COMP9414: Artificial Intelligence
Assignment 1: Week Planner

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

Value: 15%

This assignment is motivated by the problem of scheduling all your personal activities in the context of a busy week involving university studies, work, meals, travel, etc. There are both *constraints* and *preferences* on the days and times of the activities. 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 per hour giving the "penalty" for failing to schedule the activity at the preferred time (we will not consider preferences for days). The aim is to schedule all the activities so that the sum of all the costs is *minimized*, and all the constraints are satisfied.

To be more precise, let us assume activities are to be scheduled on one of the days Sunday to Saturday, starting at one of the times 7am to 7pm. Each activity will be given a fixed duration (in hours) and occur only on one day (note that it is possible for an activity to finish after 7pm). A constraint can refer to the start and end time of an activity or to the day of the activity, or can be a relation between activities (such as 'lecture' must be before 'tutorial'). A preference is that an activity should start *around* a given time (ignoring the day). The full list of constraints and preferences is defined below.

As an example, we might schedule a 'dinner' activity on Monday starting at 7pm for 1 hour – the activity will therefore finish at 8pm. There is **no need** for your code to check whether the activity finishes on the same day as it starts: you can **assume** this.

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 these scheduling problems that are specified in 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 such 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 (heuristic search is the same as A^* search where the path costs are all zero). The search will use a priority queue ordered by the values of the heuristic function that gives a cost for each node in the search. The heuristic function for use in this assignment is defined below. The nodes in this 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 (the AIPython code implements this). A goal state is an assignment of values to all variables that satisfies all the constraints. The cost of a solution is the sum of the costs for the activities in the schedule.

A CSP for this assignment is a set of variables representing activities, binary constraints on pairs of activities, and unary constraints (hard or soft) on activities. The domains are all the combinations of days 'sun', 'mon', 'tue', 'wed', 'thu', 'fri' and 'sat', and times '7am', '8am', '9am', '10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm', '5pm', '6pm' and '7pm'. So the possible values are day time pairs such as 'mon 9am'. Each activity name is a string of letters or numbers (with no spaces).

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

```
# activities with name and duration
activity ⟨name⟩ ⟨duration⟩

# binary constraints
constraint ⟨A1⟩ before ⟨A2⟩      # A1 ends when or before A2 starts
constraint ⟨A1⟩ after ⟨A2⟩       # A1 starts after or when A2 ends
constraint ⟨A1⟩ starts ⟨A2⟩      # A1 and A2 start at the same day and time
constraint ⟨A1⟩ ends ⟨A2⟩        # A1 and A2 end at the same day and time
constraint ⟨A1⟩ overlaps ⟨A2⟩    # A2 starts after A1 starts and not after A1 ends,
                                 # and ends after A1 ends
constraint ⟨A1⟩ during ⟨A2⟩      # A1 starts after A2 starts and ends before A2 ends
constraint ⟨A1⟩ equals ⟨A2⟩      # A1 and A2 start and end at the same day and time
constraint ⟨A1⟩ same-day ⟨A2⟩    # A1 and A2 start and end on the same day

# hard domain constraints
domain ⟨A⟩ on ⟨d⟩                # A starts (and ends) on day d
domain ⟨A⟩ before ⟨d⟩            # A starts (and ends) before day d
domain ⟨A⟩ after ⟨d⟩             # A starts (and ends) after day d
domain ⟨A⟩ starts-before ⟨t⟩     # A starts at or before time t on any day
domain ⟨A⟩ starts-after ⟨t⟩      # A starts at or after time t on any day
domain ⟨A⟩ ends-before ⟨t⟩       # A ends at or before time t on any day
domain ⟨A⟩ ends-after ⟨t⟩        # A ends on or after time t on any day

# soft domain constraints
domain ⟨A⟩ around ⟨t⟩ ⟨cost⟩     # cost per hour of not meeting time preference t
```

To define the cost of a solution (that may only partially satisfy the soft constraints), sum the costs associated with violating the soft constraints over all activities. Let V be the set of variables (representing activities) and C be the set of all soft constraints. Suppose such a constraint c with time preference t_c and cost $cost_c$ applies to variable v, and let (d_v, t_v) be the start day and time of v in a solution S. For example, $cost_c$ might be 10 and (d_v, t_v) might be (mon, 5pm), while the preferred time t_c is 3pm; the cost of this variable assignment is 20 (2 hours difference $\times$ cost 10).

