Imperial College London

Department of Computing

Solving optimisation problems and k-SAT with dynamical systems

Rafał R. Szymański

rs2909@doc.ic.ac.uk

Supervisor: Murray Shanahan
Second Marker: Aldo Faisal

Submitted in partial fulfilment of the requirements for the degree of Master of Engineering in Computing (Artificial Intelligence) of Imperial College London.

Project Assets: `http://doc.ic.ac.uk/~rs2909/project.zip`

June 17, 2013

Abstract

Optimisation problems, such as finding solutions to the boolean satisfiability problem (k-SAT), or finding valid itineraries in the travelling salesman problem are a very well known and studied class of problems with existing algorithmic solutions. In this report, we look at a different approach to computational optimisation by using dynamical systems with attractors corresponding to solutions of the optimisation problem - such a representation does not produce solutions faster, but it gives us insight into how combinatorial optimisation can be feasibly done in a neural medium such as the brain. The report concentrates on solving a popular NP-complete problem, boolean k-SAT, by using dynamical systems. We study two previously developed methods, and contribute another alternative system that has the benefit of using bounded variables, and of solving some problems that a different bounded system was not able to solve. We evaluate dynamical system SAT solving by developing a SAT encoding for the logic game *Bridges* and by solving the encodings for different difficulties of *Bridges*. We come to an interesting conclusion regarding escape rates of the dynamical systems and the published difficulty of the game - while "Easy" problems are indeed easier to solve by the dynamical system approach, a "Hard" puzzle is easier to solve than a "Normal" puzzle due to increased constraint density for "Hard" puzzles. In this way, we show that a higher difficulty optimisation problem for humans can actually be easier than a corresponding easier problem when encoded as SAT.

Acknowledgements

I would like to thank my supervisor Prof. Murray Shanahan for his support throughout the project and for giving me the ability to take it in the directions I found most interesting. I thank my parents for their support throughout my university life. I thank Maciek Albin for discussions about the project and for the de-stressing gym sessions. A big thank you to Maciek Albin, Suhaib Sarmad, Jamal Khan, Sam Wong, and all my other friends for making this an enjoyable four years at Imperial.

Contents

Chapter 1

Introduction

Optimisation problems, such as finding solutions to the boolean satisfiability problem (k-SAT), or finding valid itineraries in the travelling salesman problem are a very well known and studied class of problems with existing algorithmic solutions.

In this project, we look at a different approach to computational optimisation by using dynamical systems with attractors corresponding to solutions of the optimisation problem. The motivation behind the research is that while an algorithmic procedure, such as resolution for SAT solving, is perfect for a computer, it is infeasible for it to appear in a system such as the brain that uses neural computation and actually is itself a dynamical system. Moreover, expressing combinatorial optimisation problems as analog dynamical systems in a way parallelizes the search procedure, and makes it possible to construct analog circuits, such as CTRNN's (Continuous Time Recurrent Neural Network) that are specifically suited to solving a specific optimisation problem.

In this project we first study the relevant background on neural computation, present the Hopfield Network model and go over boolean satisfiability and methods of solving k-SAT problems (chapter 2). To introduce the idea of optimisation via dynamical systems, we look at how to represent popular optimisation problems such as N-Rooks, N-Queens, and the Travelling Salesman Problem as dynamical systems with attractors corresponding to solutions, and show how a Hopfield network can be trained to recognise binary images of letters even if the presented image does correspond exactly to the training image (chapter 3).

The main focus of the project is solving Boolean satisfiability (k-SAT) using dynamical systems. We present and evaluate two methods from existing recent literature, and contribute a third method which exhibits different properties to the other two methods (chapter 4).

To evaluate the dynamical systems used for SAT solving, we encode the puzzle

game *Bridges* as a SAT problem and solve the resulting formulas using the dynamical system formulations. We find that one of the three presented methods, while feasible for solving random k-SAT instances, does not work at all for this application to solving *Bridges*. Furthermore, we look at the escape rates of the dynamical system while solving *Bridges* games as published for humans in difficulties "Easy", "Normal", and "Hard". We find that the escape rate for "Easy" games is the highest, indicating the problems are indeed easy, but we also find that games classified "Hard" for humans have a higher escape rate than those classified as "Normal" meaning that "Hard" games are easier for the computer to solve. We conjecture that this is because of the critical clauses-to-variables α ratio for k-SAT solving. "Normal" games exhibit $\alpha \approx 4.3$ which indicates the hardest types of SAT problems. "Hard" games, having more constraints, have this ratio shifted to approximately 4.5 to 5 making the SAT problem easier to solve by a computer. This research and evaluation is presented in chapter 5.

Overall, this area of research is interesting since it presents ways to express combinatorial optimisation problems with huge decision spaces in ways that are potentially solvable by a real dynamical system such as the brain.

Chapter 2

Background

In this chapter we familiarise ourselves with the background information required to understand the concepts and contributions of this project, as well as with what has already been done in this area of research.

The category that this project falls under is computational optimisation and dynamical systems which includes concepts from computer science (such as SAT solving), mathematics (optimisation via Lagrangian multipliers), and dynamical systems [Izh07]. Furthermore, the purpose of this project is to explore how combinatorial optimisation could be done via neural computation. For this reason, we also look at background regarding neurons and neural computation.

2.1 Neurons and the Hopfield model

The Hopfield network, an artificial neural network capable of solving certain optimisation problems, is partially based on real neurons, and for this reason we present here a short introduction to neurons in the human brain, and their simplified realisation in the Hopfield model. Ultimately, converting optimisation problems to dynamical systems is interesting as it allows for the computation to be performed on something with a neural substrate, such as a brain.

2.1.1 Neurons

The basic building block that gives our brains computational capabilities is the neuron. It is a nerve cell present in the CNS (Central Nervous System), most importantly in the brain and the spinal cord. In the brain, there are around 10^{11} neurons interconnected together.

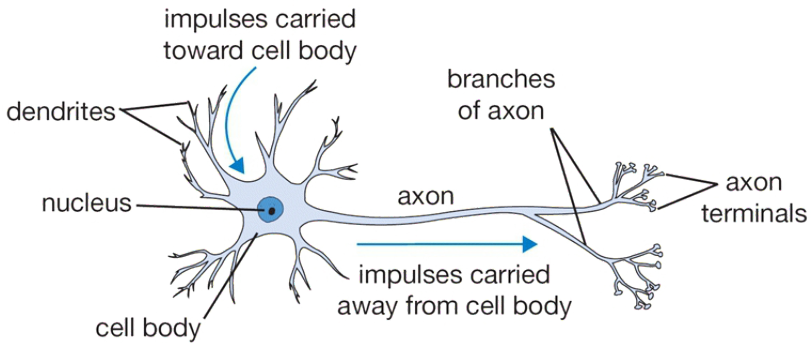


Figure 2.1 – Biological neuron. Adapted from [neu]

Figure 2.1 shows us a standard neuron. The most important parts are the dendrites, cell body, axon, and the axon terminals.

A typical neuron is connected to approximately $10,000$ other neurons via its connections at the dendritic tree, at what we call synapses (See Figure 2.2). In an abstract way, we can think of the whole setup as a directed graph $G = (N, E)$, where N is the set of all neurons, and $E \subseteq N \times N$ are the connections between individual neurons.

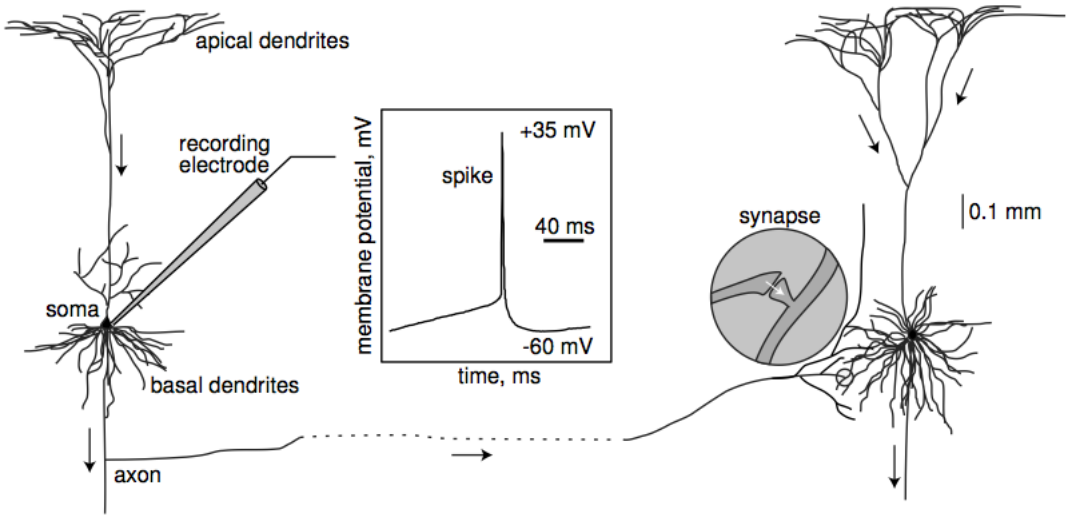


Figure 2.2 – Two interconnected neurons and a figure showing a spike in membrane potential. Adapted from [Izh07]

Spiking

The inputs to a neuron carry electrical current that changes the membrane potential of the neuron. Small currents produce small changes in the membrane potential, while larger currents have the capacity to produce an *action potential*, also called a *spike*. When a spike is emitted at one neuron, the change in voltage

propagates to other connected neurons via the axon and axon terminals which connect to other neurons at their dendrites. Figure 2.2 shows the setup, as well as a chart of a membrane potential over time.

Normally, a neuron's membrane potential is at the resting state. When the input current increases, so does the membrane potential of the neuron, until it passes a certain threshold, and then the neuron quickly depolarises, sending the change in voltage along its axon. The neuron then begins rapidly repolarazing, reaching a membrane potential below its resting potential, and then hyperpolarizing back to its resting potential. After a spike, a neuron can't spike again instantly since it is repolarizing. This period is called the refractory period of the neuron. Spiking is the way in which neurons communicate with each other [Izh07]. A neuron fires as a result of the firing of neurons connected to its dendrites. Additionally, a neuron spontaneously fires by itself at a rate of around 1Hz, which should be taken into account if one is creating realistic models. By connecting different types of neurons together, we can make different types of computational objects, such as an integrator to sum together inputs, or a thresholding unit to threshold the result of a neuron.

A very recent state of the art tool for neural modelling and simulation is Nengo, presented in [ESC+12]. Nengo allows you to create ensembles of neurons and simulate many different mathematical functions. It is for this reason that we present the background on real neurons; they are able to represent many mathematical functions and artificial neural networks are based precisely on real neurons.

Artificial Neuron

The most simplified version of a neuron is a threshold neuron that fires if the sum of its inputs is greater than a certain threshold. This is the model widely used in artificial intelligence in neural networks. The reason is that networks made up of such neurons are much easier to analyse mathematically. Figure 2.3 shows a simple artificial neuron. The neuron fires if $\sum_{i=1}^{n} w_i x_i > T$, where T is a threshold value. This model is insufficient for computational neuroscience since it has no biological plausibility - most importantly it does not exhibit spiking behaviour. Nevertheless, because of its simplicity and ease of analysis, this is the type of neuron used in the Hopfield model, first introduced in [HT85].

2.1.2 Long term memory as an optimisation problem

Donald Hebb was interested in how to the brain is able to form short and long term memories, and he originally presented the idea for a basic mechanism for synaptic plasticity, which can be summarised as *"Neurons that fire together,*

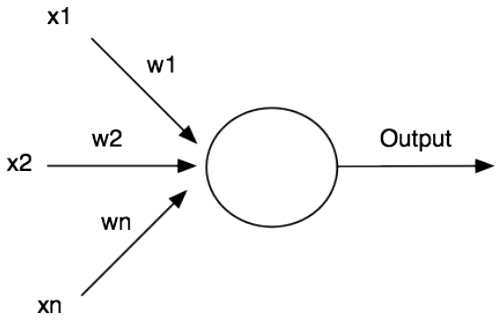


Figure 2.3 – Non-spiking threshold neuron

wire together, neurons that fire out of sync, fail to link".

Hebbian theory has been validated as the likely mechanism underlying long-term memory. Thus, human memory works in an associative, content-addressable way. For example, if you trigger thoughts about dark clouds, due to associativity of our memories, the thoughts of rain will arise. Also, humans are able to recall a memory with more detail, by simply thinking of smaller parts of the memory and its features.

There are many models that use Hebbian learning rules, the most simple one being the Hopfield network, which works like an associative memory. Assume you have a certain pattern $P = \{p_0, p_1 \ldots, p_N\} \in \{-1, 1\}^N$ that you want to store and retrieve later on. After training the network, we can present a stimulus, and the network will converge to one of its fixed point attractors and return the associated result. This represents an optimisation problem - given a Hopfield network and a pattern A, we are interested in the Hopfield network converging to a pattern B such that B is one of the training patterns, and the $d(A, B)$ is minimised, where d is some appropriate metric, for example the Euclidean distance.

2.1.3 Hopfield Networks

A Hopfield network is a single-layered, recurrent, neural network with symmetric all-to-all connections, and in which each neuron has an output value of 1 or -1. Additionally, self-connections are not allowed, $w_{ii} = 0$. On each iteration of the Hopfield network update rule, each neuron will either stay in its current state, or flip to the other state. The state of all neurons represents the state of the Hopfield network. The purpose of the Hopfield network is to simulate long-term memory, which works in an associative way. After an initial training period during which the network learns to recognise patterns, we present a pattern to the network and it returns the closest guess from the patterns that it has stored. A Hopfield network is an example of an unsupervised learning

algorithm[1].

Formally, we define a Hopfield network as a set of N neurons with each neuron x having a value x_i, $x_i \in \{-1, 1\}$, and connectivity matrix w, such that $\forall_{i,j} w_{ij} = w_{ji}$ and $\forall_i w_{ii} = 0$. Additionally, we have the update rule for neuron x:

$$x_i = \begin{cases} 1 & \text{if } h_i \geq 0 \\ -1 & \text{if } h_i < 0 \end{cases}$$

where

$$h_i = \sum_{j=1}^{N} w_{ij} x_j$$

We notice that on each time step, there are two options to update the neurons. We can do it synchronously, updating all the nodes together, or asynchronously, by choosing one neuron randomly to update. The asynchronous version is preferred since it is more biologically realistic [Eda].

Orbits and fixed points

The Hopfield network is a dynamical system, and as it progresses by its update rule, it will eventually reach a fixed point and move no further. A fixed point of a function f is a value x such that $f(x) = x$, that is, the function maps the input back to itself. When that happens, the fixed point we reached is the pattern closest to the initial pattern the network has converged to.

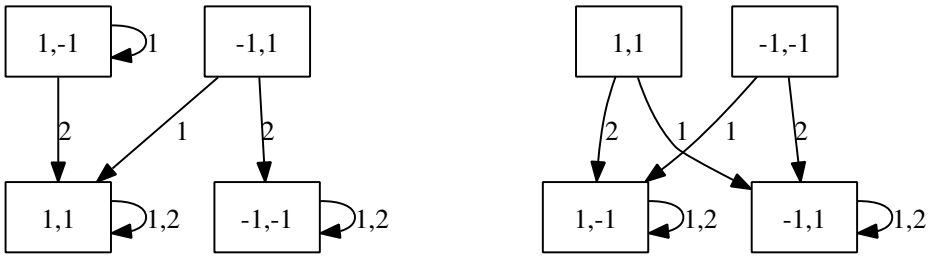
We imagine a Hopfield network comprising of two neurons. We have two non-trivial choices for the connectivities. Either $w_{12} = w_{21} = 1$, or $w_{12} = w_{21} = -1$. There are four possible states in total for the Hopfield network. Figure 2.4 shows the evolution of both of these networks according to the update rule. Both cases converge to fixed points.

For a given Hopfield network, we define an energy function

$$E = -\frac{1}{2} \sum_{i,j=1}^{N} w_{ij} x_i x_j$$

After each iteration of the update rule, the energy of the network decreases, and eventually reaches a stable equilibrium, a local minima of E. The state of the network at that time is the pattern that was closest to the input pattern.

[1] An algorithm that tries to make sense of unstructured and unlabelled data



(a) $w_{12} = w_{21} = 1$ **(b)** $w_{12} = w_{21} = -1$

Figure 2.4 – Orbit plots for the two separate cases. The fixed points in Figure 2.4a are $[1, 1]$ and $[-1, -1]$, and in Figure 2.4b, they are $[-1, 1]$, and $[1, -1]$. The edge labels represent whether we chose to asynchronously update neuron 1 or 2.

Storing a pattern

Assume we want to store a pattern $P = \{x_1, x_2, \ldots, x_N\} \in \{-1, 1\}^N$. Set $w_{ij} = \mu x_i x_j$ where $\mu > 0$ is a learning rate. Under this, P is a fixpoint of the Hopfield net.

$$h_i = \sum_{j=1}^{N} w_{ji} x_i = \mu \sum_{i \neq j}^{N} x_i x_j^2 = \mu \sum_{i \neq j}^{N} x_i = \mu(N-1)x_i$$

Each component of P does not change under the update rule, hence P is a fixpoint.

