UNIVERSITETET I AGDER

A novel learning automata game with local feedback for parallel optimization of hydropower production

JAHN THOMAS FIDJE
CHRISTIAN KRÅKEVIK HARALDSEID

SUPERVISOR
Prof. Ole-Christoffer Granmo
Assoc. Prof. Morten Goodwin, PhD
Bernt Viggo Matheussen, PhD

University of Agder, 2017
Faculty of Engineering and Science
Department of ICT

This page is left intentionally blank.

Abstract

Hydropower optimization for multi-reservoir systems is classified as a combinatorial optimization problem with large state-space that is particularly difficult to solve. There exist no golden standard when solving such problems, and many proposed algorithms are domain specific.

The literature describes several different techniques where linear programming approaches are extensively discussed, but tends to succumb to the *curse of dimensionality* problem when the state vector dimensions increase. This thesis introduces LA LCS, a novel learning automata algorithm that utilizes a parallel form of local feedback. This enables each individual automaton to receive direct feedback, resulting in faster convergence. In addition, the algorithm is implemented using a parallel architecture on a CUDA enabled GPU, along with exhaustive and random search.

LA LCS has been verified through several scenarios. Experiments show that the algorithm is able to quickly adapt and find optimal production strategies for problems of variable complexity. The algorithm is empirically verified and shown to hold great promise for solving optimization problems, including hydropower production strategies.

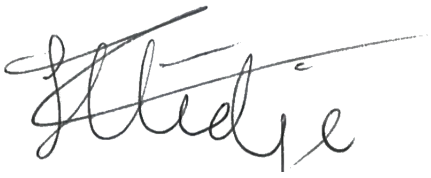
Preface

This thesis is made as a completion of the master education in communication and information technology (ICT), at the University of Agder, Norway.

Several individuals have contributed towards the completion of this master thesis, and the authors would like to thank in particular our dedicated supervisors Professor Ole-Christoffer Granmo, Associate Professor Morten Goodwin and Doctor Bernt Viggo Matheussen, who all have gone above and beyond what is expected.

Furthermore, we would like to thank our fellow masters student, Knut Eivind Sandsmark, for valuable ideas and discussion regarding the handling of invalid production-strings.

Grimstad, 21 Mai 2017.



Jahn Thomas Fidje

Christian Kråkevik Haraldseid

Contents

List of Figures

Listings

List of Tables

Abbreviations

CAIR Center for Artificial Intelligence Research

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DP Dynamic Programming

GA Genetic Algorithm

GG Gore Game

GPU Graphics Processing Unit

IWO Invasive Weeds Optimization

LA Learning Automata

LARW Learning Automata Random Walk

LCS Local Contribution Sampling

LP Linear Programming

MPPSO Multicore Parallel PSO

MST Minimum Spanning Tree

PSO Particle Swarm Optimization

RAM Random Access Memory

SAT Satisfiability Problem

SDDP Stochastic Dual Dynamic Programming

SDP Stochastic Dynamic Programming

TSP Traveling Salesman Problem

UIA University of Agder

URS Uniform Random Search

1 Introduction

Hydroelectric production are in most cases scheduled with respect to the current demand in the power network at any given time. Demand influence prices, which again increases the potential income. This creates a hierarchy of complex variables like available water-resources, production-capacity and systemflow. To maximize income, water has to be strategically stored. The stored water can then be optimally used when prices are high, maximizing profit for the producer.

The challenge with such a task is the number of possible solutions. Should the system produce at all times? Should it save water based on a likelihood of higher prices in the future? Or maybe alternate production? All these possibilities add up to thousands of different solutions that all need to be evaluated in order to find an optimal solution.

This problem falls under the domain of optimization algorithms that try to find the best element from a set of elements [26]. One branch from these optimization problems is combinatorial optimization, where the objective is to find an optimal object from a finite set of objects. A common approach to such problems is exhaustive search algorithms. However, when the search space gets too large, exhaustive search is not feasible.

Examples of combinatorial optimization problems may include navigation where we want to find the fastest route, or avoid traffic or toll-roads. Delivery of packages along a route, and other optimization problems where the feasible solutions are discrete, or can be reduced to discrete. Well defined problems withing combinatorial optimization are the traveling salesman problem (TSP) [15] and the minimum spanning tree problem (MST) [33].

Since combinatorial optimization problems search for the best element of some set of discrete items, any sort of search algorithm or meta heuristic can be used to solve them.

In this thesis an approach is taken using reinforcement learning techniques, where autonomous agents learn from experience. These agents can then learn heuristics and strengthen meta heuristic search approaches as described by Wauters et.al [35].

This thesis presents a hybrid solution where reinforced learning and meta-heuristic search are combined to solve complex combinatorial optimization

problems in parallel using learning automatas and CUDA.

The goal is to maximize the returned profit of a hydropower system simulation, by predicting the optimal production strategy over a given time-span. The upper part of *Mandalsvassdraget* [1] is used as a reference system. The hydropower system is owned by Agder Energi and is located in Mandal, Norway. Agder Energi is a research partner with CAIR (Centre for Artificial Intelligence Research) at the University of Agder.

[1] A river system consisting of several large rivers that collectively form Mandalselva. 11 power plants has been built in this system, totaling 1700GWh in yearly production.

1.1 Problem statement

Hydropower optimization is as previously mentioned a combinatorial optimization problem with a potentially large search space, and thus assumed to be limited by the same factors as other NP-complete optimization problem.

This means that it may not be possible to verify the chosen solution as the optimal, due to the drastic increase in difficulty when the problem is scaled up, resulting in an exponential increase in search space (Equation2).

This strengthens the assumption of the NP Complete properties, which means that an optimal solution may not be found in polynomial time as long as $P \neq NP$.

A good search algorithm should avoid local optima, produce profitable solutions in a short amount of time and scale well with varying problem sizes. [35].

The main contribution from this thesis is the exploration of using learning automatas to solve combinatorial optimization problems, with regards to hydropower optimization and parallel processing.

The main contribution from this thesis is the use of a novel learning automata approach to solve combinatorial optimization problems, with regards to hydropower production.

1.1.1 Research questions

I. How well suited is GPU-programming when applied to complex combinatorial optimization problems with regards to hydropower production using exhaustive search, random search and learning automata algorithms?

II. Would a learning automata algorithm be able to converge and find a global, optimal production strategy for a multi-reservoir hydropower production system?

1.2 Contributions

This thesis introduces *Local Contribution Sampling* (LCS), a replacement feedback method to the well-defined, widely used, global feedback approach used in learning automata algorithms. [21].

The local feedback in LCS enables direct feedback to each individual automaton. LCS has also been designed in a decentralized manner, making it capable of parallel execution. A parallel implementation of both GPU and CPU versions are presented.

The algorithm is applied on optimization problems related to hydropower production, and are to the best of our knowledge the first documented case where a learning automata algorithm has been applied within the studied field.

1.3 Report Outline

The report is structured in a manner where chapter 1 and 2 explains the problem, domain and current state of related research. Chapter 3 explains the proposed solution, algorithms and approach. Experiments and verification of results with a conclusion follows in chapter 4 and 5.

1.4 Theoretical Background

1.4.1 Modeling

A hydropower installation is regarded as a complex system, with a variety of variables spanning across several different aspects of the installation. In general a basic system can be decomposed into the following main parts, shown in Figure 1.

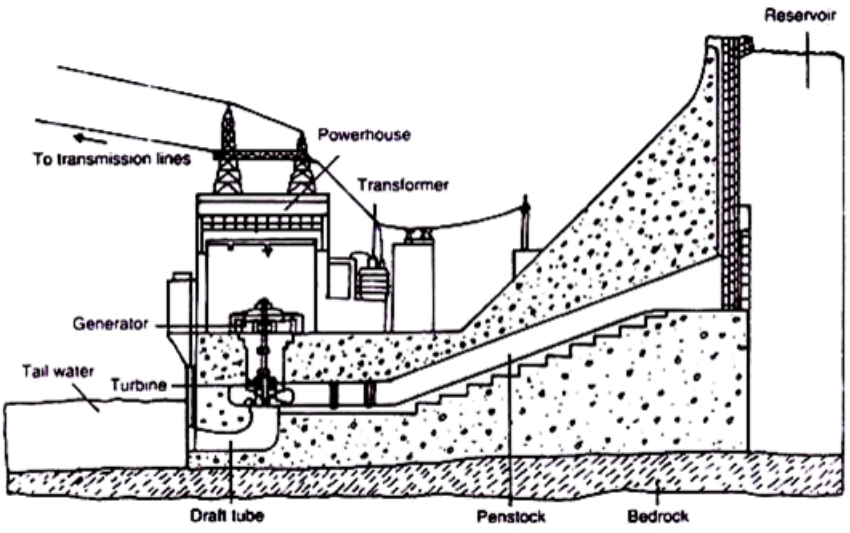


Figure 1: Diagram showing the vital parts of a impoundment[2]plant (biologydiscussion.com).

We can extract the following important features from Figure 1:

(1) Water is collected and stored in a designated reservoir.
(2) Water is fed trough the system passing a turbine to produce power.

[2]A type of powerplant where water is stored in a upstream reservoir.

To uncover further challenges with hydropower systems, we need to dive deeper into each component. The first stop is water collection. This can happen in several different ways, but some of the most common sources include rain, rivers, lakes and other reservoirs. The next aspect of complexity is the option for chained systems. Reservoir A may feed its tail-water downstream, directly into reservoir B. There may also exist dedicated reservoirs, called "non-powered dams", whose sole purpose is to collect water that can be released on demand. No production takes place during this process.

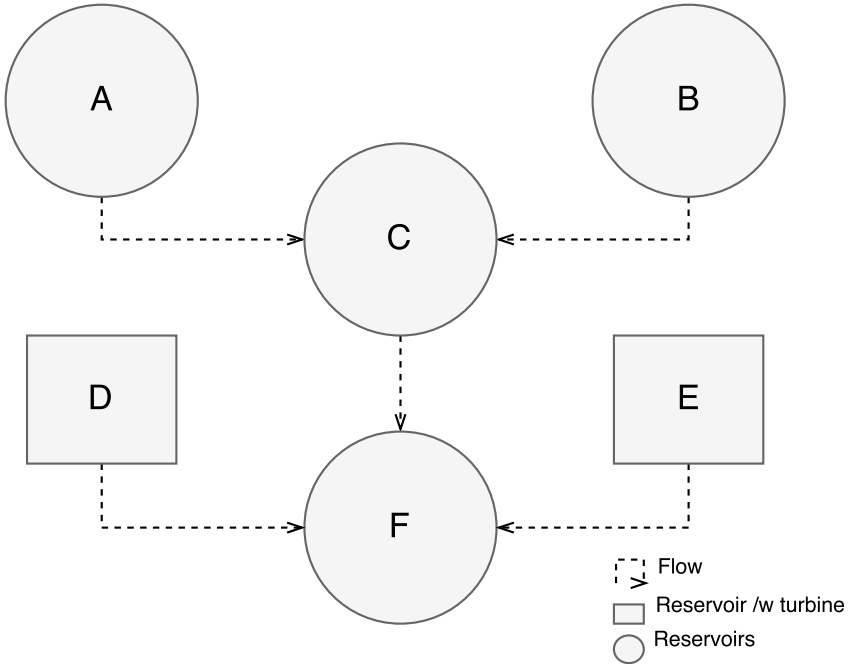


Figure 2: Diagram showing the possibilities for inter-connected systems.

Figure 2 tries to illustrate the variety of possible connections throughout a system. In the figure, we see reservoirs (system with reservoirs and turbine attached) A and B feeding water into reservoir C. The combined amount of water is then fed through C and into F, with additional water from dam (reservoir with no production) D and E. This shows the increase in complexity when systems are chained, which is often the case. Especially in larger commercial settings.

1.4.2 Cuda

A modern GPU is designed as a high compute density, fixed-function processor. The GPU was originally created to meet the needs of computer graphic workloads, but has in later years seen increasing general-purpose capabilities such as flexible control flow and random memory access. The original GPU design was focused around parallel architecture and has seen a steady performance growth. [32]

In the early 2000s, attempts were made to utilize GPUs as general-purpose devices for parallel workloads - an adaptation that was ignited by the introduction of Nvidia CUDA. CUDA exposes a clean, general-purpose interface for easy interaction with GPU hardware. The potential gains from GPU processing triggered further development, and today CUDA and other technologies are used extensively for parallel processing, providing processing capabilities several orders faster than the CPU for certain workloads.

Threading

Threading is a key point when working with a parallel architecture. The threads in a conventional sequential CPU are fairly different than the ones used in a GPU. A higher grade CPU may present threads in the 8-60 range, while the number of threads that are ready to process large parallel workloads on the GPU may be in the millions. In CUDA, the threads are organized into *blocks* and *grids*. Each block contains an amount of threads optimal for solving the problem at hand. Blocks and threads are then placed in a grid of blocks, which can exist in multiple directions (X,Y,Z). The key point of this ordering relates to the indexing of threads in terms of the problem, but also serves as a way to distribute a problem over several GPU-chips. To further elaborate on the concept of CUDA threading, a simple example is shown in Listing 1.

```
1 #include <stdio.h>
2
3 __global__ void print(int* arr_in) {
4
5
6     // Get local and global thread id
7     unsigned int idx = threadIdx.x;
8     unsigned int id = idx + (blockDim.x * blockIdx.x);
9
10    if (arr_in[id] % 2) {
11          printf("%d",arr_in[id])
12    }
13
14 }
```

Listing 1: Simple CUDA Kernel

The example kernel in Listing 1 illustrates the concepts of threads and blocks. The kernel prints a value from an array if the value is an odd number. Let's assume an array of size 6, containing the numbers 1-6. The array gets copied from host (CPU) memory into device (GPU) memory. Each spawned thread on the GPU now runs the exact same kernel code, but with their own unique ID. In this particular example we need 6 threads to process all 6 numbers in parallel. A typical high-end GPU[3] can handle 1024 threads within a single block, so a single block containing 6 threads, satisfies the requirements. When the GPU starts, each thread reads from the same array in parallel, using their unique id as index, printing the number if odd.

