COMP9414: Artificial Intelligence
Assignment 1: Constraint Optimization

Due Date: Week 5, Saturday, July 6, 11:59 p.m.

Value: 15%

This assignment concerns developing optimal solutions to a scheduling problem inspired by the scenario of hosting a number of visitors to an organization such as a university department. Each visitor must have a number of meetings, however there are both *constraints* on availability (of rooms and hosts), and *preferences* of each visitor for the days and times of each meeting. Some of the constraints are "hard" constraints (cannot be violated in any solution), while the preferences are "soft" constraints (can be satisfied to more or less degree). Each soft constraint has a cost function giving the "penalty" for scheduling the meeting at a given time (*lower* costs are preferred). The aim is to schedule all the required meetings so that the sum total of all the penalties is *minimized*, and all the constraints are satisfied.

More technically, this assignment is an example of a *constraint optimization problem*, a problem that has constraints like a standard Constraint Satisfaction Problem (CSP), but also *costs* associated with each solution. For this assignment, you will implement a *greedy* algorithm to find optimal solutions to visitor hosting problems that are specified and read in from a file. However, unlike the greedy search algorithm described in the lectures on search, this greedy algorithm has the property that it is guaranteed to find an optimal solution for any problem (if a solution exists).

You must use the AIPython code for constraint satisfaction and search to develop a greedy search method that uses costs to guide the search, as in heuristic search. The search will use a priority queue ordered by the values of the heuristic function that give a cost for each node in the search. The heuristic function for use in this assignment is defined below. The nodes in the search are CSPs, i.e. *each* state is a CSP with variables, domains and the same constraints (and a cost estimate). The transitions in the state space implement domain splitting subject to arc consistency. A goal state is an assignment of values to all variables that satisfies all the constraints.

A CSP for this assignment is a set of variables representing meetings, binary constraints on pairs of meetings, and unary constraints (hard or soft) on meetings. The domains are all working hours in one week, and meetings are all assumed to be 1 hour duration. Days are represented (in the input and output) as strings 'mon', 'tue', 'wed', 'thu' and 'fri', and times are represented as strings '9am', '10am', '11am', '12pm', '1pm', '2pm', '3pm' and '4pm'. The only possible values are a combination of a day and time, e.g. 'mon 9am'. Each meeting name is a string (with no spaces), and each constraint is hard or soft.

The possible constraints are as follows:

```
# binary constraints
constraint, ⟨m1⟩ before ⟨m2⟩
constraint, ⟨m1⟩ same-day ⟨m2⟩
constraint, ⟨m1⟩ one-day-between ⟨m2⟩   # 1 full day between m1 and m2
constraint, ⟨m1⟩ one-hour-between ⟨m2⟩  # 1 hour between end m1 and start m2
```

```
# hard domain constraints
domain, ⟨m⟩, ⟨day⟩, hard
domain, ⟨m⟩, ⟨time⟩, hard
domain, ⟨m⟩, ⟨day⟩ ⟨time⟩-⟨day⟩ ⟨time⟩, hard   # day-time range
domain, ⟨m⟩, morning, hard       # finishes at or before 12pm
domain, ⟨m⟩, afternoon, hard     # starts on or after 12pm

domain, ⟨m⟩, before ⟨day⟩, hard
domain, ⟨m⟩, before ⟨time⟩, hard
domain, ⟨m⟩, before ⟨day⟩ ⟨time⟩, hard
domain, ⟨m⟩, after ⟨day⟩, hard
domain, ⟨m⟩, after ⟨time⟩, hard
domain, ⟨m⟩, after ⟨day⟩ ⟨time⟩, hard

# soft domain constraints
domain, ⟨m⟩, early-week, soft
domain, ⟨m⟩, late-week, soft
domain, ⟨m⟩, early-morning, soft
domain, ⟨m⟩, midday, soft
domain, ⟨m⟩, late-afternoon, soft
```

Each soft constraint has a cost function, defining a "penalty" for only partially satisfying the constraint. For example, a soft constraint that a meeting be *early-week* is satisfied if the meeting is on Tuesday, but with a cost (of 1, as defined below). Costs are always integers. The cost functions are defined as follows:

early-week(d, t): the number of days from mon to d (0 if d = mon)
late-week(d, t): the number of days from d to fri (0 if d = fri)
early-morning(d, t): the number of hours from 9am to t
midday(d, t): the number of hours from 12pm to t
late-afternoon(d, t): the number of hours from t to 4pm

Finally, to define the cost of a solution (that may only partially satisfy the soft constraints), add the costs associated with each soft constraint. Let V be the set of variables and C the set of constraints. Suppose a soft constraint c applies to variable v and let $(day(v), time(v))$ be the value assigned to v in a solution S. For example, c might be *early-morning* which applies to variable m1 (v) and the value assigned to m1 might be mon 10am. Let $cost_c$ be the cost function for the constraint (defined above). Then:

$$cost(S) = \sum_{c \in C} cost_c(day(v), time(v))$$

Heuristic

In this assignment, you will implement greedy search using a priority queue to order nodes based on a heuristic function h. This function must take an arbitrary CSP and return an estimate of the distance from any state S to a solution. So, in contrast to a solution, each variable v is associated with a *set* of possible values (the current domain).

