Antoni Rosinol* & Sharan Raja‡

Constraint Satisfaction Problems

Project

*Sensing, Perception, Autonomy, and Robot Kinetics
Massachusetts Institute of Technology

‡Aerospace Control Laboratory
Massachusetts Institute of Technology

Supervision

Nikhil Bhargava

November 2, 2018

Table of Contents

Chapter 1

Project Proposal

We will explain in this project the fundamentals of Constraint Satisfaction Problems following the notes and content presented in MIT's 16.413 course: *Principles of Autonomy and Decision Making*. In particular, we will explain the following topics in an interactive way by focusing on solving the well-known **Sudoku** problem in different ways, while comparing the benefits and disadvantages of the different approaches:

- **Constraint Propagation**: The easiest instances of Sudoku can be solved by simply applying constraint propagation. We will produce examples of solvable Sudoku instances using the following algorithms:
  - AC-1 (a naïve **arc consistency** algorithm).
  - AC-3 [1]: an improved arc consistency algorithm.
  - PC-2 [1]: a **path consistency** algorithm.

  We will analyze their complexity both theoretically (as in the course), and empirically by running them on multiple instances. We will also present examples of incrementally difficult Sudoku instances to show where these approaches fail respectively, and use this opportunity to introduce the concept of **K-consistency**.

- **Backtracking Search**: Certain Sudoku instances cannot be solved by constraint propagation alone, in which case we need to search for a solution. We will hence present a backtracking algorithm that works on partial assignments, and show that while this algorithm can solve certain Sudoku problems it still struggles with some. Instead, if we combine a constraint propagation (inference) with backtracking we can achieve fast solving times, while solving any instance of Sudoku. In particular, we detail the following algorithms:
  - **Forward checking** [2] is a simple inference algorithm to prune portions of the search space by shortening the domain of variables that are arc-inconsistent with the current variable.
  - **Dynamic variable ordering** [3] provides a framework to dynamically change the order in which the variables are instantiated during tree search. **Minimum remaining value** is a popular ordering heuristic that can be used in the dynamic variable ordering framework to solve search problems faster.

- **Local Search** (also known as Iterative Repair). While backtracking works on partial assignments, local search algorithms work on complete assignments and try to iteratively modify these to converge to a solution. We will briefly cover the **min-conflicts** heuristic [4]. Fun fact is that this heuristic managed to reduce the time taken to schedule a week of observations by the Hubble Space Telescope from three weeks to around 10 minutes [5].

Finally, we will extend the provided course content by presenting the **MAC algorithm** [6] (Maintaining Arc Consistency) which improves the forward checking algorithm that we will learn in class. The basic idea is that MAC propagates constraints recursively when changes are made to the domains of variables.
Finally, we will be using [5] and [7] for reference in parallel with the lecture content.

Chapter 2

Lecture Notes

2.1 Constraint Satisfaction Problems

2.1.1 Introduction

Constraint satisfaction problem represents states using a list of variables with assignments and conditions for a solution in the form of constraints on the variables. This allows us to treat the state space with structure - using a factored representation for each state contrary to the black box view used in state space search problems. This framework allows us to model complex constrained decision problems at varying time scales (Eg: Earth Observation Decadal Planning and and Aircraft design).

2.1.2 Mathematical representation

Formally, Constraint Satisfaction Problem (CSP) can be defined as a triple of $< X, D, C >$ where X is a set of variables, D is the domain of each of the variables in V and C is the set of constraints defining the problem. Each constraint in C can be thought of as a pair of scope S which is a subset of V and a relation R between variables in S. The problem of having seeking a set with one A and two B's can be cast in the CSP framework with $V = \{A, B\}$, $D = \{1, 2\}$ and constraint $C = \{[\{A, B\}, \{\{1, 2\}, \{1, 1\}\}], [\{A, B\}, \{\{1, 2\}, \{2, 2\}\}]\}$ where the first constraint tells that there must be exactly one A and the second constraint imposes the fact that there must be two B's. The solution to the problem is when both the constraints are satisfied which is possible only when the assignment is (1,2).

2.1.3 Constraint graph

CSPs can be visualized using constraint graphs where the nodes represent the variables and the edges represent the constraint between the variables connected by it. Any CSP can be represented using a constraint graph because any constraint involving multiple variables can be converted to binary variables by adding additional variables to the problem.

2.2 Constraint Propagation

Apart from searching the state space, constraint programming also allows us to perform specific type of inference called constraint Propagation which helps in eliminating unwanted parts of the state space based on the constraint satisfaction.

2.2.1 Node consistency

A single variable in a CSP is said to be **node-consistent** if all it's unary constraints are satisfied. We can use this to delete some values in the domain of the variable that doesn't satisfy the unary constraint on that variable. A network is node-consistent if all the variables in it are also node-constraint.

2.2.2 Arc consistency

A variable in CSP is said to be **arc-consistent** if every value in its domain satisfies binary constraints imposed on the variable. Similar to node consistency, checking for arc-consistency can help reduce the domain size of a variable before performing search. Algorithm 1 shown below describes a Revise procedure that ensures that a given node X_i is arc-consistent.

