CS 6810 Project Report: on CSPs

Group members: Ellie Fassman, Jiho Cha, Yunya Zhao

December 16, 2023

1 Introduction

CSPs, or Constraint Satisfaction Problems, are an archetype of problems where we we wish to assign variables values from a finite set (called a domain) subject to a set of constraints on those variables. The objective of these problems is to maximize the total number of constraints satisfied. Each constraint locally checks the assignment of a subset of variables to see if it is satisfied or not.

The main challenge associated with CSPs is developing algorithms that can efficiently determine the maximum fraction of constraints that can be satisfied over all possible assignments of variables, subject to certain resources such as time, space, etc. The availability of these resources, along with the nature of the constraints present in the problem, affect how close we can get to the optimal number of constraints.

Example 1.1. *Formulation of Sudoku as a CSP (informal)*

Sudoku is a type of puzzle seen all throughout pop culture, and is perhaps the epitome of constraint satisfaction problems. There are 81 tiles on a 9 by 9 grid that must be labelled with a number from 1 through 9, with the constraint that no row, column, or 3 by 3 box contains no duplicate numbers.

Of course, this is a rather toy example in the grand scope of CSPs. In general, the power of CSPs can capture a host of many natural and well-studied optimization problems, such as SAT (and many of its variants), Max-Cut, Max q-colorability, etc. On the other hand, many other optimization problems can not be formulated by CSPs, such as maximum flow in a network. One might therefore ask what the value is in studying CSPs, if they only appear to capture a subset of the many natural optimization problems that computer scientists study. One answer to this question (and the answer we will be focusing on most in this report) is the fact that constraint satisfaction problems have an inherent structure associated to them, which enables clear boundaries between different classes of problems within the realm of CSPs.

As a motivating example, consider the existence of NP-intermediate languages. These languages blur the line between P and NP, but the question as to whether or not any "natural" problems fall into this class is still open. As we will discuss later in this report, the rigid configuration underlying CSPs do not permit such intermediate languages, creating an indisputable dichotomy between P and NP. The power of CSPs also extends past the well-known P vs. NP question, as it also gives rise to many other dichotomies. CSPs are a tool that allow computer scientists to study many open questions in complexity theory through the lens of a large set of structured and "natural" problems.

2 Notation and Definitions

We will now formally define CSPs. Note that the following definition is just one formulation of CSPs. Because of the general structure of CSPs, there is not one canonical definition for CSPs, and many sources use different notation and mathematical objects to express them. However, we will do our best in this report to standardize our notation to the definition provided below.

Definition 2.1. *Formally, a CSP is a tuple (V, Ω, C), where:*

- *V is the set of variables*
- *Ω, the domain, is the set of values that we can assign to $v \in V$*
  - *Typically Ω is the set $\mathbb{Z}_q = \{0, 1, ..., q-1\}$, and we are often in the case where $q = 2$*
  - *In general, Ω must be a finite set.*
- *A constraint $c \in C$ is a pair $f : \Omega^r \to \{0, 1\}$ and a tuple $(v_{i_1}, ...v_{i_r})$ of r distinct variables in V.*
  - *A constraint is satisfied if $f(v_{i_1}, ..., v_{i_r}) = 0$, and unsatisfied otherwise.*
  - *We let F be the set of all such available functions "f"s.*
  - *Intuitively, F determines the particular problem (i.e. MAX-CUT or 3-SAT), as it controls what sorts of constraints we are allowed to use, and C determines the instance of that problem.*

Definition 2.2. *We say that $A : V \to \Omega$ is an assignment of variables.*

- *Given an instance C of a CSP, $\mathsf{val}_C(A)$ is the fraction of constraints that A satisfies.*
- *val_C is the maximum value taken by $\mathsf{val}_C(A)$ over all possible assignments.*

Example 2.3. *Formulation of Sudoku as a CSP (formal)*

In this example, we set V to be a set of 81 variables. To better exemplify the connection to Sudoku, we label the variables as $v_{1,1}, v_{1,2}, ..., v_{1,9}, v_{2,1}, ..., v_{9,9}$. Ω is the set $[9]$. The set F contains only one function - $f : \Omega^9 \to \{0, 1\}$ which returns 1 if and only if all 9 of its arguments are different. We then make 27 constraints using the function f:

- $(f, v_{i,1}, ..., v_{i,9})$ $\forall i \in [9]$
- $(f, v_{1,i}, ..., v_{9,i})$ $\forall i \in [9]$
- $(f, v_{3i+1,3j+1}, v_{3i+1,3j+2}, v_{3i+1,3j+3}, v_{3i+2,3j+1}, v_{3i+2,3j+2}, v_{3i+2,3j+3}, v_{3i+3,3j+1}, v_{3i+3,3j+2}, v_{3i+3,3j+3})$ $\forall i, j \in \{0, 1, 2\}$

The first set of 9 constraints formalize the idea that no two numbers on a row share the same value, the second set of 9 constraints formalize the idea that no two numbers on a column share the same value, and the third set of 9 constraints formalize the idea that no two numbers in a 3x3 box share the same value.

Example 2.4. *Formulation of 3-SAT as a CSP*

Consider the following formulation:

- V $= \{x_1, ..., x_n\}$
- $\Omega = \{0, 1\}$
- $F = \{$ternary OR with literals$\}$

Note that there are 8 functions in total in F, since for each of the 3 literals, we can choose to negate it or not. Let $f_1 \in F$ be defined as $f_1(x, y, z) = x \vee \overline{y} \vee z$, and $f_2 \in F$ be defined as $f_2(x, y, z) = \overline{x} \vee \overline{y} \vee z$.

Then the CSP $(V = \{x_1, ..., x_5\}, \Omega, C = \{(f_1, (x_1, x_3, x_4)), (f_2, (x_2, x_4, x_5))\}$ corresponds to the Boolean expression $\Phi = (x_1 \vee \overline{x_3} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_4} \vee x_5)$.

