HAW HAMBURG

HAUSARBEIT
Kjell May

Minesweeper als Constraint Satisfaction Problem

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Inhaltsverzeichnis

Lorem ipsum ...

Keywords: Constraint Satisfaction, CSP, AC-3, Revise

1 Einleitung

Minesweeper ist ein klassisches Rätsel-/ Puzzlespiel von Microsoft Windows, welches auf früheren Windows-Versionen vorinstalliert gewesen ist. In dem Spiel geht es darum, alle Felder auf einem Spielfeld aufzudecken, unter denen sich keine Mine befindet. Schafft man das, hat man gewonnen, ansonsten verliert man das Spiel. Um dieses Ziel zu erreichen und die Minen zu lokalisieren, gibt das Spiel einem Hinweise in Form von Zahlen auf aufgedeckten Feldern, welche die Anzahl der umgebenden Minen beschreiben. In dieser Arbeit wird ein Algorithmus vorgestellt, der versucht, auf Grundlage von Constraint Satisfaction Algorithmen, ein beliebiges Minesweeper Spiel zu lösen.

2 Algorithmus

Definition

Um ein Constraint Satisfaction Problem formal beschreiben zu können, müssen eine Menge von Variablen, die Menge ihrer Wertebereiche und die Definition der Constraints festgelegt werden. Für Minesweeper habe ich dies wie folgt definiert: Für ein Spielfeld mit $n \in \mathbb{N}$ Spalten und $m \in \mathbb{N}$ Zeilen sind die Variablen $X = \{(x, y) | 2 \leq x \leq n \text{ und } 2 \leq y \leq m\}$ und ihre Wertebereiche $D = \{0, 1\}$, wobei 1 für eine Mine steht und 0 für ein sicheres Feld. Die Constraints müssen umfangreicher definiert werden. Für jede Variable (x, y) gibt es den Constraint auf ihm und alle seiner Nachbarn, dass der Wert auf dem Feld der Variablen, desweiteren als Konstante $k(x, y)$ mit $k \in \mathbb{N}$ benannt, gleich der Summe aller Werte der Nachbarsvariablen ist. Formal definiert wäre dies $C = \{C(x, y) | k(x, y) = \sum_{\substack{0 \leq a \leq n-1 \\ 0 < b < m-1}} (a, b) \text{ mit } (a, b) \in N_G((x, y))\}$. Binäre Constraints zwischen zwei benachbarten Variablen ergeben sich dann dadurch, dass eine Variable keinen Wert annehmen kann, welcher den Constraint des Nachbars verletzen würde. Diese

sind Teilschritte dahin, den eigentlichen Constraint erfüllen zu können. Der Constraint-Graph ergibt sich für dieses Spiel trivial als Graph des Spielfelds.

Besonderheiten Minesweeper

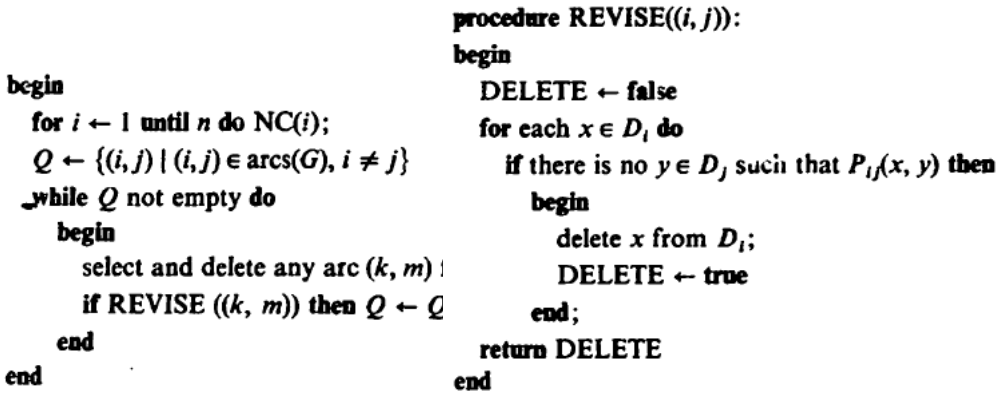
Eine der Besonderheiten, um das Spiel Minesweeper zu lösen zu können, ist das Aufdecken der Felder, womit die Konstante auf diesem Feld bekannt wird und daraus dynamisch Constraints definiert werden können. Daraus ergibt sich, dass der Algorithmus immer wiederholt wird, wenn neue Informationen durch Aufdecken erlangt werden. Mit dieser Art der unvollständigen Information über das Spielfeld unterscheidet es sich deutlich von anderen CSPs wie Sudoku, das Vier-Farben-Problem oder das Damenproblem. Eine weitere Herausforderung ist, dass nicht jedes Spiel mit Sicherheit lösbar ist. Es gibt Fälle, in denen nur mit 50% Wahrscheinlichkeit gesagt werden kann, wo sich eine Mine befindet. Beispiel siehe Abb. A

