

# 16.410 Principles of Automated Reasoning and Decision Making

## Assignment #5      Due in class and online: Session 11

### Objective

Problems 1, 2, and 3 are intended to give you experience with formulating Linear Programming path planning problems. You will implement a fixed arrival time algorithm for a single robot and for multiple robots. You will also implement a receding horizon path planner. The solver itself is provided, so the emphasis is on formulating the robot dynamics, implementing the path planning algorithms, and comparing the performance of these algorithms to each other and to themselves with differing numbers of time steps. You should select the number of time steps and the time horizon in the receding horizon case so as to clearly demonstrate the performance characteristics of your implementation.

The objective of problem 4 is to give you experience with formulating Constraint Satisfaction Problems (CSPs), and simplifying CSPs using constraint propagation.

This assignment includes both written and electronic parts. Your software should be submitted electronically on MIT Server. All files should be zipped or tarred together before submitting. The written part including all explanations, graphs and other analysis must be handed in in class.

### Background

For this assignment a simple LP solver has been provided that has a programmatic interface. The solver is implemented by the class `Simplex` in the package `mit16_410_13`.

To solve a problem using the solver you must:

1. Create an instance of `Simplex`:

- ```
Simplex myProblem=new Simplex();
```
2. Add an objective function.
  3. Add constraints.
  4. call solveProblem on your Simplex object:  

```
myProblem.solveProblem();
```
  5. Extract the solution data from the Simplex object:  

```
myProblem.getSolutionValue("z");
```

The following public methods are implemented in Simplex to support your efforts in this Problem Set:

```
public void addConstraint (String inequality,
                          double value,
                          LinkedList vars,
                          String slack)
public void addConstraint (String inequality,
                          double value,
                          LinkedList vars)
```

These two methods allow you to add constraints to a problem. The allowable inequalities are " $\leq$ " " $\geq$ " and " $=$ ". The double "value" is the constant on the right hand side of the constraint equation. All right hand sides must be positive or zero. If the right hand side of your equation is negative multiply everything in your equation by -1 and reverse the direction of the inequality. The Linked list vars should be a linked list of Variable objects that encode a variable name and coefficient. For example  $3x_1$  would be constructed as follows:

```
new Variable("x1", 3);
```

The slack parameter allows you to specify the name of the slack variable for that constraint. This is only useful when printing out the full solution with the printFullSolution method (see below).

```
public void addObjectiveFunction (String name,
                                  LinkedList vars)
```

`AddObjectiveFunction` works much like `addConstraint` except that it takes the name of the variable being maximized (such as "Z"). It doesn't take a value or a slack variable.

```
public void describeProblem (PrintStream out)
```

After entering your problem you can print it out in order to verify that the problem was entered correctly. This is primarily useful for debugging. This method takes a `PrintStream` argument to which it prints the output.

```
public long getSolutionTime ()
```

After the problem has been solved you can get the time (in milliseconds) that it took to solve. Make sure that you call `solveProblem` BEFORE calling this method.

```
public int solveProblem () throws Exception
```

The method `solveProblem` solves the problem that has been entered. An exception is raised if the problem was mal-formed. `solveProblem` returns an integer indicating the success of the LP solver. A return value of zero indicates success. A return value of -1 indicates that the objective function was unbounded. A return value of 1 indicates that there is no solution that satisfies the constraints.

```
public void printFullSolution (PrintStream out)
```

`printFullSolution` prints the entire ending tableau out to the `PrintStream`. This is only useful for toy problems as the table becomes unwieldy for anything but small problems.

```
public void printSolution (PrintStream out)
```

`printSolution` prints the values for the objective variable and all of the other variables in the problem.

```
public double getSolutionValue (String varname)
```

`getSolutionValue` returns the value of a problem variable (including the objective variable) after the problem has been solved. This is the programmatic way of getting solution values from the solver. Beware: Variables are case sensitive.

To assist you in understanding interfacing with the solver a demonstration file “ProblemSet4.java” has been supplied that solves the LP problems in problem set 4. As you can see it is not necessary to manually add slack variables – this is done by the algorithm. It is only necessary to ensure that the RHS of the constraints is none negative.

The material on CSPs is presented in the lecture slides and in AIMA, Chapter 5.

## **Problem 1 Robot Path Planner**

Using the provided solver implement a robot path planner that supports the following capabilities:

1. Allows the start state (position and velocity) to be specified for a robotic boat that maneuvers on the 2D water surface.
2. Allows the target state (position and velocity) to be specified for the autonomous boat
3. Implements dynamics for the boat that include drag that causes the robot to lose velocity when power is not supplied.
4. Collect metric information including:
  - a. the time spent running the solver in milliseconds (use the provided `Simplex.getSolutionTime()` method);
  - b. The total number of times the solver was invoked in a given run.
5. Implements a “Fixed Arrival Time” path planning method.

