UNIVERSITÄT BASEL

CSP- and SAT-based Inference Techniques Applied to Gnomine

Bachelor Thesis

Faculty of Science, University of Basel
Department of Computer Science
Artificial Intelligence
ai.cs.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Gabriele Röger

Salomé Simon
salome.simon@stud.unibas.ch
09-057-159

July 13th, 2012

Abstract

CSP- and SAT-based Inference Techniques Applied to Gnomine

In order to understand an algorithm, it is always helpful to have a visualization that shows step for step what the algorithm is doing. Under this presumption this Bachelor project will explain and visualize two AI techniques, Constraint Satisfaction Processing and SAT Backbones, using the game Gnomine as an example.

CSP techniques build up a network of constraints and infer information by propagating through a single or several constraints at a time, reducing the domain of the variables in the constraint(s). SAT Backbone Computations find literals in a propositional formula, which are true in every model of the given formula.

By showing how to apply these algorithms on the problem of solving a Gnomine game I hope to give a better insight on the nature of how the chosen algorithms work.

Table of Contents

1 Introduction

1.1 Goals

Understanding an algorithm just by looking at the plain implementation can be a hard task. In general we would like to have a practical example on which we can follow the algorithm step by step and see which consequences each command has. In this spirit my thesis visualizes different inference techniques used in Artificial Intelligence, in the hope that people not familiar with them will have an easier time comprehending how they work. The two techniques presented are:

1. Constraint Satisfaction Processing using Generalized Arc Consistency
2. Backbone Calculation of propositional formulas

As an example problem, the game Gnomine (version 2.31.92) from GNOME will be used[1]. With its logical nature it offers an easy way to formalize the problem of solving the game, while it is still intuitive to understand on a non-formal level since it is well-known.

The purpose of this document is to offer a description how the chosen algorithms work, how our Gnomine problem can be formulated in the given formal setting, and to present the results of my implementation and visualization. Additionally I will suggest possible improvements on the current implementation. A discussion of the the advantages and drawbacks of the chosen approaches concludes the thesis.

1.2 Gnomine

Gnomine is a clone of the well known game Minesweeper which was developed for Microsoft Windows. It is a logic game consisting of an area of fields where a given amount of fields have mines on them, hidden to the player. The goal of this game is detecting all mines and opening all fields where there is no mine, without opening a field that has a mine. The only

[1] `http://ftp.gnome.org/pub/GNOME/sources/gnome-games/2.31/`

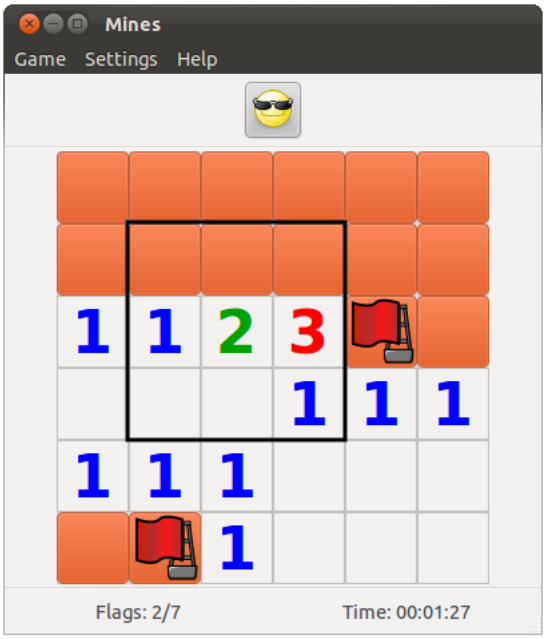

Figure 1.1: The game Gnomine

hint given while progressing in the game are numbers shown at opened fields: they indicate, how many of the neighboring fields have mines on them. In Figure 1.1 we see a screenshot of the game, where one field and its neighboring fields are highlighted with a black square. The number 2 in the middle tells us, that in the surrounding highlighted fields there are 2 mines. However, the fields with a gray background are already opened and therefore defined as no mine. The flags mark fields where the player suspects a mine.
An interesting fact about Gnomine/Minesweeper: The question whether a given state of a Minefield is consistent with the game rules is an NP hard problem[2] meaning that one cannot determine in polynomial time whether a given minefield can originate from a valid Minesweeper game. However, this problem will not be addressed in this thesis.

[2] `http://www.claymath.org/Popular_Lectures/Minesweeper/`

2

Constraint Satisfaction Problems

One way to look at Gnomine is seeing it as a set of constrains over the different fields, e.g., if you have the number 2 written in a field, the sum of mines in the surrounding fields must be 2. We use the following notations:

- $f_{(i,j)}$ denotes the field at the position (i, j) in our game
- $x_{(i,j)}$ denotes the value of field $f_{(i,j)}$ from the domain $\{0, 1\}$, where 0 corresponds to no mine and 1 to mine
- $n_{(i,j)}$ as the number written in $f_{(i,j)}$

With this definition we dervie for each uncovered field $f_{a,b}$ the condition:

$$x_{(a-1,b-1)} + x_{(a-1,b)} + x_{(a-1,b+1)} + x_{(a,b-1)} + x_{(a,b+1)} + x_{(a+1,b-1)} + x_{(a+1,b)} + x_{(a+1,b+1)} = n_{(a,b)}.$$

If we know that the field $f_{(i,j)}$ has no mine (e.g., if we opened it up already), then we can shorten the domain to $D_{(i,j)} = \{0\}$. In the same manner we can shorten the domain $D_{(i,j)}$ to $\{1\}$ if we know the field $f_{(i,j)}$ has a mine (e.g., if we set a mine flag).