Ablauf

1. Startpunkt und Aufdecken Die erste Herausforderung zum Lösen dieses Spiels ist die Wahl, welches Feld zuerst aufgedeckt wird, da alle Felder zu Beginn verdeckt sind. In der Windows-Version von Minesweeper ist sichergestellt, dass der erste Klick keine Mine aufdecken kann (Quelle X). Dies habe ich für meine Implementierung des Spiels auch übernommen, aus einfachen Komfortgründen. Ist diese Voraussetzung gegeben, kann zu Beginn also einfach ein zufälliges Feld gewählt werden.

Jedes Mal, wenn ein Feld aufgedeckt wird und es keine Mine ist, kann der Wertebereich der Variable für das Feld auf $D = \{0\}$ reduziert, damit auch der Wert auf 0 gesetzt und die binären Constraints dieser Variable zu allen Nachbarn in beide Richtungen definiert werden. Als Heuristik habe ich hier zusätzlich eingebaut, dass falls die aufgedeckte Konstante eine 0 ist, direkt alle Nachbarn aufgedeckt werden können (rekursiv für weitere Nullen). Damit werden die trivialen Fälle der 0 direkt abgehandelt, womit die zugehörigen Variablen bereits konsistent sind.

Im Laufe des Algorithmus ergeben sich weitere sichere Felder, die wie oben beschrieben wieder aufgedeckt werden können.



(a) AC-3 Algorithmus (b) Revise Algorithmus

Abbildung 1: AC-3 + Revise Pseudocode

2. AC-3 und Revise Der Kern der Algorithmen für CSPs stellt der mittlerweile weit verbreitete Algorithmus AC-3 mit Revise dar, wie er von Alan Mackworth in Quelle [y] beschrieben wird. Als Pseudocode beschrieben sind sie wie in den folgenden Abbildungen [x] und [y]:

Der AC-3 Algorithmus beginnt mit Node Consistency ($NC(i)$). Diese ist bereits durch das Aufdecken gegeben und es können sich direkt mit den binären Constraints beschäftigt werden. In meiner Implementierung beinhaltet die Q die erzeugten binären Constraints. Für jede Kante wird dann *Revise* ausgeführt. Gibt es in *Revise* eine Änderung, kann der übrig gebliebene Wert direkt gesetzt werden da der Wertebereich nur aus 0 und 1 besteht. Haben wir hier eine Mine gefunden, kann diese auch für das Spiel direkt markiert (mit einer Flagge) werden. Alle Nachbarn von k werden dann in Q hinzugefügt. Desweiteren findet hier eine Überprüfung statt, ob die Variable (xk, yk) bereits konsistent ist (Referenz Konsistenz). Ist dies der Fall, sind alle benachbarten Felder mit dem Wert 0 sicher und können aufgedeckt werden. Ist Q leer, wird erst geschaut, ob das Spiel vorbei ist, nämlich genau dann wenn eine Mine oder alle Felder, die keine Mine sind, aufgedeckt wurden. In dem Fall sind wir bereits fertig und können beenden. Ansonsten wird geschaut, ob sichere Felder zum Aufdecken hinzugefügt wurden und von vorne begonnen. Sind wir hier durch, aber nicht fertig geht es mit 3. weiter.

