Master thesis

Asynchronous Clause Exchange for Malleable SAT Solving

Malte Sönnichsen

Date: February 1, 2022

Supervisors: Prof. Dr. Peter Sanders
Dominik Schreiber
Dr. Markus Iser

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

This thesis presents and evaluates the use of gossiping communication and multi-armed bandit strategies for clause exchange in portfolio based parallel SAT solving with malleability of jobs. Malleability of jobs is the ability to add or remove processing power during computation.

Modern parallel SAT solver scales up to thousands of cores. In modern portfolio based parallel SAT solving, clause exchange is done either in an all-to-all manner over all nodes or along a fixed communication graph. Our approach is to let the SAT solvers decide from which nodes they want to receive clauses. This is done by applying the concept of gossiping communication to clause exchange in portfolio based parallel SAT solving and by only communicating useful clauses to neighbors which have benefitted from an exchange in the past. We compare an informed neighbor selection with an uninformed one, i.e., neighbor selection with uniform probability. We choose a communication graph which is both highly connected and expandable. High connectivity is important for clause distribution. The expandability is important for malleability of jobs.

Our approach did not match the performance of the state-of-the-art parallel SAT solver Mallob. However, we observed formations of clusters in our communication graph and a decrease of exchanged clause volume while improving the performance. We achieve this using an informed neighbor selection instead of the uninformed one. We conclude that our concept of a more organic clause exchange is promising and has potential.

Zusammenfassung

Diese Arbeit präsentiert und evaluiert die Verwendung von *gossiping* Kommunikation und mehrarmigen-Banditen Lösungen für Klauselaustausch in Portfolio basiertem parallelen SAT solving mit Verformbarkeit von Jobs. Die Verformbarkeit von Jobs bezieht sich auf die Möglichkeit Rechenleistung während der Berechnung hinzuzufügen oder zu entfernen.

Moderne parallele SAT solver skalieren bis zu tausenden von Kernen. Der Klauselaustausch findet in modernen parallelen SAT solvern entweder durch Versenden der Klauseln von allen Knoten zu allen Knoten oder zu allen Nachbarn in einem festen Kommunikationsgraphen. In unserem Ansatz wird die Entscheidung von welchem Knoten Klauseln empfangen werden sollen, den jeweiligen Knoten überlassen. Das wird durch die Anwendung des Konzepts der *gossiping* Kommunikation auf den Klauselaustausch in Portfolio basierten parallelen SAT solving erreicht. Außerdem werden ausschließlich nützliche Klauseln mit Nachbarn, welche bereits von einem Austausch profitiert haben, kommuniziert. Wir vergleichen eine informierte Nachbarauswahl mit einer uninformierten, d.h. eine Nachbar Auswahl mit gleichverteilter Wahrscheinlichkeit. Wir verwenden einen Kommunikationsgraphen welcher hochzusammenhängend und zugleich erweiterbar ist. Die Eigenschaft des Graphen hochzusammenhängend zu sein, ist wichtig für die weitreichende Verteilung von Klauseln. Die Erweiterbarkeit des Graphen ist bedeutsam für die Verformbarkeit der Jobs.

Unser Ansatz erreicht nicht die Leistung des aktuell besten parallelen SAT solver Mallob. Allerdings konnten wir die Bildung von Clustern in unserem Kommunikationsgraph beobachten. Außerdem konnten wir eine Verringerung der Anzahl an ausgetauschten Klauseln bei gleichzeitiger Beschleunigung des Lösens der SAT-formeln erzielen. Dies wurde durch die Verwendung einer informierten Nachbarauswahl, anstelle der uninformierten Nachbarauswahl, erreicht.

Insgesamt erwies sich dieser Ansatz eines organischeren Klauselaustausches als Vielversprechend, bedarf jedoch weiterer Forschung.

Contents

1 Introduction

The existence of fast SAT solver makes the reduction to Boolean Satisfiability (SAT) interesting for many kinds of problems. Therefore, SAT solvers are used in a vast variety of domains including planning and scheduling [27], hardware and software verification [17], and cryptography [21]. The ever-increasing complexity of the problems to be solved requires fast SAT solvers. Due to stagnation in single core speed but increase in CPU core counts, parallel SAT solving is the logical next step. In recent years, the interest in parallel SAT solving has continued to grow [12, 9, 8, 3, 28]. Nowadays, there is also interest in massive parallelized on demand SAT solving in HPC environments with several thousand nodes. The International SAT Competition features a cloud track since 2020 [10] that reflects this use case. Especially Mallob stands out with winning the cloud track in 2020 [13] and 2021 [14].

1.1 Motivation

Mallob uses a synchronous clause sharing mechanism. Periodically all learned clauses get merged and the merged buffer is distributed over all nodes and their SAT solvers. Thus, a node receives a potentially large portion of useless clauses.

Our hypothesis is that SAT solvers do not benefit from all but only some of the other SAT solvers and their learned clauses. Therefore, a potentially large part of the communication volume could be eliminated without disadvantage. In addition, the reduced clause volume for each node could lead to a solving speedup, since the overhead for adding external learned clauses is reduced. This reduction in clause volume could also be exploited to distribute clauses more frequently, allowing other SAT solvers to benefit from useful clauses earlier.

In this thesis we build a more organic communication framework for exchanging learned clauses. Nodes communicate in a decentralized peer to peer manner. For technical reasons nodes communicate only with direct neighbors and the communication graph remains fixed. However, each node maintains its own buffer and decides itself which neighbour is requested for clauses. This could lead to formation of clusters of SAT solvers, where information exchange is fast and efficient.

1.2 Contribution

Our contribution is a derivation of closed formulas for calculation of neighbor IDs for a given node ID in a hexagonal grid graph, an analysis of clause exchange in a decentralized peer-to-peer communication network, and applying multi-armed bandit (MAB) strategies to clause sharing. Furthermore the code is added to Mallob as a swappable component.

1.3 Structure of Thesis

Chapter 1 outlines the interest in parallel SAT solving and motivates our approach of improving the state-of-the-art massive parallel SAT solver. In chapter 2 we formalize definitions and notations used throughout the thesis. Moreover, in chapter 2 we introduce the reader to the topic of SAT solving and explain how modern SAT solver and parallel SAT solver work. Chapter 3 is about our approach of improving the state-of-the-art massive parallel SAT solver Mallob. We then test and analyze our approach in chapter 4. Finally, in chapter 5 we discuss our approach and give suggestions about future work.

2 Fundamentals

In this chapter we introduce concepts and definitions used throughout this thesis. First, we explain the Boolean Satisfiability (SAT) problem and related terms. Subsequently, we explain how this problem can be solved and present one modern SAT solver. Finally, we show approaches on how to parallelize SAT solving and present two practical parallel SAT solvers.

2.1 Boolean Satisfiability

The Boolean Satisfiability (SAT) problem is about determining if there exists an assignment that satisfies a given Boolean formula. A brief introduction to the most important terms:

Boolean variable
Variable x with two possible values $x \in \{\texttt{true}, \texttt{false}\}$.

Literal
Boolean variable x or the negation $\bar{x}$.

Clause
Formula C of literals with only disjunction operators $x_1 \vee \cdots \vee x_n$.

Conjunctive Normal Form (CNF)
Formula of conjunctions of clauses $C_1 \wedge \cdots \wedge C_n$.

2.2 SAT Solving

The Boolean Satisfiability (SAT) problem is NP-complete. A problem is NP-complete if (a) a solution can be verified in polynomial time and (b) all other problems for which (a) is true can be reduced to this problem in polynomial time. Thus, it can be used to solve problems of an entire problem class by reducing other problems to SAT. Therefore, building a solver for this problem is attractive. Firstly, we explain the simple but not practical algorithm for solving SAT problems. After that, we explain the state of the art solving algorithm.

2.2.1 Davis–Putnam–Logemann–Loveland

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a simple depth first search algorithm with backtracking [1, Chapter 3.5]. Intuitively it works by doing the following:

1. Choose a variable.
2. Assign `true` or `false` to the variable.
3. Simplify the formula.
4. Check satisfiability.
5. Do chronological backtracking if not satisfiable.

Furthermore, the DPLL algorithm incorporates *unit propagation* before checking for satisfiability. In *unit propagation* each unit clause gets propagated, i.e., we set variables of unit clauses to satisfy the respective unit clauses. This allows an early termination in the search tree.

2.2.2 Conflict Driven Clause Learning

We outline the important parts of Conflict Driven Clause Learning (CDCL) that are relevant for our distributed clause sharing algorithm. More precisely, we explain how clauses are learned and the Literal Block Distance (LBD) metric for clause quality [2].

CDCL [1, Chapter 4] builds upon DPLL and its depth first search algorithm with backtracking and unit propagation. However, one crucial difference for our approach is that this algorithm learns conflict clauses.

The technique which enables clause learning is called conflict analysis [1, Chapter 4.3.1.1.1]. The CDCL algorithm maintains an implication graph. This graph enables direct retrieval of responsible literals for each unit clause propagation. Furthermore, the sinks of this graph represent decisions about variable assignments. Thus, if a falsified clause is identified, the implication graph shows the responsible decisions that led to the falsified clause. A learned conflict clause is the disjunction of the negated responsible decisions. These conflict clauses prune the search space.

