Lösen von Constraint Satisfaction Problemen mit C++

Tim Luchterhand
10. September 2022

1 Einführung

Ein *Constraint Satisfaction Problem* (CSP) ist eine Problemstellung, bei der einer Menge von Variablen Werte zugewiesen werden müssen, ohne dass dabei bestimmte Beschränkungen (*Constraints*) verletzt werden. Formal besteht ein CSP aus einer Menge von Variablen V, einer Menge von Wertedomänen D und einer Menge von Constraints C. Das CSP ist gelöst, wenn jeder Variable aus V ein Wert aus ihrer zugehörigen Wertedomäne $d \in D$ zugewiesen wurde, sodass alle Constraints aus C erfüllt sind. Ein einfaches Beispiel ist das Damenproblem, bei dem insgesamt acht Damen so auf einem Schachbrett platziert werden müssen, dass sie sich gegenseitig nicht bedrohen. Hierbei entspricht V den Damen, denen ein passendes Feld zugewiesen werden muss. Bei jeder Dame umfasst die Wertedomäne d alle möglichen Felder des Schachbretts und die Constraints wurden bereits erwähnt. Neben dem eben beschriebenen Beispiel finden sich CSPs häufig im Bereich der künstlichen Intelligenz wieder. Beispielsweise werden zum Planen von wissenschaftlichen Beobachtungsmissionen auf dem Hubble Space Telescope Techniken zum Lösen von CSPs verwendet, um die Betriebszeit und damit Kosten zu reduzieren [1].

2 Lösen von CSPs

CSPs kann man in verschiedene Unterklassen unterteilen. Die allgemeinste Form lässt hierbei Variablen mit unendlichen (auch kontinuierlichen) Wertedomänen, soft-Constraints (also Präferenzen, anstatt Beschränkungen) sowie n-äre Constraints zu. Das Lösen eines CSP kann im Allgemeinen sehr kompliziert sein, da es sich um ein NP-vollständiges Problem handelt. In dieser Arbeit werden allerdings einige Einschränkungen gefordert, sodass sich die Implementierung eines Lösers vereinfacht: Zugelassen werden nur Variablem mit diskreten, endlichen Wertedomänen und harte, binäre Constraints. Letztere sind also Beschränkungen, die von genau zwei Variablen abhängen und in einer gültigen Lösung des CSP erfüllt sein *müssen*. Ein solcher Spezialfall lässt sich dann beispielsweise durch folgenden Algorithmus lösen (gegeben als C++ Pseudoimplementierung):

```
1 bool cspSolver(CSP &problem) {
2     if (problem.isSolved()) {
3         return true;
4     }
5
6     auto &currentVariable = problem.getNextUnassignedVariable();
```

```
7       for (auto value : currentVariable.domain) {
8           currentVariable.assign(value);
9           bool success = false;
10          if (currentVariable.isConsistent()) {
11              success = cspSolver(problem);
12          }
13
14          if (success) {
15              return true;
16          }
17
18          currentVariable.removeAssignment();
19      }
20
21      return false;
22  }
```

Der hier gezeigte Algorithmus ist eine rekursive Backtracking-Suche, die in jedem Rekursionsschritt versucht, einer noch nicht zugewiesenen Variable einen Wert zuzuteilen. Allerdings macht sich der Algorithmus noch nicht zunutze, dass alle Constraints binär sind. Ganz zu Beginn befindet sich die Abbruchbedingung: Wenn bereits alle Variablen einen gültigen Wert besitzen und kein Constraint verletzt ist, liefert die Methode *cspSolver* **true**. Die gültige Zuweisung ist hier im Argument *problem* (mit nicht näher definiertem Datentyp *CSP*) gespeichert, das als Referenz übergeben wurde. Ist das Problem noch nicht gelöst, wird die nächste nicht zugewiesene Variable ausgewählt und alle möglichen Werte der zugehörigen Wertedomäne werden sukzessive ausprobiert. In Zeile 10 wird überprüft, ob die aktuelle Zuweisung erlaubt ist. Im einfachsten Fall erfolgt das durch Überprüfen sämtlicher Constraints. Falls der Wert kein Constraint verletzt, wird im nächsten Rekursionsschritt die nächste Variable zugewiesen. Sollte es bei einer Variable keine gültige Zuweisung geben, ist der Algorithmus in eine Sackgasse gelaufen. Im Suchbaum läuft der Algorithmus dann so lange aufwärts, bis wieder eine gültige Zuweisung für eine Variable gefunden wird. Gibt es keine solche Variable, dann ist das Problem nicht lösbar, z.B. aufgrund widersprüchlicher Constraints.

Wie hier unschwer zu erkennen ist, besitzt diese naive Backtracking-Suche jedoch eine exponentielle Komplexität in der Anzahl der Variablen, was bei großen CSPs schnell zu sehr langen Laufzeiten führen kann. Tatsächlich ist aktuell kein effizienter Algorithmus (also mit polynomieller Laufzeit) bekannt, der ein CSP, selbst mit den oben genannten Einschränkungen, effizient lösen kann, da es sich nach wie vor um ein NP-vollständiges Problem handelt [2]. Deshalb werden in der Praxis häufig zusätzlich Heuristiken verwendet, um den Suchraum zu verkleinern und dadurch die Laufzeit zu verringern. Diese Heuristiken können beispielsweise entscheiden, welche Variable als nächstes ausgewählt werden soll oder welcher Wert als nächstes ausprobiert wird (siehe beispielsweise [3]).