The time difference between t_1 and t_2 (converted to integer hours) is simply the absolute value of $t_1 - t_2$, denoted $|t_1 - t_2|$. Then, where c_v is the soft constraint applying to variable v:

$$cost(S) = \sum_{c_v \in C} cost_{c_v} * |t_v - t_{c_v}|$$

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 that minimizes the cost of the soft constraint applying to that variable. The heuristic function sums these minimal costs over the set of all variables, similar to calculating the cost of a solution $cost(S)$. Let S be a CSP with variables V and let the domain of v, written $dom(v)$, be a set of times for v (ignoring the day assigned to v). Then, where the summation is over all soft constraints c_v as above:

$$h(S) = \sum_{c_v \in C} \min_{t_v \in dom(v)} cost_{c_v} * |t_v - t_{c_v}|$$

Implementation

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

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, extend 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, but making sure the path cost is always 0. Use this code by passing the CSP problem created from the input into a `searchProblem` (sub)class to make a search problem, then passing this search problem into a `Searcher` (sub)class that runs the search when the `search()` method is called on this search problem.

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, you may prefer to write a new class `Search_with_AC_from_Cost_CSP` that has the same methods but works with over constraint optimization problems. This involves just adding costs into the relevant methods, and modifying the constructor to calculate the cost by calculating h whenever a new CSP is created.

You should submit `weekPlanner.py` and all other files from AIPython needed to run your program. The code in `weekPlanner.py` will be run in the same directory as the AIPython files that you submit. Your program should read input from standard input (i.e. **not** hard-coded from `input1.txt`) and print output to standard output (i.e. **not** hard-coded to `output1.txt`).

Sample Input

All input will be a sequence of lines defining the activities, 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 activities on the same day where time preference cannot be met
activity lecture 3
activity tutorial 1
# two binary constraints
constraint lecture before tutorial
constraint lecture same-day tutorial
# domain constraints
domain lecture on mon
domain lecture starts-before 1pm
domain lecture starts-after 1pm
domain tutorial around 3pm 10
```

Sample Output

Print the output using Python's standard `print` function as a series of lines, giving the start day and time for each activity (in the order the activities were defined) and the cost of the optimal solution. If the problem has no solution, print 'No solution' (with capital 'N'). When there are multiple optimal solutions, your program should produce any one of them. **Important:** For auto-marking, make sure there are no extra spaces at the ends of lines, and no extra *empty* lines after the cost is printed (i.e. no additional newline characters after the one on the last line of the solution showing the cost). This is the standard behaviour of the Python `print` function. Set all display options in the AIPython code to 0.

The output corresponding to the above input is as follows:

```
lecture:mon 1pm
tutorial:mon 4pm
cost:10
```

Submission

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

  ```
  give cs9414 ass1 weekPlanner.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
  - At least three input files used to test your system (including comments to indicate the scenarios tested), and the corresponding output files (call these `input1.txt, output1.txt, input2.txt, output2.txt`, etc.); **submit only correctly formatted input files**
- When your files are submitted, a test will be done to ensure that your Python files run on the CSE machine **in the 9414 environment**; take note of any error messages
- 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: Your mark is reduced by 0.75 marks per day or part-day late for up to 5 calendar days after the due date, after which a mark of 0 is given.

Assessment Criteria

- Correctness: Assessed on *valid* input tests as follows, where input is read from a file redirected to standard input, and output is redirected to standard output (**not** hard-coded file names):

  ```
  python3 weekPlanner.py < input1.txt > output1.txt
  ```

