Pre-Conditions:

    C_0: [`ScannedTable`]
    C_1: C_0 + [`ClearX`(R, $\{X\}$)]
    C_2: C_1 + [`Holding`(X)]
    C_3: C_2 + [`ClearX`(PlaceMotion, $\{X\}$)]
    Note: I have another condition (C_3 + RobotLocNear) but it is not really used because I don't move the base around so it is always true.

- Pick(X):
  - Result: `Holding`(X)
  - Side Effects: None
  - Refinement: PickPrimitive
  - Suggestors Required: SuggestGrasps, SuggestPickMotion
  - Pre-Conditions: These aren't actually continually building. When planning the motion, we need to be looking at the object and not holding anything. After that, however, we do not need to be looking at the object to actually pick it up.

    C_0: [`ScannedTable`]
    C_1: C_0 + [`Holding`(Nothing)]
    C_2: C_1 + [`LookingAt`(X)]
    C_3: C_0 + [`ClearX`(PickupMotion, $\{X\}$)]
    C_4: C_3 + [`Holding`(Nothing)]

- Clear(R, $\{X_1, ..., X_n\}$)

- Result: `ClearX`(R, $\{X_1, ..., X_n\}$)
- Side Effects: [`Holding`(Nothing)]
- Refinements: None
- Suggestors Required: None
- Pre-Conditions: Only allowed when `ClearX`(R, $\{X_1, ...X_n\}$) is not already TRUE (that may no longer be necessary but it's still in there)

  C_0: None
  C_1: [`ClearX`(R, $\{X_1, .., X_n\}\cup$ Occluders), $\forall o \in$ Occluders, $\neg$`Overlaps`(o, R)]

- REMOVE(X, R)
  - Result: $\neg$`Overlaps`(X, R)
  - Side Effects: None
  - Refinement: None
  - Suggestors Required: SUGGESTPARKINGSPACE
  - Pre-Conditions:

    C_0: None
    C_1: [`KIn`(X, Parking)]

- UNGRASP
  - Result: `Holding`(Nothing)
  - Side Effects: $\neg$`LookedAt`(HeldObject), either puts object down where it was or in a parking space
  - Refinements: PLACEPRIMITIVE
  - Suggestors Required: SUGGESTPLACEMOTION, SUGGESTPARKINGSPACE
  - Pre-Conditions: Only allowed when holding something

    C_0: `ScannedTable`
    C_1: C_0 + [`ClearX`(PlaceMotion, {HeldObject}), `Holding`(HeldObject)]

2 Implementation

2.1 Suggestors

For the pickplace domain, I needed four suggestors: SUGGESTGRASPS, SUGGESTPICKMOTION, SUGGESTPLACEMOTION, and SUGGESTPARKINGSPACE. I've written a little blurb about each of them, but the summary is that I use the ROS planners for the motion suggestors and have hard-coded grasping and free space.

2.1.1 Grasp Suggestor

At the moment my grasp suggestor is very simple: it suggests a forward grasp in the middle of the object that I know will work for picking and placing with a cupboard. For a long time, I used the ROS grasp suggestor instead but, since I do not have a re-grasp subtask, this often generated unsuitable grasps when trying to pick and place in a cupboard.

2.1.2 Motion Suggestors

For the motion suggestors I make extensive use of the ROS object manipulator package (`www.ros.org/wiki/object_manipulator`), both in emulation and by calling their functions directly. Specifically, I break picking and placing tasks into four different possible motions: move, approach, retreat, and lift. The "approach" motion is the phase during which the robot is grasping or placing an object. I specify an approach along the vector $0.8\hat{x} - 0.6\hat{z}$ where $+\hat{x}$ is the unit vector pointing away from the robot and $+\hat{z}$ is the unit vector pointing up (this is the orientation of the base_link and torso_lift_link frames). The approach is ideally 20cm and no less than 5cm. Similarly, the retreat(lift) occurs once an object has been placed(grasped) and is along the vector $-0.8\hat{x} + 0.6\hat{z}$ and is ideally 20cm and no less than 5cm. Approaches, retreats, and lifts are all calculated using an interpolated IK planner. I believe this takes into account collisions with anything in the full collision map created by both the laser scanner and the narrow stereo (see Section 2.2). The exceptions are that during a place, I ignore collisions between the grasped object and the table (we are trying to set it down there after all!) and also the grasped object and any narrow stereo "points" that happen to be floating around. This second is because there tend to be a lot of cloud points on top of the table that are part of the table but not actually considered

