Two Heuristic Methods for Solving Generalized Nash Equilibrium Problems Using a Novel Penalty Function

by

Roie Fields

A Thesis
presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of

Master of Science
in
Mathematics & Statistics

Guelph, Ontario, Canada

ABSTRACT

TWO HEURISTIC METHODS FOR SOLVING GENERALIZED NASH EQUILIBRIUM PROBLEMS USING A NOVEL PENALTY FUNCTION

Roie Fields
University of Guelph, 2021

Advisor(s):
Dr. Monica-Gabriela Cojocaru
Dr. Edward Thommes

In this work, we solve Generalized Nash Equilibrium Problems using two novel heuristic models. We introduce the Shadow Point function, a novel penalty function for Generalized Nash Equilibrium Problems, similar to the Nikaidô-Isoda penalty function [1] to motivate the behavior of these two models. The first is an evolutionary-inspired algorithm which utilizes competitive selection and linear regression to motivate generation of new points. The other algorithm involves stochastic gradient descent of the Shadow Point function across mass numbers of agents to find solutions. These algorithms are evaluated on 2- and 3-player games in 2 and 3 dimensions, with both linear and non-linear shared constraints. The success of these algorithms is discussed, and the limitations of the algorithms are explored. Finally, we discuss potential remedies to these limitations, and additional ways to further optimize the methods.

ACKNOWLEDGEMENTS

I would first like to thank my advisors, Dr. Monica Cojocaru and Dr. Edward Thommes for their guidance, feedback and support throughout my degree and beforehand. I thank them for always pushing me to grow and to explore my limits, and for truly caring for me as both a student and a person.

I would like to thank my research partners for their camaraderie and support. In particular I would like to thank Lia Humphrey, whose friendship has helped shape my graduate experience.

I would like to thank my parents and siblings, who always push me to explore my limits and be the best version of myself I can be. Their support and dedication kept me on the path to success, and made me feel secure along the way.

Finally I would like to thank my grandparents. Their optimistic perseverance through struggle, and their dedication to family is a legacy I strive to honor every day.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This thesis investigates the use of two computational heuristic models to solve Generalized Nash Equilibrium Problems (GNEPs). Chapter 1 introduces GNEPs and current methods used to solve them. Our sample problem set is introduced and briefly explained before use in our algorithms. Chapter 2 introduces the *shadow point* function, a novel penalty function used by our algorithms, and compares it to the Nikaidô-Isoda function, a function used commonly in the numerical optimization literature. Chapter 3 introduces the Evolutionary Inspired algorithm as our first heuristic method to solve GNEPs, and tests it on our problem set. In Chapter 4, the Stochastic Gradient Descent model is introduced and tested on the same problem set as the Evolutionary-Inspired Algorithm. Chapter 5 discusses and compares the two models. This chapter also investigates the limitations of these algorithms, as well as ways to remedy these limitations, and future work required to further optimize the algorithms' performance.

1.1 Generalized Nash Equilibrium Problems (GNEP)

Game theory is defined as the study of mathematical models of conflict and cooperation between intelligent rational decision makers [2]. That is, game theory allows mathematicians to study the behaviour of intelligent rational entities when their decisions (also referred to

as "strategies") affect one another. These actors are assumed to be rational in that they seek to optimize their objectives, and intelligent in that they look for the optimal strategy in any given situation to achieve said optimum. Since these decisions affect one another, the optimal strategy for each player depends on the decision made by other players. As such, we seek to find strategy sets in which for each player, their objective cannot be further optimized assuming all other players do not change their already optimal decisions. In other words, every player's strategy is optimal given the optimal strategy sets of all other players. These strategy sets are referred to as Generalized Nash Equilibriums (GNE's) and were introduced by Debreu in 1952 as a shared-constraint expansion of Nash Equilibriums [3]. Nash Equilibriums were named after John Nash, who had introduced these equilibrium points in games in 1950 with a one-page article titled "Equilibrium points in n-person games" [4] that revolutionized the field of economics [5]. In addition to its use in economics [6],[7],[8], game theory has been expanded and utilized in other fields such as animal behaviour and cooperation [9],[10],[11], common pool resource use [12],[13],[14],[15] and consumer markets such as power and internet bandwidth allocation [16],[17].