2.1 Lösen durch Constraint Propagation

Ein Problem des zuvor vorgestellten Algorithmus besteht darin, dass die ungültige Zuweisung einer Variablen häufig erst viele Rekursionsschritte später bemerkt wird. Dadurch werden potentiell Äste im Suchbaum verfolgt, die von vornherein schon hätten ausgeschlossen werden könnten. Um dieses Problem zu lösen, bietet sich *Constraint Propagation* an. Dabei werden aus bereits

bestehenden Constraints neue hergeleitet, um die Wertedomänen von Variablen zusätzlich einzuschränken (in [3] auch *Consistency Enforcing* genannt). Dies soll an folgendem Beispiel erläutert werden. Angenommen es gibt zwei Variablen A und B, die jeweils Werte zwischen einschließlich eins und drei annehmen können. Das Constraint sei $A < B$. Daraus lässt sich folgern, dass für A unmöglich der Wert 3 gewählt werden kann, da es in der Domäne von B keinen größeren Wert gibt. Dieser Wert kann also von vornherein entfernt werden und muss nicht erst ausprobiert werden. Analog kann bei B der Wert 1 aus der Domäne entfernt werden. In diesem Beispiel stellt sich nach der erläuterten Anpassung der Wertedomänen der Variablen sogenannte *Arc Consistency* (AC) ein.

Arc Consistency Ein CSP mit ausschließlich binären Constraints lässt sich als Graph darstellen, bei dem die Knoten die Variablen repräsentieren und die Kanten die Constraints. Eine gerichtete Kante zwischen zwei Variablen A und B wird *Arc* genannt und im Folgenden mit $A \rightarrow B$ bezeichnet. Dies lässt sich interpretieren als: "A fordert von B die Einhaltung eines Constraints". Ein Constraint an sich lässt sich allgemein als binäres Prädikat angeben und definiert immer beide Richtungen, also $A \rightarrow B$ und $B \rightarrow A$. Bei dem Begriff der *Arc Consistency* (AC) hingegen spielt die Richtung eine wichtige Rolle, weshalb zwischen Arcs und Constraints unterschieden werden muss. Formell ist eine Variable A mit Wertedomäne d_A *arc consistent* zu einer Variablen B mit Wertedomäne d_B, wenn $\forall v \in d_A \; \exists w \in d_B$, sodass die Zuweisung $A = v$, $B = w$ keine Constraints zwischen A und B verletzt. Im Folgenden wird dies bezeichnet durch $A \overset{\text{ac}}{\rightarrow} B$. Hier ist zu beachten, dass im Allgemeinen aus $A \overset{\text{ac}}{\rightarrow} B$ nicht folgt $B \overset{\text{ac}}{\rightarrow} A$. Sind alle alle Arcs zwischen allen Variablenpaaren eines CSP konsistent, dann nennt man das Problem ebenfalls *arc consistent* [4]. Ein bekannter Algorithmus, der AC in einem CSP herstellt, ist der AC-3 Algorithmus (siehe beispielsweise [5]).

Auch wenn das Herstellen von AC das Problem im Allgemeinen nicht löst, kann dadurch der effektive Suchraum verkleinert werden, wodurch die Suche nach einer Lösung beschleunigt werden kann. So lässt sich beispielsweise der zuvor angegebene naive Backtracking-Algorithmus folgendermaßen anpassen:

```
1  bool cspSolver(CSP &problem) {
2      if (problem.isSolved()) {
3          return true;
4      }
5
6      auto &currentVariable = problem.getNextUnassignedVariable();
7      for (auto value : currentVariable.domain) {
8          currentVariable.assign(value);
9          if (!obtainArcConsistency(problem)) {
10             currentVariable.removeAssignment();
11             continue;
12         }
13
14         if (cspSolver(problem)) {
15             return true;
16         }
17
18         currentVariable.removeAssignment();
19     }
```

```
20
21      return false;
22  }
```

Nach der Zuweisung der aktuellen Variable in Zeile acht wird hier durch die Methode *obtainArcConsistency* das Problem auf ein äquivalentes Problem transformiert, wobei ggf. Werte aus den Domänen mancher Variablen gelöscht werden. Sollte sich die Domäne einer Variablen auf die leere Menge reduzieren, ist das Problem von diesem Rekursionsschritt aus nicht mehr lösbar, sodass direkt der nächste Wert ausprobiert werden kann. In diesem Fall gibt *obtainArcConsistency* **false** zurück. Ist das Herstellen der AC erfolgreich, kann wie zuvor der nächste Rekursionsschritt ausgeführt werden, wobei im Idealfall die Domänen der Variablen reduziert wurden. In manchen Fällen lässt sich ein CSP sogar direkt durch das Herstellen von AC lösen.

3 Konzept und Anforderungen

Im Rahmen dieser Arbeit soll ein generischer Algorithmus entwickelt werden, der beliebige CSPs mit den in Abschnitt 2 beschriebenen Einschränkungen lösen kann. Wie in Abschnitt 2.1 erörtert, bietet sich dazu eine Backtracking-Suche an, die sich zusätzlich Constraint Propagation zunutze macht. Hierfür wird der populäre AC-3 Algorithmus verwendet, da er vergleichsweise simpel zu implementieren ist und trotzdem mit fortgeschritteneren Algorithmen wie dem AC-4 mithalten kann. Teilweise hat AC-3 sogar Vorteile gegenüber AC-4 [6].

