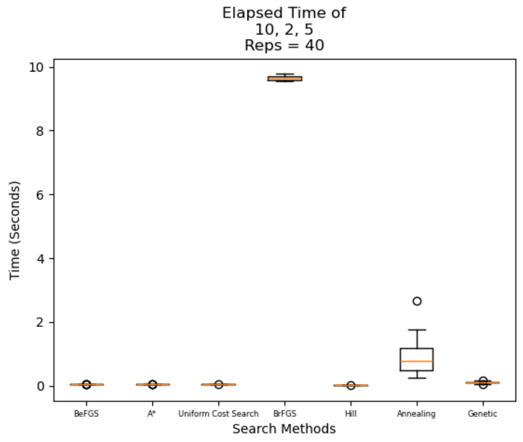
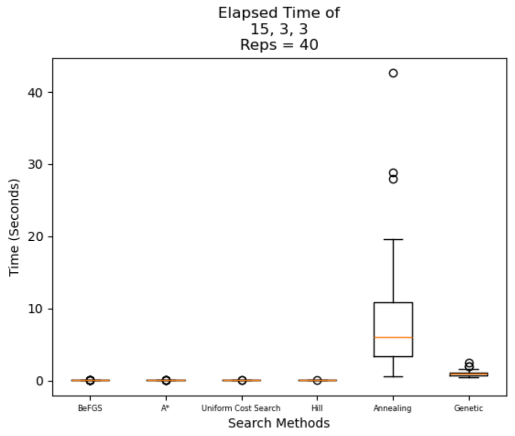
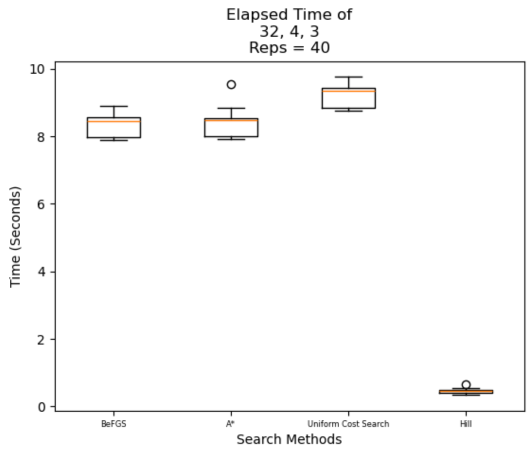
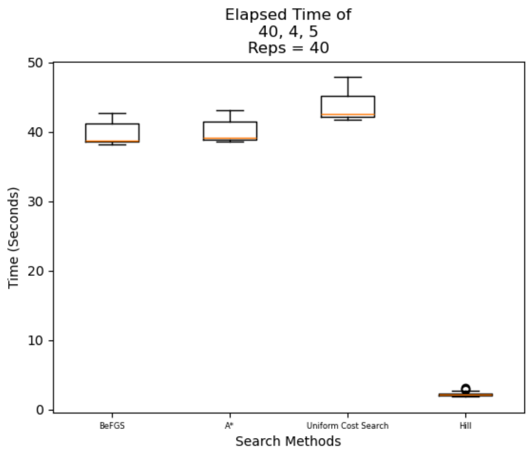
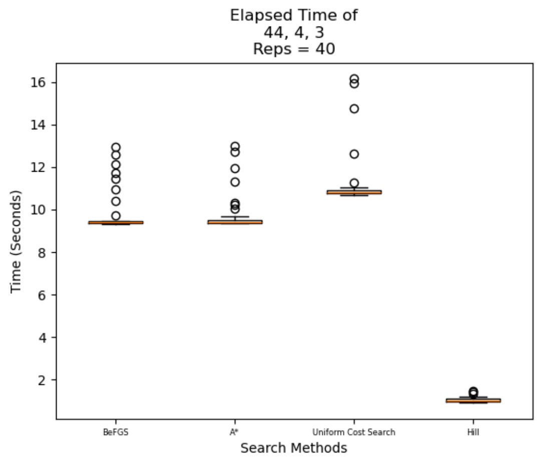
Q1

