Algorithms for Computing a Tarski Fixed Point on a Finite Lattice, Equilibria in Submodular Games and Games with Strategic Complementarities

Minsuan Teh

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

In 2012, Dang, Qi, and Ye presented an algorithm for finding a fixed point in a d-dimensional lattice of width N that requires $O(log^d N)$ queries. Fearnley, Palvolgyi, and Savani published a method in 2021 that uses $O(log^{2\lceil d/3 \rceil} N)$ queries to discover a Tarski fixed point in the lattice. This project's goal is to utilise Python to implement and analyse both algorithms, as well as a value iteration algorithm. This report goes into the details of how the algorithms function, as well as the implementation details for each algorithm. Comparisons between the algorithms are made in terms of number of queries and time needed to find a Tarski fixed point. I show that when the width of lattice is large, the algorithm by Fearnley, Palvolgyi, and Savani does use the least queries to find a Tarski fixed point. When comparing algorithms in terms of time required, the algorithm may require more time than the other 2 algorithms. Details on how large the width must be for the algorithm to require the least queries are included. Each algorithm has its own advantages and disadvantages, which are explored in detail in the study.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Minsuan Teh*)

Acknowledgements

I received a lot of help and support from my supervisor while working on this project. Therefore, I would like to acknowledge my supervisor, Professor Kousha Etessami for providing the tools and answers that I needed to successfully complete my project.

Table of Contents

Chapter 1

Introduction

Submodular games, sometimes known as supermodular games, are those in which strategic complementarities are present. If the marginal value of one player's action is increasing in the actions of the other players, the game is supermodular. This means that when a player takes a higher action, the other players want to do the same. This kind of response can be represented by a monotone function. Tarski's fixed-point theorem states that every monotone function on a complete lattice has at least a fixed point. Computing a fixed point of the monotone function is equivalent to finding a Nash equilibria of the supermodular games, as proven in [2]. Therefore, the algorithms of finding such a fixed point are much interested in this project.

1.1 Prior Work

A Tarski instance is defined as finding a fixed point of a monotone function over a complete lattice. There are many works in finding an algorithm to find such a fixed point. In 2012, Dang, Qi and Ye published a paper that mentioned an algorithm for computing a Tarski fixed point under $\Omega(log^d N)$ queries [1] in a d-dimensional Tarski instance of width N. Then, Etessami, Papadimitriou, Rubinstein, and Yannakakis further proved that the algorithm is optimal in a two-dimensional lattice [2]. They also showed that finding a Tarski fixed point is essentially the same as computing an equilibrium for a supermodular game with Euclidean grid strategy spaces. Later work of Fearnley, Palvolgyi and Savani provided a faster algorithm to compute a Tarski fixed point in three dimensions [3]. For higher-dimensional instances, they proved that the algorithm manages to find a Tarski fixed point using $O(log^{2\lceil d/3 \rceil} N)$ queries by splitting the d-dimensional instance into multiple instances of at most 3 dimensions.

1.2 Main Results

Tarski's value iteration algorithm is a simple algortihm to compute a fixed point in a complete lattice. At most Nd queries are required for this algorithm to compute a fixed point in a d-dimensional lattice of width N. On paper, this algorithm requires more queries than the Dang et al. algorithm when N is small relative to d. In this paper, I show that how small N must be for that to occur.

The number of queries required by Tarski's value iteration algorithm is also affected by the degree of convergence of the monotone function. Given a monotone function that maps a point to the nearest fixed point on the function, Tarski's value iteration algorithm only needs 2 queries to find a fixed point of the function. On the other hand, the Dang et al. algorithm requires more queries than that. This shows that Dang et al. algorithm is not always faster than Tarski's value iteration algorithm even if N is big relative to d.

The Fearnley et al. approach requires the algorithm to keep track of all prior points queried. My results show that the time needed by the algorithm is greatly affected by maintaining and utilising this set of points. When increasing dimension of the instance by one from a multiple of 3, the algorithm requires a significantly more time because of reasons that will be mentioned.

Chapter 2

Background, Definitions, and The Algorithms

2.1 Formal Definition of Related Concepts

In this paper, I use a finite discrete Euclidean grid as my underlying lattice L. The lattice L is an integer grid $[N]^d$, for some positive integers N,d where $[N] = \{1,...,N\}$. The function over L used in this paper is always a monotone function.

Definition 2.1. $+,-$ between 2 points, a,b, are defined as the component-wise addition and subtraction of a and b. Given that $a,b \in [N]^d$ for some positive integers N,d, $c = a+b$ means that $c_i = a_i + b_i$ and $c = a-b$ means that $c_i = a_i - b_i$ for all i from 1 to d.

Definition 2.2. $\preceq, \succeq$ between 2 points, a,b, are defined as follows. Given that $a,b \in [N]^d$ for some positive integers N,d, $a \preceq b$ means that for all i from 1 to d, $a_i \leq b_i$. $a \succeq b$ means that for all i from 1 to d, $a_i \geq b_i$. $\prec, \succ$ are defined similarly.

Definition 2.3. A partial ordered set $(L, \preceq)$ is a complete lattice if there exists two operations, *meet* $\vee$ and *join* $\wedge$ on any pair of elements x, y of L such that $x,y \preceq x \vee y$ *and* $x \wedge y \preceq x,y$.

Definition 2.4. A function f on a lattice $(L, \preceq)$ is monotonic or order preserving if $a \preceq b$ implies $f(a) \preceq f(b)$.

Definition 2.5. Solving a Tarski instance is defined as follows. Given a complete lattice L and a monotone function f, find a point $x \in L$ such that $f(x) = x$.

Definition 2.6. For a complete lattice $L = [N]^d$, let $L(a,b) = \{x \in [N]^d \mid a \preceq x \preceq b\}$ where $a,b \in [N]^d$ denotes the sublattice between a and b.

Definition 2.7. An up set $Up(f)$ and down set $Down(f)$ of a function f over a lattice L is defined as follows. $Up(f) = \{x \in L : x \preceq f(x)\}$. $Down(f) = \{x \in L : f(x) \preceq x\}$.

Definition 2.8. A *slice* s of a lattice L is defined as follows. Given that $s = (s_1, s_2, ..., s_d)$ where $s_i \in [N] \cup \{*\}$, if $s_i \neq *$, then the dimension i of L is fixed to be a number in $[N]$. Otherwise, the dimension can be any number. If there is exactly one dimension fixed in slice s, then L_s is called as a principle slice of lattice L. The monotone function over L_s is denoted by f_s.

Definition 2.9. A pair of points (a,u) with $a,u \in L_s$ of 3 dimensions is an up set witness if:

- $a_t \geq f(a)_t$ and $u_t \geq f(u)_t$ where t is the dimension that is fixed in L_s
- $a_i = u_i, u_j \geq a_j, u_j \geq f(u)_j$ and $f(a)_j \geq a_j$ where $i \neq j$ and i,j are the other 2 dimensions in L_s.

Definition 2.10. A pair of points (d,b) with $d,b \in L_s$ of 3 dimensions is a down set witness if:

- $d_t \leq f(d)_t$ and $b_t \leq f(b)_t$ where t is the dimension that is fixed in L_s
- $d_i = b_i, d_j \leq b_j, d_j \leq f(d)_j and f(b)_j \leq b_j$ where $i \neq j$ and i,j are the other 2 dimensions in L_s.

2.2 Tarski's Value Iteration Algorithm

A simple algorithm to compute a Tarski fixed point is given by Tarski's value iteration. To compute a fixed point of f in a d-dimensional lattice with width N, the algorithm starts from the lowest point of the lattice, $x = 1^d$, a point which has all its coordinates as 1. First, the algorithm applies f to the point. If $f(x)$ turns out to be a fixed point, then the algorithm returns the point as the solution. If not, the algorithm applies f again to $f(x)$ and continue until a fixed point is found. It should be noted that any fixed point found in this manner will be the least fixed point of f, and the biggest fixed point can be determined by starting from the highest point of the lattice, N^d.

In the worst case of this algorithm, the fixed point can only be found after applying f $(N-1)d$ times. Therefore, this algorithm has a query complexity of $O(Nd)$. The algorithm requires the most queries to compute a Tarski fixed point when N is big relative to d. However, unlike the other algorithms, we can compute either the least or greatest fixed point using this algorithm. The least and greatest fixed point are proved to have some unique properties. For example, the least fixed point is used in denotational semantics to extract a related mathematical function, known as the semantics, from a particular programme text.

2.3 Algorithm by Dang, Qi and Ye

This algorithm is a recursive binary search algorithm where a d-dimensional instance is solved by performing $log\ N$ recursive calls on $d-1$-dimensional sub-instances. In this paper, I use 1^d as the least element of a lattice and N^d as the greatest element of a lattice.

2.3.1 One-dimensional Lattice

Suppose that we are given a monotone function $f : L(a,b) \rightarrow L(a,b)$, where $a, b \in [N]^d$. Let $m = \lfloor \frac{a+b}{2} \rfloor$ and apply f on m. If $f(m) = m$, then the algorithm has found a fixed point. If $f(m) \succ m$, then we know that $f : L(m,b) \rightarrow L(m,b)$ is monotone and has a fixed point. Similarly, if $f(m) \prec m$, then we know that $f : L(a,m) \rightarrow L(a,m)$ is monotone and has a fixed point. This is repeated until we found a fixed point in L. In the worst case, the fixed point can only be found after $log_2 N$ iterations.

2.3.2 Higher Dimension

The procedure for finding a fixed point in a lattice $L(1^d, N^d)$ of dimension d where $d > 1$ is as follows.

1. We first define a function $g_{d-1,y}(x) : [N]^{d-1} \rightarrow [N]^{d-1}$ such that, for $x \in [N]^{d-1}$ and $y = \lfloor \frac{N+1}{2} \rfloor$, $g_{d-1,y}(x) = (f(x,y)_1, f(x,y)_2, ..., f(x,y)_{d-1})$. Since f is a monotone function, g is also a monotone function.

2. Compute a fixed point of $g_{d-1,y}(x)$ recursively and form $g_{d-2,y'}(x') : [N]^{d-2} \rightarrow [N]^{d-2}$ where $y' = (\lfloor \frac{N+1}{2} \rfloor, y)$ and $x' = (x_1, x_2, ..., x_{d-2})$. This is repeated until $g_{1,m}$ where $m = \lfloor \frac{N+1}{2} \rfloor^{d-1}$in which a fixed point $x^* = g_{1,m}(x^*)$ can be found easily.

3. The algorithm returns from recursion and checks the value of $g_{2,m'}((x^*, m_1))_2$ where $m' = (m_2, m_3, ..., m_{d-1})$.

   (a) $g_{2,m'}(x^*, m_1)_2 = m_1$: The algorithm returns from recursion and repeat the current step with $g_{3,m'}((x^*, m_2))_3$. But the new x^* is now (x^*, m_1) and the new m' is $(m_3, m_4, ..., m_{d-1})$.

   (b) $g_{2,m'}(x^*, m_1)_2 \prec m_1$: The current step is repeated with $g_{2,m'}((x^*, x'))_2$ where $x' = \lfloor \frac{1+g_{2,m'}(x^*,m_1)_2}{2} \rfloor$.

   (c) $g_{2,m'}(x^*, m_1)_2 \succ m_1$: The current step is repeated with $g_{2,m'}((x^*, x'))_2$ where $x' = \lfloor \frac{N+g_{2,m'}(x^*,m_1)_2}{2} \rfloor$.

