Generating Top K Solutions for Combinatorial Puzzles using Weighted CSPs

Muhammad Oneeb Ul Haq Khan
Viterbi School of Engineering, University of Southern California, Los Angeles, CA 90007, USA
mkhan250@usc.edu

Abstract

Weighted Constraint Satisfaction Problems (WCSPs) have been proven to be powerful algorithmic tools to model and then solve combinatorial optimization problems. Subsequent work on WCSPs has paved the way for generating the top K solutions to a combinatorial problem instead of a singular, optimal solution. The applications for such an algorithm are numerous, particularly any arena where "human-in-the-loop AI" is a requirement, i.e. any problem instance in which all problem parameters are not known at the beginning or there is insufficient information to model the problem completely. In this algorithmic approach the user is provided with a k number of solutions along with the solution costs, which may be interpreted as likelihoods. The user may then provide feedback as to which solution is preferred allowing the model to adjust costs and confidence values until the correct solution is the top solution. This is critical in allowing AI algorithmic techniques to penetrate industries and sectors where human discretion is an absolute necessity, e.g. courtroom settings, policy decisions and implementation, and military strategizing and planning, among many others. In this article, we are going to work on using this framework to solving combinatorial puzzles, using *The Zebra Problem* as a proof-of-concept. The core problem we will be exploring will be how to model this puzzle as a WCSP.

Introduction

The Weighted Constraint Satisfaction Problem is a combinatorial optimization problem, a variant of the Constraint Satisfaction Problems, in which the constraints for each assignment are not Boolean but rather have a cost or weight. What this means is that for a subset of variables or each tuple in a constraint there is a non-negative value or cost assignment.

WCSP Background

Mathematically, a WCSP is represented by a triplet $\mathcal{B} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X} = \{X_1, \cdots X_N\}$ is a set of N variables with a pre-specified domain size each $\mathcal{D} = \{D_1, \cdots D_N\}$. $\mathcal{C} = \{C_1, \cdots C_M\}$ is a set of M weighted constraints, discussed previously. For each weighted constraint $C_i \in \mathcal{C}$ defined on a subset of variables $\mathcal{S}_i \subseteq \mathcal{X}$, each possible combination of values to variables in $\mathcal{S}_i$ has an associated non-negative weight (cost). Our objective is to determine an assignment to all variables in $\mathcal{X}$ from their respective domains such that the sum of the weights identified by each constraint in $\mathcal{C}$ is minimized. Our objective function can therefore be represented by $\arg\min_{a \in \mathcal{A}(\mathcal{X})} \sum_{C_i \in \mathcal{C}} E_{C_i}(a|C_i)$, where $\mathcal{A}(\mathcal{X})$ represents the set of all complete assignments $|D_1| \times \cdots |D_N|$. $a|C_i$ represents the projection of a complete assignment a onto the subset of variables in C_i. E_{C_i} is a function that maps each assignment $a|C_i$ to its respective weight.

WCSPs are NP-Hard problems, however we have many tools available to us that allow us to reach the solution. For Boolean WCSPs (which are also NP Hard), we can use existing combinatorial problem solving approaches to determine the optimal solution. Each separate constraint table is first converted into *Lifted Graphical Representation*, i.e. the individual constraint tables are represented using graphs, where the nodes represent a boolean assignment and have the corresponding weight associated with them. The next step is to obtain a *Composite Constraint Graph* (CCG) by combining all the graphs from the previous step. Solving the CCG as a Minimum Weighted Vertex Cover (MWVC) problem provides us with the encoded solution for the WCSP. In the case of Non-Boolean WCSPs, such as in our case, we have powerful mathematical solvers which obtain the solution by one of several ways, e.g. Quadratization and Integer Linear Programming.

Generating Top K Solutions

Generating the top k solutions of a combinatorial problem is a generally understudied topic as our approaches to solving them usually look for the optimal solution. However, obtaining the top k solutions for any such problem has a lot of practical benefits. Particularly, generating top k solutions is ideal for any scenario in which we require the implementation of *human-in-the-loop AI* . By providing the top k solutions, we can incorporate user/expert feedback in order to iteratively improve our model until the desired results are achieved and the preferred solution is the top solution. This is especially useful when we have insufficient knowledge to model our problem completely or temporally dynamic problems where problem states and parameters are constantly changing.