3 Solving CSPs (Traditional Approaches)

Throughout the next sections we will be talking about solving CSPs. By solving CSPs, we mean finding an assignment of variables such that every constraint is satisfied. These traditional approaches below are from [Liu98].

3.1 Generate and test

We can first think about the most basic approach, the generate and test method. This is where each possible assignment of variables is generated and then tested to see if the constraints are satisfied. When trying to find the whole solution set (or in the worst case), the time complexity of this is exponential in the number of variables.

We then can ask if adding randomization helps solve this problem more efficiently. We can create randomized generate and test algorithms that select assignments to test at random (based on some given distribution). This is similar to randomized hill climbing where the distribution can be biased to draw assignments more alike the recently tested assignments. However, randomized approaches cannot prove $S = \emptyset$ since they do not necessarily test all possible assignments.

3.2 Tree search

We can approach solving CSPs using a tree search with backtracking. So in this approach, all variables are instantiated one at a time. Upon instantiating, a value is assigned to the variable (chosen from its domain), then we check that constraint are satisfied (otherwise assign a new value). If no value can be assigned (we exhausted all possible options and every assignment violated a constraint), backtrack. In backtracking, the most recently assigned variable changes assignments (if a new assignment exists). Then the algorithm may continue forward with assignments or it may backtrack further (if no values to newly assign are possible). If/when all variables are assigned a value, then we've found a solution. If all combinations of assignments have been exhausted and the constraints are not satisfied, there's no solution.

3.3 Constraint propagation

The goal of constraint propagation is to reformulate the CSP into an easier, equivalent form. We can do this by minimizing scope of domain for each variable (without eliminating possible solutions). Each of the constraints are updated after the removal of values from the domain (propagation). The goals of constraint propagation are usually either node consistency or arc consistency.

- Node consistency: For all v in the set of variables, throughout v's domain, all univariate constraints involving v are satisfied.
  - If v is not node consistent, there is a value in the domain of v that upon v being assigned this value will result in failure of satisfaction.
    - This value can be removed from v's domain since it will never be found in the solution set.
- Arc consistency: (bivariate constraints on pairs of variables) For all pairs of variables v, v', (v, v') is arc consistent if for all values in the domain of v, say value i, there exists a value in the domain of v', say value i' such that $v = i, v' = i'$ does not violate any bivariate constraints involving v and v'.
  - Note that (v, v') being arc consistent $\neq (v', v)$ being arc consistent
  - Upon discovery of inconsistencies for pair (v, v'), we can delete the domain values i which cause this inconsistency since i will never be in the solution set.

The solution set for N variate CSP will be found by attaining N consistency. However, this is not usually used to find the solutions (computationally expensive, more expensive than backtracking). After finding arc consistency, the way to find the solution set is combining constraint propagation and backtracking.

3.4 Combination

We can solve CSPs using backtracking search and constraint propagation. Backtracking will find a solution if the solution set is nonempty, however this is an inefficient method since we are frequently checking the same constraints with only one variable assignment changed. If there are bivariate constraints, there are many unnecessary re-checks of constraints.Constraint propagation will simplify the CSP, but is not guaranteed to find the solution set or even a solution.

We can embed a constraint propagation algorithm inside a backtracking search algorithm. Within the search tree created by the backtracking algorithm, every visited node has a constraint propagation algorithm run on it. This removes inconsistent values from respective domains while newly instantiating variables. If a variable's domain becomes empty, prune this node from the search tree (detecting "dead ends" early). Common CSP algorithms that employ this combination are forward checking (FC) and maintaining arc consistency (MAC). FC has been traditionally used, but recently there has been evidence that MAC is more efficient for complex CSPs. The differences between the two lie in the amount of constraint propagation carried out. in FC, at each node, partial arc consistency is attained, whereas in MAC, at each node, full arc consistency is attained. There are some key tradeoffs when designing an algorithm to solve a CSP. We must ask how much constraint propagation needs to be carried out at each node? More constraint propagation means smaller tree size, but this is more computationally heavy (may not be necessary). There is a tradeoff between the number of nodes visited in the search tree and the work done at each node.

All of these previous approaches are not necessarily efficient for very large CSPs (which contain a lot of variables). We will see that we can use sketching and streaming algorithms to find approximate solution sets for CSPs. First we will discuss what streaming and sketching algorithms are.

4 Streaming algorithms

Streaming algorithms are designed to process data as it becomes available and cannot be stored in its entirety. Streaming algorithms are useful when the data comes in as a data stream, rather than being fully available before processing. There are two main classes of streaming algorithms: counter-based algorithms and sketch algorithms.

4.1 Counter-based algorithms

The basis of these types of algorithms is to track a subset of items from the input and monitor their counts. Upon a new piece of data arriving, we must decide whether to store it or not. There is also the decision of what count to associate with each piece of data. We can think of a few different problem settings:

4.1.1 Majority problem

We can state a version of the majority problem as such: Given a large data stream of n values in some set alphabet with the guarantee that one value occurs more than $\frac{n}{2}$ times, output this value. We can also think about each piece of data as a vote for one element in the alphabet, and we are trying to output the majority vote. The majority algorithm can be designed as such:

Algorithm 1: Finding the majority element of a data stream. From [MM17].

```
def majority(datastream):
    held = next(datastream)
    counter = 1
    for item in stream:
      if item == held:
        counter += 1
      elif counter == 0:
        held = item
        counter = 1
      else:
        counter -= 1
    return held
```

Although this algorithm is very simple, it's useful to think about its correctness. If m is the majority value that we want to return, since m occurs more than $\frac{n}{2}$ times, each decrement pairs up two different items and cancels them out. Then because m will appear more than $\frac{n}{2}$ times not all of its occurrences can be canceled out. The algorithm also only passes through the data once. If the alphabet size is s and we say the size of the data stream is n pieces of data, we see that this algorithm uses $O(log(n) + log(s))$ space.