Redundant clauses from conflict analysis must be handled with care, as too many clauses can slow down the search and clog up memory. Therefore, modern solvers use a metric called LBD to assess the quality of conflict clauses [2]. The number of different decision levels of the literals in a conflict clause is the LBD metric. In the following we refer to learned conflict clauses as redundant clauses, since these clauses can be explained with sequences of resolution steps from the original clauses.

2.2.3 CaDiCaL

There are many well-performing SAT solvers [5, 20]. For our study, we chose CaDiCaL, as it is a modern, state of the art, easy to modify solver written in C++. CaDiCaL is based on CDCL with inprocessing rules [16].

As most modern SAT solver based on CDCL, CaDiCaL deletes unimportant redundant clauses. To decide which redundant clauses should be kept and which should be deleted, CaDiCaL maintains a three-tier system.

- Tier-0 clauses (LBD ≤ 2) are kept forever.
- Tier-1 clauses ($2 <$ LBD ≤ 6) survive one round of reduction.
- Tier-2 clauses can be deleted immediately if not used since the last reduction.

This is important for us since we use this information as explained in section 3.3.

2.3 Parallel SAT Solving

There are two approaches for parallelised SAT solving.

- Cube&Conquer.
- Portfolio:
  - Pure portfolio.
  - Parallel portfolio with clause sharing.

The Cube&Conquer approach divides the search space successively [15]. Following, independently working SAT solvers can solve the resulting sub-formulas. However, distributing the workload equally is challenging, since this requires a balanced split of the search space. Furthermore, the performance of SAT solvers depends highly on the combination of parameters and instance properties. The latter is exploited by portfolio based parallel SAT solving. In portfolio based parallel Boolean Satisfiability (SAT) solving the input formula is distributed over several SAT solvers with different parameters. In addition, the SAT solvers exchange redundant clauses. The first terminating SAT solver reports its results and all other solvers terminate as well. The first portfolio solver was ManySAT [12].

2.3.1 TOPOSAT

TopoSAT [9] was the first solver which use a communication graph for clause sharing. Instead of sharing clauses in an all to all manner, nodes may only communicate with each other if they are connected by an edge. They showed that their approach scales well and they gained significant speedups for up to 256 cores. However, in TopoSAT each node sends and receives clauses from each neighbor in each epoch.

2.3.2 Mallob

Mallob builds upon HordeSat [3]. HordeSat and Mallob use an all-gather operation for clause distribution. The result of this operation is that each node receives all shared clauses in each epoch. However, this distribution is implemented in an efficient manner. Thus, Mallob scales up to 2560 cores and enables SAT solving in a HPC environment. Furthermore, Mallob [28] adds load balancing and job scheduling to HordeSat and enables SAT solving on demand.

3 Gossiping SAT Clause Communicator

In this chapter we explain our approach for applying the concept of gossiping to clause sharing. At first, we show our communication graph and explain malleability of this communication graph. After that we outline the concept of gossiping, the overall sharing architecture, and how nodes exchange clauses. We then describe the process to decide which neighbor to ask for clauses.

3.1 Communication Graph

The choice of the communication graph structure affects the effectiveness of the clause distribution and communication performance. The ideal graph is sparse and highly connected. The sparsity ensures low communication volume. The high connectivity of the communication graph ensures far-reaching distribution of learned clauses. These two properties result in an efficient communication between nodes and distribution of learned clauses. Furthermore, Mallob allocates a dynamic number of workers per job by potentially removing the most recent workers and keeping the older workers. Older workers have a lower ID. Therefore, we need a graph that is appropriate for malleability, i.e., dynamic number of workers, in that sense that the remaining workers continue to communicate effectively with each other.

We use the hexagonal grid graph as shown in Figure 3.1. Each node represents one worker. In the following we refer to workers as nodes. If two nodes are connected by an edge, we call these nodes direct neighbors. Direct neighbors can communicate with each other. The hexagonal grid graph has one core node. The core node is surrounded by an arbitrary number of rings of further nodes. This graph has some properties:

- Each node has six neighbors, except the border nodes on the outer ring.
- Each node has minimum two neighbors, if the graph has more than two nodes.
- New nodes arrange themselves in a ring around the existing nodes.
- Arbitrary number of nodes possible.

We need a function which gives us six neighbor IDs to a given node ID. Since there is no trivial directly mapping, we solve this problem in a hexagonal grid coordinate system. There are different coordinate systems for hexagonal grids [22]. Inspired by [22,

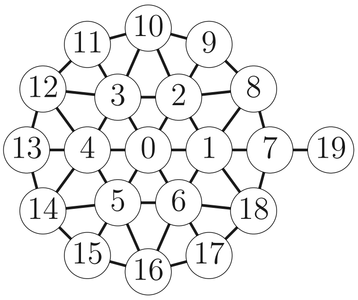


Figure 3.1: Communication graph without offset. First border node on each ring $r > 1$ has only one neighbor on inner ring.

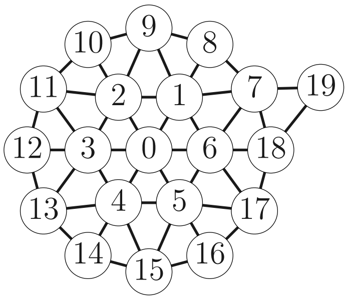


Figure 3.2: Communication graph with offset. First border node on each ring $r > 1$ has two neighbors on inner ring.

Chapter Rings] where the hexagonal grid gets traversed in a spiral manner around a core hexagon, we choose the polar coordinate system. To calculate the neighbors for a given node ID, we transform the node ID into polar coordinates and look for the polar coordinates of the neighbors and transform these coordinates back to the node ID. The number of nodes on ring x is calculated by:

$$f(x) = \begin{cases} 1 & x = 0 \\ 6x & else \end{cases}$$

The following formula calculates the last ID of a node on a given ring r:

$$g(r) = \sum_{i=0}^{r} 6i = 6 \sum_{i=0}^{r} i = 6 \frac{r(r+1)}{2} = 3r^2 + 3r$$

Through this formula we know that the IDs on ring $r > 0$ are in the interval $(g(r-1), g(r)]$. Given a ring $r > 0$ and the position $0 \leq p < 6r$ on the ring, we can calculate the ID with $h(r,p) = g(r) - 6r + 1 + p = i$.

Since we want to calculate the ring r and position p on the ring for a given ID i, we can invert $g(r)$ for $r > 0$:

$$\begin{aligned} g(r) &= 3r^2 + 3r \\ \Leftrightarrow g^{-1}(i) &= \frac{-3 + \sqrt{9 + 12i}}{6} \end{aligned}$$

The ring for a node with ID i is then given by $\lceil g^{-1}(i) \rceil = r$. To get the position on the ring, we need to subtract the value of the lowest ID on the same ring:

$$p_i = i - (g(r_i - 1) + 1) = i - 3r_i^2 + 3r_i - 1$$

To summarize for a given ID $i > 0$ we calculate the radius r_i and position p_i by:

$$\begin{aligned} r_i &= \left\lceil \frac{-3 + \sqrt{9 + 12i}}{6} \right\rceil \\ p_i &= i - 3r_i^2 + 3r_i - 1 \end{aligned}$$

For a given radius r and position p we calculate the ID i by:

$$i = 3r_i^2 - 3r_i + 1 + p_i$$

The correct polar coordinates are $(r_i, \hat{p}_i = \frac{p_i}{6r_i} 2\pi)$ where $\hat{p}_i \in [0, 2\pi)$. However, in the following we use the unscaled coordinates $(r_i, p_i)$for simpler calculations.

To calculate the neighbors for a given ID i we distinguish between three cases:

(i) The root $i = 0$.

(ii) The corner nodes on a ring $p_i \mod \frac{r_i}{6} = 0$.

(iii) The side nodes on a ring $p_i \mod \frac{r_i}{6} \neq 0$.

For case (i), the root, the neighbor IDs are $[1, 6]$. For case (ii), the corner nodes, one neighbor is one ring below, two neighbors are on the same ring, and three neighbors are one ring above. The polar coordinate $(r_{i,j}, p_{i,j})$ for neighbor j of node i one ring below is:

$$\begin{aligned} r_{i,1} &= r_i - 1 \\ p_{i,1} &= \frac{p_i}{2} \end{aligned}$$

The polar coordinates for the neighbor left and right on the same ring are:

$$\begin{aligned} r_{i,2} = r_{i,3} &= r_i \\ p_{i,2} &= p_i - 1 \mod 6r_i \\ p_{i,3} &= p_i + 1 \mod 6r_i \end{aligned}$$

The polar coordinates for the three neighbors one ring above are:

$$\begin{aligned} r_{i,4} = r_{i,5} = r_{i,6} &= r_i + 1 \\ p_{i,4} &= 2p_i - 1 \mod 6(r_i + 1) \\ p_{i,5} &= 2p_i \\ p_{i,6} &= 2p_i + 1 \mod 6(r_i + 1) \end{aligned}$$

For case (iii), the side nodes, two neighbors are one ring below, two neighbors are on the same ring, and two neighbors are one ring above. The polar coordinates for the two neighbors one ring below are:

$$
\begin{aligned}
r_{i,1} = r_{i,2} &= r_i - 1 \\
p_{i,1} &= p_i - \left\lfloor \frac{p_i}{r_i} \right\rfloor \\
p_{i,2} &= p_i - \left\lfloor \frac{p_i}{r_i} \right\rfloor - 1
\end{aligned}
$$

The polar coordinates for the neighbor left and right on the same ring are:

$$
\begin{aligned}
r_{i,3} = r_{i,4} &= r \\
p_{i,3} &= p_i - 1 \mod 6r_i \\
p_{i,4} &= p_i + 1 \mod 6r_i
\end{aligned}
$$

The polar coordinates for the two neighbors one ring above are:

$$
\begin{aligned}
r_{i,5} = r_{i,6} &= r_i + 1 \\
p_{i,5} &= p_r + \left\lfloor \frac{p_r}{r} \right\rfloor \\
p_{i,6}\cdot &= p_r + \left\lfloor \frac{p_r}{r} \right\rfloor + 1
\end{aligned}
$$

The resulting communication graph is shown in Figure 3.1.