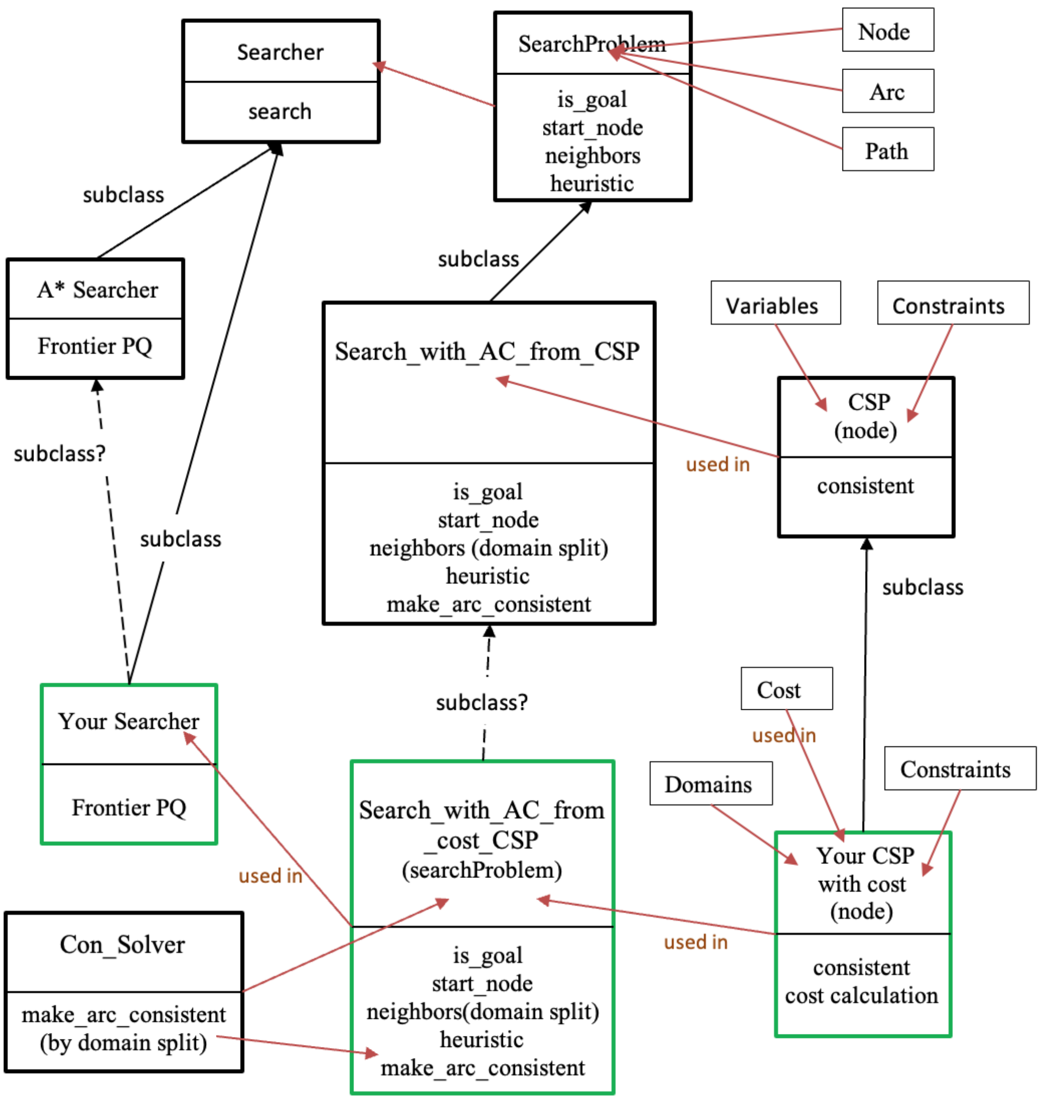
- Programming style: Understandable class and variable names, easy to understand code, good reuse of AIPython code, adequate comments, suitable test files

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. **Do not use public code repositories on sites such as github – make sure your code repository, if you use one, is private.** All submitted assignments will be run through plagiarism detection software to detect similarities to other submissions, including from past years. Do not share your code with anyone both during **and after** the course has finished. 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.

DO NOT USE ANY CODE FROM CONTRACT CHEATING "ACADEMIES" OR "TUTORING" SERVICES. THIS IS SERIOUS MISCONDUCT WITH A HEAVY PENALTY UP TO AUTOMATIC FAILURE OF THE COURSE WITH 0 MARKS.

Appendix: AIPython Classes


Searcher
search
SearchProblem
is_goal
start_node
neighbors
heuristic
Node
Arc
Path
subclass
A* Searcher
Frontier PQ
subclass
Search_with_AC_from_CSP
is_goal
start_node
neighbors (domain split)
heuristic
make_arc_consistent
Variables
Constraints
CSP
(node)
consistent
used in
subclass?
subclass
subclass
Your Searcher
Frontier PQ
subclass?
Cost
used in
Domains
Constraints
Search_with_AC_from
_cost_CSP
(searchProblem)
is_goal
start_node
neighbors(domain split)
heuristic
make_arc_consistent
Your CSP
with cost
(node)
consistent
cost calculation
used in
used in
Con_Solver
make_arc_consistent
(by domain split)