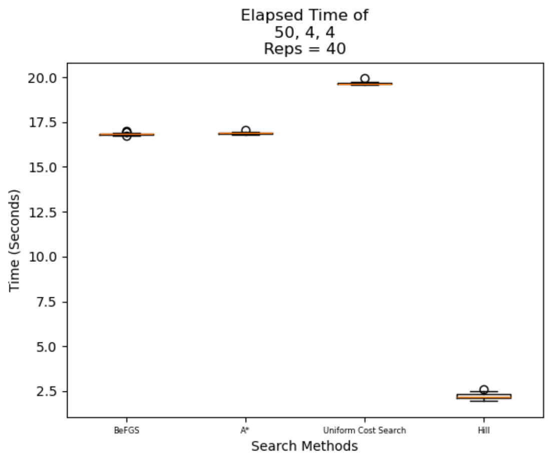

Elapsed Time of
10, 2, 5
Reps = 40
Time (Seconds)
Search Methods
BeFGS
A*
Uniform Cost Search
BrFGS
Hill
Annealing
Genetic

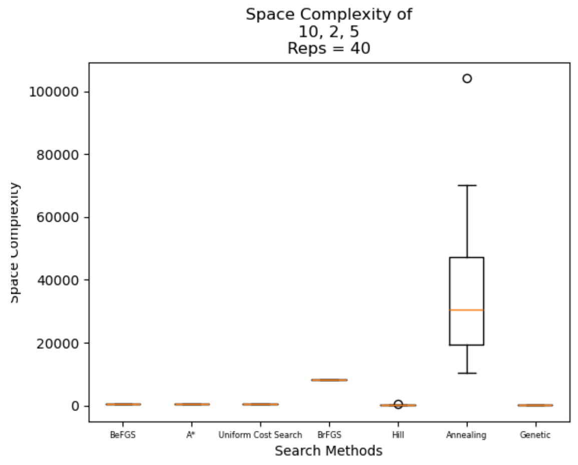

Elapsed Time of
15, 3, 3
Reps = 40
Time (Seconds)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill
Annealing
Genetic


Elapsed Time of
32, 4, 3
Reps = 40
Time (Seconds)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill

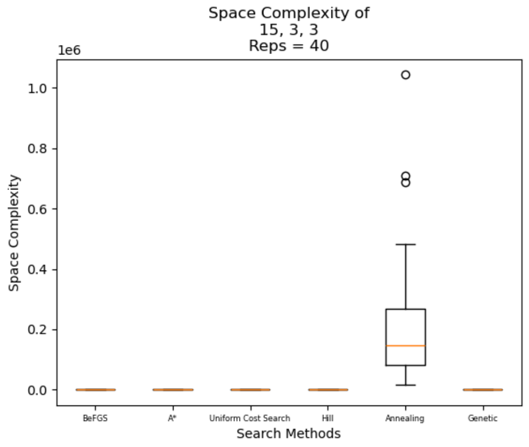

Elapsed Time of
40, 4, 5
Reps = 40
Time (Seconds)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill


Elapsed Time of
44, 4, 3
Reps = 40
Time (Seconds)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill

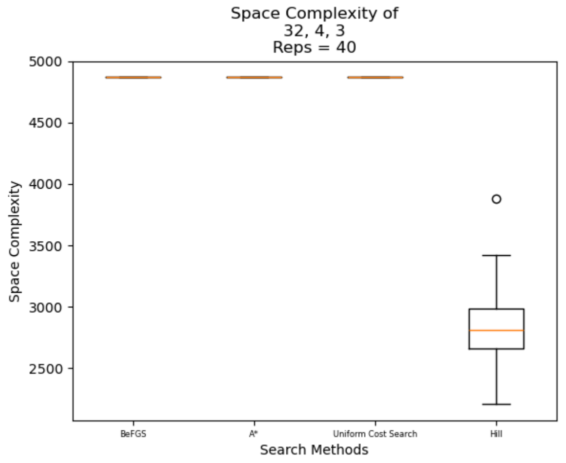

Elapsed Time of
50, 4, 4
Reps = 40
Time (Seconds)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill



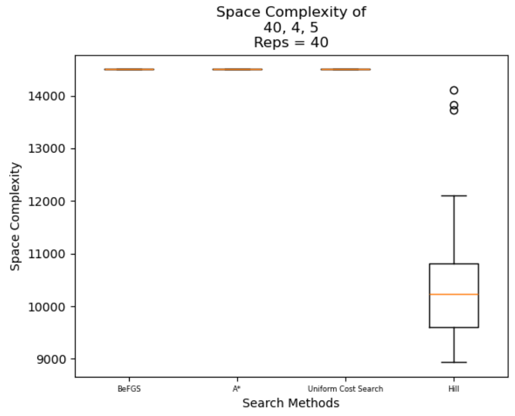
Space Complexity of
10, 2, 5
Reps = 40
Space Complexity
Search Methods
BeFGS
A*
Uniform Cost Search
BrFGS
Hill
Annealing
Genetic