Storing multiple patterns

To store p patterns $\vec{x}^k$, for $k = 1 \ldots p$, in a Hopfield network of N neurons, set $w_{ij} = \frac{1}{N} \sum_{k=1}^{p} x_i^k x_j^k$.

In the main part of the paper, we present an application of the Hopfield network to a very simple image recognition task.

2.2 Boolean Satisfiability

The core optimisation problem we are looking at in this project is boolean satisfiability. As background, we introduce some definitions and claims that are used throughout the report.

Let P be the alphabet of propositional atoms $\{p_1, p_2, \ldots\}$ and $B = \{\wedge, \vee, \rightarrow, \neg\}$ be the set of logical connectives. We then inductively define a well-formed propositional formula:

1. Any propositional atom $p_i \in P$ is a propositional formula
2. $\top$ and $\bot$ are propositional formulas
3. If A is a propositional formula, so is $\neg A$
4. If A and B are propositional formulas, then $A \wedge B$, $A \vee B$, $A \rightarrow B$ are all propositional formulas.

The above presents a propositional boolean logic whose evaluation the reader is assumed to be familiar with. Some additional required definitions follow. A, B are propositional formulas, and p is an atom.

Atomic
A formula of the form $\top$, $\bot$, p, is called an *atomic formula*

Negated
A formula of the form $\neg A$ is called a *negated formula*

Conjunction
A formula of the form $A \wedge B$ is called a *conjunction*

Disjunction
A formula of the form $A \vee B$ is a called a *disjunction*

Literal
A formula that is either atomic or negated-atomic is called a *literal*

Disjunctive Clause (clause)
A formula that is a disjunction of one or more literals is called a *disjunctive clause* or just a *clause*.

Conjunctive Clause
A formula that is a conjunction of one or more literals is called a *conjunctive clause*.

Definition 1. A propositional formula A is said to be *valid* iff it evaluates to true under all possible assignments of truth values. If so, we write $\vDash A$. If A is valid, we call A a *tautology*. For example $p \vee \neg p$ and $\bot \rightarrow p$ are both tautologies.

Definition 2. A propositional formula A is said to be *satisfiable* iff there exists an assignment of truth values $P : \text{atoms}(A) \mapsto \{\top, \bot\}$ to the propositional atoms in A such that A evaluates to $\top$. Alternatively, A is satisfiable iff $\neg A$ is not valid, that is $\nvDash \neg A$. For example, $p \vee q$ is satisfiable, but $p \wedge \neg p$ is not satisfiable.

Definition 3. Propositional formulas A and B are said to be logically *equivalent* if and only if A and B evaluate to the same truth value for all possible variable assignments. This is equivalent to $\vDash A \leftrightarrow B$.

Definition 4. An assignment $P : \text{atoms}(A) \mapsto \{\top, \bot\}$ for a propositional formula A is said to be a *model* of A iff A evaluates to $\top$ under P.

Definition 5. We say a propositional formula A is in CNF (Conjunctive Normal Form) iff A is a conjunction of clauses, i.e. $A = C_1 \vee C_2 \vee C_3 \vee \cdots \vee C_m$. For example $\neg A \wedge (B \vee \neg C)$ is in CNF, but $\neg(A \vee B)$ is not.

Definition 6. We say a propositional formula A is in DNF (Disjunctive Normal Form) iff A is a disjunction of conjunctive clauses. For example $A = (\neg p \wedge q) \vee c$ is in DNF.

Claim 1. Every propositional formula in DNF can be converted to a logically equivalent propositional formula in CNF.

Proof. Let $A = ((a_{11} \wedge a_{12} \wedge \dots) \vee (a_{21} \wedge a_{22} \wedge \dots) \vee \cdots \vee (a_{n1} \wedge a_{n2} \wedge \dots))$. We convert A to CNF by the law of distributivity. Hence

$A \equiv (a_{11} \vee a_{21} \vee \cdots \vee a_{n1}) \wedge (a_{11} \vee a_{21} \vee \cdots \vee a_{n2}) \wedge \cdots \wedge (a_{11} \vee a_{22} \vee \cdots \vee a_{n1}) \wedge (a_{11} \vee a_{22} \vee \cdots \vee a_{n2}) \wedge \dots$

Notice the exponential increase in the size of the CNF formula. □

Claim 2. Every propositional formula A can be converted to CNF.

Proof. We show this inductively. Clearly, $\top$, $\bot$, and p, for every propositional atom p are already trivially in CNF. Assume inductively that propositional formulas A and B are in CNF.

$\neg A$ can be expressed as $\neg((a_{11} \vee a_{12} \vee \dots) \wedge (a_{21} \vee a_{22} \vee \dots) \wedge \cdots \wedge (a_{n1} \vee a_{n2} \vee \dots))$ since A is in CNF. Distributing the negation yields $((\neg a_{11} \wedge \neg a_{12} \wedge \dots) \vee (\neg a_{21} \wedge \neg a_{22} \wedge \dots) \vee \cdots \vee (\neg a_{n1} \wedge \neg a_{n2} \wedge \dots))$. This formula is in DNF and be converted to CNF by Claim 1.

To translate $A \vee B$ to CNF, we distribute each clause of A over the $\vee$ operator and each clause of B, and the resulting formula is in CNF. Translating $A \wedge B$ to CNF is very similar to the case above.

$A \to B$ is equivalent to $\neg A \vee B$ and can hence be translated to CNF. □

2.2.1 Satisfiability Problems

Let A be a propositional formula. The *satisfiability problem* (SAT problem) is to decide whether A is satisfiable and return the satisfying assignment, or conclude that A is not satisfiable. Most satisfiability solvers only work with formulas in CNF.

If we have a CNF formula such that every clause is a disjunction of at most 2 literals, we have the 2-satisfiability problem. For example, $(p_1 \vee \neg p_2) \wedge (\neg p_1 \vee p_3) \wedge (p_3 \vee p_2)$ is an instance of CNF 2-satisfiability. 2-satisfiability is the simplest satisfiability problem, with solutions in polynomial, and even in linear time [KSS08, p.27].

If we have a CNF formula such that every clause is a disjunction of at most k literals, we have the k-satisfiability problem, which is the most general case of boolean satisfiability. The problem is NP-complete for $k \geq 3$. Intuitively, we can see that if A is a CNF-formula with m unique atoms, then the number of possible boolean assignments to A is 2^m. Hence, the naive satisfiability procedure runs in time $O(2^m)$ - we iterate over every possible assignment in $\{0,1\}^m$ and check whether it is equivalent to $\top$. In practice, currently available algorithms are able to significantly reduce the search complexity.

Most SAT solvers request input in DIMACS CNF format, which is what we will also use throughout this report. For a given formula, all the propositional variables in the formula are represented by integers starting from one. If a propositional variable occurs positively in clause, then the corresponding integer is positive in the corresponding formula, otherwise it is negative. As an example, the DIMACS CNF for the formula

$$(x_1 \vee \neg x_3) \quad \wedge \quad (x_2 \vee x_3 \vee \neg x_1)$$

is

```
p cnf 3 2
1 -3 0
2 3 -1 0
```

The first line says the formula is in CNF, has 3 variables, 2 clauses, and then, for every clause, it states whether each variable occurs positively or negatively in that clause. Every clause is terminated by `0`.

Decision Procedures for k-SAT

SAT solvers and their mechanisms are a vast topic and will not be discussed in detail here. Modern SAT solvers can usually be classified into two classes: ones based on the DPLL (Davis–Putnam–Logemann–Loveland) algorithm, and ones based on stochastic search techniques.

DPLL based algorithms are complete (they find a solution if there is one). They are backtracking based algorithms that explore the formula and decide whether it is satisfiable or not. Such algorithms are exponential in the worst case. [KSS08, 3.5] provides detail on the operation of such algorithms.

The other family of solvers are incomplete - that is, they might not find a satisfying assignment even if there is one. One of the techniques used in such solvers is stochastic local search, which is incomplete due to the fact that it might get stuck in local minima. We concentrate on this class of algorithms since our method of solving SAT using a dynamical system more closely resembles such incomplete algorithms that utilise local search. For a detailed discussion of incomplete SAT algorithms, see [KSS08, 6]. We now present `GSAT` and `WalkSAT` since the ideas behind them motivate the dynamical system approach.

Firstly, to aid understanding these two algorithms, it might help to think that a propositional formula A with n variables and m clauses forms a landscape in the space $\{0,1\}^n \times \{0,1,\ldots,m\}$. The $\{0,1\}^n$ corresponds to all the possible assignments of truth values to the n variables, and $\{0,1,\ldots,m\}$ represents the "height", the total number of clauses that are violated by (evaluate to $\bot$) under the particular truth assignment. It is then an optimisation problem to find an assignment that minimises this height. Figure 2.5 shows an incomplete diagram of such a landscape. It only portrays the boolean assignments, without the corresponding height values, but it nevertheless is a useful visualisation. One can imagine a local search algorithm starting off in one possible truth value assignment, and moving to a possible assignment that has a lower height than the current assignment. In this sense, SAT solving becomes gradient descent.

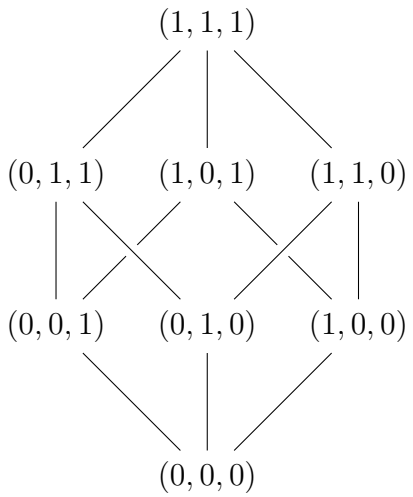


Figure 2.5 – A diagram depicting the possible truth value assignments for a k-sat problem with 3 propositional variables. A local search technique will "walk" along this diagram, and try to minimise the number of false clauses under the assignment. If such a landscape has no local minima, a greedy descent algorithm will converge to an optimal solution

It was believed that a method of SAT solving based on a greedy local descent will not work well in practice because it will quickly fall into local minima, i.e. the algorithm will be left with a small number of clauses remaining unsatisfied. This was found to be false, and a greedy search algorithm, `GSAT` was found to very

quickly converge to a feasible minimum and even outperform the backtracking-based approaches at the time of its introduction.

`GSAT` is based on a randomised local search technique. It starts off by assigning random truth values to the CNF formula. Then, it moves in the direction of greatest descent by flipping the variable that will maximise the decrease in the total number of clauses left unsatisfied. It does this either until it finds a satisfying assignment, or until a pre-set upper bound on the number of iterations is reached. This process is repeated until some predefined number MAX-TRIES. `GSAT` is presented in algorithm 1.

Algorithm 1: GSAT

```
Input: A CNF formula F
Data: Integers MAX-FLIPS and MAX-TRIES
Output: A satisfying assignment for F, or FAIL
begin
    for i ← 1 to MAX-TRIES do
        σ ← a randomly generated truth assignment to F
        for j ← 1 to MAX-FLIPS do
            if σ satisfies F then return σ
            v ← a variable flipping which results in the greatest decrease
            (possibly negative) in the number of unsatisfied clauses
            Flip v in σ
        end
    end
    return FAIL
end
```

`GSAT` performs relatively well. On a SAT problem with a large number of variables, it is actually unlikely for the procedure to get stuck in a local minima (a plateau from which there is no move (a flip of a single variable) that decreases the *break-count* - the number of clauses in the formula that become unsatisfied by the resulting of the proposed move). However, it might take `GSAT` a very long time before finding its way out of a plateau. A proposed improvement is `WalkSAT`. `WalkSAT` interleaves the greedy search strategy with random moves, similar in spirit to the Metropolis algorithm in Monte Carlo Markov Chains (MCMC). In addition, the algorithm focuses the search by selecting variables to flip that are part of a currently unsatisfied clause. `WalkSAT` is presented in

algorithm 2.

Algorithm 2: WalkSAT

```
Input: A CNF formula F
Data: Integers MAX-FLIPS, MAX-TRIES, and noise parameter p ∈ [0, 1]
Output: A satisfying assignment for F, or FAIL
begin
    for i ← 1 to MAX-TRIES do
        σ ← a randomly generated truth assignment to F
        for j ← 1 to MAX-FLIPS do
            if σ satisfies F then return σ
            C ← a random unsatisfied clause in F
            if ∃ variable x ∈ C such that break-count = 0 then
                v ← x
            end
            else
                With probability p:
                    v ← a variable from C chosen at random
                Otherwise (with probability 1 − p)
                    v ← a variable form C with lowest break-count
            end
            Flip v in σ
        end
    end
    return FAIL
end
```

The motivation for looking at various algorithmic approaches for incomplete SAT solving is to help in design and understanding of a dynamical system version of SAT solving which we present later in the paper. In such a system, variables are allowed to take truth values in the range $[-1, 1] \in \mathbb{R}$, but the approach is similar to that of incomplete algorithms like `WalkSat` and `GSAT`.

Formula Hardness

There is an interesting observation regarding the ratio $\alpha = \frac{M}{N}$ of clauses to variables for a K-SAT instance ($K \geq 3$, but since every K-SAT instance can be converted to 3-SAT, we can simply state this for 3-SAT). 3-SAT problems are the "hardest" (by counting the number of steps in the DP procedure) whenever α is around between 4 and 5, with the hardest problems occurring at $\alpha \approx 4.26$. This transition from easy, to hard, and back to easy problems is shown in Figure 2.6. This motivates benchmarking SAT solvers on problems where $\alpha \approx 4.26$.

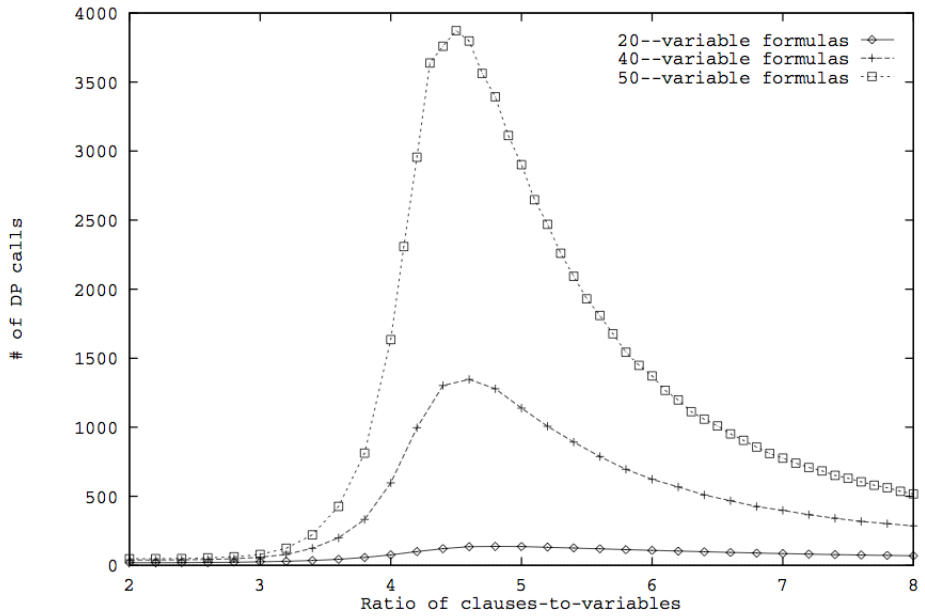


Figure 2.6 – Computational difficulty transition on 3-SAT problems. Adapted from [KSS08]

State of the Art Incomplete SAT Solving - Survey Propagation

A recent and very novel technique for incomplete SAT solving is survey propagation presented in [KSS07]. Survey propagation is based on ideas from belief propagation. The SAT formula is expressed as a graphical model, and marginal probabilities for variable assignments are computed using belief propagation. Variables with the most extreme marginal probabilities are then assigned concrete values and the process of belief propagation is continued. Even though this method is incomplete, it was shown in [KSS07] that it almost never has to backtrack. It is currently the only technique known to solve random 3-SAT instances in the hardest region ($\alpha \approx 4.26$) with over 1 million variables in near-linear time.

2.3 Dynamical Systems

Most generally, a dynamical system consists of a set of variables describing the current state of the system, together with a set of rules that deterministically govern the evolution of the state variables over time [Izh07]. The rules can be, for example, the application of a continuous map on a metric space, or a set of differential equations. We concentrate on dynamical systems that are described by differential equations.

We present a simple example of a dynamical system that models the predator

and prey population using differential equations; it is the simplest model for population dynamics and is named the Lotke-Volterra model [Hop06]. Let x represent the population of prey, and y represent the population of predators. The rules governing the evolution of both populations are:

$$\frac{dx}{dt} = (b - py)x$$
$$\frac{dy}{dt} = (rx - d)y$$

where

- b models the growth of x (prey) in the absence of predators (y)
- p measures the impact of predation. The prey population will grow if $b > py$
- d measures the death rate of the predators in the absence of prey
- r measures the immigration of new predators when prey are around. The predator population will grow if $rx > d$.