2.1 Constraint Networks

In order to describe our Gnomine problem more formally, we first need to define what a Constraint Network and the resulting Constraint Satisfaction Problem is:

Definition 1 (Constraint network)**.** *A constraint network is given as a three-tuple* $\mathcal{R} = (X, D, C)$*, where*

- $X := \{x_1, x_2, \ldots, x_n\}$ *variables of the network*
- $D := \{D_1, D_2, \ldots, D_n\}$ *domain of the corresponding variable*
- $C := \{C_1, \ldots, C_k\}$ *the set of constraints, defined by a tuple* (S_i, R_i)*, where*
  - $S_i = (x_{S_1}, \ldots, x_{S_r})$ *is an ordered subset of the variables* X

- R_i *the subset of* $D_{S_1} \times \cdots \times D_{S_r}$ *which defines the constraint*

The Constraint Satisfaction Problem to the given Constraint Network is the task of finding a valid assignment for all variables X within their domain D which satisfies all constraints in C.

2.1.1 Gnomine as Constraint Network

Now we apply this definition to our Gnomine problem. Given a certain game state with the fields $f_{(i,j)}, i \in \{0, \ldots, ysize - 1\}, j \in \{0, \ldots, xsize - 1\}$ we define:

- $X := \{x_{(i,j)} \mid i \in \{-1, 0, \ldots, ysize\}, j \in \{-1, 0, \ldots, xsize\}\}$ the values in the fields. *ysize* denotes the rows in the game area, *xsize* correspondingly the columns. We pad the game area with one field on top, bottom, left and right. As seen below this will allow us to define the constraints easier.
- $D := \{D_{(i,j)} \mid i \in \{-1, 0, \ldots, ysize\}, j \in \{-1, 0, \ldots, xsize\}\}$

$$D_{(i,j)} = \begin{cases} \{0\} & \forall i, j \text{ if } i = -1 \text{ or } i = ysize \text{ or } j = -1 \text{ or } j = xsize \\ \{0\} & \forall i, j \text{ if the corresponding field is uncovered} \\ \{1\} & \forall i, j \text{ if the corresponding field is flagged} \\ \{0, 1\} & \text{otherwise} \end{cases}$$

- $C := \{C_{(i,j)} \mid D_{(i,j)} = \{0\} \text{ and } i \in \{0, \ldots, ysize\text{-}1\}, j \in \{0, \ldots, xsize\text{-}1\}\}$. We can only build constraints from fields that are opened and give us information on how many neighbors are mined.
  - $S_{(i,j)} = (x_{(i-1,j-1)}, x_{(i-1,j)}, x_{(i-1,j+1)}, x_{(i,j-1)}, x_{(i,j+1)}, x_{(i+1,j-1)}, x_{(i+1,j)}, x_{(i+1,j+1)})$
  - $R_{(i,j)} = \{(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7) \in D_{(i-1,j-1)} \times D_{(i-1,j)} \times D_{(i-1,j+1)} \times D_{(i,j-1)} \times D_{(i,j+1)} \times D_{(i+1,j-1)} \times D_{(i+1,j)} \times D_{(i+1,j+1)} \mid \sum_{j=0}^{7} v_j = n_{(i,j)}\}$ where $n_{(i,j)}$ denotes the number written in $f_{(i,j)}$

For further reference we will use a shortened notation for constraints, since it is more intuitive but could easily be replaced with the more formal definition:

$$C_{(a,b)} := \Big(x_{(a-1,b-1)} + x_{(a-1,b)} + x_{(a-1,b+1)} + x_{(a,b-1)} + x_{(a,b+1)} + x_{(a+1,b-1)} + x_{(a+1,b)} + x_{(a+1,b+1)} = n_{ab}\Big)$$

Some additional notes on constraints in Gnomine:

- The variables in the padded area do not denote fields. These variables have a fixed domain of $\{0\}$, because then we can build the constraint in the same way as if those variables would denote fields.
- For fields where all neighbors are unambiguously defined (as either mine or no mine), the constraint still exists but does not offer any information gain. Therefore, those constraints will be omitted in future considerations.
- We can only build constraints for fields that are uncovered on the map already (fields that are defined as no mine), because those fields are the only ones where we can obtain the number of mines in the surrounding fields.

