Solving the 'Scheduling problem' and 'Dinosaur mystery' modelled as Constraint Satisfaction Problems

Radu Galan, gr. 256

1. Abstract

The scheduling problem is a very popular situation of optimizing a list of tasks and the dinosaur mystery is a classical riddle-like problem. Both of them can be modelled with emphasis on the constraints extracted from the problem description. The purpose is to detect at least one but preferably all solutions of the problems in as little time and memory space as possible. In this documentation I am going to describe the process of solving the problems through different type of methods and analyse the results obtained.

2. Introduction

In the following chapters I am going to describe the problem in more detail and analyse the limitation and possibilities relative to each of the two problems. I am going to then solve the problems through multiple methods: stochastic backtracking, two basic search algorithms (Chronological Backtracking and Iterative Broadening), and optimize using two problem reduction methods (Node Consistency and Arc Consistency). For all of these I am going to evaluate the performance based on the RAM usage, memory allocation, iterations computed, constraints evaluation computed and time needed.

A constraint satisfaction problem can be defined as a query over a set of objects defined with one or more variables (each of them existing within a certain domain). This query needs to find a solution that satisfies a list of constraints. We have just described a triple of (Z, D, C), where Z is the finite set of variables $\{x_1, ..., x_n\}$, D is a function that maps every variable to a set of objects, and C is a finite set of constraints on a subset of the variables in Z.

3. Problem descriptions

a. Scheduling problem

With 4 different types of subjects at this particular school we need to find which 3 make the best time scheduling for everyone given the constraints. The four subjects are: Mathematics, Chemistry, Philosophy, or History; which I will call by the short names: MA, CH, PH, HI. Each subject must be assigned a period from the 8 available, where morning is 1-4 and evening is 5-8. The constraints are:

- 1.Exactly 3 subjects must be selected
- 2.CHEM is offered odd periods and MATH is offered all periods
- 3.HIST is offered periods 2 and 4. PHIL is offered periods 5 and 7.
- 4.CHEM and MATH are taught on south campus.
- 5.HIST and PHIL are taught on north campus.
- 6.You can't schedule back-to-back classes on opposite ends of campus.
- 7.Only work mornings or afternoon
- 8.No first period

The 7. and 8. are optional but must also be solved.

First, we can simplify the constraints to find the correct solution:

1. Subset of 3 subjects from {MA,CH,PH,HI}
2. CH -> 1,3,5,7
3. HI -> 2,4 PH -> 5,7
4. {CH,MA} next to {HI,PH}
5. All -> {1,2,3,4} or All -> {5,6,7,8}
6. None -> {1}

Period\Subject	MA	CH	PH	HI
1	####	####	#####	####
2		####	####	
3			####	####
4		####	####	
5			✓	####
6		####	####	####
7		✓		####
8	✓	####	####	####

The selected periods can either be only 1-4 or 5-8.
For the 1-4 periods it is impossible. Because we need 3 subjects but we only have 3 periods (2,3,4) it means that we'll need a subject in each period. We only have 2 subjects in each campus and cannot repeat a subject, therefor we cannot respect the constraint of not having back-to-back subjects with different campuses.
For the 5-8 subject we cannot use history at all but we can find exactly one solution to respect the back-to-back constraint:

```
[['5', 'PH'], ['7', 'CH'], ['8', 'MA']]
```

For this problem we do not need to add any other variables other than 'subject' and 'period'.

b. Dinosaur mystery

There are 5 types of dinosaurs (Eucentrosaurus, Hadrosaurus, Herrerasaurus, Megasaurus, and Nuoerosaurus) discovered in 5 countries (Argentina, Canada, China, England, and the U.S).
We need to discover where each dinosaur has been discovered considering that each of them belong to a single country and no two dinosaurs belong to the same. I am going to use only the two letters in redacting the solution (eu, ha, he, me, nu -- and -- ar, ca, ch, en, us) for simplicity.
In order to find to find the correct solution amongst the domain of possibilities we are going to use the constraints that are provided:

- 1.The dinosaur from China is the largest of the bunch.
- 2.Herrerasaurus was the smallest.
- 3.Megasaurus was larger than Hadrosaurus or Eucentrosaurus.
- 4.The dinosaurs from China and North America were planteaters.
- 5.Megasaurus ate meat.

- 6.Eucentrosaurus lived in North America.
- 7.The dinosaurs from North America and England lived later than Herrerasaurus.
- 8.Hadrosaurus lived in Argentina or the U.S.