Der Revise-Algorithmus schaut sich die Kanten/ binären Constraints an und prüft auf mögliche Verletzung der Constraints. Es wird jeder Wert von i der Kante (i, j) bzw. $((xi, yi), (xj, yj))$ probeweise gesetzt und geschaut, ob dies dazu führt, dass j keinen Wert annehmen kann, der diesen Constraint erfüllen könnte. In dem Fall wird der getestete

Wert für i gelöscht und $DELETE$ auf *false* gesetzt. Der Test auf Verletzung dieses Constraints wird wie folgt durchgeführt: Für jeden Wert im Wertebereich von j wird geschaut, ob dies seinen Constraint zu allen Nachbarn verletzt. Dies kann der Fall sein, wenn für einen Nachbarn zu viele oder zu wenig Minen existieren würden, also die Summe der Werte aller Nachbarn größer der Konstante ist oder die Anzahl der unbestimmten Nachbarn nicht mehr die erforderliche Anzahl an Minen erreichen könnten. Für *Revise* muss beachtet werden, dass wir die Konstante von j bei (i, j) wissen müssen, das Feld also aufgedeckt sein muss. Ist das nicht der Fall, wird diese Kante einfach übersprungen.

Nach 2. und vor 3. kommt in meinem Algorithmus noch ein Zwischenschritt. Da immer wieder zu AC-3 zurückgekehrt wird, kann es sein, dass bereits alle Minen gefunden wurden. Tritt der Fall ein, können die restlichen Felder einfach aufgedeckt und der *Solver* vollständig konsistent gemacht werden. Dann ist der Algorithmus ebenfalls abgeschlossen. Ansonsten zu 3.

3. Backtracking Ist keine Lösung durch AC-3 gefunden worden, kommt Backtracking zum Einsatz, wie er auch in Quelle[y] beschrieben wird. Hierbei geht es darum, für alle nicht belegten Variablen alle möglichen Zuweisungen zu generieren, testen und gültige zu sammeln. Schon bei kleinen Wertebereichen führt dies zu einer hohen Komplexität. Im unserem Fall mit einem Wertebereich der Größe 2, liegt die Komplexität bei $\mathcal{O}(2^n)$. Bei einem Minesweeper-Spielfeld für die Schwierigkeit *Beginner* (8x8 mit 10 Minen) nimmt die Komplexität häufig ungewollte Ausmaße an. Um dem entgegenzuwirken, wurden folgende Maßnahmen getroffen:

1. Um nicht alle Möglichkeiten zu generieren und dann zu testen, wird vorzeitig abgebrochen, falls eine Zuweisung nicht gültig sein kann. Dies tritt beispielsweise ein, wenn die Summe der Werte in der Zuweisung größer als die Anzahl der übrigen Minen ist. Umgesetzt wird dies rekursiv aus Effizienz- und Lesbarkeitsgründen.

2. Da viele Berechnungen durchgeführt werden, bei Tests auf Gültigkeit, wird *Memoization* eingesetzt, also eine Art Cache eingeführt, für bereits berechnete Lösungen.

3. Um die Komplexität weiter zu reduzieren, werden außerdem Heuristiken festgelegt und genutzt. Da es regelmäßig vorkommt, dass ein Großteil der unbelegten Variablen keine aufgedeckten Nachbarn haben und damit wenig über die Gültigkeit einer Zuweisung dieser Variablen gesagt werden kann, werden diese exkludiert. In die Menge der Variablen, die für Backtracking angeschaut werden, kommen also nur

solche, welche mind. einen Nachbar besitzen, der aufgedeckt ist. Desweiteren werden davon nur maximal 10 Variablen genommen, um die Komplexität auf $\mathcal{O}(2^{10})$ zu begrenzen.

Wie werden Zuweisungen nun auf Gültigkeit geprüft?

Literatur

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original