3.1 Definieren eines CSP

Um den Lösealgorithmus für beliebige CSPs zu implementieren, werden Datenstrukturen mit einer einheitlichen Schnittstelle benötigt. Gleichzeitig muss der Nutzer in der Lage sein, ein beliebiges Problem elegant und möglichst kompakt zu definieren. Außer den Einschränkungen, die in Abschnitt 2 genannt wurden, sollen keine weiteren Beschränkungen gefordert werden, damit der Löser auf eine große Menge von Problemstellungen angewandt werden kann. Es bietet sich deshalb an, sich an der allgemeinen mathematischen Definition für CSPs zu orientieren und Datenstrukturen für Variablen, Wertedomänen und Constraints zur Verfügung zu stellen. Da die Wertedomänen eng mit den zugehörigen Variablen zusammenhängen, liegt es nahe, beide Strukturen zu kombinieren, beispielsweise indem Variablen einen Member *domain* bekommen. Bei der Umsetzung der Constraints sind mehrere Optionen denkbar: Jede Variable könnte eine Liste mit Verweisen auf andere Variablen besitzen, zu denen eine Abhängigkeit besteht, zusammen mit einem Funktionsobjekt, das die konkrete Relation angibt. Dieses Konzept wäre besonders bei der Implementierung des AC-3 Algorithmus vorteilhaft, da hier zu einer Variablen V alle eingehenden Arcs $N_i \rightarrow V$ benötigt werden. Dabei bezeichnen N_i, $i \in \{1, \dots, k\}$ die k Nachbarn von V im Constraint-Graph. Der Nachteil bei dieser Lösung ist allerdings, dass der Nutzer bei der Definition des CSP immer beide Richtungen eines Constraints explizit angeben muss. Das kann einerseits aufwendig werden und andererseits zu Fehlern führen, wodurch das CSP dann nicht wohldefiniert wäre. Deshalb wird in dieser Arbeit ein Ansatz bevorzugt, bei dem lediglich eine Richtung angegeben werden muss. Da der AC-3 Algorithmus während eines Durchlaufs mehrmals auf die eingehenden Arcs einer Variablen zugreifen muss, sollte vor Aufruf des eigentlichen Lösealgorithmus zunächst eine Tabelle mit Variablen und zugehörigen eingehenden Arcs erstellt werden. Hierfür bietet sich eine assoziative Datenstruktur wie beispielsweise eine *std*::*unordered_map* an.

3.2 Erweiterbarkeit des Algorithmus

Eine offensichtliche Anforderung an die zu implementierende Datenstruktur *Variable* ist die Unterstützung von Wertdomänen mit beliebigen Werttypen. Dies kann einfach durch eine Template-Klasse realisiert werden. Für den Algorithmus muss eine Variable nur einige wenige Methoden zur Verfügung stellen, nämlich die Zuweisung eines Werts sowie Mechanismen zur Manipulation der Wertedomäne. Möglicherweise will der Nutzer jedoch die Variable mit zusätzlicher Funktionalität versehen, sodass Erben aus der Basisklasse *Variable* möglich sein soll. Da es nicht vorgesehen (und auch nicht notwendig) ist, dass der Nutzer die Basisimplementierung der für den Algorithmus notwendigen Methoden der Klasse *Variable* überschreibt, lässt sich der Algorithmus rein statisch polymorph entwerfen. Dies hat den Vorteil, dass zusätzliche Rechenkosten vermieden werden.

Wie bereits in Abschnitt 2 angeführt, kommen beim Lösen von CSPs Heuristiken bei der Auswahl der nächsten Variablen zum Einsatz. Diese sind meistens problemspezifisch, da eine optimale allgemeine Heuristik aktuell nicht bekannt ist. Es liegt also nahe, den Algorithmus so zu implementieren, dass der Nutzer eine eigene Heuristik zum Lösen des Problems angeben kann.

4 Umsetzung

Ausgehend von den in Abschnitt 3 beschriebenen Anforderungen wird im folgenden Teil die Implementierung des generischen CSP-Lösers vorgestellt.

4.1 Zentrale Datenstrukturen

Um ein CSP zu definieren, muss des Nutzer lediglich zwei Dinge tun: Angegeben werden müssen alle vorhandenen Variablen mit zugehöriger Wertedomäne sowie eine Menge von Constraints, die die Abhängigkeiten zwischen den Variablen darstellen. Dafür werden die Template-Klassen *csp*::*Variable* sowie *csp*::*Constraint* und *csp*::*Arc* zur Verfügung gestellt.