Your implementation should not hard wire a problem into the solver it should dynamically generate the problem when given the number of time steps to use. Your implementation should include a constraint on the maximum fuel flow.

Describe the design of your planner including the dynamics model for the robot.

Use the Robot Path Planner to plan a path with a starting state of  $x=-2$   $y=-3$  and a velocity of ( $x$ -velocity=0.1,  $y$ -velocity=-0.2) to a stationary target state of  $x=1$ ,  $y=1$ . Experiment with differing numbers of time steps and demonstrate the cost of the algorithm in terms of time steps for your problem and plot the results. Also plot the path itself to show the solution path for the highest number of time steps that you demonstrate. What is the shortest time in which your robot can reach its target in this problem? Explain how you arrived at this estimate.

## **Problem 2 Multiple Robots**

Extend the fixed arrival time implementation of problem 1 to support four robots. Position the four robots starting position at four corners of a square region with the target in the center. Each robot should have a different starting velocity (which includes its direction). Solve the path planning problem for the four robots and plot the resulting paths for the four robots. How does the robot perform when compared to path planning for a single robot with the same number of time steps? Your program should print out (or dump to a file) the paths (as pairs of X Y values) for each robot. Plot the four paths to demonstrate the paths taken by the robots. Don't attempt to avoid collisions.

## **Problem 3 Receding Horizon**

Implement a receding horizon path planning algorithm. The implementation should allow the horizon length to be specified. Use the same problem definition as used in "Problem 1" to demonstrate the algorithm. Run the algorithm with a variety of different horizons and graph the computational cost per iteration for different horizons. Compare the "fuel spent" and the "time taken to reach the target" of this receding horizon algorithm and the fixed arrival time algorithm implemented in Problem 1.

## Problem 4 Constraint Satisfaction Problems

### Part A Formulating a Constraint Satisfaction Problem

Your task is to formulate a cross word puzzle as a constraint satisfaction problem. Consider the following puzzle.

|   |   |   |  |   |
|---|---|---|--|---|
| 1 |   | 2 |  | 3 |
|   |   |   |  |   |
|   | 4 |   |  |   |
|   |   | 5 |  |   |

Suppose thus far that you've figure out a candidate list of words with the right length for each numbered row and column, all that remains is to select the right word that matches the constraints of the other columns and rows:

- 1 across: hoses, laser, sheet, snail or steer.
- 4 across: aron, earn, hike, keet, or same.
- 5 across: be, it, no, or us.
- 2 down: aron, earn, hike, keet, or same.
- 3 down: eat, let, run, sun, ten or yes.

In this problem you will provide a formulation by explicitly enumerating the variable domains and the consistent assignments for each constraint.

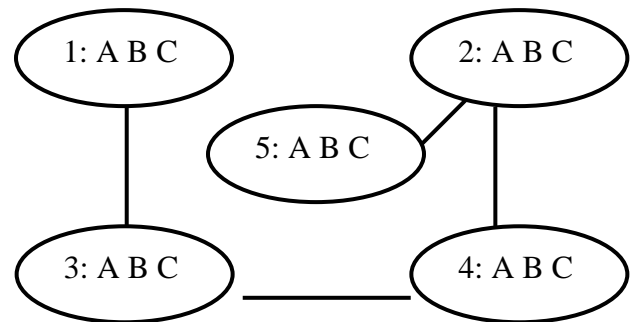
1. Specify variables and domains appropriate for this problem.
2. Specify constraints that adequately model those of the problem. Note that these could be unary or pair wise constraints, or they might involve more than one or two variables. Recall that a constraint specifies all legal assignments for a subset of the variables.
3. Draw a “constraint graph,” where each variable is a node and each constraint arc is a labeled edge between the corresponding nodes.

## Part B: Simplifying a Constraint Satisfaction Problems using Constraint Propagation

**Note – the two parts of this problem can be solved independently.**

Consider the following constraint graph. Each variable has the same domain  $\{A, B, C\}$ . The only valid assignments to pairs of constrained variables are given in the following table.

| Constraint ( $V_i - V_j$ ) | Valid Assignments ( $V_i, V_j$ ) |
|----------------------------|----------------------------------|
| 1-3                        | (A, A) or (B, C)                 |
| 2-4                        | (A, A) or (B, B)                 |
| 3-4                        | (A, B) or (B, A)                 |
| 2-5                        | (B, A) or (B, C)                 |



1. Repeatedly perform constraint propagation on the above constraint graph until you achieve arc consistency. Cross out the eliminated values on each node of the graph.
2. What is the maximum number of possible solutions, based only on knowledge of the remaining values?
3. In general, does constraint propagation guarantee that all infeasible solutions are pruned?

### Problem 5: Time

Please let us know the amount of time it took you to complete this problem set.