# COMP9414 Artificial Intelligence

## Assignment 1: Constraint Satisfaction Search

@Authors: **Wayne Wobcke, Alfred Krzywicki, Stefano Mezza**

**Due Date:** Week 5, Friday, October 17, 5.00pm

### Objective

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 simplified by ignoring customer orders and having just one machine and 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* (or *constrained 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, we will use a *greedy* algorithm to find optimal solutions to fuzzy scheduling problems that are specified as text strings. 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).

The assignment will use the AIPython code of Poole & Mackworth. You are given code to translate fuzzy scheduling problems specified as text strings into CSPs with a cost, and you are given code for several constraint solving algorithms – based on domain splitting and arc consistency, and based on depth-first search. The assignment will be to implement some missing procedures and to analyse the performance of the constraint solving methods, both analytically and experimentally.

### Submission Instructions

- This is an individual assignment.
- Write your answers in **this** notebook and submit **this** notebook on Moodle under **Assignment 1**.
- Name your submission `<zid>-<firstname>-<lastname>.ipynb` where `<firstname>-<lastname>` is your **real** (not Moodle) name.
- Make sure you set up AIPython (as done below) so the code can be run on either CSE machines or a marker's own machine.
- Do not submit any AIPython code. Hence do not change any AIPython code to make your code run.
- Make sure your notebook runs cleanly (restart the kernel, clear all outputs and run each cell to check).
- After checking that your notebook runs cleanly, run all cells and submit the notebook **with** the outputs included (do not submit the empty version).
- Make sure images (for plots/graphs) are **included** in the notebook you submit (sometimes images are saved on your machine but are not in the notebook).
- Do not modify the existing code in this notebook except to answer the questions. Marks will be given as and where indicated.
- If you want to submit additional code (e.g. for generating plots), add that at the end of the notebook.
- **Important: Do not distribute any of this code on the Internet. This includes ChatGPT. Do not put this assignment into any LLM.**

## Late Penalties

Standard UNSW late penalties apply (5% of the value of the assignment per day or part day late).

**Note:** Unlike the CSE systems, there is no grace period on Moodle. The due date and time is 5pm **precisely** on Friday October 17.

**Important: You can submit as many times as you want before the due date, but if you do submit before the due date, you cannot submit on Moodle after the due date. If you do not submit before the due date, you can submit on Moodle after the due date.**

## Plagiarism

Remember that ALL work submitted for this assignment must be your own work and no sharing or copying of code or answers is allowed. You may discuss the assignment with other students but must not collaborate on developing answers to the questions. You

may use code from the Internet only with suitable attribution of the source. You may not use ChatGPT or any similar software to generate any part of your explanations, evaluations or code. Do not use public code repositories on sites such as github or file sharing sites such as Google Drive to save any part of your work – make sure your code repository or cloud storage is private and do not share any links. This also applies after you have finished the course, as we do not want next year's students accessing your solution, and plagiarism penalties can still apply after the course has finished.

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.

Finally, do not use any contract cheating "academies" or online "tutoring" services. This counts as serious misconduct with heavy penalties up to automatic failure of the course with 0 marks, and expulsion from the university for repeat offenders.

## Fuzzy Scheduling

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 a day 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.

There are three types of constraint:

- **Binary Constraints:** These specify a hard requirement for the relationship between two tasks.
- **Hard Domain Constraints:** These specify hard requirements for the tasks themselves.
- **Soft Deadline Constraints:** These constraints specify that a task may finish late, but with a given cost.

Each soft constraint has a function defining the *cost* associated with violating the preference, that the constraint solver must minimize, while respecting all the hard constraints. The *cost* of a solution is simply the sum of the costs for the soft constraints that the solution violates (and is always a non-negative integer).

This is the list of possible constraints for a fuzzy scheduling problem (comments below are for explanation and do **not** appear in the input specification; however, the code we supply *should* work with comments that take up a full line):

```
# 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⟩, hard                                    # t
starts on given day at any time
domain, ⟨t⟩, ⟨time⟩, hard                                   # t
starts at given time on any day
domain, ⟨t⟩, starts-before ⟨day⟩ ⟨time⟩, hard               # t
starts at or before day, time
domain, ⟨t⟩, starts-after ⟨day⟩ ⟨time⟩, hard                # t
starts at or after day, time
domain, ⟨t⟩, ends-before ⟨day⟩ ⟨time⟩, hard                 # t
ends at or before day, time
domain, ⟨t⟩, ends-after ⟨day⟩ ⟨time⟩, hard                  # t
starts at or after day, time
domain, ⟨t⟩, starts-in ⟨day1⟩ ⟨time1⟩-⟨day2⟩ ⟨time2⟩, hard  # day-
time range for start time; includes day1, time1 and day2, time2
domain, ⟨t⟩, ends-in ⟨day1⟩ ⟨time1⟩-⟨day2⟩ ⟨time2⟩, hard    # day-
time range for end time; includes day1, time1 and day2, time2
domain, ⟨t⟩, starts-before ⟨time⟩, hard                     # t
starts at or before time on any day
domain, ⟨t⟩, ends-before ⟨time⟩, hard                       # t
ends at or before time on any day
domain, ⟨t⟩, starts-after ⟨time⟩, hard                      # t
starts at or after time on any day
domain, ⟨t⟩, ends-after ⟨time⟩, hard                        # t
ends at or after time on any day

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

The input specification will consist of several “blocks”, listing the tasks, binary constraints, hard unary constraints and soft deadline constraints for the given problem. A “declaration” of each task will be included before it is used in a constraint. A sample input specification is as follows. Comments are for explanation and do **not** have to be included in the input.

```
# 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 deadline constraints
domain, t1 ends-by mon 3pm 10
domain, t2 ends-by mon 3pm 10
```

# Preparation

## 1. Set up AIPython

You will need AIPython for this assignment. To find the aipython files, the aipython directory has to be added to the Python path.

Do this temporarily, as done here, so we can find AIPython and run your code (you will not submit any AIPython code).

You can add either the full path (using `os.path.abspath`), or as in the code below, the relative path.

In [ ]:
```
import sys
sys.path.append('aipython') # change to your directory
sys.path # check that aipython is now on the path
```

## 2. Representation of Day Times

Input and output are day time strings such as 'mon 10am' or a range of day time strings such as 'mon 10am-mon 4pm'.

The CSP will represent these as integer hour numbers in the week, ranging from 0 to 39.

The following code handles the conversion between day time strings and hour numbers.

In [ ]:
```
# -*- coding: utf-8 -*-

""" day_time string format is a day plus time, e.g. Mon 10am, Tue 4pm, or just T
    if only day or time, returns day number or hour number only
    day_time strings are converted to and from integer hours in the week from 0
"""
class Day_Time():
    num_hours_in_day = 8
    num_days_in_week = 5

    def __init__(self):
        self.day_names = ['mon','tue','wed','thu','fri']
        self.time_names = ['9am','10am','11am','12pm','1pm','2pm','3pm','4pm']

    def string_to_week_hour_number(self, day_time_str):
        """ convert a single day_time into an integer hour in the week """
        value = None
        value_type = None
        day_time_list = day_time_str.split()
        if len(day_time_list) == 1:
            str1 = day_time_list[0].strip()
            if str1 in self.time_names: # this is a time
                value = self.time_names.index(str1)
                value_type = 'hour_number'
```