4. When the algorithm has fully returned from the recursion, the algorithm proceeds with the same procedure as step 3 with $f(x^*, y)_d$.

   (a) $f(x^*, y)_d = y$: The algorithm proceeds to the next step.

   (b) $f(x^*, y)_d \prec y$: The current step is repeated with $f(x^*, y')_d$ where $y' = \lfloor \frac{1+f(x^*,y)_d}{2} \rfloor$

   (c) $f(x^*, y)_d \succ y$: The current step is repeated with $f(x^*, y')_d$ where $y' = \lfloor \frac{N+f(x^*,y)_d}{2} \rfloor$

5. At this last step, if $f(x^*, y) = (x^*, y)$, the algorithm returns (x^*, y) as the solution. If $f(x^*, y) \prec (x^*, y)$, the algorithm repeats all the steps from step 1 with the lattice $L(1^d, f(x^*, y))$. If $f(x^*, y) \succ (x^*, y)$, the algorithm repeats all the steps from step 1 with the lattice $L(f(x^*, y), N^d)$.

Step 5 will always return (x^*, y) as the solution if coordinates in each dimension of f do not depend on coordinates in higher dimension. If each iteration of the algorithm in step 3 has found $x = (x_1^*, x_2^*, ..., x_k^*)$ for some integer $k \leq d-1$, then $x_i = f(x,y)_i$ for all i from 1 to k where $y \in [N]^{d-k}$. An example of such monotone function $f : L \rightarrow L$ would be: $f(x)_d = max(x_d, p_d)$ where $p \in [N]^d$. A counter example would be: $g : L \rightarrow L$, $g(x)_d = max(x_i, x_j, p_d)$ where $i, j \in \{1, 2, ..., d\}$, $i > j$ and $i = 1$ if $j = d$. If coordinates in each dimension of f do not depedn on coordinates in higher dimension. Step 5 does not require the algorithm to do a rerun in smaller lattice if the algorithm finds x^* in step 3 from higher to lower dimension instead from lower to higher dimension.

Query complexity: The query complexity of this algorithm is $O(log^d\, N)$. It is worth noting that as proven in [1], this algorithm is indeed optimal in 2-dimensional space. But as shown in [2], there is a algorithm that has a smaller query complexity if $d > 2$.

2.4 Algorithm by Fearnley, Palvogyi and Savani

The algorithm has two parts, an outer algorithm and an inner algorithm. The outer algorithm is used to solve a 3-dimensional Tarski instance while the inner algorithm is called by the outer algorithm to solve a slice of the instance. In higher dimensions, the algorithm decomposes the lattice L of the Tarski instance into multiple lattices of at most 3 dimensions. Then, the algorithm recursively solves each lattice whenever the algorithm queries the monotone function over L. Details will be explained in the latter section. In this paper, all the functions used are assumed to be monotone. Therefore, steps required for checking violation of order preservation are mostly ignored. Also, the reasoning behind each step in section 2.4 is not mentioned because of the following two reasons. First, the goal of this project is to analyze, not explain, each algorithm. Second, detailed explanation of the algorithm can be read in [3].

2.4.1 The Outer Algorithm

Input: A 3-dimensional lattice L and two points $a, b \in L$

Output: A fixed point in L

The outer invariant: At the start of each iteration, it will make sure that $a \in Up(f)$ and $b \in Down(f)$ before proceeding.

The algorithm searches for a fixed point between a and b. Initially, $a = 1^d$ and $b = N^d$ are set as the lowest point and the highest point of L. On each iteration of the algorithm, a principle slice s will be defined. The dimension that will be fixed is determined by choosing the dimension i that maximizes $b_i - a_i$. After that, dimension i will be fixed as $\lfloor \frac{b_i + a_i}{2} \rfloor$.

The algorithm then calls the inner algorithm with the arguments $L(a,b), s$. The inner algorithm returns a point x such that $x \in Up(f)$ or $x \in Down(f)$. If $x \in Up(f)$, The outer algorithm sets $a = x$ and moves to the next iteration. If $x \in Down(f)$, The outer algorithm sets $b = x$ and moves to the next iteration.

Termination: The algorithm terminates if and only if for all i, $b_i - a_i < 2$. At this point, we can use Tarski's value iteration to find a fixed point in the smaller lattice $L(a,b)$ which will take at most 3 queries.

2.4.2 The Inner Algorithm

Input: A lattice $L(a,b)$ where $a,b \in L$ and a principle slice s

Output: A point x such that $x \in Up(f)$ or $x \in Down(f)$

The inner invariant: The followings have to be satisfied before proceeding at the start of each iteration.

- Either $a \in Up(f_s)$ or there is an up set witness (a,u) with $u \preceq b$
- Either $b \in Down(f_x)$ or there is a down set witness d,b with $a \preceq d$
- If we have both up set and down set witness, the algorithm requires $u \preceq d$

Throughout the description, I use the third dimension as the dimension that is fixed by the outer algorithm. For a down set witness (d,b), if $d_2 = b_2$, then it is called as a top boundary down set witness. If $d_1 = b_1$, then it is called as a right boundary down set witness. Similarly for an up set witness (a,u), if $a_2 = u_2$, then it is called as a bottom boundary up set witness. If $a_1 = u_1$, then it is called as a left boundary up set witness.

Five points are defined at the start of each iteration, namely mid, top, bot, $left$ and $right$.

- $mid = (\lfloor \frac{a_1+b_1}{2} \rfloor, \lfloor \frac{a_2+b_2}{2} \rfloor, s_3)$
- $top = (\lfloor \frac{a_1+b_1}{2} \rfloor, b_2, s_3)$
- $bot = (\lfloor \frac{a_1+b_1}{2} \rfloor, a_2, s_3)$
- $left = (a_1, \lfloor \frac{a_2+b_2}{2} \rfloor, s_3)$
- $right = (b_1, \lfloor \frac{a_2+b_2}{2} \rfloor, s_3)$

Step 1: The algorithm assumes that it has found a valid down set witness (d,b) or up set witness (a,u) in previous iteration. If not, step 1 is skipped.

1. If (d,b) is a top boundary down set witness and $d \prec top$:
   (a) The algorithm checks if (d, top) is a valid down set witness. If yes, it moves to the next iteration with $L(a, top)$.
   (b) The algorithm checks if (top, b) is a valid down set witness. If yes, it replaces d with top.
2. If (d,b) is a right boundary down set witness and $d \prec right$, the algorithm uses the same procedure as Step 1.1 with top being replaced by $right$.
3. If (a,u) is a bottom boundary up set witness and $bot \prec u$:
   (a) The algorithm checks if (bot, u) is a valid up set witness. If yes, it moves to the next iteration with $L(bot, b)$.

   (b) The algorithm checks if (a, bot) is a valid up set witness. If yes, it replaces u with bot.

4. If (a, u) is a left boundary up set witness and $left \prec u$, the algorithm uses the same procedure as Step 1.3 with bot being replaced by $left$.

Step 2: This step will be applied if step 1 did not move the algorithm to the next iteration.

1. (a) If $mid \in Up(f_s)$, the algorithm moves to the next iteration with $L(mid, b)$.

   (b) If $mid \in Down(f_s)$, the algorithm moves to the next iteration with $L(a, mid)$.

2. (a) The algorithm checks if d is an element of $Up(f)$ or $Down(f)$. If yes, the algorithm terminates and returns d to the outer algorithm.

   (b) The algorithm checks if u is an element of $Up(f)$ or $Down(f)$. If yes, the algorithm terminates and returns u to the outer algorithm.

3. (a) If $(mid, right)$ is a valid down set witness, then the algorithm moves to the next iteration with $L(a, right)$ and $d = mid$.

   (b) If (mid, top) is a valid down set witness, then the algorithm moves to the next iteration with $L(a, top)$ and $d = mid$.

   (c) If (bot, mid) is a valid up set witness, then the algorithm moves to the next iteration with $L(bot, b)$ and $u = mid$.

   (d) If $(left, mid)$ is a valid up set witness, then the algorithm moves to the next iteration with $L(left, b)$ and $u = mid$.

Special case: There are 2 special cases that require different procedures than the above steps before termination happens. The first case is a width-one instance, $b_i = a_i + 1 \textit{ and } b_j > a_j + 1$ where $i, j \in \{1, 2\} \textit{ and } i \neq j$. The second case is a width-zero instance, $b_i = a_i \textit{ and } b_j > a_j + 1$ where $i, j \in \{1, 2\} \textit{ and } i \neq j$. Throughout the description, I use $i = 1 \textit{ and } j = 2$. The case for $i = 2 \textit{ and } j = 1$ is symmetric.

1. A special step is applied when the algorithm first encounters either of the instances.

   (a) If the algorithm has found a valid bottom boundary up set witness (a, u) on previous iteration:

      i. If $a \in Down(f)$, then a is returned to the outer algorithm.

      ii. If $u \in Down(f)$, then u is returned to the outer algorithm.

      iii. If $u \in Up(f_s)$, then a is replaced by u.

      iv. If $a \in Up(f_s)$, then u is deleted.

   (b) If the algorithm has found a valid top boundary down set witness (d, b) on previous iteration:

      i. If $b \in Up(f)$, then b is returned to the outer algorithm.

ii. If $d \in Up(f)$, then d is returned to the outer algorithm.

iii. If $d \in Down(f_s)$, then b is replaced by d.

iv. If $b \in Down(f_s)$, then d is deleted.

2. (a) Width-one instance
The algorithm uses two separate runs of steps 1 and 2. However, in the second run, the algorithm changes the definition of $mid, top, bot, left$ and $right$ to use round up instead of round down. There is a chance that both runs do not make any progress. This happens when step 2.3.c or step 2.3.d are triggered in the first run and step 2.3.a or step 2.3.b are triggered in the second run. In this case, the algorithm checks if first run of $mid \in Down(f)$. If yes, then mid is a valid answer that can be returned to the outer algorithm. Note that any steps that involve u, d are not applied. This is because the special step has removed any valid up set witness or down set witness.

(b) Width-zero instance
The algorithm proceeds through steps 1 and 2 as normal. But with the same reasoning as width-one instance, any steps that involve u, d are not applied. Also, step 2.3 is replaced with a single check: the algorithm checks if $mid \in Down(f)$. If yes, then mid is returned to the outer algorithm.

Termination: The inner algorithm terminates when $b_1 - a_1 < 2$ *and* $b_2 - a_2 < 2$. In this case, $L(a,b)$ contains at most four points. Therefore, the algorithm checks all of the points and return the answer that must exist to the outer algorithm.

Query complexity: At each iteration of the outer algorithm, the size of $L(a,b)$ is reduced in half. Therefore, the outer algorithm will run for at most $O(log\ N)$ iterations. On the other hand, the inner algorithm queries at most five points at each iteration (a, b, mid, one of $\{u, d\}$ and one of $\{top, bot, left, right\}$). If the inner algorithm runs on $L(a,b)$, then the query complexity of the inner algorithm is $O(log\ N_1 + log\ N_2 + log\ N)$ where $N_1 = b_1 - a_1$ and $N_2 = b_2 - a_2$. Hence, the query complexity of the whole algorithm is $O(log^2\ N)$.