From these constraints we can discover what are the objects and what attributes has each of them. We could model the objects in two ways by choosing either the country or the dinosaur. Because it makes more sense, we are choosing the dinosaur and the country will be given by the position of the dinosaur in the resulted premutation. For example (sorted alphabetically):

```
self.countries = {'ar': '0', 'ca': '1', 'ch': '2', 'en': '3', 'us': '4'}
```

And the solution is:

```
['he', 'eu', 'nu', 'me', 'ha']
```

Then it means that Herrerasaurus is from Argentina, Eucentrosaurus is from Canada, Nuoerosaurus is from China, Megasaurus is from England, and Hadrosaurus is from the United States. This is the actual solution to the problem (this has been solved by hand using the table method but also by researching the actual origins of the dinosaurs online).

Country/Dino	EU	HA	HE	ME	NU
AR	####		✓		
CA	✓	####	####	####	
CH	#####	#####	#####	####	✓
EN	####	####		✓	
US		✓	####	####	

I color-coded the variables filtered domain relative to what constraint affected them. We can clearly see there is only one viable solution to the problem

From 1. 2. and 3. I realize that in CH there can either be ME or NU. (ME/NU -> CH)

From 4. and 5. I realize that ME can only be from AR or EN (ME -> AR/EN)

From 6. I realize that EU can be from US or CA (EU -> US/CA)

From 7. I realize that HE is not from US, CA or EN (HE -> AR/CH)

From 8. I realize that HA is from AR or US (HA -> AR/US)

A more complete model of the problem should of course include all the variables mentioned. We should then add for: vegetarian (yes or no), size (using an ordered list 1,2,3,4,5). The simple method has been used for the stochastic backtracking and the latter mentioned method with multiple variables has been used for the other methods implemented.

4. Solving descriptions

I am going to implement the following methods: stochastic backtracking (in the stochastic algorithm category), two basic search algorithms (Chronological Backtracking and Iterative Broadening), and optimize using two problem reduction methods (Node Consistency and Arc Consistency). All the algorithms have been implemented in an almost identical manner for the two problems since the purpose was to create an abstract and modular solution. Although certain little differences and particularities have been met for each problem I am going to present only once each algorithm.

a. Stochastic algorithm

The stochastic algorithm is similar for each problem. For these two methods the algorithms can be found under: 'dino_simple.py' and 'schedule_simple.py'. All other methods are found in 'dino_optim.py' and schedule_optim.py".
In order to find the wanted solution, I have used a simple algorithm that generates all the possible permutations of the dinosaurs name list. I took all the permutations and tried to extract the correct solution by checking all the constraints (but here I have used the simplified constraints since I did not implement the attributes: size, vegan-diet). I did the exact same thing for the scheduling problem but for the variables subject and period.
For the dinosaur mystery this prints out:

```
We got solutions: [['ca-eu', 'us-ha', 'ar-he', 'en-me', 'ch-nu']]
```

For the scheduling problem this prints out:

```
We got solutions: [['5', 'ph'], ['7', 'ch'], ['8', 'ma']]
```

```
    def permutation(self, lst):
        '''
                This function will generate all permutation with all the elements of given list
                In: the list of elements
                Out: a list of all the permutations
        '''

        if len(lst) == 0:
            return []

        if len(lst) == 1:
            return [lst]

        l = []
        self.iterations += 1
        for i in range(len(lst)):
            m = lst[i]

            remLst = lst[:i] + lst[i + 1:]

            self.memory_history.append([remLst])

            for p in self.permutation(remLst):
                l.append([m] + p)

        return l

    def run_simple_perm(self):
        '''
                This function will run the 'backtrack' algorithm over the variables. It will generate the
                permutations and then check which are correct
                In: -
                Out: prints our the results
        '''
          #get country names
        a = list(self.country_domain_objects.keys())

          #all the permutations
        results = self.permutation(a,0)

          #check which respect the constraints
        solutions = []
        for p in results:
            if self.check_constraints(p):
                for i in range(len(p)):
                    p[i]+= "-" + self.dinos[i]
                solutions.append(p)

        print("We got solutions: ", solutions)
```

b. Basic search algorithm

The two basic search algorithms that I have chosen are:

- Chronological Backtracking
- Iterative Broadening

Chronological backtrack will iterate over empty variables an attempt to find values that respect the constraints and when that fails to happen it will fall-back to a previous value. By this method it will iterate over all viable (not possible) choices.

```
def run_chronological_backtrack(self):

    #designed a variable for each object's property
    self.D = {'subject0':['ma', 'ch', 'ph', 'hi'], 'time0':['1','2','3','4','5','6','7','8'],
            'subject1':['ma', 'ch', 'ph', 'hi'], 'time1':['1','2','3','4','5','6','7','8'],
            'subject2':['ma', 'ch', 'ph', 'hi'], 'time2':['1','2','3','4','5','6','7','8'],
            }
    #the list of used variables
    var = ['subject','time']
    self.no = 3
    self.variables = self.D.keys()
    self.results = []

    #the function that iterates through all the choices
    self.chr_backtrack()

    #detecting unique solutions
    unique_solutions = []
    for el in self.results:
         adable = [ [el['subject0'], el['time0']], [el['subject1'], el['time1']],     [el['subject2'],
    el['time2']] ]
        adable.sort(key= lambda x: x[1], reverse = False)

        if adable not in unique_solutions:
            unique_solutions.append(adable)

    print("We got unique solutions: ", len(unique solutions))
    for el in unique_solutions:
        print(el)


def chr_backtrack(self):
    #saving the results if a proper solution
    if None not in self.variables.values():
        self.results.append(deepcopy(self.variables))
        return
    #find variable to give value to
    var = self.find_unused_var()
    self.iterations += 1

    #iterate over possible variable values
    for d_orig in self.D[var]:
        #assigning value and dealing with memory
        d = deepcopy(d_orig)
        self.variables[var] = d
        self.D[var].remove(d_orig)

        if self.constraints() and self.nu_constraints():
            #go to next variable if it's a good solution so far
            self.chr_backtrack()

        #backtracking to last setup
        self.variables[var] = None
        self.D[var].append(d)
```

The second method of searching the solution is Iterative Broadening. This relies on the idea that there is not one variable that is more important than another (like the backtracking algorithm supposes). It is basically a depth-first search method.

```
    def run_iterative_broadening(self):

        #designed a variable for each object's property
        self.D = {'subject0':['ma', 'ch', 'ph', 'hi'], 'time0':['1','2','3','4','5','6','7','8'],
                'subject1':['ma', 'ch', 'ph', 'hi'], 'time1':['1','2','3','4','5','6','7','8'],
                'subject2':['ma', 'ch', 'ph', 'hi'], 'time2':['1','2','3','4','5','6','7','8'],
                }
        #the list of used variables
        var = ['subject','time']
        self.no = 3
        self.variables = self.D.keys()
        self.results = []

        #the function that iterates through all the choices
        self.iterative_broadening()

        #detecting unique solutions
        unique_solutions = []
        for el in self.results:
            adable = [ [el['subject0'], el['time0']], [el['subject1'], el['time1']], [el['subject2'],
el['time2']] ]
            adable.sort(key= lambda x: x[1], reverse = False)

            if adable not in unique_solutions:
                unique_solutions.append(adable)

        print("We got uniques: ", len(unique_solutions))
        for el in unique_solutions:
            print(el)

    def iterative_broadening(self):
        #save if a complete solution
        if None not in self.variables.values():
            self.results.append(deepcopy(self.variables))
            return

        #choose the next domain
        var = self.find_unused_var()
        domain = self.D[var]
        counted_attempts = 0
        max_attempts = len(domain)
        while counted_attempts < max_attempts and len(domain) > counted_attempts:
            value = domain[counted_attempts]
            counted_attempts += 1
            save = deepcopy(self.variables)
            self.memory_history.append([save])
            self.variables[var] = value
            self.total_counter += 1
            if self.constraints() and self.nu_constraints():
                #iterate through possible solution
                self.iterative_broadening()
            self.variables[var] = None
            self.variables = save
```

c. Problem reduction

The methods chosen for the problem reduction algorithm are Arc consistency and Node consistency. Both of them are implemented identically for the two different problems and can be added as an extra step in the search algorithm. All the methods were added on top of the chronological backtracking method

The algorithm for the arc consistency method is the following:

```
    def ac3(self):
        '''
        Function that check the arc consistency and removes unfit values from the domain
automatically
        '''
        for a in self.D.keys():
            for var_a in self.D[a]:
                to_delete = True
                #for each value of node (one end of the arc)
                for b in self.D.keys():
                    if a != b:
                        for var_b in self.D[b]:
                            #and each different value of a node (the other end of the arc)
                            self.variables[a] = var_a
                            self.variables[b] = var_b

                            #check if the cosntraint have anything against the pair
                            if self.constraints_ac() is True:
                                to_delete = False

                            self.variables[a] = None
                            self.variables[b] = None
                if to_delete:
                    #delete the node if it is the case
                    self.D[a].remove(var_a)
                    print("Fake: ", var_a, "_",a)
```