The first node on a new ring has only one neighbor. This leads to a reduced connectivity. To ensure that each node has two neighbors, we introduce an offset per ring of one. For a given ID $i > 0$ we calculate the radius r_i and position p_i by:

$$
\begin{aligned}
r_i &= \left\lceil \frac{-3 + \sqrt{9 + 12i}}{6} \right\rceil \\
p_i &= i - 3r_i^2 + 3r_i \mod 6r_i
\end{aligned}
$$

For a given radius r and position p we calculate the ID i by:

$$
i = 3r^2 - 3r + 1 + (p - 1 \mod 6r)
$$

The resulting communication graph is shown in Figure 3.2. This is the graph we use in all our experiments.

3.2 Sharing Architecture

Our sharing architecture is based on gossiping. First introduced by Demers et al. [7] as a randomized decentralized peer to peer protocol to ensure consistency in distributed databases, gossiping is inspired by epidemic spreads. Modern database systems still work according to that principle [18]. We extend the original definition in the regard that we do not distribute all information but only what we consider useful. Our extension is that (a) a node only shares and forwards what it considers useful itself and that (b) nodes tend to communicate with each another based on how fruitful this particular exchange has been in the past. Our hypothesis is that these two characteristics lead to formation of clusters where the information gain with exchanged clauses is high.

In the following let A be the requesting and receiving node and B a sending node. Node A has one or more direct neighbors. Node A chooses one direct neighbor B and sends a request message to node B. Node B receives the request message, collects his most useful clauses, and sends the collected clauses back to node A. The clause sources for the collection are the received clauses from other nodes than A combined with the clauses recently learned by the local solvers on node B. Node A receives the clauses from node B, adds the clauses to its internal buffer, and feeds the clauses to the internal SAT solver.

To assess the quality of clauses received from some neighbor, we count (a) duplicate clauses and (b) clauses which are quickly deleted by the local solver. For further distribution, we store all received clauses in clause buffers, which sort the clauses by Literal Block Distance (LBD) and time of arrival in decreasing order. The buffers discard the trailing clauses if there are too many clauses stored. To summarize, our architecture consists of the following components:

- Filter for duplicate clauses.
- Filter for deleted clauses.
- Clause buffers.
- Duplicate clauses counter.
- Deleted clauses counter.

The counters store the number and point of time of received duplicate and deleted clauses for each neighbor in a separate manner. Our neighbor evaluation algorithm uses this information to determine the most promising neighbor to ask, i.e., the neighbor with the lowest number of duplicate or deleted clauses sent. We present different strategies for neighbor selection in section 3.4.

3.2.1 Clause Filter

As firstly introduced by Balyo et al. [3], Mallob filters clauses using a bloom filter. We reuse the clause filter from Mallob to reject received and deleted clauses before importing them. A bloom filter consists of a fixed size bit set of size m and k hash functions. Each hash function h applied to a clause returns a bit position in the bit set $h : C \rightarrow [0, m)$. To add a clause to the set, feed the clause into the k hash functions and set the bits at the resulting positions to true. To check if a clause is in the set, feed the clause into the k hash functions. The clause is assumed to be contained in the set if all bits at the resulting positions are true. The bloom filter allows registration of clauses with constant time complexity and memory consumption. This comes at the cost of possible false positives. The SAT solving algorithm stays sound, since we insert and filter redundant clauses only. The filter for duplicate clauses ensures that the internal buffer does not contain any duplicate clauses. The filter for deleted clauses ensures that the internal buffer does not contain any clauses that were deleted in the internal solver.

3.2.2 Requesting Procedure

A request message initiates an exchange of learned clauses between the solvers. Algorithm 1 outlines this procedure. To get up-to-date information about its neighbors, node A starts to search for deleted clauses in the internal buffer (lines $1 - 2$), removes the deleted clauses from its internal buffer (line 3), and updates the respective counters (line 4). Following this, node A selects one neighbor B based on the available information (line 5) and sends a request message to the chosen neighbor B (line 6). We explain the decision process in greater detail in section 3.4.

Algorithm 1: Requesting Clauses

```
1 for c ∈ buffer do
2     if IsDeleted(c) then
3         buffer.remove(c)
4         IncreaseDeleteCounter(c.source)
5 neighbor ← SelectNeighbor()
6 SendMessage(neighbor, msg_request)
```

3.2.3 Sending Procedure

Node B receives a request message from the requesting node A. Following this, node B collects his top k clauses and sends them to node A. Algorithm 2 outlines this procedure. To ensure that node B only sends meaningful clauses, node B removes all clauses that got deleted (line 3), updates the respective counters (line 4) and skips clauses that have already been sent to node A (line 6). Finally, node B sends the collected clauses to node A.

Algorithm 2: Collecting Top K Clauses

```
Data: i: neighbor ID, k: max number of clauses, n: replicate factor
Result: s
for $c \in$ buffer do
    if IsDeleted(c) then
        buffer.remove(c)
        IncreaseDeleteCounter(c.source)
        continue
    if $c.source = i$ or $c.send.test(i)$ then
        continue
    c.send.set(i)
    s.add(c)
    if $c.send.count() = n$ then
        buffer.remove(c)
    if $s.size() = k$ then
        break
return S
```

3.2.4 Receiving Procedure

Node A receives clauses from node B. Algorithm 3 outlines this procedure. Node A filters out deleted and known clauses (line 2). Then node A inserts all clauses passing its filters into the internal buffer (line 6) and into the internal SAT solver (line 7).

Algorithm 3: Receiving Clauses

```
  Data: r: received clauses, i: source neighbor
1 for $c \in r$ do
2     if IsDeleted(c) or $c \in$ buffer then
3         r.remove(c)
4         continue
5     c.send.set(i)
6     buffer.add(c)
7 learn(r)
```

3.2.5 Edge Cases

In the malleable setting nodes may be removed from the computation. There are three edge cases to consider in our sharing scheme:

- The requested node B is removed from the computation after sending the request message.
- The buffer of the requested node B is empty.
- The requesting node A is removed from the computation after sending the request message.

If the requested node B is removed or the buffer is empty, the requesting node A gets no response message. In this case, A does not block, but will just proceed to send another request to a new selected neighbor in the next round. Since we only update counters when we receive clauses, these two cases do not have any impact on the rating of the requested neighbor B. If the requesting node A is removed after sending the request message, the requested node B proceeds with the sending algorithm. After the collection, the requested node B tries to send the clauses to the requesting node A. The message is discarded because the requesting node A does not exist any longer.

3.3 Neighbor Rating

We use three signals to gain information about how useful a neighbor is. The first signal is the Literal Block Distance (LBD) metric of each clause. We consider a clause with a small LBD as more useful [2]. Therefore, neighbors who send clauses with smaller LBD are rated higher.

The second signal is whether an imported clause was deleted in the meantime. Modern SAT solvers commonly delete redundant clauses if they are not useful. If received clauses are deleted very quickly, we rate the source neighbor down. In addition to the three-tier system described in subsection 2.2.3, we may delete received clauses directly during import in CaDiCaL. A clause is deleted immediately during import if the clause contains (a) a literal that is marked as witness, (b) a literal that was eliminated or substituted, or (c) a fixed literal.

The last signal is the amount of already known received clauses. The clause can be already known for two reasons, either because another neighbor has sent the clause before or because the internal solver already learned the clause.

3.4 Neighbor Selection

In this section, we describe the problem of selecting a neighbor to request clauses from. To reduce communication volume, only one neighbor can be selected in each epoch. Therefore, we want to select the neighbor where we get the highest expected reward. On the other hand, we must explore to gain information on the reward to expect from each neighbor. This dilemma of exploration versus exploitation can be formalized as a multi-armed bandit (MAB) problem. At first, we describe the theoretical background of this problem. Then we motivate our choice of distribution for the expected reward. After that, we show three strategies for selecting one neighbor based on the available information on the expected reward.

3.4.1 Multi Armed Bandit Problem

The MAB problem, originally formulated by Robbins [25], is about the dilemma of exploration versus exploitation. The setting is that we have k arms from which we can choose one in each round. We want to select the arm that gives us the greatest reward. Since we do not know beforehand how much reward we get from each arm, we must explore to gain information.