WCSPs are a great tool that enable us to do exactly this with great ease. To generate the kth solution, we introduce a prohibitive, global constraint in the kth iteration of the algorithm such that the previous *k-1* solutions are not consid-

ered. Naturally, the top solution returned in this iteration, in which the solution space does not include the previous *k-1* solutions, is the top kth solution.

WCSP Solver using Gurobi

At this junction, we already have a script available to us that solves WCSPs using the python package for the Gurobi Optimizer (a powerful mathematical solver), which further employs an ILP based approach. This solver also has the capacity to generate the top k solutions for any given WCSP by introducing prohibitive, global constraints which are represented using linear inequality constraints. Our solver however can only calculate a solution for a problem with either unary constraint tables or binary constraint tables or both, i.e. ternary or higher degree constraint tables cannot be solved by the solver.

The solver requires the number of variables, the domain size of each variable and the constraint tables as input. The solver then provides the cost, solution and processing time for all top k solutions of the WCSP. Increasingly complex WCSPs will have increasingly longer processing times. The solution provided is an assignment of a value to all the variables in the problem such that the sum of all costs is minimized. The top solution will have the minimum cost, with the cost increasing with the index of the kth solution.

The Zebra Problem

For this report, we will be working on representing *The Zebra Problem* as a WCSP and then using the WCSP Solver to obtain the solutions as a proof-of-concept for utilizing this framework to solving combinatorial puzzles in general.

The Zebra Problem has varying origin stories, attributed to having been concocted by many famous logicians. Interestingly, it is also referred to as *Einstein's Logic Problem*. The problem also has many versions with differing variables and variable assignments. In any case, the problem always sticks to the same format. There are N variables with J possible values each. The problem comes with a list of *hard* constraints, and there is only one possible, unique solution. Already, this problem seems very reminiscent of the characteristics of a WCSP, as discussed earlier. An assignment each of a variable is grouped with an assignment each of all other variables.

In our version of *The Zebra Problem*, the problem statements are as follows: There are five houses. Each house is colored one of five possible colors. In each of the five houses resides a person of one of five possible nationalities. The house resident has one of five possible favorite drinks, owns one of five possible pet animals and plays one of five possible games.

- Nationalities: [Englishman, Spaniard, Irishman, Nigerian, Japanese]
- Color: [Red, Green, Ivory, Yellow, Blue]
- Drinks: [Coffee, Tea, Milk, Orange Juice, Guinness]
- Pets: [Dog, Snails, Fox, Horse, Zebra]
- Games: [Go, Cricket, Judo, Poker, Polo]

The question that gives our problem its name is: *Who owns the Zebra?* The answer to the problem posed by this question can be arrived at by coming to a conclusion regarding the assignment of each value of each variable above to every one of the five houses, i.e. by knowing which color, nationality, pet, drink and sport belong to which house we can confidently and easily answer who owns the Zebra.

The problem is a seemingly simple one. However, as we will see it is deceivingly complex. Each variable has $5! = 120$ possible assignments. Seeing as that we have 5 variables, the size of our solution space is:

$$(5!)^5 = (120)^5 = 24,883,200,000$$

The problem comes with the following list of constraints:

- The Englishman lives in the red house.
- The Spaniard owns a dog.
- The man in the green house drinks coffee.
- The Irishman drinks tea.
- The green house is to the right of the ivory house.
- The Go player owns snails.
- The man in the yellow house plays cricket.
- The guy in the house in the middle drinks milk.
- The Nigerian lives in the first house.
- The judo player lives next to the man who has a fox.
- The cricketer lives next to the man who has a horse.
- The poker player drinks orange juice.
- The Japanese plays polo.
- The Nigerian lives next to the blue house.