Given the initial populations and the above rules with sensible parameter values, we can model and plot both populations.

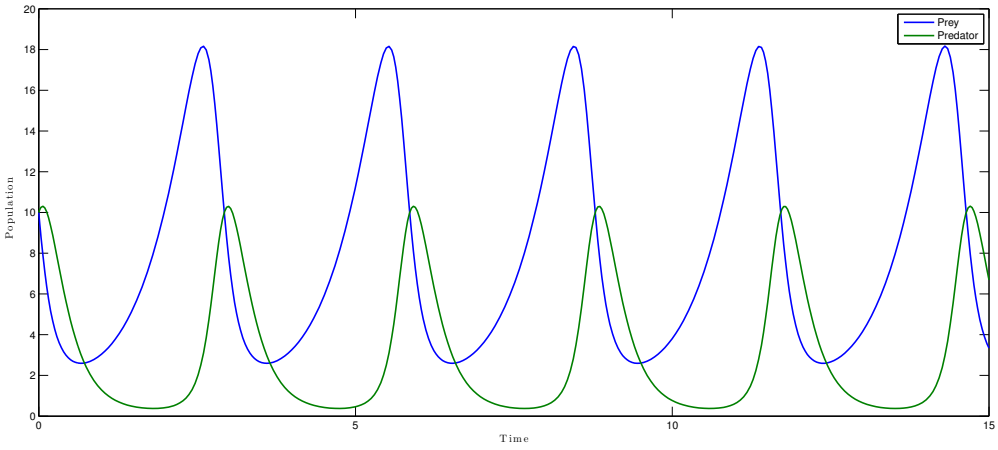


Figure 2.7 – Simulating the predator-prey example with $d = 4$, $r = 0.5$, $b = 1.5$, $p = 0.5$. Notice that the system exhibits periodic behaviour.

2.3.1 Simulating differential equations on a computer

Given rules as differential equations, we need a way to simulate them on a computer. The simplest and most inaccurate is the Euler Method. The family of Runge-Kutta methods gives much higher precision.

Euler Method

Given $y(0)$, $\frac{dy}{dt} = f(y,t)$, and a time-step Δt, we simulate the system as follows:

$$y(t + \Delta t) = y(t) + \Delta t f(y(t), t)$$

Runge-Kutta Methods

While simple, the Euler method is usually not precise enough to simulate many dynamical systems. There is a whole family of Runge-Kutta integration methods much more precise than the Euler method, but the most popular one is Runge-Kutta-4 (RK4, the 4 standing for the number of approximations used), and works as follows:

$$\begin{aligned} k_1 &= f(y(t), t) \\ k_2 &= f(y(t) + 0.5\Delta t k_1, t + 0.5\Delta t) \\ k_3 &= f(y(t) + 0.5\Delta t k_2, t + 0.5\Delta t) \\ k_4 &= f(y(t) + \Delta t k_3, t + \Delta t) \\ y(t + \Delta t) &= y(t) + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

For the simulations in this report, we use Matlab's `ode45`, which is an adaptive version of RK45, which is required to make sure the simulations are correct. The method works by computing $y(t+\Delta t)$ for both fourth-order and fifth-order RK methods. A large difference between the two indicates a large error. If that's the case, the `ode45` method will reduce the step size Δt so as to reduce the error below a predefined threshold.

2.4 Optimisation Problems

An optimisation problem is the problem of finding the best possible solution from all feasible solutions. Usually, optimisation problems are classified either as combinatorial or continuous optimisation problems. In this paper, we are looking at how problems that are usually solved via combinatorial optimisation can be cast to the continuous domain, solved there, and mapped back to the right domain.

In its most general form, a continuous optimisation problem is

$$\text{minimise} f(x)$$

subject to

$$g_i(x) \leq 0 \quad \forall i \in 1 \dots m$$
$$h_j(x) = 0 \quad \forall j \in 1 \dots p$$

where $f(x) : \mathbb{R}^n \to \mathbb{R}$ is called the objective function, $g_i(x)$ is a set of inequality constraints, and $h_j(x)$ is a set of equality constraints. A maximisation problem can easily be cast to a minimisation problem: $\max f(x) = -\min -f(x)$.

In contrast, a combinatorial optimisation problem is the problem of finding an optimal solution from a finite and discrete set of solution objects. Common combinatorial optimisation problems include the Travelling Salesman Problem (TSP), N-Rooks, N-Queens, and the N-SUM problems. We look at the problem statements of each these problems.

2.4.1 Examples

Travelling Salesman Problem

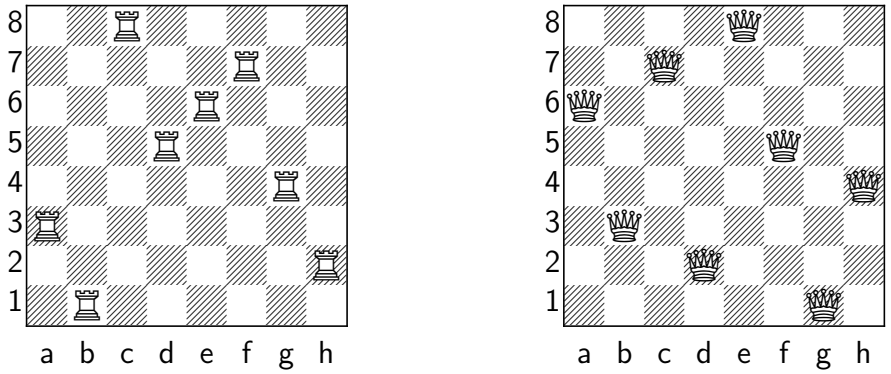
The travelling salesman problem is as follows: Given a list of cities and distances between them, find the shortest route that visits each city exactly once and returns to the point of origin.

N-Rooks

The N-Rooks puzzle is to figure out how to place N rooks onto an an $N \times N$ chessboard such that no rook attacks any other rook.

N-Queens

The N-Queen puzzle is very similar to the N-Rooks puzzle, except we have to place N queens onto an $N \times N$ chessboard such that no queen attacks any other queen. Figure 2.8 shows visually the solutions to the N-rooks and N-queens problems



(a) A solution to the N-rooks Problem **(b)** A solution to the N-Queens Problem

Figure 2.8 – Solutions of N-rooks and N-queens problems

2.5 Related Work

A large part of the paper is based on understanding, replicating, and trying to simplify the work presented in [ERT11]. The referenced paper presents a way of solving SAT problems using continuous dynamical systems which we present and explore further in this paper.

2.5.1 Solving Sudoku using continuous dynamical systems

The same authors who presented a way of solving SAT using continuous dynamical systems applied their technique to solving Sudoku puzzles [ERT12]. The idea is to map the Sudoku problem to an instance of a SAT problem, such that solving the SAT problem yields a solution to the original Sudoku puzzle. The authors find such a mapping, and then solve the resulting logical formulas using the SAT solver they developed. Interestingly, the difficulty of the resulting SAT problem is an indicator of the difficulty of the original Sudoku puzzle.

2.6 Implementation

All the simulations for this project are ran in Matlab. Python was used as a scripting language for various tasks throughout the project.

Chapter 3

Preliminary work

The purpose of this section is to document my exploration of casting optimisation problems other than SAT as dynamical systems and exploring and interpreting the results.

3.1 N-Rooks

We want to place N rooks on an $N \times N$ board such that no rook attacks any other rook. Let V_{ij} represent whether there is a rook in row i, column j on the chessboard. We can express the game with the following constraints:

$$\sum_{i,j} \sum_{k \neq j} V_{ij} V_{ik} = 0 = E_1 \tag{3.1}$$

$$\sum_{j,i} \sum_{k \neq i} V_{ij} V_{ik} = 0 = E_2 \tag{3.2}$$

$$(\sum_{i,j} V_{ij} - N)^2 = 0 = E_3 \tag{3.3}$$

The first constraint says that every row can have at most one non-zero value. The second constraint says that every column can have at most one non-zero value. The final constraint ensures that there are exactly N rooks on the board. We can then define the energy function $E = E_1 + E_2 + E_3$. A state with $E = 0$ corresponds to a solution of the N-Rooks problem.

As shown in the Background section and [HT85], the general form for an energy function of a Hopfield network is

$$E = -\frac{1}{2} \sum_{i,j} w_{ij} v_i v_j$$

The input potential u_i to neuron i is $-\frac{\partial E}{\partial v_i} = \sum_j w_{ij} v_j$.

Instead of using the sgn function to threshold the neuron, we will use the smooth function tanh which approximates the sgn function but is smooth around 0. Then, we have:

$$v(u) = \frac{1}{2}\left(1 + \tanh(\alpha u)\right)$$

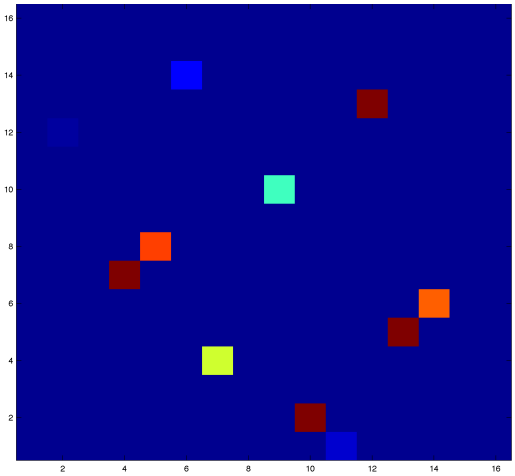
Then, as shown in [HT85], we have the update rule:

$$U_{ij} \leftarrow -\left(A\sum_{k\neq i} V_{ij} + B\sum_{k\neq j} V_{ij} + C(\sum_{i,j} V_{ij} - n + \sigma)\right) \tag{3.4}$$

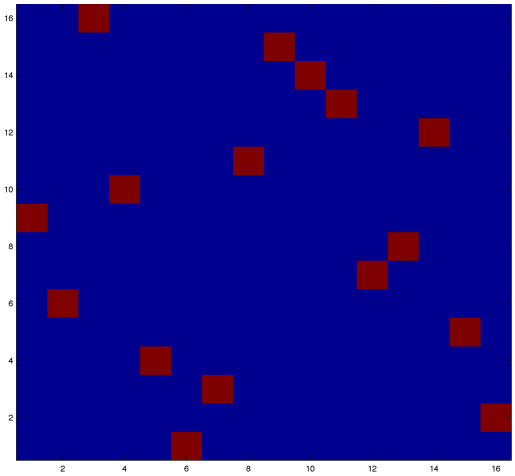
$$V_{ij} \leftarrow v(U_{ij}) \tag{3.5}$$

where A,B,C,σ are parameters.

I run the network with $A = B = 500$, $C = 200$, $\alpha = 50$, $\sigma = 3$, and a random initial assignment to U. The network converges to a solution of the N rooks problem, as shown in Figure 3.1



(a) The network while working. The colour of the square corresponds to U_{ij}, the current potential of cell U_{ij}.



(b) The network in its final state - a solution to N-rooks

Figure 3.1 – Evolution of the network that solves the N-rooks problem

3.2 Travelling Salesman Problem (TSP)

In TSP, we have a list of N cities with their locations. The aim is to start at any one city, visit each city exactly once, come back to the starting city, all while minimising the total distance travelled. In graph theoretical terms, given an undirected weighted graph, we want to find a Hamiltonian cycle that minimises the weight.

We want to solve this in a way similar to the way we solved N-Rooks. Lets define an $N \times N$ matrix V, where $V_{ij} = 1$ if and only if the j'th stop on the final tour is city with index i. Given that, we need to express the following constraints:

- We must visit every city at least once
- At each stage of the tour, we can visit at most one city

These two constraints imply that every row and column of V must have exactly one cell with a 1 in it. This is exactly N-Rooks for which we already have a solution. Additionally, we need to express a constraint that will minimise the total cost. Let d_{ij} be the distance between cities i and j. Then, to minimise the length of the total tour we must minimise

$$L = \sum_{i,j,k} d_{ij} x_{ik} x_{j,k+1}$$

where the indices are taken modulo N, the number of cities, so that $x_{j,N+k} = x_{j,k}$. This allows us to "wrap around" the V matrix. We must then minimise

$$E = \sum_{i,j} \sum_{k \neq j} V_{ij} V_{ik} + \sum_{j,i} \sum_{k \neq i} V_{ij} V_{ik} + (\sum_{i,j} V_{ij} - N)^2 + \sum_{i,j,k} d_{ij} x_{ik} x_{j,k+1}$$

This gives the following update rule for U_{ij} the potential for cell (i, j).

$$U_{ij} \leftarrow -\left(A \sum_{k \neq i} V_{ij} + B \sum_{k \neq j} V_{ij} + C(\sum_{i,j} V_{ij} - n + \sigma) + D \sum_{y} d_{iy} V_{y,j+1} V_{y,j-1} \right)$$
$$V_{ij} \leftarrow v(U_{ij})$$

I simulate the above system in Matlab, placing randomly 10 cities onto the unit square. A resulting solution is shown in Figure 3.2.

While this is a very interesting way to solve TSP, it is by no means good at all. We are not guaranteed to converge to a minimal tour, and we are not even guaranteed to converge to a correct tour. This is still an interesting area for research since it is a way of thinking of a discrete, step-by-step optimisation process in a global and parallel way. At every update of the system, we are taking into account many different possibilities of what tour to take. Another interest for expressing combinatorial problems this way is that we can construct analog, specialised circuits to solve such problems, as presented originally by Hopfield and Tank in the paper that introduces the Hopfield network [HT85].

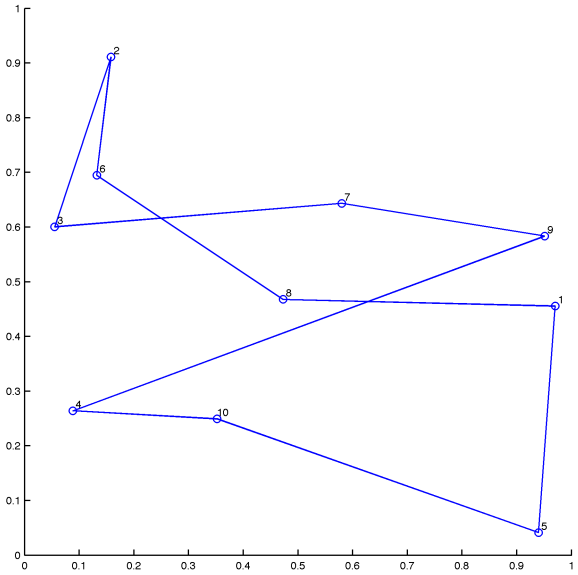
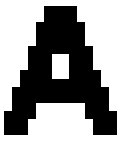


Figure 3.2 – 10 cities on the unit square, and a solution arrived at using the network as presented above

3.3 Image Reconstruction

In this section, we use a Hopfield network to perform simple image reconstruction. We train the network to store binary images of the letters A,B,C, where each image is 10 by 10 pixels wide, and hence resides in $\{-1, 1\}^{100}$ where 1 represents white, and -1 black. After the training phase, I present the network with a distorted image of one of the letters (some pixels flipped, but still resembling the original letter - see Figure 3.3), and allow it to converge iteratively to one of the stored patterns, as described in the background section on Hopfield Networks.



(a) An example training pattern for the network

(b) An example of a broken pattern that the network is able to reconstruct back to Figure 3.3a

Figure 3.3 – Input to the Hopfield Network

Figure 3.4 shows a graphical implementation of the procedure. The user can pick a "broken" image of a letter from a predefined set of images, and see the live reconstruction happen as the network eventually converges to one of the initially stored patterns.