The expected reward of each arm can be modeled as distributions $R = \{R_1, ..., R_K\}$. The parameters of each distribution can be learned from observed outcomes of chosen actions.

The general setting for a MAB problem consists of three components. A set of actions X with size $|X| = k$, an observable outcome y_t at time t for executing action x_t, and a reward function $r : Y \to [0, 1]$ for mapping an observed outcome to a reward $r_t = r(y_t)$. The goal is to find a sequence of actions with minimal regret. Let μ^* be the maximal reward mean $\mu^* = \max_k \mathbb{E}[R_k]$, the regret is:

$$\rho = T\mu^* - \sum_{t=1}^{T} r_t$$

In the next subsections, we explain strategies for choosing an action based on the available information.

3.4.2 Expected Reward

In the following, we assume that the selected neighbor exists and that we get a set of clauses back. We execute an action, observe an outcome and map the outcome to a reward. In our setting the action is sending a request message to a neighbor i. The outcome is a set of received clauses C from neighbor i. We use the signals that we describe in section 3.3 to map the outcome to a reward. Our objective is to get as many useful clauses as possible. Therefore, our reward is $r = 1$ for a useful clause and $r = 0$ for a useless clause. The expected reward can be approximated by a Bernoulli distribution, where we interpret a useful clause as success and a useless clause as failure. The expected reward for neighbor i

is the parameter p_i of the Bernoulli distribution, i.e., $\mathbb{E}[R_i] = p_i$. The maximum likelihood estimator for the parameter p_i of a Bernoulli distribution R_i is the sampled mean [24, Chapter 5.1.2]:

$$p_i = \frac{1}{\sum_{t=1}^{T} \delta_{x_t i}} \sum_{t=1}^{T} r_t \delta_{x_t i}$$

In our case we do not get only one clause per action, but $\leq k$ many clauses. Due to the requirement of a reward $r \in [0, 1]$, we choose the following reward function. Let A_i be the set of accepted clauses and D_i the set of immediately deleted clauses in observation y_i. Our reward for action x_i is:

$$r_i = r(y_i) = \frac{|A_i| - |D_i|}{\max\limits_{0 \leq j \leq i} (|A_j| - |D_j|)} \tag{3.4.1}$$

Additionally, we adapt this rating function to incorporate the Literal Block Distance (LBD) metric. Let $m_{i,j}$ be the LBD of clause $c_{i,j}$ in observation y_i.

$$f(y_i) = \sum_{c_{i,j} \in A_i} \frac{2}{m_{i,j}} - \sum_{c_{i,j} \in D_i} \frac{2}{m_{i,j}} \tag{3.4.2}$$

$$r_i = \hat{r}(y_i) = \frac{f(y_i)}{\max\limits_{0 \leq j \leq i} f(y_j)} \tag{3.4.3}$$

3.4.3 Uniform

In the simplest strategy, we choose the neighbor at random with uniform probability in each epoch. One advantage of not using any past information is that a drift in the reward distribution has no consequences. This leads to optimal exploration. In addition, in the malleable setting newly added nodes eventually receive request messages. The drawback is that gained information about neighbors do not get exploited. This leads to inefficient clause exchanges where a high proportion of received clauses is immediately discarded. Furthermore, the sum of regrets is potentially high, since we keep asking useless neighbors. Therefore, we waste epochs where we could have requested clauses from useful neighbors.

3.4.4 Greedy

The greedy strategy uses information about the expected reward described in subsection 3.4.2. At each time the greedy strategy chooses the action with the current maximum expected reward $\arg\max\limits_{i} p_i$. Without any adaptations, this is problematic, as the following example illustrates: Assume the real expected reward does not change and the real expected reward for neighbor one is $\hat{p}_i = 0.8$ and the observed expected reward for neighbor one is

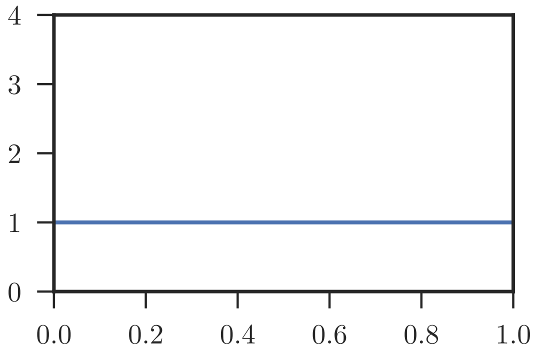


Figure 3.3: Beta distribution with parameters $\alpha = 1$ and $\beta = 1$.

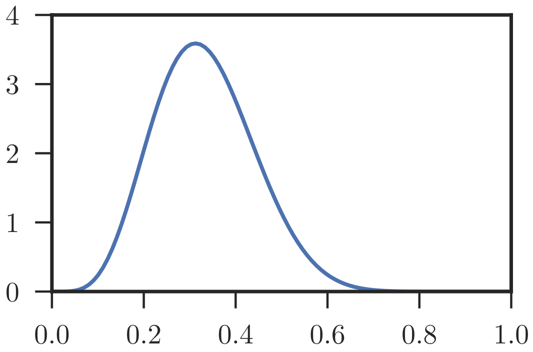


Figure 3.4: Beta distribution with parameters $\alpha = 6$ and $\beta = 12$.

$p_i = 0.4$. If the real and observed expected reward of neighbor two is $p = 0.5$, the greedy strategy will never query neighbor one again and does not get the higher reward. One solution for this behavior is to introduce an exploration round and an exploration parameter $\epsilon \in [0, 1]$. The parameter ϵ indicates the probability that the uniform strategy described in subsection 3.4.3 chooses the neighbor instead of the greedy strategy.

3.4.5 Thompson Sampling

The problem of large differences between the observed and the real expected reward can be tackled by introducing uncertainty. The goal is to have a high uncertainty for a small sample size. We use Thompson sampling [30] to tackle this problem. Thompson sampling draws rewards from a beta distribution and chooses the action with the maximum drawn reward. This is in contrast to the greedy strategy, where the action with maximum expected reward is chosen. The beta distribution is defined on the interval $[0, 1]$ and is parameterized by two positive shape parameters α and β. The expected value and variance of $X \sim \text{Beta}(\alpha, \beta)$ is:

$$\mathbb{E}[X] = \frac{\alpha}{\alpha + \beta}$$

$$\mathbb{V}[X] = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$$

The variance can be interpreted as the uncertainty. Since the denominator grows faster than the numerator it follows that $\mathbb{V}[X] \to 0$ as $\alpha, \beta \to \infty$. We use the same update rule as used by Russo et al. [26]:

$$(\alpha_i, \beta_i) = \begin{cases} (\alpha_i, \beta_i) & x_t \neq i \\ (\alpha_i + r_t, \beta_i + 1 - r_t) & x_t = i \end{cases}$$

As the parameters are monotonically increasing, our uncertainty is monotonically decreasing. For $\alpha = \beta = 1$ the distribution is uniform and therefore represents maximum uncertainty as illustrated in Figure 3.3. As $\alpha + \beta$ increase, the distribution becomes more concentrated as illustrated in Figure 3.4. Thus, at the beginning, the strategy focuses on exploration and shifts to exploitation with an increasing number of samples.

In our setting the real reward is likely to be non-stationary [4]. A useful neighbor can become a useless neighbor. The search space of both can diverge and the search direction from one node no longer complement the other. On the other hand, a useless neighbor can become a useful neighbor. In order to adapt to a drift in the real reward, we discount the reward over time. A discount factor is commonly used for this [23, 6]:

$$(\alpha_i, \beta_i) = \begin{cases} (d\alpha_i, d\beta_i) & x_t \neq i \\ (d\alpha_i + r_t, d\beta_i + 1 - r_t) & x_t = i \end{cases}$$

This leads to decreasing importance of observed rewards in the past.

4 Experimental Evaluation

We first describe implementation details of our gossiping clause exchange framework and present our hardware and software setup. Next, we explain how we tune the parameters we introduced in chapter 3. Following, we present our selection of Boolean Satisfiability (SAT) instances for testing and comparing our implementation. Finally, we show our results and make a profound analysis of our hypothesis about expert clusters.

4.1 Implementation

We integrated our algorithm into Mallob [28]. Mallob and our gossiping framework are implemented in C++17. The interprocess communication is done via an open Message Passing Interface (MPI) implementation [11]. MPI is a message-passing standard commonly used in high performance computing for communication between nodes. We use the beta distribution implementation of Mansfield [19] for the Thompson sampling strategy. Other components, such as clause filters, are reused from Mallob. The hash function introduced in [3] is used in the clause filter. The internal bitset of this clause filter resides in memory shared between the MPI process and the SAT solving process. This allows for immediate checking of deleted clauses without further interaction between communicator and SAT solver. In addition, this ensures the least possible overhead when registering deleted clauses and passing this information on to the communicator for the SAT solver. In addition to internal support for a hexagonal grid communication graph, our implementation supports parsing of custom communication graphs in Pajek NET Format.

4.2 Experimental Setup

Due to availability, we conducted our experiments on two computers. On computer A, equipped with one AMD EPYC™ 7702 running at 2.0GHz and 1TB DDR4 RAM and on computer B, equipped with two AMD EPYC™ 7713 running at 2.0GHz and 1TB DDR4 RAM. The operation system is Ubuntu 20.04 LTS using version 5.4.0-generic of the Linux kernel. The code was compiled using GCC version 9.3.0 with OpenMPI version 4.0.3.

