**CCT College Dublin**

**Assessment Cover Page**

|  |  |
| --- | --- |
| **Module Title:** | Artificial Intelligence  Data Visualisation & Communication |
| **Assessment Title:** | L8\_CA\_2\_AI\_DV v5 |
| **Lecturer Name:** | David McQuaid  Sam Weiss |
| **Student Full Name:** | Rayen Bentemessek |
| **Student Number:** | 2021378 |
| **Assessment Due Date:** | 29/12/2024 |
| **Date of Submission:** |  |

**Declaration**

|  |
| --- |
| By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution. |

Table of Contents

[Part 1: 3](#_Toc185513045)

[Scenario 1: 3](#_Toc185513046)

[Scenario 2: 4](#_Toc185513047)

[Part 2: 5](#_Toc185513048)

[How using Constraint satisfaction finds an answer to the problem: 5](#_Toc185513049)

[Difference between CSP and standard algorithms in finding solutions: 6](#_Toc185513050)

[Part 3: 7](#_Toc185513051)

[References: 12](#_Toc185513052)

# Part 1:

## Scenario 1:

In order to solve this problem, I made sure to choose the roles as my variables, then the combination of teams as my domains. This approach was helpful to avoid redundant solutions and get only unique ones and combinations of employees. Because the computer will count “Peter, Juan” and “Juan, Peter” as 2 separate solutions while it’s not as we only need 2 people for AI in scenario 1 for example. That is why we only get [“Peter”, “Juan”] as the only solution that can have these 2 persons.

Then I’ve created a function called “constraints” that will ensure I only assign 2 roles as a maximum per person because it’s a hard constraint. I did it through a dictionary called “employee\_counts” that will keep iterating for roles, adding employee counts and return true if maximum value of the dictionary is less than or equal to 2. Within the same function, I have ensured we only hire 4 people (excluding Ciara) which is our second hard constraint using a set of unique employees. The logic behind this is I treated hiring Ciara to be a soft constraint, because “she knows python “which doesn’t mean she has to be hired and it’s worth exploring solutions where she is not hired because she is the recruiter, and she may change her mind. I only focused on the fact that she has the funds to hire 4 people, and it was doable without hiring her in 2 solutions and she was present in 52 solutions.

A screenshot of a computer

Description automatically generated

To have a better layout, I created two lists, one where Ciara is hired and the other where she is not and added my solutions to the relevant lists, after that I printed my solutions in both cases, and I showed which person is hired for each position.

I have to mention that I have considered solutions where the same people are hired but in different roles to be different because I got different combinations in the roles which I considered to be different because it satisfies all the constraints that we have and it’s unique at the same time which means no other solution will have the same repartition but we may have the same people to hire as shown in the following picture.

A screenshot of a computer

Description automatically generated

## Scenario 2:

For scenario 2, it was pretty similar to scenario 1, because we needed one more AI Engineer and a security employee which is not critical (soft constraint) so I have added more combinations in my AI variable, and added a security variable with the possible employees (Mary and Maria) and an empty list [] which represents the option of not hiring security. I’ve also thought that Ciara knowing python to be in scenario 2 and I hired Juan at first ( which made me got 1014 solutions ) but then I changed it and realised that the 2 scenarios are independent, and Juan becoming a partner means he is no longer a candidate to consider so I’ve removed Juan from the possible employees and also removed Ciara from the Python variable. This resulted in 5 solutions, 4 of them not having a security and one where a security employee was hired as shown in the following picture.

A screenshot of a computer

Description automatically generated

# Part 2:

## How using Constraint satisfaction finds an answer to the problem:

The constraint satisfaction problem is basically a mathematical problem where it requires certain solution for certain variables, this solution must meet all the constraints. To do that we need first to start by defining:

**Variables**: These are the objects that need values assigned to them to satisfy the constraints. It can be using any type of variables such as Boolean, Integer, Strings etc. And they are finite as well (V1, V2…Vn). To be more specific, in our case where we need to assign employees to certain roles, I chose to define my variables as the roles: Python, AI, Web etc. It’s important to mention that this is not the only way to do that and there is not a “correct” way to do it as well, but whatever the choice of variables is, the domains must follow it (GeeksforGeeks, 2023).

**Domains**: The domains are basically the range of values that the variables can get and they cannot be empty because each variable needs one value at least, for example in my case Since one of the variables is Python, the values are the possible employees that can get the roles, like Ciara, Jane, Peter and Bruce. In other words, I’m telling my CSP that for Python you can only assign 3 of the 4 employees, I also chose to that in terms of combinations to avoid potential redundancy, for example: Ciara, Peter is the same as Peter, Ciara but the algorithm can’t tell so I forced certain combinations as shown in the picture below to avoid that potential issue (GeeksforGeeks, 2023).

**Constraints:** Constraints are basically the limitations or the conditions that give me meaning to the previous variables and domains. In our case, in scenario 1 for example, we can only hire a maximum of 4 people, and each employee can take a maximum of 2 roles, those are constraints that must be met to solve the problem, they’re also called hard constraints. There are also soft constraints where they’re necessary but it’s good to have which is for example, ‘Ciara knows how to code’ doesn’t mean she has to be part of the team or hiring a security employee in scenario 2.