Enforcing all of these constraints, limits our possible solutions from approximately 25 billion to one. Therefore, an exhaustive, brute force approach where we search all possible solutions until all our listed constraints are satisfied will already be an expensive operation to undertake. If we extend the problem size by a single variable the solution space grows too much to allow this to be a feasible approach. This is where WCSPs come in. We know that the WCSP framework is a very powerful algorithmic framework and therefore we want to leverage its strengths to solve our combinatorial puzzle. The only question that remains to be answered is: *How do we model our puzzle as a WCSP?*

We have already observed that there are some fundamental similarities between how this problem is formatted and the basic format of a WCSP, so the only issue we need to address is converting the list of constraints above into a format that is interpreted by our WCSP Solver.

Methodology

In this section, we are going to discuss how to model our combinatorial puzzle and it's constraints as a WCSP so that our WCSP Solver can solve it and provide us with the solution, as well as some of the challenges faced along the way and how they are overcome.

Modeling the problem

The first task we have is to introduce a sixth variable *Position* to our problem, the value for which informs the relative position of the house, starting from the left. We then represent the variables [Nationality, Game, Drink, Pet, Color, Position] numerically assigning values 0 to 5, respectively. We then provide an index number, 0 to 4, to each possible value of a variable.

We will now create Joint Distribution (JD) tables for every possible variable combinations. Since our solver can only use binary constraint tables at maximum, each of our Joint Distribution table will comprise of two variables. We will then construct

$$C_2^6 = 15$$

JD tables. All JD tables will be of size: 5×5. Each cell will represent the probability of the variable assignment represented by the cell's row index and the variable assignment represented by the cell's column index. All probability values will be initialized to

$$probability = 1/D_i = 1/5 = 0.2$$

.

Table 1: Shows the JD table with the initialized probability values for each Nationality and the corresponding Color of the house they reside in.

Nat. \ *Color*	Red	Green	Ivory	Yellow	Blue
Englishman	0.2	0.2	0.2	0.2	0.2
Spaniard	0.2	0.2	0.2	0.2	0.2
Irishman	0.2	0.2	0.2	0.2	0.2
Nigerian	0.2	0.2	0.2	0.2	0.2
Japanese	0.2	0.2	0.2	0.2	0.2

For the pairings we are certain should be in our solution we assign a high probability value, and for pairings we are certain should not be in our solution we assign a very low probability value.

$$p = 0.99, n = 0.001$$

such that p is a pairing in our solution and n is a pairing not in our solution.

We go through each constraint provided to us in the problem statement and update the corresponding JD tables, such that the probability values now reflect the likelihood of each pairing after the constraint is enforced. Using the first constraint as an example, *The Englishman lives in the Red house* the updated JD table is shown in $Table2$.

The values represented within the JD tables are not the weights for the respective assignments. Therefore, once we map all constraints in the associated JD tables, we calculate the weights of each assignment combination using

$$w = -ln(probability)$$

Thus, the higher the probability value for a combination, the lower the weight value that will be assigned to it. Since our solver will always try to minimize the total cost, the solution will include the combination with the lower weight value. The logarithmic operation on the probability value means that the total sum of costs of the solution will be equivalent to the product of probabilities of the combinations present in the solution. Therefore, the cost of the solution will allow us to know the likelihood of that solution.

Table 2: Shows the same JD Table as in $Table1$ with the updated probability values for the constraint.

Nat. \ *Color*	**Red**	Green	Ivory	Yellow	Blue
Englishman	**0.99**	0.001	0.001	0.001	0.001
Spaniard	0.001	0.25	0.25	0.25	0.25
Irishman	0.001	0.25	0.25	0.25	0.25
Nigerian	0.001	0.25	0.25	0.25	0.25
Japanese	0.001	0.25	0.25	0.25	0.25

Table 3: Shows the Weighted Constraint Table for Nationality and Color

Nat. \ *Color*	**Red**	Green	Ivory	Yellow	Blue
Englishman	**0.01**	6.91	6.91	6.91	6.91
Spaniard	6.91	1.39	1.39	1.39	1.39
Irishman	6.91	1.39	1.39	1.39	1.39
Nigerian	6.91	1.39	1.39	1.39	1.39
Japanese	6.91	1.39	1.39	1.39	1.39