4.2.1 Tuning Parameters

Our framework consists of many components, which results in a large configuration space. For this reason, we are committing ourselves to a hexagonal grid communication graph and two neighbor selection strategies, namely an uninformed uniform selection strategy and an informed Thompson sampling strategy. Furthermore, we use the CaDiCaL SAT solver for all our experiments. Our baseline is a pure portfolio approach, i.e., 64 CaDiCaL SAT solvers without any communication and clause exchange. We use as many nodes as physical cores are available. We noticed that the amount of shared clauses is relatively small in our setup. This is probably due to the requirements for a clause to be sent, as described in subsection 3.2.3 and the limited buffer size of 4000 clauses per neighbor. Therefore, we conduct all experiments without a limit on the number of clauses to be sent. The informed strategy uses the rating function described in Equation 3.4.1. The usage of the reward function described in Equation 3.4.3 did not lead to a significant improvement in our setting. We compare our approaches to the default settings of Mallob.

4.2.2 Instances

We choose 80 instances [28] of the SAT2020 competition [10] for testing and comparing.

4.3 Results

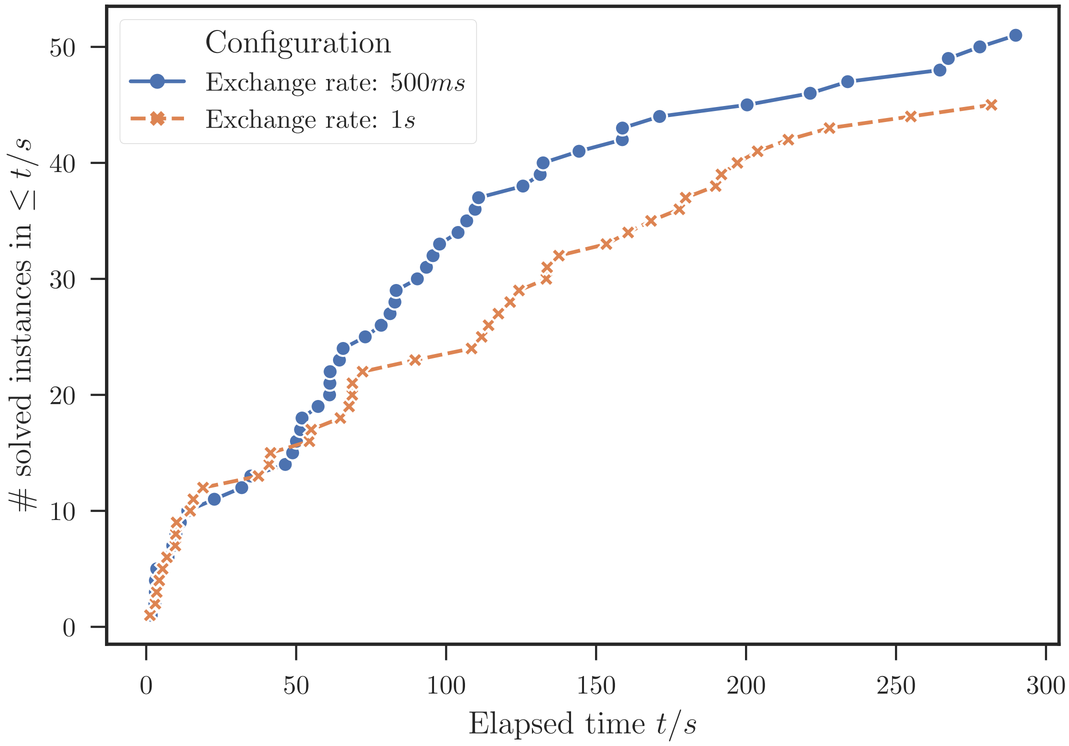
Before evaluating and comparing our approach with the default settings of Mallob, we need to find an appropriate reward function and parameters. We first evaluate the uninformed neighbor selection approach described in subsection 3.4.3. Using this selection strategy, we observed a significant speedup by adjusting the exchange rate. Continuing, we discuss some reward functions and justify our choice of the reward function described in Equation 3.4.1. We then compare our best run with informed strategy, our best run with uninformed strategy, the baseline without sharing, and the synchronous clause exchange of Mallob.

After that, we perform an in-depth analysis of the formation of expert clusters in our hexagonal grid communication graph. We compare the informed neighbor selection strategy with Thompson sampling with the uninformed one. We show indicators for the formation of expert clusters when using the informed strategy.

Finally, we test the support of our gossiping framework for malleability of jobs.

4.3.1 Comparison

One parameter from which the gossiping communication strongly benefits is the exchange rate. The exchange rate indicates the time interval at which clauses are exchanged. For

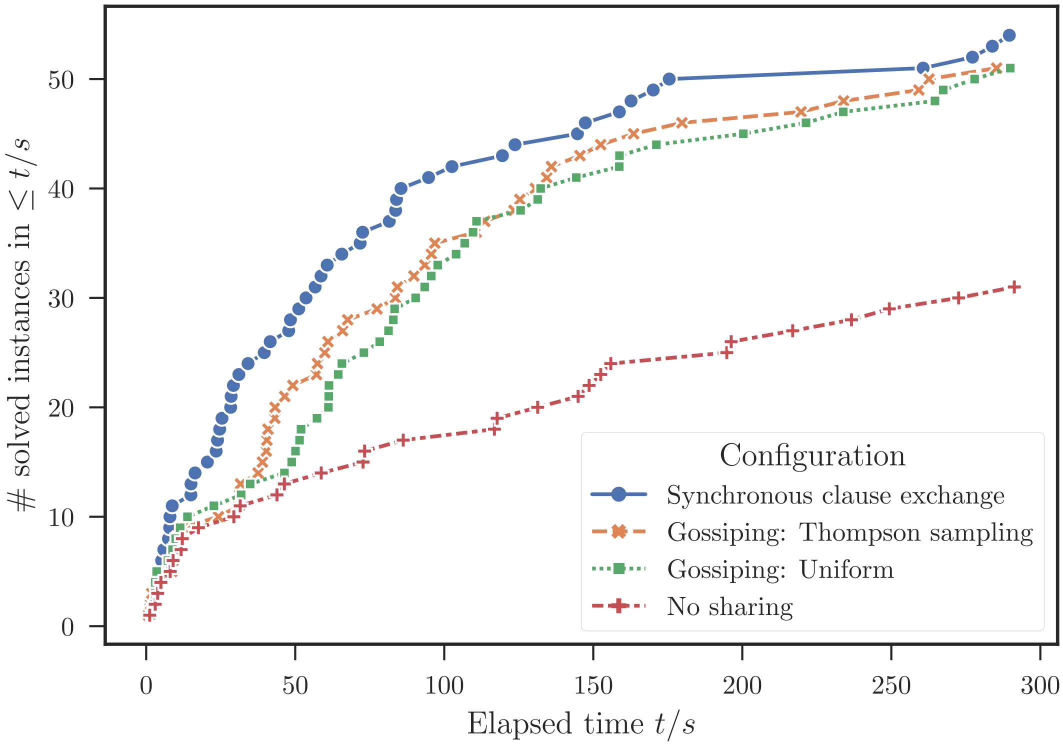


Configuration	#Solved	PAR-2 Score
Exchange rate 500ms	51	22160.07
Exchange rate 1s	45	25468.16

Figure 4.1: Comparsion of two exchange rate parameters using uniform neighbor selection strategy. The experiment was conducted on computer A.

example, at an exchange rate of 1 second, every second clauses are requested from a single neighbor. This has a major impact on the time required to distribute clauses, especially in the case of massive parallelization, as the communications graph and thus the maximum distance between 2 nodes increases. Furthermore, the time a node needs to distinguish useful from useless neighbors,i.e., the time of the exploration phase, also depends on the exchange rate. We show the results of two choices for the exchange rate in Figure 4.1 using the uninformed neighbor selection strategy. Doubling the exchange rate from once per second to twice per second results in significant improvement. We observe similar results when using Thompson sampling for neighbor selection. A further increase of the exchange rate did not lead to significant improvements.

We use the reward function described in Equation 3.4.1. The incorporating of the Literal Block Distance (LBD) metric in Equation 3.4.3 and the greedy neighbor selection strategy do not lead to a significant improvement. Figure 4.2 shows our best runs against syn-



Configuration	#Solved	PAR-2 Score
Synchronous clause exchange	54	19488.17
Gossiping: Thompson sampling	51	21721.83
Gossiping: Uniform	51	22160.07
No sharing	31	32539.28

Figure 4.2: SotA Mallob, baseline without communication and best runs of two different neighbor selection strategies. The experiment was conducted on computer A.

chronous clause exchange and our pure portfolio baseline. It also shows that the informed neighbor selection strategy outperforms the uninformed strategy.

4.3.2 Expert Clusters

Received clauses are rejected if the clause is registered in the deleted clause filter or in the duplicate clause filter. As described in section 3.2 our clause filter has a potential increasing rate of false positives as the number of registered clauses increases. However, as Figure 4.3 illustrates, the number of imported clauses per node does not decrease significantly over time. This indicates that a periodic clearing of the clause filter is unnecessary.