In mathematics, heuristic algorithms are methods which, on the basis of experience or judgement, seem likely to yield a good solution to a problem but which cannot be guaranteed to produce an optimum [18]. Many heuristic approaches have been taken to solving GNEPs. Nikaidô and Isoda developed a function to measure the fitness of any given point, known as the Nikaidô-Isoda function, which uses the difference between the objective function values at the given point versus at optimal points [1]. This function has been used by many other researchers as a basis with which to motivate algorithms that search for, most often, a single point which optimizes an objective function. Various gradient-type and relaxation methods have been used to compute GNE's in jointly convex GNEPs using the Nikaidô-Isoda function [19], [20]. Isolated solution points can be found by converting GNEPs into

Variational Inequalites, however some of these methods do not allow for larger solution sets to be found, and can only be used under certain conditions [21], [22]. In the last few years, complete methods developed based on the use of Variational Inequalities have been presented by Migot and Cojocaru in [23], [24]. Other methods include transforming these Generalized Nash Equilibrium Problems into Quasi-Variational Inequality Problems (QVI's) to allow for more solutions to be found [25], however QVI's are inefficient to solve. Wild developed an evolutionary technique to approximate solutions for GNEPs using the concept of Nash dominance to rank fitness of the agents [26].

In this work, we introduce a new function with which to evaluate fitness of points chosen in an expanded version of the feasible set for a GNEP, and we use this penalty function to motivate the behaviour of our algorithms to search for solutions to this GNEP, and to evaluate whether or not certain points may be part of the solution set of the GNEP. This function, a variation of the Nikaidô-Isoda function, is given by the distance to the *shadow point*, as described in Chapter 2. We then explore two methods with which to solve GNEPs based on the *shadow point* function, one evolutionary-inspired, and one using stochastic gradient descent. These techniques aim to solve the system directly, avoiding reformulation of the problem and convexity or concavity assumptions.

1.2 Mathematical Formulation of GNEP

When Nash first introduced the Nash Equilibrium problem, it involved an n-player game in which each player sought to minimize payoff (penalty) by choosing the optimal strategy from their strategy set. This described a series of players making isolated decisions, and observing how that would affect the system as a whole [4], [27]. The key limitation of this style of game is that it cannot describe situations where strategy spaces of some players are affected by the decisions of others. This is a key limitation when describing systems

where individuals have shared constraints, such as when modeling the use of common-pool resources or competitive economies. This limitation was remedied by Arrow and Debreu who introduced a generalized version of Nash Equilibrium Problems, aptly named Generalized Nash Equilibrium Problems [3], [28]. Here, we formulate and define the GNEP similar to Facchinei et al [22].

There are N players, with each player v controlling the variables $x^v \in \mathbb{R}^{n_v}$. The strategy vector formed by the decision variables of each player is denoted by x, where $x := (x^1, ..., x^N)^T$, with the decision vector for all the players *except for* player v is denoted x^{-v}. Each of these players has an objective function $\theta_v : \mathbb{R}^N \to \mathbb{R}$ that depends on their own variables x^v, as well as other players' variables x^{-v}. The strategies of each player is constrained by the set $X_v(x^{-v}) \subseteq \mathbb{R}^{n_v}$, dependent on other player's strategies. The goal of a player v is to choose the strategy x^v such that is solves the minimization problem

$$\begin{aligned} &\text{minimize}_{x^v} \;\; \theta(x^v, x^{-v}) \\ &\text{subject to} \quad x^v \in X_v(x^{-v}) \end{aligned} \tag{1.1}$$

For any x^{-v}, the solution set is denoted by $S_v(x^{-v})$. A solution of the GNEP is a vector $\hat{x} := (\hat{x}^1, ..., \hat{x}^N)^T$ such that $\hat{x}^v \in S_v(x^{-v}) \;\forall\; v \in (1, ..., N)$
We call $\hat{x}$ an equilibrium if no player can further optimize their objective function by changing their strategy to any other point in the feasible set. By definition, when $X_v(x^{-v})$ are independent of other players strategies, this reduces to a NEP.