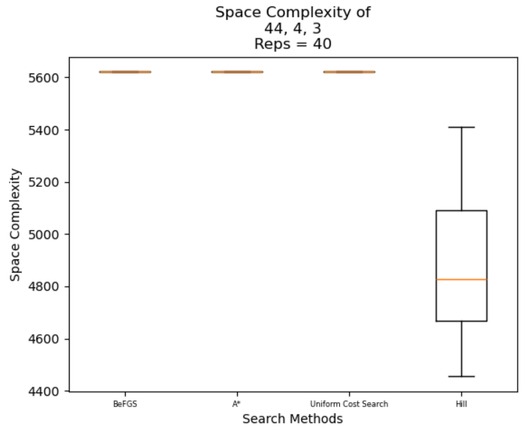
Space Complexity of
15, 3, 3
Reps = 40
1e6
Space Complexity
Search Methods
BeFGS
A*
Uniform Cost Search
Hill
Annealing
Genetic



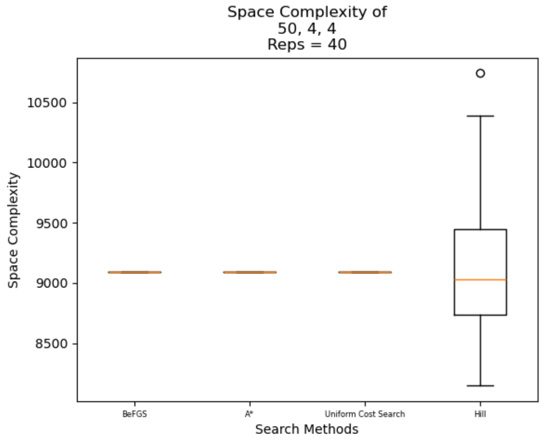
Space Complexity of
32, 4, 3
Reps = 40
Space Complexity
Search Methods
BeFGS
A*
Uniform Cost Search
Hill



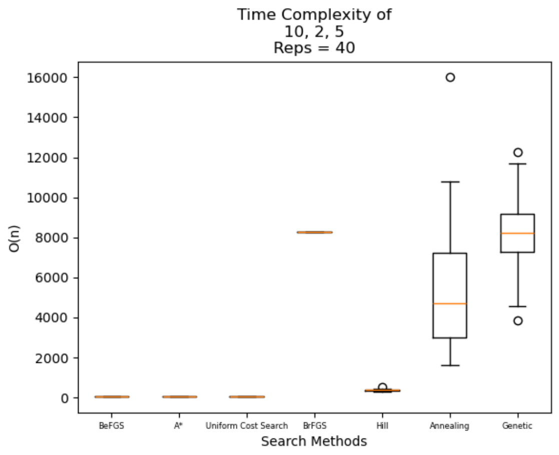
Space Complexity of
40, 4, 5
Reps = 40
Space Complexity
Search Methods
BeFGS
A*
Uniform Cost Search
Hill

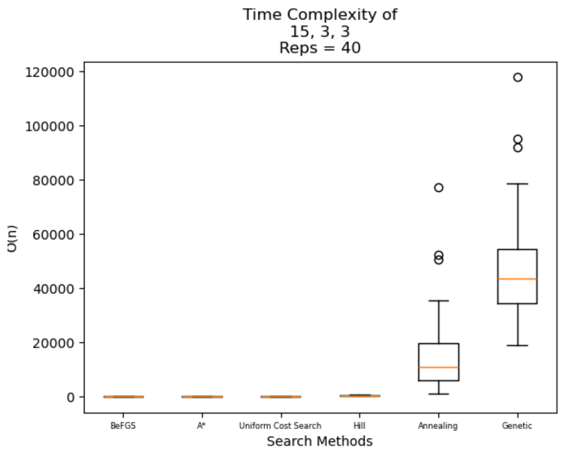


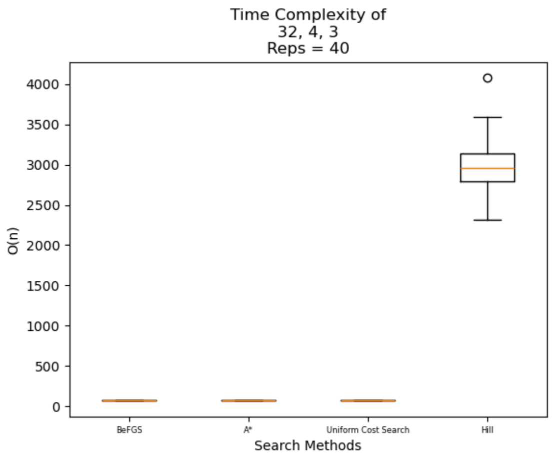
Space Complexity of
44, 4, 3
Reps = 40
Space Complexity
Search Methods
BeFGS
A*
Uniform Cost Search
Hill