The move action does the long movements to set up for an approach. This is calculated using OMPL and then filtered. This takes into account collisions with anything in the full collision map.

When suggesting motions, I first try to suggest a path that does not collide with anything. If this fails, I try to suggest a path that does not collide with obstacles I would like not to move according to the current goal (since we do regression, this isn't really very useful, but since I only have tried this with two moveable obstacles it hasn't really made much difference right now). If that fails, I suggest a path that might collide with any object in the space we consider moveable.

When executing a pick or a place, if the object position has not changed, I simply execute the pick or place suggested by the suggestor. This is why you may see the robot picking something up without looking at it first; it already knows where it is and has, in fact, calculated a pickup path for it. One issue with this is that I do not know where the paths start. In general, this has not been much of a problem because I always move the robot arm back to the same place while looking (to clear it out of the robot's view) so paths tend to start either from there or from the current location of the robot arm. This has yet to fail for me, but really I should be calculating a collision free path for the arm from its current position back to the start of the planned motion.

2.1.3 Free Space Suggestor

I have a sampling routine implemented for this, but it usually fails because the robot is sensitive to its current configuration when deciding if an IK solution for a particular point exists. If I ever implement moving the base, that would probably solve most of this problem. For the videos, I simply give it two locations that are out of the way and I know (from extensive experimentation) can be reached by the robot after picking the items out of the cabinet.

The region returned by the suggestor is always 3cm X 3cm X 30cm (the z direction doesn't really matter). I only just realized it uses these dimensions instead of the object's bounding box dimensions, which was a silly oversight. I must have put it in while debugging and never taken it out. Using the object's bounding box should work just as well.

2.2 Geometry

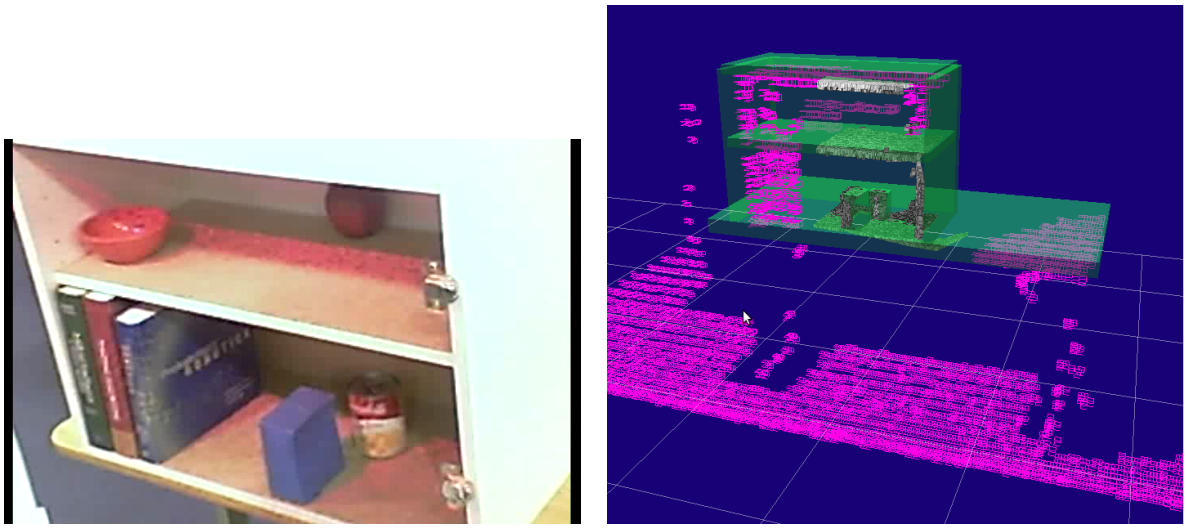
For pickplace, I must be able to calculate overlaps (between regions or between motions and regions) and whether an object is in a region. To simplify this, I consider an object to be "in" a region if it overlaps that region. Therefore, all of the geometry I do is calculating overlaps.