Algorithm 1 Revise

```
1: procedure REVISE(CSP, X_i, X_j)
2: Input: A constraint satisfaction problem, CSP = <X,D,C>
3: Output: returns true iff domain of X_i is revised
4: revised ← False
5:     for x ∈ D_i do
6:         if no value of y in D_j allows (x,y) to satisfy constraints between X_i and X_j then
7:             delete x from D_i
8:     revised ← True
9: return revised
```

2.2.3 AC-1

AC-1 is a constraint propagation algorithm that repeatedly checks for arc-consistency between two nodes and removes those values in the domain of each of the two variables that are not consistent. The algorithm is described below:

Algorithm 2 AC-1

```
1: procedure AC-1(CSP)
2:     Input: A constraint satisfaction problem, CSP =< X, D, C >
3:     Output: CSP', the largest arc-consistent subset of CSP
4:     repeat
5:     for C_ij ∈ C do
6:         REVISE(X_i, X_j)
7:         REVISE(X_j, X_i)
8:     until no domain is changed
```

2.2.4 AC-3

In AC-1, the domain of a variable is changed by deleting the values in the variable's domain that are arc inconsistent with another variable. We notice that everytime the domain of the variable shrinks, some equivalent values in other variables now become inconsistent. AC-1 doesn't take advantage of this and hence it must be run iteratively until the domain of all the variables don't change. AC-3, on the contrary, tries to take advantage of this fact by using a queue that has information about the next arcs to be checked based on the modifications made in the current iteration. This leads to a much faster constraint propagation. The algorithm is described below

TODO: Specify how much better is the complexity fo constraint propagation using AC-3 slide 77

TODO: Specify sound and completeness of arc consistency slide 78

Algorithm 3 AC-3

```
1: procedure AC-3(CSP)
2: Input: A constraint satisfaction problem, CSP = <X,D,C>
3: Output: CSP', the largest arc-consistent subset of CSP
4:     for C_ij ∈ C do
5:         queue ← queue ∪{< X_i, X_j >, < X_j, X_i >}
6:     while queue ≠ {} do
7:         select and delete arc < X_i, X_j > from queue
8:         REVISE(X_i, X_j)
9:         if REVISE(X_i, X_j) caused a change in D_i then
10:            queue ← queue ∪{< X_k, X_i > |k ≠ i, k ≠ j}
```

TODO: Precise that arc consistency does not remove out all infeasible solutions slide 79

2.3 Search in CSPs

Constraint propagation alone is not sufficient to solve many CSPs. **TODO:** Provide example, slide 79 seems to show a case where arc-consistency alone does not solve the problem (but I think that path consistency would do constraint propagation up to the solution...). Can we find a sudoku example where we cannot rule out all possibilities by doing constraint propagation? In this case, a solution must be found using search algorithms.

2.3.1 Standard Search

One option would be to use standard search algorithms such as:

- Breadth first search.
- Depth first search.
- Depth-limited search.

Where a *state* would be a partial assignment, and an *action* (operator) would be assigning a value to a variable. The *initial state* is the one where no variable is assigned a value. The *goal state* is the one where all variables have an assigned value, and all constraints are satisfied.

Complexity and Commutativity of CSPs

Unfortunately, given a CSP with n variables with a domain size of d, results in a branching factor at the top level of nd. This is because any of the n variables can be assigned any of d values. Once a variable has been assigned a value, we are still left with $n-1$ variables which can be assigned d values. By induction, we end up with a tree having $n!d^n \sim \mathcal{O}((nd)^n)$ leaves, which is even larger than the possible d^n complete assignments.
Nevertheless, we can observe than in a CSP the order when assigning variables does not influence the solution. This property is known as **commutativity**. More specifically, a problem is commutative if the order of application of any given set of actions does no effect on the outcome. Therefore, there is no need to consider all variables at each level of the search tree, but just one at a time.
This makes sense intuitevely, for example, if we consider the coloring map problem, we might decide whether to color a certain region red or blue, but we would not decide between coloring one region red or another region blue. Therefore, the number of leaves is reduced to d^n, where we just have to decide on an assignment for each variable at a time.

2.3.2 Backtracking

Backtracking search suits the commutative property of CSPs, as it consists on a depth-first search that chooses one variable at a time. Moreover, it backtracks when an inconsistent partial assignment is reached. Although trivial, backtracking search works because any extension of an inconsistent partial assignment remains inconsistent.
We show in 4 the actual backtracking search algorithm, as presented in [5]. In plain words, the algorithm first chooses an unassigned variable, and loops over its domain by picking one value at a time. Every time a value is taken, if it is consistent, then the algorithm continues to look for a solution. If it is not consistent, then the algorithm backtracks by trying with another value instead.