4.1.2 Frequency problem

We can state the frequency problem as such: find all values in a sequence of data whose frequency exceeds $\frac{1}{k}$ of the total amount of data values. The frequency problem generalizes the majority problem and we can modify the majority algorithm to solve the frequency problem. For each new data value we increment the counter if the value is already stored. If $< k$ values stored, then store the new value with the counter set to 1. Otherwise we decrement all of the counters and if any counter hits 0, delete the value that corresponds with this counter. The frequency algorithm can

be designed as such:

Algorithm 2: Finding the frequent elements of a data stream. From [CH09].

```
def frequency(k, datastream):
    counter = 1
    T = NULL
    while(datastream != NULL):
      val = next(datastream)
      if val ∈ T:
        counter_val += 1
      elif |T| < k − 1:
        T = T ∪ val
        counter_val = 1
      else:
        for all val' ∈ T:
          counter_val' -= 1
          if counter_val' == 0:
            T = T\val'
    return T
```

4.1.3 Other counting problems

We can design algorithms for other various data streaming problems such as:

- Lossy Counting Algorithm: find the values in a data stream that exceed a frequency given by a parameterized threshold.
- SpaceSaving Algorithm: very similar to the majority problem, but finding all values that occur over $\frac{n}{2}$ times in the data stream. This algorithm requires less resources than the majority algorithm described above.

4.1.4 Summary

As a quick aside, there can be arrival-only and arrival-departure data streams.

- Arrival-only streams: data values arrive sequentially. This can model, for example, packets on a network.
- Arrival-departure streams: data values can both arrive and depart sequentially. That is, data values can be nullified later in the stream. This can model fluctuating quantities.
  - For example: (a, 4), (b,1), (a, -1) means the final state is (a, 3), (b, 1).

Counter algorithms are very efficient for arrival-only data streams. They use $O(\frac{1}{\epsilon})$ space and ensure $\epsilon * n$ accuracy (an ϵ-approximation).

4.2 Sketch algorithms

Sketching algorithms are a class of algorithms that operate on a large data set which has a much smaller summary. Sketching algorithms are powerful in that they compress data in order to answer queries efficiently without using a lot of space. That is, for an input of n data items, each data item

is mapped by hash functions into a much smaller sketch vector that records summary information like frequency. The sketch vector does not actually store data items, but rather summary statistics (like frequency distribution, range, and inner product).

4.2.1 Some sketch algorithms

Other sketching algorithms include (from [MM17]):

- Count Min Sketch, which is used for estimating the frequencies of elements in a data stream. It uses hash functions to map elements to a matrix of counters.
- Bloom Filter, which is used for testing whether a given element is a member of a set.
- HyperLogLog, which is used for estimating the cardinality of a multiset.
- MinHash, which is used for estimating Jaccard similarity between sets.
- Random Projection, which is used dimensionality reduction (projecting high-dimensional data into lower-dimensional space while preserving specific properties).
- Count Sketch, which is similar to Count Min Sketch in that it is used for approximate counting in data streams.

4.2.2 Count Min Sketch structure

We will go into the specifics of the Count Min Sketch algorithm. The Count Min Sketch can be implemented with a two-dimensional d $\times$ w array of counters and d independent hash functions. $d = log(\frac{1}{\delta})$ and $w = \frac{2}{\epsilon}$. For each incoming data item i with value v_i counters are updated based on the hash values of the new data item. When we want to find some sort of summary value on v_i, we return the minimum of the counters involving the value v_i. Collisions will happen between some v_i's and v_j's but the amount of collisions is bounded based on the has functions used. The minimum value of the counters is returned since we want to return the estimate that has been involved in the smallest amount of collisions. Thus the collisions have a bounded impact on the quality of the summary approximations of the data stream.

4.3 Summary

We will compare counter-based versus sketching approaches:

4.3.1 Counter-based algorithms

The purpose of counter-based algorithms is counting occurrences or frequencies of elements in a streamed dataset. These algorithms typically use integer valued counters and aim to provide exact counts.

4.3.2 Sketch algorithms

The purpose of sketch algorithms is provide approximate summaries of large streamed datasets, using much less memory than storing the whole dataset and then computing summary statistics. Sketch algorithms typically use sketches vectors to store information about the dataset. These sketch vectors are far more compressed than the overall dataset. There is always a tradeoff between accuracy and efficiency (both space and time efficiency).

4.3.3 Key differences between these approaches

- Counter-based approaches typically require more memory as compared to sketching approaches.
- Sketching algorithms provide summaries of the data stream, whereas counter algorithms provide frequencies of specific values.
- Counter algorithms prioritize accuracy while sketching algorithms prioritize efficiency.

Counter-based algorithms are preferred when frequencies of data values are needed and sufficient memory is available. Sketch algorithms are preferred summary statistics are sufficient and there are limited memory resources.

5 Streaming and Sketching Algorithms for Boolean CSPs

5.1 What is an approximation algorithm for a CSP?

An approximation algorithm is a randomized polytime algorithm that finds a solution that is within some approximation threshold of the optimal solution. The optimal solution can be measured on different objectives based on the CSP. Some objectives include:

- Maximize the number of satisfied constraints. An α-approximation will find a solution that satisfies $\geq \alpha * OPT$ constraints.
- Given a $(1-\epsilon)$-satisfiable CSP, find a solution that satisfies $(1-f(\epsilon))$ fraction of the constraints.
- Minimize the number of unsatisfied constraints.