The purpose of this section was not to go into too much detail, but to show the

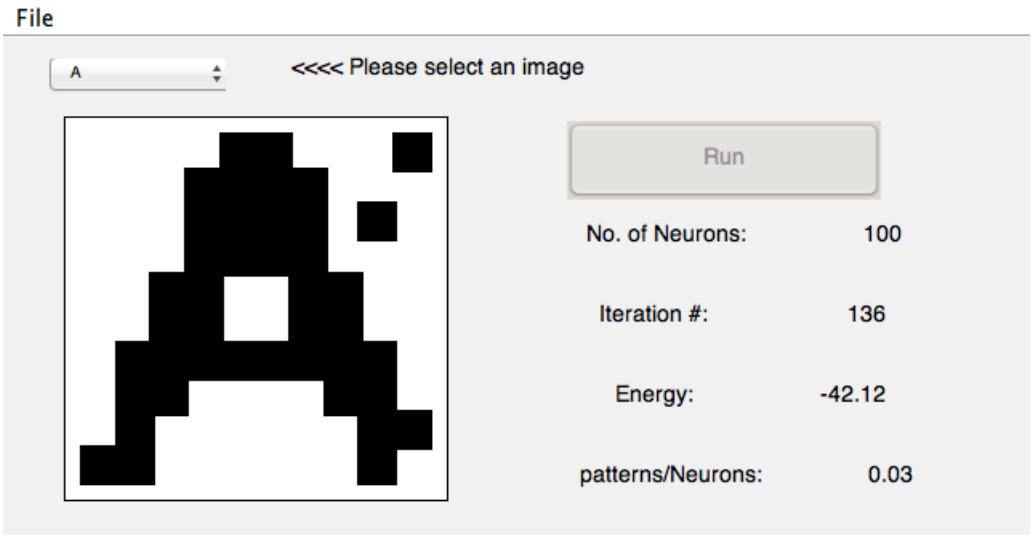


Figure 3.4 – A GUI showing the application of Hopfield networks to the problem of retrieving a stored pattern. In this case, the stored patterns are binary images of letters, and the task is to reconstruct the original image given a broken image.

application of dynamical systems, in particular the Hopfield network, to solve combinatorial problems, even NP-complete ones such as TSP. From now on, we will concentrate on exploring and solving SAT problems using dynamical systems.

Chapter 4

Dynamical Systems for SAT Solving

In this chapter, we look at three different ways of expressing the satisfiability problem as a continuous dynamical system. The first two methods presented were developed in [ERT11] and [MTER12], and the third method is unpublished work by Murray Shanahan.

Keep in mind that the purpose of the systems is not to create a faster SAT solver (these systems are many orders of magnitude slower), but rather to explore the casting of a problem that appears completely discrete to the continuous domain. When expressed as a dynamical system, the optimisation problem can be solved on a specialised analog circuit, or even on something based on real neurons.

4.1 Modelling with unbounded variables

The approach shown here is presented in [ERT11].

We are given a K-SAT problem with N propositional variables and M clauses. For each propositional variable x_i, we introduce the continuous variable $s_i \in [-1, 1]$, with $s_i = 1$ corresponding to the i'th propositional variable being $\top$, and $s_i = -1$ to the x_i'th variable being equal to $\bot$. All the other possible continuous values represent the degree of "truthness" of propositional variable x_i.

Let C_m represent the m'th clause, for example $C_1 = x_4 \vee \neg x_2 \vee x_1$. The whole CNF formula can be encoded as a matrix c_{mi}:

$$c_{mi} = \begin{cases} 1 & \text{if } x_i \in C_m \\ -1 & \text{if } \neg x_i \in C_m \\ 0 & \text{if } x_i \notin C_m \text{ and } \neg x_i \notin C_m \end{cases}$$

We define a constraint function $K_m : [-1,1]^N \mapsto \mathbb{R}$:

$$K_m(\mathbf{s}) = \prod_{i \in V(m)} \frac{1}{2}(1 - c_{mi}s_i) = 2^{-k} \prod_{i=1}^{N} (1 - c_{mi}s_i) \tag{4.1}$$

where V_m are the propositional variables occurring in clause m. The intuition is to have this function express how far away a clause is from being satisfied. $\frac{1}{2}(1 - c_{mi}s_i) \in [0,1]$ expresses how far away the variable value s_i is from what it needs to be in clause m, c_{mi}. If it agrees on the value, then $c_{mi} = s_i$, and $\frac{1}{2}(1 - c_{mi}s_i) = 0$. Since the constraints are in CNF, only one of the literals needs to evaluate to $\top$, and this is why we take in the function K_m we take the product. Hence, clause m evaluates to $\top$ under $\mathbf{s}$ iff $K_m(\mathbf{s}) = 0$.

We can now define an energy function $E(\mathbf{s})$:

$$E(\mathbf{s}) = \sum_{m=1}^{M} K_m(\mathbf{s})^2$$

The goal is then to find a solution $\mathbf{s}* \in \{-1,1\}^N$ with $E(\mathbf{s}*) = 0$. Such solutions $\mathbf{s}*$ will the global minima of E, although finding these minima directly will usually not succeed because of trapping non-solution attractors. To solve this, the authors of [ERT11] provide a modified energy function $V(\mathbf{s}, \mathbf{a})$:

$$V(\mathbf{s}, \mathbf{a}) = \sum_{m=1}^{M} a_m K_m(\mathbf{s})^2$$

where the variables $a_m \in (0, \infty)$ are auxiliary variables similar to Lagrange multipliers. The idea is that the growth of these variables will eventually force us out of local minima. The continuous time dynamical system for solving SAT is then defined as:

$$\frac{ds_i}{dt} = (-\nabla_s V(\mathbf{s}, \mathbf{a}))_i = \sum_{m=1}^{M} 2a_m c_{mi} K_{mi}(\mathbf{s}) K_m(\mathbf{s}), \qquad i = 1 \ldots N$$

$$\frac{da_m}{dt} = a_m K_m(\mathbf{s}), \qquad m = 1 \ldots M$$

where $K_{mi} = \frac{K_m}{1 - c_{mi}s_i}$. The initial conditions for $\mathbf{s} \in [-1,1]^N$ are arbitrary, but a_m has to be strictly positive. [ERT11] shows how solutions to a SAT problem will be attractive fixed points of this defined dynamical system. The key properties of this system include:

- The dynamics of s stays confined to $[-1,1]^N$.
- Solutions to a SAT problem will be attractive fixed points of the system
- There are no limit cycles

- For satisfiable formulae, the only fixed point attractors are the global minima of V with $V = 0$.
- The formulation uses unbounded variables a_m.

Figure 4.1 presents simulating the system. The important concept to notice is that the clause weights are monotonically increasing. This is the feature of the system that makes it eventually get out of minima.

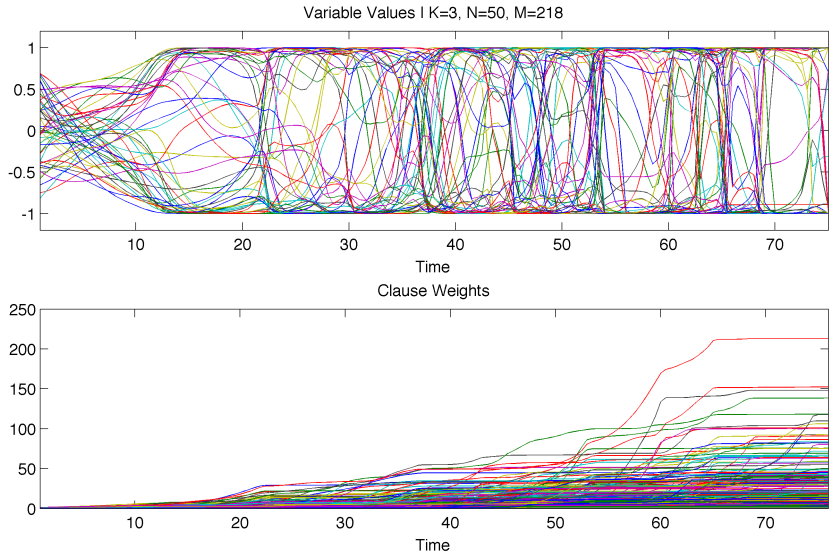


Figure 4.1 – Solving a 50 variable hard SAT problem using the unbounded dynamical system as presented in [ERT11]

4.2 Modelling as a Recurrent Neural Network with bounded variables

In [MTER12], the same authors present a similar system, this time encoded as a CTRNN (Continuous-Time Recurrent Neural Network), akin to a continuous version of the Hopfield network shown in the Background section.

The general form of dynamics in CTRNNs is:

$$\frac{d_{xi}}{dt} = -x_i(t) + \sum_j w_{ij} f(x_j(t)) + u_i$$

where x_i is the current value of a cell, w_{ij} is the connectivity between cells i and j, f is an output function (usually sigmoidal), and u_i is a bias term for cell i.

To encode a SAT problem in this way, we first express it as a bipartite graph with connections between variable cells and clause cells. This is shown in Figure 4.2.

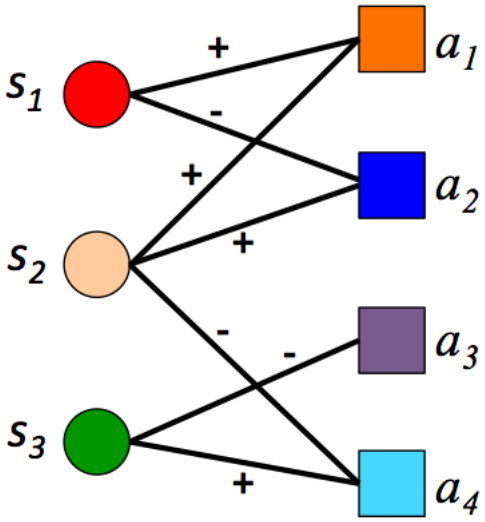


Figure 4.2 – A bipartite graph that encodes a SAT problem. The connection labels are based on the connectivity matrix c_{mi}. If a variable i occurs positively in clause m, then we have positive connection between s_i and a_j and if it occurs negatively, we have a negative connection. Figure adapted from [MTER12]

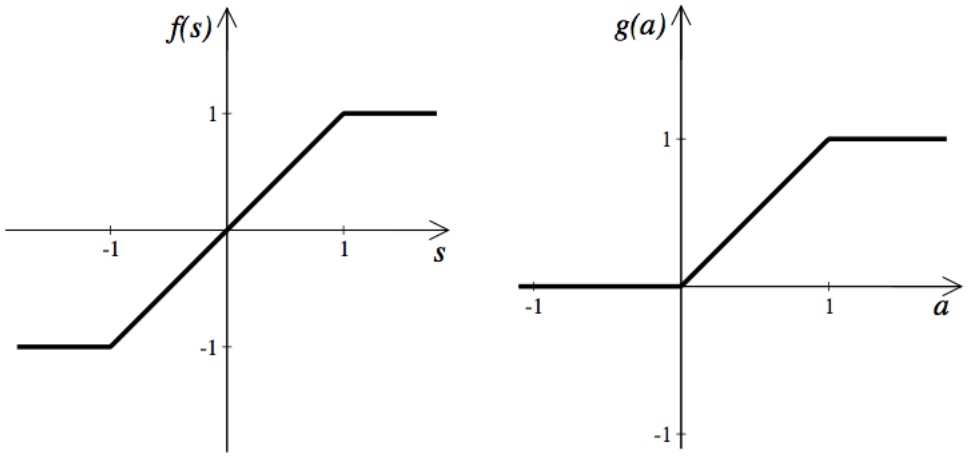


Figure 4.3 – Output functions for the two cell types: variable cells, and clause cells. They work as threshold units.

There are two output functions, one for each of variable and clause cells. For variables, we have:

$$f(s_i) = \frac{1}{2}(|s_i + 1| - |s_i - 1|)$$

and for clause cells:

$$g(a_m) = \frac{1}{2}(1 + |a_m| - |1 - a_m|)$$

This is easier to interpret graphically, as shown in Figure 4.3. There is no bias term for variable cells, but we need a $1 - k$ bias term for all clause cells. The

dynamical system, parametrized by A, the self-coupling parameter for variable cells, and B, the self-coupling parameter for clause cells, is:

$$\frac{ds_i}{dt} = -s_i(t) + Af(s_i(t)) + \sum_m c_{mi} g(a_m(t)) \tag{4.2}$$

$$\frac{da_m}{dt} = -a_m(t) + Bg(a_m(t)) - \sum_i c_{mi} f(s_i(t)) + (1 - k) \tag{4.3}$$

[MTER12] shows the following important properties of the system:

- All the variables remain bounded
- Every K-SAT solution has a corresponding stable fixed point.
- A stable fixed point of the system always corresponds to a K-SAT solution

Even though a stable fixed point of the system always corresponds to a K-SAT solution, the dynamics can still get trapped in a limit cycle and not converge to a solution.

We have implemented this dynamical system in Matlab and solved 75 randomly chosen hard 3-SAT problems with 20 variables ($\alpha \approx 4.26$) and varied the A and B parameters. The results are shown in Figure 4.4.

The system performs well and solves many hard 20 and 50 variable problem 3-SAT problems. Figure 4.5 shows the system eventually converging to a solution fixpoint after a long chaotic transient. As noted, the system can get stuck in a limit cycle, which is demonstrated in Figure 4.6.

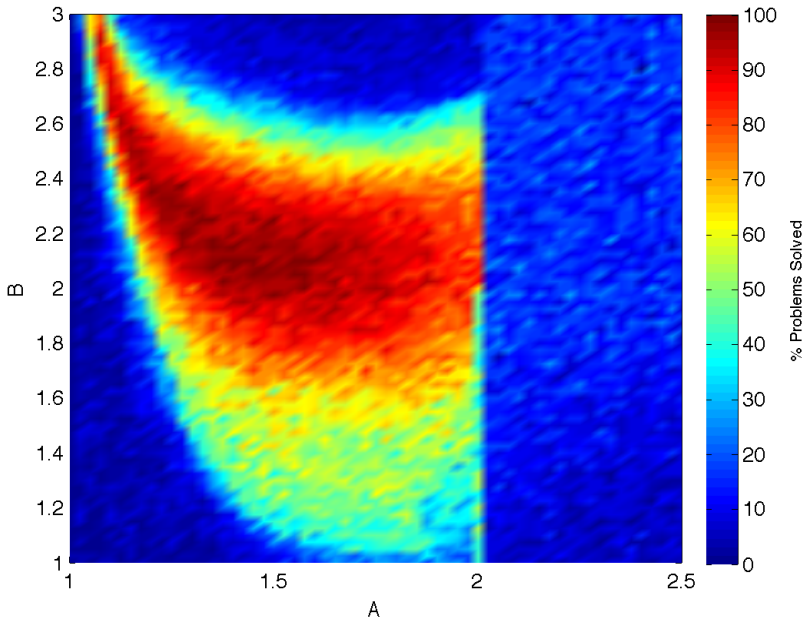


Figure 4.4 – Proportion of solved hard random 3-SAT problems with time limit 10000 with various parameters of A,B when solving using the bounded dynamical system for SAT as presented in [MTER12]

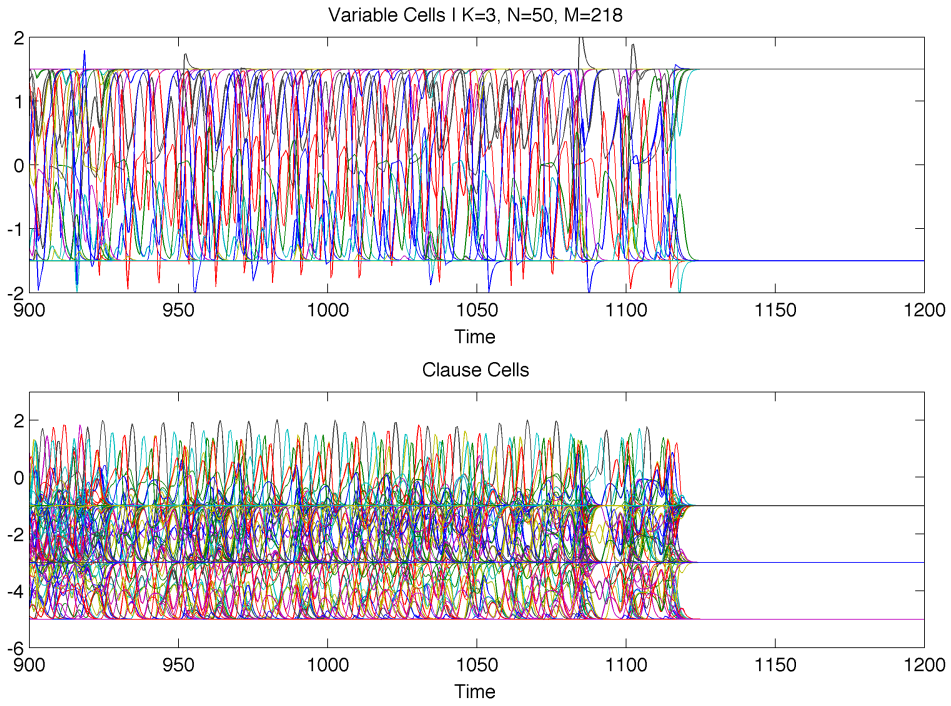


Figure 4.5 – Using the bounded SAT dynamical system as presented in [MTER12] to solve a problem. A successful solution is found after a long chaotic transient.

