COMP9414: Artificial Intelligence
Assignment 1: Temporal Planner

Due Date: Week 6, Friday, July 9, 11:59 p.m.

Value: 15%

This assignment concerns developing optimal solutions to planning problems for complex tasks inspired by the scenario of building a house, which requires a range of basic tasks to be performed, sometimes in sequence, sometimes in parallel, but where there are constraints between the tasks. We can assume that any number of basic tasks can be scheduled at the same time, provided all the constraints between all the tasks are satisfied. The objective is to develop a plan to finish each of the basic tasks as early as possible.

For simplicity, let us represent time as days using integers starting at 0, i.e. days are numbered 0, 1, 2, etc., up to 99. A temporal planning problem is specified as a series of basic *tasks*. Each task has a fixed duration in days. In addition, there can be *constraints* both on single tasks and between two tasks, for example, a task might have to start after day 20, or one task might have to start after another task finishes (the complete list of possible constraints is given below). A solution to a planning problem is an assignment of a start day to each of the tasks so that all the constraints are satisfied. The objective of the planner is to develop a solution for any given planning problem where each task finishes soonest, i.e. the solution such that the sum of the end days over all the tasks is the lowest amongst all the possible plans that satisfy the constraints.

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 temporal planning 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 temporal planning 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 end days of the basic tasks in the plan.

A CSP for this assignment is a set of variables representing tasks, binary constraints on pairs of tasks, and unary domain constraints on tasks. The domains for the start days of the tasks the integers from 0 to 99, and a task duration is in days > 0. The only possible values for the start and end days of a task are integers, however note that it *is* possible for a task to finish after day 99. Each task name is a string (with no spaces).

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

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

```
# binary constraints
constraint ⟨t1⟩ before ⟨t2⟩     # t1 ends before t2 starts
constraint ⟨t1⟩ after ⟨t2⟩      # t1 starts after t2 ends
constraint ⟨t1⟩ starts ⟨t2⟩     # t1 and t2 start on the same day
constraint ⟨t1⟩ ends ⟨t2⟩       # t1 and t2 end on the same day
constraint ⟨t1⟩ meets ⟨t2⟩      # t2 starts the next day after t1 ends
constraint ⟨t1⟩ overlaps ⟨t2⟩   # t2 starts after t1 starts and not after t1 ends,
                                # and ends after t1 ends
constraint ⟨t1⟩ during ⟨t2⟩     # t1 starts after t2 starts and ends before t2 ends
constraint ⟨t1⟩ equals ⟨t2⟩     # t1 and t2 must be over the same interval

# domain constraints
domain ⟨t⟩ starts-before ⟨d⟩       # t starts on or before d
domain ⟨t⟩ starts-after ⟨d⟩        # t starts on or after d
domain ⟨t⟩ ends-before ⟨d⟩         # t ends on or before d
domain ⟨t⟩ ends-after ⟨d⟩          # t ends on or after d
domain ⟨t⟩ starts-in ⟨d1⟩ ⟨d2⟩     # t starts in the range [d1,d2]
domain ⟨t⟩ ends-in ⟨d1⟩ ⟨d2⟩       # t ends in the range [d1,d2]
domain ⟨t⟩ between ⟨d1⟩ ⟨d2⟩       # t starts and finishes in the range [d1,d2]
```

To define the cost of a solution, simply sum over the end days of all the tasks in the plan. The end day of a task is the start day of the task assigned in the solution plus the given duration of the task, minus 1. More formally, let V be the set of variables (representing tasks) in the CSP and let s_v be the start day of a task v, which has duration d_v, in a solution S. Then:

$$cost(S) = \sum_{v \in V} (s_v + d_v - 1)$$

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), which here we take as the possible start days of the task.

The heuristic estimates the cost of the best possible solution reachable from a given state S by assuming each variable can be assigned a value that minimizes the end day of the task. 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 start days for v. Then, where the sum is over all variables $v \in V$ representing a task with duration d_v as above:

$$h(S) = \sum_{v \in V} \min_{s_v \in dom(v)} (s_v + d_v - 1)$$

Implementation

Put **all** your code in one Python file called `temporalPlanner.py`. You may (in one or two cases) copy code from AIPython to `temporalPlanner.py` and modify that code, but do not copy large amounts of AIPython code to your file. Instead, in preference, write classes in `temporalPlanner.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 your `temporalPlanner.py` and any other files from AIPython needed to run your program (see below). The code in `temporalPlanner.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** (i.e. **not** hard-coded as `input.txt`) and print output to standard output (i.e. **not** hard-coded to a file called `output.txt`).

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.

```
# four unconstrained tasks that are all before a final task
task wall1 10
task wall2 15
task wall3 12
task wall4 10
task roof 20
# binary constraints
constraint wall1 before roof
constraint wall2 before roof
constraint wall3 before roof
constraint wall4 before roof
# domain constraints
domain wall1 starts-after 5
domain roof starts-after 10
```

Sample Output

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

```
wall1:5
wall2:0
wall3:0
wall4:0
roof:15
cost:82
```

Submission

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

  ```
  give cs9414 ass1 temporalPlanner.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: The maximum mark you can obtain is reduced by 3 marks per day or part-day late for up to 3 calendar days after the due date. Any submission more than 3 days late receives 0 marks.

Assessment Criteria

- Correctness: Assessed on *valid* input tests as follows (where the input file can have any name, not just `input1.txt`, so read the file name from `sys.argv[1]`):

  ```
  python3 temporalPlanner.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. 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 ONLINE "TUTORING" SERVICES. THIS COUNTS AS SERIOUS MISCONDUCT WITH A HEAVY PENALTY UP TO AUTOMATIC FAILURE OF THE COURSE WITH 0 MARKS.

Appendix: AIPython Classes