```
            else:
                value = self.day_names.index(str1) # this is a day
                value_type = 'day_number'
            # if not day or time, throw an exception
        else:
            value = self.day_names.index(day_time_list[0].strip())*self.num_hour
                + self.time_names.index(day_time_list[1].strip())
            value_type = 'week_hour_number'
        return (value_type, value)

    def string_to_number_set(self, day_time_list_str):
        """ convert a list of day-times or ranges 'Mon 9am, Tue 9am-Tue 4pm' int
            e.g. 'mon 9am-1pm, mon 4pm' -> [0,1,2,3,4,7]
        """
        number_set = set()
        type1 = None
        for str1 in day_time_list_str.lower().split(','):
            if str1.find('-') > 0:
                # day time range
                type1, v1 = self.string_to_week_hour_number(str1.split('-')[0].s
                type2, v2 = self.string_to_week_hour_number(str1.split('-')[1].s
                if type1 != type2: return None # error, types in range spec are
                number_set.update({n for n in range(v1, v2+1)})
            else:
                # single day time
                type2, value2 = self.string_to_week_hour_number(str1)
                if type1 != None and type1 != type2: return None # error: type i
                type1 = type2
                number_set.update({value2})
        return (type1, number_set)

    # convert integer hour in week to day time string
    def week_hour_number_to_day_time(self, week_hour_number):
        hour = self.day_hour_number(week_hour_number)
        day = self.day_number(week_hour_number)
        return self.day_names[day]+' '+self.time_names[hour]

    # convert integer hour in week to integer day and integer time in day
    def hour_day_split(self, week_hour_number):
        return (self.day_hour_number(week_hour_number), self.day_number(week_hou

    # convert integer hour in week to integer day in week
    def day_number(self, week_hour_number):
        return int(week_hour_number / self.num_hours_in_day)

    # convert integer hour in week to integer time in day
    def day_hour_number(self, week_hour_number):
        return week_hour_number % self.num_hours_in_day

    def __repr__(self):
        day_hour_number = self.week_hour_number % self.num_hours_in_day
        day_number = int(self.week_hour_number / self.num_hours_in_day)
        return self.day_names[day_number]+' '+self.time_names[day_hour_number]
```

## 3. Constraint Satisfaction Problems with Costs over Tasks with Durations

Since AI Python does not provide the CSP class with an explicit cost, we implement our own class that extends `CSP`.

We also store the cost functions and the durations of all tasks explicitly in the CSP.

The durations of the tasks are used in the `hold` function to evaluate constraints.

In [ ]:
```
from cspProblem import CSP, Constraint

# We need to override Constraint, because tasks have durations
class Task_Constraint(Constraint):
    """A Task_Constraint consists of
    * scope: a tuple of variables
    * spec: text description of the constraint used in debugging
    * condition: a function that can applied to a tuple of values for the variab
    * durations: durations of all tasks
    * func_key: index to the function used to evaluate the constraint
    """
    def __init__(self, scope, spec, condition, durations, func_key):
        super().__init__(scope, condition, spec)
        self.scope = scope
        self.condition = condition
        self.durations = durations
        self.func_key = func_key

    def holds(self, assignment):
        """returns the value of Constraint con evaluated in assignment.

        precondition: all variables are assigned in assignment

        CSP has only binary constraints
        condition is in the form week_hour_number1, week_hour_number2
        add task durations as appropriate to evaluate condition
        """
        if self.func_key == 'before':
            # t1 ends before t2 starts, so we need add duration to t1 assignment
            ass0 = assignment[self.scope[0]] + self.durations[self.scope[0]]
            ass1 = assignment[self.scope[1]]
        elif self.func_key == 'after':
            # t2 ends before t1 starts so we need add duration to t2 assignment
            ass0 = assignment[self.scope[0]]
            ass1 = assignment[self.scope[1]] + self.durations[self.scope[1]]
        elif self.func_key == 'starts-at':
            # t1 starts exactly when t2 ends, so we need add duration to t2 assi
            ass0 = assignment[self.scope[0]]
            ass1 = assignment[self.scope[1]] + self.durations[self.scope[1]]
        else:
            return self.condition(*tuple(assignment[v] for v in self.scope))
        # condition here comes from get_binary_constraint
        return self.condition(*tuple([ass0, ass1]))

# implement nodes as CSP problems with cost functions
class CSP_with_Cost(CSP):
    """ cost_functions maps a CSP var, here a task name, to a list of functions
    def __init__(self, domains, durations, constraints, cost_functions, soft_day
        self.domains = domains
        self.variables = self.domains.keys()
        super().__init__("title of csp", self.variables, constraints)
        self.durations = durations
```

```
        self.cost_functions = cost_functions
        self.soft_day_time = soft_day_time
        self.soft_costs = soft_costs
        self.cost = self.calculate_cost()

    # specific to fuzzy scheduling CSP problems
    def calculate_cost(self):
        """ this is really a function f = path cost + heuristic to be used by th
        total_cost = 0
        day_time = Day_Time()
        # TODO: write cost function
        for task, domain in self.domains.items():

            # skip the task, if it without deadline
            if task not in self.soft_day_time:
                continue

            # define the deadline,duration and cost rate for each task
            soft_deadline = self.soft_day_time[task]
            duration = self.durations[task]
            cost_rate = self.soft_costs.get(task,0)

            # convert soft_deadline from string to int
            date_type, date_value = day_time.string_to_week_hour_number(soft_dea
            if date_type == "week_hour_number":
               soft_deadline = date_value
            elif date_type == "day_number":
               soft_deadline = date_value * day_time.num_hours_in_day + (day_tim
            elif date_type == "hour_number":
               soft_deadline = (day_time.num_days_in_week - 1) * day_time.num_ho

            min_cost = float('inf') # the minmum cost for task
            for start_time in domain:
                # store task finish time and calculate lateness
                finish_time = start_time + duration
                if finish_time > soft_deadline:
                    lateness = finish_time - soft_deadline
                else:
                    lateness = 0

                # calculate the cost when task delay
                if callable(cost_rate):
                   cost = cost_rate(lateness)
                else:
                   cost = cost_rate * lateness
                if cost < min_cost:
                    min_cost = cost

            total_cost += min_cost

        return total_cost

    def __repr__(self):
        """ string representation of an arc"""
        return "CSP_with_Cost("+str(list(self.domains.keys()))+':'+str(self.cost
```

This formulates a solver for a CSP with cost as a search problem, using domain splitting with arc consistency to define the successors of a node.

In [ ]:

```
from cspConsistency import Con_solver, select, partition_domain
from searchProblem import Arc, Search_problem
from operator import eq, le, ge

# rewrites rather than extends Search_with_AC_from_CSP
class Search_with_AC_from_Cost_CSP(Search_problem):
    """ A search problem with domain splitting and arc consistency """
    def __init__(self, csp):
        self.cons = Con_solver(csp) # copy of the CSP with access to arc consist
        self.domains = self.cons.make_arc_consistent(csp.domains)
        self.constraints = csp.constraints
        self.cost_functions = csp.cost_functions
        self.durations = csp.durations
        self.soft_day_time = csp.soft_day_time
        self.soft_costs = csp.soft_costs
        csp.domains = self.domains # after arc consistency
        self.csp = csp

    def is_goal(self, node):
        """ node is a goal if all domains have exactly 1 element """
        return all(len(node.domains[var]) == 1 for var in node.domains)

    def start_node(self):
        return CSP_with_Cost(self.domains, self.durations, self.constraints,
                             self.cost_functions, self.soft_day_time, self.soft_

    # # Code for Question 6
    # @staticmethod
    # def partition_domain(dom, k):
    #     items = list(dom)
    #     n = len(items)

    #     # make sure k is be a valid number
    #     if k < 1: # at least one partition
    #         k = 1
    #     if k > n: # don't generate empty partition
    #         k = n

    #     domain_list = [] # store partition
    #     remain_partition = k # the remain chunk of partition
    #     i = 0 # store the index of value we take from dom