[3]Nvidia Tesla K80 used as reference http://www.nvidia.com/object/tesla-k80.html

To keep count of threads, a global thread ID is supplied through the CUDA API. The previous example is constrained to the X-dimension. It is possible to spawn blocks in a grid for all 3 dimensions (x,y,z), as long as it is contained within the bounds of the max-threads limit, for the specific hardware used. The *idx* variable contains the unique id for the current thread within this block, and the *id* variable contains the global id for this thread on the grid.

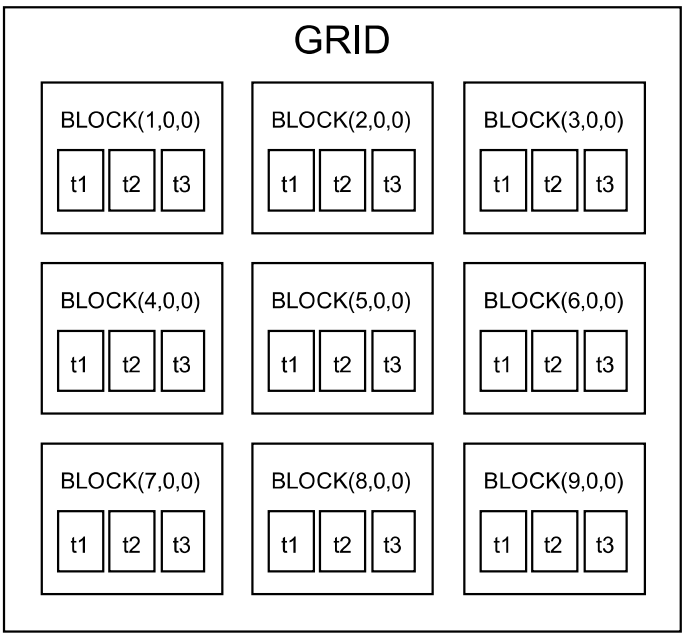


Figure 3: An illustration of a 1D thread arrangement

Memory

Memory is another aspect in the realm of CUDA processing. The initial copying of data between host and device is slow, and has a lot of overhead attached. It is therefore important to plan ahead and execute kernels with this in mind. The only way to maximize throughput is with proper memory management. The GPU itself has several different memory locations: Global, Texture, Constant, Shared and registers. Table 1 explains the different memory types used in this thesis.

Global Memory	Slow and big, global access for all threads.
Shared Memory	Fast and small, shared access for threads within block.
Registers	Fastest but smallest, local thread access only.

Table 1: Different memory types in the GPU[4].

[4]Excluded Texture Cache and Constant Memory.

1.4.3 Learning Automata

In larger, complex systems, decentralization is often seen and can even be traced back to nature [17]. Decentralized systems forces decision makers to take action based on a limited set of knowledge. This is due to the decentralized nature and its impact on information exchange between all agents. To deal with such problems, the "theory of teams" was developed. This theory addresses questions related to decision and rule design [31]. Later the mathematical foundation was laid to analyze decentralized problems trough game theory [6].

Game theory is particularly suited for political, social and economic problems due to the lack of knowledge in the decentralized system. Each agent takes an individual choice based on their preferences or knowledge. This makes game theory applicable to problems where conflict of interest may be an issue among agents.

A game can be seen as a decentralized problem consisting of different agents (decision makers). In such an example the uncertainty is usually linked to the unknown actions of other players. To solve such problems, learning schemes are used to seek asymptotic performance that can coincide with rational behavior built on the concepts of game theory [21].

To further elaborate on this concept in the context of the learning automata algorithm (LA), a game is used as an example. The game is played repeatedly in conjunction with a large amount of uncertainty. Each automata is blind to other players in the game, and also knows nothing about the strategy chosen by other players, including the response given to the players, except itself. The automata chooses a strategy and receives a probability response from the environment. This response is used in conjunction with a stochastic element to determine the final outcome.

A more applicable presentation of the above scenario is presented by ML. Tsetlin[21], mentioned in [23],[21] and [13], known as a *Goore Game* (GG). The game is presented below using the formulation of B.J Oommen et al. [23].

"Imagine a large room containing N cubicles and a raised platform. One person (voter) sits in each cubicle and a Referee stands on the platform. The Referee conducts a series of voting rounds as follows. On each round the voters vote "Yes" or "No" (the issue is unimportant) simultaneously and in-

dependently (they do not see each other) and the Referee counts the fraction, A, of "Yes" votes. The Referee has a uni-modal performance criterion G(A), which is optimized when the fraction of "Yes" votes is exactly A*. The current voting round ends with the Referee awarding a dollar with probability G(A) and assessing a dollar with probability 1 -G(A) to every voter independently. On the basis of their individual gains and losses, the voters then decide, again independently, how to cast their votes on the next round."

A few features of this game can be extracted and tied with game-theory. First, the Gore Game is a practical example of a *non-zero-sum* game. This means that the participants of the game's aggregated gains and losses can be less or more than zero. Since there is no knowledge of the other participants the game can be classified as a distributed game. Another important feature is that the environment function penalizing or rewarding the participants can be arbitrary as long as it is uni-modal[23].

When the LAs adapts to the referee's feedback, or from the responses from the environment, they will asymptotically optimize their responses towards the optimal solution.

In the GG presented above, each player represents a learning automata, while the referee is the environment. In other words, a group of automata interacting with an environment. The GG can be used as a case-study of LA performance. Learning is accomplished by interacting with the environment. The LA processes and replies based on this interaction, while gradually converging towards an optimal solution. It is shown that a group of automatons, when given enough memory, is able to asymptotically optimize their collective responses [21].

It is assumed LA A_n has an even number of states S_n. The initial number of states is given as a single value s, which is multiplied by the size of the action set A_n^0, A_n^1. When the environment responds to an action A_n^n, the state moves towards the extreme of either end. States $S_{s-(s-1)}$ to S_s yields action A_n^0, while S_{s+1} to S_{s+s} yields action A_n^1. A_n^0 is defined as a negative action, while A_n^1 is a positive action. A response r is returned from the environment. Here r represents the probability of reward. A reward-probability gives an indication of how the environment wants the states to change. A low-probability r would give a high chance for punish with an associated state-change towards the center of the state-space. I.e if the current selected action is in the range $S_{s-(s-1)}$ to S_s, a punish would move the state towards S_{s+1}.

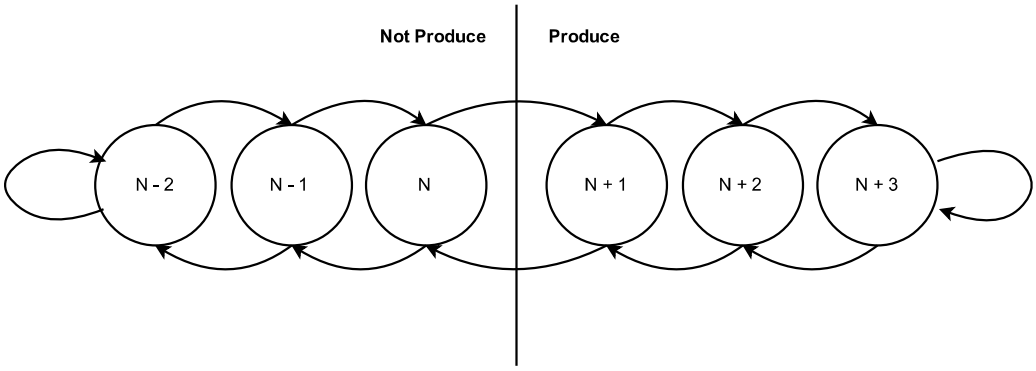


Figure 4: State illustration for a 2 action LA implementation.

On the environment side a uni-modal function is responsible for determining the output for an input set of actions. In the GG description the referee or environment handles the counting of votes from the participants. The GG utilizes a global feedback scheme where the response to each automaton from the environment is a direct result of the joint inputs. This means that each automaton has an impact on the feedback given to each of the participants.

The environment returns a probability for success/reward. This probability is then returned to each automaton and a stochastic process decides the final environment action.

2 State of Art

Hydropower optimization techniques are a broad field, and several different approaches exists. A broad range of classical optimization algorithms, like linear programming, are extensively discussed in the literature [1, 3, 2, 8]. One common form of solving problems related to hydropower production is the use of dynamic programming (DP). Within this field several optimization algorithms exists. One form of DP *Stochastic Dynamic Programming* (SDP) was introduced early by R.Belleman [5] in 1957, and has been extensively researched in the field of hydropower optimization [28, 29].

SDP is shown to perform well on single-reservoir systems but the added complexity of multi-reservoir systems tends to limit the success of solutions affected by the *curse of dimensionality*[5].

For a multi-reservoir system the number of solutions increase exponentially, and is directly tied to the number of timesteps, reservoirs and production choices available. Since the reservoirs are interconnected, production in one reservoir may affect the possibility of production in another. This creates complex relationships.[29]

In dynamic programming, the size of the state-space greatly increase the computational requirements when the dimensions of the state vector increase. This is especially true for multi-reservoir systems. Each reservoir adds a separate dimension to the problem, where different water levels makes up the states. The state-space is the product of the allowable water states for all dimensions.

With added dimensions, efficient search policies, like exhaustive search (for low state problems), is not feasible and approximation or aggregation of states into a single dimension is necessary to avoid falling into the curse of dimensionality. This greatly affects the accuracy of the algorithm and fails to properly account for the complex relationships between reservoirs.

Limited by the curse of dimensionality, additional algorithms were developed. *Stochastic Dual Dynamic Programming* (SDDP) was introduced in 1987 as a solution to the curse of dimensionality problem for hydropower optimization, making use of *Bender's decomposition* [7, 14, 27]. With SDDP the need to discretize the search space disappears. However, results show that SDDP struggles to handle extreme variations in inflow due to the limits in inflow model complexity [14]. It is argued that "SDDP need for inflow-

modeling yields a statistical difference compared to simulations executed with historical-records (A.Helseth et. al [14]). SDDP relies on aggregated reservoirs, and can solve a given problem in reasonable time.

Martinez et. al showed that SDDP caused lower average power-generation and higher operation cost compared to SDP. This is explained by the discretization of the state space by not calculating every part of the problem, accounting for all relationships in every step [20].

Lately, *Genetic Algorithms* (GA) has found its place in hydropower optimization. GA mimics the behaviour of genes in a gene-pool, where each individual in the population has its own set of genes. The individuals are rated using a fitness function, and based on their fitness a subset of individuals are chosen to have their genes put through a crossover function. New individuals are created and given a set of genes returned from the crossover function. Mutations may also occur in the new individuals' genes. GAs are suitable for parallel search within a search space, due to working with a population of possible solutions[16].

MV. Devisree et. al applied GA alongside a Linear Programming model (LP) and showed an increase in power generation with the GA approach compared with the LP model[9].

Other non-linear approaches, like *Particle Swarm Optimization* (PSO), have produced interesting results and is characterized as a good optimization approach for global optimum problems [18, 4, 25]. S. Liao et.al [19] reviews these approaches and points out the difficulties related to PSO. A concern is the stochastic nature of the problem, making initial parameters a source of trapping the PSO in a local optimum. The PSO algorithm also suffers from poor fine-tuning capabilities.

Recently M. Azizipour et. al (2016) presented the *Invasive Weed Optimization Algorithm* (IWO) [3]. IWO is a stochastic optimization algorithm inspired from weed colonization. Weed colonization is initialized by invading a cropping system by means of dispersal. The weeds are located in unused space between the crops and take remaining resources so that they can grow to flowering weeds and produce seeds. Weeds better adapted to the environment has increased probability of producing more seeds, which consequently leads to more new weeds in the system. This process is repeated until the maximum number of weeds is reached.

IWO mimics this process by defining an initial population of seeds that gets

Algorithm 1 High-level pseudo code for IWO

```
Generate random population of N_0 solutions
for i ← 0, maximum number of generations do
    Calculate maximum and minimum fitness in the colony
    for all w ∈ W do
        Calculate number of seeds from w according to its fitness
        Randomly distribute seeds over the search space with normal distribution around w
        Add generated seeds to the solution set, W
    end for
    if W > W_max then
        Sort the population W in ascending order of their fitness
        Truncate population of weeds with worse fitness until W = W_max
    end if
end for
```

spread randomly over the field, where seeds and field represent randomly generated initial solutions and N-dimensional problem space, respectively. A predefined function is used to calculate the fitness of each seed in the colony. The seed then calculates the number of new seeds that it is allowed to produce based on its own fitness, and the max fitness in the colony. The results presented by M. Azizipur et.al shows that IWO is more efficient and effective than both PSO and GA for single and multi-reservoir systems [3].