4.1.1 csp::Variable

Die Klasse *csp*::*Variable* ist, wie schon in Abschnitt 3.1 beschrieben, sehr einfach gehalten. Sie besitzt einen Member vom Typ *std*::*list*, der die zugehörige Wertedomäne darstellt. Der Datentyp der Werte in der Domäne lässt sich durch den Template-Typparameter der Klasse angeben. Die Entscheidung ist hier bewusst auf eine *std*::*list* gefallen, da später durch den AC-3 Algorithmus Werte an beliebigen Stellen in der Domäne gelöscht werden dürfen. Eine *std*::*list* garantiert für eine solche Operation konstanten Aufwand [7]. Da bidirektionales Iterieren nicht benötigt wird, hätte auch eine *std*::*forward_list* verwendet werden können. Allerdings stellt diese Datenstruktur keine Methode *size*() zur Verfügung. Ein weiterer Kandidat war ein *std*::*unordered_set*. Der Vorteil dabei wäre, dass die Wertedomäne dann garantiert einzigartige Elemente enthält. Allerdings funktioniert ein *std*::*unordered_set* nur, wenn eine passende Hash-Funktion angegeben wird. Zwar gibt es mit *std*::*hash* für einige häufig verwendete Datentypen Standardimplementierungen. Wird allerdings ein eigener Datentyp verwendet, so müsste auch explizit eine Hash-Funktion angegeben werden. Dies wäre auch möglich über einen weiteren Template-Typparameter mit Default-Typ, würde den Nutzer aber trotzdem zur Implementierung einer Hash-Funktion zwingen, sobald *std*::*hash* keine Spezialisierung für den verwendeten Datentyp anbietet. Damit der Nutzer bei der Angabe der Wertedomäne auch andere Containertypen verwenden kann, existiert für den Setter *setValueDomain* eine Überladung, die beliebige Containertypen annimmt und deren

Inhalt in die *std*::*list* kopiert. Die Container müssen lediglich Iterieren unterstützen, damit das Kopieren mit *std*::*copy* durchgeführt werden kann.

Die Klasse *csp*::*Variable* besitzt keinen eigenen Member, der den zugewiesenen Wert repräsentiert. Eine Variable gilt als zugewiesen, wenn die Wertedomäne genau einen Wert enthält. Ein Nutzer könnte aber bei Bedarf aus *csp*::*Variable* ableiten und beispielsweise eine Methode *getValue*() hinzufügen.

4.1.2 csp::Constraint und csp::Arc

Constraints werden entweder durch *csp*::*Constraint* oder durch *csp*::*Arc* angegeben. Dabei erbt *csp*::*Arc* aus *csp*::*Constraint*. Der Unterschied zwischen beiden Klassen wurde bereits in Abschnitt 2.1 erläutert: Constraints geben eine bidirektionale Abhängigkeit zwischen zwei Variablen an, wohin Arcs gerichtet sind. Beide lassen sich jedoch fast identisch verwenden. Zunächst soll die Klasse *csp*::*Constraint* genauer betrachtet werden. Sie enthält zwei Pointer-Typen, die jeweils auf eine *csp*::*Variable* zeigen und das Constraint zwischen den Variablen wird durch ein binäres Prädikat in Form einer *std*::*function* repräsentiert. Der genaue Typ des Pointers wird nicht vorgeschrieben und lässt sich über einen Template-Typparameter angeben. Grundsätzlich sind alle Typen zulässig, die den Dereferenzoperator $*$ unterstützen. Möglich sind also beispielsweise einfache "Raw-Pointer" oder auch Smart-Pointer, solange sie den Copy-Konstruktor unterstützen. Alle weiteren Hilfsklassen und -methoden sowie der Lösealgorithmus selbst verwenden alle denselben Pointer-Typ. Der Pointer selbst muss auf einen Typ zeigen, der aus *csp*::*Variable* ableitet. Verwendet der Nutzer eine eigene Variablen-Klasse, die aus *csp*::*Variable* erbt und weitere Member besitzt, so ist es weiterhin über den Pointer möglich auf diese Member zuzugreifen, auch wenn *csp*::*Variable* selbst nicht polymorph ist. Dies ermöglicht mehr Flexibilität beim Schreiben von eigenen Heuristiken zur Variablenauswahl, wie später noch genauer erläutert wird.

Mithilfe von SFINAE-Techniken lässt sich zur Übersetzzeit überprüfen, ob der gegebene Typparameter die oben genannten Anforderungen erfüllt, also dereferenzierbar ist und auf einen Typ zeigt, der aus *csp*::*Variable* erbt:

```
1   namespace type_traits {
2       namespace implementations {
3           template<typename T>
4           std::true_type pointerTest(decltype(*std::declval<T>(), std::declval<T>()));
5
6           template<typename T>
7           std::false_type pointerTest(...);
8
9           template<typename VarType, template<typename...>typename Domain>
10          std::true_type derivedTest(Variable<VarType, Domain> *);
11
12          std::false_type derivedTest(...);
13      }
14      template<typename T>
15      struct is_dereferencable : decltype(implementations::pointerTest<T>(std::declval<T>()))
            {};
16      template<typename T>
17      struct is_derived_from_var : decltype(implementations::derivedTest(std::declval<std::
            remove_reference_t<T>*>())) {};
```

Durch **static_assert** lassen sich dann aussagekräftige Fehlermeldungen zur Übersetzzeit generieren:

```
1  template<typename VarPtr>
2  class Constraint {
3  public:
4      static_assert(type_traits::is_dereferencable<VarPtr>::value,
5                    "Constraints must be constructed from pointer type to csp::
                          Variable");
6      static_assert(type_traits::is_derived_from_var<decltype(*std::declval<VarPtr>())>::
           value,
7                    "Type referenced by VarPtr must derive from csp::Variable");
```

Ein *csp*::*Arc* lässt sich identisch zu einem *csp*::*Constraint* konstruieren. Allerdings besitz ein *csp* ::*Arc* folgende zusätzliche Member: *from*() liefert einen (allgemeinen) Pointer auf die Variable, von der der Arc ausgeht. Für den Arc $A \rightarrow B$ wäre dies beispielsweise A. Analog würde *to* () einen Pointer auf die Variable B liefern. Mithilfe von *reverse*() lässt sich der Arc umdrehen. Man würde also den Arc $B \rightarrow A$ erhalten. Beide Arcs wären jedoch nach wie vor durch dasselbe Constraint repräsentiert. Die Member *from*() und *to*() stellen dabei unter Beachtung der Richtung des Arcs sicher, dass immer der korrekte Pointer geliefert wird. Außerdem wird die Auswertung des binären Prädikats durch den Member *constraintSatisfied*() vereinheitlicht, der mit zwei Werten aus den Domänen von *from*() und *to*() aufgerufen werden sollte und dann sicherstellt, dass die Argumente des Prädikats in der richtigen Reihenfolge angegeben werden.

