Artificial Intelligence
Fuzzy Scheduling

This assignment concerns developing optimal solutions to a scheduling problem inspired by the scenario of a manufacturing plant that has to fulfil multiple customer orders with varying deadlines, but where there may be constraints on tasks and on relationships between tasks. Any number of tasks can be scheduled at the same time, but it is possible that some tasks cannot be finished before their deadline. A task finishing late is acceptable, however incurs a cost, which for this assignment is a simple (dollar) amount per hour that the task is late.

A fuzzy scheduling problem in this scenario is specified by ignoring orders and giving a number of *tasks*, each with a fixed duration in hours. Each task must start and finish on the same day, within working hours (9am to 5pm). In addition, there can be *constraints* both on single tasks and between two tasks. One type of constraint is that a task can have a deadline, which can be "hard" (the deadline must be met in any valid schedule) or "soft" (the task may be finished late – though still at or before 5pm – but with a "cost" per hour for missing the deadline). The aim is to develop an overall schedule for all the tasks (in a single week) that minimizes the total cost of all the tasks that finish late, provided that all the hard constraints on tasks 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 a *cost* associated with each solution. For this assignment, you will implement a *greedy* algorithm to find optimal solutions to fuzzy scheduling 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 tasks, binary constraints on pairs of tasks, and unary constraints (hard or soft) on tasks. The domains are all the working hours in one week, and a task duration is in hours. 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', '4pm' and '5pm'. The only possible values for the start and end times of a task are combinations of such days and times, e.g. 'mon 9am'. Each task name is a string (with no spaces), and the only soft constraints are the soft deadline constraints.

The possible input (tasks and constraints) are as follows:

```
# binary constraints
constraint, ⟨t1⟩ before ⟨t2⟩          # t1 ends when or before t2 starts
constraint, ⟨t1⟩ after ⟨t2⟩           # t1 starts after or when t2 ends
constraint, ⟨t1⟩ same-day ⟨t2⟩        # t1 and t2 are scheduled on the same day
constraint, ⟨t1⟩ starts-at ⟨t2⟩       # t1 starts exactly when t2 ends


# hard domain constraints
domain, ⟨t⟩ ⟨day⟩                      # t starts on given day at any time
domain, ⟨t⟩ ⟨time⟩                     # t starts at given time on any day
domain, ⟨t⟩ starts-before ⟨day⟩ ⟨time⟩          # at or before given time
domain, ⟨t⟩ starts-after ⟨day⟩ ⟨time⟩           # at or after given time
domain, ⟨t⟩ ends-before ⟨day⟩ ⟨time⟩            # at or before given time
domain, ⟨t⟩ ends-after ⟨day⟩ ⟨time⟩             # at or after given time
domain, ⟨t⟩ starts-in ⟨day⟩ ⟨time⟩-⟨day⟩ ⟨time⟩   # day-time range
domain, ⟨t⟩ ends-in ⟨day⟩ ⟨time⟩-⟨day⟩ ⟨time⟩     # day-time range
domain, ⟨t⟩ starts-before ⟨time⟩       # at or before time on any day
domain, ⟨t⟩ ends-before ⟨time⟩         # at or before time on any day
domain, ⟨t⟩ starts-after ⟨time⟩        # at or after time on any day
domain, ⟨t⟩ ends-after ⟨time⟩          # at or after time on any day

# soft deadline constraints
domain, ⟨t⟩ ends-by ⟨day⟩ ⟨time⟩ ⟨cost⟩ # cost per hour of missing deadline

# tasks with name and duration
task, ⟨name⟩ ⟨duration⟩
```

To define the cost of a solution (that may only partially satisfy the soft deadline constraints), add the costs associated with violating the soft constraints over all tasks. Let V be the set of variables (representing tasks) and C be the set of all soft deadline constraints. Suppose such a constraint c with deadline (d_c,t_c) and penalty cost $cost_c$ applies to variable v, and let (d_v,t_v) be the end day and time of v in a solution S. For example, $cost_c$ might be 100 and (d_v,t_v) might be (mon, 5pm) while the deadline (d_c,t_c) is (mon, 3pm); the cost of this variable assignment is 200.

Define the *delay* $\delta((d_1,t_1),(d_2,t_2))$ to be the number of hours that (d_1,t_1) is after (d_2,t_2) if this is positive, and 0 otherwise, where a full day counts as 24 hours. Then, where c_v is the soft deadline constraint applying to variable v:

$$cost(S) = \sum_{cv \in C} cost_{cv} * \delta((d_v,t_v),(d_{cv},t_{cv}))$$

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 that minimizes the cost of the soft deadline constraint applying to that variable. The heuristic function adds these minimal

costs over the set of all variables, similar to calculating the cost of a solution *cost*(*S*) above. Let *S* be a CSP with variables *V* and let the domain of *v*, written *dom*(*v*), be a set of end days and times for *v*. Then, where the summation is over all soft deadline constraints c_v as above:

$$h(S) = \sum_{cv \in C} \min_{(d_v,t_v) \in dom(v)} cost_{c_v} * \delta((d_v,t_v),(d_{c_v},t_{c_v}))$$

Implementation

Put **all** your code in one Python file called `fuzzyScheduler.py`. You may (in one or two cases) copy code from AIPython to `fuzzyScheduler.py` and modify that code, but do not copy large amounts of AIPython code. Instead, in preference, write classes in `fuzzyScheduler.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 `SearchwithAC fromCSP` 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, you may prefer to write a new class `SearchwithAC fromCostCSP` that has the same methods but implements domain splitting over constraint optimization problems.

You should submit your `fuzzyScheduler.py` and any other files from AIPython needed to run your program (see below). The code in `fuzzyScheduler.py` will be run in the same directory as the AIPython files that you submit. 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 tasks, 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 tasks with two binary constraints and
soft deadlines task, t1 3 task, t2 4
# two binary
constraints
constraint, t1 before
t2 constraint, t1
same-day t2 # domain
constraint domain, t2
mon # soft deadlines
domain, t1 ends-by mon
3pm 10 domain, t2
ends-by mon 3pm 10
```

Sample Output

Print the output to standard output as a series of lines, giving the start day and time for each task (in the order the tasks 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:

```
t1:mon 9am
t2:mon 12pm
cost:10
```