S. Liao et.al [19] released a paper in 2017 that proposed an extension to the PSO algorithm to compensate for some of the drawbacks described when working with large search spaces (as described above). The paper discovered that an increase in sub-populations (population quantity) with equivalent population sizes help PSO converge towards global optimum. An increase also improves reliability through randomness in the population. The downside of increasing population sizes is increased computational penalty in terms of processing time. A parallel implementation of the algorithm, called *Multi-core Parallel PSO* (MPPSO) is therefore presented which leverages the multi-threaded capabilities of modern computers.

In later years, multi-core processing has been widely used for parallelized processing in computer science. The increase in cores available, as well as the growing maturity of GPU related frameworks like Nvidia CUDA, enables a severe increase in compute capabilities as long as the algorithm applied can be rebuilt for efficient parallel execution.

Several advances has been made when harnessing the power of parallel-processing. One recent example is the extension of PolyAco to PolyAco+, where T. Tufteland et.al parallelized the single-threaded PolyAco algorithm and optimized it to run on a CUDA enabled GPU, showing a performance increase with a factor of 80, and improved accuracy [34]. Other advances have also been made in recent years, including parallel implementations of IWO [11] and GA [30].

A untested [to the best of our knowledge] algorithm in the field of hydropower optimization is the learning automata, a reinforced learning capable algorithm that falls under the group of policy iterators (direct manipulation of the policy π), and can be classified under the same roof as other policy iterators, like evolutionary algorithms (hereby GA, PSO and IWO) as described earlier. A learning automata can learn the optimal actions when acting against a stochastic environment. As with IWO, learning automatas have a low computational cost, combined with rapid and accurate convergence and has been proposed as a solution to combinatorial optimization problems. Several papers discussing the use of this algorithm exists. In [24, 10] learning automatas are used to solve the classical equipartitioning problem, with several orders of magnitude faster convergence than any other known solution at the time. In [22] B. J Oommen et.al presents a solution to the uniform graph partitioning problem that outperforms other algorithms, such as local search and genetic algorithm implementations.

In addition to the above findings, a report by O. C Granmo et. al presents a learning automata solution to the *Satisfiability Problem* (SAT)[12], where learning automatas are incorporated with the Random Walk algorithm (LARW). Here each variable in the SAT equation is replaced by a learning automaton. The automaton chooses an action, true or false, and the whole equation is evaluated based on the changes introduced by this single LA. This sequential form of local feedback proved to be efficient in solving the SAT problem and serves as a basis to the LCS scheme presented in this thesis.

Previous results have shown that learning automatas are capable of solving complex combinatorial optimization problems when interacting with unknown stochastic environments, and as such could be a potential approach for solving complex issues related to hydropower.

3 Approach

A model is implemented to emulate an environment. This environment is used in the verification and testing of all algorithms. The algorithms input a solution i, and is returned a profit p as shown in Figure 5.

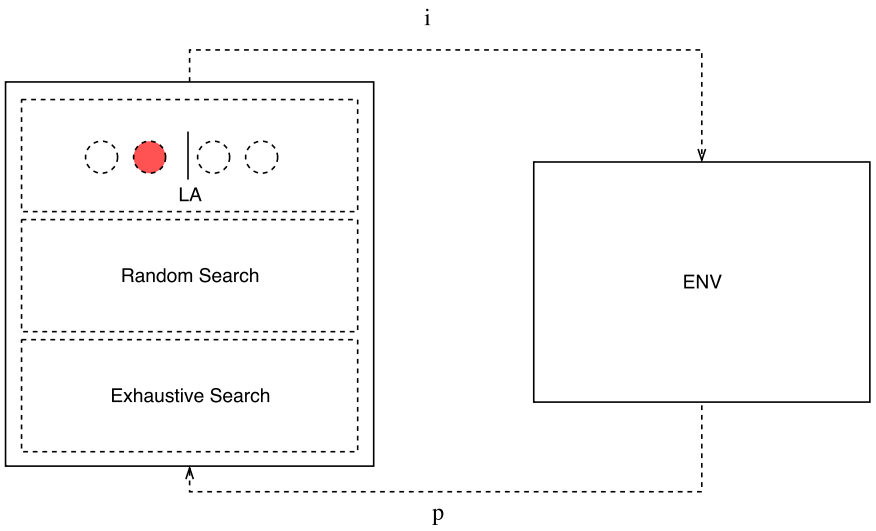


Figure 5: Relationship between environment and algorithms.

3.1 Environment

The model is made to emulate a realistic environment for use when developing and verifying algorithms. The model is based on a simple graph structure, where each *vertex* is an element from a hydropower system. The vertices are connected using *edges*, which simulate water flow in the system. A simple system is described in Figure 6, which shows the simplest model of a single-reservoir system. Here the *reservoir* refers to a magazine collecting water, and an attached *turbine* that produces energy from the water. The reservoir is filled by *inflow*. The inflow can be from a defined inflow-array (rain, rivers etc..) or be inflow from an upstream reservoir. This means that a vertex can have multiple edges as shown in Figure 7. The turbine vertex acts a the power-generating component that receives water through the *penstock*[5] edge,

[5]**Penstock:** waterway that leads water from the reservoir to the turbine.

and generate power when given an amount of water units. The water then flows downstream as *tail-water*[6].

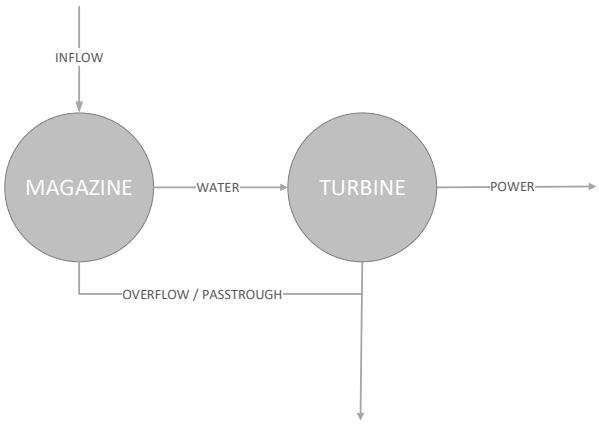


Figure 6: A simple model of a basic hydropower system.

The overflow / pass trough vertex defines the possibility of excess water being released downstream, circumventing the turbine. This may be done when the optimal production strategy requires such a move to maximize profit, or when the reservoir is full. Some systems may also require a constant release of water to keep in line with local environment policies. It is important to note that even though Figure 6 show the same edge for overflow and tail water, separate edges and vertices are a valid configuration. For instance a reservoir may have an edge for overflow that is separate from the tail water edge.

[6]**Tailwater:** water downstream from the turbine is called tail-water.

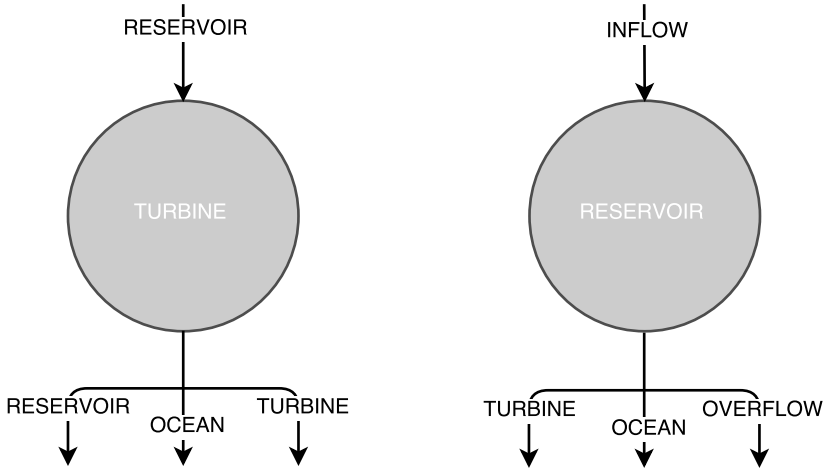


Figure 7: Possible connections between vertices: magazine and turbine

The *reservoir* vertex is represented programmatically using a composite data type in the programming language C, called a *struct*. The model is written in CUDA-compatible C for easy porting to GPU when necessary. The vertex has the following important properties.

water_capacity	The maximum allowable water-level before overflow.
water_level	The initial water-level at simulation start.
water_unit	Definition of one power-producing water-unit[7].
next_mag	Next reservoir for overflow / water downstream.
turbine	The turbine attached to the reservoir.

Table 2: Overview of reservoir properties

Next is the turbine vertex. This vertex is responsible for generating power from released water units. A released water unit yields a predefined amount of power.

reservoir	The designated reservoir receiving tail-water from upstream reservoir.
power	Modifier determining output efficiency gained from each water-unit.

Table 3: Overview of turbine properties

3.2 Assumptions

In terms of physics the model is greatly simplified compared to a real world system. This simplification is done for several reasons. The first reason is the limited time available for the project. A trade-off was necessary in order to accomplish the goals set for algorithm development. Another point of view is the complexity added to more advanced models. Verification of the model has been an ongoing task, made even harder when executed on a GPU. This makes a valid argument for keeping it simple. The aim is to provide results that may serve as a proof of concept for further development. There is no need for a complex model if the general algorithm is incapable of optimizing simpler systems.

The model consists of the most important elements in a hydropower plant; water, waterways and turbines. This excludes a magnitude of variables and functions. Most notable is the lack of physics in the production part of the plant. Since there is no virtual implementation of penstocks or intakes there is not enough data present in the model to calculate the water pressure. Without water pressure the calculation of the turbine speed / operating frequency is left out. This means that the effect from the turbine is constant, and X amount of water will yield Y amount of power in every scenario with no ties to reservoir levels. Another simplification is the movement of water, in the described model water moves each timestep. This means that water used in timestep t_0 is ready for usage in the next reservoir in timestep t_1.

In this preliminary research, input (inflow,prices) are treated as deterministic, i.e no uncertainty is assumed. For a real-world implementation, this would not be the case, as both the inflow and prices are of a stochastic nature. It is assumed that an algorithm that handles deterministic input can deal with stochastic input by running several simulations based on the uncertainty of the input.

3.3 Reference system

To test the proposed solutions, a reference system was built. This system is supposed to emulate the upper-part of the previously described *Mandalsvass-draget*. The system is Y-shaped, and consists of 3 reservoirs A, B, C. Figure 8 illustrates the system as a graph.

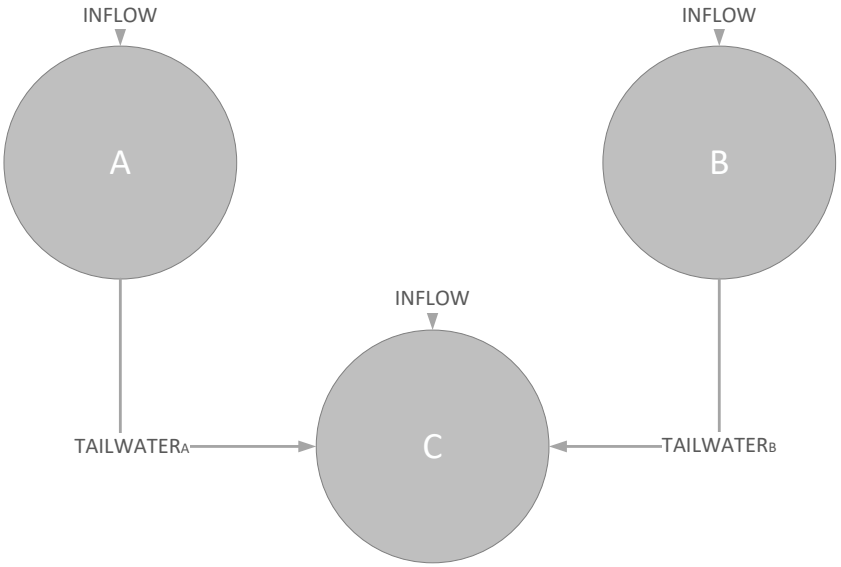


Figure 8: Presentation of reference model

The model serves as a heuristic function for a given solution. The input is a *production string* that instructs the model how to run the simulation over T timesteps. Figure 9 shows a production string. The presented string contains 3 reservoirs, divided by the stipulated lines, each number within these lines contains the production action for each reservoir for each timestep. As an example, the left most reservoir should not produce in t_1, produce in t_2 and not produce in t_3.

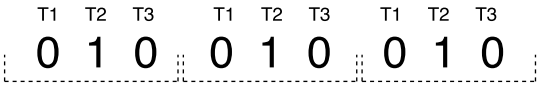


Figure 9: Production-string

The returned value from a production string simulation is the total profit, used when measuring success. The model is timestep based, which means

that the calculations done are valid for a given timestep range. In the experiments, 1 timestep is defined as a whole day, 24 hours. The simulation will start at t_0 and process all graphs in a downstream fashion, starting at the top.

The model can take 2 different values as a production action. These are 0 and 1. 0 denotes a strategy where the graph should not produce, while 1 denotes a production strategy where power is being produced, generating profit.

Profit Generation
The profit is a direct result of the inputted production string, and is calculated with the help of the price in the current timestep, water available in the reservoir and the current production string. In addition, each reservoir has a modifier variable that makes it possible to tweak input/output efficiency.