From $Table3$ we can see that higher probability values are shown as lower weight values, however in a problem involving many constraints, we want there to be a more drastic difference in weight values for combinations that should be in our solutions and combinations that should not be in our solution. Therefore, we modify our weight calculation slightly:

$$w = 10^{-ln(probability)}$$

Table 4: Shows the updated Weighted Constraint Table

Nat. \ *Color*	**Red**	Green	Ivory	Yellow	Blue
Englishman	**1.02**	8.1E+6	8.1E+6	8.1E+6	8.1E+6
Spaniard	8.1E+6	24.33	24.33	24.33	24.33
Irishman	8.1E+6	24.33	24.33	24.33	24.33
Nigerian	8.1E+6	24.33	24.33	24.33	24.33
Japanese	8.1E+6	24.33	24.33	24.33	24.33

The problem is now modeled so that it can be interpreted by the solver and provide the optimal solution. However, the solution provided will give us a single assignment per variable, whereas what we actually require is a solution providing the assignment of all variable values to each house, and since each variable assignment and grouping informs

all other variable assignments the partial answer is not the correct answer. Therefore, we must introduce four more instances of each variable so that we may have the complete solution. Thus instead of the original 6, we now have $6 \times 5 = 30$ variables. This also means that the number of joint distribution and weighted constraint tables has increased to

$$C_2^{30} = 435$$

Since there are several instances of each variable, we have to ensure that the assignment is not repeated twice, therefore in JD tables for two variables of the same kind, we assign an extremely low probability to the same assignments, i.e. an extremely high weight value.

Table 5: Shows the JD Table for different instances of the same variable: Color

	Red	Green	Ivory	Yellow	Blue
Red	**0.0001**	0.25	0.25	0.25	0.25
Green	0.25	**0.0001**	0.25	0.25	0.25
Ivory	0.25	0.25	**0.0001**	0.25	0.25
Yellow	0.25	0.25	0.25	**0.0001**	0.25
Blue	0.25	0.25	0.25	0.25	**0.0001**

Table 6: Shows the Weighted Constraint Table for different instances of the same variable: Color

	Red	Green	Ivory	Yellow	Blue
Red	**1.6E+10**	24.3	24.3	24.3	24.3
Green	24.3	**1.6E+10**	24.3	24.3	24.3
Ivory	24.3	24.3	**1.6E+10**	24.3	24.3
Yellow	24.3	24.3	24.3	**1.6E+10**	24.3
Blue	24.3	24.3	24.3	24.3	**1.6E+10**

The modeling of the problem is complete with some limitations which will be discussed in the next sections. Our problem and constraints are now encapsulated by our constraint tables. We now call our WCSP Solver on the problem - which then provides us with the solution.

Results

The solver generated results are shown in $Table7$ while the actual solution to our Zebra Problem are shown in $Table8$

Table 7: Shows the solver generated results for The Zebra Problem

1	*Englishman*	*Poker*	*Tea*	*Fox*	*Yellow*
2	*Irishman*	*Go*	*Milk*	*Snails*	*Red*
3	*Japanese*	*Judo*	*OJ*	*Horse*	*Ivory*
4	*Nigerian*	*Cricket*	*Coffee*	*Zebra*	*Green*
5	*Spaniard*	*Polo*	*Guinn.*	*Dog*	*Blue*

Table 8: Shows the actual solution for The Zebra Problem

1	*Nigerian*	*Cricket*	*Guinn.*	*Fox*	*Yellow*
2	*Irishman*	*Judo*	*Tea*	*Horse*	*Blue*
3	*Englishman*	*Go*	*Milk*	*Snails*	*Red*
4	*Spaniard*	*Poker*	*OJ*	*Dog*	*Ivory*
5	*Japanese*	*Polo*	*Coffee*	*Zebra*	*Green*

As can be observed, the model result does not match the actual solution for the problem. There are some limitations in our modeling which prevent our script from generating the correct solution

Limitations

Relative Constraints From the list of constraints in the problem statements, the ones highlighted in red are the constraints which define their position relative to another variable's position.