Definition 1. *Let X be the set of all possible situations of a GNEP. A strategy set $\hat{x} \in X$*

is a Generalized Nash Equilibrium iff

$$\theta_v(\hat{x}) \leq \theta_v(x^v, x^{-v}) \ \forall \ v \in (1, ..., N), \ \textit{and} \ \forall \ x^v \in X_v(x^{-v}) \tag{1.2}$$

1.3 Test Problems

In order to evaluate our 2 algorithms, we test them on the problem set shown in Table 1.1. This problem set is taken from examples that appear in literature and were initially compiled by Erin Wild [26]. Example 1 is the simplest example, with 2 players with shared constraints each controlling one variable, and whose objective functions only depend on that variable. It is taken from [22] and was initially introduced as a basic demonstration of a GNEP. Examples 2, 3, 5 and 6 were used by [29] to demonstrate the ability of Variational Inequalities to solve GNEP's. Example 6 was also used by [20] to demonstrate the ability to use relaxation algorithms in conjunction with the Nikaidô-Isoda function to solve GNEPs. Example 2 is similar to Example 1, however with the distinction that the objective functions depend on all variables, not only those under the respective player's control. Example 3 is the first example to use non-linear constraints. Example 5 expands our set to problems in 3 dimensions, however in this case it only has 2 players, with one of the players controlling 2 variables. For this reason, our solution set here is given by a plane rather than a line. Example 6 is a 3-dimensional, 3-player game, with each player controlling 1 dimension, resulting in a line segment solution. This game is especially interesting as it models a theoretical river-basin pollution problem. In this problem, 3 players seek to optimize their respective payoffs in a shared a river basin by controlling their own degree of pollution, subject to shared constraints. Example 4 was introduced by Harker [25] to demonstrate the ability to find additional solutions GNEPs to those found by Variational Inequalities (VIs) by using Quasi-Variational Inequalites (QVIs). In addition to the lone solution at $(5, 9)$

found by the VI, Harker found a line segment $(t, 15-t)$ for $9 \leq t \leq 10$ using QVIs. Finally, Example 7 was constructed by Wild [26] as a 3-dimensional extension of Example 3, with 3 players sharing both linear and non-linear constraints.

Table 1.1: GNEP set we seek to solve, taken from [26].

	Objective functions	**Constraints**	**Optimal solutions**
Ex. 1	$P_1 : \min_x (x-1)^2$ $P_2 : \min_y (y-\frac{1}{2})^2$	$x, y \geq 0$ $x + y \leq 1$	$\begin{pmatrix} t \\ 1\text{-}t \end{pmatrix}$, $\frac{1}{2} \leq t \leq 1$
Ex. 2	$P_1 : \min_x x^2 - xy - x$ $P_2 : \min_y y^2 - \frac{1}{2}xy - 2y$	$x, y \geq 0$ $x + y \leq 1$	$\begin{pmatrix} t \\ 1\text{-}t \end{pmatrix}$, $0 \leq t \leq \frac{2}{3}$
Ex. 3	$P_1 : \min_x x^2 - xy - x$ $P_2 : \min_y y^2 - \frac{1}{2}xy - 2y$	$x, y \geq 0$ $x^2 + y^2 \leq 1$	$\begin{pmatrix} t \\ \sqrt{1-t^2} \end{pmatrix}$, $0 \leq t \leq \frac{4}{5}$
Ex. 4	$P_1 : \min_x x^2 + \frac{8}{3} - 34x$ $P_2 : \min_y y^2 - \frac{5}{4}xy - 24.25y$	$x, y \geq 0$ $x, y \leq 10$ $x + y \leq 15$	$\begin{pmatrix} 5 \\ 9 \end{pmatrix} \cup \begin{pmatrix} t \\ 15-t \end{pmatrix}$, $9 \leq t \leq 10$
Ex. 5	$P_1 : \min_{x,y} x^2 + xy + y^2 + (x+y)z - 25x - 38y$ $P_2 : \min_z z^2 + (x+y)z - 25z$	$x, y, z \geq 0$ $x + 2y - z \leq 14$ $3x + 2y + z \leq 30$	$\begin{pmatrix} t \\ 11\text{-}t \\ 8\text{-}t \end{pmatrix}$, $0 \leq t \leq 2$
Ex. 6	$P_1 : \min_{x_1}(\alpha_1 x_1 + \beta(x_1+x_2+x_3) - \chi_1)x_1$ $P_2 : \min_{x_2}(\alpha_2 x_2 + \beta(x_1+x_2+x_3) - \chi_2)x_2$ $P_3 : \min_{x_3}(\alpha_3 x_3 + \beta(x_1+x_2+x_3) - \chi_3)x_3$	$x_1, x_2, x_3 \geq 0$ $3.25x_1 + 1.25x_2 + 4.125x_3 \leq 100$ $\alpha_1 = 0.001$, $\alpha_2 = 0.05$, $\alpha_3 = 0.01$, $\beta = 0.01$, $\chi_1 = 2.9$, $\chi_2 = 2.88$, $\chi_3 = 2.85$	
Ex. 7	$P_1 : \min_x x^2 - xy - x$ $P_2 : \min_y y^2 - \frac{1}{2}xy - 2y$ $P_3 : \min_z (z - \frac{1}{2})^2$	$x, y, z \geq 0$ $x^2 + y^2 \leq 1$ $z + y \leq 1$	