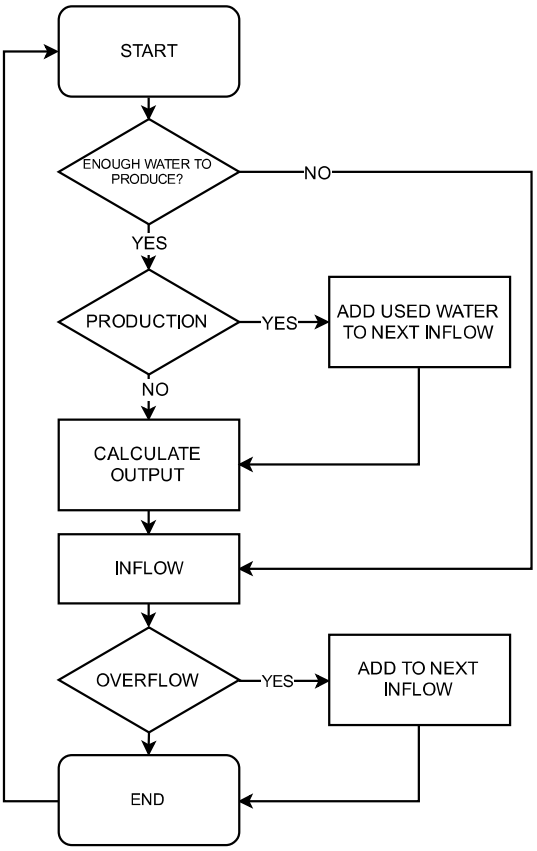


Figure 10: A flowchart illustrating the inner workings of the model.

3.4 Parallelization

The model has a sequential design and can therefore be run on singe threads. This means that each thread is responsible for running their own simulation, and enables the execution of several thousand simulations in parallel.

Global inflow and price arrays gets copied from host-to-device, along with initial parameters such as water level, turbine efficiency constants and reservoir water capacity variables. This set of input is shared among all threads running this simulation. The next step is to create the base production string that will be run through the model for simulation. The end result (profit) is returned to identify the simulation(s) that made the best profit.

Each block of threads shares the same scenario (input parameters), but the architecture supports running different scenarios on multiple blocks as long as there is enough memory available on the GPU.

The result of the model is a highly customizable system enabling simulation of different hydropower scenarios.

One concern is the linearity of the model. The loss of important properties may make the model unable to simulate the dynamics in a real world system, because of the less accurate detailed description. This may be due to the simplification of the hydraulic model as described in section 3.2.

The model itself is sequential and not distributed across several threads. The run-cost of a single simulation is low, which serves as the main reason for this implementation. A sequential design is also easier to verify and can be executed on the CPU for testing and debugging. The low-processing costs also make it possible to utilize parallel threads running the same scenario searching for an optimal solution.

The model is verified in terms of conservation of mass after each alteration. A test set is used for simulation to verify that the integrity of all values is intact.

3.5 Algorithms

The defined environment model is used in the implementation of the presented algorithms; exhaustive search, random search and learning automata.

3.5.1 Exhaustive Search

As a preliminary task, exhaustive search was implemented to create a baseline for further algorithms. As mentioned in the introduction, exhaustive search is not viable for an NP Complete problem because the search space and computational cost will increase exponentially in both size and difficulty as the problem is scaled (see Equation 2).

The exhaustive search implementation was designed for parallel execution on a GPU. This enabled the algorithm to find optimal solutions in larger search spaces due to increased computing capabilities across many parallel threads. This was seen as very beneficial as the results could be used to verify the validity of other algorithms. Manually calculating the optimal solution gets harder for each added additional timestep. The increase in possible solutions of the reference system is given by Equation 1, and further elaborated in Equation 2.

$$A^{R \cdot T} \tag{1}$$

Here,P denotes the number of different production actions that is possible (produce / not produce), R denotes the number of reservoirs and T denotes the number of timesteps. A few example calculations reveal the exponential nature of the problem.

$$\begin{aligned} 2^{3 \cdot 5} &= 3,27 \cdot 10^4 \\ 2^{3 \cdot 10} &= 1,07 \cdot 10^9 \\ 2^{3 \cdot 20} &= 1,15 \cdot 10^{18} \end{aligned} \tag{2}$$

This greatly limits the total number of timesteps an exhaustive search algorithm is able to solve for.

Implementation
The implementation of exhaustive search is done by calculating the total size of the search space, and then queue an equal number of threads on the GPU to simulate one unique production string each. All necessary variables gets copied into GPU memory, and when the threads have finished their simulation and stored the results in GPU global memory, the data gets copied from the GPU to CPU memory (RAM). If needed, the algorithm is split into separate batches in cases where there is not enough memory available on the GPU. Flowchart 11 shows this process in more detail.

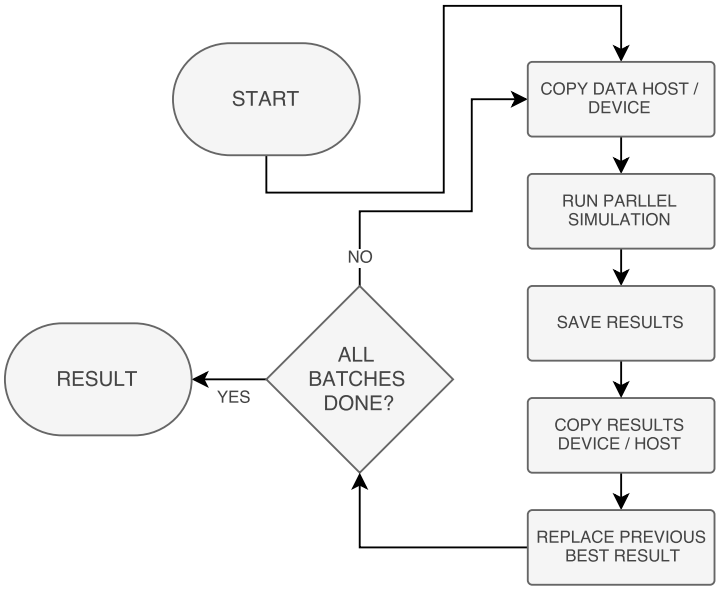


Figure 11: Exhaustive search flowchart

A method was needed to generate all the unique production-strings, and it had to be computationally viable. This was solved by exploiting the fact that when programming on a GPU, each thread has its own unique ID. A function was written to convert each thread's unique ID into its binary representation. As long as the number of spawned threads equals the size of the search space, this method ensures that each thread gets its own unique production-string, and that all possible production-strings are created.

A scenario with 1 timestep and 3 reservoirs can be used as an example. Here, the size of the search space is $2^{3*1} = 8$, and can be translated to 8 unique production strings. A binary conversion table for this scenario is presented in Table 4. As can be seen, zero padding is necessary to ensure that the

production string have the correct length.

Thread ID	**Binary**	**Padded**
0	0	000
1	1	001
2	10	010
3	11	011
4	100	100
5	101	101
6	110	110
7	111	111

Table 4: Thread ID to binary conversion example

3.5.2 Random Search

The next step involved creating a simplified version of random search, and have it run in parallel on the GPU. The basic principle of a random search algorithm involves minimizing the distance to an optimal solution be taking small, iterative steps within a defined hypersphere of possible solutions. The hypersphere can be defined in several different ways, but is usually determined by some sort of stochastic process.

The variant of random search implemented in this thesis is called *Uniform Random Search* (URS). URS is, as the name entitles, a variant of random search that uses an all random approach. This means that there are no defined hyperspheres, except the bounds defined by the number of timesteps in the simulation. The random element is the unique production string. The reward / solution element is the returned profit from the simulation. After each iteration, the production-strings yielding the highest profit are compared to a global high, and saved if equal to or greater than this value. The algorithm is finished after the preset number of iterations have been run.

In the first iteration of this algorithm, a similar technique to the one used by exhaustive search was created. To generate the random production strings, the GPU was first instructed to generate a given number of random floating point numbers in the range $[0, 1]$. The numbers were stored in global GPU memory. The main part of the algorithm were then launched with an equal amount of threads as was used to generate the previous numbers. Each thread will then get its designated random number, multiply it with the total size of the search space and then type cast the result into an integer. The threads will then take this new integer value and convert it to a binary representation before running it through a simulation. A flowchart demonstrating these steps are shown in Figure 12.

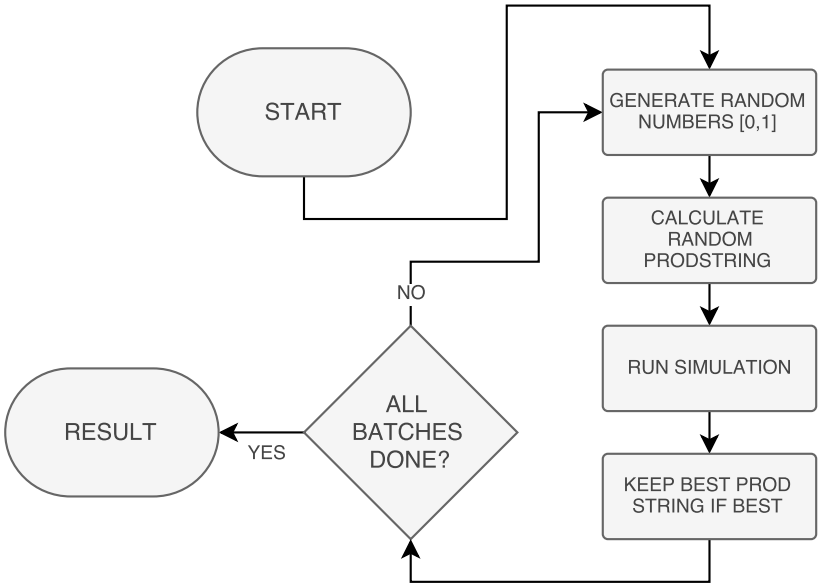


Figure 12: Flowchart illustrating the Random Search GPU implementation

This approach was both fast and efficient, but as the search space grew larger, the floating point numbers started to lose precision[8] when the threads did their multiplication. This lead to precision errors that in some cases kept the algorithm from finding an optimal solution.

[8]http://docs.nvidia.com/cuda/floating-point/#axzz4gxODUGkr

To counter this issue, the algorithm was rebuilt and the integer to binary method was abandoned. A new design approach was taken, being careful to avoid potential precision loss. It works by doing the following steps:

1. Generate n uniform random numbers in the interval $[0, 1]$, storing them in global GPU memory. The amount of random numbers generated are decided by doing the the following calculation: $parallel_simulations \times timesteps \times reservoirs$.
2. Allocate empty space in global GPU memory to hold the number of random production string, equal to the amount of $parallel_simulations$. Launch the GPU with as many threads as there were generated random numbers. Each thread will then apply its designated random number to equation 3, storing the output in its designated memory location in the production string array.
3. The GPU is now launched a third time, but now with as many threads as there are productions strings available in the global GPU memory from the previous step. Each thread copies their designated production string, runs it through a simulation and stores the result back into global memory.

An illustrative explanation is done with the help of Figure 13. The figure is separated into 3 different blocks (stipulated lines), and each block represents one of the steps in the algorithm. In the provided example, we are operating with a simulation count of 3, combined with 3 timesteps and 3 reservoirs.

$$f(n_x) = \begin{cases} 0 & n_x < 0.5 \\ 1 & n_x \geq 0.5 \end{cases} \tag{3}$$

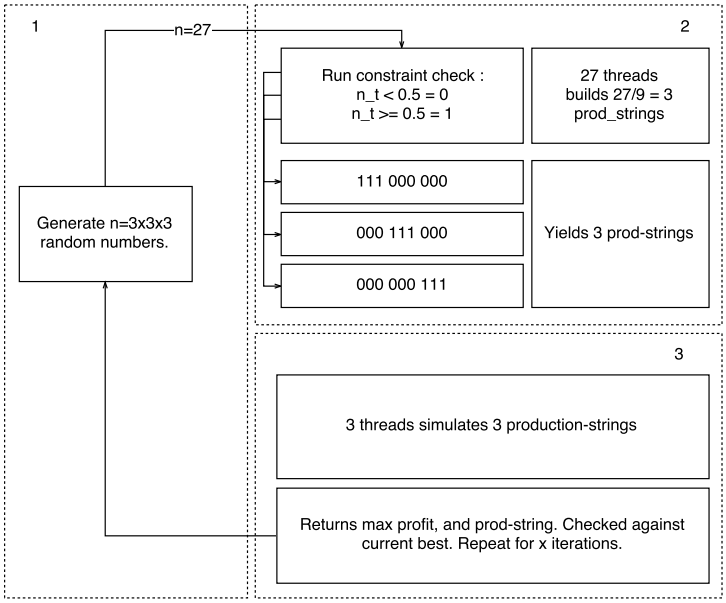


Figure 13: Figure illustrating the inner workings of the improved random search algorithm

As will be shown in the experiments section, if the exact number of possible solutions are known, it is possible to calculate the probability for Random Search to find an optimal solution after a given number of attempts. These calculations can be done using the following equations:

$$P(x) = \frac{n!}{(n-x)!x!} \cdot p^x \cdot q^{n-x} \tag{4}$$

$$P(x \geq 1) = 1 - P(x = 0) \tag{5}$$

In Equation 4, n denotes the number of trials and x denotes the total number of successes. p and q denotes the probabilities for success and failure, respectively. Since our interest lies in the probability to find any one arbitrary optimal solution, Equation 5 can be simplified to Equation 6.

$$P = 1 - q^n \tag{6}$$

3.5.3 Learning Automata

The main algorithm presented in this thesis is the learning automata implementation. The algorithm is based on a finite-state automata and makes use of parallel computation techniques.