4.1.3 csp::Csp

Die Klasse *csp*::*Csp* repräsentiert ein vollständiges CSP. Sie besteht aus einer Liste von Pointern auf Variablen, einer Liste von Arcs und einer Tabelle, die eingehende Arcs Variablen zuordnet. Analog zu *csp*::*Arc* und *csp*::*Constraint* sind hier der genaue Pointer-Typ und der genaue Variablen-Typ nicht festgelegt. *std*::*vector* bietet sich für die Liste der Variablen an, da hier keine Elemente hinzugefügt oder gelöscht werden müssen. Der zusammenhängende Speicher ermöglicht außerdem effizienten Random-Access und effizientes Iterieren. Da der AC-3 Algorithmus auf einer Queue von Arcs arbeitet, bietet sich für die Liste der Arcs eine *std*::*list* an. Denkbar wäre auch eine *std* ::*queue* gewesen. Allerdings unterstützt diese kein Iterieren und kein elegantes Einfügen von Elementen am Ende durch *std*::*back_inserter* und *std*::*copy*. Für die Zuordnung von eingehenden Arcs zu Variablen wurde eine *std*::*unordered_map* verwendet, die für jede Variable einen *std*:: *vector* aus Arcs verwaltet. Das Finden von Elementen in einer solchen Datenstruktur benötigt im Durchschnitt konstanten Aufwand [8], sodass im AC-3 Algorithmus eingehende Arcs zu einer gegebenen Variable effizient bestimmt werden können. Alle Member von *csp*::*Csp* sind **const**, sodass sich die bei der Konstruktion festgelegte Topologie des Problems nicht mehr ändern lässt. Es ist also nicht möglich, neue Variablen oder Constraints hinzuzufügen oder zu entfernen. Allerdings lassen sich die Variablen über die Pointer verändern, sodass der Löser später beispielsweise die Wertedomänen anpassen kann. Da die korrekte Zuordnung von eingehenden Arcs zu Variablen essentiell für die Korrektheit des Lösers ist, darf ein *csp*::*Csp* nur über die **friend**-Methode *csp*::*make_csp* konstruiert werden.

4.2 Definieren eines CSP

Wie schon in Abschnitt 3 erörtert, soll der Nutzer lediglich eine Menge von Variablen und Constraints angeben müssen, um ein CSP vollständig zu definieren. Hierfür gibt es die Methode

csp::*make_csp*, die zwei Container entgegennimmt: Im ersten sind wieder allgemeine Pointer-Typen enthalten, die auf alle Variablen des Problems zeigen. Im zweiten sind entweder *csp*::*Constraint*s oder *csp*::*Arc*s enthalten. Je nach Problem kann es manchmal einfacher sein, beide Richtungen der Constraints explizit anzugeben (wie beispielsweise in der Implementierung des Sudoku-Lösers) oder auch nicht. Deshalb werden beide Typen akzeptiert, jedoch keine Mischung aus beiden. Die Containertypen werden durch Template-Typparameter festgelegt, sodass beliebige Arten von Containern verwendet werden können, die Iteration unterstützen. Werden die Constraints als *csp*::*Arc*s angegeben, so lässt sich das *csp*::*Csp* ohne großen zusätzlichen Aufwand konstruieren. Lediglich die Variablen mit ihren zugehörigen eingehenden Arcs müssen in die *std*::*unordered_map* eingetragen werden. Wird stattdessen eine Menge aus *csp*::*Constraint*s angegeben, so müssen diese zuvor noch in ihre äquivalenten Arcs umgewandelt werden. Dies geschieht durch die Memberfuntion *getArcs*() von *csp*::*Constraint*.

Da beide Überladungen von *csp*::*make_constraint* eine identische Signatur besitzen (bis auf die Namen der Template-Typparameter), muss SFINAE eingesetzt werden, damit die Wahl der richtigen Methode zur Übersetzzeit eindeutig ist. Dazu besitzen beiden Methoden einen dritten Template-Parameter, der lediglich dazu dient, die unpassende Methode auszuschließen:

```
1  template<typename VarContainer, typename ArcContainer, std::enable_if_t<
2          type_traits::is_arc<std::remove_reference_t<decltype(*std::begin(std::declval<
               ArcContainer>()))>>::value,
3          int> = 0>
4  auto make_csp(const VarContainer &variables, const ArcContainer &arcs) -> Csp<std::
       decay_t<decltype(
```

Das Konstrukt nutzt *std*::*enable_if_t*, um einen Template-Parameter vom Typ **int** mit Standardwert 0 zu deklarieren, sodass bei korrekter Benutzung der Methode der dritte Parameter nicht explizit angegeben werden muss. Sollte der im Typ *ArcContainer* enthaltene Typ nicht vom Typ *csp*::*Arc* sein, so definiert *std*::*enable_if_t* keinen Typ und der Ausdruck des Standardwerts ist nicht gültig. Dadurch wird diese Überladung von *csp*::*make_csp* durch SFINAE ausgeschlossen. Bei der Überladung der Methode, die *csp*::*Constraint*s annimmt, wird ein analoger Test durchgeführt, sodass maximal eine der beiden Überladungen ausgewählt werden kann. Die dafür verwendeten Type-Traits sind ähnlich aufgebaut wie jene, die in Abschnitt 4.1.2 vorgestellt wurden.