    #     while remain_partition:
    #         take = (n - i + remain_partition - 1) // remain_partition # calcul
    #         domain_list.append(set(items[i:i + take])) # add chunk to list
    #         # update index and remaining partition
    #         i += take
    #         remain_partition -= 1
    #     return domain_list

    # def neighbors(self, node):
    #     """returns the neighboring nodes of node."""
    #     neighs = []
    #     var = select(x for x in node.domains if len(node.domains[x]) > 1)  # c
    #     if var:
    #         k = 4  # set number of partition
    #         parts = self.partition_domain(node.domains[var], k)
    #         to_do = self.cons.new_to_do(var, None)
    #         # discard empty partition
```

```
    #         non_empty_parts = []
    #         for part in parts:
    #             if len(part) > 0:
    #                 non_empty_parts.append(part)

    #         self.display(2, "Splitting", var, "into", non_empty_parts)
    #         for dom in parts:
    #             newdoms = node.domains | {var: dom} # overwrite domain of var
    #             cons_doms = self.cons.make_arc_consistent(newdoms, to_do)
    #             if all(len(cons_doms[v]) > 0 for v in cons_doms):
    #                 # all domains are non-empty
    #                 # make new CSP_with_Cost node to continue the search
    #                 csp_node = CSP_with_Cost(
    #                     cons_doms, self.durations, self.constraints,
    #                     self.cost_functions, self.soft_day_time, self.soft_cos
    #                 )
    #                 neighs.append(Arc(node, csp_node))
    #             else:
    #                 if len(dom) == 0:
    #                     continue
    #                 else:
    #                     self.display(2, "...", var, "in", dom, "has no solutio
    #     return neighs


    def neighbors(self, node):
        """returns the neighboring nodes of node.
        """
        neighs = []
        var = select(x for x in node.domains if len(node.domains[x]) > 1) # chos
        if var:
            dom1, dom2 = partition_domain(node.domains[var])
            self.display(2, "Splitting", var, "into", dom1, "and", dom2)
            to_do = self.cons.new_to_do(var, None)
            for dom in [dom1,dom2]:
                newdoms = node.domains | {var: dom} # overwrite domain of var wi
                cons_doms = self.cons.make_arc_consistent(newdoms, to_do)
                if all(len(cons_doms[v]) > 0 for v in cons_doms):
                    # all domains are non-empty
                    # make new CSP_with_Cost node to continue the search
                    csp_node = CSP_with_Cost(cons_doms, self.durations, self.con
                             self.cost_functions, self.soft_day_time, self.soft_
                    neighs.append(Arc(node, csp_node))
                else:
                    self.display(2,"...",var,"in",dom,"has no solution")
        return neighs


    def heuristic(self, n):
        return n.cost
```

## 4. Fuzzy Scheduling Constraint Satisfaction Problems

The following code sets up a CSP problem from a given specification.

Hard (unary) domain constraints are applied to reduce the domains of the variables before the constraint solver runs.

In [ ]:

```
# domain specific CSP builder for week schedule
class CSP_builder():
    # list of text lines without comments and empty lines
    _, default_domain = Day_Time().string_to_number_set('mon 9am-fri 4pm') # sho

    # hard unary constraints: domain is a list of values, params is a single val
    # starts-before, ends-before (for starts-before duration should be 0)
    # vals in domain are actual task start/end date/time, so must be val <= what
    def apply_before(self, param_type, params, duration, domain):
        domain_orig = domain.copy()
        param_val = params.pop()
        for val in domain_orig: # val is week_hour_number
            val1 = val + duration
            h, d = Day_Time().hour_day_split(val1)
            if param_type == 'hour_number' and h > param_val:
                if val in domain: domain.remove(val)
            if param_type == 'day_number' and d > param_val:
                if val in domain: domain.remove(val)
            if param_type == 'week_hour_number' and val1 > param_val:
                if val in domain: domain.remove(val)
        return domain

    def apply_after(self, param_type, params, duration, domain):
        domain_orig = domain.copy()
        param_val = params.pop()
        for val in domain_orig: # val is week_hour_number
            val1 = val + duration
            h, d = Day_Time().hour_day_split(val1)
            if param_type == 'hour_number' and h < param_val:
                if val in domain: domain.remove(val)
            if param_type == 'day_number' and d < param_val:
                if val in domain: domain.remove(val)
            if param_type == 'week_hour_number' and val1 < param_val:
                if val in domain: domain.remove(val)
        return domain

    # day time range only
    # includes starts-in, ends-in
    # duration is 0 for starts-in, task duration for ends-in
    def apply_in(self, params, duration, domain):
        domain_orig = domain.copy()
        for val in domain_orig: # val is week_hour_number
            # task must be within range
            if val in domain and val+duration not in params:
                domain.remove(val)
        return domain

    # task must start at day/time
    def apply_at(self, param_type, param,domain):
        domain_orig = domain.copy()
        for val in domain_orig:
            h, d = Day_Time().hour_day_split(val)
            if param_type == 'hour_number' and param != h:
                if val in domain: domain.remove(val)
            if param_type == 'day_number' and param != d:
                if val in domain: domain.remove(val)
            if param_type == 'week_hour_number' and param != val:
                if val in domain: domain.remove(val)
        return domain
```

```
    # soft deadline constraints: return cost to break constraint
    # ends-by implementation: domain_dt is the day, hour from the domain
    # constr_dt is the soft const spec, dur is the duration of task
    # soft_cost is the unit cost of completion delay
    # so if the tasks starts on domain_dt, it ends on domain_dt+dur
    """
    <t> ends-by <day> <time>, both must be specified
    delay = day_hour(T2) - day_hour(T1) + 24*(D2 - D1),
    where day_hour(9am) = 0, day_hour(5pm) = 7
    """
    def ends_by(self, domain_dt, constr_dt_str, dur, soft_cost):
        param_type,params = Day_Time().string_to_number_set(constr_dt_str)
        param_val = params.pop()
        dom_h, dom_d = Day_Time().hour_day_split(domain_dt+dur)
        if param_type == 'week_hour_number':
            con_h, con_d = Day_Time().hour_day_split(param_val)
            return 0 if domain_dt + dur <= param_val else soft_cost*(dom_h - con
        else:
           return None # not good, must be day and time

    def no_cost(self, day ,hour):
        return 0

    # hard binary constraint, the rest are implemented as gt, lt, eq
    def same_day(self, week_hour1, week_hour2):
        h1, d1 = Day_Time().hour_day_split(week_hour1)
        h2, d2 = Day_Time().hour_day_split(week_hour2)
        return d1 == d2

    # domain is a list of values
    def apply_hard_constraint(self, domain, duration, spec):
        tokens = func_key = spec.split(' ')
        if len(tokens) > 1:
           func_key = spec.split(' ')[0].strip()
           param_type, params = Day_Time().string_to_number_set(spec[len(func_ke
           if func_key == 'starts-before':
              # duration is 0 for starts before, since we do not modify the time
              return self.apply_before(param_type, params, 0, domain)
           if func_key == 'ends-before':
               return self.apply_before(param_type, params, duration, domain)
           if func_key == 'starts-after':
              return self.apply_after(param_type,params,0,domain)
           if func_key == 'ends-after':
              return self.apply_after(param_type, params, duration, domain)
           if func_key == 'starts-in':
               return self.apply_in(params, 0, domain)
           if func_key == 'ends-in':
               return self.apply_in(params, duration, domain)
        else:
           # here we have task day or time, it has no func key so we need to par
           param_type,params = Day_Time().string_to_week_hour_number(spec)
           return self.apply_at(param_type, params, domain)

    def get_cost_function(self, spec):
        func_dict = {'ends-by':self.ends_by, 'no-cost':self.no_cost}
        return [func_dict[spec]]