I assume all regions are bounding boxes (either the bounding box of an object or a place where we want to put the bounding box of an object or a pre-specified goal location), although not necessarily axis aligned. Intersecting bounding boxes is then just a small amount of geometry, which I implemented myself.

For intersecting motions with regions, I use the ROS get_trajectory_validity service (`www.ros.org/wiki/planning_environment`). This uses a collision map, which consists of three components:

1. Known objects (both moveable and unmoveable) detected by the narrow stereo camera or added by me.
2. Points from the narrow stereo camera that could not be classified as objects
3. Points from the laser scanner that do not overlap the points from the narrow stereo camera

When doing motion planning, collisions are avoided with any of the three things above. However, only moveable, known objects can be manipulated. The full collision map for the initial configuration of the swap domain is shown in Figure 1.



(a) Actual initial configuration

(b) RViz view

Figure 1: The full collision map for the initial configuration of the swap domain. The "known" objects are shown in green. The narrow stereo is overlaid in grey; any points not covered by the green are considered points to be avoided but not manipulated. The laser scanner collision map minus the narrow stereo is shown in magenta (note that you see a good deal of the floor). The robot should be planning collision free paths that avoid everything shown.

If the region is actually an object that exists in my collision map, I tell the service to ignore collisions with all objects in the collision map (this does not ignore collisions with the laser-scan acquired map,

but since we cannot move anything in that map presumably the motion already avoids it). Otherwise, I temporarily add a fake "object" to the collision map and tell it to ignore all other objects in the collision map. For a while, much of the planning time was being spent calculating overlaps, but this stopped once I started caching the results (and clearing them, of course, when a primitive is executed).

2.3 World Specification

The domain I use has seven objects the robot is aware of: the table, the cabinet walls (right, left, top, shelf), and the two objects it can manipulate. To set up this world, I scan the table, finding the table height and front edge in the process. I also find the rightmost object and set that to be the location of the right wall of the cabinet. I assume the left wall is 10cm right of the left edge of the table (it's always filled with books anyway so it's not that sensitive) and the front edge is aligned with the front edge of the table. I then create a cabinet that is 50cm high and 33cm deep with a shelf at 27cm. For the cabinet we have, this matches the robot's observations. I add this to the world by adding the cabinet walls as boxes 3cm thick.

After adding the cabinet, I look for objects in the world using the narrow stereo. Anything that overlaps the cabinet or table I assume to be part of them. Anything else is considered something the robot might be able to manipulate. To avoid getting weird noise off the table top and the books I also require the objects to have their upper corner between 3cm and 9cm off the table. This also prevents objects on the second shelf from being detected (although the ones shown are also mostly purposely out of where the robot can see).

The map of the objects the robot knows about for the initial swap domain configuration shown in Figure 1(a) is shown in Figure 2.

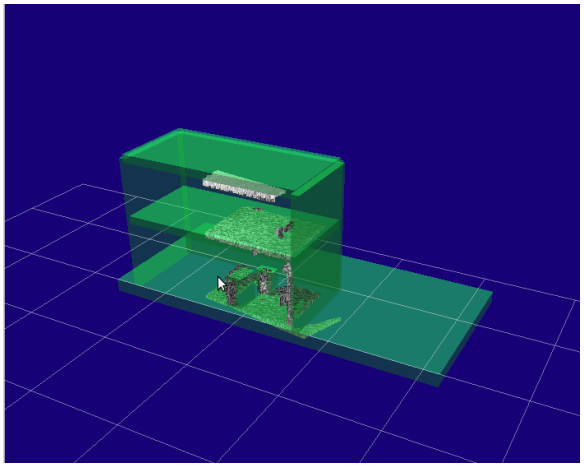


Figure 2: The known objects in the swap initial configuration. The narrow stereo point cloud is overlaid in grey.