We can summarize some of the known results for Boolean CSPs using the objective to maximize the number of constraints (this table is from [MM17]).

problem	**constraints** (z_i **is either** x_i **or** $\bar{x}_i$)	**approx. factor**	**optimal? upper bound**
Max Cut	$x_i \neq x_j$	0.87856 [21]	yes [33]
Max 2-Lin(2)	$x_i \oplus x_j = c_{ij}$		
Max 2-SAT	$z_i \vee z_j$	0.94016 [39]	yes [6]
Max Di-Cut	$\bar{x}_i \wedge x_j$		0.87856 [33]
Max 2-And	$z_i \wedge z_j$	0.87401 [39]	0.87435 [6]
Any Boolean 2-CSP	Boolean 2-CSP		0.87435 [6]
Max 3-SAT	$\bigvee_{j=1}^{t} z_{i_j}$ $(t \leq 3)$	7/8 [31, 54]	yes [26]
Max E3-SAT	$\bigvee_{j=1}^{3} z_{i_j}$		
Any Boolean k-CSP	Boolean k-CSP	$\frac{(0.62661-o(1))k}{2^k}$ [43]	$\frac{(1+o(1))k}{2^k}$ [7, 13]
Max k-And	$z_{i_1} \wedge \cdots \wedge z_{i_k}$		
Max SAT	$\bigvee_{j=1}^{t} z_{i_j}$	0.7968 [8]	7/8 [26]
		conj. 0.8434 [8]	
Max Ek-SAT ($k \geq 3$)	$\bigvee_{j=1}^{k} z_{i_j}$	$1-1/2^k$	yes [26]
Max k-All-Equal	$z_{i_1} = \cdots = z_{i_k}$	$\frac{0.88007\,k}{2^k}$ [15]	$\frac{(2+o(1))k}{2^k}$ [7, 13]
Max k-NAE-SAT	$\overline{z_{i_1} = \cdots = z_{i_t}}$ $(t \leq k)$	0.7499 [52]	0.87856 [33]
		conj. 0.8279 [8]	
Max k-Lin(2) ($k \geq 3$)	$z_{i_1} \oplus \cdots \oplus z_{i_k} = 0$	1/2	yes [26]

5.2 Streaming and sketching algorithms for CSPs

Streaming algorithms for CSPs are made to process data in a streaming fashion, so make decisions based on a sequence of data elements that arrive one at a time. This is useful when dealing with large datasets (e.g. those with a large amount of literals). Streaming algorithms provide a way to approximately solve CSPs with limited memory and processing capabilities. Some approaches include (from [Sud22]):

- Randomization: Randomized streaming algorithms can be used to make random choices to quickly explore the solution space without exhaustively searching all possibilities.
- Using sketching techniques: Sketching techniques involve maintaining a compressed summary (sketch) of the data stream. Sketches can be used to estimate different key properties of the data. In CSPs we can summarize the constraints and variables in the problem.
- Memory efficient data structures: Data structures like bloom filters, count min sketches, and hyperloglog help the processing of constraints in a way that is memory efficient.

Sketching algorithms can be thought of as a restriction of streaming algorithms. In CSP solving the most common algorithms used are linear-sketching algorithms. The sketch of a vector v can be projected down to some low-dimensional subspace. This compresses large N dimensional inputs into small s dimensional sketches that end up giving significant information about the original input. Some ways that linear sketching can be applied to CSPs include:

- Count Min sketch is a probabilistic data structure used for estimating the frequency of elements in a stream. In the context of CSPs, it can be used to estimate the frequency of different variable assignments or constraint satisfaction patterns.
- We can use random projections to represent constraints in a lower-dimensional space while preserving properties of the original constraints, which reduces the dimensionality of the CSP.
- We can apply the Johnson-Lindenstrauss transform to embed the variables of the CSP into a lower-dimensional space while approximately preserving the pairwise distances between variables.
- For CSPs that involve linear equations, we can use sketching techniques to maintain a compact representation of the coefficient matrix.
- Hyperloglog is usually used for estimating the cardinality of a multiset. In CSPs we can use it to estimate the number of distinct variable assignments.
- We can represent constraints sparsely using feature hashing.
- If the CSP can be represented as a graph, we can apply graph sketching techniques to represent the graph in a compressed form.

Overall we can use streaming and sketching to reduce the computational load of solving large CSPs. The goal with the techniques above is to find a balance between computational efficiency while still having an informative representation of the CSP. The key tradeoff is between the amount you want to compress the CSP and how close of an approximation you want to get. Obviously if you do not compress the CSP at all you can use a traditional approach to solving a CSP in order to find an exact solution, but in cases where this is infeasible due to computation/memory requirements, you may settle for an approximate solution that is far easier to compute.

6 Dichotomy in Boolean CSPs

6.1 Background

Boolean CSPs are known to exhibit a dichotomy due to the famous Schaefer's Dichotomy Theorem [Sch78] from the 1970s. Schaefer proved that the exact satisfiability version of any Boolean CSP is either in P or is NP-complete. The dichotomy is proven by explicitly giving six classes of Boolean CSPs that have polynomial time algorithms, and that anything outside these classes will be NP-complete. We state the theorem here but omit the proof.

Theorem 6.1 (Schaefer's Dichotomy Theorem)**.** *A finite set Γ of relations over the Boolean domain defines a polynomial time computable Boolean CSP if one of the following holds,*

- *all relations, except those that are constantly false, are true on the all-*1 *assignment*
- *all relations, except those that are constantly true, are false on the all-*0 *assignment*
- *all relations can be equivalently written as 2-CNF*
- *all relations can be equivalently written as CNF with at most one negated literal in each clause*
- *all relations can be equivalently written as CNF with at most one non-negated literal in each clause.*
- *all relations can be equivalently written as a conjunction of affine formulae*

otherwise, the Boolean CSP over Γ is NP*-complete.*

It is not hard to see that for each of the listed class has polynomial time algorithms. The main technical result of Schaefer's dichotomy is to show that all other Boolean CSPs are reducible to 3SAT. Indeed, this is the general approach for showing dichotomies: give tractable algorithms for the positive class(es), and establish intractability arguments for the negative class(es) based on hardness theorems/conjectures, complete problems, and/or existing lower bounds for other classes of problems.

Departing from Schaefer's dichotomy, we ask problems leading different directions: (1) does any finite domain CSP, i.e. not necessarily Boolean, exhibit similar dichotomy in general? (2) instead of exact satisfiability, are there finite classification results regarding *approximability* of the maximization version of CSPs? (3) besides dichotomy with respect to polynomial *time*, can we characterize the approximability of CSPs with respect to *space* in the context of streaming and sketching algorithms?