4.3 Lösealgorithmus

Mithilfe der zuvor dargestellten Datenstrukturen lässt sich der eigentliche Lösealgorithmus recht einfach implementieren. Der während der Backtracking-Suche verwendete AC-3 Algorithmus ist quasi eine Eins-zu-eins-Implementierung des Pseudocodes aus [5]. Der Unterschied ist, dass in der hier vorgestellten Implementierung nur binäre Constraints erlaubt sind. Unäre Constraints lassen sich allerdings trivial auf die Wertedomäne einer Variable übertragen, sodass diese Aufgabe an den Nutzer delegiert wurde. Die Backtracking-Suche selbst ist in der Methode *csp*::*recursiveSolve* implementiert und ist ebenfalls sehr ähnlich zu der in Abschnitt 2.1 vorgestellten Pseudoimplementierung. Auch sie arbeitet direkt auf einer Referenz des Problems und verändert die Domänen der Variablen über die enthaltenen Pointer. Ob das gegebene (Teil-)Problem erfolgreich gelöst werden konnte, wird auch hier durch einen Boolean signalisiert. Ein paar Unterschiede gibt es dennoch:

```
1  template<typename VarPtr, typename Strategy>
2  bool recursiveSolve(Csp<VarPtr> &problem, const Strategy &strategy) {
3      using Domain = typename Csp<VarPtr>::VarT::DomainT;
4      VarPtr nextVar = strategy(problem);
```

```
5           if (nextVar->isAssigned()) {
6               return true;
7           }
```

Zusätzlich zum CSP muss ein Strategie-Funktor übergeben werden, der die nächste, noch nicht zugewiesene Variable aus dem *csp*::*Csp* auswählt. Es werden beliebige Funktoren akzeptiert, die eine korrekte Signatur besitzen. Somit erhält der Nutzer die Möglichkeit, beliebige, problemspezifische Heuristiken zur Auswahl der nächsten Variablen anzugeben. Werden eigene Variablentypen verwendet, die aus *csp*::*Variable* ableiten, kann eine benutzerdefinierte Heuristik aufgrund des statischen Polymorphismus auch auf dessen zusätzliche Felder und Methoden zugreifen, sodass mehr Informationen bei der Auswahl der nächsten Variable zur Verfügung stehen. Es wird gefordert, dass die Heuristik immer eine Variable liefert. Im Falle, dass es keine nicht zugewiesene Variable mehr gibt, muss eine beliebige gewählt werden. Dadurch vereinfacht sich der Goal-Test zu der einfachen **if**-Abfrage in Zeile fünf. Danach werden für die aktuelle Variable sukzessive Werte aus ihrer Wertedomänen ausprobiert:

```
8           Domain valueDomain = std::move(nextVar->valueDomain());
9           for (auto &val : valueDomain) {
10              nextVar->assign(std::move(val));
11              auto cp = util::makeCspCheckpoint(problem);
12              if (!util::ac3(problem)) {
13                  util::restoreCspFromCheckpoint(problem, cp);
14                  continue;
15              }
16
17              if (recursiveSolve(problem, strategy)) {
18                  return true;
19              }
20
21              util::restoreCspFromCheckpoint(problem, cp);
22          }
23
24          return false;
```

Da die Zuweisung eines Wertes durch *assign* die Wertedomäne einer Variable verändert, muss diese zunächst gesichert werden, damit Iterieren weiterhin möglich ist. Außerdem werden vor dem Aufruf des AC-3 Algorithmus alle Wertedomänen aller Variablen gesichert, da diese durch das Herstellen von AC oder den folgenden Rekursionsschritt verändert werden können. Dies geschieht durch *util*::*makeCspCheckpoint*. Der ursprüngliche Zustand lässt sich danach durch *util*::*restoreCspFromCheckpoint* wiederherstellen.

Anstatt für *csp*::*recursiveSolve* einen in-out-Parameter zu nutzen, also das Problem als nicht konstante Referenz zu übergeben, hätte auch ein reiner in-Parameter verwendet werden können, der nicht verändert werden darf. Eine einfache **const**-Referenz reicht dabei nicht aus, da sich die Wertedomänen der Variablen dann trotzdem über die Pointer verändern lassen. Um zu gewährleisten, dass das ursprüngliche Problem wirklich unverändert bleibt, muss vor dem Anpassen der Variablen eine Deep-Copy des *csp*::*Csp* erzeugt werden. Dadurch wird dann gleichzeitig das Sichern der Wertedomänen durch *util*::*makeCspCheckpoint* überflüssig. Allerdings ist das Erstellen einer Deep-Copy nicht trivial, da nicht nur der Speicher hinter den Variablen-Pointern kopiert werden muss, sondern auch die zum Problem zugehörigen *csp*::*Arc*s neu erstellt