3 Plans

The goal of the swap domain is to change the position of the blue box and the soup can in the cabinet. This leads to two possible plans: (1) Put the blue box in the goal location then the soup can or (2) Put

the soup can in the goal location then the blue box. Plan (1) is the simpler plan; the robot moves both items out of the cupboard, then puts first the blue box and then the soup back into the cupboard.

For Plan (2), the robot moves the blue box out of the cupboard and then puts the soup can in its goal position. It then picks up the blue box, intending to put it in the goal location and only then, as it plans the place path, discovers that the soup can is in the way. So it puts down the blue box, moves the soup can out of the cupboard, puts the blue box in the goal, and finally puts the soup can back in the goal.

These plans are shown in Figures 3 and 4. Note that I have costs turned off right now. Also, these plans weren't the ones actually generated running on the robot (those dot files tend to look a little odd, actually, I've been meaning to ask you about that). I have a little simulated world where motions overlap exactly what I know they overlap in real life, which lets me debug the subtasks, and I used that to generate these. But I'm pretty sure this is what the robot is doing too! The order of the primitives is right.

4 Videos and Screenshots

Videos of both swap plans are linked to from `http://projects.csail.mit.edu/pr2/videos/hpn/swap/`. Lots of screenshots of RViz and camera views from the robot's stereo cameras are also linked there.

You can also get all of the source files for this document, all of the screenshots, and all of the source code for the pickplace domain from SVN:

```
svn co svn+ssh://svn.csail.mit.edu/afs/csail.mit.edu/proj/pr2/REPOS/planning_domains/pickplace/
```

If you just want the source files and screen shots:

```
svn co svn+ssh://svn.csail.mit.edu/afs/csail.mit.edu/proj/pr2/REPOS/planning_domains/pickplace/doc/
```

The username and password should be your CSAIL username and password.

5 Frequently Asked Questions

I got a lot of questions over and over while I was filming this so I figured you might want to know the answers too.

Question: Why does the arm move so slowly sometimes?

Answer: It's working in a very confined space and this is what smoothing the RRT trajectory looks like. At least it isn't jerking around horribly anymore.

Question: Is it sensitive to the relative location of the robot and the cupboard?

Answer: Yes, very! Although in principle this should work generally, in practice finding spots for things so that the robot can find non-colliding IK solutions is pretty difficult.

Question: What is taking so long when the robot is sitting there not moving?

Answer: Two things take a long time: planning certain motion paths and doing the object detection. Motion path planning can take a long time because of a bug in the planner where it will return a plan that actually collides with something in the environment. In this case, we may have to replan