To verify our hypothesis of formations of expert clusters, we start by analyzing whether

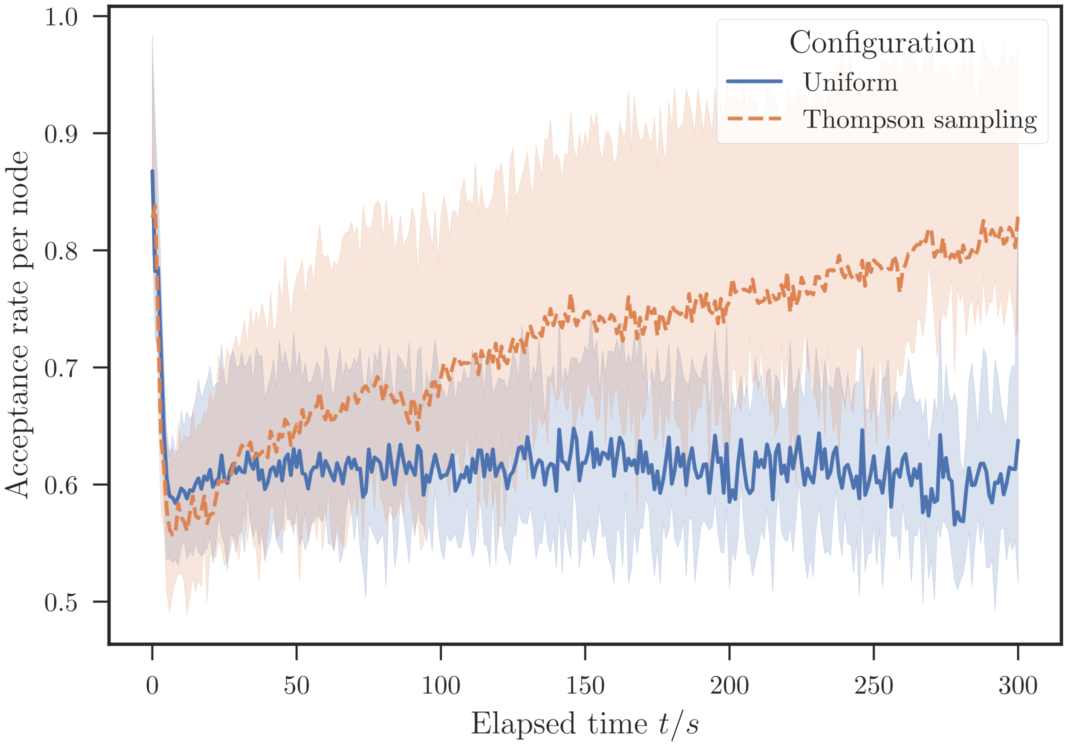


Figure 4.3: Acceptance rate, i.e., $\frac{\text{\# accepted clauses}}{\text{\# rejected clauses}}$.

nodes tend to prefer requesting clauses from some neighbors over the others. Due to the high dimensionality of our data, that is 64 nodes working on 80 Boolean Satisfiability (SAT) instances and selecting one from up to 6 neighbors in each epoch, we need a metric to reduce the dimensionality down to two dimensions. For this we reformulate our original question to the question of how ordered our system is or how predictable the requests are. The order of a system can be measured using the Shannon entropy [29]. To be able to assess how much the nodes explore, we count the occurrences of each neighbor ID within a certain time interval, i.e., the frequency of neighbor IDs. We interpret the counted occurrences as probability $p(x_i)$ that the respective neighbor x_i will be requested. The entropy of this probability distribution is:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i)$$

Intuitively, uniform probability yields maximum entropy, thus maximum exploration. Contrary, if only one neighbor will be requested, the entropy is zero, thus no exploration. We refer to this entropy as exploration rate.

Since the border nodes do not have as many neighbors as the inner nodes, i.e., six neighbors, we only consider the inner nodes. For easier interpretation, we divide the calculated

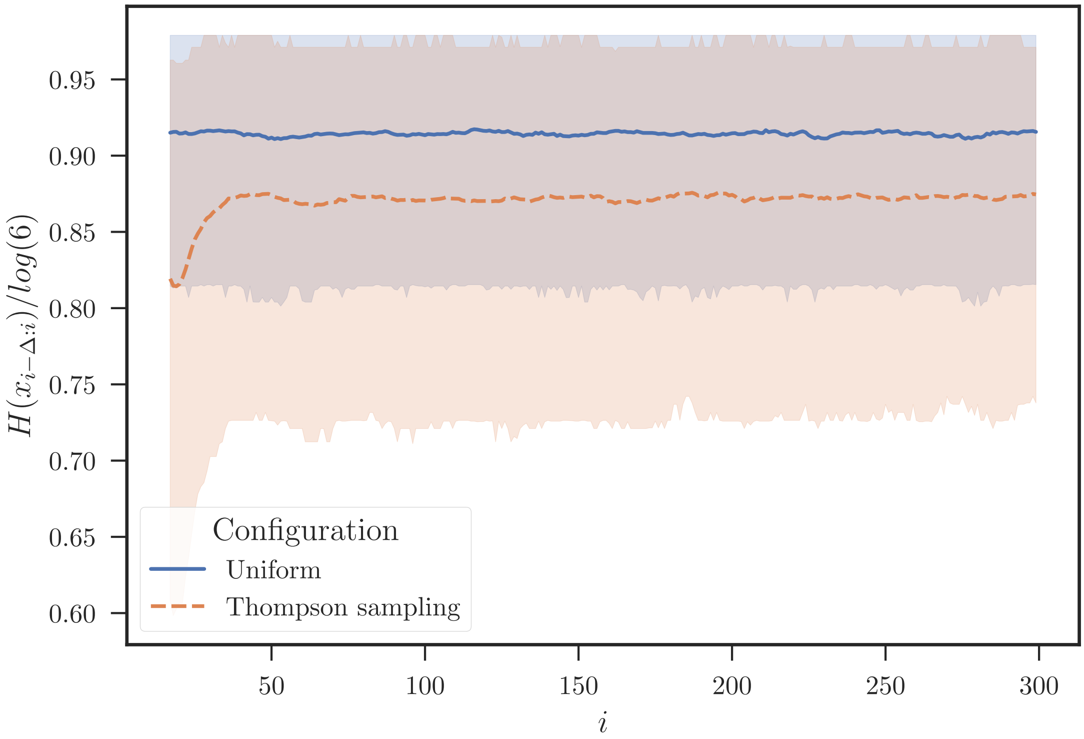


Figure 4.4: Exploration rate aggregated over all instances and nodes. i is the number of request messages send so far and $\Delta = 18$.

entropy by the maximum possible entropy. In our case the maximum entropy is the uniform probability distribution over six neighbors $X \sim \mathcal{U}\{1, 6\}$:

$$H(X) = -\sum_{i=1}^{6} p(x_i) \log p(x_i) = -\sum_{i=1}^{6} \frac{1}{6} \log \frac{1}{6} = 6\frac{1}{6} \log 6 = \log 6$$

Figure 4.4 illustrates the exploration rate of an uniformed and an informed neighbor selection strategy. The exploration rate of the uniform strategy represents maximal exploration. The median and minimum exploration rate of Thompson sampling is lower than the exploration rate of the uniform strategy. However the maximum exploration of both strategies is maximal. This indicates that there are nodes which tend to prefer some neighbors, but also nodes that do not prefer any neighbor. There are several explanations, such as (a) insufficient diversification of SAT solvers, (b) multiple neighbors are equally useful or useless, or (c) many instances do not benefit from gossiping communication.

Figure 4.5 shows the exploration rate, while solving one specific instance. This instance was solved much faster by the Thompson sampling strategy than by the uninformed strategy, with an speedup of 2.8, indicating that this instance strongly benefits from informed gossiping communication. Furthermore, the graph shows an oscillating exploration factor,