    # spec is the text of a constraint, e.g. 't1 before t2'
    # durations are durations of all tasks
```

```
    def get_binary_constraint(self, spec, durations):
        tokens = spec.strip().split(' ')
        if len(tokens) != 3: return None # error in spec
        # task1 relation task2
        fun_dict = {'before':le, 'after':ge, 'starts-at':eq, 'same-day':self.sam
        return Task_Constraint((tokens[0].strip(), tokens[2].strip()), spec, fun

    def get_CSP_with_Cost(self, input_lines):
        # Note: It would be more elegant to make task a class but AIpython is no
        # CSP_with_Cost inherits from CSP, which takes domains and constraints f
        domains = dict()
        constraints = []
        cost_functions = dict()
        durations = dict() # durations of tasks
        soft_day_time = dict() # day time specs of soft constraints
        soft_costs = dict() # costs of soft constraints

        for input_line in input_lines:
            func_spec = None
            input_line_tokens = input_line.strip().split(',')
            if len(input_line_tokens) != 2:
                return None # must have number of tokens = 2
            line_token1 = input_line_tokens[0].strip()
            line_token2 = input_line_tokens[1].strip()
            if line_token1 == 'task':
                tokens = line_token2.split(' ')
                if len(tokens) != 2:
                    return None # must have number of tokens = 3
                key = tokens[0].strip()
                # check the duration and save it
                duration = int(tokens[1].strip())
                if duration > Day_Time().num_hours_in_day:
                    return None
                durations[key] = duration
                # set zero cost function for this task as default, may add real
                cost_functions[key] = self.get_cost_function('no-cost')
                soft_costs[key] = '0'
                soft_day_time[key] = 'fri 5pm'
                # restrict domain to times that are within allowed range
                # that is start 9-5, start+duration in 9-5
                domains[key] = {x for x in self.default_domain \
                                if Day_Time().day_number(x+duration) \
                                == Day_Time().day_number(x)}
            elif line_token1 == 'domain':
                tokens = line_token2.split(' ')
                if len(tokens) < 2:
                    return None # must have number of tokens >= 2
                key = tokens[0].strip()
                # if soft constraint, it is handled differently from hard constr
                if tokens[1].strip() == 'ends-by':
                    # need to retain day time and cost from the line
                    # must have task, 'end-by', day, time, cost
                    # or task, 'end-by', day, cost
                    # or task, 'end-by', time, cost
                    if len(tokens) != 5:
                        return None
                    # get the rest of the line after 'ends-by'
                    soft_costs[key] = int(tokens[len(tokens)-1].strip()) # last
                    # pass the day time string to avoid passing param_type
                    day_time_str = tokens[2] + ' ' + tokens[3]
```

```
                    soft_day_time[key] = day_time_str
                    cost_functions[key] = self.get_cost_function(tokens[1].strip
                else:
                    # the rest of domain spec, after key, are hard unary domain
                    # func spec has day time, we also need duration
                    dur = durations[key]
                    func_spec = line_token2[len(key):].strip()
                    domains[key] = self.apply_hard_constraint(domains[key], dur,
            elif line_token1 == 'constraint': # all binary constraints
                constraints.append(self.get_binary_constraint(line_token2, durat
            else:
                return None

        return CSP_with_Cost(domains, durations, constraints, cost_functions, so

def create_CSP_from_spec(spec: str):
    input_lines = list()
    spec = spec.split('\n')
    # strip comments
    for input_line in spec:
        input_line = input_line.split('#')
        if len(input_line[0]) > 0:
            input_lines.append(input_line[0])
            print(input_line[0])
    # construct initial CSP problem
    csp = CSP_builder()
    csp_problem = csp.get_CSP_with_Cost(input_lines)
    return csp_problem
```

## 5. Greedy Search Constraint Solver using Domain Splitting and Arc Consistency

Create a GreedySearcher to search over the CSP.

The *cost* function for CSP nodes is used as the heuristic, but is actually a direct estimate of the total path cost function *f* used in A* Search.

In [ ]:
```
from searchGeneric import AStarSearcher

class GreedySearcher(AStarSearcher):
    """ returns a searcher for a problem.
    Paths can be found by repeatedly calling search().
    """
    def add_to_frontier(self, path):
        """ add path to the frontier with the appropriate cost """
        # value = path.cost + self.problem.heuristic(path.end()) -- A* definitio
        value = path.end().cost
        self.frontier.add(path, value)
```

Run the GreedySearcher on the CSP derived from the sample input.

**Note: The solution cost will always be 0 (which is wrong for the sample input) until you write the cost function in the cell above.**

In [ ]:
```
# Sample problem specification

# sample_spec = """
```

```
# # 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_spec = """
task, t1 5
task, t2 1
task, t3 3
task, t4 5
task, t5 4
constraint, t1 same-day t2
constraint, t2 before t3
constraint, t3 same-day t4
constraint, t4 before t5
domain, t1 tue
domain, t1 ends-by mon 12pm 8
domain, t2 ends-by thu 3pm 4
domain, t3 ends-by thu 11am 8
domain, t4 ends-by tue 10am 7
domain, t5 ends-by mon 3pm 3
"""
# sample_spec = generate_problem(3)
# print(sample_spec)
# sample_spec = """
# task, t1 4
# task, t2 1
# task, t3 1
# constraint, t1 before t2
# constraint, t2 before t3
# domain, t1 mon
# domain, t2 mon
# domain, t3 mon
# domain, t1 ends-by mon 12pm 100
# domain, t2 ends-by mon 4pm 1
# domain, t3 ends-by mon 4pm 1
# """
```

In [ ]:

```
# display details (0 turns off)
Con_solver.max_display_level = 0
Search_with_AC_from_Cost_CSP.max_display_level = 2
GreedySearcher.max_display_level = 0

def test_csp_solver(searcher):
    final_path = searcher.search()
    if final_path == None:
        print('No solution')
    else:
        domains = final_path.end().domains
        result_str = ''
        for name, domain in domains.items():
            for n in domain:
                result_str += '\n'+str(name)+': '+Day_Time().week_hour_number_to
```

```
        print(result_str[1:]+'\ncost: '+str(final_path.end().cost))

csp_problem = create_CSP_from_spec(sample_spec)
solver = GreedySearcher(Search_with_AC_from_Cost_CSP(csp_problem))
test_csp_solver(solver)
```

## 6. Depth-First Search Constraint Solver

The Depth-First Constraint Solver in AIPython by default uses a random ordering of the variables in the CSP.

We need to modify this code to make it compatible with the arc consistency solver.

Run the solver by calling `dfs_solve1` (first solution) or `dfs_solve_all` (all solutions).

In [ ]:
```
num_expanded = 0
display = False

# # Code for Question 5
# def dfs_solver(constraints, domains, context, var_order, durations, soft_day_t
#     """ generator for all solutions to csp
#         context is an assignment of values to some of the variables
#         var_order is a list of the variables in csp that are not in context
#     """
#     global num_expanded, display
#     to_eval = {c for c in constraints if c.can_evaluate(context)}
#     if all(c.holds(context) for c in to_eval):
#         if var_order == []:
#             print("Nodes expanded to reach solution:", num_expanded)
#             print("Total soft cost of solution:", total_cost) # print cost of
#             yield context
#         else:
#             rem_cons = [c for c in constraints if c not in to_eval]

#             day_time = Day_Time() # create a object to covert deadline
#             min_cost = float('inf') # best cost for each variable
#             next_var = None # the next varibale will choose (best one)
#             vals_cost_pair = None # list of value and cost pair