The implementation is based on a state-of-the-art technique using a new proposed form of parallel local feedback. As described in section 1, a learning automata consists of two different parts; the automata and the environment. The environment returns output based on the input from a single or group of automatons. The output is called feedback and is a reward probability for the automaton based on the effect their actions has on the environment.

A commonly used form of feedback is the global feedback scheme. Here all automatons are working together in a goore-game situation where they share the results as an input to the environment (a unified solution), the automatons are then given the same probability reward from the environment and a random process occurs to decide the final outcome.

In this type of feedback, no direct relationship between each automaton and their decisions are obvious. This lays the ground for the introduction of *local contribution sampling*, or LCS, a novel contribution for a local feedback approach. LCS looks at all the actions given from the automatas collectively, as well as looking at each action individually. By separately assessing each automatons contribution to the whole, LCS is able to converge more quickly to a global optimum.

The main concept behind LCS is optimization by exploiting the potential profit in small, individual changes, from the current best strategy. This is done through systematic testing of alternative strategies where each automaton changes its current action to test if this individual change moves the unified solution towards a more optimal strategy. This gives better ground for individual reward and allows the algorithm to create individual reward probabilities based on each automatons contribution.

A detailed process overview of LCS is presented in Figure 14. Opposed to the two-step generic global-feedback automata described earlier, LCS has 3 main steps. As with the other versions, a automata implementation is joined with an environment function, but an extra step called *action change* is added. Here each automaton reports its current chosen action, which is used to build a base-strategy. This is represented in Figure 14 as *base*. Each automaton

then do a action change from the *base* position corresponding to its own. I.e A_n flips B_n, where A is an automaton, and B is the base-strategy. Each of these alternative production strategies, $B, A_1, A_2 ... A_n$, is then simulated and the profit is returned to the uni-modal environment function 7.

$$f(x) = 1 - \frac{(x - min) \cdot 0.6}{max - min} + 0.2 \tag{7}$$

This algorithm act as the environment in the system and enables the local feedback scheme by calculating the reward probability with consideration of the global max and min profits from all A. x is the profit from the current automaton X_{A_n}. The calculation is done for all X_{A_n} and a calculated probability R is returned to each automaton where a stochastic process determines if a state-change occurs, based on the returned probability.

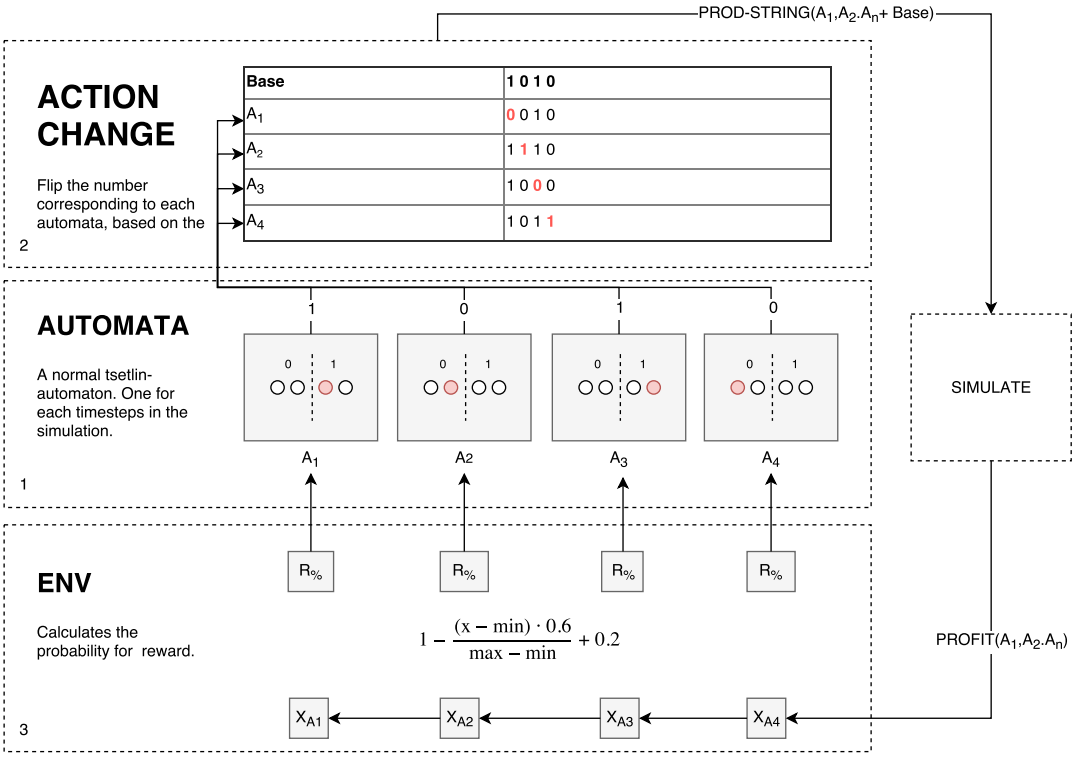


Figure 14: Learning Automata setup

This process is then iterated for a set number of times, and the highest profit is kept as the current optimal solution.

Another aspect of the LCS method is the decentralization properties. In the action change function each automaton needs to run its corresponding simulation, which can be done in parallel. When all sub-production strings $A_1..A_n$ are calculated, a min-max function is executed to get all the necessary parameters for Equation 7. On the GPU the min-max function is calculated in parallel, while on the CPU it is happening in a sequential manner.

GPU

The first implementation of the algorithm was made for GPU with the help of CUDA. To make the algorithm run on several threads the grid, block and thread layout was exploited. A block in a grid was reserved for an instance of the algorithm, where a set of threads within this block represented the individual automata in the simulation (one for each element in the production string).

Figure 15 represents the sequential processing in each thread based on the overall idea presented in Figure 14 and the section above.

Figure 15: Sequential gpu-thread overview

When keeping each automaton in separate threads, shared memory is used for communication between them. Since the implementation is using the LCS feedback-scheme, parallel execution is possible. The min-max function is parallelized having several of the threads work together, based on a technique called parallel reduction (see Figure 16)[9]. A custom approach where made to fit the needs of our algorithm. The method utilizes all threads to find the biggest and lowest values in parallel. This is done by two-on-two cascading comparison of numbers until both numbers are found. A graphical representation of this approach is shown in Figure 16. Here we represent the profit for all automatons in an arbitrary iteration as an array with the length of threads within the block (or the number of automatons).

In Figure 16, eight automatons are running on eight separate threads. The numbers represent the profit from each automaton in the current iteration. The goal is to sort the array so the highest value is at the left-most position, while the lowest value is at the middle. This divides the array in the middle into a high- and low end. In the first step, half of the threads compares their element to the associated position in the other end. After this step, it is guaranteed that the highest value resides somewhere on the left half, while the lowest value resides somewhere in the right. The next step compare numbers within the same sides, and the last moves the high-low combinations to position zero in each half. This method greatly reduces the computational cost, and keeps iteration count low, as shown by Equation 8 and 9. Here $N \in \{2^x \mid x > 1\}$, denotes the length of the array. Equation 8 calculates the number of steps needed to find the min-max in a sequential manner, while Equation 9 shows the number of steps for parallel computation. As seen the parallel approach is much more efficient.

$$2N - 1 \tag{8}$$

$$\log_2 N \tag{9}$$

When parallelizing the min-max function the only sequential part in the entire algorithm is the simulation of the base production-string (the production-string created by querying all the automatas in the beginning of each iteration). This calculation is done by thread zero, before this thread start calculating its assigned sub-string.

[9] http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

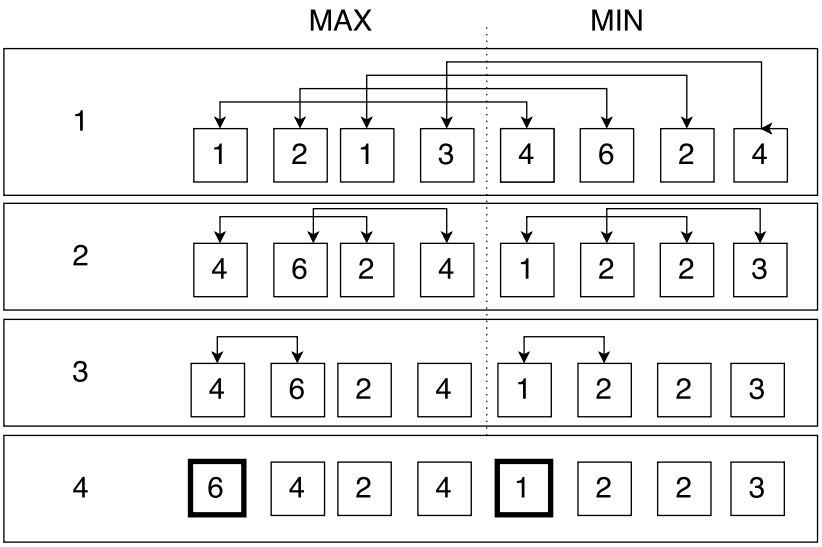


Figure 16: Parallel reduction example

By keeping each instance of the algorithm within a single block, several different instances can be executed in parallel. This may be beneficial when testing different hyper-parameters, or when working with uncertainty in stochastic input.

The results of the GPU implementation is a highly-scalable (block-wise) algorithm that handles a high count of parallel algorithm instances.

CPU

For performance testing, a multi-threaded CPU implementation where made. This implementation is fundamentally the same as the GPU implementation, without the added benefit of massive parallel execution possibilities due to core and thread limitations. Here, the parallelization is performed while simulating all the production strings (the action change stage of the algorithm). Everything else is done in a sequential manner. This resulted in the execution times shown in Table 5 The results confirms that to really benefit from a GPU implementation the algorithm needs to keep as many threads as possible occupied on the GPU at all times. To accomplish this, several instances needs to be executed. This means that the GPU implementation is a great addition when exploring hyper-parameters, or working with stochastic input, but the CPU-implementation is still faster for smaller scenarios with a moderate amount of timesteps.

Timesteps	Processor	Number of iterations		
		1000	5000	10000
10	GPU	0.26 ±0.0860	1.17 ±0.0736	2.10 ±0.0588
	CPU	0.09 ±0.0561	0.22 ±0.0812	0.38 ±0.0951
50	GPU	2.77 ±0.0825	13.36 ±0.0941	26.44 ±0.0973
	CPU	0.17 ±0.0787	0.49 ±0.0980	0.90 ±0.0759

Table 5: Average computation time in seconds

Minimizing impact of invalid production-strings.
In later stages a fundamental change was made to account for non-valid production strings. The model handles all action changed production-strings ($A_1...A_n$ in Figure 14) based on available resources in the system at all times during the simulation. Since the environment is unknown to the automata a production-string may instruct the model to produce in cases where there is not enough resources available. In earlier versions, this was handled by returning a profit of 0 for this individual production-string A_n. The idea behind this were that the automaton should have a low probability of changing to the state causing the invalid production-string.

In the improved implementation this is handled by returning a partial profit for invalid production-strings. Here an invalid action results in a penalty to the overall profit but is not returned as 0. Instead, the returned profit is the sum of all valid actions in that production-string.

To further elaborate upon the implications of this change, an example is presented. As shown in Figure 17 an arbitrary base is presented, with corresponding production-strings $A_1..A_n$. In addition to this method 1 and method 2 columns represent the profit calculation for each A_n. Method 1 refers to the old approach, returning 0 for invalid production-strings. Method 2 refers to the new approach where the returned profit is the sum of all valid actions. The marked element in A_2 represents the non-valid production-string. As seen, the production string is actually the second best, even tough it contains an invalid action.

		METHOD 1	METHOD 2
BASE	1 0 1 0	-	-
A_1	**0** 0 1 0	5550	5550
A_2	1 **1** 1 0	**0**	**6000**
A_3	1 0 **0** 0	5800	5800
A_4	1 0 1 **1**	6200	6200

Figure 17: Cost calculation for a given base with the old and new method[10]

[10]In this example the profit gained from base is not included.

To help visualize this, Figure 18 presents the calculated reward probability from the environment (Equation 7) for all A_n sorted by probability. As shown this approach results in an uneven distribution of probabilities where A_1, A_3, A_4 gets a fairly low probability of reward, while A_2 gets a high probability. Note that reward in this context refers to *not* changing the coherent action in the base production-string. This should occur when the modified action-change alternation A_n returns a higher profit than the base production-string. Since a profit of 0 is treated as an extreme the algorithm rewards with an uneven scaling, forcing all probability rewards towards one of the ends (0.2 / 0.8). This keeps the algorithm from converging towards the optimum solution because automatons with low profit production-strings may change their action due to the low reward probability given to them. A_1 is in our example the worst performing production-string with a returned profit of 5550, but with the 0-profit strategy it is rewarded with a fairly low reward probability increasing the chance that this automaton will alter its state, even though this should normally not occur. A better scenario would be a more even distribution (ref Figure 18).