2.4.3 Higher Dimension

Decomposing a d-dimensional lattice L is as follows. Let $A \in [N]^a$ be a a-dimensional lattice and $B \in [N]^b$ be a b-dimensional lattice. Given that $a + b = d$, we have the lattice $L = A \times B$ where $L \in [N_1, N_2, ...N_a, N_{a+1}, ...N_{a+b}] = [N]^d$. To find a fixed point in L if $d > 3$, the lattice is decomposed into a product of lattices $L = L_1 \times L_2 \times ... \times L_{\lceil d/3 \rceil}$. Each lattice L_i has dimension of three except for the last lattice $L_{\lceil d/3 \rceil}$ which can have $k \in \{1, 2, 3\}$ dimensions. Given that $x \in A$ and $y \in B$, $(x, y) = (x_1, x_2, ..., x_a, y_1, y_2, ..., y_b)$ is used to denote a point in L.

The algorithm: L is decomposed into 2 lattices A and B of dimension 3 and $d - 3$ respectively. Let H be an algorithm that can find a fixed point in L, H_A be an algorithm that can find a fixed point in A and H_B be an algorithm that can find a fixed point in B. If x is a fixed point returned by H_A and y is a fixed point returned by H_B, then (x, y) is a fixed point that can be returned by H, proven in [3].

First, H calls on H_A to find a fixed point in A. Whenever H_A queries a point $x \in A$, a function $f_A(x)_i = f((x,y^*))_i$ for $i \leq a$, $f_A : [N]^a \to [N]^a$, $y^* \in B$ is a fixed point, is defined. Since f_A relies on a fixed point, y^*, on B to answer the query, H_B is called. H_B find a fixed point on B which is defined as a slice $s_x = (x_1, x_2, ..., x_a, *_1, *_2, ... *_b)$ of L. Therefore, any queries of y on B is defined as $f((x,y))_i = x_i$ if $i \leq a$ and $f((x,y))_i = f((x,y))_i$ otherwise. To maintain the correctness of the algorithm, the search of a fixed point on B is bounded by two points l, u. Initially, the lower bound l and the upper bound u of B are set as the lowest point and highest point of B. Before each call of H_B, the value of l, u are determined with the following procedures:

1. Each time H_A queries $x \in A$ and H_B responds with $y \in B$, $(x,y) \in L$ is added to set P.

2. H finds a set $D = \{y' \mid There\ exists\ an\ x'\ such\ that\ (x',y') \in P\ and\ x' \preceq x\}$. Then, it chooses l as the upper bound of D.

3. H finds a set $U = \{y' \mid There\ exists\ an\ x'\ such\ that\ (x',y') \in P\ and\ x \preceq x'\}$. Then, it chooses u as the lower bound of U.

After finding a fixed point $y^* \in B_{l,u}$, H_A can then use f_A to query a point $x \in A$. If x^* is the fixed point found by H_A and y^* is the associated fixed point in $B_{l,u}$, then (x^*, y^*) is returned by H. If $d-3 > 3$, the $(d-3)$-dimensional lattice B is decomposed into 2 smaller lattices of dimension 3 and $d-6$. This is repeated until all lattices have at most 3 dimensions so that the Fearnley et al. algorithm can be used. If the last lattice has dimension of one or two, the Dang et al. algorithm is used instead to solve the last lattice.

Query complexity: Since L is decomposed to $\lceil d/3 \rceil$ lattices and each lattice needs $O(log^2 N)$ queries, the algorithm needs $O(log^{2\lceil d/3 \rceil} N)$ queries.

Chapter 3

Implementation of The Algorithms

Python is used to implement the algorithms in this study. The reason for this is that Python is simple enough for those who aren't experts in coding to understand, even if they don't have much expertise. The use of C/C++ will almost certainly improve the speed with which the algorithms are executed. But if all algorithms have a shorter runtime, the difference in runtime between the algorithms do not change by much. The algorithm is also compared based on the number of queries to the function. As a result, the choice of programming language used in this paper doesn't actually matter.

3.1 Tarski Instance

Tarski Instance is implemented in `tarski.py` (A.1). The two basic components of a Tarski instance are a complete lattice and a monotone function. The lattice used in this paper is a discrete Euclidean grid. Therefore, using a list to represent a coordinate in the lattice is a simple approach to implement it. There are three types of monotone functions used to evaluate each algorithm. Define a monotone function $f : L \to L$ over a d-dimensional lattice $L \in [N]^d$ as the followings. For all the functions, $p^{(1)}, p^{(2)}, p^{(3)}, p^{(4)} \in L$, $x \in L$ and $i, j, k \in d$.

1. $f(x)_i = F(x_j, x_k, p_i^{(1)})$ where F is either a minimum or maximum function.
2. $f(x)_i = a_i x_j + b_i x_k + c_i$ where $a_i = \frac{p_i^{(1)}}{p_i^{(1)}+p_i^{(2)}+p_i^{(3)}}$, $b_i = \frac{p_i^{(2)}}{p_i^{(1)}+p_i^{(2)}+p_i^{(3)}}$, $c_i = \frac{p_i^{(3)} p_i^{(4)}}{p_i^{(1)}+p_i^{(2)}+p_i^{(3)}}$
3. $f(x)_i = a_i x_i + b_i$ where $a_i = \frac{p_i^{(1)}+1}{p_i^{(1)}+p_i^{(2)}+2}$, $b_i = \frac{p_i^{(3)}(p_i^{(2)}+1)}{p_i^{(1)}+p_i^{(2)}+2}$

The first type of function returns a discrete point if x and p are discrete. However, this is not the case for the second and third type of functions. Therefore, every point returned by the functions are rounded down. Because of this, any fixed point found by the algorithms in my implementation are an approximate to the actual fixed point of the functions. The approximation is very accurate as the difference between the approximated fixed point and the actual fixed point is at most 1 in any dimension. The functions are implemented by defining a class called `Monotone_function`. It has a single class function and 6 class variables. `math` and `random` libraries are used in the class.

```
import math
import random

class Monotone_function:
    def __init__(self, N, dimension, ftype) -> None:
        self.ftype = ftype
        self.dimension = dimension
        self.N = N
        self.random = []
        self.nquery = 0

    def output(self, x, raw=False):
```

`ftype` determines the type of monotone function to be used. `N` is an integer which represents the maximum value in a dimension of the function. `dimension` is an integer which represents the dimension of the function. `random` holds a list of random number that is used as coefficients or constants of the function. `nquery` is an integer that records the total number of queries that have been done by the function.

`output` takes a single point, `x`, in the lattice and returns another point according to the monotone function used. When `output` is called, it reads the class variable `ftype` to choose the type of function to use. Then, `output` returns a point, y which is computed using `x` and `random`. If $y_i < 1$ for any $i \in$ `dimension`, y_i is set as 1 before being returned by the function. The case for $y_i > N$ is similar. If `raw=True`, then the point returned by `output` will not be rounded down. This is only used after an algorithm returns a fixed point to make sure that the point is an approximate fixed point of the actual function.

At initialization of the class, `N`, `dimension` and `ftype` are needed as the class arguments. Then, the class creates a list of random number in `random` according to the type of function, `ftype`. `ftype` can be set as 4 values. 0 indicates the first type of function with F being a minimum function. 1 indicates the first type of function with F being a maximum function. 2 indicates the second type of function and 3 indicates the third type of function.

3.2 Tarski's Value Iteration Algorithm

This algorithm is implemented in `algorithms.py` (A.2).

```
from tarski import *

def tarski_iteration(instance, fromBottom=True):
```

The implementation of this algorithm is straight forward. Therefore, there is only a function responsible for the algorithm in `algorithms.py`, which is `tarski_iteration`. `tarski_iteration` requires an argument and optionally another argument. The first argument, `instance`, is a Tarski instance to find a fixed point in. The second argument, `fromBottom`, is a boolean value which indicates the starting point to search for a fixed point. The default is `True` which means that `tarski_iteration` will search for a fixed point from 1^d where d is the dimension of the Tarski instance. If `fromBottom=False`, the algorithm will search for a fixed point from N^d instead.

`tarski_iteration` contains a `for` loop. It loops for at most N^d times as N^d is the total number of coordinate in a lattice of width N and dimension d. But do note that the algorithm requires at most Nd iterations to find a fixed point. It contains a local variable, `current`, which records the current point to query on the monotone function. At each iteration, it checks if the point returned by the monotone function is a fixed point. If not, `current` is set as the returned point and the algorithm moves to the next iteration. This continues until a fixed point is found, which is then returned by `tarski_iteration`.

This algorithm will almost certainly performs better than the other 2 algorithms when first type of monotone function is used. This is because the first type of algorithms converges to a fixed point quickly. Therefore, to ensure a fair comparison between the algorithms, a boolean variable called `smallStep` is added to `Monotone_function`. When `smallStep=True`, any point y returned by querying a point x on the monotone function will have the property: $|y_i - x_i| \leq 1$ for all $i \in d$. Default value of `smallStep` is `False`.

```
class Monotone_function:
    def __init__(self, N, dimension, ftype) -> None:
        self.smallStep = False
```

3.3 Dang et al. Algorithm

This algorithm is implemented in `algorithms.py` (A.2).

```
from functions import *
from tarski import *

def dqy_algorithm(instance, x, y):
    global answer
    global min_index
    global max_index

def dqy_helper(instance, depth):
```

`dqy_algorithm` is the main function to be called while `dqy_helper` is a helper function for `dqy_algorithm`. `dqy_algorithm` requires 3 arguments: `instance`, `x` and `y`. It will return a fixed point in the lattice `instance` between `x` and `y`. `dqy_helper` requires 2 arguments: `instance`, `depth`. It returns a point that is either a fixed point, a point in up set or a point in down set of `instance`. There are three global variables in my implementation, `min_index`, `max_index` and `answer`. `answer` holds the point to be returned by `dqy_helper` whereas `min_index` and `max_index` represent `x` and `y` initially.

At each iteration of `dqy_algorithm`, `dqy_helper` is called with `depth=0`. Then, `answer` is returned by `dqy_helper`. If `answer` is a fixed point of f, then it is returned by the algorithm. If `answer` $\in Up(f)$, then `min_index` is set as `answer`. Otherwise, `max_index` is set as `answer`. After that, the algorithm moves to the next iteration.

At each iteration of `dqy_helper`, it creates a slice s that fixes all dimensions of `instance` except the `depth` dimension. Then, it queries the midpoint of `min_index` and `max_index`. If the point, y, is a fixed point of f_s, then it increments `depth` by 1 and moves to the next iteration. Otherwise, it sets `min_index` as y if $y \in Up(f_s)$ and `max_index` as y if $y \in Down(f_s)$.

3.4 Fearnley et al. Algorithm

This algorithm is implemented in `fps_algorithm.py` (A.3).

3.4.1 Three Dimensions

```
import math
from functions import *
from tarski import *

def outer(instance, x, y):

def inner(instance, principle_slice, principle_index, a, b, u, d,
                                      width1_secondrun, midone):
```

The outer part of the algorithm is implemented as `outer` and the inner part of the algorithm is implemented as `inner`. `outer` requires 3 arguments like `dqy_algorithm` while `inner` requires 9 arguments: `instance, principle_slice, principle_index, a, b, u, d, width1_secondrun, midone`. Like the other algorithms, `instance` is a Tarski instance. `outer` creates a principle slice of `instance` and passes it to `inner`. `principle_slice` is the value of the dimension that is fixed. `principle_index` is the dimension that is fixed. `a` and `b` are the lower bound and upper bound of the slice on `instance`. `u` and `d` are the up set witness and down set witness when paired with `a` and `b`. `width1_secondrun` is a boolean which is used to indicate that the step for second run of width-one instance should be applied. Lastly, `midone` is only used when both runs of width-one instance do not make progress.

