Goal Oriented Action Planning (GOAP) Guide

Introduction

Task automation requires that the system be able to understand a complex instruction from the user and decompose it into a sequence of basic operations. For example, a command to retrieve a container from the vault can be decomposed into:

1. Query database for container location.
2. Travel to container location.
3. Open cabinet/drawer.
4. Verify container identity.
5. Grasp container.
6. Close cabinet/drawer.
7. Return to vault entrance.

These tasks could be broken down into even more basic steps. Traditionally, this kind of task execution has been organized using a state machine architecture, in which a behavior and state transition is explicitly defined by the system designer for every possible state. However, this approach quickly becomes untenable when dealing with increasingly large sets of states and state variables. It is also time consuming to add new behavior to the system, since the system designer must create explicit transitions between any new states and the existing states.

GOAP is a planning procedure in which the system is defined in terms of actions, unlike a classic state machine which defines a system in terms of states. As in a state machine, the world is modeled as a collection of Boolean state variables. However, the discrete world states are not associated with any particular behavior. Rather, behavior is associated with the actions themselves, which are defined by their state variable preconditions and their effects on the state variables.

Cost functions for the actions can also be defined. When given a task, the system will find the lowest cost action path from the current worldstate to the specified goal worldstate. An “action stack” for the system is built that traverses this path. The system can then pull actions off the stack one at a time, running the function associated with each one. If an action fails or if a state variable changes unexpectedly (for example, we detect a hazard), then the system can stop and call the planner algorithm to generate a new plan which accounts for the new world conditions.

The task_planning Package

The task_planning package provides a C++ GOAP library for use in robot demos. Simply include the necessary headers and link to the built library in your CMakeLists.

Implementation

An action space is built out of three concepts: Entites, Propositions, and Actions.

- **Entities** are objects in the world, such as robots, inventory items, locations, doors, etc. Related Entities are organized into categories called **EntitySets**.
- **Propositions** are true/false statements about the world. A Proposition can take Entities as arguments. The **Worldspace** is the set of all defined Propositions.

Examples:

> *To state that a robot R is at location L, we could say At(R,L). To state that R is NOT at L, we would say ~At(R,L).*
>
> *To state that a radiation anomaly has been detected, RadAnomaly(). If there is no anomaly detected, ~RadAnomaly().*

- **Actions** are constructed from Propositions. An action has preconditions that must be met in order for it be performed, and effects that mutate the Worldspace.

Examples:

> *To define an action to move a robot from one location to another, we could say:*
>
> *MoveRobot(r:robot, origin:location, destination:location)* *// Declare the name of the action and the parameters. A parameter is declared with a label and type.*
>
> *Preconditions: At(r,origin)* *// r must initially be at origin*
>
> *Effects: At(r,destination), ~At(r,origin)* *// After the action, r is at destination and not at origin*

The **Action Space** is the set of all possible actions that the system can perform to mutate the WorldSpace. A **Worldstate** is a particular

permutation of the Worldspace.

ActionPlanner Class

The ActionPlanner class implements an Action Space and performs planning in it. It is located in task_planning/action_planner.h. The steps to use ActionPlanner are:

1. Declare the Entities, Propositions, and Actions in the world.
2. Build the action space from the declared Entities, Propositions, and Actions.
3. Define the initial worldspace.
4. Define the goal worldspace.
5. Call the planner to produce an action path from the initial worldstate to the goal worldstate.