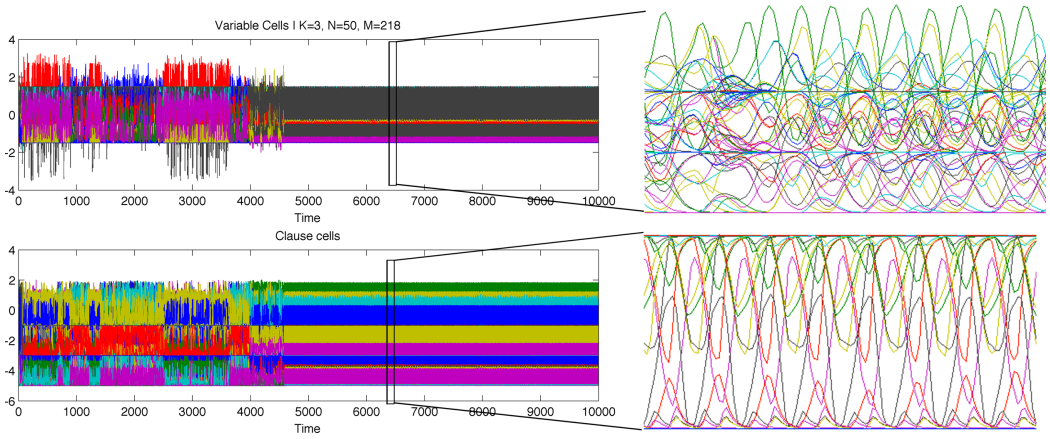


Figure 4.6 – Using the bounded SAT dynamical system as presented in [MTER12] to solve a problem. The system gets trapped in a limit cycle

4.3 A third approach

In this section we present another formulation as developed by Murray Shanahan. We use again Equation 4.1, $K_m(\mathbf{s})$, that expresses the current cost of the clause, and is 0 if and only if clause m is satisfied.

The main idea is that clauses push the variable values towards the value that is required of it in that clause, in an attempt to satisfy it. Figure 4.7 shows this kind of push-pull on a bipartite graph.

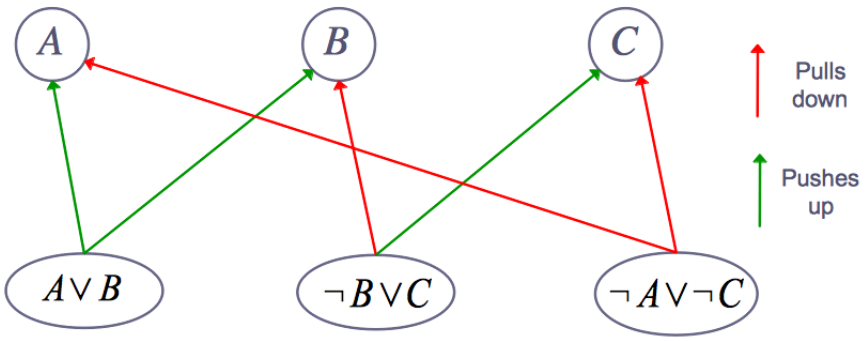


Figure 4.7 – A bipartite graph representation of the 2-SAT problem $(A \vee B) \wedge (\neg B \vee C) \wedge (\neg A \vee \neg C)$. Clauses push or pull variables towards the values that would make the clause satisfiable. Diagram adapted from [Sha]

The amount of pull clause m exerts on variable i is weighted by:

- λ_m - A weighted exponential that keeps track of the clause cost K_m
- The sum of individual variable costs $\frac{1}{2}(1-(s_j c_{mj}))$ for variables j in clauses m other than i.

This system is different and interesting, in the sense that unlike the previous two configurations, in here we take into account past clause costs K_m by keeping track of them using a weighted exponential λ_m. That is, we want $\frac{d\lambda_m}{dt} \propto K_m - \lambda_m$. This leads to the following dynamical system:

$$\frac{d\lambda_m}{dt} = \alpha(K_m - \lambda_m) \tag{4.4}$$

$$\frac{ds_i}{dt} = \frac{1}{\Gamma}\sum_{m=1}^{M}\left(\lambda_m\left(\sum_{j\in V(m), j\neq i}\frac{1}{2}(1-s_j c_{mj})\right)(1-|s_i|)c_{mi}\right) \tag{4.5}$$

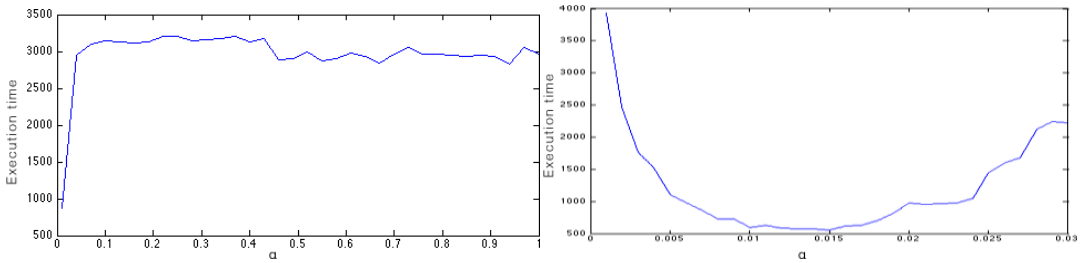
where $\Gamma = \sum_{m=1}^{M}\lambda_m$, and $0 < \alpha \leq 1$.

The system is parametrized by the one parameter α, the influence of the current clause cost on the cumulative clause cost. Figure 4.8 shows experimental results for different values of α. We find the optimal α parameter to be ≈ 0.015.

The system is different from the previous two shown above, but exhibits the following properties:

- A solution $\mathbf{s}*$ is not a fixed point of the dynamical system. Nevertheless, if $\mathbf{s}*$ is a solution to a SAT problem, then $\forall_m K_m(\mathbf{s}*) = 0$, and hence λ_m will decay with rate α towards 0.
- The system can get stuck in non-solution attractors $(\mathbf{s}', \lambda')$. The influences from all clauses can cancel each other out even if $\neg\forall m K_m(\mathbf{s}') = 0$. An example of this is shown in Figure 4.10.
- All the variables in the system are bounded between -1 and 1.
- We simulate the system until all the cumulative clause costs λ_m are less than a certain threshold, taken to be 0.1. Then, the converged solution is taken to be $\text{sgn}(\mathbf{s})$.

Interestingly, even though the system does not establish the key fixed point properties we observed in the previous two, it still deals surprisingly well with random 3-SAT instances, at times dealing with a problem where the previous bounded method would get stuck in. Figure 4.9 shows the dynamical system converging to a solution of a hard (clause-to-variable ratio ≈ 4.26) 50 variable problem.



(a) Varying α parameter between 0 and 1. **(b)** Varying α between 0 and 0.03

Figure 4.8 – Varying the α parameter on two scales. The y axis shows the mean number of steps taken to solve the problem. A large number indicates that a solution was not found in the allotted time period. The tests were ran on a suite of random, hard, 3-SAT, 20 variable problems. We find the optimal α parameter to be ≈ 0.015.

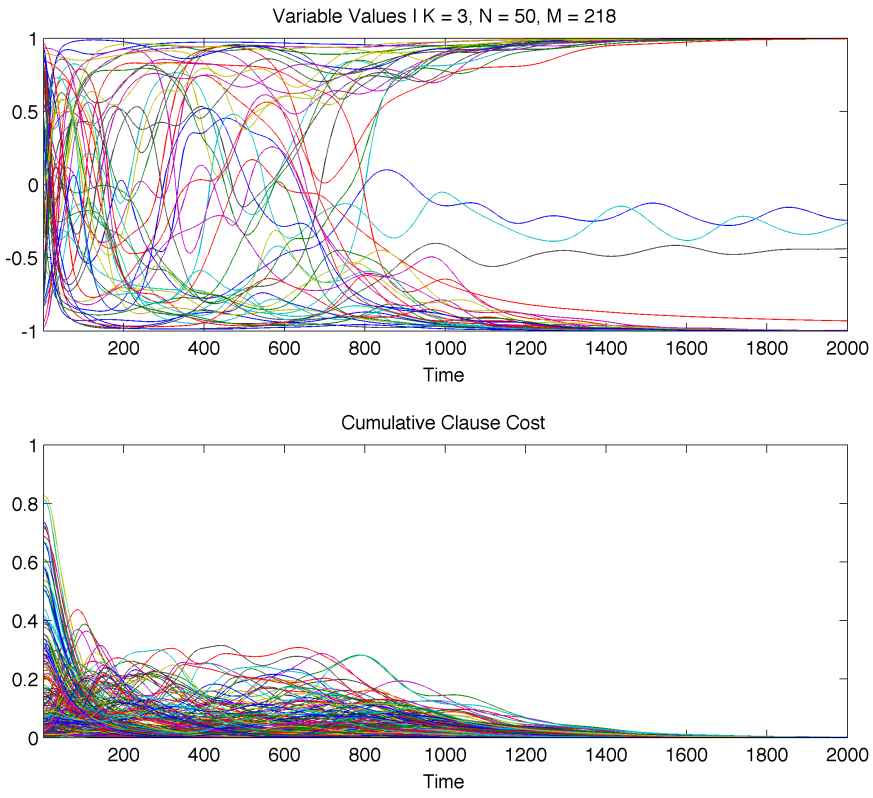


Figure 4.9 – Running the dynamical system as developed by Shanahan. All the clause costs decay to zero and we find a correct solution

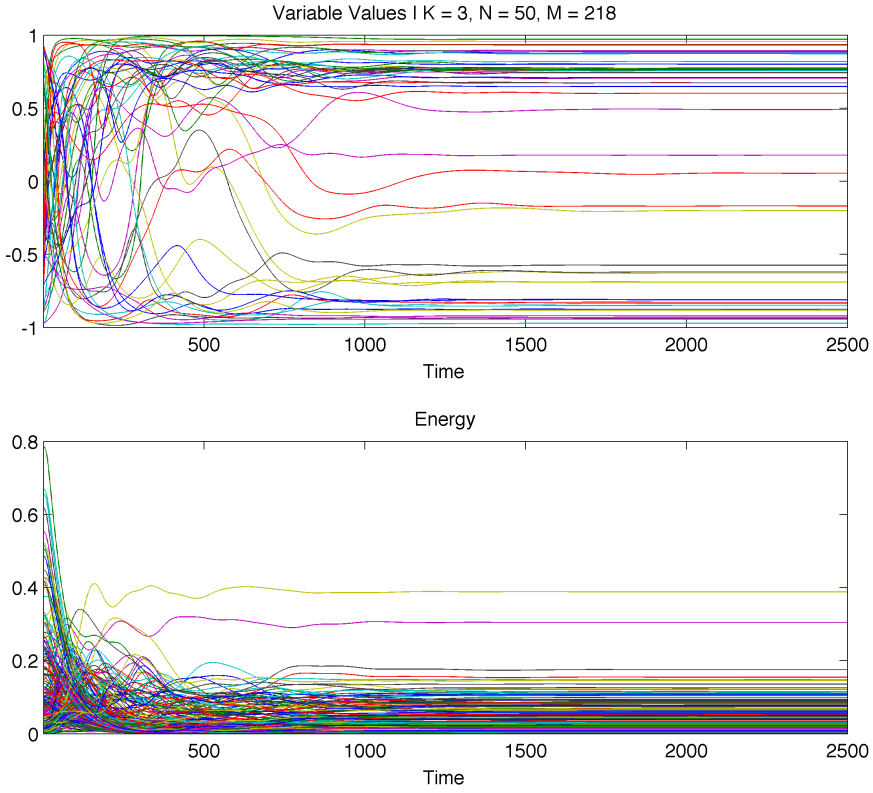


Figure 4.10 – Running the dynamical system as developed by Shanahan. We get stuck in a non-solution attractor. The cumulative clause costs do not drop down to zero

4.3.1 A possible alteration

A possible alteration of this approach we propose, is replacing the summation in Equation 4.5 with a product, leading to an approach similar in spirit to that present in [ERT11], except with bounded variables. The adapted system is then:

$$\frac{d\lambda_m}{dt} = \alpha(K_m - \lambda_m) \tag{4.6}$$

$$\frac{ds_i}{dt} = \frac{1}{\Gamma}\sum_{m=1}^{M}\left(\lambda_m(K(m)-1)\left(\prod_{j\in V(m),j\neq i}\frac{1}{2}(1-s_j c_{mj})\right)(1-|s_i|)c_{mi}\right) \tag{4.7}$$

where $K(m)$ is the number of variables in clause m. This alteration leads to a much larger variation in both the variable values and cumulative clause costs and to much longer solve times for easier problems - a problem that the unmodified version usually solves in around 2000 time steps takes around 8000 steps to solve with this alteration. Nevertheless, a hard 50 variable 3SAT problem[1] that the unmodified version always gets stuck is, is solved by this modified version at the expense of large fluctuations of the variable values and cumulative clause costs. A successful solving of the hard 3SAT formula is shown in Figure 4.11

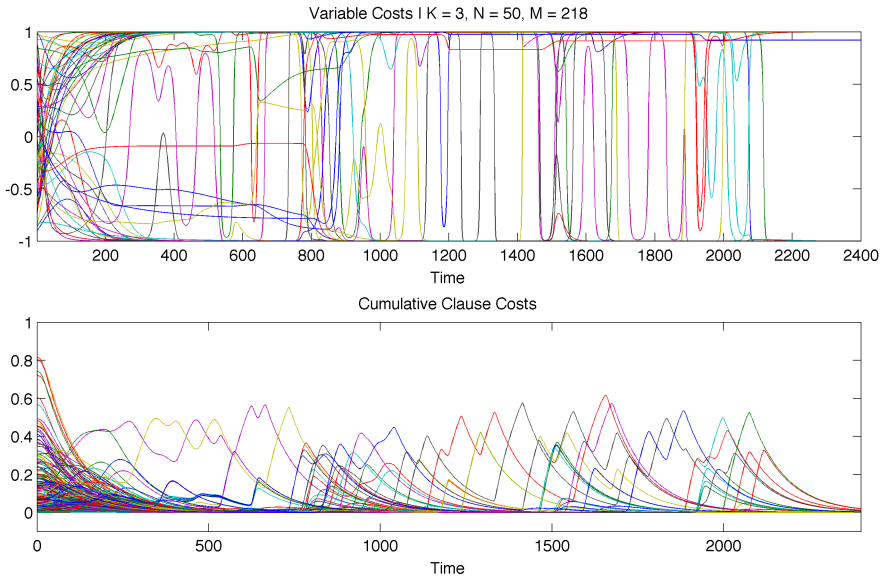


Figure 4.11 – Solving a 50 variable hard SAT problem with a modified version of the third approach. The system successfully converges to a solution, at the expense of large fluctuation in both variable and cumulative clause weight values

[1] `uf50-02.cnf` from a test suite of randomly generated hard 3SAT problems

Chapter 5

SAT Applications

Having looked at different possible dynamical systems for solving boolean k-SAT, we now look at possible applications to examine the feasibility of SAT solving via dynamical systems. We encode N-Rooks and N-Queens as SAT problems, and find that all formulations can deal with these simple problems. For a more interesting application, we encode the logic game *Bridges* into SAT, and apply the dynamical system approach.

5.1 N-Rooks and N-Queens

In this section we create a representation of the N-Rooks problem as a SAT problem, its natural extension to the N-Queens problem, and solve both problems using the continuous dynamical system presented earlier.

5.1.1 N-Rooks

As a reminder, in the N-Rooks problem we are to place N rooks on an $N \times N$ chessboard such that no rook attacks any other rook.

Let r_{ij} be a boolean variable representing whether there is a rook present at tile (i, j) on the chessboard. We can then express the required constraints in SAT as follows:

1. Every row and column has to have at least one rook

$$\underbrace{\left(\bigwedge_{i\in 1\ldots N}\bigvee_{j\in 1\ldots N} r_{ij}\right)}_{\text{Row Constraints}} \wedge \underbrace{\left(\bigwedge_{i\in 1\ldots N}\bigvee_{j\in 1\ldots N} r_{ji}\right)}_{\text{Column Constraints}}$$

2. Every row and column has to have at most one rook

   This can be expressed by saying that for every pair $(r_{i,j}, r_{i,k})$ where $i \neq k$, we have $\neg(r_{ij} \wedge r_{ik})$ which is equivalent to $\neg r_{ij} \vee \neg r_{ik}$ which is in CNF as desired. We do the same procedure for the all the column constraints.

The two constraints expressed above together imply that each row and column has exactly one rook, and is hence a solution to the problem. The total number of propositional variables this representation uses is N^2, one for each r_{ij}, and the number of clauses is $2N + N^2(N-1)$, a third degree polynomial.

5.1.2 N-Queens

The above solution is easily extended to the N-Queens problem. We need to add additional constraints that disallow a queen attacking another queen across a diagonal.

If two separate coordinate points (i, j),(k, l) lie on a common diagonal, then we must add the constraint $\neg(r_{ij} \wedge r_{kl})$ which is equivalent to the CNF formula $\neg r_{ij} \vee \neg r_{kl}$

All the systems as presented in the previous section were able to deal with these encodings for $N = 8$. Since N-Rooks and N-Queens are very simple problems, we do not analyse the dynamical systems further with respect to those two problems, but rather, to evaluate the dynamical systems, we look at a more interesting application to the logic puzzle *Bridges*.

