Assignment 4

Riley Herman 200352833

2020-06-15

Question 1

Variables Each cell is a variable: $x_0, \ldots, x_9$

Domains Each variable is a digit; their domains are $D_0, \ldots, D_9 = [0, 9]; D_0, \ldots, D_9 \in \mathbb{W}$

Constraints Each variable represents how many times that digit is represented in the number. That can be represented by

$$x_i = \sum_{i=0}^{9} f(x_i, i)$$

where

$$f(x_i, i) = \begin{cases} 1 & x_i = i \\ 0 & x_i \neq i \end{cases}$$

Question 2 See code accompanying this document. Code is also available in this public repo: https://github.com/rileytoddherman/cs890BR-a4q2

Question 3 This course was aptly named Constraint Programming. The core of the course was to study an approach to programming wherein the problem is formulated as a Constraint Satisfaction Problem (or CSP). In order to understand that, one must first understand what a Constraint Satisfaction Problem is, then how to solve it.

To solve a problem, it must first be expressed in a form the solver can understand. This is what is intended for the modelling phase of solving a CSP. A CSP is expressed in three parts:

- Variables
  - Variables can be thought of placeholders for eventual values. This will be how the solution is expressed: an assignment of a value for each variable.
- Domains
  - Each variable will have a domain. A domain is a set of possible values. This can be expressed discretely or continuously. Often when expressing a CSP it is tempting to limit the domains using the constraints on the outset; however, this is not ideal as an over-aggressive elimination on the outset may remove a potential solution. Another useful aspect of domains is that they tell whether a solution may exist; if a domain is empty then the CSP has no solution.
- Constraints
  - Constraints are exactly that: they constrain the problem. Constraints are usually expressed as a series of possible value combinations, but they can also be expressed as functions or equations using the variables. A constraint can constrain any number of variables (except for none). Usually, a constraint cannot be broken in a valid solution, although some problems may express flexibiliy around their constraints.
- Objective function (optional)
  - An objective function is an appendium to a CSP if the desired outcome is to be optimized towards a certain solution. This creates what is known as a COP, or Constraint Optimization Problem. An optimization function has no "solution" per se; it is simply a function that a solution (or a set of solutions) is subject to in order to determine the "best fit" solution.

It is common that a given problem can be expressed in a variety of ways: especially considering different methods of expressing constraints.

Another common obstacle in modelling a problem as a CSP is the transfer of real world scenarios into equations and computer-readable functions. This is especially relevant in terms of time and temporal reasoning. This is where Allen primitives become useful. Allen primitives are a set of possible relations between temporal events. These include concepts such as $before$ and $after$; *overlaps* and *during* and so on. There are thirteen temporal relations of this sort, and their relations with each other can be determined using a 13×13 grid of possible relations. Some of these are easy to ascertain or only have on possibility (for example, if event A happens *during* event B and event B has the *meets* relation with event C, then A must be $before\ C$) while others are harder to narrow (for example, if event A happens $before$ event B and event B happens $after$ event C, then we cannot tell what the relationship between A and C is). Similar to temporal reasoning using Allen primitives, spatial reasoning can be acheived through the use of RCC8. There are eight RCC8 relations which include *covers* and *inside* among others.

There are a few basic tools needed before solutions can be discussed: projections and equivalence. Projections are the full instantiation of each variable from its domain and are commonly expressed using []. A CSP is equivalent to another CSP if and only if they have the same set of solutions. Equivalence can be used to repeatedly simplify a complex CSP until a manageable CSP is reached. During this process, one may encounter a so-called "failed" CSP. A failed CSP is a CSP in which either one or more domains is empty or it contains the false constraint. Similarly, a so-called "solved" CSP is one in which all of the constraints have been solved and no domain is empty. One other concept that may arise is normalization; a CSP is normalized if at most each pair of variables has one constraint.

Once the CSP has been modelled, it can be solved. There are several methods to solve a CSP: as mentioned above, solving can be done in a domain specific way or a general way. One of the most basic generalized methods is backtracking. Backtracking can be visualized as a tree searching algorithm wherein nodes are equivalent CSPs that have been generated "on-the-fly", leaves are either solved or failed CSPs, and the tree is traversed similar to a depth first search. If the problem is a COP as described above, then a common general algorithm which follows a similar pattern is branch and bound. Like backtracking, branch and bound is a tree searching algorithm. A leaf node is reached in the same scenarios as with backtracking with the addition of one more scenario: for each node, the value with respect to the objective function is calculated and stored. Presuming the desire is for one optimal solution, if the current objective is already worse than an objective that has been reached then the branch is "bounded" and processing on that branch stops.

Naturally, exact methods alone are not the fastest way to solve a CSP. One method of cutting the computational complexity of the problem is constraint propagation. The idea is to coninually reduce either the domains or the constraints of the problem while maintaining an equivalent problem. There are several methods to doing this: some use a theoretical framework, such as unification and Gaussian elimination techniques, and others are based on the idea of consistency.

Consistency in terms of CSPs comes in a variety of scales. The smallest scale is node consistency: if there is a unary constraint on a variable, then the domain of that variable must reflect that constraint. Fundamentally, consistency relies on the domain reflecting the constraints. That is, all values included in the domain of a variable must be attainable with respect to the constraints on that variable. This is also true of arc consistency, which is concerned with consistency between binary constraints and with respect to two variables. Hyper-arc consistency reflects this concept for n variables. Arc consistency does not necessarily need to be reflected both ways; that is, an arc can be consistent in one direction and not the other. In this situation we refer to the consistency as directionally arc consistent. Path consistency is slightly more complex: in this situation, the path is consistent if, assuming a normalized CSP, the constraint between two variables is reflectant of the constraints between the first of those variables, a second related variable, and the remaining variable in the original pair. Of course, like arc consistency, this "path" must be traversed in both directions for the constraints to be considered path consistent. If it is only consistent in one direction, it is known as directionall path consistent. Note that consistency in these terms does not reflect whether the CSP on it's whole has a solution; it is a tool one can use to eliminate irrelevant constraints or values in a domain.

Besides using consistency, in order to speed up exact methods of solving CSPs, one can propagate the constraints such that states that will inevitably violate a constraint are not met. Three methods of constraint propagation were presented in this course: forward checking, partial look-ahead, and full look-ahead (or maintaining arc conisitency; MAC). Forward checking removes any value from any domain of any variable such that that value is in violation of a constraint given the current instantiation of this variable. Partial look-ahead imposes directional arc consistency from the currently instantiated variable upon all variables sharing a constraint with this one. Full look-ahead imposes arc consistency across the entire current instantiation.

Solving a CSP can take many forms; some may be domain specific and others may be more generalized. For some problems, the desired answer is whether the problem is consistent, a single solution, all of the solutions, an optimal solution, or all optimal solutions. The problem will determine the desired solution.

In terms of applications, constraint programming has been applied to a variety of subjects. Some of these include (but are cetainly not limited to):

- interactive graphic systems
  - for example, expressing geometric coherence for scene analysis
- molecular biology
  - for example, DNA sequencing
- financial applications
  - for example, portfolio optimization
- electrical engineering
  - for example, finding faults in a circuit

- numerical computation
  - for example, solving polynomial constraints
- natural language processing
  - for example, optimization of NLP parsers
- code optimization