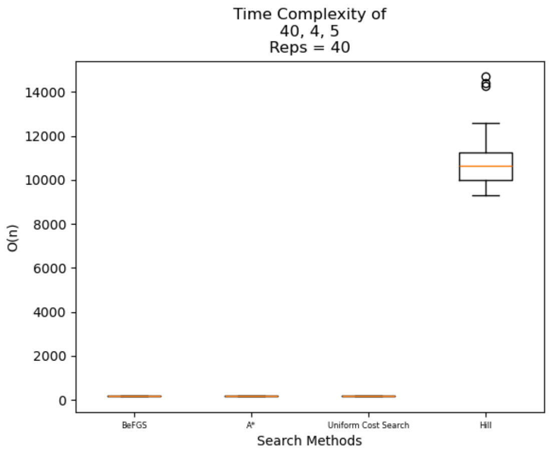


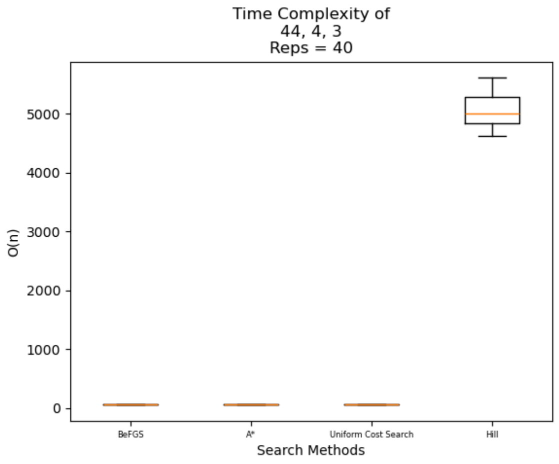
Space Complexity of
50, 4, 4
Reps = 40
Space Complexity
Search Methods
BeFGS
A*
Uniform Cost Search
Hill

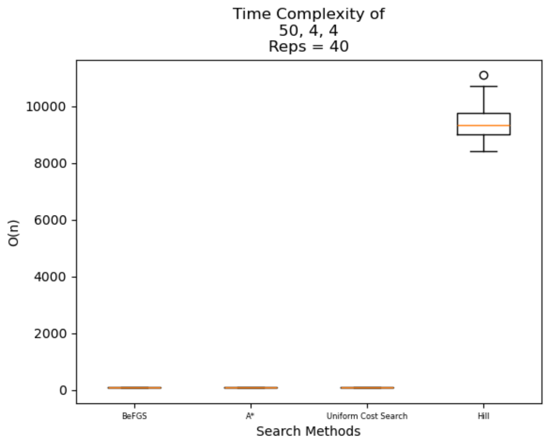

Time Complexity of
10, 2, 5
Reps = 40
O(n)
Search Methods
BeFGS
A*
Uniform Cost Search
BrFGS
Hill
Annealing
Genetic


Time Complexity of
15, 3, 3
Reps = 40
O(n)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill
Annealing
Genetic


Time Complexity of
32, 4, 3
Reps = 40
O(n)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill


Time Complexity of
40, 4, 5
Reps = 40
O(n)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill


Time Complexity of
44, 4, 3
Reps = 40
O(n)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill


Time Complexity of
50, 4, 4
Reps = 40
O(n)
Search Methods
BeFGS
A*
Uniform Cost Search
Hill

Findings of Project 1:

Genetic Search and Hill Climbing have significant variance, but often perform better than the worst case scenario for Elapsed Time (worst case being BrFS). Hill Climbing often has a higher O(n) Time Complexity than other best methods (Best Methods being BeFGS, A*, and UCS) however, it has a variance on space complexity that oftentimes is below that of the other methods, as well as a variance on the elapsed time which is lower. Variance on Tree Searches is significantly lower than Iterative Improvement Algorithms, but Iterative Improvement algorithms often perform better than at least the worst-case. Simulated Annealing is inconsistent at best and often fails altogether.