5.2 Game of *Bridges*

In this section we look at formulating the game *Bridges*[1], also called Hashiwokakero as a SAT problem, and see whether the continuous time dynamical system for SAT solving is able to deal with the resulting problem. The decidability of this puzzle is NP-Complete [And]. The aim of the study is to see the performance of the solver on a real-life problem since experimental results in [MTER12] only look at random SAT problems.

5.2.1 Game Definition

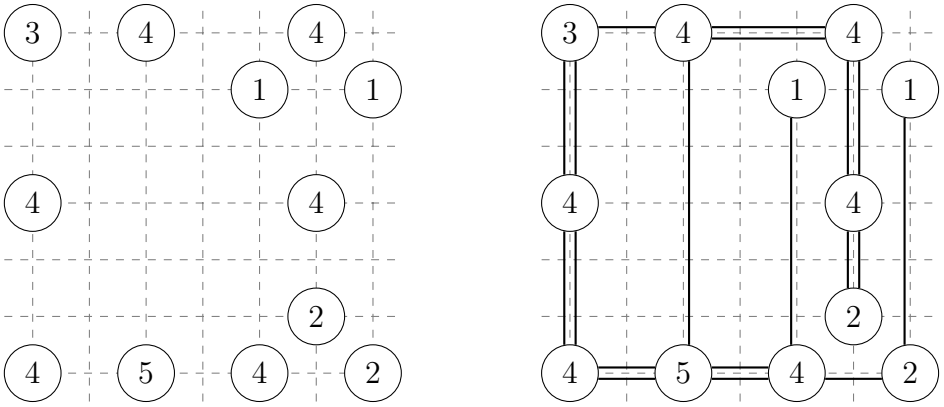
Bridges is played on an $N \times N$ grid. At the beginning, each grid intersection is either empty, or has an island node with an associated number n, $1 \leq n \leq 8$.

[1] `http://www.puzzle-bridges.com`

The goal is to place horizontal and vertical *bridges* on the grid intersections such that:

1. Each grid intersection is either empty, has at most two bridges of the same type (either vertical or horizontal, no diagonal bridges are allowed), or is an island node
2. A bridge must be continuous and must have two island nodes at its endpoints
3. Bridges cannot cross (no grid intersection point can have a vertical and horizontal connection at the same time)
4. The total number of bridges connected to an island must equal the degree of the island.
5. Every island must be reachable from every other island

Figure 5.1 shows an initial board state and a solution to it. In all figures and equations about *Bridges* we place the origin $(1, 1)$ at the top left, and so grid intersection $(2, 5)$ in Figure 5.1 is the island with degree 1.



(a) An initial state for the Bridges game **(b)** A possible solution

Figure 5.1

In the following sections we describe a progressive attempt at formalising this problem as a SAT instance, look at the complexity of the resulting problem, and see if the continuous dynamical system solver is able to converge to a solution to this real life problem.

5.2.2 First attempt

In the first attempt at formalising *Bridges*, I ignore constraint 5 (I allow disconnected components), and furthermore, restrict myself to allowing at most one bridge. This means each island can now have degree at most 4.

The idea is to have for each of the $N \times N$ grid intersections two propositional variables: v_{ij} (for a vertical bridge), and h_{ij} (for a horizontal bridge) and express the constraints of the game using these $2N^2$ propositional variables.

- Connectivity Constraints

  We want the bridges to be connected, so if we have a horizontal bridge at tile (i, j), then tiles $(i, j-1)$ and $(i, j+1)$ must also be a horizontal bridge. Similarly, for a vertical bridge at tile (i, j) we require tiles $(i-1, j)$ and $(i+1, j)$ to have a vertical bridge. We can express this as:

$$h_{i,j} \to h_{i,j-1} \wedge h_{i,j+1} \equiv (\neg h_{i,j} \vee h_{i,j-1}) \wedge (\neg h_{i,j} \vee h_{i,j+1})$$
$$v_{i,j} \to v_{i-1,j} \wedge v_{i+1,j} \equiv (\neg v_{i,j} \vee v_{i-1,j}) \wedge (\neg v_{i,j} \vee v_{i+1,j})$$

- Bridges cannot cross

  This means for every non-island tile (i, j) we can't have both types of bridges. This can be expressed as

$$\neg(h_{i,j} \wedge v_{i,j}) \equiv \neg h_{i,j} \vee \neg v_{i,j}$$

  for all tiles (i, j) which are empty in the original problem.

- Bridges have to start and end at an island

  For every island node (x, y), force $v_{x,y}$ and $h_{x,y}$ to be both true. This means that an island node will allow other bridges to connect to it. The connectivity constraint together with this constraint, imply that every bridge starts at and ends at an island, and is continuous throughout.

- Degree constraints

  We now need to express that if an island node has a connectivity constraint n, then exactly a total of n horizontal and vertical bridges connect to it. This is more difficult to express than the previous constraints. I express it as a simple enumeration. Given an island node at (x, y) with degree n, look at which directly neighbouring tiles are empty (there can't be another node directly there, and it can't be outside of the game board). There must be $k \geq n$ such tiles. Then, take every one of $\binom{k}{n}$ combinations and express that the particular n variables chosen in this combination are true, while the other $k - n$ are false.

  As an example, see Figure 5.2 which shows an initialised *Bridges* board. The island at $(4, 2)$ (marked in red) with degree 4 requires all possible four connections. The possible choices are marked by dashed lines. The relevant constraints for that node are then:

$$v_{3,2} \wedge v_{5,2} \wedge h_{4,1} \wedge h_{4,3}$$

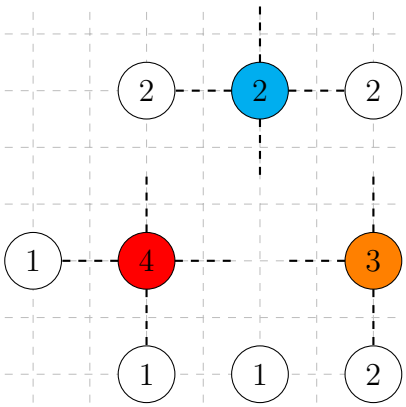


Figure 5.2 – Generating island degree constraints

The node at $(1, 4)$ with degree 2 (marked blue) has four possible bridge locations, but it requires only two. There are $\binom{4}{2} = 6$ ways to choose how the island should be connected.

$$(h_{1,3} \wedge h_{1,5} \wedge \neg v_{0,4} \wedge \neg v_{2,4}) \vee$$
$$(h_{1,3} \wedge \neg h_{1,5} \wedge v_{0,4} \wedge \neg v_{2,4}) \vee$$
$$(\neg h_{1,3} \wedge h_{1,5} \wedge v_{0,4} \wedge \neg v_{2,4}) \vee$$
$$(h_{1,3} \wedge \neg h_{1,5} \wedge \neg v_{0,4} \wedge v_{2,4}) \vee$$
$$(\neg h_{1,3} \wedge h_{1,5} \wedge \neg v_{0,4} \wedge v_{2,4}) \vee$$
$$(\neg h_{1,3} \wedge \neg h_{1,5} \wedge v_{0,4} \wedge v_{2,4}) \vee$$

The orange node with degree 3 has only $\binom{3}{3} = 1$ choice of what bridges need to be present in the solution, since it is at the edge of the board and only has 3 candidate intersections to place a bridge at.

Notice that the first three conversions shown above produce formulas that are already in CNF. The formula ensuring degree constraints for each node is in DNF, and hence before being able to solve it with a SAT solver, we must convert it to a CNF formula. This procedure is shown in Claim 1. As mentioned, this procedure is exponential, and hence for a formula in DNF with x conjunctive clauses, each with y literals, the resulting CNF formula will have y^x clauses each with x literals. This is inconvenient, and we later look at a way to reduce the number of generated clauses for the formulas responsible for degree constraints.

5.2.3 Allowing for two bridges

We now modify the above construction to allow for the possibility of two bridges between a pair of nodes. Instead of having variables $h_{i,j}$,$v_{i,j}$ for each grid intersection, we need to have $h^1_{i,j}, h^2_{i,j}, v^1_{i,j}, v^2_{i,j}$ - a horizontal and vertical variable for

each possible bridge. This increases the total number of variables to $4N^2$. All the constraints require only a small change to adapt to more bridges:

- Connectivity Constraints

  We simply duplicate the connectivity constraints to account for another bridge:

$$
\begin{aligned}
h^1_{i,j} \to h^1_{i,j-1} \wedge h^1_{i,j+1} &\equiv (\neg h^1_{i,j} \vee h^1_{i,j-1}) \wedge (\neg h^1_{i,j} \vee h^1_{i,j+1}) \\
v^1_{i,j} \to v^1_{i-1,j} \wedge v^1_{i+1,j} &\equiv (\neg v^1_{i,j} \vee v^1_{i-1,j}) \wedge (\neg v^1_{i,j} \vee v^1_{i+1,j}) \\
h^2_{i,j} \to h^2_{i,j-1} \wedge h^2_{i,j+1} &\equiv (\neg h^2_{i,j} \vee h^2_{i,j-1}) \wedge (\neg h^2_{i,j} \vee h^2_{i,j+1}) \\
v^2_{i,j} \to v^2_{i-1,j} \wedge v^2_{i+1,j} &\equiv (\neg v^2_{i,j} \vee v^2_{i-1,j}) \wedge (\neg v^2_{i,j} \vee v^2_{i+1,j})
\end{aligned}
$$

- Bridges cannot cross

  For this constraint, we enumerate the four possibilities that are not allowed to happen in a correct solution:

$$
\begin{aligned}
\neg(h^1_{i,j} \wedge v^1_{i,j}) &\equiv \neg h^1_{i,j} \vee \neg v^1_{i,j} \\
\neg(h^1_{i,j} \wedge v^2_{i,j}) &\equiv \neg h^1_{i,j} \vee \neg v^2_{i,j} \\
\neg(h^2_{i,j} \wedge v^1_{i,j}) &\equiv \neg h^2_{i,j} \vee \neg v^1_{i,j} \\
\neg(h^2_{i,j} \wedge v^2_{i,j}) &\equiv \neg h^2_{i,j} \vee \neg v^2_{i,j}
\end{aligned}
$$

- Bridges have to start and end at an island

  Same idea as in the one-bridge solution - force all of $h^1_{i,j}, h^2_{i,j}, v^1_{i,j}, v^2_{i,j}$ all to be true.

- Degree constraints

  Again, we use the same idea as in the one-bridge solution - enumerate all the $\binom{k}{n}$ options. The resulting formula is in DNF and can be converted to CNF by Claim 1 resulting in an exponential increase in the amount of clauses. This was still feasible in the one-bridge case, but is no longer feasible here. In the worse case, we have an island node with required degree 4 and 4 empty slots around in, each admitting 2 possible bridges, for a total of $\binom{8}{4} = 70$ combinations. Each of those combinations has 8 variables, so the conversion to CNF produces $8^{70} \approx 1.65 \times 10^{63}$ with 8 literals each. This is infeasible for solving. In the next section, we show how to reduce the number of clauses in the CNF representation so that solving the problem is feasible.

Claim 3. For a given *Bridges* game, let A be a CNF formula representing the constraints for a given *Bridges* game as itemised above. Then a model M of A is a solution to the given game. This follows by construction of the constraints - we simply translate each game constraint into an equivalent SAT constraint.

Linear size DNF to CNF conversion

Claim 1 presents a way to convert a DNF formula to an *equivalent* (Definition 3) CNF formula. G.S. Tseitin shows in [Tse68] how to convert an arbitrary formula to an *equisatisfiable* CNF formula such that the CNF formula scales linearly with the size of the arbitrary formula.

Definition 7. Formulas A and B are said to be *equisatisfiable* iff either both A and B have no models, or both have some models (and hence are satisfiable).

Here we are interested at converting a formula in DNF to equisatisfiable CNF formula. The construction introduces new variables, and proceeds as follows. Given a formula in DNF:

$$A = (a_{11} \wedge a_{12} \wedge \cdots \wedge a_{1k}) \vee (a_{21} \wedge a_{22} \wedge \cdots \wedge a_{2k}) \vee \cdots \vee (a_{n1} \wedge a_{n2} \wedge \cdots \wedge a_{nk})$$

construct a formula

$$\begin{aligned} B = (\neg z_1 \vee a_{11}) \wedge (\neg z_1 \vee a_{12}) \wedge \cdots \wedge (\neg z_1 \vee a_{1k}) \bigvee \\ (\neg z_2 \vee a_{21}) \wedge (\neg z_2 \vee a_{22}) \wedge \cdots \wedge (\neg z_2 \vee a_{2k}) \bigvee \\ \vdots \\ (\neg z_n \vee a_{n1}) \wedge (\neg z_n \vee a_{n2}) \wedge \cdots \wedge (\neg z_n \vee a_{nk}) \bigvee \\ (z_1 \vee z_2 \cdots \vee z_n) \end{aligned}$$

The idea is to introduce a new variable z_i for every clause in the DNF formula, and say that $z_i \rightarrow a_{1i} \wedge a_{2i} \wedge \ldots a_{ki}$, which is logically equivalent to the construction above. In essence, variable z_i is controlling whether to enable clause i.

Claim 4. A and B are equisatisfiable. This follows by the Tseitin construction. In addition, if an interpretation P is a model of B, then P is a model of A. This is immediate since B has constraints that look like $z_i \rightarrow a_{1i} \wedge a_{2i} \wedge \ldots a_{ki}$, so if $P(z_i) = \top$ then $\forall_{1 \leq j \leq k} P(a_{ij}) = \top$ and hence the clause $(a_{i1} \wedge a_{i2} \wedge \cdots \wedge a_{1k})$ evaluates to $\top$ under P and so A is equivalent to $\top$ since it is in DNF.

Claim 5. For a given *Bridges* game, let A be a CNF formula representing the constraints for a given *Bridges* game as itemised above, and let B a CNF formula got by using the Tseitin conversion on the island degree constraints part of A. A model of B is then a solution to the original game.

Proof. Assume there exists a solution to the given game. Then A is satisfiable and a model M of A is a solution to the original game by Claim 3. A and B are equisatisfiable by the Tseitin construction, so there exists a model M' of B. By Claim 4, M' is also model of A, and so M' represents a solution to the original game. If the game has no solutions, then A has no models, and neither does B by equisatisfiability. □

5.2.4 Reducing the number of clauses

My first attempt at formulating *Bridges* as a SAT problem is correct, but it uses more variables and creates more clauses than necessary, even after translation to a small equisatisfiable formula. In this section, I show how to think of *Bridges* in a graph-theoretical sense and hence express it using less variables and constraints. We will use the game setup shown in Figure 5.3 to demonstrate the technique.

The intuition is to create variables $v^1_{(i,j),(k,l)}$ and $v^2_{(i,j),(k,l)}$ iff the island node (i, j) can directly on a vertical line see a different island node (k, l). The same intuition is used for the horizontal case.

More formally, let $T = \{1 \dots N\}^2$ be a set of all tiles in an $N \times N$ *Bridges* game. Then T is partially ordered under the lexicographical ordering $(i, j) \leq (k, l) \leftrightarrow i < k$ or $(i = k$ and $j \leq l)$ and $(i, j) < (k, l) \leftrightarrow i < k$ or $(i = k$ and $j < l)$. I will use a bold font to represent a valid grid location tuple using a single variable, for example $(i, j) = \mathbf{x} \in T$.

Let island($\mathbf{x}$) be a predicate representing that there is an island on tile $\mathbf{x}$, and let $I = \{\mathbf{x} \mid \mathbf{x} \in T, \text{island}(\mathbf{x})\}$, the set of nodes that are islands. Then, let $H(i, j) = \{(i, j) \mid 1 \leq k \leq N, k \neq j, (i, k) \in I\}$ be the set of all island nodes that share the same horizontal line as an island at (i, j). Similarly, let $V(i, j) = \{(i, j) \mid 1 \leq k \leq N, k \neq i, (k, j) \in I\}$ be the set of all island nodes that share the same vertical line with as an island at (i, j). We can then define the direct horizontal $H_d(\mathbf{x})$ and direct vertical $V_d(\mathbf{x})$ neighbours of island node $(i, j) = \mathbf{x}$.

$$H_d(\mathbf{x}) = \{\max\{\mathbf{p} \mid \mathbf{p} \in H(\mathbf{x}), \mathbf{p} < \mathbf{x}\}\} \bigcup \{\min\{\mathbf{p} \mid \mathbf{p} \in H(\mathbf{x}), \mathbf{p} > \mathbf{x}\}\}$$
$$V_d(\mathbf{x}) = \{\max\{\mathbf{p} \mid \mathbf{p} \in V(\mathbf{x}), \mathbf{p} < \mathbf{x}\}\} \bigcup \{\min\{\mathbf{p} \mid \mathbf{p} \in V(\mathbf{x}), \mathbf{p} > \mathbf{x}\}\}$$

Then, $N_d(\mathbf{x}) = H_d(\mathbf{x}) \cup V_d(\mathbf{x})$ is the set of direct neighbours of island node $(i, j) = \mathbf{x}$. By the definition of the bridges game, $1 \leq |N_d| \leq 4$. As an example, for island node B in Figure 5.3, we have $H_d(B) = \{A, C\}$,$V_d(B) = \{F\}$, and so $N_d(B) = \{A, C, F\}$.