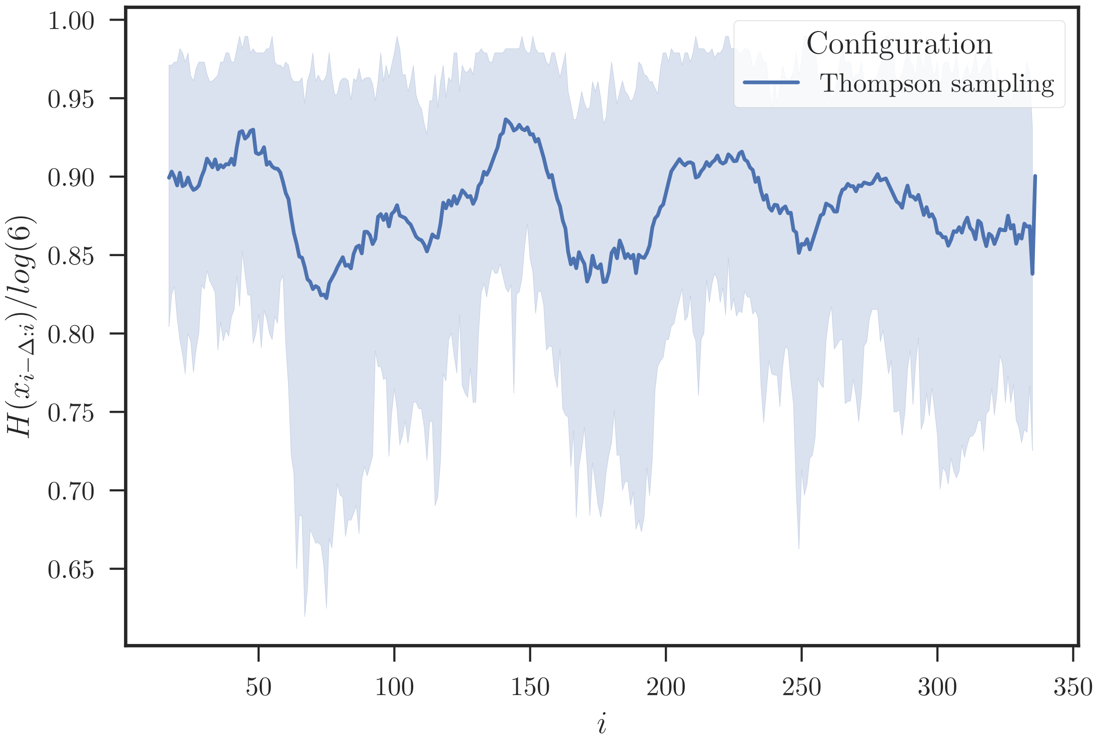


Figure 4.5: Exploration rate aggregated over all nodes, solving the `gto p50c314` instance. i is the number of request messages send so far and $\Delta = 18$.

indicating that useful neighbors can become useless, which supports the assumption of a non-stationary reward.

The second indicator for expert clusters is the increasing acceptance rate per node as illustrated in Figure 4.3. We calculate the acceptance rate by dividing the number of accepted clauses, i.e., the clauses that are passed to the solver, by the number of rejected clauses, i.e., the clauses that are assumed to be in the duplicated or deleted clause filter. Although the rating function used does not include the number of rejected clauses, the acceptance rate improves.

Finally, in Figure 4.6 and Figure 4.7 we visualize the popularity of neighbors relative to other neighbors and the popularity of nodes relative to all other nodes for one specific instance.

In Figure 4.6 the color of the edge arrows represents how frequent the source node requested clauses from the destination node. For example, node 4 requested node 0 relatively often, whereas node 3 almost never requested node 0. In Figure 4.7 the color of the nodes represents how frequent the respective node was requested, normalized over all nodes.

Figure 4.6 shows several clusters. For example, Node 2 and 9 seem to benefit strongly from each other, whereas there is relative few communication between node 2 and node 0.

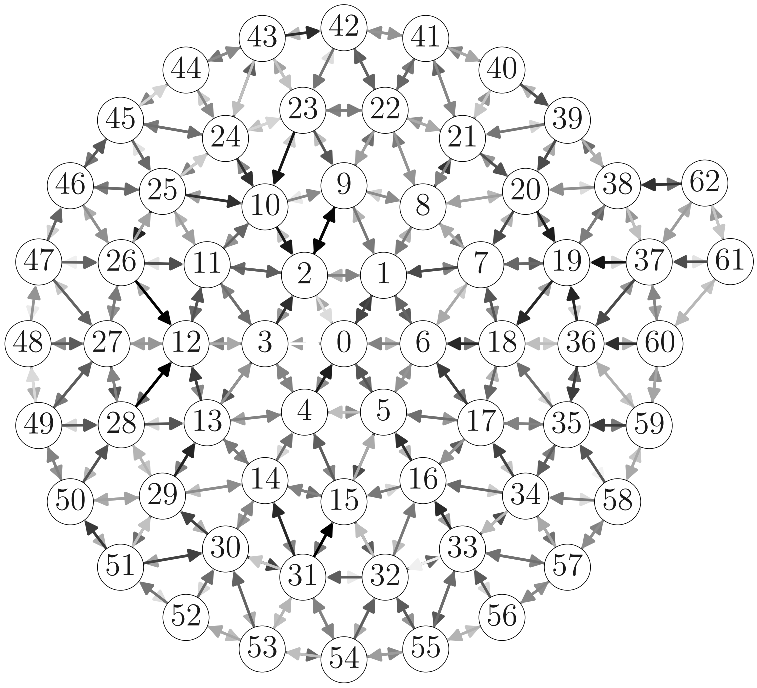


Figure 4.6: Probability of neighbors getting requested. The darker the color of the edge, the higher the probability.

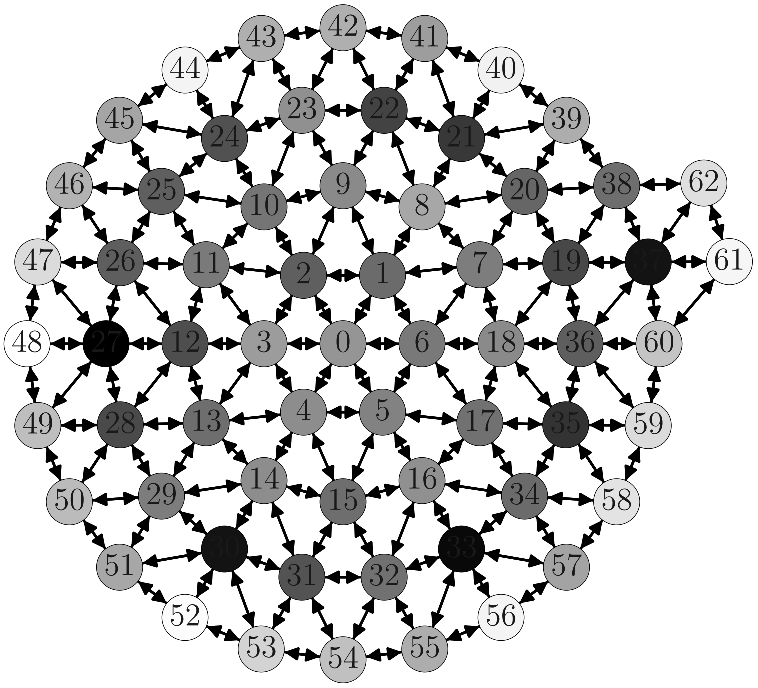


Figure 4.7: Node popularity, i,e, how frequent a node gets requested from its neighbors. The darker the color of the node the higher the popularity.

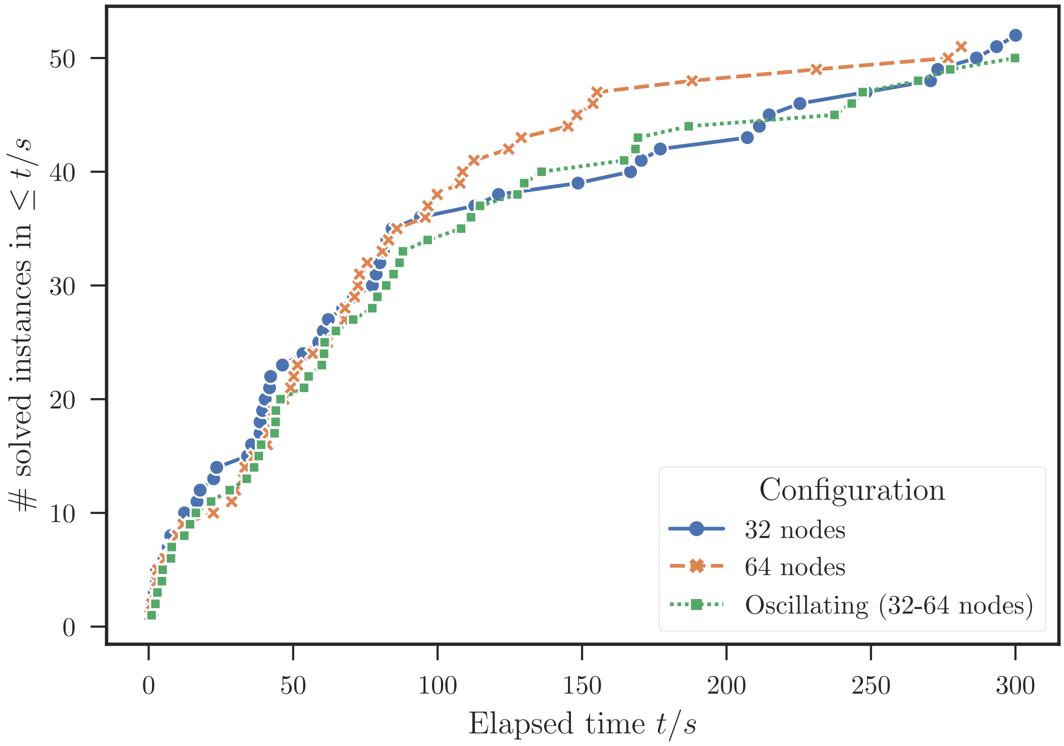


Figure 4.8: Test of support for malleability of jobs. This experiment was conducted on computer B

However, this could also be an indicator that node 0 produces relatively few useful clauses. This is contradicted by the high communication volume between node 0, 1, and 4, which indicates another cluster.

Moreover, both Figures reveal a strong emergence of queries from the border nodes inward to the core of the graph. This makes sense, since the clause production is much higher in the core than on the border of the graph, due to a larger number of workers and stronger connectivity.