Figure 2.1: A small Gnomine problem

Example 2.1.1. *Let us analyze a very small Gnomine problem and define all (relevant) constraints in the short notation. The cutout is seen in Figure 2.1*
Variables

$$x_{(-1,-1)}, x_{(-1,0)}, x_{(-1,1)}, x_{(-1,2)}, x_{(-1,3)}, x_{(0,-1)}, x_{(0,0)}, x_{(0,1)}, x_{(0,2)}, x_{(0,3)},$$
$$x_{(1,-1)}, x_{(1,0)}, x_{(1,1)}, x_{(1,2)}, x_{(1,3)}, x_{(2,-1)}, x_{(2,0)}, x_{(2,1)}, x_{(2,2)}, x_{(2,3)}, x_{(3,-1)},$$
$$x_{(3,0)}, x_{(3,1)}, x_{(3,2)}, x_{(3,3)}$$

Domains

$$\begin{aligned}\{0\} :& D_{(-1,-1)}, D_{(-1,0)}, D_{(-1,1)}, D_{(-1,2)}, D_{(-1,3)}, D_{(0,-1)}, D_{(0,3)}, D_{(1,-1)},\\ & D_{(1,3)}, D_{(2,-1)}, D_{(2,3)}, D_{(3,-1)}, D_{(3,0)}, D_{(3,1)}, D_{(3,2)}, D_{(3,3)},\\ & D_{(0,0)}, D_{(0,1)}, D_{(1,0)}, D_{(1,1)}\\ \{1\} :& D_{(1,2)}\\ \{0,1\} :& D_{(0,2)}, D_{(2,0)}, D_{(2,1)}, D_{(2,2)}\end{aligned}$$

Constraints

$$\begin{aligned}C_{(0,1)} &:= \big(x_{(-1,0)} + x_{(-1,1)} + x_{(-1,2)} + x_{(0,0)} + x_{(0,2)} + x_{(1,0)} + x_{(1,1)} + x_{(1,2)} = 1\big)\\ C_{(1,0)} &:= \big(x_{(0,-1)} + x_{(0,0)} + x_{(0,1)} + x_{(1,-1)} + x_{(1,1)} + x_{(2,-1)} + x_{(2,0)} + x_{(2,1)} = 1\big)\\ C_{(1,1)} &:= \big(x_{(0,0)} + x_{(0,1)} + x_{(0,2)} + x_{(1,0)} + x_{(1,2)} + x_{(2,0)} + x_{(2,1)} + x_{(2,2)} = 2\big)\end{aligned}$$

2.2 Arc Consistency

A common approach to solve Constraint Satisfaction Problems is reducing the variable domains with Constraint Propagation. There are several methods that detect different classes of inconsistency. The one I am using in my project is called *arc consistency*.

Arc consistency is usually defined over binary constraints (meaning that the constraint only concerns two variables):

Definition 2 (Arc Consistency (Dechter, 2003, p.54)). *A variable x_i is* arc-consistent *relative to x_j if and only if for every value $a_i \in D_i$ there exists a value $a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$ ($R_{ij} \in C$) [where R_{ij} belongs to the constraint with $S_{ij} = (x_i, x_j)$*

Since the constraints in Gnomine are not binary, we need to use a more general understanding of arc consistency.

2.2.1 Generalized Arc Consistency

Definition 3 (Generalized Arc Consistency (Dechter, 2003, p.71)). *Given a constraint network $\mathcal{R} = (X, D, C)$, with $R_S \in C$, a variable x is arc-consistent relative to R_S if and only if for every value $a \in D_x$ there exists a tuple, in the domain of variables in $S, t \in R_S$ such that $t[x] = a$. t can be called a support for a. The constraint R_S is called arc-consistent iff it is arc-consistent relative to each of the variables in its scope. A constraint network is arc-consistent if all its constraints are arc-consistent.*

With this definition, Dechter (2003, p.71) derived a fomula which eliminates all values in a given domain D_x which are not a support for x:

$$D_x \leftarrow D_x \cap \pi_x(R_S \bowtie D_{S-(x)}) \tag{2.1}$$

The operators π and $\bowtie$ have the usual meaning known from relational algebra. π_A denotes a projection of a set of attributes to the subset A, in this case a projection of the ordered set of variables in S to the single variable x. $X \bowtie Y$ stands for the natural join of two ordered sets of tuples X and Y, which means a selection of the Cartesian product where the common attributes in X, Y must have the same value.
We define our new domain D_x as the intersection between the so far existing domain D_x and the join of all possible solutions with all domains except D_x, projected on the variable x. The join between R_S and $D_{S-(x)}$ eliminates those models from the set of possible solutions which contradict a given domain.

Applied to our Gnomine game: A field is arc-consistent relative to a constraint, if for each value within the fields domain there exists an assignment of the remaining variables which satisfies the constraint. While propagating we can reduce the domains, meaning we can crop a value v from the domain $D_{(i,j)}$ if we cannot find a valid assignment for the given constraint where $x_{(i,j)} = v$.

Example 2.2.1. *Consider the following constraint:*
$C_{(1,1)} := x_{(0,0)} + x_{(0,1)} + x_{(0,2)} + x_{(1,0)} + x_{(1,2)} + x_{(2,0)} + x_{(2,1)} + x_{(2,2)} = 3$
$D_{(0,0)} := D_{(0,1)} = D_{(1,0)} = D_{(2,0)} = D_{(2,2)} = 0$
$D_{(2,1)} := 1$
$D_{(0,2)} := D_{(1,2)} = \{0, 1\}$
When testing $x_{(1,2)}$ for the value $v = 0$ we will not be able to find a model that satisfies the constraint anymore. Thus we can reduce $D_{(1,2)}$ to $D_{(1,2)} - \{v\} = \{0, 1\} - \{0\} = \{1\}$.