At the start of `outer`, the algorithm makes sure that the outer invariant is satisfied. This is only done once because the point returned by `inner` will always be in $Up(f)$ or $Down(f)$. Then, the algorithm looks for the dimension that has the biggest $\mathtt{y}_i$ - $\mathtt{x}_i$. This dimension is stored as `principle_index` and the midpoint in this dimension is stored as `principle_slice`. `a` and `b` are set as $\mathtt{x}_s$ and $\mathtt{y}_s$ respectively where s is the principle slice set by the outer algorithm. `width1_secondrun` is set as False while `u`, `d` and `midone` are set as `[0,0,0]`. After setting up every arguments, `inner` is called. It will either returns a point which is either in $Up(f)$ or $Down(f)$. The next iteration of `outer` starts after updating the corresponding `x` or `y`. At the termination of `outer`, the algorithm uses `tarski_iteration` to find a fixed point in the remaining lattice.

Other than the inner invariant, `inner` checks if the termination condition has been reached at the start of its iteration. At the termination step, `inner` finds a solution in {`a,b,a1,a2`} to return where `a1,a2` are the only 2 points between `a,b` at this step. Also, the algorithm checks if steps for special case (mentioned in section 2.4.2) is needed. If width-zero instance or width-one instance is triggered, the special step is applied and step 2 is changed accordingly. Otherwise, it proceeds to create variables for mid, top, bot, $left$ and $right$. Then, step 1 and 2 of inner algorithm are applied. Note that the implementation of step 1 and 2 are based on [3] whereas the algorithm described in section 2.4 is a summary of the algorithm. Therefore, there is a difference in structure between the implementation of step 1 and 2 and section 2.4.2. Also, any of the points, `a, b, u, d, mid, top, bot, left, right`, is queried for at most 1 time within `inner`. The reason is to prevent unnecessary query to the monotone function and hence improve the accuracy of measuring the total number of queries done.

3.4.2 Higher Dimension

When a Tarski instance has dimension of more than three, the lattice is decomposed into multiple lattices which is then passed to `outer` to solve. To identify which dimension of the lattice to solve, 2 new variables that indicate the range of dimension to solve are needed. Whenever `outer` queries a point, the point is added to a set. Therefore, there should be a class variable to store the set. Note that this variable is a set of set. Lastly, a variable is needed to record the slice of L. Supposed that a lattice, L, of dimension 6 is split to 2 lattices, A and B, where A is the first 3 dimension of L and B is the last 3 dimensions of L. When `outer` tries to query a point in B on slice s_x where $x \in A$, s_x is needed to be known for the query. Therefore, a total of 4 new variables are needed in `Montone_function`.

```python
class Monotone_function:
    def __init__(self, N, dimension, ftype) -> None:
        self.start = 0
        self.end = dimension
        self.P = []
        self.queryPoint = []
```

`start` and `end` are added to both `outer` and `dyq_algorithm` because of this change. Reason for adding these 2 arguments to `dyq_algorithm` is that `outer` uses the algorithm if the last lattice after decomposing L has s dimension of 1 or 2.

```python
def dqy_algorithm(instance, start, end, x, y):

def outer(instance, x, y, start=0, end=3, first=False):
```

There are some changes to `output` as well. Usually, `output` uses `x` to return a point based on the monotone function. When `start` and `end` do not cover all dimensions of the lattice, it uses `queryPoint` instead. Before this, `queryPoint[start:end]` is replaced by `x`. If `end` is not the last dimension of the lattice, then `output` calls `outer` to solve the higher dimensional lattice which starts from dimension `end`. This continues until `end` is the last dimension of the lattice as the lattice cannot be further decomposed.

Before calling outer, the 2 arguments of the call, x and y, are chosen from the set, Monotone_function.P according to the step in section 2.4.3. Also, first is set to True in each call of outer from output. When first is set to True, the set which contains points of dimension (dimension - start) is deleted. Note that at each split of the lattice, a new set is created within Monotone_function.P. Each set within Monotone_function.P contains queried points on different slice of the lattice. Therefore, queried points on slice which is different than the current slice cannot be used to choose x and y. Also, all fixed points found by outer are added to the set after returning from the recursive call.

3.5 Helper Function and Testing

All helper functions are implemented in functions.py (A.4).

```
def below(x, x_prime):

def above(x, x_prime):

def compare(x, y, principle_index):

def up(x, fx, principle_index):

def down(x, fx, principle_index):

def invariant(x, y, fx, fy):

def inner_invariant(a, b, u, d, fa, fb, fu, fd, principle_index):

def down_witness(d, b, fd, fb, principle_index):

def up_witness(a, u, fa, fu, principle_index):

def last_case(instance, x):

def below_filter(setP, queryPoint, start, x, y):

def above_filter(setP, queryPoint, start, x, y):

def same(x,y):
```

Also, a simple python file called test.py (A.5) is provided to demonstrate ways to use the code.

```
import timeit
from tarski import *
from algorithms import *
from fps_algorithm import *

def do_the_tests(d, N, t, n, smallstep, algorithm):

if __name__ == "__main__":
```

Chapter 4

Experimental Evaluation of The Implemented Algorithms

In this paper, I compare the algorithms in term of number of queries each algorithm has done to the monotone function, and the time needed for each algorithm to find a fixed point. For this chapter, I use monotone function of type 1 and 2 mentioned in 3.1. Monotone function of type 3 is not used as it produces the same result as type 2 function.

4.1 Convergence of Function

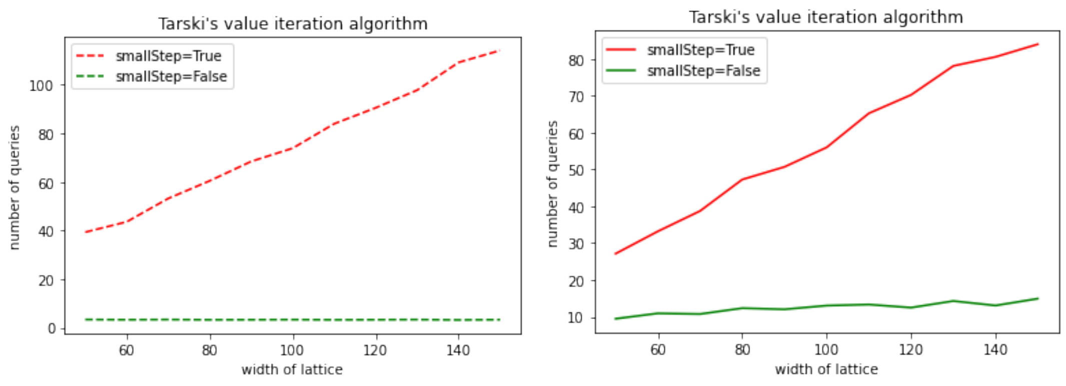

Figure 4.1: Tarski's value iteration algorithm using monotone function of type 1 on the left and type 2 on the right. Max function is used in type 1 function and `fromBottom=True`. Note that the algorithm requires only 1 query to solve any Tarski instance using monotone function of type 1 if `smallStep=True`.

In this section, I show that when `smallStep=False`, Tarski's value iteration algorithm and Dang et al. algorithm requires significantly less number of queries to solve a Tarski instance. On the other hand, Fearnley et al. algorithm requires lesser number of queries as well, but the difference is not huge. Throughout this section, the dimension of the monotone function is set as 3. 500 monotone functions are generated randomly and the average. However, there are still some fluctuations present in the graphs.

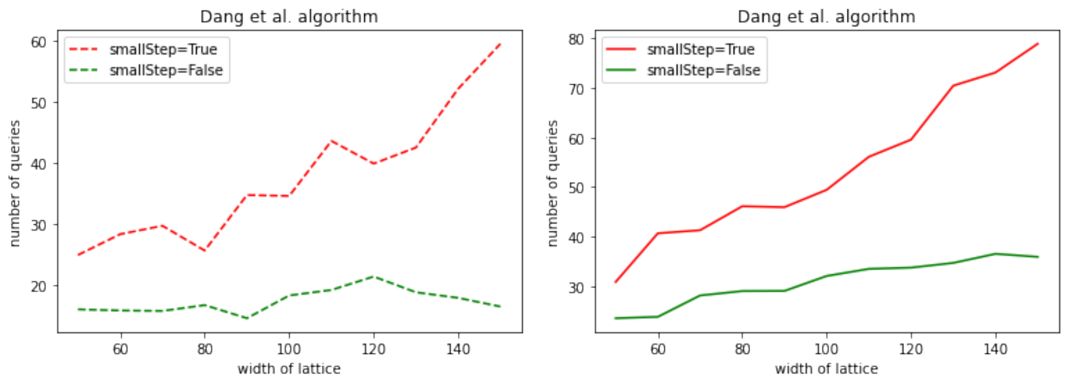

Figure 4.2: Dang et al. Algorithm using monotone function of type 1 on the left and type 2 on the right.

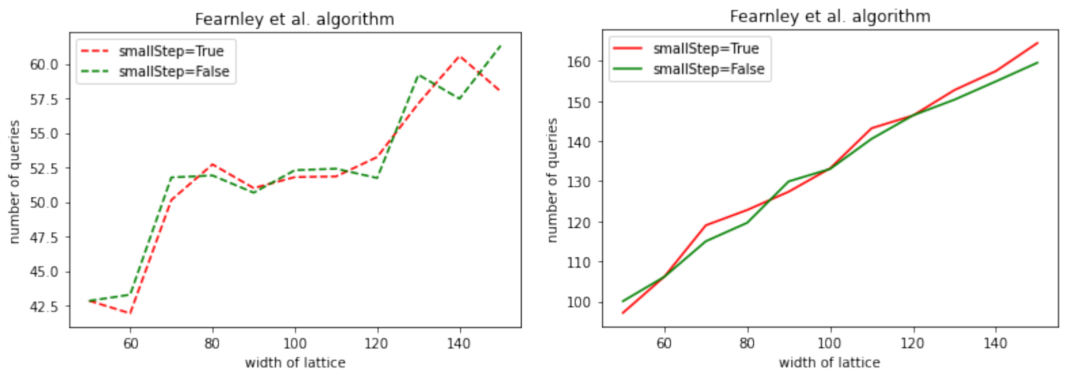

Figure 4.3: Fearnley et al. Algorithm using monotone function of type 1 on the left and type 2 on the right.

Figure 4.1 shows that Tarski's value iteration requires a significantly less queries if `smallStep=False`. This is because both type of function converges to a fixed point quickly. This is especially true when `fromBottom=True` and minimum function is used or `fromBottom=False` and maximum function is used.

Dang et al. algorithm is similar to Tarski's value iteration in this aspect. Recall that in step 3 of Dang et al. algorithm, the algorithm checks the value of $g(x^*x')_i$ for some $i \leq d-1$. Then, the algortihm uses $x' = \lfloor \frac{1+g(x^*x')_i}{2} \rfloor$ or $x' = \lfloor \frac{N+g(x^*x')_i}{2} \rfloor$ in the next iteration. Because `smallStep=True` restricts the difference between $g(x^*x')_i$ and x' to at most 1, the difference between x' in the next iteration and x' in the current iteration is smaller. Therefore, the algorithm now requires more iterations to find the fixed point of g. Step 4 is similar to step 3 as well.