To partially recap, and further formalise *Bridges*, for a given game we have:

- N - the size of one side of a playing board. There are a total of N^2 grid intersections
- $T = \{1 \dots N\}^2$ - the set of all grid intersections
- $I \subseteq T$ - the set of grid intersections that have an island on them.
- $N_d(\mathbf{x})$ - the set of islands directly accessible from grid intersection $\mathbf{x}$.

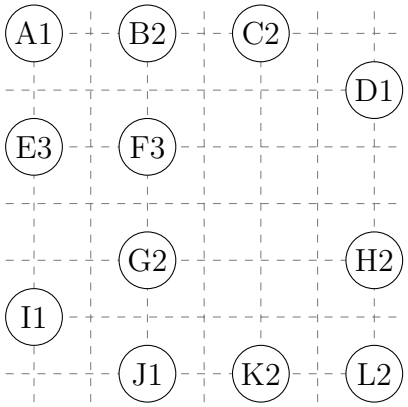


Figure 5.3 – An initial board state. The required island degree is prefixed with a letter so that we can refer to a specific island by its assigned letter

- $D : I \mapsto \{1 \dots 8\}$, the mapping giving each island its degree constraint

Then, an *instance* of the game *Bridges* is a tuple $B = (N, I, D)$.

We will now use a graph to represent all the possible connections between all pairs of island nodes for a given game $B = (N, I, D)$.

Definition 8. A *graph* $G = (V, E)$ is a tuple consisting of a set of vertices V together with a set $E \subseteq V^2$ of edges. An *edge labelling* function $L_E : E \mapsto X$ for a graph G gives a label from the set X of labels to edges of G. A *node labelling* function $L_V : V \mapsto Y$ gives a label from the set Y of labels to nodes of G.

Let $G = (I, C)$ be a graph, where

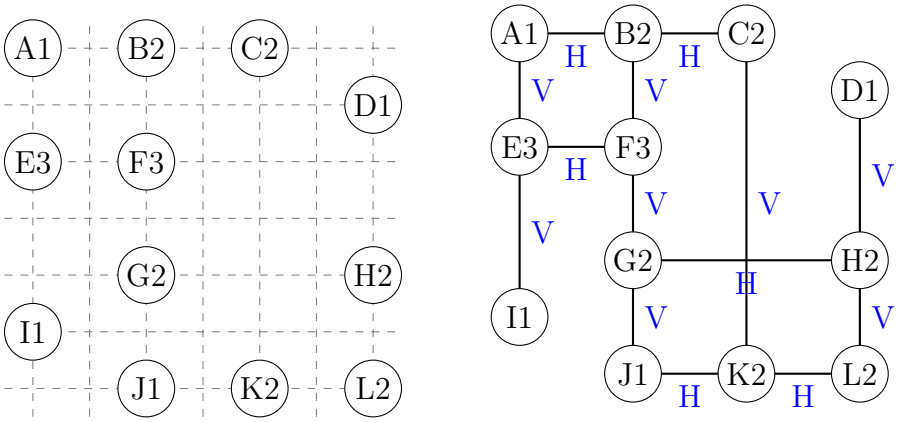
$$C = \bigcup_{\mathbf{i} \in I} \{\mathbf{i}\} \times N_d(\mathbf{i})$$

is the edge set of all possible connections between islands. Define an edge labelling function L_C,

$$L_C((\mathbf{x}, \mathbf{y})) = \begin{cases} \texttt{H} & \text{if } \mathbf{y} \in H_d(\mathbf{x}) \\ \texttt{V} & \text{if } \mathbf{y} \in V_d(\mathbf{x}) \end{cases}$$

which tells us the type of bridge (`H` for horizontal, and `V` for vertical) that can be made between nodes at $\mathbf{x}$ and $\mathbf{y}$. Additionally, use $L_I = D$ as a node labelling function. For an example of transforming a game B to a connection graph G, see Figure 5.4. In Figure 5.4b I label the nodes with a tag that gives them a name and the node degree. For example, the node labelled A1 is actually node $(1, 1)$ and $D((1, 1)) = 2$ the degree constraint.

We can now express the constraints for a game $B = (N, I, D)$ in terms of the definitions in this section.



(a) An initial state for the Bridges game B

(b) A connection graph created from B

Figure 5.4

- Propositional variables

  We need a propositional variable for every possible connection between two island nodes. Moreover, since each node can have up to 2 bridges, we need twice as many variables as we would if only 1 bridge was allowed. We also only need one variable to express possible bridges $(\mathbf{a}, \mathbf{b})$ and $(\mathbf{b}, \mathbf{a})$ since our graph is undirected. Hence, the propositional variables are:

$$\{h^1_{\mathbf{a},\mathbf{b}}, h^2_{\mathbf{a},\mathbf{b}} \mid (\mathbf{a}, \mathbf{b}) \in C, \mathbf{a} \leq \mathbf{b}, L_C(\mathbf{a}, \mathbf{b}) = \texttt{H}\} \bigcup$$
$$\{v^1_{\mathbf{a},\mathbf{b}}, v^2_{\mathbf{a},\mathbf{b}} \mid (\mathbf{a}, \mathbf{b}) \in C, \mathbf{a} \leq \mathbf{b}, L_C(\mathbf{a}, \mathbf{b}) = \texttt{V}\}$$

- Bridges cannot cross

  I say that a *bridge* is an element of the edge set C. The intuition is to state that we can't have bridges $x, y \in C$ at the same time if they cross when drawn like the graph in Figure 5.4b. While this is very clear from the illustration, I define it formally in terms of bridges x and y *crossing*.

  Definition 9. Let $P : C \mapsto \mathcal{P}(T)$ be the function that takes a bridge $(\mathbf{a}, \mathbf{b})$ and returns all grid intersections that the bridge occupies.[2] Bridges $x, y \in E$ are said to *cross* iff $P(x) \cap P(y) \neq \{\}$. It then follows that x crosses y iff y crosses x.

  Then, for every pair $(x, y) = ((\mathbf{i}, \mathbf{j}), (\mathbf{k}, \mathbf{l}))$ of bridges, if x crosses y, $\mathbf{i} \leq \mathbf{j}$ and $\mathbf{k} \leq \mathbf{l}$, add constraints to encode that in a correct solution, such a crossing is not allowed to happen. If $L_C(x) = \texttt{H}$, add the constraints:

$$\neg h^1_x \vee \neg v^1_y \qquad \neg h^1_x \vee \neg v^2_y \qquad \neg h^2_x \vee \neg v^1_y \qquad \neg h^2_x \vee \neg v^2_y$$

  and similarly, if $L_C(x) = \texttt{V}$, add the constraints

$$\neg v^1_x \vee \neg h^1_y \qquad \neg v^1_x \vee \neg h^2_y \qquad \neg v^2_x \vee \neg h^1_y \qquad \neg v^2_x \vee \neg h^2_y$$

[2] This is just all the points on the line segment connecting a with b

The above encodes exactly that no bridges are allowed to cross in a solution.

- Degree Constraints

  Now I need to encode the relevant degree constraints, that is, make sure that in a solution, an island **a** with $L_I(i) = n$ connects to exactly n bridges. We encode this using the same enumeration construction as shown in subsection 5.2.3 ensuring that we use the Tseitin translation to reduce the number of clauses. The total number of clauses generated this way is $\sum_{i \in I} \binom{k}{L_I(i)} k + 1$ where $k = 2N_d(i)$, the total number of possible bridges. This is because there are $\binom{k}{L_I(i)}$ ways to choose a particular set of $L_i(i)$ bridges, and each such possibility is a conjunctive clause with k variables and one extra clause introduced during the Tseitin translation.

The above construction produces a formula A such that if P is a model of A then P represents a solution to the original *Bridges* game. In the following section, I describe the implementation of formula generation, and try to apply the continuous time SAT solver to solve a simple instance of the game.

5.3 Experimental Results for *Bridges*

I have implemented the generation of CNF formulas in Python for the all the encodings for *Bridges* presented in the previous section, but will only study in more detail the final, most optimal, graph-based representation.

As a comparison, I present in Table 5.1 the number of clauses generated using the various methods for the variable, degree, and crossing constraints. We see instantly that the naive methods without using the Tseitin conversion produce an exponential number of clauses for degree constraints, and hence are unsuitable, although they were a motivating example along the way to finding a better representation.

5.3.1 Data source for *Bridges*

To run experiments on my encoding, I used the games available from `http://puzzle-bridges.com`. The website classifies puzzles according to a difficulty between 0 and 11. Difficulties 0,1,2 are on boards size 7×7, 3-5 on 10×10, 6-8 on 15×15, and 9-11 on boards sized 25×25. I scraped the first 200 puzzles of each difficulty to use in my experiments for a total of 2400 games. On the website, game boards are displayed as an image, and hence I wrote a simple image recognition parsing tool to parse the boards into a format usable by the other tools I have built.

	Variable	Degree	Crossing
Naive 1-Bridge	$2N^2$	$\sum_{i \in I} 4^{\binom{4}{L_I(i)}}$	$\mathcal{O}(N^2)$
Naive 2-Bridge	$4N^2$	$\sum_{i \in I} 8^{\binom{8}{L_I(i)}}$	$\mathcal{O}(N^2)$
Naive 2-Bridge-Tseitin	$4N^2$	$\sum_{i \in I} \binom{8}{L_I(i)} 8 + 1$	$\mathcal{O}(N^2)$
Graph-Based Tseitin	$\|C\|$	$\sum_{i \in I} \binom{k}{L_I(i)} k + 1$	$\mathcal{O}(\|C\|)$

Table 5.1 – The number of clauses generated for each of the different types of constraints (Variable, Degree and Connectivity) for the different encoding methods described.

5.3.2 Testing the encoding

The first interesting statistic to look at is how many clauses are generated per game given a board size. I show the results in Figure 5.5. The more difficult the problem is, the higher the variance in the number of clauses, with some problems on a 25×25 board requiring as many clauses as a problem on a 10×10 board.

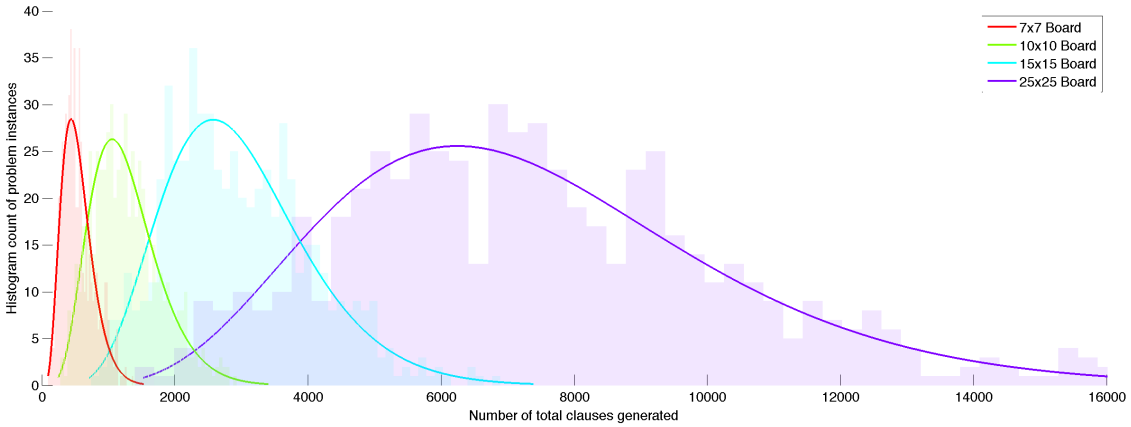


Figure 5.5 – A histogram with a distribution fit showing the number of clauses generated for *Bridges* games with various board sizes using the most optimal SAT encoding I have developed.

	Mean	Standard Deviation
7x7	537	238
10x10	1263	515
15x15	2975	1093
25x25	7420	2880

Table 5.2 – Means and standard deviations for total number of clauses generated for various board sizes using the optimal SAT encoding

To ensure that my SAT encoding is correct, I solve some games from my ex-

perimental game data set using MiniSat[3], the open source minimalist and fast SAT solver which can easily handle all the formulas I generate and solve them in no longer than 0.01 seconds for the games on 25×25 boards with difficulty 11. Figure 5.6 shows a solution to a large instance of the game confirming that my encoding is correct, and giving me the ability to easily get the high-score on `http://puzzle-bridges.com`.

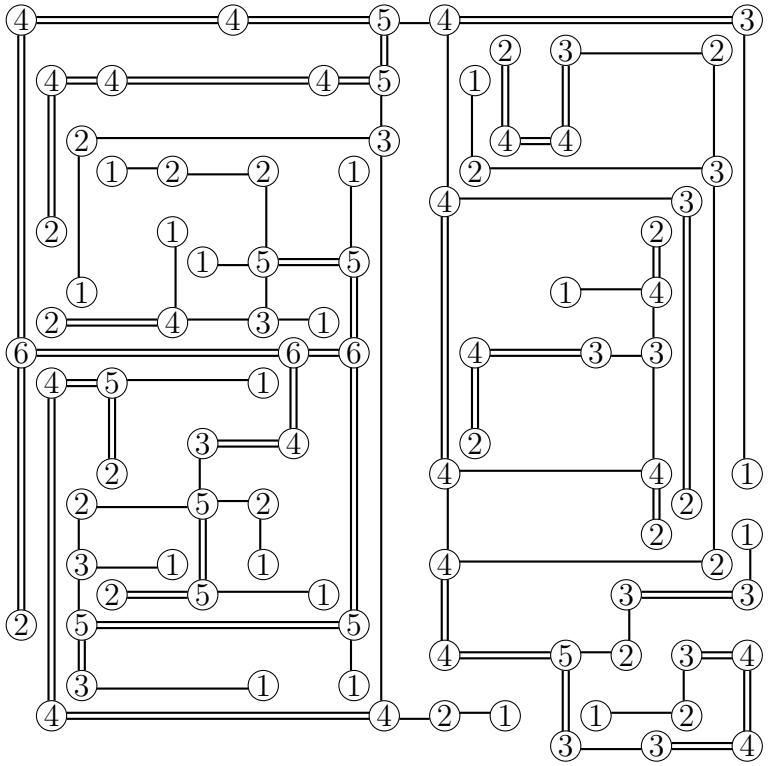


Figure 5.6 – A solution to a difficult instance solved by Minisat with my encoding

5.3.3 Solving using the SAT dynamical system

I now look at whether the continuous dynamical system for SAT solving will be able to converge to a solution of *Bridges*. The motivation is that while it was shown to work relatively well for random SAT instances, other applications weren't looked at in detail. This evaluates whether such a system for SAT solving can deal with non-random problems.

I have tried using the bounded dynamical system from [MTER12] (shown in section 4.2), but irregardless of the A and B parameters, **the system always got stuck in a limit cycle**. I conclude that while this system does solve random 3SAT instances well, it does not cope with the SAT encoding to solve *Bridges*.

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

In [ERT12], the authors apply the unbounded dynamical system SAT solver to solve Sudoku games. To evaluate my encoding and puzzle difficulty, I solve 200 puzzles each in difficulties Easy, Normal and Hard on a board size 7×7 and 10×10 using the unbounded dynamical system for SAT as shown in section 4.1. Solution times for larger difficulties proved too computation-intensive for the model.

Method

I solve 200 puzzles each in difficulties Easy, Normal and Hard on a board size 7×7 and 10×10, with a continuous time limit of 175, and store whether the system found a solution, and if so, how much continuous time it took. The integration was done using Matlab's `ode45` integrator (with a relative tolerance of 10^{-3}), which uses an adaptive step size to reduce error. In [ERT12], the authors show that the length of chaotic transients in hyperbolic systems such as this one decays exponentially. That is, let $p(t)$ be the probability that the system has not found a solution by time t. Then $p(t) \sim e^{-kt}$ where k is called the escape rate of the system. Using my collected data, I fit the function e^{-kt} and hence the find the escape rate, which I use to quantify the difficulty of the puzzle. Figure 5.7, Figure 5.8, and Table 5.3 show the experimental results.