A special characteristics of Gnomine is that the variable-domains are binary and that a constraint network built from a Gnomine game is always satisfiable (if no wrong flags are set). Therefore, if we can eliminate a value from a domain, the variable has only a domain size of 1 which means it is unambiguously defined (since it must take a valid value, otherwise the constraint network would not be satisfiable).

2.3 Implementation

First off I needed to decide on how to represent the constraints in a program. The Gnomine code stores a variable for each field that denotes how many neighbors are mines. Therefore I decided not to store the whole constraint, but merely keep an array consisting of the field-numbers which have relevant constraints (where relevant means that some neighbors are not defined yet). The exact information about the constraint (which neighbors are affected, which domain do they have, what value does $n_{(a,b)}$ have) can always be read from the game state.
The domains of the corresponding fields are saved in a separate integer array in a simplified manner: If the domain consists of only 1 value, its value (0 or 1) is stored in the corresponding field, if the domain equals $\{0, 1\}$, -1 is stored.
Since the fields in Gnomine are stored as one-dimensional array, I chose to store the constraints and domains in the same manner. The field (i, j) is stored in the array at position $i \cdot xsize + j$.

For testing a constraint on arc-consistency all concerned variables need to be tested whether for all elements of their respective domain there exists an assignment of the other variables which satisfies the constraint. For variables that have a defined value, this test is not necessary. This means, the next step consists of finding all neighbors of the current constraint-field that are still undefined (in Gnomine the orange fields) and testing them on the above criteria.

Each variable $x_{(i,j)}$ that needs to be tested has the domain $\{0, 1\}$. My algorithm always keeps an array of 8 integers as current constraint. The values possible in this array are 0 for no mine, 1 for mine and -1 for undefined. When testing a variable the array index of the currently tested variable is saved and its value is first set on 0. Then it loops over all 2^{un} possible assignments (where *un* denotes the number of uncertain neighbors without the one currently tested) and tests whether the current assignment is valid. If so, it breaks out of the loop and continues with arr[currentVariable] = 1, where it again tests all possible assignments until either one is found or the loop is finished. If, in the end, one assignment for the current variable is valid while the other is not, the corresponding entry in the domain-array is set on the valid value (instead of -1).
In the loop itself the loop-counter is used to encode the current assignment. Let's assume we have the array arr = [0,0,-1,1,0,-1,0,0]. In this case the counter would go from 0 to 3. Binary encoded this would be 00, 01, 10, and 11. These values are inserted into the fields that have a -1 as value, e.g., for loop-counter = 0, we would alter the array to arr =[0,0,**0**,1,0,**0**,0,0], for loop-counter = 2 the array would be arr=[0,0,**1**,1,0,**0**,0,0]. Then the sum of all array entries is calculated, and if it equals $n_{(a,b)}$ in the current constraint, the assignment is valid.