As shown in Figure 4.3, Fearnley et al. algorithm requires similar amount of queries regardless of `smallStep`, unlike the other 2 algorithms. This is because the algorithm does not rely on the value of $f(x)$, where x is the current point of query and f is the monotone function, to find the next point to query. Instead, Fearnley et al. algorithm defines its own set of points: $mid, top, bot, left$ and *right* and uses them as the points to query. Therefore, to ensure a fair comparison between the algorithms, `smallStep` is set to `True` in all the latter sections.

4.2 Fixing Dimension and Increases Width

Fearnley et al. algorithm uses lesser queries than Dang et al. algorithm when the width of lattice is large. For lattice with dimension of 3, this happens when the width of the lattice is more than 300 in type 1 function. For type 2 function, the threshold is around 900. Dang et al. algorithm and Fearnley et al. algorithm requires lesser queries than Tarski's value iteration algorithm in both functions as the width of lattice increases. This is expected as shown by their query complexities. Also, Fearnley et al. algorithm requires the most queries and Tarski's value iteration requires the least queries when the width of lattice is small when using type 2 function.

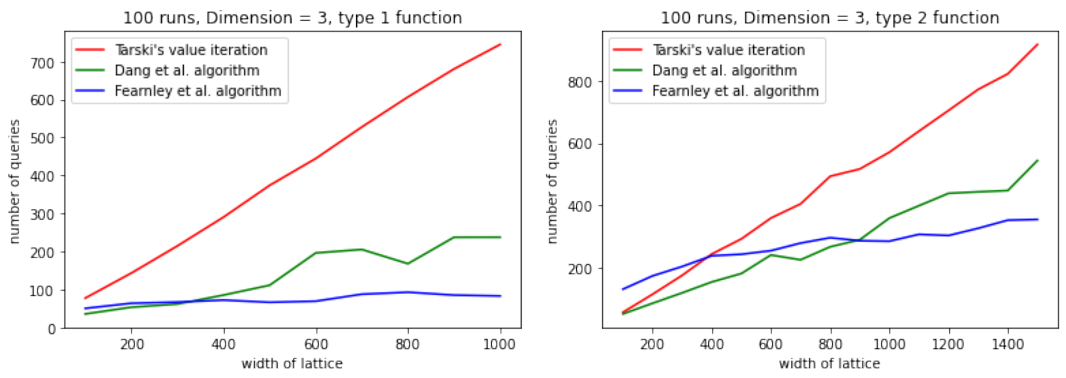

Figure 4.4: Generating 100 random monotone functions and taking the average, fixing dimension as 3, using type 1 function on the left and type 2 function on the right.

4.3 Fixing Width and Increases Dimension

When the width of lattice is fixed and dimension of lattice increases, Fearnley et al. algorithm requires more queries than the other 2 algorithms. Using width of 500, this happens when dimension is more than 5 in type 1 function and 3 in type 2 function. Note that Fearnley et al. algorithm uses Tarski's value iteration algorithm or Dang et al. algorithm when the last lattice after decomposing has dimension of 1 or 2 but not dimension of 3. Therefore, the reason of Fearnley et al. algorithm requiring more queries starting from dimension of 6 is not because of using the other 2 algorithms. It is clear than when width of lattice is small, using the steps in section 2.4.3 to solve lattice of dimension 4 or higher requires more queries than the other 2 algorithms.

On the other hand, Dang et al. algorithm requires more queries than Tarski's value iteration algorithm as dimension increases. For type 1 function and width of 500, the threshold is around 6 dimensions. For type 2 function and width of 500, the threshold is around 7 dimensions. Therefore, we can see that Dang et al. algorithm requires more queries than Tarski's value iteration as dimension of lattice increases. The thresholds are similar even when the width is increased, as shown in Figure 4.6.

There is a significant increases for Fearnley et al. algorithm when dimension is increased from 5 to 6 and 8 to 9. This is because the algorithm does not use Dang et al. algorithm or Tarski's value iteration algorithm to solve the last lattice after decomposing the original lattice in dimension of 6 or 9. When width is small, the other 2 algorithms requires less queries than Fearnley et al. algorithm hence the increase in number of queries. This statement is further proved in Figure 4.6.

As the width of lattice increases, the dimension required for Fearnley et al. algorithm to be slower than Tarski's value iteration increases. For example, Fearnley et al. algorithm requires more queries than Tarski's value iteration starting from dimension of 5 when the width is 500 using type 2 function. But the dimension changes to 8 when the width is increased to 50000.

When using type 1 function, Tarski's value iteration requires the most queries when dimension is less than 6 and width is 3000. In the same lattice, Dang et al. algorithm requires the most queries when dimension is more than 6 and less than 8. Starting from dimension of 9 with the same width, Fearnley et al. algorithm requires the most queries. Therefore, it is safe to say that the increase in dimension of lattice affects Fearnley et al. algorithm the most and Dang et al. algorihm is more affected by the increase in dimension when compare with Tarski's value iteration algorithm.

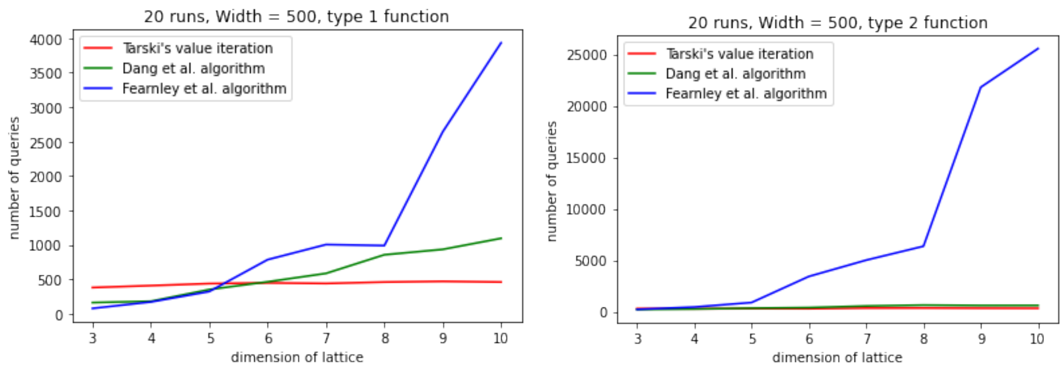

Figure 4.5: Generating 20 random monotone functions and taking the average, fixing width as 500, using type 1 function on the left and type 2 function on the right.

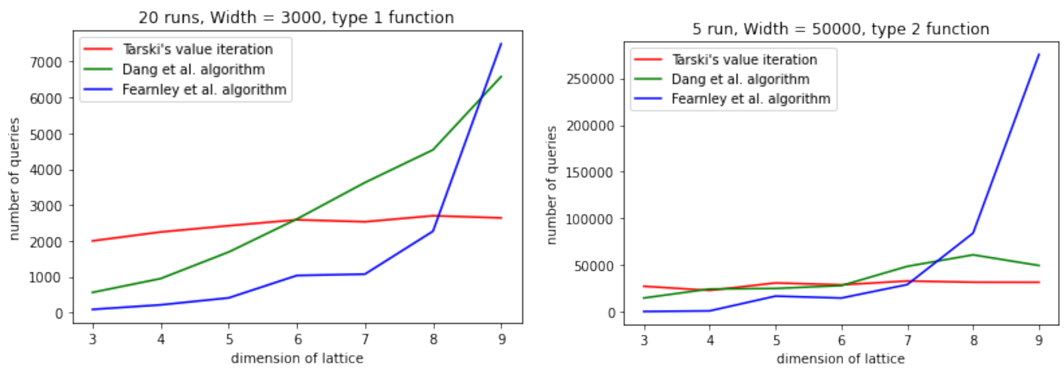

Figure 4.6: On the left, generating 20 random monotone functions, fixing width as 3000 and using type 1 function. On the right, generating 5 random monotone functions, fixing width as 50000 and using type 2 function.

4.4 Runtime

Lesser queries usually equals to faster runtime as shown when comparing Figure 4.4 and Figure 4.7. Although the monotone functions used in Figure 4.4 are different from that in Figure 4.7, the trend in both figures are similar. It is safe to say that the width of lattice affects the runtime and number of queries similarly.

Comparing Figure 4.5 and Figure 4.8 is more interesting. In Figure 4.8, there is a big increase in runtime between dimension 6 and 7 and dimension 9 and 10 using Fearnley et al. algorithm to solve both function. But Figure 4.5 shows that there is a big difference when increasing dimension from 8 to 9 and from 5 to 6.

The reason behind this inconsistency is because Fearnley et al. algorithm decomposes the lattice to 4 smaller lattices if the original lattice has dimension of 10. If the original lattice has dimension of 9, the algorithm decomposes the lattice to 3 smaller lattices. There are more recursive calls done by the algorithm when the lattice is decomposed to more smaller lattices, which causes more set P to be created. Each recursive calls requires the algorithm to search in set P which requires a lot of time. Therefore, a significantly more time is needed to solve lattice of dimension 10 compare with lattice of dimension 9. Same reasoning works for when dimension increases from 6 to 7.

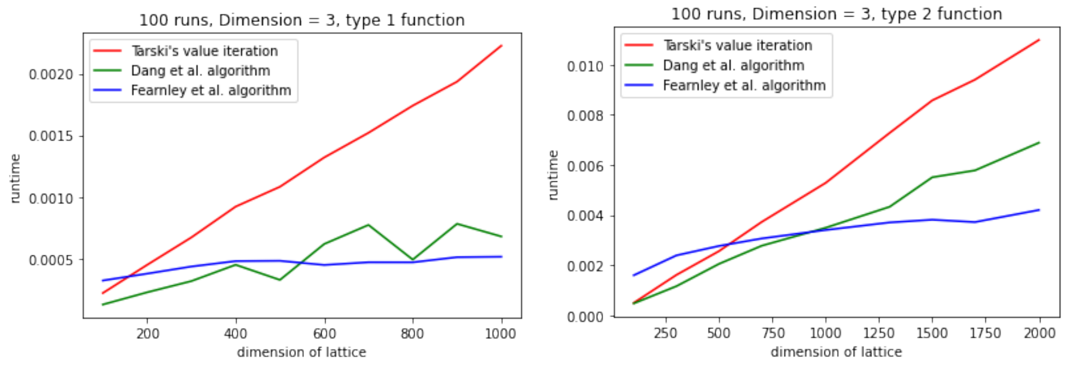

Figure 4.7

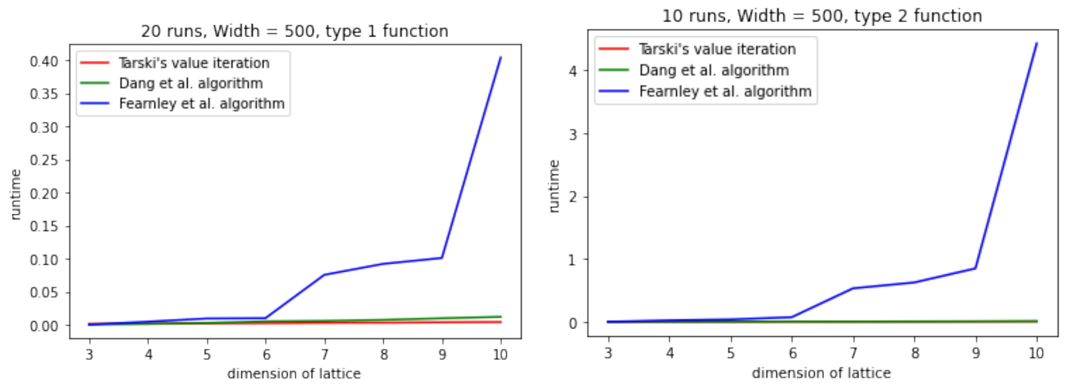

Figure 4.8

Chapter 5

Conclusion and Discussion

On paper, Fearnley et al. algorithm has the least amount of query complexities and Tarski's value iteration algorithm has the biggest query complexities to solve a Tarski instance when N is big relative to d. In chapter 4, I show that $O(log^{2\lceil d/3 \rceil} N)$ is indeed smaller than $O(Nd)$ as N increases while fixing d. However, different thresholds of N for that to remain true is needed when the dimension of lattice changes. On the other hand, as dimension increases and width of lattice is kept the same, Tarski's value iteration does require the least amount of queries as expected.