Chapter 2

The Shadow Point Function

Before we can introduce the two algorithms, we must first introduce a penalty function that helps us select "better" points among all points we search in a given step of the algorithm. This penalty function will allow us to evaluate the fitness of any point or agent at any given stage of either algorithm, and thus tell our algorithms how to treat these points.

2.1 Construction of the Shadow Point function

When designing this penalty function, we wanted to ensure that the output exhibits four key properties:

Firstly, the output of the function should be equal for all solution points (1). No solution should have a better or worse penalty than another solution, they should all be treated equally.

Secondly, the output of the penalty function should be minimized for our feasible set by all solutions (2). In other words, there should not exist feasible points outside our solution set with a lower penalty values than those inside the solution set.

Thirdly, the penalty value of points on our solution set should be unique to points on the set

(3). In other words, this minimal penalty value is given *only* by points in our solution set. Finally, our fourth property is that the penalty of all points should be easily interpretable under the definition of a Nash Equilibrium (4). When an individual familiar with the penalty function is given the penalty of a point, they should have a clear understanding of the quality of that solution, regardless of their depth of knowledge into the specific problem. If a point in one example that is similarly near their solution as a point in another example is to its solution, they should have similar penalty values. Admittedly, this final requirement is much more subjective in nature.

In summary, we want the function to give an **equal** (1) and **optimal** (2) penalty value if and **only if** (3) a point is a member of the solution set, and to be **easily understandable** (4) without deep knowledge of the problem for all points in the feasible set .

Definition 2. *Let us assume a game with n players, and a constraint set $C \in \mathbb{R}^n$. Let p_i be a decision vector of variables controlled by player i. Then a point $P = (p_1, p_2, ..., p_n)$ is a set of decisions vectors p_i made by each player i.*

Definition 3.

1. *We define $\hat{p}_i$, the **shadow coordinate** of p_i, as $\hat{p}_i = p_i^*$ subject to $C|p_{-i}$, where p_i^* is the optimum of p_i. That is, $\hat{p}_i$ is the optimum of p_i under the constraint set C given the decision vectors of all other players p_{-i}.*

2. *Then the **shadow point** $\hat{P} = (\hat{p}_1, \hat{p}_2, ..., \hat{p}_i)$ is the set of shadow coordinates associated with P.*

3. *We define $penalty(P) = ||P - \hat{P}||_2$, the Euclidean distance of P to its shadow point $\hat{P}$. All together, the penalty of any point P is given by*

$$penalty(P) = \sqrt{\sum_{i=1}^{n}((p_i^*|p_{-i}) - (p_i))^2} \tag{2.1}$$

For the problem set described in Table 1.1, we will use Example 1 as a demonstration.

Given a point $P = (x, y) = (0.4, 0.9)$, we solve for the shadow point $\hat{P} = (\hat{x}, \hat{y})$.

$$\hat{x} = (x^*|y) = min((x-1)^2) \;\; s.t. \;\; x + 0.9 \leq 1 \Rightarrow \;\; \hat{x} = 0.1 \tag{2.2}$$

$$\hat{y} = (y^*|x) = min((y-0.5)^2) \;\; s.t \;\; y + 0.4 \leq 1 \Rightarrow \;\; \hat{y} = 0.5 \tag{2.3}$$

Thus, the shadow point is $\hat{P} = (\hat{x}, \hat{y}) = (0.1, 0.5)$, and

$$penalty(P) = ||P - \hat{P}||_2 = \sqrt{(0.1-0.4)^2 + (0.5-0.9)^2} = 0.5 \tag{2.4}$$

By the definition of a Nash Equilibrium, we have the property that $x = \hat{x}$ and $y = \hat{y}$ if and only if P is a Generalized Nash Equilibrium, thus $P = \hat{P}$ and $penalty(P) = 0$. Thus, any algorithm utilizing this penalty function should seek to minimize penalty in order to find solutions. The fact that all solutions have an equivalent penalty satisfies requirement (1). Since for any member of the feasible set the penalty is always non-negative and the penalty of members of the solution set is 0, we satisfy requirement (2). Since a penalty of 0 is unique only to members of the solution set, we satisfy requirement (3). Finally, since the penalty is simply a Euclidean distance of the decision vector to a theoretical optimum, it is easy to understand the meaning of a penalty, regardless of the parameters of the problem.