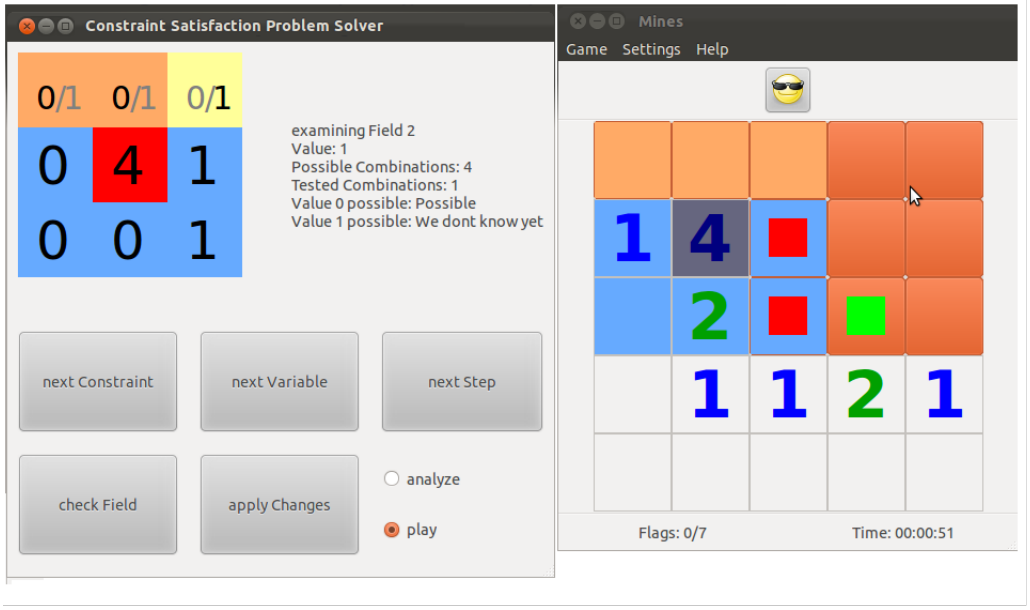

Figure 2.2: Visualization of the CSP algorithm

Let us look at an example in detail. We have a Constraint $C_{(3,4)} := x_{(2,3)} + x_{(2,4)} + x_{(2,5)} + x_{(3,3)} + x_{(3,5)} + x_{(4,3)} + x_{(4,4)} + x_{(4,5)} = 4$. We know that $D_{(2,3)} = D_{(3,3)} = D_{(4,3)} = D_{(4,4)} = 0$ and $D_{(2,5)} = 1$. Our current constraint-array would be: arr = [0,-1,1,0,-1,0,0,-1]. We want to test variable $x_{(2,4)} = 0$, therefore we alter the array to arr = [0,**0**,1,0,-1,0,0,-1]. Now we need to loop over all 4 possible assignment for the remaining variables $x_{(3,5)}$ and $x_{(4,5)}$. The resulting arrays would look the following: [[0,0,1,0,**0**,0,0,**0**]; [0,0,1,0,**0**,0,0,**1**]; [0,0,1,0,**1**,0,0,**0**]; [0,0,1,0,**1**,0,0,**1**]]. The corresponding sums over all array entries are: [1, 2, 2, 3]. As we see, no assignment satisfies our constraint; therefore we know that $x_{(2,4)} \neq 0$. If we make the same test for $x_{(2,4)} = 1$, we would find a valid assignment. Therefore we can say for sure that $x_{(2,4)} = 1$.

2.4 Visualization

The GUI of the CSP-Solver consists of 3 parts: a drawing area where the current constraint is visualized, a text area where some additional information is shown, and a button area where we can interact with the program. Additionally, if the solver can determine for a field whether it is a mine or not, this information will be shown on the actual minefield: if it is a mine, a red square will appear on the corresponding field, if it is not a mine, a green square.

Drawing Area The current constraint is marked both in the solver GUI and on the actual Gnomine minefield, where the current constraint field has a dark blue background, known neighbor fields a lighter blue, and unknown neighbor fields an orange one. By applying this format to the original minefield, we always know which constraint we are looking at without the need of painting the whole minefield in the solver GUI as well. The number in the middle field stands for the $n_{(a,b)}$ in the current constraint $C_{(a,b)}$, the numbers in the neighboring

fields stand for the current domain of the variables corresponding to the field.
While analyzing a constraint, the currently tested variable has a yellow background. In all fields that have a domain of {0,1}, the current assigned number is highlighted black, while the other one is gray. If the constraint is satisfied with the current assignment, the background of the middle field turns green, otherwise red.

Text Area The text area displays information like which constraint and which variable we are testing at the moment, tells us how many possible assignments there are, and how many of these we have tested already. It also saves whether value 0 respectively 1 has a support or not, or if we did not finish testing it yet. Therefore we always know where in the algorithm we are.

Button Area In the button area we get several options on how to interact with the solver. We can tell it to jump to the next constraint (from top left to bottom down), to the next variable in this constraint, or to execute the next step. We can also check the entire field, and apply the changes saved in the solver onto the actual minefield, meaning that all fields where the solver found out whether it is a mine or not, will either be opened up or marked with a flag. Finally we can switch between playing normally on the actual minefield or analyzing the field; where clicking on uncovered field with a number in analyze mode makes the algorithm jump to this constraint.

2.5 Possible Improvements

The current implementation of the CSP solver only detects inconsistencies within a single constraint. In our Gnomine problem with binary domains this also means that we can only gain information from constraints where either all still undefined fields are mines or all are no mines, because if some of the fields were mines and others were not we could not tell which ones are which.
For future work, it might be interesting to look at a different kind of consistency which takes more into consideration than one constraint at a time, e.g., path consistency.

2.5.1 Path Consistency

Definition 4 (Path Consistency (Dechter, 2003, p.62))**.** *Given a constraint network $\mathcal{R} = (X, D, C)$, a two-variable set (x_i, x_j) is path consistent relative to variable x_k if and only if for every consistent assignment $((x_i, a_i), (x_j, a_j))$ there is a value $a_k \in D_k$ such that the assignment $((x_i, a_i), (x_k, a_k))$ is consistent and $((x_k, a_k), (x_j, a_j))$ is consistent. Alternatively, a binary constraint R_{ij} is path-consistent relative to x_k iff for every pair $(a_i, a_j) \in R_{ij}$ where a_i and a_j are from their respective domains, there is a value $a_k \in D_k$ such that $(a_i, a_k) \in R_{ik}$ and $(a_k, a_j) \in R_{kj}$. A subnetwork over three variables $\{x_i, x_j, x_k\}$ is path-consistent iff for every R_{ij} (including universal binary relations) and for every $k \neq i, j$ R_{ij} is path-consistent relative to x_k.*

Again, we would need to generalize this level of consistency in order to widen it up to non-binary constraints. Path consistency basically dictates, that if you have a valid assignment for a constraint C_a, all constraints C_b that share the variable $x_i, ..., x_k$ must be consistent