All monotone functions are generated randomly, therefore the average time to query a point on a monotone function should be roughly the same. But as shown in Figure 5.1, Fearnley et al. algorithm requires more time per query than the other 2 algorithms as d increases. The reason is because of the extra time needed by Fearnley et al. algorithm to search for the upper bound of set D and lower bound of set U. Therefore, a possible optimization would be decreasing the time needed to find the upper bound of D and lower bound of U.

Output of the functions are rounded down to the nearest integer, resulting some points to be slightly off from the actual points. Therefore, `invariant` and `inner_invariant` have a chance of showing an error because it finds a violation of order preservation in the lattice. My current workaround of this problem is as follows. Two points, x and y, are treated as the same if $abs(y_i - x_i) \leq 1$ for all $i \in d$ checking for the conditions of `invariant` and `inner_invariant`. Although this fixes the problem of `fps_algorithm`, there might be a better way to solve the problem.

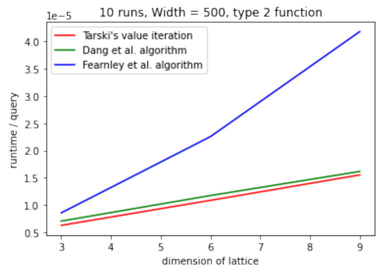

Figure 5.1: runtime per query against dimension

Bibliography

[1] Chuangyin Dang, Qi Qi, and Yinyu Ye. Computations and complexities of tarski's fixed points and supermodular games. 2012.

[2] Kousha Etessami, Christos Papadimitriou, Aviad Rubinstein, and Mihalis Yannakakis. Tarski's theorem, supermodular games, and the complexity of equilibria. 2020.

[3] John Fearnley, Dömötör Pálvölgyi, and Rahul Savani. A faster algorithm for finding tarski fixed points. 2021.

Appendix A

Code

A.1 tarski.py

```python
import math
import random
from functions import *
from fps_algorithm import *
from algorithms import *

class Monotone_function:
    def __init__(self, N, dimension, ftype) -> None:
        self.ftype = ftype
        self.N = N
        self.dimension = dimension
        self.queryPoint = []
        self.random = []
        self.nquery = 0
        self.smallStep = False
        self.start = 0
        self.end = dimension
        self.P = []
        for i in range(math.floor((dimension-1)/3)):
            self.P.append([])
        for i in range(self.dimension):
            temp = []
            temp.append(random.randint(0, self.dimension - 1))
            temp.append(random.randint(0, self.dimension - 1))
            self.random.append(temp)
            self.queryPoint.append(math.floor((N+1)/2))
        if ftype == 1 or ftype == 0:
            for i in range(self.dimension):
                self.random.append(random.randint(1, self.N))
        if ftype == 2:
            for i in range(self.dimension):
                p = []
                p0 = random.randint(1, self.N)
                p1 = random.randint(1, self.N)
                p2 = random.randint(1, self.N)
                p3 = random.randint(1, self.N)
                total = p0+p1+p2
                p0 = p0 / total
                p1 = p1 / total
                p2 = p2 / total
                p.append(p0)
                p.append(p1)
                p.append(p2*p3)
                self.random.append(p)
        if ftype == 3:
            for i in range(self.dimension):
                self.random.append(random.randint(1, self.N))

    def output(self, inp, raw=False):
        answer = []
        fp = []
        if self.end - self.start != self.dimension:
            for i in range(self.start, self.end):
                self.queryPoint[i] = inp[i-self.start]
            if self.end != self.dimension:
                x = []
                y = []
                for i in range(0, self.dimension):
                    x.append(1)
                    y.append(self.N)
                for k in range(math.floor(self.start/3)+1):
                    below_a = below_filter(self.P[k], self.
                                                       queryPoint, 3*
                                                       k, x, y)
                    above_a = above_filter(self.P[k], self.
                                                       queryPoint, 3*
                                                       k, x, y)
                    x = x[3:self.dimension]
                    y = y[3:self.dimension]
                    for j in range(len(below_a)):
                        bigger = True
                        for m in range(len(x)):
                            if below_a[j][m + (3*k) + 3] < x[m]:
                                bigger = False
                                break
                        if bigger:
                            for m in range(len(x)):
                                x[m] = below_a[j][m +(3*k)+3]
                    for j in range(len(above_a)):
                        smaller = True
                        for m in range(len(y)):
                            if above_a[j][m + (3*k) + 3] > y[m]:
                                smaller = False
                                break
                        if smaller:
                            for m in range(len(y)):
                                y[m] = above_a[j][m+(3*k)+3]
                for i in range(min(len(x),3)):
                    if y[i] < x[i]:
                        temp = y[i]
                        y[i] = x[i]
                        x[i] = temp
                old_start = self.start
                old_end = self.end
                fp = outer(self, x[0:min(len(x),3)], y[0:min(len(y),
                                                  3)], self.end, min
                                                  (self.end+3, self.
                                                  dimension), first=
                                                  True)
                self.start = old_start
                self.end = old_end
                for j in range(self.end, self.dimension):
                    self.queryPoint[j] = fp[j-self.end]
                b = tuple(self.queryPoint[:])
                self.P[math.floor(self.start/3)].append(b)
                self.P[math.floor(self.start/3)] = list(dict.
                                                  fromkeys(self.P[
                                                  math.floor(self.
                                                  start/3)]))
            inp = self.queryPoint[:]
        if self.ftype == 0 or self.ftype == 1:
            for i in range(self.dimension):
                j = self.random[i][0]
                k = self.random[i][1]
                if self.ftype == 1:
                    ans = max(inp[j], inp[k], self.random[self.
                                                      dimension+i])
                else:
                    ans = min(inp[j], inp[k], self.random[self.
                                                      dimension+i])
                if self.smallStep:
                    if ans > inp[i]:
                        ans = inp[i] + 1
                    elif ans < inp[i]:
                        ans = inp[i] - 1
                if not raw:
                    ans = math.floor(ans)
                answer.append(ans)
        elif self.ftype == 2:
            for i in range(self.dimension):
                j = self.random[i][0]
                k = self.random[i][1]
                p0 = self.random[self.dimension+i][0]
                p1 = self.random[self.dimension+i][1]
                p2 = self.random[self.dimension+i][2]
                ans = p0*inp[j] + p1*inp[k] + p2
                if ans < 1:
                    ans = 1
                elif ans > self.N:
                    ans = self.N
                if self.smallStep:
                    if ans > inp[i]:
                        ans = inp[i] + 1
                    elif ans < inp[i]:
                        ans = inp[i] - 1
                if not raw:
                    ans = math.floor(ans)
                answer.append(ans)
        elif self.ftype == 3:
            for i in range(self.dimension):
                p0 = self.random[i][0]
                p1 = self.random[i][1]
                p2 = self.random[self.dimension+i]
                j = (p0 + 1) / (p0 + p1 + 2)
                k = (p1 + 1) * p2 / (p0 + p1 + 2)
                ans = j*inp[i] + k
                if self.smallStep:
                    if ans > inp[i]:
                        ans = inp[i] + 1
                    elif ans < inp[i]:
                        ans = inp[i] - 1
                if not raw:
                    ans = math.floor(ans)
```

A.2 algorithms.py

```
import math
from functions import *
from tarski import *

def tarski_iteration(instance, fromBottom=True):
    current = []
    if fromBottom:
        for i in range(instance.dimension):
            current.append(1)
    else:
        for i in range(instance.dimension):
            current.append(instance.N)
    for j in range(instance.dimension):
        for i in range(instance.N):
            next = instance.output(current)[0]
            if next == current:
                return current
            current = next
    return []

def dqy_algorithm(instance, start, end, x, y):
    global answer
    global max_index
    global min_index
    answer = []
    max_index = []
    min_index = []
    if x == y:
        return x
    for i in range(end-start):
        answer.append(math.floor((x[i] + y[i])/ 2))
        max_index.append(y[i])
        min_index.append(x[i])
    fanswer = instance.output(answer)[0]
    while answer != fanswer:
        dqy_helper(instance, 0, start, end)
        fanswer = instance.output(answer)[0]
        if not below(answer, fanswer): #strictly above
            for i in range(end-start):
                min_index[i] = fanswer[i]
        elif not above(answer, fanswer): #strictly below
            for i in range(end-start):
                max_index[i] = fanswer[i]
        else:
            return answer
        for i in range(end-start):
            if max_index[i] < min_index[i]:
                temp = max_index[i]
                max_index[i] = min_index[i]
                min_index[i] = temp
    return answer

def dqy_helper(instance, depth, start, end):
    while True:
        answer[depth] = math.floor((max_index[depth]+min_index[depth
                                            ])/2)
        fanswer = instance.output(answer)[0]
        if answer[depth] == fanswer[depth]:
            if depth == end-start-1:
                return
            dqy_helper(instance, depth+1, start, end)
            return
        elif answer[depth] < fanswer[depth]:
            min_index[depth] = fanswer[depth]
        else:
            max_index[depth] = fanswer[depth]
        if max_index[depth] < min_index[depth]:
            temp = max_index[depth]
            max_index[depth] = min_index[depth]
            min_index[depth] = temp
```