Now after identifying the components of the CSP, we’re getting one step closer to highlighting how it works, so after we defined the variables, domains and constraints here comes the beauty of CSP as everything is done in the background using these techniques or algorithms:

* **Backtracking Algorithm:**

This algorithm plays a major role in the efficiency of the CSP and how fast it returns a solution, because as soon it faces an issue or solution that doesn’t work, it discards it and goes back or backtracks to the previous path. This is done for all the possible extensions of the solution and then it keeps going, if the solution meets the requirements, it is valid, if it does not, we don’t have to continue and we go back instead (GeeksforGeeks, n.d.).

Now let’s see how see this works in our case:

Say for the Python team we chose Ciara, Peter, Jane, we move to the AI team, and we choose Juan, Jim. remember that we already chose 4 people, so if we add Mary as a web developer, one of the constraints is violated so here the program backtracks and chooses Anita, but we have the same issue, so we must choose Juan as a web developer. Then for the database, we choose Jim, when we move to the Systems, if we choose Juan, we have another issue because Juan is hired twice, same thing for Jim. The backtracking algorithm will flag that this solution is incorrect, and the solution will be marked as invalid.

* **Forward-checking algorithm:**

This algorithm is basically an optimisation of the backtracking algorithm, as this algorithm will keep track of assigned values and dynamically remove them (temporarily ) from the domain list for the specific variable, in other words, in scenario 1, if Juan has already achieved his limit of 2 roles, we don’t assign and then backtrack like we do in backtracking, but we Just remove Juan from the possible role leaving less options but with less issues, in case no domains are left that means the solution is invalid and the algorithm backtracks (Cuni.cz, 2024).

* **Constraint propagation algorithms:**

Constraint propagation is a major concept in CSP, as it works by narrowing down the domains of variables based on constraints. This approach will help in reducing the search space as it ensures consistency across the different components, nodes consistency for example. So, it will eliminate the values that violate the constraints before doing the search. In our example it will initialise the domains for each role and constraints are applied simultaneously and iteratively to eliminate possible confusions, for example hiring someone for 3 roles which cannot be, and it keeps going until the solutions are found (GeeksforGeeks, 2024).

## Difference between CSP and standard algorithms in finding solutions:

In this discussion, we will talk about the differences between CSP and standard algorithms in solving problems, the main question is not can they do it? Because they both can. But in order for a CSP to find a solution, we need to define variables domains and constraints, then the algorithm will do the rest using -not limited to- the previous techniques and algorithms which will eliminate the invalid paths as soon as they’re faced which will save tons of processing power and time as well. However, the standard algorithm will require defining all the variables on their own, than implement tailored solution for the problem which can be good for some problems, but if it’s not implemented the right way, some disasters can happen. Speaking from a personal perspective where the running time for one cell took 4353 seconds to run as shown in the picture below and the bigger the problem, the less efficient it becomes. So, on one hand, CSP will guarantee reliable solutions if it’s implemented the right way with great efficiency which will avoid invalid solutions early, on the other hand, standard algorithms give more freedom in the implementation, but it may struggle in problems as it will explore the invalid paths unnecessarily which will make it slower. So CSP works much better in these kinds of situations the same way standard algorithms are amazing in most of the application cases but here it’s like comparing a Porsche 911 to a Land Rover Defender, they’re both amazing but one is a sports car and the other is an off-road SUV.

A screenshot of a computer

Description automatically generated

# Part 3:

In order to implement this algorithm, I started with A\* but the results were horrible, and I got too many solutions that I didn’t need, also the program was so heavy as shown in the previous picture and took ages to run and I found some issues defining the heuristics. So, I decided to switch to Depth First search, with its nature of exploring all possible combinations it was so helpful solving my problem. DFS was much more efficient in terms of memory usage and I found no issues with it while implementing because the time complexity is dependent on the roles and the possible combinations which was manageable because we only needed 6 roles maximum with limited predefined combinations which contributed in optimising its time efficiency.

I created 2 dictionaries for each scenario with the key being the role and the value as the possible combination. Then I implemented two functions, one for each scenario with a very similar approach to the CSP constraints functions to make sure no one is hired more than 2 times and respect the number of employees based on the scenario.

Then I implemented my DFS function, which will take roles from the combinations dictionary mentioned earlier, I’ve also iterated through a stack that starts with an initial state which is a tuple containing “index”, “current\_solution” and “employee\_counts” and then we use LIFO to work with the current state. Then we just start assigning the roles from the dictionaries with the combinations of employee making sure to get their employee\_counts and creating a new solution based on the combinations. Once we’re done, we create “next\_state” tuple with the updated components and we add it to the stack, if our index meets the length of our roles, we check it with the scenario function, if it is satisfied, we got a new solution. If not, we just keep iterating. Then we return the found solutions.

Everything here went according to our plan because we found the same number of solutions using DFS as we did with the CSP framework.

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

# References:

GeeksforGeeks. (2023). *Constraint Satisfaction Problems (CSP) in Artificial Intelligence*. [online] Available at: <https://www.geeksforgeeks.org/constraint-satisfaction-problems-csp-in-artificial-intelligence/>.

GeeksforGeeks. (n.d.). *Backtracking Algorithms*. [online] Available at: <https://www.geeksforgeeks.org/backtracking-algorithms/>.

Cuni.cz. (2024). *Constraint Guide - Constraint Propagation*. [online] Available at: https://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html#FC [Accessed 13 Dec. 2024].

GeeksforGeeks (2024). *Constraint Propagation in AI*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/constraint-propagation-in-ai/ [Accessed 16 Dec. 2024].

‌

‌

‌

‌