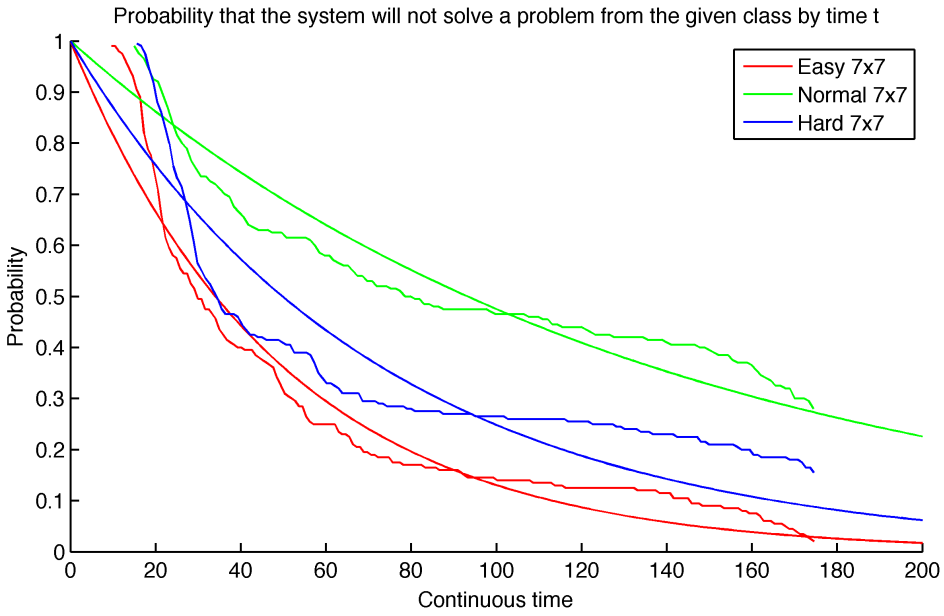


Figure 5.7 – Probability that the system will not solve a problem from a given class by continuous time t for 7×7 boards. The difficulty is taken from games on `http://puzzle-bridges.com`

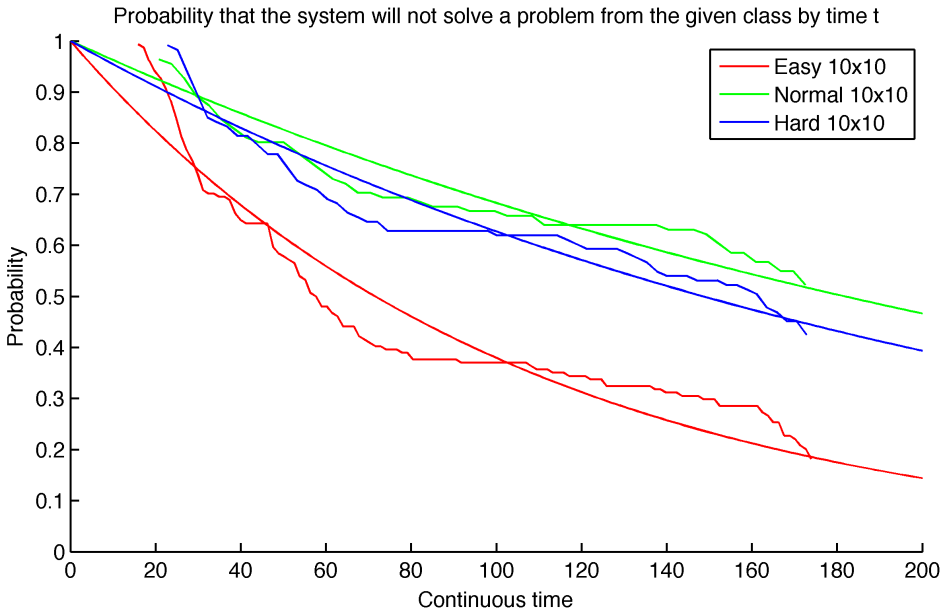


Figure 5.8 – Probability that the system will not solve a problem from a given class by continuous time t for 10×10 boards. The difficulty is taken from games on `http://puzzle-bridges.com`

	k	$-log_{10}k$
Easy 7x7	0.0204	1.69
Hard 7x7	0.0139	1.86
Easy 10x10	0.0097	2.01
Normal 7x7	0.0076	2.12
Hard 10x10	0.0047	2.33
Normal 10x10	0.0038	2.42

Table 5.3 – The escape rate k of the dynamical system used to solve the given class of problems, together with its negative log to indicate puzzle hardness on a logarithmic scale. The puzzles are listed in order of increasing puzzle hardness as indicated by the escape rate k.

Implications for hardness

The interesting result is that in both the 7×7 and 10×10 boards, the game that is claimed as "Hard" (for humans) has a higher escape rate k than the games marked as "Normal". See Figure 5.9 for a diagram showing the functions $p(t) \sim e^{-kt}$ for various difficulties and board sizes.

We conjecture that this is because of the *clause-to-variable* $\frac{M}{N} = \alpha$ ratio. α is a very good indicator of formula hardness, with $\alpha \approx 4.26$ marking the hardest region for any current SAT solver (see explanation and diagram in section 2.2.1).

This result is interesting since it is an example of a combinatorial puzzle where a higher puzzle hardness for humans does not translate to a puzzle more difficult

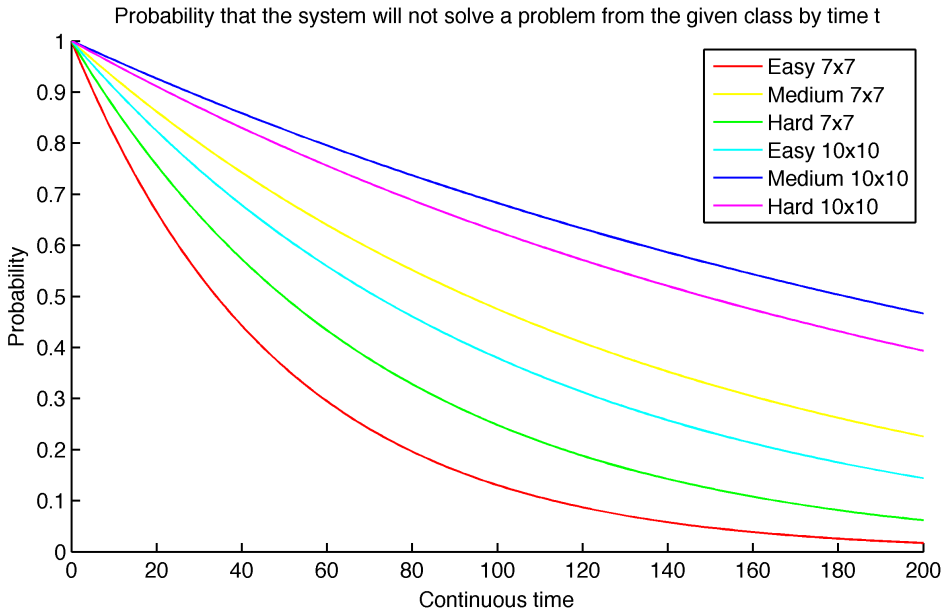


Figure 5.9 – Probability that the dynamical system for SAT will not solve a problem from a given class by continuous time t for various difficulties as taken from `http://puzzle-bridges.com`. Interestingly, an "Easy" 10×10 problem has a larger escape rate than a "Normal" 7×7, but a smaller one than than "Hard" 7×7

for a computer to solve. In [ERT12], the authors show how Sudoku games deemed more difficult for humans are actually more difficult for the computer to solve (indicated by a lower escape rate), while this is not the case for *Bridges*. In *Bridges*, a "Hard" game has more constraints than a "Normal" game, making it more difficult for a human to keep track of all the constraints and whether a next possible move will violate any constraints. For SAT algorithms, the extra number of constraints increases the α ratio, which guides the SAT solver more quickly towards a solution. See Figure 5.10 for probability distributions of α for different difficulty games on a 7×7 board. "Normal" difficulty produces problems with an expected value for α just around the hardest region of ≈ 4.3. "Hard" problems, due to the increased number of constraints, have a higher expected value for α and hence are easier to solve using a SAT solver.

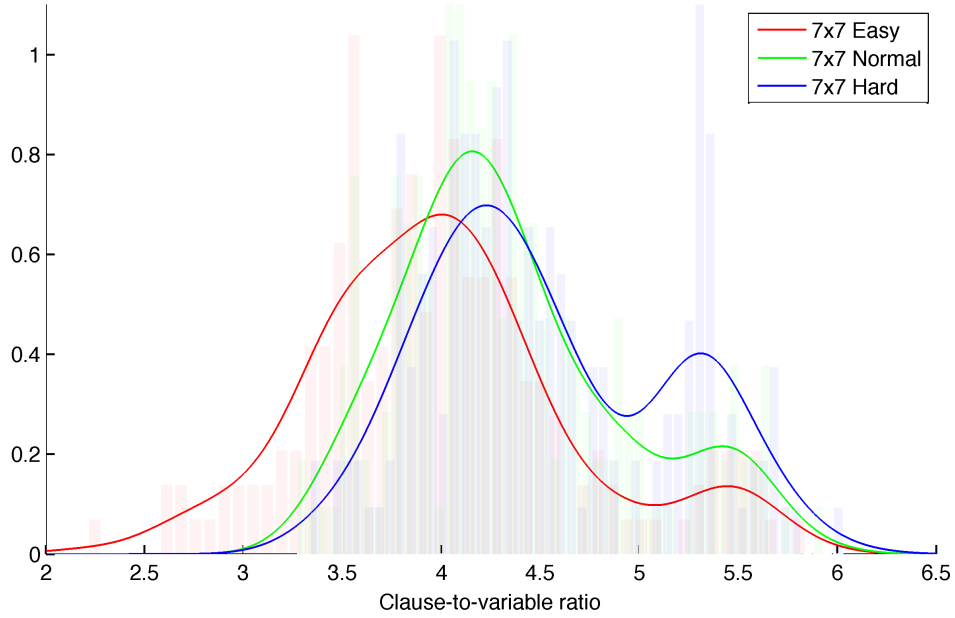


Figure 5.10 – Probability density functions for α ratio for "Easy", "Normal", and "Hard" problems on a 7×7 board. Easy problems tend to have a lower α. The expected value for "Normal" difficulty is just around 4.3, the hardest region for SAT, while for "Hard" problems, the extra additional constraints shift the expected α value to the right, making the problem easier again.

Solving SAT with dynamical systems

In this chapter, we have shown how to encode N-Rooks and N-Queens in CNF and found that solving the resulting problems using all the dynamical systems is feasible.

We then analysed a much more interesting and challenging problem, the logic game *Bridges*. We developed a CNF encoding for it, and solved the resulting problems using the unbounded dynamical system for SAT. Finally, we analysed system escape rates for different difficulty problems, and reached interesting conclusions regarding difficulty for humans vs difficulty in terms of SAT.

Chapter 6

Conclusion and Future Work

6.1 Achievements

The purpose of the project was an exploration of combinatorial optimization using dynamical systems, with a focus on boolean k-SAT. We showed how to express popular combinatorial optimization problems such as N-Queens and the Travelling Salesman Problem as dynamical systems with fixed points corresponding to solutions to the original problem.

For boolean k-SAT, we have shown and further evaluated two previously studied systems, and contributed a third system that has all its variables bounded, and is able to solve some problems that the previously studied bounded version could not. The motivation for keeping the variables bounded is that in a neural computation setting, such as the brain, or an analog circuit, every computation will be bounded so it would be infeasible to run a potentially unbounded computations such as the one in [ERT11].

To evaluate the idea of solving SAT using continuous-time dynamical systems, the report contributes:

- Three progressively better CNF encodings for the game *Bridges* together with their analysis for games of various difficulties
- An evaluation of the escape rates of the SAT dynamical system when solving various difficulty *Bridges* games with the contributed encoding.

We find that the bounded dynamical systems as shown in [MTER12], while suitable for solving random k-SAT instances, was not able to solve almost any CNF encoding for *Bridges*.

Using the unbounded dynamical system, we compute escape rates given games

of different difficulty, and find that, as expected, "Easy" games are indeed the easiest for the dynamical system to solve, but "Hard" games get a higher escape rate than "Normal" games due to the increased number of constraints but a similar number of variables. This increases the *clause-to-variable* ratio and makes the resulting formula easier to solve for a computer, letting us conclude that increasing puzzle hardness for a human does not necessarily correspond to an increased puzzle hardness for a computer.

6.2 Future Work

The following are possibilities for future work in this topic:

- The game *Bridges* is very similar to the problem of FPGA routing which involves finding a way to connect different points on a circuit using only certain allowed connection places and in a way that no connections cross each other. An extension of this project could be to develop an encoding for FPGA routing and see whether solving it with the dynamical system exhibits interesting properties such as a measure of difficulty using escape rates.
- Studying the effect of noise on the SAT dynamical system. It is possible that transforming the deterministic dynamical systems as presented in this paper to stochastic dynamical systems might help it get out of local minima without sacrificing performance.
- While a neural system such as the brain is not able to run algorithms the way a computer can, it does have the ability to run computations using different unit types such as integrators, or threshold neurons. An optimization problem expressed as a dynamical system should be possible to express and simulate in a network of spiking neurons. A proposed future work is then to investigate the possibility of creating a network of spiking neurons capable of solving SAT using a dynamical system approach as presented in this report. Furthermore, a network of spiking neurons will usually be subject to constant noisy firing, which will also give insight to effect of noise for the system.

6.3 Final Remarks

The project was an exploration of solving combinatorial optimization problems using dynamical systems, with a focus on solving boolean k-SAT and a novel application to the logic game *Bridges*. As a final comment, we note that in

their current form, none of these systems are actually any good for solving their corresponding combinatorial problems - they are orders of magnitude slower than their deterministic algorithmic counterparts, and many times we are not even guaranteed to find a solution. Nevertheless, the motivation is that casting these discrete problems as dynamical systems over the reals makes it possible to solve them in a neural medium such as the brain, and gives new insights to neural computation.

Bibliography

[Ami96] Daniel J. Amit. The hebbian paradigm reintegrated: Local reverberations as internal representations, 1996.

[And] Daniel Anderson. Hashiwokakero is np-complete.

[Bra86] Valentino Braitenberg. *Vehicles*. MIT Press, 1986.

[BW01] Nical Brunel and Xiao-Jing Wang. Effects of neuromodulation in a cortical network model of object working memory dominated by recurrent inhibition. *Journal of Computational Neuroscience*, 2001.

[DSS00] Daniel Durstewitz, Jeremy Seamansm, and Terrence Sejnowski. Neurocomputational models of working memory. *Nature Neuroscience*, 3, 2000.

[Eda] Abbas Edalat. Neural networks and their applications. pdf.

[ERT11] Mária Ercsey-Ravasz and Zoltán Toroczkai. Optimization hardness as transient chaos in an analog approach to constraint satisfaction. *Nature Physics*, 2011.

[ERT12] Maria Ercsey-Ravasz and Zoltán Toroczkai. The chaos within sudoku. *Nature*, 2012.

[ESC+12] Chris Eliasmith, Terrence Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Charlie Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *Science*, 338, November 2012.

[Heb49] Donald Hebb. *The organization of behavior*. Wiley and Sons, 1949.

[Hop06] Frank Hoppensteadt. Predator-prey model. `http://www.scholarpedia.org/article/Predator-prey_model`, 2006.

[HT85] J.J. Hopfield and D.W. Tank. "neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[Izh03] Eugene Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6), November 2003.

[Izh07] Eugene Izhikevich. *Dynamical Systems in Neurosience*. MIT Press, 2007.

[KSS07] Lukas Kroc, Ashish Sabharwal, and Bart Selman. Survey propagation revisited. 2007.

[KSS08] Henry Kautz, Ashish Sabharwal, and Bart Selman. *Handbook of Satisfiability*. IOS Press, 2008.

[KSS+11] Yuichi Katori, Kazuhiro Sakamoto, Naohiro Saito, Jun Tanji, Hajime Mushiake, and Kazuyauki Aihara. Representational switching by dynamical reorganization of attractor structure in a network model of the prefrontal cortex. *PLoS Computational Biology*, 7(11), November 2011.

[Man95] Jacek Mandziuk. Solving the n-queens problem with a solving the n-queens problem with a binary hopfield-type network. *Biological Cybernetics*, 1995.

[MTER12] Botond Molnar, Zoltan Toroczkai, and Maria Ercsey-Ravasz. Continuous-time neural networks without local traps for solving boolean satisfiability. *IEEE*, 2012.

[neu] Neuron image. `http://www.wpclipart.com/medical/anatomy/cells/neuron/neuron_T.png`.

[Roj96] Raul Rojas. *Neural Networks - A Systematic Introduction*. Springer-Verlag, 1996.

[Sha] Murray Shanahan. A funny way to get satisfaction.

[SI10] Botond Szatmary and Eugene Izhikevich. Spike-timing theory of working memory. *PLoS Computational BiologyS*, 6(8), 2010.

[TCW02] Jesper Tegner, Albert Compte, and Xiao-Jing Wang. The dynamical stability of reverberatory neural circuits. *Biological Cybernetics*, 2002.

[Tse68] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.

[Wan01] Xiao Jing Wang. Synaptic reverberation underlying mnemonic persistent activity. *Trends in Neurosciences*, 24(8), August 2001.

[WW] Benjamin Wah and Zhe Wu. The theory of discrete lagrange multipliers for nonlinear discrete optimization.

[ZKLF93] David Zipser, Brandt Kehoe, Gwen Littlewort, and Joaquin Fuster. A spiking network model of short-term active memory. *The Journal of Neuroscience*, 13(8), 1993.