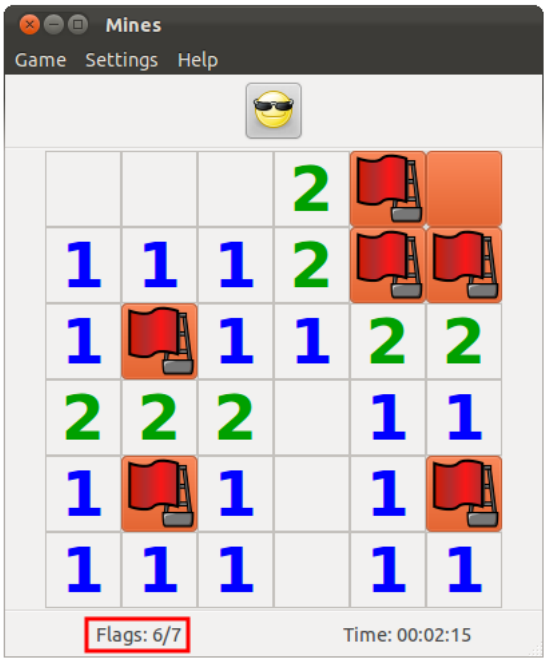

Figure 2.3: A situation where the total amount of mines is a helpful constraint

given the assignment $((x_i, a_i), ..., (x_k, a_k))$. However, a formal definition of generalized arc consistency would be a task for future work.

2.5.2 Total amount of mines

One constraint that is not considered in our formalization of the Gnomine problem is the total amount of mines, an information given by Gnomine. If we define n_T as the total amount of mines, the simplified formulation for this constraint would be:

$$\sum_{i=0}^{ysize-1} \sum_{j=0}^{xsize-1} x_{ij} = n_T \tag{2.2}$$

This constraint can help us in cases, where a field is completely surrounded by mines. On those fields we can get no information from local constraints, because constraints can only be build from uncovered fields and only concern their direct neighbors. One such case is shown in Figure 2.3

3
Backbones

Backbone computation is an inference technique used in the area of SAT-problems. It calculates for a given propositional formula which literals must be true in every model. If we can describe our Gnomine problem in such a propositional formula, this approach will be very helpful to solve our problem. Finding only one model that satisfies the given problem will not be enough for Gnomine: In most cases there are different ways of filling up the game area with the information given in a certain game state, and we cannot say for sure which assignment is the right one. One of those situations is shown in Figure 3.1.
However, if we know that a certain field gets assigned the same value in all possible models, then (and only then) we can be certain that this value is the right value for this field. Hence, the backbone calculation returns all assignments that can be concluded logically from the given game state.

Figure 3.1: A situation with multiple solutions

3.1 Propositional Satisfiability Problem and SAT solver

A Propositional Satisfiability Problem consists of a propositional formula and the task of finding a complete assignment to all variables such that it satisfies the formula. Such an assignment is called a model. This task is in general a NP-hard problem; however, depending on the given formula, SAT solvers can find solutions in reasonable time.
A propositional formula consists of Boolean variables and Boolean operations, such as NOT, AND and OR. No matter how complex, all propositional formulas can be converted in both *Conjunctive Normal Form (CNF)* and *Disjunctive Normal Form (DNF)*, where CNF is a conjunction of disjunctions (e.g., $[(X_{00} \vee X_{01}) \wedge (X_{10} \vee X_{11} \vee X_{12})]$); and DNF is a disjunction of conjunctions (e.g., $[(X_{00} \wedge X_{01} \wedge X_{02}) \vee (X_{10})]$). This conversion may however result in an exponential increase of the formula size.

The SAT solver used in my program is Minisat [3] (Eèn and Sörensson, 2004). The algorithm alternates between assigning a truth value to a literal, and simplifying the formula using unit propagation until either a model or a contradiction is found. If a model is found, it is stored in a class variable of the solver for easy access. In case of contradiction it backtracks to the last variable set and sets it on the negation of this value (or continues upwards if both true and false have been tested already). If the algorithm backtracks to the top-level and finds a contradiction, it returns unsatisfiable.
Minisat requires the input to be in CNF. It is built by successively adding clauses to the Minisat instance. Clauses are a disjunction of literals, thus a CNF can be described by a conjunction of clauses.
Minisat is also able to solve a SAT-problem under a given assumption without adding the assumption to the clause database. We will see below that this will be very useful for the backbone computation.

3.1.1 Gnomine as SAT-Problem

At this point the question arises how to describe Gnomine as as propositional formula. Additionally, we also need to consider that this formula will need to be in CNF. My implementation uses the following approach:

The variables $x_{(i,j)}$ correpsond to the value on the fields of the minefield area. The literal $x_{(i,j)}$ denotes that the variable $x_{(i,j)}$ is *true*, meaning the field on position (i, j) is a mine, $\neg x_{(i,j)}$ denotes that $x_{(i,j)}$ is *false*, meaning the field is no mine.
The information which is added easiest to the formula are the fields that have a defined value. If we know field (i, j) has no mine, we can add the clause $(\neg x_{(i,j)})$. Correspondingly, if the field (i, j) is flagged, we add the clause $(x_{(i,j)})$
The information given by the number $n_{(i,j)}$ shown on an open field $f_{(i,j)}$ that tells us how many neighbors are mined is harder to describe. I chose the option of building a truth table for all neighbors that are still undefined. The result column is true, if the corresponding assignment of the undefined neighbors would satisfy the neighbor constraint, false otherwise.