A.3 fps_algorithm.py

```
import math
from functions import *
from tarski import *
from algorithms import *

def outer(instance, x, y, start=0, end=3, first=False):
    instance.start = start
    instance.end = end
    if first:
        for i in range(math.floor(start/3), math.floor((instance.
                                            dimension-1)/3)):
            instance.P[i] = []
        if end - start < 3:
            return dqy_algorithm(instance, start, end, x, y)
        fx = instance.output(x)[0]
        fy = instance.output(y)[0]
    maximize = -1
    maximize_value = -1
    mid = -1
    for i in range(3):
        value = y[i] - x[i]
        if value > maximize_value:
            maximize_value = value
            maximize = i
            mid = math.floor((y[i]+x[i])/2)
    if maximize_value <= 1:
        return last_case(instance, x)
    if first and invariant(x, y, fx, fy) == False:
        print("invariant failed")
        print(x, y, fx, fy)
        return []
    a = []
    b = []
    u = [0, 0, 0]
    d = [0, 0, 0]
    for i in range(3):
        if i == maximize:
            a.append(mid)
            b.append(mid)
        else:
            a.append(x[i])
            b.append(y[i])
    p = inner(instance, mid, maximize, a, b, u, d, False, [0,0,0],
                                                    True)
    if len(p) != 2:
        return p
    else:
        (pp, fpp) = p
    if fpp == 1 or fpp == -1:
        if fpp == 1:
            x = pp
        elif fpp == -1:
            y = pp
        return outer(instance, x, y, start, end)
    if up(pp, fpp, -1):
        x = pp
    elif down(pp, fpp, -1):
        y = pp
    else:
        print("outer failed")
        print(p)
        return p
    return outer(instance, x, y, start, end)

def inner(instance, principle_slice, principle_index, a, b, u, d,
                                  width1_secondrun, midone, first=
                                  False):
    index = []
    for i in range(len(a)):
        if i == principle_index:
            continue
        index.append(i)
    fa = instance.output(a)[0]
    fb = instance.output(b)[0]
    #Termination
    if b[index[0]] < a[index[0]] + 2 and b[index[1]] < a[index[1]] +
                                                    2:
        a1 = []
        a2 = []
        for i in range(len(a)):
                a1.append(a[i])
                a2.append(a[i])
        temp = [0,1,2]
        temp.remove(principle_index)
        a1[temp[0]] += 1
        a2[temp[1]] += 1
        if down(a, fa, -1) or up(a, fa, -1):
            return (a, fa)
        elif up(b, fb, -1) or down(b, fb, -1):
            return (b, fb)
        fa1 = instance.output(a1)[0]
        if down(a1, fa1, -1) or up(a1, fa1, -1):
            return (a1, fa1)
        fa2 = instance.output(a2)[0]
        if down(a2, fa2, -1) or up(a2, fa2, -1):
            return (a2, fa2)
        return []
    if u != [0,0,0]:
        fu = instance.output(u)[0]
    else:
        fu = u
    if d != [0,0,0]:
        fd = instance.output(d)[0]
    else:
        fd = d
    #check for invariant
    if first and not inner_invariant(a, b, u, d, fa, fb, fu, fd,
                                    principle_index):
        print("inner invariant failed")
        print([a, b, u, d, fa, fb, fu, fd, principle_index])
        return [a, b, u, d, fa, fb, fu, fd, principle_index]
    width1_firstrun = False
    width0 = False
    #width-one instance
    if not width1_secondrun:
        if b[index[0]] == a[index[0]] + 1 and b[index[1]] > a[index[
                                                    1]] + 1:
            if up_witness(a, u, fa, fu, principle_index) and a[index
                                                    [1]] == u[index[1]]:
                if down(a, fa, -1):
                    return (a, fa)
                elif down(u, fu, -1):
                    return (u, fu)
                if up(a, fa, principle_index):
                    u = [0,0,0]
                elif up(u, fu, principle_index):
                    a = u
                    u = [0,0,0]
            elif down_witness(d, b, fd, fb, principle_index) and d[
                                                 index[1]] == b[index[1
                                                 ]]:
                if up(b, fb, -1):
                    return (b, fb)
                elif up(d, fd, -1):
                    return (d, fd)
                if down(b, fb, principle_index):
                    d = [0,0,0]
                elif down(d, fd, principle_index):
                    b = d
                    d = [0,0,0]
            width1_firstrun = True
        elif b[index[1]] == a[index[1]] + 1 and b[index[0]] > a[
                                             index[0]] + 1:
            if up_witness(a, u, fa, fu, principle_index) and a[index
                                                 [0]] == u[index[0]]:
                if down(a, fa, -1):
                    return (a, fa)
                elif down(u , fu, -1):
                    return (u, fu)
                if up(a, fa, principle_index):
                    u = [0,0,0]
                elif up(u, fu, principle_index):
                    a = u
                    u = [0,0,0]
            elif down_witness(d, b, fd, fb, principle_index) and d[
                                                 index[0]] == b[index[0
                                                 ]]:
                if up(b, fb, -1):
                    return (b, fb)
                elif up(d, fd, -1):
                    return (d, fd)
                if down(b, fb, principle_index):
                    d = [0,0,0]
                elif down(d, fd, principle_index):
                    b = d
                    d = [0,0,0]
            width1_firstrun = True
    #width-zero instance
    if b[index[0]] == a[index[0]] and b[index[1]] > a[index[1]] + 1:
        if up_witness(a, u, fa, fu, principle_index) and a[index[1]]
                                             == u[index[1]]:
            if down(a, fa, -1):
                return (a, fa)
            elif down(u, fu, -1):
                return (u, fu)
            if up(a, fa, principle_index):
                u = [0,0,0]
            if up(u, fu, principle_index):
                a = u
                u = [0,0,0]
        elif down_witness(d, b, fd, fb, principle_index) and d[index
                                             [1]] == b[index[1]]:
            if up(b, fb, -1):
                return (b, fb)
            elif up(d, fd, -1):
                return (d, fd)
            if down(b, fb, principle_index):
                d = [0,0,0]
            if down(d, fd, principle_index):
                b = d
                d = [0,0,0]
        width0 = True
    elif b[index[1]] == a[index[1]] and b[index[0]] > a[index[0]] +
                                      1:
        if up_witness(a, u, fa, fu, principle_index) and a[index[0]]
                                           == u[index[0]]:
            if down(a, fa, -1):
                return (a, fa)
            elif down(u, fu, -1):
                return (u, fu)
            if up(a, fa, principle_index):
                u = [0,0,0]
            if up(u, fu, principle_index):
                a = u
                u = [0,0,0]
        elif down_witness(d, b, fd, fb, principle_index) and d[index
                                          [0]] == b[index[0]]:
            if up(b, fb, -1):
                return (b, fb)
            elif up(d, fd, -1):
                return (d, fd)
            if down(b, fb, principle_index):
                d = [0,0,0]
            if down(d, fd, principle_index):
                b = d
                d = [0,0,0]
        width0 = True
    #important points
    if not width1_secondrun:
        mid = [math.floor((a[index[0]] + b[index[0]])/2), math.floor
                                          ((a[index[1]] + b[index[1]
                                          ])/2)]
        mid.insert(principle_index, principle_slice)
        bot = [math.floor((a[index[0]] + b[index[0]])/2), a[index[1]
                                          ]]
        bot.insert(principle_index, principle_slice)
        top = [math.floor((a[index[0]] + b[index[0]])/2), b[index[1]
                                          ]]
        top.insert(principle_index, principle_slice)
        left = [a[index[0]], math.floor((a[index[1]] + b[index[1]])/
                                          2)]
        left.insert(principle_index, principle_slice)
        right = [b[index[0]], math.floor((a[index[1]] + b[index[1]])
                                          /2)]
        right.insert(principle_index, principle_slice)
        midone = [0,0,0]
    #width-one instance, second run
    else:
        mid = [math.ceil((a[index[0]] + b[index[0]])/2), math.ceil((
                                          a[index[1]] + b[index[1]])
```

```
                                    /2)]
    mid.insert(principle_index, principle_slice)
    bot = [math.ceil((a[index[0]] + b[index[0]])/2), a[index[1]]
                                    ]
    bot.insert(principle_index, principle_slice)
    top = [math.ceil((a[index[0]] + b[index[0]])/2), b[index[1]]
                                    ]
    top.insert(principle_index, principle_slice)
    left = [a[index[0]], math.ceil((a[index[1]] + b[index[1]])/2
                                    )]
    left.insert(principle_index, principle_slice)
    right = [b[index[0]], math.ceil((a[index[1]] + b[index[1]])/
                                    2)]
    right.insert(principle_index, principle_slice)
fmid = [0,0,0]
ftop = [0,0,0]
fbot = [0,0,0]
fleft = [0,0,0]
fright = [0,0,0]
#Step 1
if down_witness(d, b, fd, fb, principle_index):
    if d[index[1]] == b[index[1]] and compare(top, d, -1) != -1:
        ftop = instance.output(top)[0]
        if top[index[0]] > ftop[index[0]]:
            width1_secondrun = False
            if width1_firstrun:
                width1_secondrun = True
            return inner(instance, principle_slice,
                                           principle_index, a
                                           , top, u, d,
                                           width1_secondrun,
                                           midone)
        if top[principle_index] <= ftop[principle_index] and top
                                       [index[0]] <= ftop[
                                       index[0]]:
            d = top
    elif d[index[0]] == b[index[0]] and compare(right, d, -1) !=
                                     -1:
        fright = instance.output(right)[0]
        if right[index[1]] > fright[index[1]]:
            width1_secondrun = False
            if width1_firstrun:
                width1_secondrun = True
            return inner(instance, principle_slice,
                                           principle_index, a
                                           , right, u, d,
                                           width1_secondrun,
                                           midone)
        if right[principle_index] <= fright[principle_index] and
                                        right[index[1]] <=
                                       fright[index[1]]:
            d = right
if up_witness(a, u, fa, fu, principle_index):
    if a[index[1]] == u[index[1]] and compare(u, bot, -1) != -1:
        fbot = instance.output(bot)[0]
        if bot[index[0]] < fbot[index[0]]:
                width1_secondrun = False
                if width1_firstrun:
                    width1_secondrun = True
                return inner(instance, principle_slice,
                                                 principle_index,
                                                 bot, b, u, d,
                                                 width1_secondrun,
                                                 midone)
            if bot[principle_index] >= fbot[principle_index] and bot
                                               [index[0]] >= fbot[
                                               index[0]]:
                u = bot
        elif a[index[0]] == u[index[0]] and compare(u, left, -1) !=
                                                -1:
            fleft = instance.output(left)[0]
            if left[index[1]] < fleft[index[1]]:
                width1_secondrun = False
                if width1_firstrun:
                    width1_secondrun = True
                return inner(instance, principle_slice,
                                                 principle_index,
                                                 left, b, u, d,
                                                 width1_secondrun,
                                                 midone)
            if left[principle_index] >= fleft[principle_index] and
                                               left[index[1]] >=
                                               fleft[index[1]]:
                u = left
    #Step 2
    fmid = instance.output(mid)[0]
    if mid[index[0]] <= fmid[index[0]] and mid[index[1]] <= fmid[
                                     index[1]]:
        width1_secondrun = False
        if width1_firstrun:
            width1_secondrun = True
        return inner(instance, principle_slice, principle_index, mid
                                         , b, u, d,
                                         width1_secondrun, midone)
    if mid[index[0]] >= fmid[index[0]] and mid[index[1]] >= fmid[
                                     index[1]]:
        width1_secondrun = False
        if width1_firstrun:
            width1_secondrun = True
        return inner(instance, principle_slice, principle_index, a,
                                         mid, u, d,
                                         width1_secondrun, midone)
    if width0:
        if down(mid, fmid, -1):
            return (mid, fmid)
    if mid[index[0]] <= fmid[index[0]] and mid[index[1]] > fmid[
                                     index[1]]:
        if mid[principle_index] <= fmid[principle_index]:
            if fright == [0,0,0]:
                fright = instance.output(right)[0]
            if right[index[0]] < fright[index[0]]:
                return (d, 1)
            if mid[principle_index] <= fmid[principle_index] and
                                              right[principle_index]
                                               <= fright[
                                              principle_index] and
                                              right[index[0]] >=
                                              fright[index[0]]:
                if width1_secondrun and midone != [0,0,0]:
                    fmidone = instance.output(midone)[0]
                    if midone[index[1]] == fmidone[index[1]]:
                        return (midone, fmidone)
                width1_secondrun = False
                if width1_firstrun:
                    width1_secondrun = True
                return inner(instance, principle_slice,
                                                 principle_index, a
                                                 , right, u, mid,
                                                 width1_secondrun,
                                                 midone)
        else:
            if fbot == [0,0,0]:
                fbot = instance.output(bot)[0]
            if bot[index[1]] > fbot[index[1]]:
                return (u, -1)
            if mid[principle_index] >= fmid[principle_index] and bot
                                              [principle_index] >=
                                              fbot[principle_index]
                                              and bot[index[1]] <=
                                              fbot[index[1]]:
                if mid[index[0]] == fmid[index[0]]:
                    return (mid, fmid)
                width1_secondrun = False
                if width1_firstrun:
                    midone = mid
                    width1_secondrun = True
                return inner(instance, principle_slice,
                                                 principle_index,
                                                 bot, b, mid, d,
                                                 width1_secondrun,
                                                 midone)
    if mid[index[0]] > fmid[index[0]] and mid[index[1]] <= fmid[
                                        index[1]]:
        if mid[principle_index] <= fmid[principle_index]:
            if ftop == [0,0,0]:
                ftop = instance.output(top)[0]
            if top[index[1]] < ftop[index[1]]:
                return (d, 1)
            elif mid[principle_index] <= fmid[principle_index] and
                                              top[principle_index] <
                                              = ftop[principle_index
                                              ] and top[index[1]] >=
                                               ftop[index[1]]:
                if width1_secondrun and midone != [0,0,0]:
                    midone = instance.output(midone)[0]
                    if midone[index[0]] == midone[index[0]]:
                        return (midone, fmidone)
                width1_secondrun = False
                if width1_firstrun:
                    width1_secondrun = True
                return inner(instance, principle_slice,
                                                 principle_index, a
                                                 , top, u, mid,
                                                 width1_secondrun,
                                                 midone)
        else:
            if fleft == [0,0,0]:
                fleft = instance.output(left)[0]
            if left[index[0]] > fleft[index[0]]:
                return (u, -1)
            elif mid[principle_index] >= fmid[principle_index] and
                                               left[principle_index]
                                               >= fleft[
                                               principle_index] and
                                               left[index[0]] <=
                                               fleft[index[0]]:
                if mid[index[1]] == fmid[index[1]]:
                    return (mid, fmid)
                width1_secondrun = False
                if width1_firstrun:
                    midone = mid
                    width1_secondrun = True
                return inner(instance, principle_slice,
                                                 principle_index,
                                                 left, b, mid, d,
                                                 width1_secondrun,
                                                 midone)
```