2.2 Comparison to the Nikaidô-Isoda function

The Nikaidô-Isoda function was introduced in 1955 by Hukukane Nikaidô and Kazuo Isoda in a short paper published by the Pacific Journal of Mathematics [1], with the goal of improving Nash's method of generalizing von Neumann's game matrices into n-dimensional

non-cooperative games played over infinite-dimensional convex sets, now known as Generalized Nash Games and Generalized Nash Equilibria [4]. The Nikaidô-Isoda function is given by

$$penalty(P) = \sum_{i=1}^{n} (u_i(p_i^*|p_{-i}) - u_i(p_i)) \tag{2.5}$$

where u_i is the objective function of player i. As an extension of Nash's recent introduction of the Nash Equilibrium, this function presents a method by which to evaluate how far a point is from being a Nash Equilibrium. At the time, Nash Games were thought of as an economic tool. In economics, the payoff function is what is most important. An economic player may ask "how far away are my results from the optimal results?". By summing the displeasure among each player's economic result, this function presents a method of measuring a version of the distance from an equilibrium for non-equilibrium points as well. The Nikaidô-Isoda function has been used by Rosen to motivate the gradient descent method [19], and by Krawczyk *et al.* for relaxation algorithms [20]. More recently, it was used for path-following methods using an approximating scheme by Hintermuller *et al.* [30] to solve GNEPs.

Recall that the shadow point function is given by

$$penalty(P) = \sqrt{\sum_{i=1}^{n} ((p_i^*|p_{-i}) - (p_i))^2} \tag{2.6}$$

The key difference between the *shadow point* function and the Nikaidô-Isoda function is that while the Nikaidô-Isoda function sums the difference in objective functions relative to their optima, the *shadow point* function sums the squares of the difference in strategy required to reach the optima. We argue that from a computational perspective, the later approach makes more sense than the former: Consider the case where the objective function

has a local max with a value very close to that of the global max, but for which the strategy set to achieve this max is very far. This point would be considered very good, and close to a solution by the Nikaidô-Isoda function, since the difference in objective function is very small. This is not problematic for economic purposes, since in economics, it may be acceptable for a strategy to be very slightly suboptimal if it yields very close to optimal results.

However, for the purposes of finding true Nash Equlibiria using heuristics, this is very problematic as it treats a point as if it is close to being a solution even if it is not. While the Nikaidô-Isoda penalty will not be zero in these situations, they may be low enough to fall within the penalty threshold decided by the user, leading to false positive results. By measuring the distance from the optimal strategy itself, rather than comparing the results of the suboptimal strategies to the optimal ones, the *shadow point* function avoids these issues of local optima in our objective functions.

Another case in which the *shadow point* function is preferred is in the case where the objective function fluctuates significantly near the optimum. In these cases, a point slightly off the optimal strategy may have a very large gap in objective function value. The Nikaidô-Isoda function would treat this as a very suboptimal point, which may motivate large movements in some algorithms. The *shadow point* function recognizes that the strategies only require slight changes, and may have an easier time finding the optimal values as a result.

The other difference between the two penalty functions is that the Nikaidô-Isoda function simply sums the differences, while the *shadow point* function takes the square root of the sum of squared differences. Both functions ensure that all summed terms are non-negative. The Nikaidô-Isoda does this naturally, as the optimal value of the objective function is always greater or equal to the selected value by definition, allowing for a non-negative difference, and an easy summation. This is sensible in an economic context, as one may simply want to know what the total economic losses are as a result of this suboptimal strategy.