There have been lines of work along each of the above directions. Feder and Vardi [FV99] conjectured that the exact satisfiability of any finite domain CSP is either in P or NP-complete. The ultimate proof [Bul17] [Zhu17] came from the universal algebra community in 2017, after some 20 years of hard work since the conjecture. In this report, however, we will not discuss this direction any further than noting that modern attempts at this problem have been mostly algebraic, pulling heavy mathematical resources beyond the interest of this TCS project.

As mentioned in direction (2) and (3), it is natural to turn our attention to approximation algorithms when exact satisfiability is too good to hope for or when approximation suffices in practice. It turns out CSPs again exhibit some "near-dichotomy" (with ε error in the appropriate sense). On the time complexity front, in a seminal work [Rag08], Raghavendra gave a characterization of the polynomial time approximability of the maximization version of every finite-domain CSP based on the Unique Games Conjecture— if UGC is true, then the best approximation ratio for every CSP is given by a semidefinite programming. On the space complexity front, [CGSV22] gave a complete characterization of the approximability of every finite-domain CSP in the context of linear sketching algorithms.

Leading to these results, researchers often started by considering the less general, but equally— if not more— insightful Boolean versions of these problems. Austrin in [Aus07b] showed tight Unique Games hardness results for every 2-CSP over the Boolean domain under certain additional conjecture. In an earlier work [CGSV21] shortly before [CGSV22], the authors showed a complete characterization for approximating Boolean CSPs with linear sketches.

In the rest of this report, we will discuss these "near-dichotomy" characterizations of **Boolean CSPs** in greater detail— to do this, we redefine or formalize some notions introduced earlier and add new definitions to better suit the purpose of analysis in this section.

6.2 Preliminaries

Definition 6.2 (Max-CSP)**.** *An instance of* Max-CSP$(\mathcal{F})$ *over a family of functions $\mathcal{F}$ on n variables with m constraints is given by $\Psi = (C_1, \cdots, C_m)$, where each C_i is a constraint on $X_1, \cdots, X_n$. A constraint C is given by a pair $(f, (j_1, \cdots, j_k))$ where $f \in \mathcal{F}$ and $(j_1 \cdots, j_k)$ is a sequence of distinct indices in $[n]$. C is satisfied at an assignment a if $f(a_{j_1}, \cdots, a_{j_k}) = 1$.*

The value of an instance at an assignment a $\mathsf{val}_\Psi(a)$ is defined to be the fraction of the constraints that are satisfied. The value of an instanceval *is the maximum value over all assignments.*

Given an instance, the Max-CSP*problem is to compute or approximate the value.*

Definition 6.3 (Max-CUT)**.** *Given an undirected graph $G = (V, E)$, the* Max-CUT*problem is that of finding a partition $C = (V_1, V_2)$ which maximizes the size of the set $(V_1 \times V_2) \cap E$. Given a weight-function $w : E \to \mathbb{R}^+$, the weighted* Max-CUT*problem is that of maximizing the sum of the cut's weight*

$$\sum_{e \in (V_1 \times V_2) \cap E} w(e)$$

Definition 6.4 (Max-2SAT)**.** *An instance of the* Max-2SAT*problem is a set of Boolean variables and a set of disjunctions over two literals each, where a literal is either a variable or its negation. The problem is to assign the variables so that the number of satisfied literals is maximized. Given a nonnegative weight function over the set of disjunctions, the weighted* Max-2SAT*problem is that of maximizing of the sum of weights of satisfied disjunctions.*

There are various notions of approximation in the literature, we present two most common ones in the context of CSPs. The line of work on UGC-based polynomial time inapproximability mostly adopts the notion of α-approximation (Definition 6.5), while Chou et al. analyze (γ, β)-approximation (definition 6.6) in the context of streaming and sketching algorithms.

Definition 6.5 (α-approximation)**.** *For $\alpha \in [0, 1]$, an α-approximation algorithm $\mathbf{A}$ for* Max-CSP$(\mathcal{F})$ *is one that for every instance Ψ outputs a value $\mathbf{A}(\Psi)$ satisfying $\alpha \cdot \mathsf{val}_\Psi \leq \mathbf{A}(\Psi) \leq \mathsf{val}_\Psi$.*

Definition 6.6 ((γ, β)-approximation)**.** *For $0 \leq \beta < \gamma \leq 1$, an algorithm $\mathbf{A}$ solves the (γ, β)-approximation version of* Max-CSP$(\mathcal{F})$ *if the following two conditions hold:*

1. *For every Ψ such that $\mathsf{val}_\Psi \geq \gamma$, $\mathbf{A}(\Psi) = 1$*
2. *For every Ψ such that $\mathsf{val}_\Psi \leq \beta$, $\mathbf{A}(\Psi) = 0$*

We denote the (γ, β)-version of Max-CSPas (γ, β)-Max-CSP$(\mathcal{F})$, it can be thought of as a "gapped promise problem" where an algorithm solving it needs to distinguish instances where at least γ fraction of the constraints can be satisfied from instances where at most β fraction of the constraints can be satisfied.

It is worth discussing how the two notions of approximation translate. In short, (γ, β)-Max-CSP($\mathcal{F}$) is at least as powerful as the α-approximability of Max-CSP($\mathcal{F}$). To see this, suppose we have α-approximation algorithm $\mathbf{A}$, consider (γ, β) where $\beta < \alpha\gamma$, then we can make another algorithm $\mathbf{A}'$ that outputs 1 if the original $\mathbf{A}(\Psi) > \beta$ and 0 otherwise. We also have the converse hold in the following sense: if for some α we have that for every $\gamma \in [0,1]$, the (γ, β)-Max-CSP($\mathcal{F}$) is solvable for $\beta = \alpha\gamma$, then for every ε we have that Max-CSP($\mathcal{F}$) is $(\alpha - \varepsilon)$-approximable[1] [CGSV22].

We also formalize the concepts of streaming and sketching algorithms introduced in Section 5 in the context of CSPs. We use $\mathsf{C}_{\mathcal{F},n}$ to denote the set of all constraints of Max-CSP(f) on n variables (for simplicity, here $\mathcal{F} = \{f\}$ the singleton). A stream is thus an element of $(\mathsf{C}_{\mathcal{F},n})^*$ and we use λ to denote the empty stream.