[3] `http://www.minisat.se/`

Figure 3.2: A cutout from Gnomine

$x_{(0,0)}$	$x_{(0,1)}$	$x_{(0,2)}$	$x_{(1,0)}$	$x_{(1,2)}$	$x_{(2,0)}$	$x_{(2,1)}$	$x_{(2,2)}$	**res**
0	0	0	0	1	0	**0**	**0**	**false**
0	0	0	0	1	0	**0**	**1**	**false**
0	0	0	0	1	0	**1**	**0**	**false**
0	0	0	0	1	0	**1**	**1**	**true**

Table 3.1: Truth table for the Gnomine constraint shown in Figure 3.2

To build CNF clauses from that, one needs to take the assignments where the result column is false as a conjunction of literals, negate the expression and add it as clause to the formula.

Example 3.1.1. *Let us look at Figure 3.2 and Table 3.1. The fields represented by the first 6 columns in the table are already defined as mine (flag) or no mine (uncovered). Therefore we can add the following clauses:*

$$(\neg x_{(0,0)}), (\neg x_{(0,1)}), (\neg x_{(0,2)}), (\neg x_{(1,0)}), (x_{(1,2)}), (\neg x_{(2,0)})$$

The clauses we can build from the truth table are the following:

$$(x_{(2,1)} \vee x_{(2,2)}), (x_{(2,1)} \vee \neg x_{(2,2)}), (\neg x_{(2,1)} \vee x_{(2,2)})$$

We could also add the information about the first 6 fields into those clauses. Then the first (negated) clause would be: $(x_{(0,0)} \vee x_{(0,1)} \vee x_{(0,2)} \vee x_{(1,0)} \vee \neg x_{(1,2)} \vee x_{(2,0)} \vee x_{(2,1)} \vee x_{(2,2)})$. *We know already that the first 6 literals are false (concluding from the the unit clauses above). Therefore we can as well take them out of the clause, leaving a shortened version as it was described above.*

3.2 Backbone Computation

After describing our problem as a propositional formula, the next step consists of finding the backbones of the given formula. Marques-Silva et al. (2010) distinguished two procedures on how to calculate backbones: enumeration-based and iterative.

Enumeration Based Approach The enumeration based algorithm first finds a model I which leads to the initial backbones candidates: $b = \{l \mid l \text{ literal}, I \models l\}$. These candidates are a superset of the actual backbones. Then it searches for additional models while blocking already found models by adding a blocking clause to the formula for each model found.

Each time a new model I' is found, the existing backbone estimate gets cropped with: $b := \{l \mid l \in b \text{ and } I' \models l\}$ (only those literals are kept in the backbone estimate which are true in the new model). E.g., if $I = \{a \mapsto \text{false}, b \mapsto \text{true}, c \mapsto \text{false}\}$ and $I' = \{a \mapsto \text{false}, b \mapsto \text{false}, c \mapsto \text{true}\}$, $b = \{\neg a, c\}$. This procedure is repeated until no more solution exists. The resulting b is the exact set of backbones.

Iterative Approach Iterative algorithms test for each variable a whether the literal a or $\neg a$ is part of the backbone. The simple way to do so is test for every variable whether the formula is satisfiable with the clause a or $\neg a$. If one test is positive, the other negative, the literal that was assumed during the positive test is a backbone. However, this is a costly algorithm and can be improved.
Marques-Silva et al. (2010) developed Algorithm 1 for calculating the backbones of a formula in an iterative approach. First an initial model for the formula is found. The only literals that need to be tested now are the negations of the literals that are true in the model. Those literals are saved in a list Λ. E.g., if we have a formula with variables $\{a, b, c\}$ and the first model found returns $\{a \mapsto \text{true}, b \mapsto \text{false}, c \mapsto \text{false}\}$, the only literals we still need to test and that are therefore saved in Λ are $\{\neg a, b, c\}$, the negations of the backbone candidates $\{a, \neg b, \neg c\}$.
Most modern SAT solver like Minisat often do not return a complete model that satisfies the formula, if possible they return a partial assignment that already implies that the formula is true. This means that (given the partial assignment) the variables where the solver does not return an assignment can be assigned either true or false, and the formula is still true. Thus, if the solver returns $\{a \mapsto \text{true}, d \mapsto \text{false}, e \mapsto \text{false}\}$ when given a formula with the variables $\{a, b, c, d, e\}$, we know already that the literals b, $\neg b$, c and $\neg c$ cannot be backbones of this formula, and Λ will be initialized with $\{\neg a, d, e\}$.
If for either of those literals l the test returns unsatisfiable, we know that $\neg l$ is a backbone. But even if the formula is still satisfiable with the clause $\{l\}$ we can gain some information from the newly found partial assignment:

1. If a variable a is not assigned in the new partial assignment but one of the literals a or $\neg a$ is in Λ, we can remove this literal from Λ since we know that variable a can be assigned different values.

2. If an assignment for variable a is found which makes a literal l in Λ true, we can take this literal away, since we know now that the formula is still satisfiable with the clause l.