4.3.3 Malleable Setup

In this subsection we analyze the performance of our gossiping communication framework in the malleable setting. For this, we run an experiment with an oscillating number of available nodes. To accomplish this, we introduce one demanding job every 30 seconds with a timeout of 15 seconds, in addition to the main job. In our setup, the number of available nodes oscillates between 32 and 64 nodes. Furthermore, we run an experiment with 32 nodes and an experiment with 64 nodes. All three experiments use the uniform selection strategy. The oscillating experiment should perform worse than the experiment with 64 nodes and should perform better than the experiment with 32 nodes. However, as seen in Figure 4.8, the oscillating experiment performed as good as the experiment with 32 nodes, thus indicating poor support for malleability of jobs. One explanation could be that the former border nodes start requesting the new nodes as soon as they get available. However, since the new nodes have not been around long enough to have produced useful clauses, the previous edge nodes do not benefit from a request. A solution could be to penalize new neighbors and thus delay the first request.

5 Discussion

5.1 Conclusion

In this thesis we motivated parallel SAT solving and presented parallel SAT solvers we have built upon. We motivated our approach of gossiping communication and the choice of our communication graph and derived formulas for calculation of neighbor IDs given a node ID.

Following, we explained which signals we use to distinguish between useful and useless clauses. We showed a formal definition of our sub problem of neighbor selection and presented different approaches to solve this problem. The problem of neighbor selection was addressed with an uninformed and informed strategy. The uninformed strategy consists of selecting a neighbor with uniform probability. The informed strategy is called Thompson sampling and uses the signals about usefulness of clauses to identify useful neighbors.

We compared our approach with a pure portfolio approach and the default synchronous clause exchange in Mallob. After that we analyzed our hypothesis of formation of expert clusters. Finally we showed results for solving SAT instances in the malleable setting, where we observed and discussed a decline in performance.

While we cannot match the performance of the synchronous clause exchange of Mallob, we observed interesting behavior in the neighbor selection. We showed that a informed neighbor selection strategy outperforms the uninformed one. This is an encouraging result for further research in this direction and indicates potential of dynamic clause exchange.

5.2 Future Work

Due to the large number of possible directions, such as the choice of communication graph, neighbor selection strategies, or signals for the usefulness of clauses, we had to limit ourselves to a small subset. However, it might be worthwhile to make broader investigations. The most promising directions from the author's perspective are listed below:

Communication Graph

Our observations showed that even if some nodes have committed themselves to specific neighbors, other nodes do not do so. This could be solved through better diversification of SAT solvers. Furthermore, in the selected communication graph, clauses require several epochs to be fully propagated throughout the graph. A communication graph in which nodes have both local and global distributed neighbors could lead to a faster distribution.

Neighbor Selection

There are many possible informed neighbor selection strategies. We have only presented two and have not done any intensive parameter tuning. Further research in this direction could be worthwhile. Additionally, it might be worthwhile to include more signals as an indicator of the usefulness of clauses, such as how often a clause was used by the respective SAT solver.

Asynchronous Clause Exchange

We observed an increase in performance with an increased exchange rate. This indicates that the gossiping communication benefits of early and fast clause distribution. Therefore, an asynchronous clause exchange, in which clauses are sent immediately, could lead to a further increase in performance.

Bibliography

[1] Biere A., Heule M., and Maaren H. van. *Handbook of Satisfiability : Second Edition.* Number v.336 in Frontiers in Artificial Intelligence and Applications Ser. IOS Press, 2021. ISBN 978-1-64368-160-3. URL `http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx%3fdirect%3dtrue%26db%3dnlebk%26AN%3d2934688%26site%3dehost-live`.

[2] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc. event-place: Pasadena, California, USA.

[3] Tomáš Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A Massively Parallel Portfolio SAT Solver. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, volume 9340, pages 156–172. Springer International Publishing, Cham, 2015. ISBN 978-3-319-24317-7 978-3-319-24318-4. doi: 10.1007/978-3-319-24318-4_12. URL `http://link.springer.com/10.1007/978-3-319-24318-4_12`. Series Title: Lecture Notes in Computer Science.

[4] Omar Besbes, Yonatan Gur, and Assaf Zeevi. Stochastic Multi-Armed-Bandit Problem with Non-stationary Rewards. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL `https://proceedings.neurips.cc/paper/2014/file/903ce9225fca3e988c2af215d4e544d3-Paper.pdf`.

[5] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[6] Emanuele Cavenaghi, Gabriele Sottocornola, Fabio Stella, and Markus Zanker. Non Stationary Multi-Armed Bandit: Empirical Evaluation of a New Concept Drift-Aware Algorithm. *Entropy*, 23(3):380, March 2021. ISSN 1099-4300. doi: 10.3390/e23030380. URL `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8004723/`.

[7] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0-89791-239-X. doi: 10.1145/41840.41841. URL `https://doi.org/10.1145/41840.41841`. event-place: Vancouver, British Columbia, Canada.

[8] Thorsten Ehlers and Dirk Nowotka. Tuning Parallel SAT Solvers. *EasyChair Preprints*, 2018.

[9] Thorsten Ehlers, Dirk Nowotka, and Philipp Sieweck. Communication in Massively-Parallel SAT Solving. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 709–716, Limassol, Cyprus, November 2014. IEEE. ISBN 978-1-4799-6572-4. doi: 10.1109/ICTAI.2014.111. URL `http://ieeexplore.ieee.org/document/6984547/`.

[10] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2020. *Artificial Intelligence*, 301:103572, 2021. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103572. URL `https://www.sciencedirect.com/science/article/pii/S0004370221001235`.

[11] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[12] Youssef Hamadi and Lakhdar Sais. ManySAT: a parallel SAT solver. *JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT)*, 6, 2009.

[13] Marijn Heule, Matti Jarvisalo, Martin Suda, Markus Iser, Tomáš Balyo, and Nils Froleyks. SAT Competition 2020, 2020. URL `https://satcompetition.github.io/2020/results.html`.

[14] Marijn Heule, Matti Jarvisalo, Martin Suda, Markus Iser, Tomáš Balyo, and Nils Froleyks. SAT Competition 2021, 2021. URL `https://satcompetition.github.io/2021/results.html`.

[15] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, pages 50–65, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-34188-5. doi: 10.1007/978-3-642-34188-5_8.

[16] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing Rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages

355–370, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31365-3.

[17] Marko Kleine Büning, Tomáš Balyo, and Carsten Sinz. Using DimSpec for Bounded and Unbounded Software Model Checking. In Yamine Ait-Ameur and Shengchao Qin, editors, *Formal Methods and Software Engineering*, pages 19–35, Cham, 2019. Springer International Publishing. ISBN 978-3-030-32409-4.

[18] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922. URL `https://dl.acm.org/doi/10.1145/1773912.1773922`.

[19] Joseph Mansfield. A beta distribution random number distribution for C++11., March 2013. URL `https://gist.github.com/sftrabbit/5068941`.

[20] Norbert Manthey. The MergeSat Solver. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 387–398, Cham, 2021. Springer International Publishing. ISBN 978-3-030-80223-3.

[21] Fabio Massacci and Laura Marraro. Logical Cryptanalysis as a SAT Problem. *Journal of Automated Reasoning*, 24(1):165–203, February 2000. ISSN 1573-0670. doi: 10.1023/A:1006326723002. URL `https://doi.org/10.1023/A:1006326723002`.

[22] Amit Patel. Red Blob Games: Hexagonal Grids, October 2021. URL `https://www.redblobgames.com/grids/hexagons/`.

[23] Vishnu Raj and Sheetal Kalyani. Taming Non-stationary Bandits: A Bayesian Approach. *ArXiv*, abs/1707.09727, 2017.

[24] Rossi Richard J. *Mathematical Statistics : An Introduction to Likelihood Based Inference.*, volume 1st edition. Wiley, 2018. ISBN 978-1-118-77104-4. URL `http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx%3fdirect%3dtrue%26db%3dnlebk%26AN%3d1828554%26site%3dehost-live`.

[25] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952. Publisher: American Mathematical Society.

[26] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A Tutorial on Thompson Sampling. *arXiv:1707.02038 [cs]*, July 2020. URL `http://arxiv.org/abs/1707.02038`. arXiv: 1707.02038.

[27] Dominik Schreiber. Lilotane : A Lifted SAT-based Approach to Hierarchical Planning. *Journal of artificial intelligence research*, 70:1117–1181, 2021. ISSN 1076-9757. doi: 10.1613/jair.1.12520. Publisher: AI Access Foundation.

[28] Dominik Schreiber and Peter Sanders. Scalable SAT Solving in the Cloud. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, volume 12831, pages 518–534. Springer International Publish-

ing, Cham, 2021. ISBN 978-3-030-80222-6 978-3-030-80223-3. doi: 10.1007/978-3-030-80223-3_35. URL https://link.springer.com/10.1007/978-3-030-80223-3_35. Series Title: Lecture Notes in Computer Science.

[29] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[30] William R. Thompson. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 25(3/4):285, December 1933. ISSN 00063444. doi: 10.2307/2332286. URL https://www.jstor.org/stable/2332286?origin=crossref.