#             for var in var_order: # loop each unassign variable

#                 soft_deadline = soft_day_time[var]
#                 cost_rate = soft_costs.get(var,0)
#                 duration = durations.get(var,0)

#                 # convert and calculate soft deadline
#                 if soft_day_time is not None and var in soft_day_time:
#                     date_type, date_value = day_time.string_to_week_hour_numbe
#                     if date_type == "week_hour_number":
#                         soft_deadline = date_value
#                     elif date_type == "day_number":
#                         soft_deadline = date_value * day_time.num_hours_in_da
#                     elif date_type == "hour_number":
#                         soft_deadline = (day_time.num_days_in_week - 1) * day

#                 vals_with_costs = [] # a list to hold start-time and cost pair
#                 min_cost_for_var = float('inf') # tracks the minmum cost of va
```

```
#                 # for each start time to calculate finish time and delay
#                 for start_time in domains[var]:
#                     finish_time = start_time + duration
#                     if finish_time > soft_deadline:
#                         lateness = finish_time - soft_deadline
#                     else:
#                         lateness = 0

#                     # calculate the cost when task delay
#                     if callable(cost_rate):
#                         cost = cost_rate(lateness)
#                         vals_with_costs.append((start_time, cost))
#                     else:
#                         cost = cost_rate * lateness
#                         vals_with_costs.append((start_time, cost))
#                     if cost < min_cost: # update the min cost
#                         min_cost = cost

#                 # sorts pair by cost
#                 pairs = [(cost, start_time) for start_time, cost in vals_with_
#                 pairs.sort()

#                 # pass the global min cost to local variable, remember its sor
#                 if min_cost < min_cost_for_var:
#                     min_cost_for_var = min_cost
#                     next_var = var
#                     vals_cost_pair = vals_with_costs

#             # new var order without current best varialbe
#             order_based_on_cost = []
#             for val in var_order:
#                 if val != next_var:
#                    order_based_on_cost.append(val)

#             for value, cost in vals_cost_pair:
#                 if display:
#                     print("Setting", next_var, "to", value)
#                 num_expanded += 1
#                 soft_cost = total_cost + cost
#                 yield from dfs_solver(rem_cons, domains, context | {next_var:

# def dfs_solve_all(csp, var_order=None):
#     """ depth-first CSP solver to return a list of all solutions to csp """
#     global num_expanded
#     num_expanded = 0
#     if var_order == None:    # use an arbitrary variable order
#         var_order = list(csp.domains)
#     return list(dfs_solver(csp.constraints, csp.domains, {}, var_order, csp.du

# def dfs_solve1(csp, var_order=None):
#     """ depth-first CSP solver """
#     global num_expanded
#     num_expanded = 0
#     if var_order == None:    # use an arbitrary variable order
#         var_order = list(csp.domains)
#     for sol in dfs_solver(csp.constraints, csp.domains, {}, var_order, csp.dur
#         return sol  # return first one

def dfs_solver(constraints, domains, context, var_order):
```

```
    """ generator for all solutions to csp
        context is an assignment of values to some of the variables
        var_order is a list of the variables in csp that are not in context
    """
    global num_expanded, display
    to_eval = {c for c in constraints if c.can_evaluate(context)}
    if all(c.holds(context) for c in to_eval):
        if var_order == []:
            print("Nodes expanded to reach solution:", num_expanded)
            yield context
        else:
            rem_cons = [c for c in constraints if c not in to_eval]
            var = var_order[0]
            for val in domains[var]:
                if display:
                    print("Setting", var, "to", val)
                num_expanded += 1
                yield from dfs_solver(rem_cons, domains, context|{var:val}, var_

def dfs_solve_all(csp, var_order=None):
    """ depth-first CSP solver to return a list of all solutions to csp """
    global num_expanded
    num_expanded = 0
    if var_order == None:    # use an arbitrary variable order
        var_order = list(csp.domains)
    return list(dfs_solver(csp.constraints, csp.domains, {}, var_order))

def dfs_solve1(csp, var_order=None):
    """ depth-first CSP solver """
    global num_expanded
    num_expanded = 0
    if var_order == None:    # use an arbitrary variable order
        var_order = list(csp.domains)
    for sol in dfs_solver(csp.constraints, csp.domains, {}, var_order):
        return sol  # return first one
```

Run the Depth-First Solver on the sample problem.

**Note: Again there are no costs calculated.**

In [ ]:
```
def test_dfs_solver(csp_problem):
    solution = dfs_solve1(csp_problem)
    if solution == None:
        print('No solution')
    else:
        result_str = ''
        for name in solution.keys():
            result_str += '\n'+str(name)+': '+Day_Time().week_hour_number_to_day
        print(result_str[1:])

# call the Depth-First Search solver
csp_problem = create_CSP_from_spec(sample_spec)
test_dfs_solver(csp_problem) # set display to True to see nodes expanded
```

## 7. Depth-First Search Constraint Solver using Forward Checking with MRV Heuristic

The Depth-First Constraint Solver in AIPython by default uses a random ordering of the variables in the CSP.

We redefine the `dfs_solver` methods to implement the MRV (Minimum Remaining Values) heuristic using forward checking.

Because the AIPython code is designed to manipulate domain sets, we also need to redefine `can_evaluate` to handle partial assignments.

In [ ]:
```
num_expanded = 0
display = False

def can_evaluate(c, assignment):
    """ assignment is a variable:value dictionary
        returns True if the constraint can be evaluated given assignment
    """
    return assignment != {} and all(v in assignment.keys() and type(assignment[v

def mrv_dfs_solver(constraints, domains, context, var_order):
    """ generator for all solutions to csp.
        context is an assignment of values to some of the variables.
        var_order is a list of the variables in csp that are not in context.
    """
    global num_expanded, display
    if display:
        print("Context", context)
    to_eval = {c for c in constraints if can_evaluate(c, context)}
    if all(c.holds(context) for c in to_eval):
        if var_order == []:
            print("Nodes expanded to reach solution:", num_expanded)
            yield context
        else:
            rem_cons = [c for c in constraints if c not in to_eval] # constraint
            var = var_order[0]
            rem_vars = var_order[1:]
            for val in domains[var]:
                if display:
                    print("Setting", var, "to", val)
                num_expanded += 1
                rem_context = context|{var:val}
                # apply forward checking on remaining variables
                if len(var_order) > 1:
                    rem_vars_original = list((v, list(domains[v].copy())) for v
                    if display:
                        print("Original domains:", rem_vars_original)
                    # constraints that can't already be evaluated in rem_cons
                    rem_cons_ff = [c for c in constraints if c in rem_cons and n
                    for rem_var in rem_vars:
                        # constraints that can be evaluated by adding a value of
                        any_value = list(domains[rem_var])[0]
                        rem_to_eval = {c for c in rem_cons_ff if can_evaluate(c,
                        # new domain for rem_var are the values for which all ne
                        rem_vals = domains[rem_var].copy()
                        for rem_val in domains[rem_var]:
                            # no constraint with rem_var in the existing context
                            for c in rem_to_eval:
                                if not c.holds(rem_context|{rem_var: rem_val}):
                                    if rem_val in rem_vals:
```

