16.413 Project Description

AUTONOMOUS LUNAR VEHICLE:
As your 16.413 term project you will develop an autonomous system that is capable of executing a lunar logistics mission. The mission is to control an autonomous lunar vehicle to transport cargo between certain sites, and conduct science experiements on the moon surface. The mission requires completing several such activities.

Traveling between sites may require a long drive on the lunar surface, which is full of craters and rocks that your vehicle can not traverse. So, for full autonomy your vehicle will have to plan paths around these obstacles to reach its goal destination.

A list of activities is uploaded from a ground station on earth as a goal expression in the PDDL format. During the day, your vehicle (one of many autonomous vehicles operating on the moon surface) has to autonomously complete all these activities.

Part 1: Building the activity planner

Introduction

Your first step to building the autonomoy architecture for your autonomous system is constructing the activity planner (the second part will consist of building the path planner to plan trajectories around obstacles).

After going through numerious technical papers such as [1–4], the technical board of your company decided to do the activity planning by posing it as a *satisfiability problem for propositional logic* (will be denoted as a SAT problem).

Since the on-board computational equipment supports Java, as the head of the technical team you decide to build your activity planner in Java. Your task is to build a code architecture that reads in a PDDL specification of the problem (e.g., predicates and actions) as well as the problem instance (e.g., objects, initial conditions, and the goal expression), converts this specification to a SAT problem, and solves the SAT problem using an off-the-shelf SAT solver. The technical challenge is to come up with an effective encoding of the planning problem as a SAT problem, and implement this encoding in Java.

After a brief search on the web, you find out that there is a neat PDDL parser for Java called `PDDL4J` and a SAT solver called `SAT4J`. You decide to use these tools to build your activity planner.

Technical background

Solving Satisfiability Problems:

Let $\{p_1, p_2, \ldots, p_n\}$ be a set of atomic propositions, each of which can be either true or false. In general, an expression consists of negations, conjunctions, and disjunctions of these atomic propositions. For instance, for the simple expression, namely the *disjunction* of the first two atomic propositions, i.e., $p_1 \vee p_2$, to be true, either p_1 or p_2 must be true. A *conjunction* of the same variables, i.e., $p_1 \wedge p_2$, is true, whenever both p_1 and p_2 are true. Finally, the expression $\neg p_1$, the *negation* of the first atomic proposition, is true, whenever p_1 is false. An expression is said to be in the Conjunctive Normal Form (CNF), if it is a conjunction of disjunctions where negations are only applied to the atomic propositions. For instance,

$$\left(p_1 \vee p_2 \vee (\neg p_3)\right) \bigwedge \left((\neg p_1) \vee p_3 \vee p_5\right) \bigwedge \left((\neg p2) \vee p_4\right) \bigwedge \left(\neg p_4\right) \bigwedge \left(p_2 \vee (\neg p_5)\right)$$

is in CNF. Each disjunct in the CNF is called a *clause.* For instace, in the example above, there are five clauses, namely,

- $C_1 := p_1 \vee p_2 \vee (\neg p_3)$
- $C_2 := (\neg p_1) \vee p_3 \vee p_5$
- $C_3 := (\neg p2) \vee p_4$
- $C_4 := \neg p_4$
- $C_5 := p_2 \vee (\neg p_5)$

Given a logical expression involving these propositions in the Conjunctive Normal Form (CNF), the *SAT problem* is to determine whether there is an assignment to all the variables that makes this expression true, i.e., makes all of the clauses true. For instance, in the example provided above assigning $p_1 =$ `true`, $p_2 =$ `false`, $p_3 =$ `true`, $p_4 =$ `false`, and $p_5 =$ `false` makes all the clauses true. Propositional logic and the SAT problem will be covered in Lecture 10 in detail.

The SAT problem is fundamental to computer science and the theory of complexity. In fact, the problem is NP-complete when each clause has exactly 3 literals. This problem is one of Karp's NP-complete problems. Although the problem is computationally challenging, most interesting instances of the SAT problem can be solved quite quickly. One of the best solvers (winner of many SAT competitions) is the `miniSAT`, a Java implementation of which is the `SAT4J`.

`SAT4J` takes in a number of clauses (which may contain arbitrary number of literals in them), and checks whether all clauses are satisfiable all at once. If so, the solver outputs also a *model*, i.e., an assignment to each variable which makes every clause true.

Parsing the PDDL files:

The Planning Domain Description Language (PDDL) is almost a standard today for representing planning problems.

A good PDDL parser for Java is the `PDDL4J`, which you will use in your term project. `PDDL4J` takes in two files. The first file describes the planning problem by providing the predicates and the actions. This file, essentially, describes the capabilities of your lunar vehicle by providing the actions that it can execute. The second file describes the problem instance by providing the objects, initial condition, and the goal expression. You can think of this as the file that is uploaded from the earth-based ground station each day. This file will include names of all the sites and the cargo (the objects), the current location of the cargo (initial states), and a list of science experiments that have to be done that day as well as the destination location for each cargo (goal expression).

`PDDL4J` will combine these two files into a single data structure from which you can extract the predicates, actions, objects, initial condition, and the goal expression easily.