3.3 Visualization

The main part of the solver GUI is a copy of the minefield area. The uncovered fields are light gray, the flagged fields red, the (on the original game) undefined fields orange. The currently tested field is highlighted in bright yellow.
While progressing through the algorithm, the discovered backbones are shown as a green square if there is no mine on the field, or red if there is a mine. Fields that are confirmed to be no backbone have a gray square.
For fields that are neither defined on the original game nor have a backbone have a black

```
Input: CNF formula φ, with variables X
Result: Backbone ν_R
(sat,ν) ← SAT(φ);
if sat = false then
  | return ∅;
end
Λ ← {l | l̄ ∈ ν};
ν_R = ∅;
foreach l ∈ Λ do
  | (sat,ν) ← SAT(φ ∪ {l});
  | if sat = false then
  |   | ν_R ← ν_R ∪ {l̄};
  |   | φ ← φ ∪ {l̄};
  | else
  |   | foreach x ∈ X do
  |   |   | if x ∉ ν ∧ x̄ ∉ ν then
  |   |   |   | Λ ← Λ − {x, x̄}
  |   |   | end
  |   | end
  |   | foreach l_ν ∈ ν do
  |   |   | if l_ν ∈ Λ then
  |   |   |   | Λ ← Λ − {l_ν}
  |   |   | end
  |   | end
  | end
end
return ν_R
```

Algorithm 1: Optimized iterative Algorithm for calculating Backbones

Figure 3.3: Visualization of the Backbone algorithm

diagonal line and two small squares that are either green, red or gray. Those small squares denote the two last found partial assignments. Again, green means no mine, red mine and gray if the field was unassigned. Thus we can see the effect if we find two partial assignments that have different assignments for the same field: As seen in Figure 3.3 the two models found different assignments for the fields 0,2,4 and 5, which means those fields cannot have a backbone. We also see if a field did not get assigned in a model and therefore cannot have a backbone.
If for the currently tested field the tested literal yields unsatisfiable, the small squares denoting the second assignment will be black to symbolize that no assignment is possible.

In the button area we have buttons to perform the next step of the algorithm, test the whole field on backbones and to apply the found backbones to the actual minefield. Also we can switch between a play and analyze mode. In the analyze mode, we can click on a unknown field on the actual minefield area and the algorithm will test this field on backbones.

3.4 Possible Improvements

One important information we left out when describing our Gnomine Problem is the total amount of mines. As shown in the last chapter this information can help us find assignments to fields that we could not conclude otherwise.
We could use the same approach as above: building a truth table of all possible assignments and add the negation of all assignments that result in false as clauses to our solver. Since we would not only have 8 variables but the entire minefield with m unassigned fields, the

truth table would take the size 2^m, which indicates an exponential increase in complexity. Thus this approach is not applicable to larger minefields.

There are several known encodings of the at-most-k constraint (Bailleux and Boufkhad (2003), Sinz (2005)). Frisch and Giannaros (2010) presented a survey including an empirical evaluation without an exponential increase of clauses; their approach works with additional variables. With this foundation it should be possible to derive an encoding of a *exactly-k* constraint into a propositional formula, and apply it to the Gnomine problem. However, this approach is not considered in my thesis.

4 Conclusions

4.1 Comparison

After implementing and visualizing the two inference techniques arc consistency and SAT backbone computation, it will be interesting to compare them and find out how they differ from each other.

The main difference is the amount of information the algorithms consider at a time. Arc-Consistency only considers one constraint at a time, therefore it only works with local information. Backbones on the other hand are calculated over a formula, that describes the current global game state, so it considers all information we have over the given problem (except the global amount of flags).
As a result, the backbone algorithm is able to conclude everything that can possibly be concluded from the information given, while Arc-Consistency does not resolve all inconsistencies in the given network. Thus Arc-Consistency will often not define as many variable assignments as backbone computation. The only assignments that backbone computation cannot find on the other hand are those assignments, where one has to guess (and situations where information about the global amount of flags is needed to derive an assignment).

The fact that backbone computation describes the global game state also means that the complexity grows. While the CSP solver takes a constant amount of time to infer information from one constraint, the Backbone solver needs to find a partial assignment of a formula that grows linear for each constraint, which can be very time consuming. For usual minefield sizes the algorithm still works in a reasonable time, but for larger benchmarks the difference will be significantly higher. However, in my program this fact will not become apparent since drawing the GUI is the dominant part of computing time.

4.2 Probability based approaches

An important weakness of the presented algorithms is that they cannot handle situations where we need to guess. This is due to the fact that those algorithms infer logically on

the information given to them, and guessing is not a logical conclusion. However, it might be useful in certain situations if the solver can tell you at least which assignment is more probable. A simple approach which could be implemented as an expansion to the Backbone algorithm would be to count the number of models in which the field in question was assigned true and the number of models in which it was assigned false. For this approach to work though we would need to consider each possible partial assignment, which is not necessarily computed in the current backbone implementation.

Bibliography

Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *Principles and Practice of Constraint Programming – CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003.

Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

Niklas Eèn and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

Alan M. Frisch and Paul A. Giannaros. SAT encodings of the at-most-k constraint – some old, some new, some fast, some slow. In *9th International Workshop on Constraint Modelling and Reformulation (ModRef 2010)*, 2010.

Joa Marques-Silva, Mikoláš Janota, and Inês Lynce. On computing backbones of propositional theories. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 15–20. IOS Press, 2010.

Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Principles and Practice of Constraint Programming – CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.