```
                                        rem_vals.remove(rem_val)
                        domains[rem_var] = rem_vals
                        # order remaining variables by MRV
                        rem_vars.sort(key=lambda v: len(domains[v]))
                    if display:
                        print("After forward checking:", list((v, domains[v]) fo
                if rem_vars == [] or all(len(domains[rem_var]) > 0 for rem_var i
                    yield from mrv_dfs_solver(rem_cons, domains, context|{var:va
                # restore original domains if changed through forward checking
                if len(var_order) > 1:
                    if display:
                        print("Restoring original domain", rem_vars_original)
                    for (v, domain) in rem_vars_original:
                        domains[v] = domain
            if display:
                print("Nodes expanded so far:", num_expanded)

def mrv_dfs_solve_all(csp, var_order=None):
    """ depth-first CSP solver to return a list of all solutions to csp """
    global num_expanded
    num_expanded = 0
    if var_order == None:    # order variables by MRV
        var_order = list(csp.domains)
        var_order.sort(key=lambda var: len(csp.domains[var]))
    return list(mrv_dfs_solver(csp.constraints, csp.domains, {}, var_order))

def mrv_dfs_solve1(csp, var_order=None):
    """ depth-first CSP solver """
    global num_expanded
    num_expanded = 0
    if var_order == None:    # order variables by MRV
        var_order = list(csp.domains)
        var_order.sort(key=lambda var: len(csp.domains[var]))
    for sol in mrv_dfs_solver(csp.constraints, csp.domains, {}, var_order):
        return sol  # return first one
```

Run this solver on the sample problem.

**Note: Again there are no costs calculated.**

In [ ]:
```
def test_mrv_dfs_solver(csp_problem):
    solution = mrv_dfs_solve1(csp_problem)
    if solution == None:
        print('No solution')
    else:
        result_str = ''
        for name in solution.keys():
            result_str += '\n'+str(name)+': '+Day_Time().week_hour_number_to_day
        print(result_str[1:])

# call the Depth-First MRV Search solver
csp_problem = create_CSP_from_spec(sample_spec)
test_mrv_dfs_solver(csp_problem) # set display to True to see nodes expanded
```

# Assignment

**Name: Yu Lok Fu**

**zID: z5693605**

## Question 1 (4 marks)

Consider the search spaces for the fuzzy scheduling CSP solvers – domain splitting with arc consistency and the DFS solver (without forward checking).

- Describe the search spaces in terms of start state, successor functions and goal state(s) (1 mark)
- What is the branching factor and maximum depth to find any solution for the two algorithms (ignoring costs)? (1 mark)
- What is the worst case time and space complexity of the two search algorithms? (1 mark)
- Give one example of a fuzzy scheduling problem that is *easier* for the domain splitting with arc consistency solver than it is for the DFS solver, and explain why (1 mark)

For the second and third part-questions, give the answer in a general form in terms of fuzzy scheduling CSP size parameters.

**Answers for Question 1**

- Describe the search spaces in terms of start state, successor functions and goal state(s) (1 mark)

Fuzzy scheduling CSP solvers:The start state is the start time of the task has not yet been assigned, and all task variables have a complete domian of start time. The successor function refers to selecting a variable and dividing its domain into two subdomains, and using arc consistency to discard infeasible values in the subdomains. The goal state refers to the complete allocation of all tasks and allocation satisfies all constraints.
DFS solvers: The start state is an empty assignment that has not yet been assigned the value to a variable. The successor function is to select an unallocated variable and assign a possible value from the variable's domain. The goal state refers to all functions being assigned values and satisfying all constraint conditions.

- What is the branching factor and maximum depth to find any solution for the two algorithms (ignoring costs)? (1 mark)

If there n variables with domian size n and maximum domian size is D:
Fuzzy scheduling CSP solvers: The branching factor is 2, each domian is split into two subdomain. Since the domain of each variable will be continuously split until it becomes a singleton, the maximum depth is n * logs2 (D).
DFS solvers: The branching factor depends on the size of the maximum domain, so it is D. DFS solver may have only one variable per layer, so the maximum depth may be n.

- What is the worst case time and space complexity of the two search algorithms? (1 mark)

If there n variables with domian size n, maximum domian size is D, and there is e binary constraints:
Fuzzy scheduling CSP solvers: The worst case time is Θ(e * D^n). The space complexity is Θ(n).
DFS solvers: The worst case time is O(e * D^n+3). The space complexity is Θ(n * D+e).

- Give one example of a fuzzy scheduling problem that is *easier* for the domain splitting with arc consistency solver than it is for the DFS solver, and explain why (1 mark)

Assuming we have a task schedule t1, t2, t3... tk,
the binary constraint t1 must be completed before t2, t2 must be completed before t3, and the unary constraint t3 must be completed at Monday.
When Fuzzy scheduling CSP solvers run, it will propagate these columns of constraints. When t3 is scheduled on Monday, all earlier tasks will be automatically restricted to the Monday time before it. This can avoid exploring impossible scheduling. DFS does not have arc consistency functionality, so it will try many incorrect schedules, which will lead to an increase in time costs.

## Question 2 (5 marks)

Define the *cost* function for a fuzzy scheduling CSP (i.e. a node in the search space for domain splitting and arc consistency) as the total cost of the soft deadline constraints violated for all of the variables, assuming that each variable is assigned one of the best possible values from its domain, where a "best" value for a variable *v* is one that has the lowest cost to violate the soft deadline constraint (if any) for that variable *v*.

- Implement the cost function in the indicated cell and place a copy of the code below (3 marks)
- What is its computational complexity (give a general form in terms of fuzzy scheduling CSP size parameters)? (1 mark)
- Show that the cost function *f* never decreases along a path, and explain why this means the search algorithm is optimal (1 mark)

In [ ]:
```
# Code for Question 2
# Place a copy of your code here and run it in the relevant cell
def calculate_cost(self):
        """ this is really a function f = path cost + heuristic to be used by th
        total_cost = 0
        day_time = Day_Time()
        # TODO: write cost function
        for task, domain in self.domains.items():

            # skip the task, if it without deadline
            if task not in self.soft_day_time:
                continue

            # define the deadline,duration and cost rate for each task
            soft_deadline = self.soft_day_time[task]
            duration = self.durations[task]
            cost_rate = self.soft_costs.get(task,0)
```

```
            # convert soft_deadline from string to int
            date_type, date_value = day_time.string_to_week_hour_number(soft_dea
            if date_type == "week_hour_number":
               soft_deadline = date_value
            elif date_type == "day_number":
               soft_deadline = date_value * day_time.num_hours_in_day + (day_tim
            elif date_type == "hour_number":
               soft_deadline = (day_time.num_days_in_week - 1) * day_time.num_ho

            min_cost = float('inf') # the minmum cost for task
            for start_time in domain:
                # store task finish time and calculate lateness
                finish_time = start_time + duration
                if finish_time > soft_deadline:
                    lateness = finish_time - soft_deadline
                else:
                    lateness = 0

                # calculate the cost when task delay
                if callable(cost_rate):
                   cost = cost_rate(lateness)
                else:
                   cost = cost_rate * lateness
                if cost < min_cost:
                    min_cost = cost

            total_cost += min_cost

        return total_cost
```

**Answers for Question 2**

- What is its computational complexity (give a general form in terms of fuzzy scheduling CSP size parameters)? (1 mark)

We assume that there are n tasks and the maximum domain size is D.
There are two loop in calculate_cost() function, "for task, domain in self.domains.items():" is used to loop every task in CSP, "for start_time in domain:" is an embedded loop used to traverse all possible start times in the task domain. Therefore, the time complexity of this function should be O(n * D). In addition, only a few temporary variables were used in this function. Thus, the space complexity is O (1).

- Show that the cost function *f* never decreases along a path, and explain why this means the search algorithm is optimal (1 mark)

The cost function $f$ examines the domain of each task and selects the lowest possible cost for that task. In fuzzy scheduling CSP, the domain will be divided into two subdomains, which are subsets of its parent domain. When we choose the minimum cost in a subdomain, the minimum cost in the subdomain can only be equal to or greater than the minimum cost in the parent domain. Therefore, the total cost will only be greater or the same, and the cost function $f$ will never decrease on the search path. This ensures that the cost of exploring nodes is always minimized, and there will be no lower costs in future explorations than currently. This also means that the search algorithm is optimal.

## Question 3 (4 marks)

Conduct an empirical evaluation of the domain splitting CSP solver using the cost function defined as above compared to using no cost function (i.e. the zero cost function, as originally defined in the above cell). Use the *average number of nodes expanded* as a metric to compare the two algorithms.

- Write a function `generate_problem(n)` that takes an integer `n` and generates a problem specification with `n` tasks and a random set of hard constraints and soft deadline constraints in the correct format for the constraint solvers (2 marks)

Run the CSP solver (with and without the cost function) over a number of problems of size `n` for a range of values of `n`.

- Plot the performance of the two constraint solving algorithms on the above metric against `n` (1 mark)
- Quantify the performance gain (if any) achieved by the use of this cost function (1 mark)

In [ ]:
```python
# Code for Question 3
import random