For each predicate `PDDL4J` will provide the name of the predicate, the number of variables that this predicate takes as well as the names of these variables. For every action, it will provide you with the parameters (a list of variables), the precondition expression, and the effect expression. The expressions in `PDDL4J` are encoded in a special structure that can be decoded easily using a recursion (see the example Java code). The initial condition will be provided as a list of predicates that are true initially (all other predicates are false initially). Finally, the goal expression will be provided to you in the form of a `PDDL4J` expression, which can be parsed easily using recursion as noted before. The example code provided to you will illustrate all these concepts embedded in `PDDL4J`.

NOTE that you may want to use `PDDL4J` expressions when creating your SAT problem instance since `PDDL4J` expressions provide many tools such as conversion to CNF. Using `PDDL4J`'s `Exp` class (its children `NotExp`, `OrExp`, `AndExp`, `ImplyExp`, `AtomicFormula` classes), you can create an expression yourself, for instance, to model your planning problem instance, and then you can use the `.toConjunctiveNormalForm()` call to convert that expression to CNF. Then, you can feed the CNF to `SAT4J` more easily.

Converting an activity planning problem to a SAT problem:

Given a planning problem described by its actions with their preconditions and effects, objects, the initial condition, and the goal expression, converting the planning problem into a SAT problem has attracted several researchers during the past couple of decades [1–4], motivated by the success that SAT solvers achieved in solving very large instances of the SAT problem (although SAT is a provably hard problem).

There are, in fact, many ways to convert a planning problem (given its actions with preconditions and effects) to a SAT problem. In this project, you are advised to start by looking at the conversion method in the textbook [5], and also to take a look at the seminal papers in the field such as [1].

Notice that all these methods encode the actions, initial condition, and the goal expression naturally in propositional logic and check whether there is a satisfying assignment to all the atomic propositions, which naturally induces a sequence of actions that leads to the satisfaction of the goal expression. In most formulations, there is a fixed time horizon. The problem is first formulated by assuming that it can be solved in one time instance, e.g., by executing one action. If there is no satisfying assignment to the corresponding clauses, then the time horizon is increased by one, and the SAT solver is invoked again. This process is continued until a maximum time horizon is reached (see [5] for details).

Installing PDDL4J

To install PDDL4J go to the following website:

`http://sourceforge.net/projects/pddl4j/`

to download a tar.gz archive file.

Once you extract the `PDDL4J` archive file, you will find the PDDL library in the /lib/pddl4j.jar file. Make sure to reference to this file when you are compiling your software. If you are using Eclipse, make sure that you go to *Project* menu and then select *Preferences*. In the popup dialogbox, click on *Java Build Path* and then click on the *Libraries* tab. Finally, click on *Add External JARs...* and select the pddl4j.jar file.

Importing the classes provided by pddl4j.jar is explained in the example code provided to you (see *Example Java files* section later in this document).

Installing SAT4J

To install SAT4J go to the following website:

http://www.sat4j.org/howto.php

where the installation instructions are provided.

Important note that might save you lots of time in this step: If you are using Eclipse, `SAT4J` is provided as a default plugin with many new distributions. Just go to the folder that Eclipse is installed, and go to the /plugins folder, where you will find org.sat4j.core*.jar and org.sat4j.core*.jar files (* represents the version number). Make sure to include these files in your libraries list as described for PDDL4J installation.

Documentation for PDDL4J and SAT4J

You will find the documentation for both the PDDL4J and SAT4J very helpful in many respects as they include all the class descriptions together with their methods. For instance, you can learn how to use the `Exp`, `NotExp`, `OrExp`, `AndExp`, and `ImplyExp` classes of SAT4J looking at its documentation. The documentation explains that methods using which, e.g., you can create an *and expression* that binds a *not expression* with an *or expression*. Then, you can use the `.toConjunctiveNormalForm()` call, which is also explained in the documentation, to convert the resulting expression into CNF.

The documentation for PDDL4J can be found under the /doc/api/ folder after extracting the archive containing PDDL4j. The documentation for SAT4J can be found online at: http://www.sat4j.org/r17/doc/

Example Java files

You are provided with two example Java applications. The first Java application provides examples for interacting with the PDDL4J library, whereas the second one provides an example for invoking SAT4J libraries. Please review the examples as it might save you some time in starting to use the libraries.

For testing the PDDL4J parser, you are provided with two different domains. The first domain is the canonical blocksworld.pddl (an example problem instance is also provided in the blocksworld_pb1.pddl), which may help you test your pddl parsing code. The second domain is the lunar.pddl (and example problem instance is given in lunar_pb1.pddl), which provides the lunar logistics problem that you will be working on.

Requirements

On the due date of the part 1 of the term project, you should submit your code together with a report explaining your approach to encoding the planning problem as a SAT problem. You should run your activity planner on the example lunar logistics mission provided in the PDDL format. You should test your software with at three different problem instances (i.e., initial conditions and goal expression) that you would consider simple, moderate, and hard to solve.

Your report can be no longer than 5 pages (a 2-3 page report is recommended).

References

[1] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, 1992.

[2] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[3] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Prooceedings if the Thirteenth National Conference on Artificial Intelligence (AAAI)*, 1996.

[4] H. Kautz, B. Selman, and J. Hoffman. SatPlan: Planning as satisfiability. 5th International Planning Competition, 2006.

[5] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, third edition, 2009.