The algorithm for the node consistency I have changed the regular permutation function as follows:

```
    def permutation_nc(self):
        #saving the results if a proper solution
        if None not in self.variables.values():
            self.results.append(deepcopy(self.variables))
            return


        #find variable to give value to
        var = self.find_unused_var()
        add_back = []

        #for each value of the chosen variable eliminate all values that are not fitting from the
domain
        for value in self.D[var]:
            self.variables[var] = value
            order = []
            order_time = []
            for i in range(self.no):
                order.append(self.variables["subject" + str(i)])
                order_time.append(self.variables["time" + str(i)])

            if (self.constraints(order, order_time) and self.nu_constraints(order, order_time)) is
False:
                add_back.append(value)
                self.D[var].remove(value)

        self.variables[var] = None
        #iterate over possible variable values
        for d_orig in self.D[var]:
            #assigning value and dealing with memory
            d = deepcopy(d_orig)
            self.variables[var] = d
            self.D[var].remove(d_orig)

            if self.constraints() and self.nu_constraints():
                #go to next variable if it's a good solution so far
                self.permutation_nc()

            #swithcing to last values
            self.variables[var] = None
            self.D[var].append(d)

        #changing values
        for el in add_back:
            self.D[var].append(el)
```

5. Comparison

We will compare the methods with one another against different metrics as described before in the introduction.

a. Time

Cost of time average for 30 random attempts

Prob\Method	Permutations Optim	Chronological Backtracking	Chr. Back. With NC	Chr. Back. With AC	Iterative Broadening
Dino Mystery	0.0002	0.2879	0.2616	0.1409	0.6273
Schedule Prob	0.9224	0.0300	0.0356	0.0556	0.2784

b. Size and memory

Usage of space allocation:

Prob\Method	Permutations Optim	Chronological Backtracking	Chr. Back. With NC	Chr. Back. With AC	Iterative Broadening
Dino Mystery	1920	200312	178016	87616	361280
Schedule Prob	578928	26728	26728	33920	26728

Usage of RAM memory:

Prob\Method	Permutations Optim	Chronological Backtracking	Chr. Back. With NC	Chr. Back. With AC	Iterative Broadening
Dino Mystery	90.6mb	97.7mb	97.0mb	94.0mb	94.0mb
Schedule Prob	101.4mb	91.1mb	91.2mb	91.6mb	91.5mb

c. Constraint graph

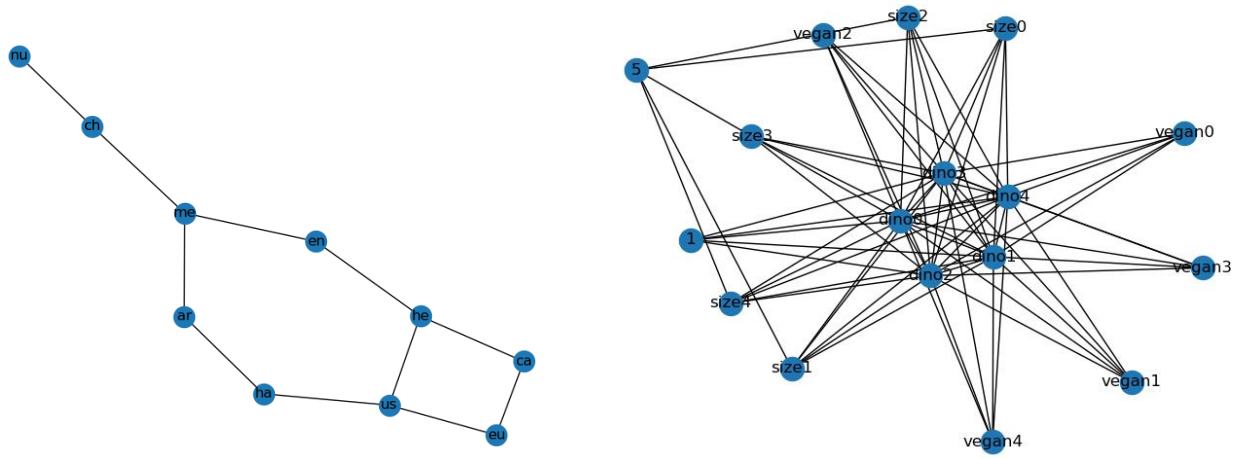
Constraint function count:

Prob\Method	Permutations Optim	Chronological Backtracking	Chr. Back. With NC	Chr. Back. With AC	Iterative Broadening
Dino Mystery	120	12278	10812	26260	6430
Schedule Prob	1344	1552	1636	1552	2056

Iteration count:

Prob\Method	Permutations Optim	Chronological Backtracking	Chr. Back. With NC	Chr. Back. With AC	Iterative Broadening
Dino Mystery	86	3085	2365	3247	1448
Schedule Prob	28978	231	216	231	151

Here we can observe the stochastic and the chronological backtracking constraint graph.



6. Conclusion

I have compared multiple methods on the two presented problems and the results are mostly as expected. The complexity of both problems lays in the constraint verification and so the methods of optimization implemented did not consistently or properly improved the performance overall. Although the number of iterations significantly decreased the cost in the number of extra calls of the constraints function proved to be quite computationally intensive.