def generate_problem(n):
    day_time = Day_Time()
    days = day_time.day_names
    hours = day_time.time_names
    binary_contraints = ["before", "after", "same-day", "starts-at"]
    hard_contraints = ["day_only", "time_only","starts-before", "starts-after",
    problems = []

    # generate task and duration
    for i in range(1, n+1):
        duration = random.randint(1,5)
        problems.append(f"task, t{i} {duration}")

    # generate binary constraints
    for i in range(1, n):
        constraint = random.choice(binary_contraints)
        problems.append(f"constraint, t{i} {constraint} t{i+1}")

    # generate one domain constraints
    def day_time_range(): # generate day time range
        day1 = random.randrange(len(days))
        day2 = random.randrange(day1, len(days)) # make sure day2 is not before
        time1 = random.randrange(len(hours))

        if day1 == day2:
            time2 = random.randrange(time1 + 1,len(hours))
        else:
            time2 = random.randrange(len(hours))
        return f"{days[day1]} {hours[time1]}-{days[day2]} {hours[time2]}"

    t = random.randint(1, n)
    constraint_type = random.choice(hard_contraints)
    match constraint_type:
```

```
        case "day_only":
            problems.append(f"domain, t{t} {random.choice(days)}")
        case "time_only":
            problems.append(f"domain, t{t} {random.choice(hours)}")
        case "starts-before":
            problems.append(f"domain, t{t} starts-before {random.choice(days)} {
        case "starts-after":
            problems.append(f"domain, t{t} starts-after {random.choice(days)} {r
        case "ends-before":
            problems.append(f"domain, t{t} ends-before {random.choice(days)} {ra
        case "ends-after":
            problems.append(f"domain, t{t} ends-after {random.choice(days)} {ran
        case "starts-in":
            problems.append(f"domain, t{t} starts-in {day_time_range()}")
        case "ends-in":
            problems.append(f"domain, t{t} ends-in {day_time_range()}")

    # generate soft deadlines
    for i in range(1, n + 1):
        cost = random.randint(1, 10)
        problems.append(f"domain, t{i} ends-by {random.choice(days)} {random.cho

    # output random problems
    result = ""
    for problem in problems:
        result += problem + "\n"

    return result
```

**Answers for Question 3**

- Plot the performance of the two constraint solving algorithms on the above metric against `n` (1 mark)

When CSP has a cost function, it can reduce the number of nodes in the tie extension. Due to the cost function prioritizing smaller costs, this can provide exploration guidance for solvers to accelerate the exploration speed and ensure the optimality of date selection

- Quantify the performance gain (if any) achieved by the use of this cost function (1 mark)

The performance gain can be obtained through this calculation formula: Gain=(Nno-cost - Nwith-cost)/Nno-cost * 100

## Question 4 (5 marks)

Compare the Depth-First Search (DFS) solver to the Depth-First Search solver using forward checking with Minimum Remaining Values heuristic (DFS-MRV). For this question, ignore the costs associated with the CSP problems.

- What is the worst case time and space complexity of each algorithm (give a general form in terms of fuzzy scheduling problem sizes)? (1 mark)

- What are the properties of the search algorithms (completeness, optimality)? (1 mark)
- Give an example of a problem that is *easier* for the DFS-MRV solver than it is for the DFS solver, and explain why (1 mark)
- Empirically compare the quality of the first solution found by DFS and DFS-MRV compared to the optimal solution (1 mark)
- Empirically compare DFS-MRV with DFS in terms of the number of nodes expanded (1 mark)

For the empirical evaluations, run the two algorithms on a variety of problems of size `n` for varying `n`. Note that the domain splitting CSP solver with costs should always find an optimal solution.

In [ ]:
```
# Code for Question 4
# Use your current spec generator or paste a fixed spec string
spec = generate_problem(2)
csp_problem = create_CSP_from_spec(spec)
# CSP solver
print("=== CSP solver ===")
solver = GreedySearcher(Search_with_AC_from_Cost_CSP(csp_problem))
test_csp_solver(solver)

# DFS
print("\n=== DFS ===")
test_dfs_solver(csp_problem)

# DFS-MRV
print("\n=== DFS-MRV + FC ===")
test_mrv_dfs_solver(csp_problem)
```

**Answers for Question 4**

- What is the worst case time and space complexity of each algorithm (give a general form in terms of fuzzy scheduling problem sizes)? (1 mark)

We assume that there are n tasks, the maximum domain size is D.
DFS:The worst case time is O(d^n), each task have D possible value, DFS may explores the entire tree. When there is n tasks, the deepest path of DFS is n, then the worst case space is O(n).
DFS-MRV: The worst case time is still O(d^n). In this case, the forward checking never discard anything. The forward checking will keep the domain list for each unassigned variable, there are n task and D value for each task. So, The worst case space is O(n * d).

- What are the properties of the search algorithms (completeness, optimality)? (1 mark)

Both DFS and DFS-MRV is not optimal, they just return the first feasible solution instead of the lowest cost solution. DFS have completeness when domain is finite, it will enumerate each task. DFS-MRV have completeness as well, the forward checking just discard the feasible value and MRV just reorder.

- Give an example of a problem that is *easier* for the DFS-MRV solver than it is for the DFS solver, and explain why (1 mark)

We use this problem as example:
task, t1 5
task, t2 5
task, t3 1
constraint, t1 after t2
constraint, t2 starts-at t3
domain, t1 ends-before tue 2pm
domain, t1 ends-by thu 2pm 4
domain, t2 ends-by fri 12pm 4
domain, t3 ends-by tue 10am 5
For DFS-MRV, this scheduling problem is easier to solve, as forward checking and MRV can quickly identify both hard domain constraints and soft deadline constraints for t1, t2, and t3: t2=t3 ends, t1 after t2, t1 ≤ Tue 2 pm. It discard the invalid time in advance, and DFS expands many useless nodes before detecting the conflict. That's why the DFS-MRV extended nodes are 7 and DFS is 87 when running this issue.

- Empirically compare the quality of the first solution found by DFS and DFS-MRV compared to the optimal solution (1 mark)

When the task size is small, the solutions produced by DFS and DFS-MRV are often the same as CSP solver. As the scale of the problem grows, DFS-MRV always produces better solutions than DFS due to its forward checking and variable sorting functions, which are closer to the solutions provided by CSP solver.

- Empirically compare DFS-MRV with DFS in terms of the number of nodes expanded (1 mark)

DFS-MRV always expand fewer nodes than DFS. MRV first selects the most constrained variables to identify dead ends in the early stages of exploring solutions. Forward checking will remove inconsistent values from adjacent fields to reduce the search space and thus decrease the number of node expansions.

## Question 5 (4 marks)

The DFS solver chooses variables in random order, and systematically explores all values for those variables in no particular order.

Incorporate costs into the DFS constraint solver as heuristics to guide the search. Similar to the cost function for the domain splitting solver, for a given variable *v*, the cost of assigning the value *val* to *v* is the cost of violating the soft deadline constraint (if any) associated with *v* for the value *val*. The *minimum cost* for *v* is the lowest cost from amongst the values in the domain of *v*. The DFS solver should choose a variable *v* with lowest minimum cost, and explore its values in order of cost from lowest to highest.

- Implement this behaviour by modifying the code in `dfs_solver` and place a copy of the code below (2 marks)
- Empirically compare the performance of DFS with and without these heuristics (2 marks)

For the empirical evaluations, again run the two algorithms on a variety of problems of size `n` for varying `n`.

In [ ]:
```
# Code for Question 5
# Place a copy of your code here and run it in the relevant cell
num_expanded = 0
display = False