- The Englishman lives in the red house.
- The Spaniard owns a dog.
- The man in the green house drinks coffee.
- The Irishman drinks tea.
- The green house is to the right of the ivory house.
- The Go player owns snails.
- The man in the yellow house plays cricket.
- The guy in the house in the middle drinks milk.
- The Nigerian lives in the first house.
- The judo player lives next to the man who has a fox.
- The cricketer lives next to the man who has a horse.
- The poker player drinks orange juice.
- The Japanese plays polo.
- The Nigerian lives next to the blue house.

Compared to the other *IS-A* constraints which directly map the probability of a variable assignment occurring with another, these constraints only partially inform the solution. For our consideration, let us look at the constraint: *The green house is to the right of the ivory house.*

Table 9: Shows the JD Table for Color and Position

	1	2	3	4	5
Red	-	-	-	-	-
Green	**0.001**	0.25	0.25	0.25	0.25
Ivory	0.25	0.25	0.25	0.25	**0.001**
Yellow	-	-	-	-	-
Blue	-	-	-	-	-

From this constraint we know that the Green house will not be the left most house, and the Ivory house will not be the right most house and therefore the probability that either of them are in any of the four remaining houses is equal. Hence we have improved our probabilities and encouraged our model in a particular direction, but we are unsuccessful in ensuring that the Green house is to the right of the Ivory house.

Problem Size The WCSP model for this combinatorial puzzle involves too many constraint tables resulting in a very high processing time. For the given solution, the run time was $10180 seconds$, i.e. approximately 3 hours. If we were to increase the number of variables the problem size and constraints involved our run time will only drastically increase. For a system that could benefit from user feedback, the run time is too large to make *human-in-the-loop AI* feasible.

Top K Solutions and Human-In-The-Loop AI

We can use the existing WCSP solver to generate the top k solutions to our Zebra problem. The cost for each solution can be used to obtain the likelihood of the solution. From our results:

$$w = 1527197$$

$$probability = e^{-log(w)} = e^{-log(1527197)}$$

$$probability = 0.002$$

The top solution will have the minimum weight and therefore the maximum probability. The kth top solution will have the kth minimum weight and hence the kth maximum probability. This is particularly useful if a subset of the list of constraints provided to us in the problem statement are available to us. In such a scenario our solution space increases from one unique to several possible solutions with varying degrees of likelihood. We can then use our WCSP results to select the solution that best fits our requirements.

As observed, at the instance of modeling the problem, we had insufficient data to model the problem exactly. This is where *human-in-the-loop AI* allows the user to provide feedback and improve upon the provided solution by adjusting probability and weight values. The feedback cycle continues until the top solution is the correct/preferred solution.

Conclusion and Future Work

In this report we discussed the idea of using WCSPs to model combinatorial puzzles, such as *The Zebra Problem*, and then generate the top k solutions using an existing WCSP solver. We ran into some minor roadblocks that prevented us from obtaining an accurate solution using our current model. Future work on this involves considering the addition of auxiliary variables that allow us to circumvent the problem of **relative constraints** and ensuring that we can accurately enforce relative positions according to the constraint requirements. Future work also involves working on modeling approaches involving fewer constraint tables so that our run time is reduced. In terms of the script, we used *The Zebra Problem* as a proof-of-concept, once we are able to successfully achieve the desired results, we should generalize our front-end solver to be utilized for a wider array of combinatorial problems.

References

[1] K. Sun, K. Maddali, S. Salian, and T. K. S. Kumar, "Top K Hypotheses Selection on a Knowledge Graph," in *Proceedings of the Thirty-Second International FLAIRS Conference*, 2019

[2] A. Li, Y. Guan, S. Koenig, S. Haas, T.K. S. Kumar, "Generating the Top K Solutions to Weighted CSPs: A Comparison of Different Approaches"