ActionPlanner Usage Example

Expand source

```
// Declare ActionPlanner
  goap::ActionPlanner ap;

  /*
   * Define actions using ADL:
https://en.wikipedia.org/wiki/Action_description_language
   *
   * Action parameters are defined by (name:type).
   * This defines the action MoveRobot which takes three parameters.
   * Preconditions are propositions which must be met in order to perform the action.
   * Effects are propositions that will be set by the action.
   * Propositions can have parameters as well. Using a param named in the base
definition will link the proposition to it.
   *
   * Requirements:
   *     Definitions must use correct syntax.
   *     Overloaded actions are not allowed; the root names must be unique.
   *     An action must have at least one effect.
   *     An action param must have both a name and type.
   *     Proposition params only have a name.
   *     Actions and propositions are allowed to take no parameters, ie MoveRobot().
   *     Negate a proposition by putting ~ in front.
   */

  ap.addAction("Survey(r:robot, a:location)", // Base definition
           {"At(r,a)", "~Is_Surveyed(a)"},    // Precondition list
           {"Is_Surveyed(a)"});               // Effect list

  ap.addAction("MoveRobot(r:robot, destination:location)",
           {"~Docked(r)", "~At(r,destination)"},
           {"At(r,destination)", "~At(r,~destination:location)"}); // negated
parameters refer to all entities of a set EXCEPT the provided one

  // Survey location_3 twice
  ap.addAction("DoubleCheck(r:robot)",
           {"At(r, $location_3:location)", "Is_Surveyed($location_3:location)"},    //
literal parameters refer to a specific entity in a set
           {"Double_Checked_3()"});                                                 //
void parameter take no arguments
```

```
  ap.addAction("Dock(r:robot)",
           {"~Docked(r)"},
           {"Docked(r)", "~At(r,ALL:location)"});    // ALL parameters refer to all
entities of a set

  ap.addAction("Undock(r:robot)",
           {"Docked(r)"},
           {"~Docked(r)"});

  /*
   * Declare the entities in the worldspace
   *
   * addEntitySet( set_name, list_of_members )
   * The sets are linked to the param types named in the action definitions.
   * EntitySets can be declared before the actions or vice versa; order doesn't
matter.
   */
  ap.addEntitySet("robot", {"Pioneer"});

  /*
   * You can specify a range of entities.
   * This example will produce location_1, location_2... location_[num_waypoints]
   */
  ap.addEntitySet("location", num_waypoints);

  /*
   * buildActionSpace() takes the declared actions and entities and compiles them.
   *
   * It creates a version of each actions for each valid permutations of their
parameters.
   */
  int t0 = time(NULL);
  if (!ap.buildActionSpace()) {
    printf("Error: Failed to build action space.");
    return 1;
  }
  int t1 = time(NULL);

  // Print out the created actions and propositions.
  std::cout << ap.description() << std::endl;

  /*
   * Define the initial worldstate.
   * Propositions in a worldspace are initialized to false.
   * Here we set the proposition Docked(Pioneer) = true
   */
  goap::Worldstate start;
  ap.setProposition(&start, "Docked(Pioneer)", true);

  /*
   * Define the goal worldstate.
   *
   * You can also set a whole group of propositions using ADL.
   * Below we set Is_Surveyed(location) for all values of location.
   */
  goap::Worldstate goal;
  ap.setProposition(&goal, "Is_Surveyed(ALL:location)", true);
  ap.setProposition(&goal, "Double_Checked_3()", true);
  ap.setProposition(&goal, "Docked(Pioneer)", true);
```

```
// Print the starting worldstate
std::string desc;
desc = ap.worldstateDescription(&start);
printf("Start: %s\n", desc.c_str());

// Print the goal worldstate
desc = ap.worldstateDescription(&goal);
printf("GOAL: %s", desc.c_str());

std::vector<goap::Worldstate> states;
goap::action_plan_t plan;

// Create the action plan
int t2 = time(NULL);
const int cost = ap.createPlan(start, goal, &plan, &states);
int t3 = time(NULL);

// Print the action plan
printf("ACTION PLAN\nPlan cost = %d\n", cost);
for (int i = 0; i < plan.size(); ++i)
{
  desc = ap.worldstateDescription(&states.at(i));
  printf("%d: %-20s\n", i, plan.at(i).c_str());
}
```

```
  printf("Build time: %u secs\n",  t1 - t0);
  printf("Plan time: %u secs\n",  t3 - t2)
```

ActionPlanner Efficiency

The planning algorithm is based on an A* graph search over the action space. This has a time complexity of O(b^d), where b is the average number of possible transitions per worldspace, and d is the shortest path length.

In short, adding actions and entities to the planner will increase the computation time. Restricting the actions with more preconditions will reduce computation time.

The planning time of the above example is shown below for different numbers of waypoints using a (slow) LANL workstation.

n	T (sec)
50	0.40
100	3.38
150	13.37
200	35.04

TaskManager Class

The TaskManager class located in task_planning/TaskManager.h is used to handle execution of plans produced by ActionPlanner. It runs an operating loop that iterates through the plan. On each step it activates the callback function linked to that action. The callback functions use special return codes to signal the result to TaskManager.