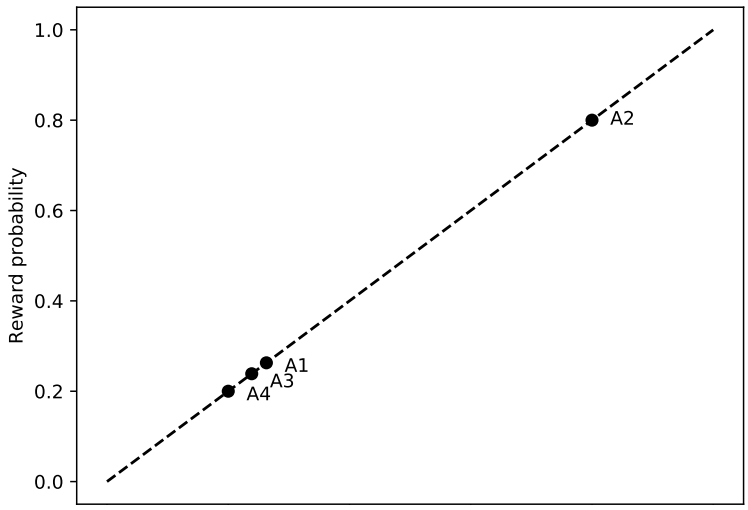


Figure 18: Cost calculation for a given base with method 1

To counter this behavior method 2 was applied to the algorithm. As seen in Figure 17 a partial profit of 6000 is returned for production-string A_2, this shows that the strategy chosen here is fairly profitable, and is the second best after A_4. The correct action here would be to yield a low probability of reward for A_2 since this is a favorable strategy. Figure 19 demonstrates

how this method looks for all A_n. As seen the distribution of probabilities are evenly distributed, motivating convergence towards an optimal strategy.

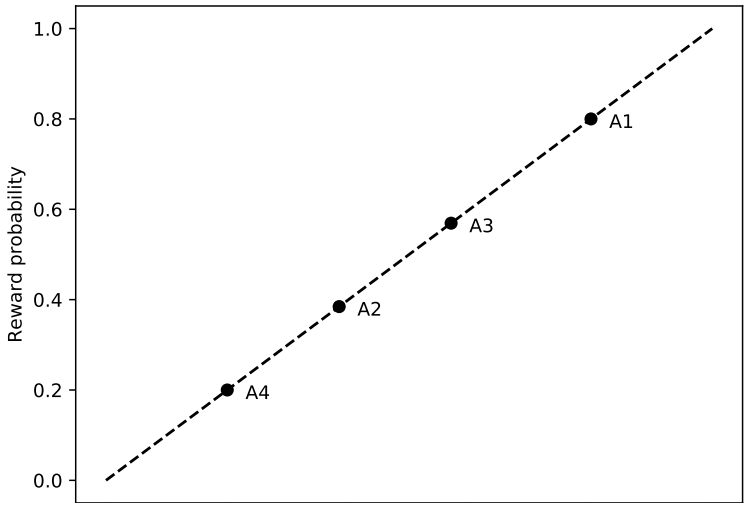


Figure 19: Cost calculation for a given base with method 2

To verify the superiority of method 2 a practical test were created. In this test an experiment is executed using method 1 and method 2. The test environment is a 3 reservoir scenario with 10 timesteps, yielding a total of 30 automatons with a fixed low/high price input and a static low inflow. This scenario is designed in a way that the algorithm needs to conserve the water until the price shifts from low to high to an the optimal solution (highest profit). Further elaboration and verification of this test is shown in section 4. The experiment compares the number of iterations needed to find the optimal solution and is presented in Table 6. As shown, method 2 clearly outperforms method 1, with an iteration reduction factor of ~ 495.

Reward function	Iterations
Method 1	29596.73 $\pm$853.54
Method 2	60.82 $\pm$0.85

Table 6: Reward Function Comparison

Since an optimal solution created with method 2 may contain illegal moves, the solution is simulated a final time after the last iteration to change invalid

actions to the "not-produce" position, denoted with 0. This ensures that all outputted final production-strings are valid. Given the benefits of method 2, it is used in all experiments presented in chapter 4.

Even though exhaustive search and random search are rather naive, they serve a purpose in terms of testing and verifying as section 4 shows. Since the problem is assumed to be NP-Complete an optimized answer is hard to calculate, and as timesteps and complexity increase the possibility to make realistic and verifiable tests disappears. Exhaustive search techniques like the ones presented in this section helps search for optimal (exhaustive search) and sub-optimal (random search) answers to help debug and verify the LA algorithm. As shown later the naive approaches were able to solve several complex experiments which helped the overall research.

4 Experiments

To verify the validity of the algorithms 5 experiments is created. The experiments are designed to validate the behavior of the algorithms in a range of different hyper-parameters and inputs. For all experiments, the efficiency of the reservoirs / turbines are static, and are documented in Table 7.

Reservoir	**Input**	**Output**
A	$25m^3/s$	45 MWh
B	$16m^3/s$	100 MWh
C	$25m^3/s$	40 MWh

Table 7: Reservoir / Efficiency

The important part of Table 7 is the relationship between the numbers. Reservoir B is the most efficient power-producer, with the highest power output for the least amount of water. The calculations within the algorithm are done in terms of cubic metre, denoting inflow as $(m^3)/s$, reservoir capacity as $m^3 \cdot 10^6$, and power output as MWh. Profit is calculated using the following formula: $Profit = \sum_{t=0}^{T} \sum_{r=0}^{2} Output_r \cdot Price_t \cdot 24$, where T equals the total number of timesteps, $Output_r$ corresponds to the power outputs in Table 7 and $Price_t$ is the available price at timestep t. In layman terms, this means that the returned profit from a production string equals the sum of the power output of all producing reservoirs r, multiplied with the price at all timesteps T, and lastly multiplied with 24 hours.

Preceding the results in each experiment, a probability table for Random Search algorithm will be presented. This table shows pre-calculated probabilities for Random Search to find a global optimum. The table includes both the probability for a sequential, non parallel version, run on a single thread, and a parallel version, run on a large number of GPU threads.

The results in each experiment are presented in two forms; First, a table showing the percentage of how many times, across all test runs, global optima was found, using Random Search, LA with global feedback (LA Global) and LA with LCS (LA LCS). Second, a graph is presented showing the average convergence rate of the learning automata using LCS throughout all test runs. In each experiment, both LA Global and LA LCS have been tested

with 1000 and 10000 iterations, respectively. To make the Random Search algorithm comparable, the number of iterations used in the LA algorithms were translated into an equal amount of attempts, letting each thread in the Random Search algorithm make as many attempts at randomly generating an optimal solution as the number of iterations used by the LA algorithms.

Due to the difference in computation time between the LA algorithms and Random Search, the LA is averaged over 1000 test runs in each experiment, while Random Search is averaged over 100 test runs. In addition, the scenarios used in experiments 1 through 4 are all limited to 10 timesteps, with the exception of experiment 5 which is a 50 timestep scenario. This enabled exhaustive search to find the highest achievable profit, and the exact number of optimal solution. Preliminary test had shown that running exhaustive search on more than 10 timesteps, using the current GPU implementation, the computation time increased drastically. Using 12 timesteps, the algorithm finished after $\sim$ 15minutes, and with 13 timesteps the algorithm finished after $\sim$ 60minutes. As a consequence of this, exhaustive search has not been run in any experiments using a scenario with more than 10 timesteps, and thus neither the highest achievable profit, nor the exact number of optimal solutions are know. The only exceptions to this are if a scenario with more than 10 timesteps has been explicitly designed in such a way the the optimal solutions are known.

The LA algorithm with global feedback is implemented using the techniques described in section 1.4. The algorithm utilizes a modified version of the reward function used by LA LCS, shown by Equation 10 and Equation 11.

Remark A production run refers to starting an algorithm and letting it run for a given number of iterations.

$$g(x) = \frac{(x - min) \cdot 0.6}{max - min} + 0.2 \tag{10}$$

$$P(x) = \begin{cases} 0.2 & x < min \\ 0.8 & x > max \\ g(x) & min \leq x \leq max \end{cases} \tag{11}$$

Figure 20 presents the different scenarios used in the experiments.

Scenario	Optimal Solutions	Reservoir Start Levels	Inflow	Price
1	20	$A = 10m^3 \cdot 10^6$ B = $10\text{m}^3 \cdot 10^6$ C = 0m^3	$2.0m^3/s$	$(p_0, ..., p_t), p_t = \begin{cases} \$5 & t < \frac{T}{2} \\ \$15 & t \geq \frac{T}{2} \end{cases}$
2	200	$A = 10m^3 \cdot 10^6$ B = $10\text{m}^3 \cdot 10^6$ C = 0m^3	$2.0m^3/s$	$(p_0, ..., p_t), p_t = \begin{cases} \$15 & t < \frac{T}{2} \\ \$5 & t \geq \frac{T}{2} \end{cases}$
3	2	$A = 10m^3 \cdot 10^6$ B = $10\text{m}^3 \cdot 10^6$ C = 0m^3	$2.0m^3/s$	$(p_t) = (\$10, \$19, \$13, \$16, \$10, \$10, \$6, \$10, \$9, \$8)$
4	1	$A = 2.16m^3 \cdot 10^6$ B = 0m^3 C = 0m^3	$0.0m^3/s$	$(p_t) = (\$0, \$20, \$0, ..., \$0, \$20)$

Figure 20: Scenario overview

4.1 Experiment 1

Experiment 1 is constructed in such a way that to reach global optima, the algorithm is forced to be conservative on how much water is used in the first part of the production run. As shown in Figure 20, scenario 1 has a low amount of water inflow ($2.0m^3/s$) throughout the entire run, and thus there is only a limited amount of water available for production. The production price in the scenario is split into two halves, where the first half is set at \$5, and the second is set at \$15.

Iterations	**Parameters**	**Probabilities**	
		1 thread (Seq)	10^5 threads (Par)
1000	$q = 1 - \frac{20}{2^{30}}, n_{par,seq} = 10^8, 10^3$	1.86×10^{-5}	0.84
10000	$q = 1 - \frac{20}{2^{30}}, n_{par,seq} = 10^9, 10^4$	1.86×10^{-4}	0.99

Table 8: Binomial Probability Distribution for Experiment 1

Algorithm	**Initialization**	**Number of iterations**	
		1000	10000
Random Search	-	0.91 $\pm$0.0561	1.0
LA Global	0	0.0	0.0
LA Global	1	0.0	0.0
LA Global	Random	0.0	0.0
LA LCS	0	1.0	1.0
LA LCS	1	1.0	1.0
LA LCS	Random	1.0	1.0

Table 9: Results from Experiment 1

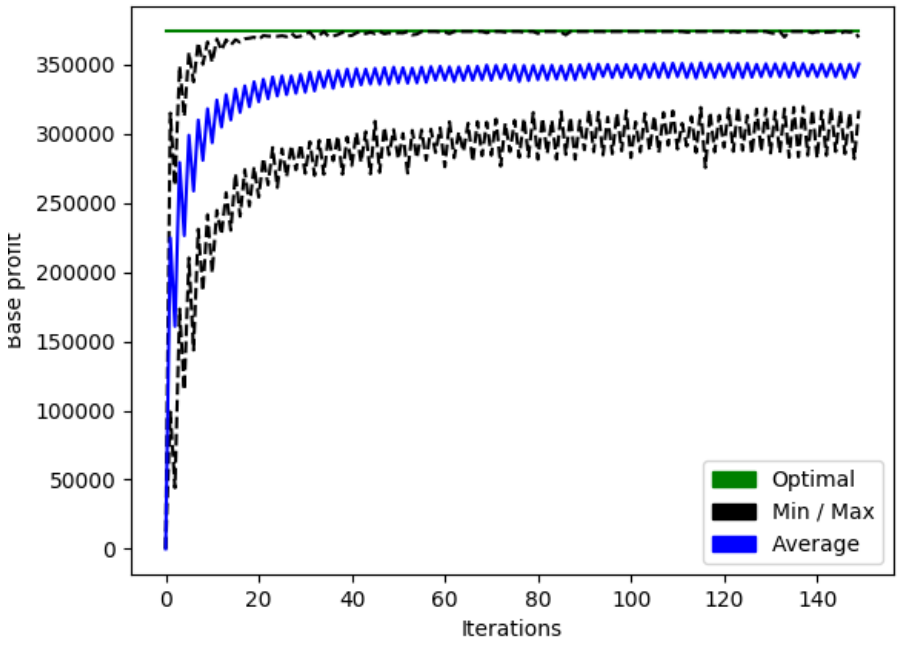


Figure 21: Experiment 1 convergence graph.

The results from Table 9 shows that LA LCS is capable of finding global optima from every initialization state, which indicates that the algorithm is capable of accounting for complex requirements, such as conservation of water. LA Global is not able to reach a global optima in any initialization setup.

Figure 21 indicates that LA LCS converges to a near optimal state, where global optima can be reach by local search of the state space.

4.2 Experiment 2

In Experiment 2 the scenario has been reversed with regard to the production price. The first half of the input is set at \$15, and the second half is set at \$5. The consequence of this reversal is that it is no longer necessary to conserve water in the beginning of the run. Instead, it is more profitable to have the production rate high in the first half. This change also leads to a tenfold increase in the number of optimal solutions.