def dfs_solver(constraints, domains, context, var_order, durations, soft_day_tim
    """ generator for all solutions to csp
        context is an assignment of values to some of the variables
        var_order is a list of the variables in csp that are not in context
    """
    global num_expanded, display
    to_eval = {c for c in constraints if c.can_evaluate(context)}
    if all(c.holds(context) for c in to_eval):
        if var_order == []:
            print("Nodes expanded to reach solution:", num_expanded)
            print("Total soft cost of solution:", total_cost) # print cost of so
            yield context
        else:
            rem_cons = [c for c in constraints if c not in to_eval]

            day_time = Day_Time() # create a object to covert deadline
            min_cost = float('inf') # best cost for each variable
            next_var = None # the next varibale will choose (best one)
            vals_cost_pair = None # list of value and cost pair


            for var in var_order: # loop each unassign variable

                soft_deadline = soft_day_time[var]
                cost_rate = soft_costs.get(var,0)
                duration = durations.get(var,0)

                # convert and calculate soft deadline
                if soft_day_time is not None and var in soft_day_time:
                    date_type, date_value = day_time.string_to_week_hour_number(
                    if date_type == "week_hour_number":
                         soft_deadline = date_value
                    elif date_type == "day_number":
                         soft_deadline = date_value * day_time.num_hours_in_day
                    elif date_type == "hour_number":
                         soft_deadline = (day_time.num_days_in_week - 1) * day_t

                vals_with_costs = [] # a list to hold start-time and cost pair
                min_cost_for_var = float('inf') # tracks the minmum cost of vari

                # for each start time to calculate finish time and delay
                for start_time in domains[var]:
                    finish_time = start_time + duration
                    if finish_time > soft_deadline:
                        lateness = finish_time - soft_deadline
```

```
                    else:
                        lateness = 0

                    # calculate the cost when task delay
                    if callable(cost_rate):
                        cost = cost_rate(lateness)
                        vals_with_costs.append((start_time, cost))
                    else:
                        cost = cost_rate * lateness
                        vals_with_costs.append((start_time, cost))
                    if cost < min_cost: # update the min cost
                        min_cost = cost

                # sorts pair by cost
                pairs = [(cost, start_time) for start_time, cost in vals_with_co
                pairs.sort()

                # pass the global min cost to local variable, remember its sorte
                if min_cost < min_cost_for_var:
                    min_cost_for_var = min_cost
                    next_var = var
                    vals_cost_pair = vals_with_costs

            # new var order without current best varialbe
            order_based_on_cost = []
            for val in var_order:
                if val != next_var:
                   order_based_on_cost.append(val)

            for value, cost in vals_cost_pair:
                if display:
                    print("Setting", next_var, "to", value)
                num_expanded += 1
                soft_cost = total_cost + cost
                yield from dfs_solver(rem_cons, domains, context | {next_var: va

def dfs_solve_all(csp, var_order=None):
    """ depth-first CSP solver to return a list of all solutions to csp """
    global num_expanded
    num_expanded = 0
    if var_order == None:    # use an arbitrary variable order
        var_order = list(csp.domains)
    return list(dfs_solver(csp.constraints, csp.domains, {}, var_order, csp.dura

def dfs_solve1(csp, var_order=None):
    """ depth-first CSP solver """
    global num_expanded
    num_expanded = 0
    if var_order == None:    # use an arbitrary variable order
        var_order = list(csp.domains)
    for sol in dfs_solver(csp.constraints, csp.domains, {}, var_order, csp.durat
        return sol  # return first one
```

**Answers for Question 5**

- Empirically compare the performance of DFS with and without these heuristics (2 marks)

Compared to the original DFS algorithm, the improved algorithm is cost oriented in searching for solutions. When the problem size is small, the solutions generated by the two algorithms are often consistent. As the size of the problem increases, the solutions generated by improved algorithms typically have lower costs and tend towards the optimal solution. Therefore, in terms of performance, the improved DFS algorithm outperforms the original algorithm.

## Question 6 (3 marks)

The CSP solver with domain splitting splits a CSP variable domain into *exactly two* partitions. Poole & Mackworth claim that in practice, this is as good as splitting into a larger number of partitions. In this question, empirically evaluate this claim for fuzzy scheduling CSPs.

- Write a new `partition_domain` function that partitions a domain into a list of `k` partitions, where `k` is a parameter to the function (1 mark)
- Modify the CSP solver to use the list of `k` partitions and evaluate the performance of the solver using the above metric for a range of values of `k` (2 marks)

In [ ]:
```python
# Code for Question 6
@staticmethod
def partition_domain(dom, k):
    items = list(dom)
    n = len(items)

    # make sure k is be a valid number
    if k < 1: # at least one partition
        k = 1
    if k > n: # don't generate empty partition
        k = n

    domain_list = [] # store partition
    remain_partition = k # the remain chunk of partition
    i = 0 # store the index of value we take from dom

    while remain_partition:
        take = (n - i + remain_partition - 1) // remain_partition # calculate ch
        domain_list.append(set(items[i:i + take])) # add chunk to list
        # update index and remaining partition
        i += take
        remain_partition -= 1
    return domain_list

def neighbors(self, node):
    """returns the neighboring nodes of node."""
    neighs = []
    var = select(x for x in node.domains if len(node.domains[x]) > 1)  # chosen
    if var:
        k = 3  # set number of partition
        parts = self.partition_domain(node.domains[var], k)
        to_do = self.cons.new_to_do(var, None)
        # discard empty partition
        non_empty_parts = []
        for part in parts:
```

```
            if len(part) > 0:
                non_empty_parts.append(part)

        self.display(2, "Splitting", var, "into", non_empty_parts)
        for dom in parts:
            newdoms = node.domains | {var: dom} # overwrite domain of var with d
            cons_doms = self.cons.make_arc_consistent(newdoms, to_do)
            if all(len(cons_doms[v]) > 0 for v in cons_doms):
                # all domains are non-empty
                # make new CSP_with_Cost node to continue the search
                csp_node = CSP_with_Cost(
                    cons_doms, self.durations, self.constraints,
                    self.cost_functions, self.soft_day_time, self.soft_costs
                )
                neighs.append(Arc(node, csp_node))
            else:
                if len(dom) == 0:
                    continue
                else:
                    self.display(2, "...", var, "in", dom, "has no solution")
    return neighs
```

**Answers for Question 6**

- Modify the CSP solver to use the list of `k` partitions and evaluate the performance of the solver using the above metric for a range of values of `k` (2 marks)

When the domain is large, splitting an appropriate number of partitions can increase the pruning efficiency of arc consistency on subdomains, reduce the depth of the search tree, and decrease the running time of the algorithm. However, excessive partitions can greatly increase the number of AC runs, which can increase node expansion and space utilization.