Definition 6.7 (Streaming Algorithm)**.** *A space s general streaming algorithm $\mathbf{A}$ for* Max-CSP(f) *on n variables is given by a (state-evolution) function $S : \{0,1\}^s \times \mathsf{C}_{\mathcal{F},n} \to \{0,1\}^s$ and a (output) function $v : \{0,1\}^s \to [0,1]$. Let $\tilde{S} : (\mathsf{C}_{\mathcal{F},n})^* \to \{0,1\}^s$ given by $\tilde{S}(\lambda) = 0^s$ and $\tilde{S}(\sigma_1, \cdots, \sigma_m) = S(\tilde{S}(\sigma_1, \cdots, \sigma_{m-1}), \sigma_m)$ denote the iterated state-evolution map. Then the output of $\mathbf{A}$ on input $\sigma = (\sigma_1, \cdots \sigma_m)$ is $v(\tilde{S}(\sigma))$. We think of a randomized streaming algorithm as simply a distribution on the pairs (S, v).*

Definition 6.8 (Sketching Algorithm)**.** *A (deterministic) space s streaming algorithm $\mathbf{A} = (S, v)$ is a sketching algorithm if there exists a compression function* $\mathsf{COMP} : (\mathsf{C}_{\mathcal{F},n})^* \to \{0,1\}^s$ *and a combination function* $\mathsf{COMB} : \{0,1\}^s \times \{0,1\}^s \to \{0,1\}^s$ *such that the following hold:*

- $S(z, C) = \mathsf{COMB}(z, \mathsf{COMP}(C))$ *for every $z \in \{0,1\}^s$ and $C \in \mathsf{C}_{\mathcal{F},n}$.*
- *For every pair of streams $\sigma, \tau \in (\mathsf{C}_{\mathcal{F},n})^*$, we have*

$$\mathsf{COMB}(COMP(\sigma), \mathsf{COMP}(\tau)) = \mathsf{COMP}(\sigma \circ \tau)$$

where $\sigma \circ \tau$ represents the concatenation of the streams σ and τ. A randomized algorithm $\mathbf{A}$ is a randomized sketching algorithm if it is a distribution over deterministic sketching algorithms.

A linear sketching algorithm roughly associates with elements of a vector space V and COMB *is simply vector addition in V.*

Finally, we introduce the Unique Games Conjecture which sits as the center of polynomial time α-approximability of CSPs. The proof of [Rag08], as is common in literature, uses a formulation of the Unique Games Conjecture in terms of a Unique Label Cover problem, but here we give an equivalent version that is much lighter in notation.

Conjecture 6.9 (Unique Games Conjecture)**.** *For any $\delta > 0$, there is a large enough number p such that: given a set of linear equations of the form $x_i - x_j = c_{ij} \mod p$, it is* NP*-hard to distinguish between the following two cases:*

1. *There is a solution to the system that satisfies $(1 - \delta)$ fraction of the equations*
2. *No solution satisfies more than δ fraction of the equations*

With these in hand, we are ready to look at time and space "near-dichotomies".

[1] One thing we are hiding here is the class $\mathcal{C}$ that the algorithms come from. For the above analysis to hold, we require $\mathcal{C}$ to have certain closure properties.

6.3 Unique Games Conjecture and Inapproximability of Boolean CSPs

Theorem 6.10 (Goemans-Williamson Max-CUT algorithm [GW95]). *There is a randomized $(\alpha_{GW}, \varepsilon)$-approximation algorithm for* Max-CUT *where ε is any positive number and*

$$\alpha_{GW} = \min_{0 \le \theta \le \pi} \frac{2}{\pi} \frac{\theta}{1 - \cos\theta} \approx 0.878567$$

Theorem 6.11 ([KKMO04]). *Assuming Unique Games Conjecture, then for every constant $-1 < \rho < 0$ and $\varepsilon > 0$, it is* NP*-hard to distinguish between instances of* Max-CUT *that are at least $(\frac{1}{2} - \frac{1}{2}\rho)$-satisfiable from those at most $(\frac{\arccos\rho}{\pi} + \varepsilon)$-satisfiable. In particular, choosing $\rho = \rho^*$ where*

$$\rho^* = \arg\min_{-1 < \rho < 0} \frac{(\arccos\rho)/\pi}{\frac{1}{2} - \frac{1}{2}\rho}$$

implies that it is NP*-hard to approximate* Max-CUT *to within any factor greater than the Goemans-Williamson constant α_{GW}.*

Theorem 6.10 and Theorem 6.11 put together tell us that the Unique Games-hardness of Max-CUT precisely matches the algorithmic guarantee $\alpha_{GW} \approx 0.878567$ of Goemans-Williamson for all $-1 < \rho < \rho^*$. Similarly, it was shown in [Aus07a] that the UGC-hardness of Max-2SAT matches the best known LLZ algorithm [LLZ02] with $\alpha = \alpha_{LLZ} \approx 0.94016$. Is this all coincidence?

Unsurprisingly, the answer is no. The breakthrough Goemans-Williamson algorithm for Max-CUT was the first use of **semidefinite programming (SDP)** in the design of approximation algorithms. As it turns out, hardness results obtained from UGC are deeply connected with the limitations of SDP. In most cases, the choice of optimal parameters at the heart of the hardness result are derived by analyzing worst-case scenarios for the semidefinite relaxation of the problem. In fact, Raghavendra [Rag08] showed that if the UGC is true, then the best approximation ratio of any finite-domain CSP is given by a certain simple SDP.

6.3.1 Semidefinite Programming Relaxation

Max-CSP is essentially a problem of optimization. A common practice for approximation algorithms for optimization problems is convex relaxation, including linear programming and semidefinite programming relaxations. In this section we briefly introduce semidefinite programming through an example.