Iterations	**Parameters**	**Probabilities**	
		1 thread (Seq)	10^5 threads (Par)
1000	$q = 1 - \frac{200}{2^{30}}, n_{par,seq} = 10^8, 10^3$	0.18×10^{-4}	0.99
10000	$q = 1 - \frac{200}{2^{30}}, n_{par,seq} = 10^9, 10^4$	0.18×10^{-3}	0.99

Table 10: Binomial Probability Distribution for Experiment 2

Algorithm	**Initialization**	**Number of iterations**	
		1000	10000
Random Search	-	1.0	1.0
LA Global	0	0.16 ±0.0227	0.16 ±0.0227
LA Global	1	1.0	1.0
LA Global	Random	0.002 ±0.0028	0.002 ±0.0028
LA LCS	0	1.0	1.0
LA LCS	1	1.0	1.0
LA LCS	Random	1.0	1.0

Table 11: Results from Experiment 2

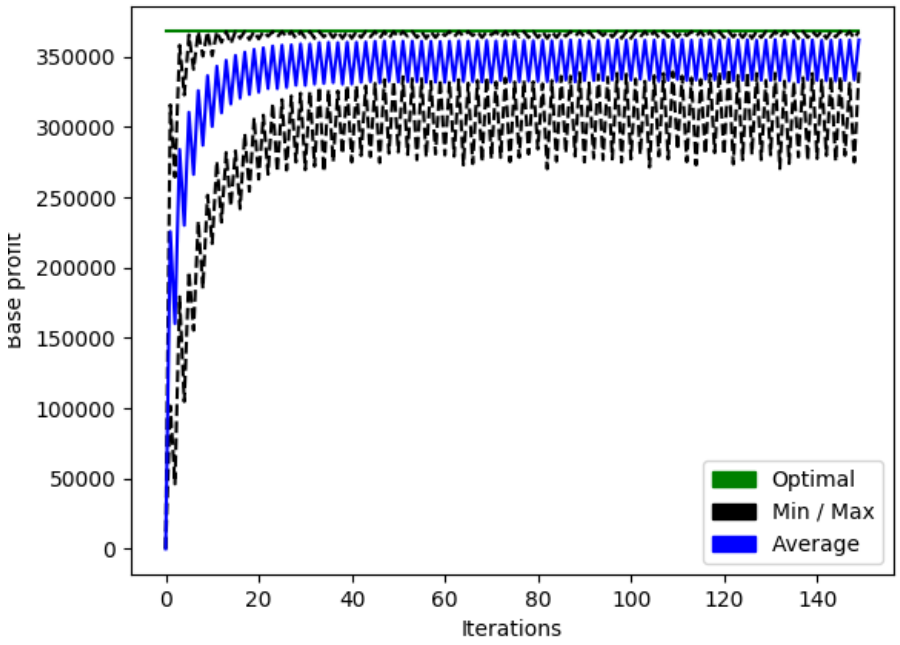


Figure 22: Experiment 2 convergence graph.

The results from Table 11 shows the effect from having an increased number of optimal solutions. All algorithms and variations were able to locate a global optimum with this specific scenario. Despite this, however, there is a significant difference between the performance of LA Global and LA LCS. While LA LCS is able to reach global optima 100% of the runs, LA Global struggles when initialized to 0 and random. When initialized to 1, LA Global is able to find global optima every run. It can be reasoned that this behavior is in coherence with the properties of an optimal production string for this specific scenario. Since it is not necessary to conserve water in the first half of the run, there will be more ones than zeros in the production string. This then means that fewer state changes are necessary to be able to find an optimal solution if the algorithm gets initialized to 1.

Figure 22 shows a smaller gap between the minimum and maximum line, and the average line. This indicates that the algorithm is converging to a more stable state than in Experiment 1.

4.3 Experiment 3

Experiment 3 follows the same direction as 1 and 2. The initial parameters are identical with the same inflow. The price is randomly generated and selected based on its property of only containing 2 optimal solutions. This makes for an interesting case because it limits the probability of the LAs to rely on the initialization method 1/0/stochastic to be in close vicinity to an optimal solution. This is further verified by looking at the two optimal production-strings in Figure 23, found by the exhaustive search algorithm. Here we see a variable distribution of state-actions along the production interval. It is also apparent that the only alternation is possible in later timesteps in reservoir A.

Experiment 3 follows in the same direction as experiment 1 and 2. The initial parameters and input are the same as the previous scenarios, except for the production price, which has been randomly generated for this scenario. With the generated price, only 2 optimal solutions exist, and they have the same distribution of ones and zeros as the solutions that exist for scenario 2. Despite this, given there are only 2 solutions it is unlikely that the initialization of the LA algorithms will have an impact on the results in this experiment.

```
     A          B          C
1111000100 1111110110 0111111111
1111010000 1111110110 0111111111
```

Figure 23: The two optimal production strings that exist for scenario 3

Iterations	**Parameters**	**Probabilities**	
		1 thread (Seq)	10^5 threads (Par)
1000	$q = 1 - \frac{2}{2^{30}}, n_{par,seq} = 10^8, 10^3$	1.86×10^{-6}	0.16
10000	$q = 1 - \frac{2}{2^{30}}, n_{par,seq} = 10^9, 10^4$	1.86×10^{-5}	0.84

Table 12: Binomial Probability Distribution for experiment 3

Algorithm	**Initialization**	**Number of iterations**	
		1000	10000
Random Search	-	0.18 ±0.0753	0.84 ±0.0719
LA Global	0	0.015 ±0.0075	0.022 ±0.0091
LA Global	1	0.03 ±0.0106	0.027 ±0.0100
LA Global	Random	0.0	0.0
LA LCS	0	1.0	1.0
LA LCS	1	1.0	1.0
LA LCS	Random	1.0	1.0

Table 13: Results from Experiment 3

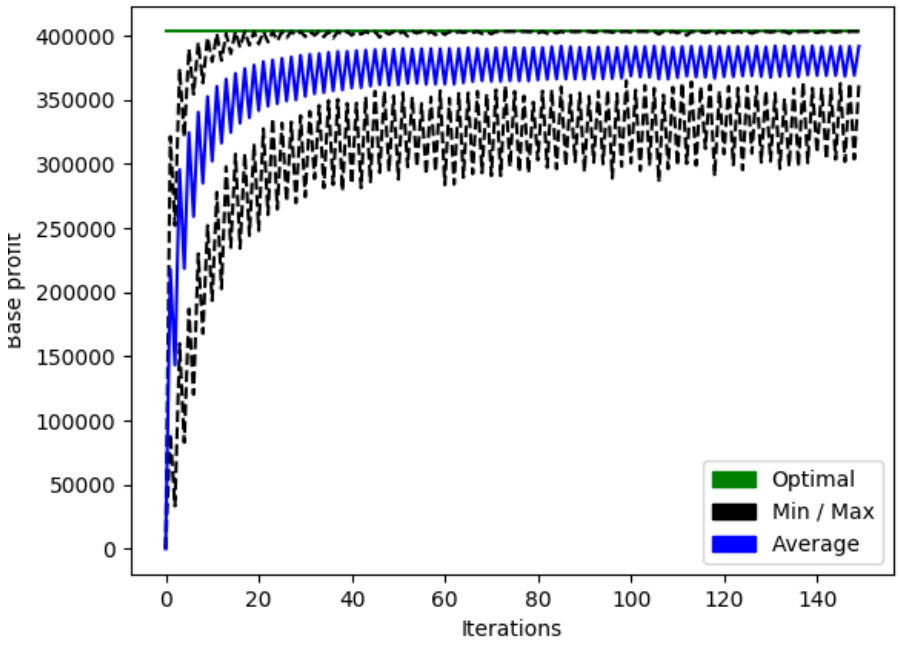


Figure 24: Experiment 3 convergence graph

Table 13 show that even though the optimal production strings were similar to the ones that exist for scenario 2, LA Global returned overall worse results compared with Experiment 2. When initialized at 1, LA Global did not return with 100% success rate as it did in Experiment 2, and when initialized with random states it was unable to find an optimal solution all together.

LA LCS returned with 100% success rate in this experiment as well, but this time Figure 24 shows signs of a slower rate of convergence than the previous experiments, indicating an impact in performance for LA LCS.

4.4 Experiment 4

Experiment 4 uses a scenario that has been designed in such a way the there will always exist only one optimal solution, and this solution is always known. This scenario has been deprived of almost all water, leaving just enough to let reservoir A be able to produce power once. The water will then flow into reservoir C, and can now be used to produce power one more time. The price has also been set to \$0 for all timesteps, except for the second and last. By doing this, the algorithm has to learn that after using the water in reservoir A at the second timestep, it has to wait until the very last timestep before permitting reservoir C to produce.

```
     A          B          C
0100000000 0000000000 00000000001
```

Figure 25: The single optimal production string for experiment 4.

Iterations	**Parameters**	**Probabilities**	
		1 thread (Seq)	10^5 threads (Par)
1000	$q = 1 - \frac{1}{2^{30}}, n_{par,seq} = 10^8, 10^3$	9.31×10^{-7}	0.09
10000	$q = 1 - \frac{1}{2^{30}}, n_{par,seq} = 10^9, 10^4$	9.31×10^{-6}	0.60

Table 14: Binomial Probability Distribution for Experiment 4

Algorithm	**Initialization**	**Number of iterations**	
		1000	10000
Random Search	-	0.08 ±0.0532	0.64 ±0.0941
LA Global	0	0.065 ±0.0153	0.022 ±0.0091
LA Global	1	0.056 ±0.0143	0.07 ±0.0158
LA Global	Random	0.032 ±0.0109	0.051 ±0.0136
LA LCS	0	1.0	1.0
LA LCS	1	1.0	1.0
LA LCS	Random	0.963 ±0.0117	1.0

Table 15: Results from 4

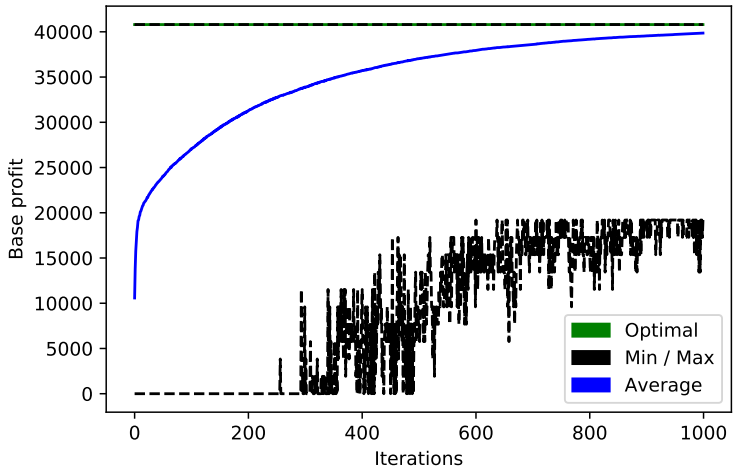


Figure 26: Experiment 4 convergence graph for 0-state initialization.

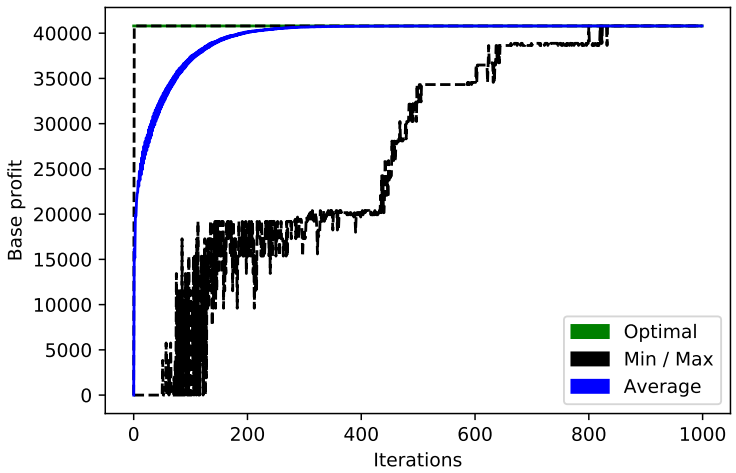


Figure 27: Experiment 4 convergence graph for random state initialization.

The results in Table 15 shows again how LA LCS is able to outperform LA Global. An interesting observation is how LA LCS has experiences a decrease in performance when initiated with random states. As an attempt to explain this behavior, it can be argued that when the states in LA LCS is initialized to all 1 or 0, the Learning Automata start moving towards their optimal state more quickly. Also, when the algorithm is initialized with random states, the Learning Automata has to spend more iterations exploring the local search space, learning how their individual actions are contributing to the whole,

before starting to move towards their optimal state. By comparing Figure 26 with Figure 27, there is a noticeable difference in how the algorithm converges, supporting what has been argued.

4.5 Experiment 5

Experiment 5 is an extension of Experiment 4, using the same scenario but with the number of timesteps increased to 50. The total size of the search space in this experiment is 2^{150}, or $\sim 1.43 \times 10^{45}$, number of possible solutions. As explained in the beginning of the experiments section, the size of the search space is to large for an exhaustive search, but because of the way this scenario was designed, the optimal solution is known.