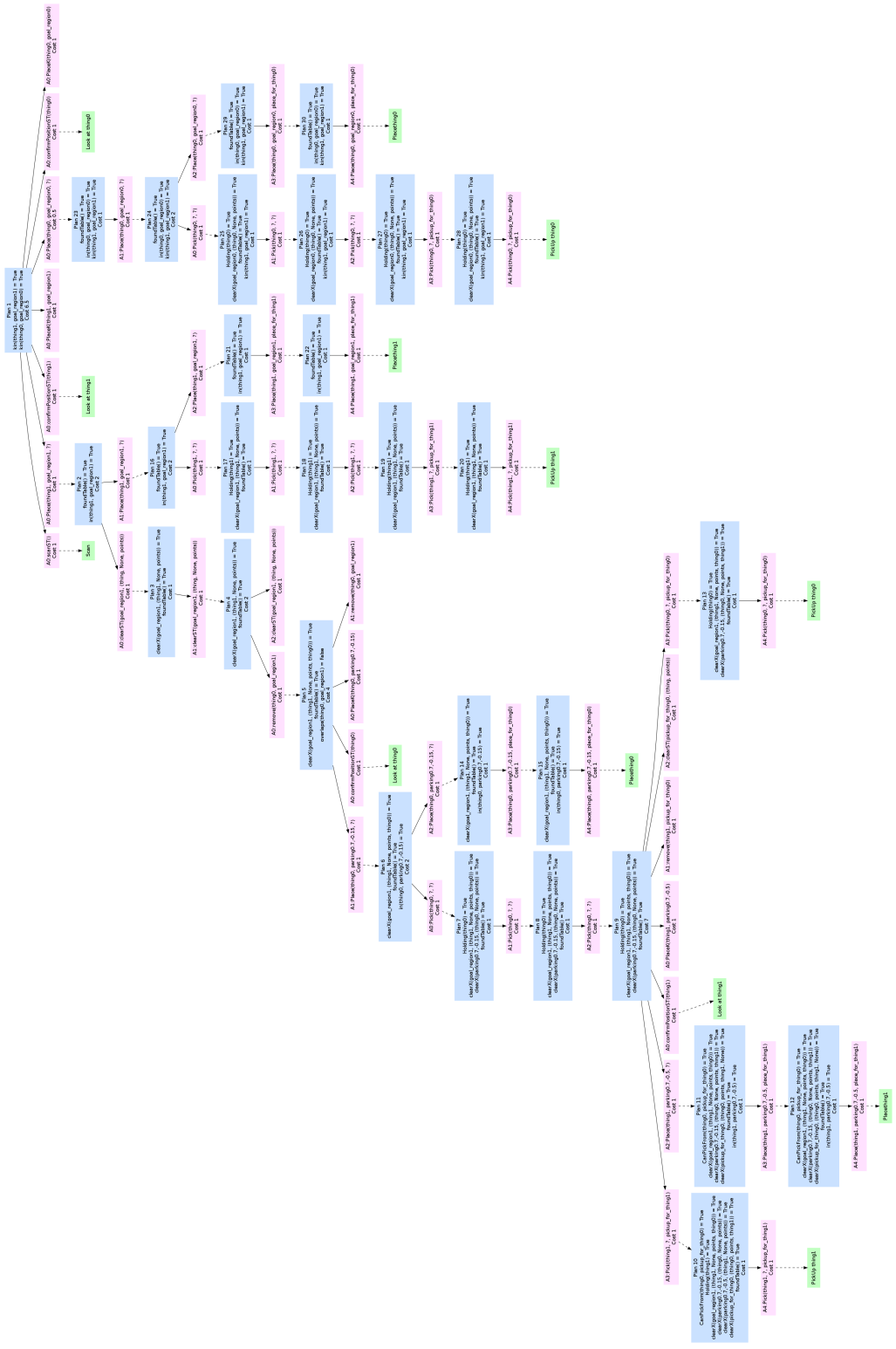


Figure 3: Swap plan for placing the blue box (thing1) then the soup can (thing0).