Definition 6.12 (Semidefinite program). *A semidefinite program is the problem of optimizing a linear function of a symmetric matrix subject to linear equality constriaints and the constraint that the matrix be positive semidefintite*

We return to the Goemans-Williamson algorithm as an example of how to formulate CSPs like Max-CUT into an SDP. Given graph $G = (V, E)$, a cut C partitions the vertices into V_1, V_2, say we assign value 1 to every vertex in V_1 and value -1 to every vertex in V_2, then the Max-CUT problem on graph G can be thought of as the following quadratic programming:

$$\text{maximize } \frac{1}{4} \sum_{(i,j) \in E} (x_i - x_j)^2 = \frac{1}{4} \sum_{(i,j) \in E} x_i^2 + x_j^2 - 2x_i x_j = \frac{1}{2} \sum_{(i,j) \in E} 1 - x_i x_j$$

$$\text{subject to } \forall i \in V, x_i \in \{-1, 1\}$$

We do not hope to solve it efficiently for it's NP-hard, so we relax the constraints to form a vector programming, thinking of $x_i x_j$ as inner product of vectors in $\mathbb{R}^n$:

$$\text{maximize } \frac{1}{2} \sum_{(i,j)\in E} 1 - \mathbf{u_i} \cdot \mathbf{u_j}$$

$$\text{subject to } \forall i \in [n], \mathbf{u_i} \in \mathbb{R}^n, ||\mathbf{u_i}||^2 = 1$$

This is a relaxation because if we take $\mathbf{u_i} = (x_i, 0, \cdots, 0)$, this become an instance of the quadratic programming. The vector programming is further equivalent to the following semidefinite programming:

$$\text{maximize } \frac{1}{2} \sum_{(i,j)\in E} 1 - X_{i,j}$$

$$\text{subject to } \forall i \in [n], X_{i,i} = 1, X \succeq 0$$

where X is a matrix with entries $X_{i,j}$.

In general, once we have an SDP, the approach is to solve the SDP in polynomial time, and through a "rounding" process convert the fractional optimum of the relaxation into a solution that is feasible for the original problem. In the case of Goemans-Williams, we Cholesky-decompose the optimal matrix $X = U^T U$, with $\mathbf{u_i}$'s being the column vectors, we then choose a random hyperplane in $\mathbb{R}^n$ through the origin, and the cut we find is the u_i's above or below the plane. A detailed analysis in [GW95] shows that we can achieve approxiamtion ratio $\alpha \approx .878$

Approximation through convex relaxation has its limitation, and one could imagine the limitation will give rise to inapproximability results.

6.3.2 Integrality Gap, Dictatorship Test, and Unique Games

Definition 6.13 (Integrality Gap)**.** *The integrality gap of a relaxation is the worst-case ratio between the true optimum of a combinatorial problem and the optimum of the relaxation.*

Raghavendra gave a formal explanation to the phenomenon that UGC-based inapproximability matches SDP-based approximation ratio in his seminal paper [Rag08] by showing a generic conversion from SDP integrality gaps to UGC hardness results for every Boolean CSP (the result generalizes to very finite-domain CSP as well). In other words, for every Boolean CSP, if one takes a canonical SDP relaxation, then every integrality gap g also implies a reduction showing that no approximation can be better than g assuming UGC.

We capture relationships between integrality gap, dictatorship test, and Unique Games hardness in the following sequence of implications after adding a few final ingredients.

Definition 6.14 (Dictator)**.** *A function $f : \{0,1\}^n \to \{0,1\}$ is a dictator if the function is given by $f(x) = x_i$ for some fixed $i \in [n]$.*

Dictatorship testing is the property testing task that, given function f (not explicitly), query the function at a few locations, and distinguish whether the function is a dictator, or far from every dictator. We define a generalized CSP (GCSP) to be a CSP with the predicate returning real value in $[-1, 1]$ instead of $\{0, 1\}$. In other words, instead of satisfied/unsatisfied, we now have some sort of payoff function, and allowing the payoff to be negative, we can deal with minimization tasks as well.

The connection between integrality gap instance of a SDP and Unique Game hardness is made through dictatorship testing. Let Φ be an instance of a GCSP Λ, using the SDP solution to Φ,

we can construct a dictatorship test DICT_Φ whose completeness (the probability of success of a true dictator function) is nearly equal to the SDP value. For every function f, there is a generic rounding scheme Round_f for the GCSP Λ.

- (Integrality Gaps $\implies$ Dictatorship Tests) It can be shown that the soundness of the dictatorship test is very close to the performance of the rounding scheme.
- (Dictatorship Test $\implies$ Unique Games Hardness) It can be shown that assuming the UGC, it is NP-hard to distinguish between if the optimal assignment has value higher than the completeness of dictatorship test, or if every assignment has value lower than soundness of dictatorship test.
- (Unique Games Hardness $\implies$ Integrality Gaps)It can be shown that for every Φ, there exists Φ' that has higher SDP value than Φ, and integral optimum lower than the soundness of dictatorship of Φ.

This concludes our discussion on the hardness of approximability of Boolean CSPs in the polynomial *time* setting. In the next section, we turn to the approximability of Boolean CSPs with respect to *space* using sketching algorithms.

6.4 Space dichotomy in the context of linear sketches

In this section we discuss the recent tight dichotomy result on CSP approximation with sketching algorithms working in subpolynomial space regime. The proofs in [CGSV21] is rather convoluted as are most dichotomy results, so instead of giving a complete proof sketch, we will only highlight the contrast between this result and the polynomial time inapproximability results introduced in the previous section in an attempt to draw a broader picture of CSP approximability. We first state the dichotomy as the following theorem (we state the Boolean version, and note that [CGSV22] generalizes it to any finite-domain CSP):

Theorem 6.15. *For every $k \in \mathbb{N}$, for every function $f : \{-1,1\}^k \to \{0,1\}$, and for every $0 \le \beta < \gamma \le 1$, at least one of the following always holds:*