werden müssen. Eine Version des Lösers mit in-Parameter wurde ebenfalls implementiert [1]. Hier liefert die Backtracking-Suche bei Erfolg eine Kopie des Problems zurück, bei der alle Variablen korrekt belegt sind. Auch der AC-3 Algorithmus arbeitet nicht mehr direkt auf dem Problem, sondern erstellt ebenfalls eine Kopie, wodurch das Sichern der Wertedomänen nicht mehr benötigt wird. *csp*::*Csp* besitzt eine Methode *clone*(), die die Deep-Copy erstellt. Experimente mit dem Sudokulöser zeigen jedoch, dass diese Version des Lösealgorithmus, selbst mit der gcc-Optimierung $-O3$, um bis zu Faktor drei langsamer ist als die Version mit in-out-Parameter und ohne Deep-Copy [2]. Im Folgenden wird deshalb weiterhin die ursprüngliche Version des Lösers vorgestellt.

Zur einfacheren Verwendung wird mit *csp*::*solve* ein Wrapper für die rekursive Backtracking-Suche bereitgestellt:

```
1   template<typename VarPtr, typename Strategy = strategies::Mrv<VarPtr>>
2   bool solve(Csp<VarPtr> &problem, const Strategy &strategy = Strategy()) {
3       static_assert(std::is_invocable_r_v<VarPtr, Strategy, Csp<VarPtr>>,
4               "Invalid strategy object! Must map from csp::Csp -> VarPtr");
5       if (std::empty(problem.variables)) {
6           return true;
7       }
8
9       if (!util::ac3(problem)) {
10          return false;
11      }
12
13      return recursiveSolve(problem, strategy);
14  }
```

Hier wird außerdem mithilfe des **static_assert** und *std*::*is_invocable_r_v* zur Übersetzzeit überprüft, ob die angegebene Heuristik eine gültige Signatur besitzt. Standardmäßig wird die bereits implementierte Heuristik *strategies*::*Mrv* verwendet. Diese wählt die Variable mit den wenigsten möglichen Werten aus (*Mrv* steht für *Minimum Remainig Values*). Eine weitere Möglichkeit ist die *strategies*::*First* Heuristik, die einfach die nächstbeste nicht zugewiesene Variable auswählt. Darüber hinaus lassen sich aber auch beliebige eigene Heuristiken verwenden, solange die Signatur zulässig ist. Außerdem prüft der Wrapper anfangs, ob das Problem leer ist und ruft einmal den AC-3 Algorithmus auf, da die rekursive Suche davon ausgeht, dass das Problem bereits *arc consistent* ist. Sollte schon dieser erste Schritt fehlschlagen, ist das Problem nicht lösbar und die Backtracking-Suche obsolet.

5 Testsuite und Beispielanwendung

Nachdem ein Überblick über den Aufbau des Lösealgorithmus gegeben wurde, sollen nun kurz die Implementierung eines Sudokulösers sowie die Testsuite vorgestellt werden.

[1] Der Code findet sich auf `https://github.com/Timmifixedit/CSP-Solver/tree/DeepCopy` auf dem experimentellen *DeepCopy*-Branch

[2] Die Laufzeiten wurden beim Lösen des sehr schwierigen Sudokus gemessen, das im Ordner *res* beigefügt ist

5.1 Lösen von Sudokus

Die bekannten Sudoku-Rätsel lassen sich recht einfach als CSP beschreiben. Dabei stellen die einzelnen Felder die Variablen dar, die mit Zahlen von eins bis neun ausgefüllt werden müssen. Die Constraints äußern sich darin, dass Felder in derselben Zeile, Spalte oder im selben 3×3 Block unterschiedliche Werte haben müssen. Für den Löser wurde ein eigener Variablen-Typ erstellt, der im Konstruktor automatisch die Wertedomäne korrekt setzt. Außerdem lässt sich der Wert der Variable bequem mit *getValue*() abfragen:

```
1  using VariableBase = csp::Variable<unsigned int, std::deque>;
2  class SudokuNode : public VariableBase {
3  public:
4      using Domain = VariableBase::DomainT;
5      explicit SudokuNode(unsigned int val) : VariableBase(
6              val == 0 ? Domain{1, 2, 3, 4, 5, 6, 7, 8, 9} : Domain{val}) {
7          assert(val <= 9);
8      }
9
10     [[nodiscard]] auto getValue() const -> std::optional<unsigned int> {
11         if (this->isAssigned()) {
12             return this->valueDomain().front();
13         }
14
15         return {};
16     }
17 };
```

Um die *SudokuNode*s zu verwalten, wurde außerdem die Hilfsklasse *Sudoku* implementiert. Deren Konstruktor liest die gegebenen Werte aus einem *std::istream* ein und instanziiert automatisch alle Variablen mit den korrekten Werten. Der Wert 0 gibt hierbei an, dass ein Sudoku-Feld noch leer ist. Außerdem lässt sich das Sudoku mit *print*() anzeigen und mit *solve*() lösen:

```
1      bool solve() {
2          std::vector<csp::Arc<std::shared_ptr<SudokuNode>>> arcs;
3          arcs.reserve(81 * 20);
4          for (std::size_t index = 0; index < fields.size(); ++index) {
5              auto neighbours = getNeighbours(index);
6              for (auto &nb : neighbours) {
7                  arcs.emplace_back(fields[index], std::move(nb), std::not_equal_to<>());
8              }
9          }
10
11         auto sudokuProblem = csp::make_csp(fields, arcs);
12         return csp::solve(sudokuProblem, csp::strategies::First<std::shared_ptr<SudokuNode>>())
               ;
```