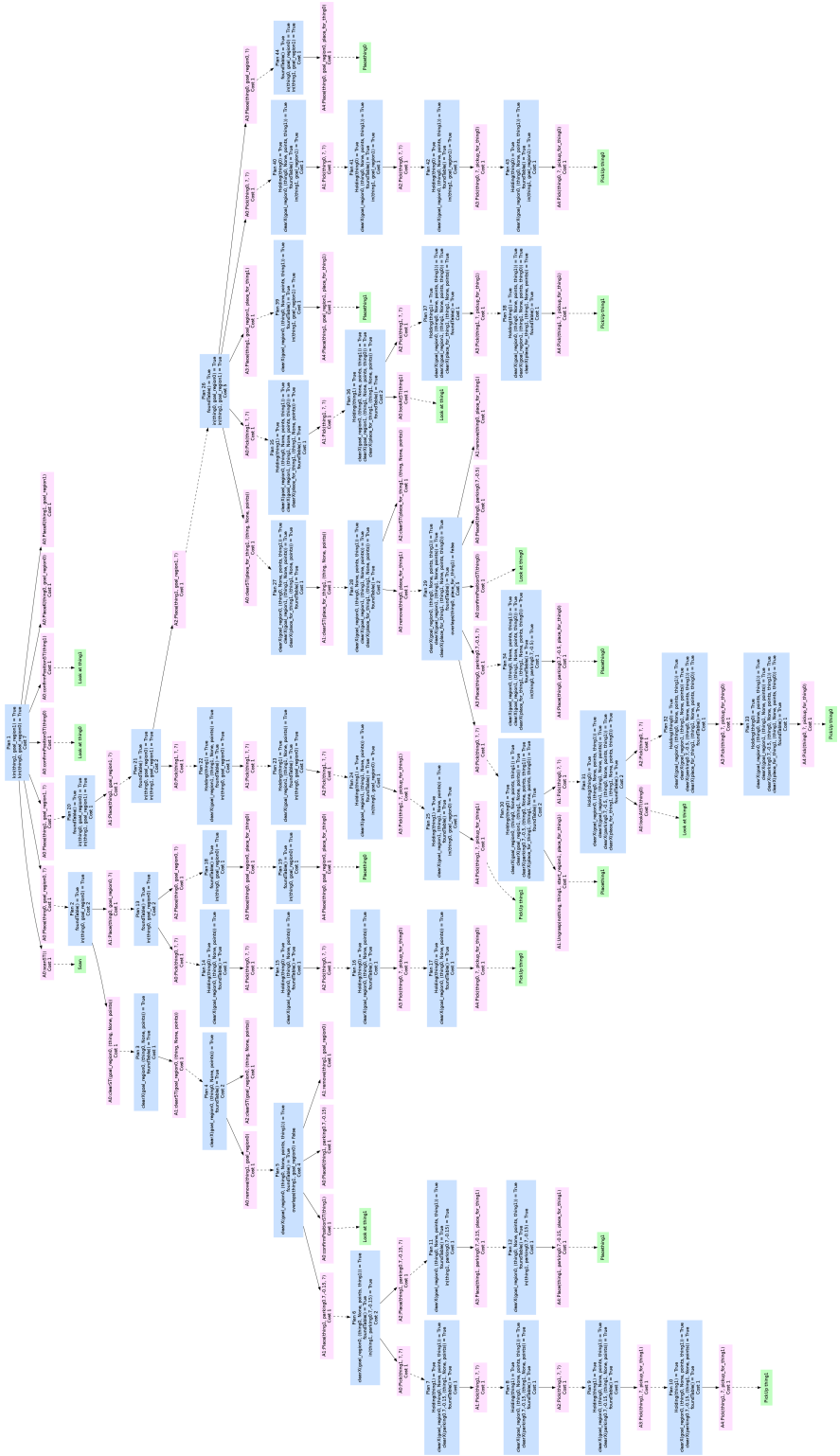


Figure 4: Swap plan for placing the soup can (thing0) then the blue box (thing1).

a number of times before finding a valid plan. Any time the robot sits and thinks for a long time before doing a pickup or place, this is what has happened. This does not happen for every motion. The WG folks say the bug is fixed in Diamondback so perhaps I will try that.

Object detection takes a long time because it is segmenting a point cloud. However, when it really takes a long time is when the robot detects the shelf of the cabinet as the "table". When this happens, it moves its head downwards slightly and tries again. There is about a 45 second clip in the video of plan 2 that is the robot doing this.

I know exactly what took a long time in the videos I have posted (and even if I forget, I said it while taping) so you can ask me if you're curious.

Question: Can I use the robot now?