- *(γ, β)-$\mathsf{Max\text{-}CSP}(f)$ has a $O(\log n)$-space linear sketching algorithm.*
- *For every $\varepsilon > 0$, any sketching algorithm that solves $(\gamma-\varepsilon, \beta-\varepsilon)$-$\mathsf{Max\text{-}CSP}(f)$ requires $\Omega(\sqrt{n})$ space. In particular, if $\gamma = 1$, then any sketching algorithm that solves $(1, \beta+\varepsilon)$-$\mathsf{Max\text{-}CSP}(f)$ requires $\Omega(\sqrt{n})$ space.*

*Moreover, this dichotomy is **decidable**, that is, there is an algorithm that uses space $\mathsf{poly}(2^k, \ell)$ that decides which of the two conditions holds given the truth table of f, and γ, β as ℓ-bit rationals.*

6.4.1 Contrast with dichotomies in the polynomial setting

In [CGSV21] the authors gave an excellent comparison between the results in that paper with previous dichotomies including the one in [Rag08]. The first and foremost difference is that the negative result is unconditional, i.e. not dependent on complexity assumptions like the UGC is true or $\mathsf{P} \neq \mathsf{NP}$. The constructions in [CGSV21] and [CGSV22] are also more explicit, in the sense that they deal with (γ, β)-approximability instead of $\forall \varepsilon > 0 (\gamma - \varepsilon, \beta + \varepsilon)$-approximability where ε needs to be given as an input.

7 Conclusion and Open Problems

In this report, we've seen examples of constraint satisfaction problems and algorithmic approaches to solve them exactly or approximately. We discussed various dichotomy-like results associated with exact satisfiability, polynomial time approximability, as well as space complexity of approximation with linear sketches. Some of the results have very deep connections with open problems in computation complexity. We conclude this project by listing a few interesting directions going off of what was discussed in this report regarding CSPs, a family of problems of great theoretical and practical interest: can we extend the sketching dichotomy to streaming algorithms; can we show the space dichotomy for function family with more than one f; what happens if we consider multiple pass streaming algorithms? One could also take the opposite direction and look for approximation-resistant functions with respect to each model we mentioned.

References

[Sch78] Thomas J. Schaefer. "The complexity of satisfiability problems". In: *Proceedings of the tenth annual ACM symposium on Theory of computing* (1978) (cit. on p. 10).

[GW95] Michel X. Goemans and David P. Williamson. "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming". In: *J. ACM* 42.6 (Nov. 1995), pp. 1115–1145. ISSN: 0004-5411. DOI: 10.1145/227683.227684 (cit. on pp. 13, 14).

[Liu98] Zhe Liu. "Algorithms for constraint satisfaction problems (csps)". PhD thesis. University of Waterloo, 1998 (cit. on p. 3).

[FV99] Tomás Feder and Moshe Y. Vardi. "The Computational Structure of Monotone Monadic SNP and Constraint Satisfaction: A Study through Datalog and Group Theory". In: *SIAM J. Comput.* 28 (1999), pp. 57–104 (cit. on p. 10).

[LLZ02] Michael Lewin, Dror Livnat, and Uri Zwick. "Improved Rounding Techniques for the MAX 2-SAT and MAX DI-CUT Problems". In: *Conference on Integer Programming and Combinatorial Optimization*. 2002 (cit. on p. 13).

[KKMO04] S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. "Optimal inapproximability results for MAX-CUT and other 2-variable CSPs?" In: *45th Annual IEEE Symposium on Foundations of Computer Science*. 2004, pp. 146–154. DOI: 10.1109/FOCS.2004.49 (cit. on p. 13).

[Aus07a] Per Austrin. "Balanced Max 2-Sat Might Not Be the Hardest". In: *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*. STOC '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 189–197. ISBN: 9781595936318. DOI: 10.1145/1250790.1250818 (cit. on p. 13).

[Aus07b] Per Austrin. "Towards Sharp Inapproximability For Any 2-CSP". In: *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*. 2007, pp. 307–317. DOI: 10.1109/FOCS.2007.41 (cit. on p. 11).

[Rag08] Prasad Raghavendra. "Optimal Algorithms and Inapproximability Results for Every CSP?" In: STOC '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 245–254. ISBN: 9781605580470. DOI: 10.1145/1374376.1374414 (cit. on pp. 10, 12–15).

[CH09] Graham Cormode and Marios Hadjieleftheriou. "Finding the Frequent Items in Streams of Data". In: *Commun. ACM* 52.10 (Oct. 2009), pp. 97–105. ISSN: 0001-0782. DOI: 10.1145/1562764.1562789 (cit. on p. 6).

[Bul17] Andrei A. Bulatov. "A Dichotomy Theorem for Nonuniform CSPs". In: *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)* (2017), pp. 319–330 (cit. on p. 10).

[MM17] Konstantin Makarychev and Yury Makarychev. "Approximation Algorithms for CSPs". In: *The Constraint Satisfaction Problem: Complexity and Approximability*. Ed. by Andrei A. Krokhin and Stanislav Zivný. Vol. 7. Dagstuhl Follow-Ups. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 287–325. DOI: 10.4230/DFU.VOL7.15301.11 (cit. on pp. 5, 7, 8).

[Zhu17] Dmitriy Zhuk. "A Proof of CSP Dichotomy Conjecture". In: *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. 2017, pp. 331–342. DOI: 10.1109/FOCS.2017.38 (cit. on p. 10).

[CGSV21] Chi-Ning Chou, Alexander Golovnev, Madhu Sudan, and Santhoshini Velusamy. "Approximability of all Boolean CSPs with linear sketches". In: 2021 (cit. on pp. 11, 15).

[CGSV22] Chi-Ning Chou, Alexander Golovnev, Madhu Sudan, and Santhoshini Velusamy. "Approximability of all finite CSPs with linear sketches". In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. 2022, pp. 1197–1208. DOI: 10.1109/FOCS52979.2021.00117 (cit. on pp. 10–12, 15).

[Sud22] Madhu Sudan. *Streaming and Sketching Complexity of CSPs: A survey*. 2022. arXiv: 2205.02744 [cs.CC] (cit. on p. 9).