Zunächst müssen alle Abhängigkeiten zwischen den Feldern angegeben werden. Diese werden im *std::vector arcs* gespeichert. Das Verwenden eines *std::array* ist nicht möglich, da ein *csp::Arc* nicht default-konstruierbar ist. Danach wird das zu lösende Problem durch *csp::make_csp* definiert und anschließende durch *csp::solve* gelöst. Hier wird explizit die Heuristik *csp::strategies*

::*First* verwendet, die bei Experimenten mit verschiedenen Sudokus bessere Performance gezeigt hat als die *csp*::*strategies*::*Mrv* Strategie. Nach dem Aufruf von *solve*() kann, falls lösbar, die Lösung des Sudokus erneut mit *print*() angezeigt werden.

5.2 Unit-Tests mit Google Test

Während der Entwicklung wurden insgesamt 21 Unit-Tests mit dem Google-Test-Framework geschrieben, um die korrekte Funktionalität der einzelnen Komponenten zu gewährleisten. Der Aufbau eines solchen Unit-Test soll an einem Beispiel kurz dargestellt werden:

```
1 TEST(util_test, remove_inconsistent_indicate_removal) {
2     using namespace csp::util;
3     auto varA = std::make_shared<TestVar>(std::list{2, 3, 1});
4     auto varB = std::make_shared<TestVar>(std::list{2, 3, 1});
5     TestArc a(varA, varB, std::less<>());
6     EXPECT_TRUE(removeInconsistent(a));
7     EXPECT_FALSE(removeInconsistent(a));
8 }
```

Ein einzelner Test wird mit dem Makro *TEST* definiert, wobei ein Name für die übergeordnete Testgruppe und ein spezifischer Test-Name angegeben werden muss. Innerhalb des Test-Blocks wird dann der Code für den Test platziert. Der hier gezeigte Test überprüft, ob die Methode *util*::*removeInconsistent*, die im AC-3 Algorithmus zum Einsatz kommt, den korrekten Wert zurückliefert. Dies lässt sich mit den von Google Test zur Verfügung gestellten Makros wie *EXPECT_TRUE* überprüfen. Es gibt darüber hinaus weitere Makros, die auch komplexeres Verhalten überprüfen können, beispielsweise auch, ob eine bestimmte Exception ausgelöst wurde.

6 Zusammenfassung

Im Rahmen dieses Projekts wurde ein Algorithmus implementiert, der beliebige Constraint Satisfaction Probleme mit endlichen, diskreten Wertedomänen und binären Constraints lösen kann. Dazu wurde eine rekursive Backtracking-Suche in Kombination mit dem AC-3 Algorithmus verwendet. Der Algorithmus ist komplett statisch polymorph gestaltet und erlaubt somit die Verwendung von beliebigen Wert-Typen, unterstützt Erweiterungen des zur Verfügung gestellten Variablen-Typs und erlaubt das Angeben von benutzerdefinierten Heuristiken zur Variablenauswahl während des Lösens. Mit den vorhandenen Strukturen lassen sich CSPs ähnlich zur mathematischen Formulierung definieren, sodass sich eine Vielzahl von Problemen ausdrücken lässt. Als beispielhafte Anwendung wurde ein Sudoku-Löser implementiert. Weitere bekannte CSPs wie das Damenproblem oder das in einer Übungssitzung vorgestellte topologische Sortieren sollten sich ebenfalls einfach umsetzten lassen.

Literatur

[1] M. Johnston, „Spike: AI Scheduling for Hubble Space Telescope After 18 Months of Orbital Operations“, Jan. 1992.

[2] D. Frost und R. Dechter, „In Search of the Best Constraint Satisfaction Search“, in *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, Ser. AAAI '94, Seattle, Washington, USA: American Association for Artificial Intelligence, 1994, S. 301–306, ISBN: 0262611023.

[3] N. Sadeh und M. S. Fox, „Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem“, *Artif. Intell.*, Jg. 86, Nr. 1, S. 1–41, Sep. 1996, ISSN: 0004-3702. DOI: 10.1016/0004-3702(95)00098-4. Adresse: https://doi.org/10.1016/0004-3702(95)00098-4.

[4] *Arc Consistency*, https://www.sciencedirect.com/topics/computer-science/arc-consistency, zuletzt besucht am 03.08.2020.

[5] *AC-3 algorithm*, https://en.wikipedia.org/wiki/AC-3_algorithm, zuletzt besucht am 03.08.2020.

[6] C. Bessière, „Arc-Consistency and Arc-Consistency Again“, *Artif. Intell.*, Jg. 65, Nr. 1, S. 179–190, Jan. 1994, ISSN: 0004-3702. DOI: 10.1016/0004-3702(94)90041-8. Adresse: https://doi.org/10.1016/0004-3702(94)90041-8.

[7] *std::list*, https://en.cppreference.com/w/cpp/container/list, Zuletzt besucht am 04.08.2020.

[8] *std::unordered_map*, https://en.cppreference.com/w/cpp/container/unordered_map, Zuletzt besucht am 04.08.2020.

Abkürzungsverzeichnis

CSP Constraint Satisfaction Problem

AC Arc Consistency

SFINAE Substitution Failure Is Not An Error