From a computational perspective, we do not care about the total losses of our objective

functions, we want to know how much we need to move to find our equilibrium points. As such, we care about the difference in strategy, not the results of those strategies. The *shadow point* function ensures non-negativity in the summed terms by squaring the difference. This is because the optimal strategy could be higher or lower in value to the chosen one, losing the guarantee that this difference is positive. We chose to square this difference rather than taking the absolute value of this difference in order to penalize larger distances more harshly than smaller ones. This is especially important when dealing with inherently imprecise optimizers. Due to rounding errors and computation being based on threshold precision, calculated optima may be slightly different than true optima. When dealing with strategies extremely close to optima, this imprecision accounts for a much larger percentage of the penalty than in more suboptimal strategies. For this reason, we diminish the effect of small distances to optima and accentuate the effect of larger differences.

Chapter 3

Evolutionary-Inspired Algorithms

The first of our two heuristic methods is an Evolutionary Inspired algorithm. While very different than the method introduced by Wild [26], this method utilizes some of the same basic framework of traditional evolutionary algorithms; the algorithm seeks to find solutions by selecting the most fit agents in each generation, and uses them to generate replacements for those agents not selected.

3.1 Evolutionary-Inspired Algorithm Method

The general process of the Evolutionary-Inspired algorithm (EIA) can be described by repeating 4 steps, with a 0-th step to initialize the process in the first generation.

0. Initialize points (Initialization)

1. Evaluate Fitness of each point (Evaluation)

2. Select the most fit points (Selection)

3. Perform a regression between selected points (Regression)

4. Generate replacement points for those not selected (Replacement)

Step 0 - Initialization: Before we can begin our algorithm, we need a set of initial points. For each point, we randomly initialize the value of each dimension within that dimension's global bounds under the shared constraints, independent of all other dimensions. In example 1, each of x and y have global bounds [0,1], and so we initialize each of x and y as a random value in [0,1] independently of each other, despite the fact that this allows for the generation of points that violate the shared constraint $x + y \leq 1$.

When solving complex problems, the feasible set can be difficult to calculate and describe. For this reason, initializing each dimension in a 'best case scenario' global bounds allows for implementation of the algorithm without needing a full understanding and description of the feasible set.

Step 1 - Evaluation: To evaluate the fitness of each point, we use the penalty function given by Equation 2.1 introduced in Chapter 2. Since this penalty function requires an optimization, we use the mulitvariable optimization function optimize.minimize_scalar built into the Python library Scipy, using the SLSQP method for bounded optimization.

Step 2 - Selection: In the selection step, the existing points are ordered according to penalty, and the m lowest penalty scores are selected, with the remaining unselected points deleted.

Step 3 - Regression: In the regression step, we perform a regression of the selected points, to achieve a curve of best fit for that generation of points. For problems with linear solution sets, a linear regression is sufficient. For nonlinear solution sets, a higher order regression is required, however that is outside the scope of this project and is left for explo-

ration in the future.

Step 4 - Replacement: In this final step, we must replace the $n - m$ deleted points from step 2 using the regression curve found in step 3. To do so, we must first generate x coordinates, and use the regression curve, as well as some manufactured noise, to find corresponding y coordinates. Since in general the solution set does not cover the entire range of x coordinates, we want our selection range of x coordinates to converge onto the solution range of x coordinates. Since early iterations of the algorithm are unlikely to produce the full range of x coordinates, we must search past our current range of selected points. To do so, we take the current upper and lower x limits of our selected points, and extend them $\delta^{generation}$ of the way to the global bounds of x, such that $\delta < 1$, and $generation$ is the current generation (iteration) of the algorithm.
Let $xMinSelected$ and $xMaxSelected$ represent the minimum and maximum x values of our selected points respectively, and let $xGMin$ and $xGMax$ represent the global min and max of x based on the constraint set respectively. Based on this operation, our lower bound $xMinSearch$ and upper bound $xMaxSearch$ is given by

$$xMinSearch = xMinSelected - (xMinSelected - xGMin) \cdot \delta^{generation} \tag{3.1}$$

$$xMaxSearch = xMaxSelected + (xGMax - xMaxSelected) \cdot \delta^{generation} \tag{3.2}$$

Once these new bounds are defined, we generate $n - m$ replacement x coordinates from a uniform distribution within this range $(xMinSearch, xMaxSearch)$. We select a δ very close to 1, such that in the early part of the algorithm, we generate new points from almost the entire global bounds of x, regardless of the coordinates and quality of the selected points. As the algorithm progresses and the generation count grows, the range of x coordinates in

selected points should converge onto the true range of x in the solution set, allowing us to search a much more local area in later generations with a much lower risk of restricting our search space too quickly. As the confidence of the algorithm towards the range of solution grows, $\delta^{iterations} \to 0$ and thus the extra space it searches beyond those bounds shrinks. In the case of more than 2 dimensions, this process is repeated for all but one dimension.