A.4 functions.py

```
#return True if x precede or equal x_prime
def below(x, x_prime):
    for i in range(len(x)):
        if x_prime[i] > x[i]:
            return False
    return True

#return True if x_prime precede or equal x
def above(x, x_prime):
    for i in range(len(x)):
        if x_prime[i] < x[i]:
            return False
    return True

#return -1 if x precede or equal y,
#ignoring the dimension of principle_index
def compare(x, y, principle_index):
    for i in range(len(x)):
        if i == principle_index:
            continue
        if x[i] > y[i]:
            return 1
    return -1

#return True if x precede or equal fx,
#ignoring the dimension of principle_index
def up(x, fx, principle_index):
    if x == [0,0,0]:
        return False
    if compare(x, fx, principle_index) == -1:
        return True
    return False

#return True if fx preced or equal x,
#ignoring the dimension of principle_index
def down(x, fx, principle_index):
    if x == [0,0,0]:
        return False
    if compare(fx, x, principle_index) == -1:
        return True
    return False

#return True if outer invariant is satisfied
def invariant(x, y, fx, fy):
    if compare(x, y, -1) == 1:
        return False
    principle_index = -1
    if not up(x, fx, principle_index) and not same(x, fx):
        return False
    if not down(y, fy, principle_index) and not same(y, fy):
        return False
    return True

#return True if inner invariant is satisfied
def inner_invariant(a, b, u, d, fa, fb, fu, fd, principle_index):
    if (up(a, fa, principle_index) or same(a, fa)) or (compare(u, b,
                                                -1) == -1 and up_witness(a, u
                                               , fa, fu, principle_index)):
        if (down(b, fb, principle_index) or same(b, fb)) or (compare
                                                    (a, d, -1) == -1 and
                                                    down_witness(d, b, fd, fb,
                                                     principle_index)):
            return True
    return False

#return True if (d, b) is a valid down set witness
def down_witness(d, b, fd, fb, principle_index):
    if d == [0,0,0] or b == [0,0,0]:
        return False
    if d[principle_index] > fd[principle_index] or b[principle_index
                                               ] > fb[principle_index]:
        return False
    index = []
    for i in range(len(d)):
        if i == principle_index:
            continue
        index.append(i)
    if d[index[0]] == b[index[0]] and d[index[1]] <= b[index[1]]:
        if d[index[1]] <= fd[index[1]] and fb[index[1]] <= b[index[1
                                                    ]]:
            return True
    elif d[index[1]] == b[index[1]] and d[index[0]] <= b[index[0]]:
        if d[index[0]] <= fd[index[0]] and fb[index[0]] <= b[index[0
                                                    ]]:
            return True
    return False

#return True if (a, u) is a valid up set witness
def up_witness(a, u, fa, fu, principle_index):
    if a == [0,0,0] or u == [0,0,0]:
        return False
    if a[principle_index] < fa[principle_index] or u[principle_index
                                        ] < fu[principle_index]:
        return False
    index = []
    for i in range(len(a)):
        if i == principle_index:
            continue
        index.append(i)
    if a[index[0]] == u[index[0]] and u[index[1]] >= a[index[1]]:
        if u[index[1]] >= fu[index[1]] and fa[index[1]] >= a[index[1
                                                    ]]:
            return True
    elif a[index[1]] == u[index[1]] and a[index[0]] <= u[index[0]]:
        if u[index[0]] >= fu[index[0]] and fa[index[0]] >= a[index[0
                                                    ]]:
            return True
    return False

#Called at termination step of outer,
#uses Tarskis value iteration to find a fixed point in instance,
#starting from the point x
def last_case(instance, x):
    while True:
        newx = instance.output(x)
        if x == newx[0]:
            ans = newx[0][:]
            if len(newx) > 1:
                for i in range(len(newx[1])):
                    ans.append(newx[1][i])
            return ans
        x = newx[0]

#Find the upper bound of set D mentioned in section 2.4.3
def below_filter(setP, queryPoint, start, x, y):
    ans = []
    for i in range(len(setP)):
        below = True
        for j in range(3):
            if setP[i][start+j] > queryPoint[start+j] or setP[i][
                                                   start+j] < x[j] or
                                                   setP[i][start+j] > y[j
                                                   ]:
                below = False
                break
        if below:
            ans.append(setP[i][:])
    return ans

#Find the lower bound of set U mentioned in section 2.4.3
def above_filter(setP, queryPoint, start, x, y):
    ans = []
    for i in range(len(setP)):
        above = True
        for j in range(3):
            if setP[i][start+j] < queryPoint[start+j] or setP[i][
                                                  start+j] < x[j] or
                                                  setP[i][start+j] > y[j
                                                  ]:
                above = False
                break
        if above:
            ans.append(setP[i][:])
    return ans

#Check if the difference between 2 points,
#x and y, is at most 1 in any dimension.
def same(x,y):
    for i in range(len(x)):
        if abs(y[i] - x[i]) <= 1:
            continue
        else:
            return False
    return True
```

A.5 test.py

```
import timeit
from tarski import *
from algorithms import *
from fps_algorithm import *

def do_the_tests(d, N, t, n, smallstep, algorithm):
    t0 = 0
    t1 = 0
    t2 = 0
    ss = True if smallstep == 1 else False
    nqueries = [0,0,0]
    number = n
    if len(algorithm) != 3:
        return
    tarski = algorithm[0]
    dqy = algorithm[1]
    fps = algorithm[2]
    for i in range(n):
        if fps == 1:
            ttt = Monotone_function(N=N, dimension=d, ftype=t)
            ttt.smallStep = ss
            tx = []
            ty = []
            for j in range(3):
                tx.append(1)
                ty.append(ttt.N)
            ttt.nquery = 0
            start0 = timeit.default_timer()
            tfp = outer(ttt, tx, ty, 0, 3, first=True)
            end0 = timeit.default_timer()
            nqueries[2] += ttt.nquery
            t0 += end0 - start0
            ttt.start = 0
            ttt.end = ttt.dimension
            if len(tfp) != d or ttt.output(tfp)[0] != tfp:
                print(i, "fsp failed")
                break
        if dqy == 1:
            ttt1 = Monotone_function(N=N, dimension=d, ftype=t)
            ttt1.smallStep = ss
            ttt1.start = 0
            ttt1.end = ttt.dimension
            x = []
            y = []
            for i in range(ttt1.dimension):
                x.append(1)
                y.append(ttt1.N)
            ttt1.nquery = 0
            start1 = timeit.default_timer()
            tfp1 = dqy_algorithm(ttt1, 0, ttt.dimension, x, y)
            end1 = timeit.default_timer()
            nqueries[1] += ttt1.nquery
            t1 += end1 - start1
            if len(tfp1) != d or ttt1.output(tfp1)[0] != tfp1:
                print(i, "dqy failed")
                break
        if tarski == 1:
            ttt2 = Monotone_function(N=N, dimension=d, ftype=t)
            ttt2.smallStep = ss
            ttt2.start = 0
            ttt2.end = ttt.dimension
            ttt2.nquery = 0
            start2 = timeit.default_timer()
            tfp2 = tarski_iteration(ttt2, True)
            end2 = timeit.default_timer()
            nqueries[0] += ttt2.nquery
            t2 += end2 - start2
            if len(tfp2) != d or ttt2.output(tfp2)[0] != tfp2:
                print(i, "t_iteration failed")
                break
    for i in range(3):
        nqueries[i] /= number
        t0 /= number
        t1 /= number
        t2 /= number
    if tarski == 1:
        print("\nTarski's value iteration (in second): ", t2)
        print("Tarski's value iteration (in number of queries): ",
                                            nqueries[0])
    if dqy == 1:
        print("\nDang et al. algorithm (in second): ", t1)
        print("Dang et al. algorithm (in number of queries): ",
                                            nqueries[1])
    if fps == 1:
        print("\nFearnley et al. algorithm (in second): ", t0)
        print("Fearnley et al. algorithm (in number of queries): ",
                                            nqueries[2])

if __name__ == "__main__":
    tarski = input("Do you want to test Tarski's value iteration? (
                                        yes: 1, no: 0)\n")
    if tarski == '':
        tarski = '1'
    dqy = input("Do you want to test Dang et al. algorithm? (yes: 1,
                                         no: 0)\n")
    if dqy == '':
        dqy = '1'
    fps = input("Do you want to test Fearnley et al. algorithm? (yes
                                        : 1, no: 0)\n")
    if fps == '':
        fps = '1'
    N = input("Specific the width of lattice.\n")
    if N == '':
        N = "10000"
    d = input("Specific the dimension of lattice.\n")
    if d == '':
        d = "3"
    t = input("Specific the type of monotone function. (min: 0, max:
                                         1, type 2: 2, type 3: 3)\n")
    if t == '':
        t = '2'
    smallStep = input("Specific if smallStep should be turned on. (
                                        yes: 1, no: 0)\n")
    if smallStep == '':
        smallStep = '1'
    number = input("Specific the number of tests.\n")
    if number == '':
        number = "100"
    algorithm = [int(tarski), int(dqy), int(fps)]
    do_the_tests(int(d), int(N), int(t), int(number), int(smallStep)
                                        , algorithm)
```