Algorithm 4 Backtracking-Search

```
1:  function BACKTRACKING-SEARCH(csp)
2:      return BACKTRACK({},csp)
3:  function BACKTRACK(assignment, csp)
4:      if assignment is complete then return assignment
5:      var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
6:      for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
7:          if value is consistent with assignment then
8:              add {var = value} to assignment
9:              inferences ← INFERENCE(csp, var, assignment)
10:             if inferences ≠ failure then
11:                 add inferences to assignment
12:                 result ← BACKTRACK(assignment, csp)
13:                 if result ≠ failure then
14:                     return result
15:         remove {var = value} and inferences from assignment
16:     return failure
```

TODO: Implement algorithm in notebook and play with different implementations of the sub-functions. Show practical examples for Sudoku example
The backtracking algorithm presented above makes use of a set of sub-functions that we detail below:

- SELECT-UNASSIGNED-VARIABLE(*csp*, *assignment*)

  Decides which variable should be assigned next.

- ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*)

  Decides the order in which the values of the variable should be tried. We will see in section 2.3.4 how modifications of this and the previous function result in different general-purpose heuristics.

- INFERENCE(*csp*, *var*, *assignment*)

  Every time a variable has been assigned a value, there is the opportunity to further reduce the domains of the rest of unassigned variables using inference, such as checking for arc-consistency. While the call to this function is not strictly necessary, we will see in section 2.3.3 that interleaving search and inference results in a more efficient algorithm.

2.3.3 Backtracking with Forward Checking

By modifying the call to the function `Inference` in algorithm 4, we can interleave search with inference to reduce the domain of yet unassigned variables, thereby reducing the search space. A simple form of inference is **forward checking**, which consists in establishing arc-consistency for the recently assigned variable with respect to all connected, yet unassigned, variables. **TODO:** Add at this point an example of forward-checking in sudoku problems.

2.3.4 Dynamic Variable Ordering

The function `Select-Unassigned-Variable` in algorithm 4 could be trivially set to pick the next unassigned variable in order, or alternatively pick a random variable each time. Nevertheless, these strategies rarely result in an efficient search. **TODO:** Code an example where this is the case.
The **Most Constrained Variable** (MCV) heuristic. It consists in taking as next variable the one having the smallest domain. It is therefore also known as the minimum-remaining-values heuristic, or the fail-first heuristic. The advantage of using MCV is clear when we have a variable that has no legal values left, in this case, the MRV will select this variable first and failure will be detected immediately, which avoids further searches through other variables.
The MCV heuristic usually performs better than random or static ordering, sometimes by a factor of 1000 or more, although the results vary widely depending on the problem. **TODO:** Can we show this using the Sudoku implementation?
We can also modify the function `Order-Domain-Values` in algorithm 4 to reach a solution faster. The **Least Constraining Value** (LCV) heuristic consists in taking as next value for the variable in hand the one that reduces by the least amount the domains of the neighbor variables. **TODO:** Give Sudoku example or map coloring. The heuristic is therefore just trying to leave the maximum flexibility for subsequent variable assignments. Note that if our objective is to list all possible solutions, this heuristic does not bring any advantage.

2.3.5 Conflict-directed Back Jumping

TODO: Should we explain this?

2.4 Local Search for CSPs (iterative repair)

min-conflicts heuristic. **TODO:** Should we explain this?

2.5 Elimination for Constraints

TODO: We do not have any reference for this??

2.5.1 Variable Elimination

Join and Projection operations.

2.5.2 Bucket Elimination

TODO: Should we give an example, or just mention it?

Chapter 3

Maintaining Arc Consistency

While forward checking (section 2.3.3) detects many inconsistencies, it does not detect all of them. The problem is that forward checking only makes the current variable arc-consistent, yet it does not proceed further, checking that other variables are still themselves arc-consistent. **TODO:** Show example where forward-checking would not detect an inconsistency similar to the one presented in AIMA

The algorithm called **Maintaining Arc Consistency (MAC)** [6] uses as `Inference` function in algorithm 4 the AC-3 algorithm (see algorithm 3). In particular, after a variable X_i is assigned a value, MAC calls AC-3 with the arcs (X_i, X_j) for all X_j that are unassigned variables neighboring X_i (instead of using AC-3 with all the arcs in the CSP).

From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set , the call to AC-3 fails, thereby backtracking immediately.

MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC's queue; but unlike MAC, forward checking does not **recursively** propagate constraints when changes are made to the domains of variables.

TODO: Since at this point we would have already gone through all functions in algorithm 4 and implemented them together with the reader, presenting MAC should not take too much time, as we would just pick the AC-3 algorithm that we implemented previously and plug it into the backtracking algorithm 4.

Bibliography

[1] A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99 – 118, 1977.

[2] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artificial Intelligence*, vol. 14, no. 3, pp. 263 – 313, 1980.

[3] F. Bacchus and P. Van Run, "Dynamic variable ordering in csps," in *International Conference on Principles and Practice of Constraint Programming*, pp. 258–275, Springer, 1995.

[4] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, "Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method.," in *AAAI*, vol. 90, pp. 17–24, 1990.

[5] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[6] D. Sabin and E. C. Freuder, "Contradicting conventional wisdom in constraint satisfaction," in *Principles and Practice of Constraint Programming* (A. Borning, ed.), (Berlin, Heidelberg), pp. 10–20, Springer Berlin Heidelberg, 1994.

[7] R. Dechter and D. Cohen, *Constraint processing*. Morgan Kaufmann, 2003.