After generating x coordinates, we are able to select corresponding y coordinates using our regression. Since the points in early generations are not guaranteed to be of high quality, the regression curve resulting from an early generation is also not guaranteed to be representative of the true solution set. As such, it is important to add noise to the y coordinate beyond its location on the curve, to ensure new locations are explored, and that the regression does not converge too early and get stuck despite the selection of new points due to new x coordinates. This noise is accomplished in a similar manner as the expansion of the x search space, by expanding the search space by a proportion of the space to the global bounds of y. Let $yRegression$ represent the y coordinate on the regression curve at the newly generated x coordinate. Let $yGMin$ and $yGMax$ represent the global min and max bounds of y based on the constraint set. Based on these definitions and this operation, our lower bound $yMinSearch$ and upper bound $yMaxSearch$ are given by

$$yMinSearch = yRegression - (yRegression - yGMin) \cdot \epsilon^{generations} \tag{3.3}$$

$$yMaxSearch = yRegression + (yGMax - yRegression) \cdot \epsilon^{generations} \tag{3.4}$$

For each of the $n - m$ x coordinates generated, a y coordinate is generated from a uniform distribution in $(yMinSearch, yMaxSearch)$. Similar to the δ used previously, we select an $\epsilon < 1$, but very close to 1. As the algorithm progresses and the regression becomes more

representative of the true solution set, $\epsilon^{generations} \rightarrow 0$, and so we generate y coordinates much closer to the regression curve.

These 4 steps are repeated until threshold convergence is met, or until a maximum number of iterations are met. In order to get a more complete picture of the solution set, this algorithm is run from scratch many times, and the results of each model are attached to create a plot of the solution set. The following describes pseudo-code for running one EIA model:

```
Generate vector xPoints of length numPoints, each in
 (xMinConstraint,xMaxConstraint);
Generate vector yPoints of length numPoints, each in
 (yMinConstraint,yMaxConstraint);
while (iterations ≤ maxIterations) AND (mean(penalty) ≥ threshold) do
    myPenalty = penalty(xPoints, yPoints);
    ordered = np.argsort(penalty);
    bestX = [xPoints[ordered[i]] for i in range(numSelect)];
    bestY = [yPoints[ordered[i]] for i in range(numSelect)];
    slope, intercept = sp.stats.linregress(bestX, bestY);
    yPoints = nextGenY(bestY, inter + 1, slope, intercept, Epsilon);
    xPoints = nextGenX(bestX, inter + 1, slope, intercept, Delta);
    iterations += 1
end
```

3.2 Evolutionary-Inspired Algorithm Results

As mentioned previously, we test the two algorithms using the problem set described in Section 1.3 by Table 1.1.

Example 1 is the simplest of the problems. In this example, we have linear constraints, and the objective functions of each player are functions only of variables controlled by that player, albeit still with shared constraints. In Figure 3.1 and Figure 3.2 we see the solutions