The heuristic estimates the cost of the best possible solution reachable from a given state S by assuming each variable can be assigned the value which minimizes the total cost of the soft constraints applying to that variable. For example, the cost of a meeting with two soft constraints, *early-week* and *early-morning*, is 2 if the meeting is assigned Tuesday (cost 1) at 10am (cost 1).

The heuristic function adds these minimal costs over the set of all variables. Let S be a CSP with variables V and let the domain of v be $dom(v)$. Suppose C_v are the constraints that apply to variable v. Then:

$$h(S) = \sum_{v \in V} \min_{(day(v), time(v)) \in dom(v)} (\sum_{c \in C_v} cost_c(day(v), time(v)))$$

Implementation

Put **all** your code in one Python file called `cspOptimizer.py`. You may (in one or two cases) copy code from AIPython to `cspOptimizer.py` and modify that code, but do not copy large amounts of AIPython code. Instead, write classes in `cspOptimizer.py` that extend the AIPython classes.

Use the Python code for generic search algorithms in `searchGeneric.py`. This code includes a class `Searcher` with a method `search` that implements depth-first search using a list (treated as a stack) to solve any search problem (as defined in `searchProblem.py`). For this assignment, modify the `AStarSearcher` class that extends `Searcher` and makes use of a priority queue to store the frontier of the search. Order the nodes in the priority queue based on the cost of the nodes calculated using the heuristic function.

Use the Python code in `cspProblem.py`, which defines a CSP with variables, domains and constraints. Add costs to CSPs by extending this class to include a cost and a heuristic function h to calculate the cost. Also use the code in `cspConsistency.py`. This code implements the transitions in the state space necessary to solve the CSP. The code includes a class `Search_with_AC_from_CSP` that calls a method for domain splitting. Every time a CSP problem is split, the resulting CSPs are made arc consistent (if possible). Rather than extending this class, write a new class `Search_with_AC_from_Cost_CSP` that has the same methods but implements domain splitting over constraint optimization problems.

You should submit your `cspOptimizer.py` and any other files from AIPython needed to run your program (see below). Your program should read input from a file passed as an argument and print output to standard output.

Sample Input

All input will be a sequence of lines defining a number of meetings, binary constraints and domain constraints, in that order. Comment lines (starting with a '`#`' character) may also appear in the file, and your program should be able to process and discard such lines. All input files can be assumed to be of the correct format – there is no need for any error checking of the input file.

Below is an example of the input form and meaning. Note that you will have to submit at least three input test files with your assignment. These test files should include one or more comments to specify what scenario is being tested.

```
# two meetings with one binary constraint and the same domain constraints
meeting, m1
meeting, m2
# one binary constraint
constraint, m1 before m2
# domain constraints
domain, m1, mon, hard
domain, m2, mon, hard
domain, m1, early-morning, soft
domain, m2, early-morning, soft
```

Sample Output

Print the output to standard output as a series of lines, giving a day and time for each meeting (in the order the meetings were defined). If the problem has no solution, print 'No solution'. When there are multiple optimal solutions, your program should produce one of them. **Important:** For auto-marking, make sure there are no extra spaces at the ends of lines, and no extra line at the end of the output. Set all display options in the AIPython code to 0.

The output corresponding to the above input is as follows:

```
m1:mon 9am
m2:mon 10am
cost:1
```

Submission

- Submit all your files using the following command (this includes relevant AIPython code):

  ```
  give cs9414 ass1 cspOptimizer.py search*.py csp*.py display.py *.txt
  ```

- Your submission should include:
  - Your `.py` source file(s) including any AIPython files needed to run your code
  - A series of `.txt` files (at least three) that you have used as input files to test your system (each including comments to indicate the scenarios tested), and the corresponding `.txt` output files (call these `input1.txt, output1.txt, input2.txt, output2.txt`, etc.); **submit only valid input test files**
- When your files are submitted, a test will be done to ensure that your Python files run on the CSE machine **(take note of any error messages printed out)**
- Check that your submission has been received using the command:

  ```
  9414 classrun -check ass1
  ```

Assessment

Marks for this assignment are allocated as follows:

- Correctness (auto-marked): 10 marks
- Programming style: 5 marks

Late penalty: 3 marks per day or part-day late off the mark obtainable for up to 3 (calendar) days after the due date

Assessment Criteria

- Correctness: Assessed on standard input tests, using calls of the form:

  ```
  python3 cspOptimizer.py input1.txt > output1.txt
  ```

- Programming style: Understandable class and variable names, easy to understand code, good reuse of AIPython code, adequate comments, suitable test files

Plagiarism

Remember that ALL work submitted for this assignment must be your own work and no code sharing or copying is allowed. You may use code from the Internet only with suitable attribution of the source in your program. All submitted assignments will be run through plagiarism detection software to detect similarities. You should **carefully** read the UNSW policy on academic integrity and plagiarism (linked from the course web page), noting, in particular, that *collusion* (working together on an assignment, or sharing parts of assignment solutions) is a form of plagiarism.