TaskManager Usage Example

Expand source

```
void SurveyDemo::setUpTaskManager()
{
  goap::ActionPlanner planner;

  // Declare actions
  planner.addAction("MoveRobot(a:waypoint)",
            {"~Docked()", "~At(a)"},
            {"At(a)", "~At(~a:waypoint)"});

  planner.addAction("Survey(a:waypoint)",
            {"At(a)", "~IsSurveyed(a)"},
            {"IsSurveyed(a)"});

  planner.addAction("Dock()",
            {"~Docked()"},
            {"Docked()", "~At(ALL:waypoint)"});

  planner.addAction("Undock()",
            {"Docked()"},
            {"~Docked()"});

  // Declare entities
  planner.addEntitySet("waypoint", waypoints_.poses.size());
```

```
  // Build action space
  planner.buildActionSpace();

  std::cout << planner.description() << std::endl;

  // Apply the action planner to the task manager
  task_manager_ = new goap::TaskManager(planner);

  // Define start state
  task_manager_->setStartStateProposition("Docked()", true);

  // Define goal state
  task_manager_->setGoalStateProposition("Docked()", true);
  task_manager_->setGoalStateProposition("IsSurveyed(ALL:waypoint)", true);

  /*
   * Associate the callback functions with the actions.
   * Use std::bind to create a function object that can be passed as an argument to
the TaskManager::setActionCallback function.
   * The TaskManager will call this function when it reaches its action in the stack.
   * Further explanation of what std::bind does:
http://www.cplusplus.com/reference/functional/bind/
   *
   * The Dock() and Undock() actions are straightforward since the actions take no
parameters.
   */
  goap::TaskManager::ActionCallback dock_func =
std::bind(&SurveyDemo::dockActionCallback, this);
  task_manager_->setActionCallback("Dock()", dock_func);

  goap::TaskManager::ActionCallback undock_func =
std::bind(&SurveyDemo::undockActionCallback, this);
  task_manager_->setActionCallback("Undock()", undock_func);

  /*
   * For the MoveRobot and Survey actions, we need to pass the goal pose to the
callback functions.
   * The setActionCallback function can take std::vectors of parameters to use with
the callbacks.
   * The elements of these vectors will be bound to the associated actions inside the
task manager.
   *
   * The size of a parameter vector must equal the size of the EntitySet which it
represents, and be ordered the same.
   * Currently, TaskManager can support callback functions of up to five parameters.
   * The callback function must take as many parameters as the action definition.
   *
   * In this example we provide the poses in a nav_msgs::Path message, which defines
the waypoints of the survey.
   */
  goap::TaskManager::ActionCallback_1<geometry_msgs::PoseStamped> move_robot_func =
std::bind(&SurveyDemo::driveActionCallback, this, std::placeholders::_1);

task_manager_->setActionCallback<geometry_msgs::PoseStamped>("MoveRobot(a:waypoint)",
move_robot_func, waypoints_.poses);

  goap::TaskManager::ActionCallback_1<geometry_msgs::PoseStamped> measure_rad_func =
std::bind(&SurveyDemo::measureRadiationActionCallback, this, std::placeholders::_1);
```

```
  task_manager_->setActionCallback("Survey(a:waypoint)", measure_rad_func,
waypoints_.poses);
}

/*
 * These are the callback functions for our actions.
 * They must return goap::TaskManager::RETURN_TYPE
 *
 * The return type tells the execution loop inside TaskManager what to do next:
 *
 * CONTINUE - Action executed successfully. Modify internal worldstate and continue to
next action.
 * REPEAT - Action did not execute successfully. Try again.
 * TERMINATE - Action did not execute successfully. End the action plan.
 * REPLAN - Update the plan and continue. Use this if an external change occurs to the
worldstate which we must account for.
 * BAD_CALL - Use to flag errors unrelated to the actual execution of the action. A
useful distinction for debugging.
 *
 * BAD_CALL will be thrown if we reach an action that does not have an associated
callback function.
 */
goap::TaskManager::RETURN_TYPE
SurveyDemo::driveActionCallback(geometry_msgs::PoseStamped goal)
{
  ROS_INFO("driveActionCallback");
  return RETURN_TYPE::CONTINUE;
}
goap::TaskManager::RETURN_TYPE
SurveyDemo::measureRadiationActionCallback(geometry_msgs::PoseStamped goal)
{
  ROS_INFO("measureRadiationActionCallback");
  return RETURN_TYPE::CONTINUE;
}
goap::TaskManager::RETURN_TYPE SurveyDemo::dockActionCallback()
{
  ROS_INFO("dockActionCallback");
  return RETURN_TYPE::CONTINUE;
}
goap::TaskManager::RETURN_TYPE SurveyDemo::undockActionCallback()
{
```

```
  ROS_INFO("undockActionCallback");
  return RETURN_TYPE::CONTINUE;
}
```