In the graphs, all algorithms that take longer than 2 minutes for the given task (for one iteration) are omitted. Iterative Improvement algorithms are repeatedly rerun until elapsed time is equal to 2 minutes, or the solution was found.

Q2

Formulate the scheduling problem as a constraint satisfaction problem:

Variables: $x_{0,0}, x_{0,1}, \dots, x_{0,n-1}, x_{1,0}, x_{1,1}, \dots, x_{p-1,n-1}$

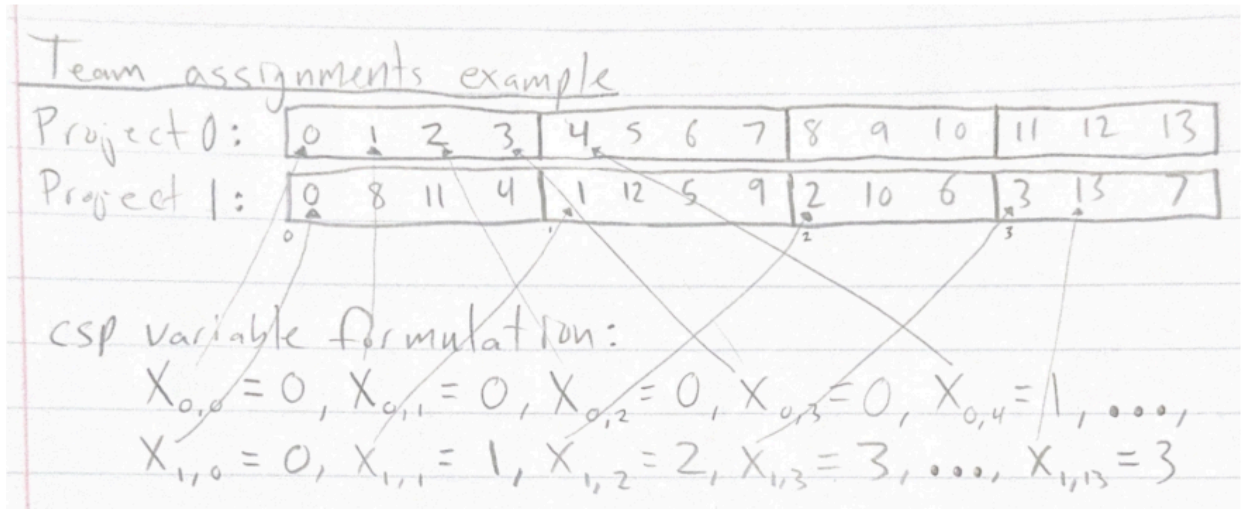
Domains: $D_i = \{0, 1, \dots, \lceil \frac{n}{k} \rceil - 1\}$

Each variable represents a person that can be assigned a value of the team they will reside in a certain project. Each person has a different variable for each project

Constraints:

1. No team is overfilled.
   (The first $\lceil \frac{n}{k} \rceil + (n - 1)\%k + 1 - k$ teams are size k, and the rest are $k - 1$)
2. $x_{i,j} = x_{i,k} \wedge j \neq k \Rightarrow \neg\exists_{i' \neq i}: x_{i',j} = x_{i',k}$
   Aka, nobody is in a team with anybody more than once

Illustration:


Team assignments example
Project 0: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
Project 1: 0 8 11 4 1 12 5 9 2 10 6 3 13 7
0
1
2
3
CSP variable formulation:
$X_{0,0}=0, X_{0,1}=0, X_{0,2}=0, X_{0,3}=0, X_{0,4}=1, \ldots,$
$X_{1,0}=0, X_{1,1}=1, X_{1,2}=2, X_{1,3}=3, \ldots, X_{1,13}=3$

This illustration shows how you get from a project and team assignment to values for each variable.

Modify the code so that it can handle the constraints you have defined:

The main change that I made was changing the constraint function to also take an "assignment" parameter. This allowed my code to analyze more than just 2 variables.

```
class SchedulingCSP(csp.CSP):
   def __init__(self, n, k, p):
       self.n = n
       self.k = k
       self.p = p

       variables = [(i, j) for i in range(p) for j in range(n)]

       domains = {}
       domain = [i for i in range(math.ceil(n / k))]
       for var in variables:
           domains[var] = domain
```

```
        neighbors = {(i, j): (i, j) for i in range(p) for j in range(n)}

        constraints = self.valid_value

        super().__init__(variables, domains, neighbors, constraints)

    def nconflicts(self, var, val, assignment):
        """Return the number of conflicts var=val has."""
        return 0 if self.constraints(var, val, assignment) else 1

    def valid_value(self, var, value, assignment):
        if self.team_full_already(var[0], var[1], value, assignment):
            return False
        return self.team_can_be_joined(var[0], var[1], value, assignment)

    def team_full_already(self, project, person, team, assignment):
        count = 0
        for i in range(person):
            if assignment[(project, i)] == team:
                count += 1
        return count == team_size(self.n, self.k, team)

    def team_can_be_joined(self, project, person, team, assignment):
        conflicts = {i for i in range(person) if assignment[(project, i)] ==
team}
        for i in range(project):
            for j in range(self.n):
                if j == person:
                    continue
                if assignment[(i, j)] == assignment[(i, person)] and j in
conflicts:
                    return False
        return True
```

As a result, this algorithm can solve this problem much faster than the tree searches. It can solve **(20, 4, 4)** in **4.1** seconds. However, this is still highly inefficient. It cannot solve **(20, 4, 5)** in a reasonable amount of time. This is because every time the constraint function is called, it has to recompute team size and conflicts. It is hard to avoid this because we are relying on a general backtracking algorithm instead of writing one just for this problem. In the following example I wrote my own separate backtracking algorithm which is much more efficient. The `solve()` method does the backtracking:

```python
import math
import random
import time


def solve(arrangements, conflicts, n, k, p, person, project_num):
    if project_num == p:
        return True
    if person == n:
        new_conflicts = update_conflicts(arrangements[project_num], conflicts, n)
        return solve(arrangements, new_conflicts, n, k, p, 0, project_num + 1)

    team_num = 0
    while team_num < len(arrangements[project_num]):
        team = arrangements[project_num][team_num]
        if not member_conflicts(arrangements[project_num], conflicts, n, k, team_num, person):
            team.add(person)
            if solve(arrangements, conflicts, n, k, p, person + 1, project_num):
                return True
            team.remove(person)
            if len(team) == 0:
                if team_size(n, k, team_num) == k:
                    team_num = num_k_sized_teams(n, k) - 1
                else:
                    return False
        team_num += 1

    return False



def new_arrangement(n, k):
    return array_of_sets(math.ceil(n / k))

def array_of_sets(n):
    return [set() for i in range(n)]

def update_conflicts(arrangement, old_conflicts, n):
    new_conflicts = [c.copy() for c in old_conflicts]
    for team in arrangement:
        for member1 in team:
            for member2 in team:
                new_conflicts[member1].add(member2)
    return new_conflicts

def member_conflicts(arrangement, conflicts, n, k, team_num, person):
    if len(arrangement[team_num]) == team_size(n, k, team_num):
        return True
    for person2 in arrangement[team_num]:
        if person2 in conflicts[person]:
            return True
    return False

def team_size(n, k, team_num):
    return k if team_num < num_k_sized_teams(n, k) else k-1

def num_k_sized_teams(n, k):
    return n//k - (0 if n%k == 0 else k - n%k - 1)

def valid(n, k, p):
    return num_k_sized_teams(n, k) >= 0 and p <= (n-1) // (k-1) and (p == 1 or math.ceil(n / k) - 1 >= k - 1)
```

```python
n = 20
k = 4
p = 5

arrangements = [new_arrangement(n, k) for i in range(p)]

if valid(n, k, p):
   start = time.time()
   solve(arrangements, array_of_sets(n), n, k, p, 0, 0)
   print("time elapsed: " + str(time.time() - start))
   print(arrangements)
else:
   print("infeasble (n, k, p) values")
```

This code is able to directly work with a structure that better represents the situation, rather than encoding it into a list of variables.

It no longer requires counting team size, because len() keeps track, and it no longer requires checking conflicts every time we assign a value to a variable because we keep track with a conflicts parameter.

Finally, we introduce pruning by forcing teams to be sorted by first team member. This means we can sooner find out if we've hit a dead end, because if we remove someone from a team and it results by being empty, then that means that trying the next team would result in unsorted teams in the end (and order doesn't matter, so we must have exhausted our options) so we backtrack.

The result is that it can solve **(20, 4, 4)** in **0.3** seconds instead of 4.1. And it *can* solve **(20, 4, 5)** in a reasonable time: **0.33** seconds. In fact, it can solve **(40, 4, 5)** in **2.23** seconds.

This is why I believe for this problem, it would be better to write the backtracking algorithm yourself rather than using someone else's backtracking code.