Iterations	**Parameters**	**Probabilities**	
		1 thread (Seq)	10^5 threads (Par)
1000	$q = 1 - \frac{1}{2^{150}}, n_{par,seq} = 10^8, 10^3$	8.88×10^{-13}	8.88×10^{-8}
10000	$q = 1 - \frac{1}{2^{150}}, n_{par,seq} = 10^9, 10^4$	8.88×10^{-12}	8.88×10^{-7}

Table 16: Binomial Probability Distribution for experiment 5

Algorithm	**Initialization**	**Number of iterations**	
		1000	10000
Random Search	-	-	-
LA Global	0	-	-
LA Global	1	-	-
LA Global	Random	-	-
LA LCS	0	0.125 $\pm$0.0205	0.494 $\pm$0.0310
LA LCS	1	0.037 $\pm$0.0117	0.441 $\pm$0.0308
LA LCS	Random	0.0	0.0

Table 17: Results from Experiment 5

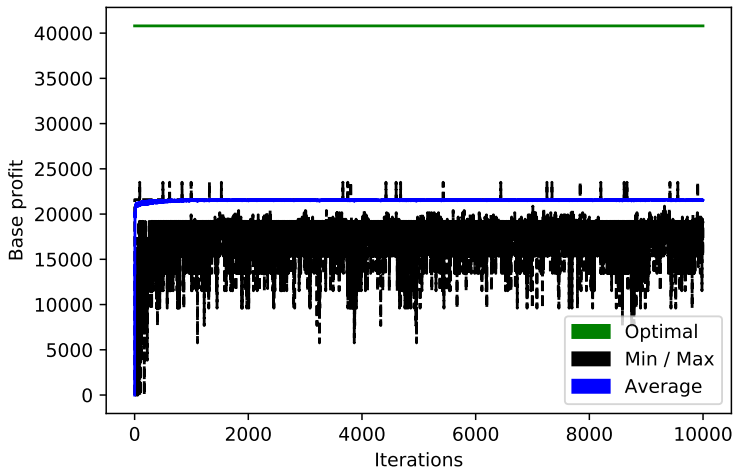


Figure 28: Experiment 5 convergence graph for zero state initialization.

The results shown in Table 17 clearly reflects the added complexity that has been introduced from the increased search space. The only algorithm that is able to find the optimal solution is LA LCS, but with a much lower success rate than in the previous experiments.Figure 28 shows that LA LCS is unable to converge to a state where the learning automaton representing the timesteps of interest for reservoirs A and C both settles at their optimal states. The fact that LA LCS is able to reach the optimal solution in nearly 50% of the runs shows that even when the algorithm converges to a local optimum, it is still able to search far enough to find the global optimum.

When running experiments with more than 50 timesteps, the GPU becomes indispensable. There are two reasons for this. First, the computational requirements when processing scenarios of this scale is a challenge for the limited number of available threads on a CPU. Second, the uncertainty that gets added when the search space is increased with a timestep amount of this size, greatly impacts the probability of finding an optimal solution. This can be deduced from Table 15 and Table 17. To counter this trend, additional runs needs to be executed in order to increase the probability of finding an optimal solution. Much like the GPU was used to increase the probability for random search, the GPU can be used to run several thousand instances of LA LCS, in parallel, thus increasing the probability of finding an optimal production-string. This is backed by the execution time comparison shown in Table 18. The comparison is made executing a 100 000 run with 1000 iterations on both the CPU, and GPU. This clearly shows the benefit of using the GPU.

Method	**Time**
LA LCS GPU	∼18
LA LCS CPU	∼72

Table 18: GPU vs CPU scenario execution runtime in minutes.

5 Conclusion

In this thesis we introduce Local Contribution Sampling (LCS), a novel optimization algorithm which is derived from the learning automata algorithm that relies on reinforcement learning for optimization, and parallel computing for decreasing runtime. LCS adds several modifications and improvements to the original learning automata, such as individual evaluation and feedback of automata actions, efficient parallelization design and the possibility for handling uncertain input using parallel GPU architecture.

LCS employs a novel approach in the interaction between the algorithm environment, and the learning automata. By individual evaluation of each automaton's action, LCS is able to give a more precise and decisive feedback to the actions. When applying this approach on scenarios where the global optimums are known, the performance of the algorithm could be precisely measured. In several of the cases LCS had a success rate of 100%, and in examples where the size of the search space prohibited the use of exhaustive search to decisively find the global optimas, LCS returned results that strongly indicated convergence into an optimal state.

The learning automata algorithm that LCS is derived from is a purely sequential algorithm, while LCS has been designed to work using parallel computing techniques. The introduction of local feedback ads more complexity to the algorithm, making it more computationally heavy. LCS mitigates this by dividing the extra workload between multiple threads, reducing the runtime compared to a sequential version.

The potency of GPU parallelization is also demonstrated using a random search algorithm. By utilizing the large amounts of threads that is available on a GPU, the probability to find a solution were improved by a factor of over 45'000.

This thesis shows empirically that the introduced LCS successfully finds global optima in vast search spaces utilizing parallel computing. This conclusively shows that LCS creates promise for solving optimization problems including hydropower production strategies.

5.1 Future Work

Several approaches can be taken to further strengthen and explore the possibilities of LA LCS. Two known necessities are mentioned in this section. The first is related to the parallel design choices, and the implications this presents in the current state. The other is related to further strengthen the results by comparing the LA LCS against other known algorithms that has shown good results when applied to similar problems within the domain.

5.1.1 Improved GPU parallelization

When executing large instances with regard to input and number of timesteps, the memory requirements increases. This is due to the nature of the algorithm where each block shares input parameters and stores shared variables in the shared memory on the GPU. On our reference card, the Nvidia K80, available shared memory for each block is limited to 49152 bytes. With the current implementation of LA LCS, the GPU is able to execute scenarios with $\leq$ 71 timesteps. A breakdown of the memory requirements are presented in Table 19 for a 71 and 72 timestep scenario. The breakdown reveals that the result struct is the biggest constraining element for higher GPU timestep counts. When increasing the number of timesteps to 72 the block uses more shared memory than available and crashes the simulation.

Description	Type	Type size	71 Timesteps	72 Timesteps
States	char	1 byte	1 byte	1 byte
Automatas	char	1 byte	213 bytes	216 bytes
Inflow array	float	4 bytes	852 bytes	864 bytes
Price array	float	4 bytes	285 bytes	288 bytes
Result struct	char	1 bytes	46655 bytes	47960 bytes
Result			**48006 bytes**	**49329 bytes**

Table 19: Bytes used of shared memory for 71 and 72 timesteps in bytes.

To combat this constraint, a redesign of the GPU implementation is necessary. Redesigning the implementation would allow for execution of a higher number of timesteps, while freeing the algorithm of the constraints presented by the max allowable number of threads in a block. In the current design each instance of the algorithm is contained within a block. Each thread is running an individual automaton, meaning that the number of automatons for a 71

timestep scenario is $71 \cdot 3 = 213$, a practical limit of ~ 340 timesteps is therefore imposed on the algorithm in its current design, due to using all available threads in the block $340 \cdot 3 = 1024$. Another side effect of dividing instances down in blocks is the impact on performance, this is due to utilizing a low thread count within the block for timesteps < 340, meaning several threads within this block are idling, when they could provide computing power for maximum GPU utilization.

The new design should avoid dividing separate runs into individual blocks, and use a more free floating design pattern, where all threads are free to be utilized.

5.1.2 Further verification and testing

The algorithm should be verified with known, successful approaches like the earlier presented Invasive Weed Optimization algorithm (IWO)[11] or the Particle Swarm Optimization algorithm (PSO) [4]. The IWO algorithm has attached pseudo code, and has a fairly simple design, making it an ideal candidate that could be implemented and tested against our environment. This should produce interesting material for verification and comparison of the algorithms.

References

[1] TW Archibald, KIM McKinnon, and LC Thomas. "An aggregate stochastic dynamic programming model of multireservoir systems". In: *Water Resources Research* 33.2 (1997), pp. 333–340.

[2] E Arnold, P Tatjewski, and Wo. "Two methods for large-scale nonlinear optimization and their comparison on a case study of hydropower optimization". In: ().

[3] Mohammad Azizipour et al. "Optimal operation of hydropower reservoir systems using weed optimization algorithm". In: *Water Resources Management* 30.11 (2016), pp. 3995–4009.

[4] Alexandre M Baltar and Darrell G Fontane. "Use of multiobjective particle swarm optimization in water resources management". In: *Journal of water resources planning and management* 134.3 (2008), pp. 257–265.

[5] RICHARD Bellman. "Dynamic programming". In: *Princeton, USA: Princeton University Press* 1.2 (1957), p. 3.

[6] Richard Bellman and David Blackwell. "Some two-person games involving bluffing". In: *Proceedings of the National Academy of Sciences* 35.10 (1949), pp. 600–605.

[7] Jacques F Benders. "Partitioning procedures for solving mixed-variables programming problems". In: *Numerische mathematik* 4.1 (1962), pp. 238–252.

[8] Philip D Crawley and Graeme C Dandy. "Optimal operation of multiple-reservoir system". In: *Journal of Water Resources Planning and Management* 119.1 (1993), pp. 1–17.

[9] MV Devisree and PT Nowshaja. "Optimisation of Reservoir Operations Using Genetic Algorithms". In: *IJSER* 5 (7 2014).

[10] William Gale, Sumit Das, and Clement T. Yu. "Improvements to an algorithm for equipartitioning". In: *IEEE Transactions on Computers* 39.5 (1990), pp. 706–710.

[11] "GPU-based variation of parallel invasive weed optimization algorithm for 1000D functions". In:

[12] Ole-Christoffer Granmo and Noureddine Bouhmala. "Solving the Satisfiability Problem Using Finite Learning Automata." In: *IJCSA* 4.3 (2007), pp. 15–29.

[13] Ole-Christoffer Granmo and Sondre Glimsdal. "Accelerated Bayesian learning for decentralized two-armed bandit based decision making with applications to the Goore game". In: *Applied intelligence* 38.4 (2013), pp. 479–488.

[14] A Helseth, B Mo, and G Warland. "Long-term scheduling of hydrothermal power systems using scenario fans". In: *Energy Systems* 1.4 (2010), pp. 377–391.

[15] Karla L Hoffman, Manfred Padberg, and Giovanni Rinaldi. "Traveling salesman problem". In: *Encyclopedia of operations research and management science*. Springer, 2013, pp. 1573–1578.

[16] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[17] Steven Johnson. *Emergence: The connected lives of ants, brains, cities, and software*. Simon and Schuster, 2002.

[18] D Nagesh Kumar and Falguni Baliarsingh. "Folded dynamic programming for optimal operation of multireservoir system". In: *Water Resources Management* 17.5 (2003), pp. 337–353.

[19] Sheng-li Liao et al. "Long-Term Generation Scheduling of Hydropower System Using Multi-Core Parallelization of Particle Swarm Optimization". In: *Water Resources Management* (2017), pp. 1–17.

[20] Luciana Martinez and Secundino Soares. "Primal and dual stochastic dynamic programming in long term hydrothermal scheduling". In: *Power Systems Conference and Exposition, 2004. IEEE PES*. IEEE. 2004, pp. 1283–1288.

[21] Kumpati S Narendra and Mandayam AL Thathachar. *Learning automata: an introduction*. Courier Corporation, 2012.

[22] B. John Oommen and EV de St Croix. "Graph partitioning using learning automata". In: *IEEE Transactions on Computers* 45.2 (1996), pp. 195–208.

[23] B John Oommen, Ole-Christoffer Granmo, and Asle Pedersen. "Using stochastic AI techniques to achieve unbounded resolution in finite player Goore Games and its applications". In: *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*. IEEE. 2007, pp. 161–167.

[24] B. John Oommen and Daniel C. Y. Ma. “Deterministic learning automata solutions to the equipartitioning problem”. In: *IEEE Transactions on Computers* 37.1 (1988), pp. 2–13.

[25] Leila Ostadrahimi, Miguel A Mariño, and Abbas Afshar. “Multi-reservoir operation rules: multi-swarm PSO-based optimization approach”. In: *Water resources management* 26.2 (2012), pp. 407–427.

[26] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.

[27] Mario VF Pereira and Leontina MVG Pinto. “Multi-stage stochastic optimization applied to energy planning”. In: *Mathematical programming* 52.1-3 (1991), pp. 359–375.

[28] MVF Pereira. “Optimal stochastic operations scheduling of large hydroelectric systems”. In: *International Journal of Electrical Power & Energy Systems* 11.3 (1989), pp. 161–169.

[29] MVF Pereira and LMVG Pinto. “Stochastic optimization of a multireservoir hydroelectric system: a decomposition approach”. In: *Water resources research* 21.6 (1985), pp. 779–792.

[30] Petr Pospichal, Jiri Jaros, and Josef Schwarz. “Parallel genetic algorithm on the cuda architecture”. In: *Applications of Evolutionary Computation* (2010), pp. 442–451.

[31] Roy Radner. “Team decision problems”. In: *The Annals of Mathematical Statistics* 33.3 (1962), pp. 857–881.

[32] Mark Silberstein. “GPUs: High-performance Accelerators for Parallel Applications: The multicore transformation (Ubiquity symposium)”. In: *Ubiquity* 2014.August (2014), p. 1.

[33] Minimum Spanning Tree. “Minimum Spanning Tree”. In: (2007).

[34] Torry Tufteland, Guro Ødesneltvedt, and Morten Goodwin. “Optimizing PolyACO Training with GPU-Based Parallelization”. In: *International Conference on Swarm Intelligence*. Springer. 2016, pp. 233–240.

[35] Tony Wauters et al. “Boosting metaheuristic search using reinforcement learning”. In: *Hybrid Metaheuristics*. Springer, 2013, pp. 433–452.