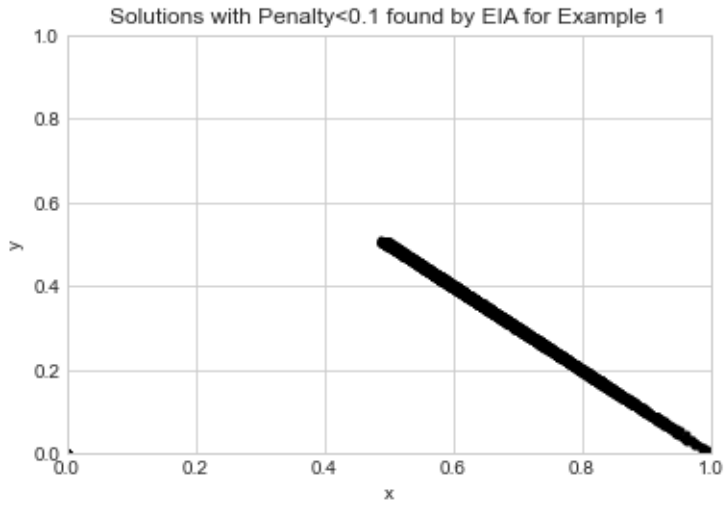

Figure 3.1: Solutions found by EIA for Example 1

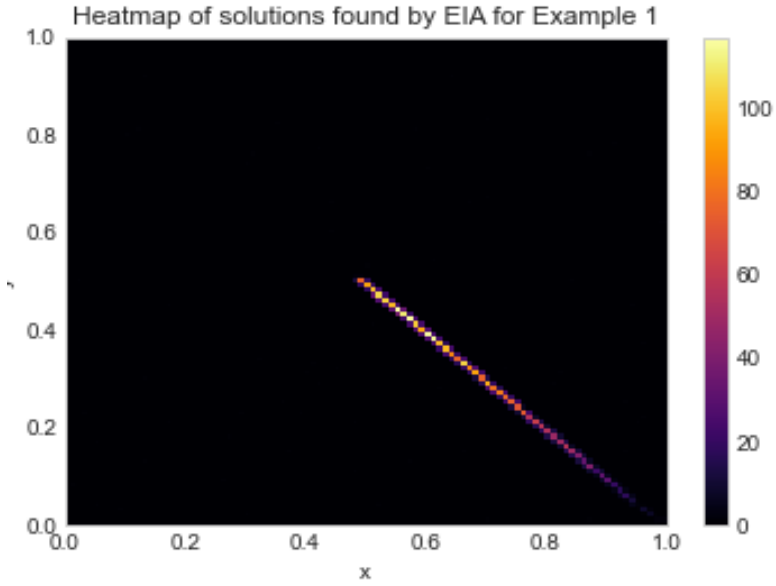

Figure 3.2: Heatmap of Solutions found by EIA for Example 1

found by the Evolutionary Inspired Algorithm (EIA) for this example represent the true solution set. When observing the heatmap in Figure 3.2, we see that solutions in the middle of the search space are more common than those towards the edges. This is a result of the point generation function described in Section 3.1. The algorithm generates a random number within a uniform distribution between the upper and lower bounds calculated by the

bounds of each dimension, the proximity to those bounds and the current iteration count. When closer to the bounds, the search space is much larger on the opposite direction of the bound than the direction toward the bound. As such, the randomly generated number is biased towards the middle of the set, and thus more solutions are likely to be found in this region. This might be remedied by tweaking the generation function to choose a direction first (with equal chance in each direction), and then pick a value in the chosen direction based on the proximity to the bounds in that direction and iteration count as is currently performed. Out of the 5000 points initialized across 500 models, 4807 (96%) converged onto the solution set with penalty ≤ 0.1.

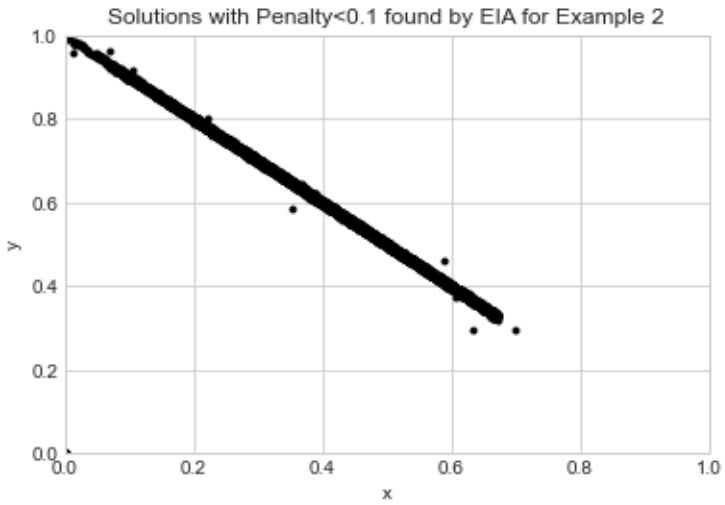

Figure 3.3: Solutions found by EIA for Example 2

Example 2 is similar to Example 1 in that it has the same shared linear constraint, however now with objective functions that depend on the strategies of both players. The EIA model was successful at finding solutions for this problem as well, as demonstrated by Figures 3.3 and 3.4. We observe that the set plotted represents the true solution set for this GNEP. Out of the 5000 points given by the 500 models, 4911 (98.22%) converged within a penalty ≤ 0.1. The points observed that are not touching the line are simply points so

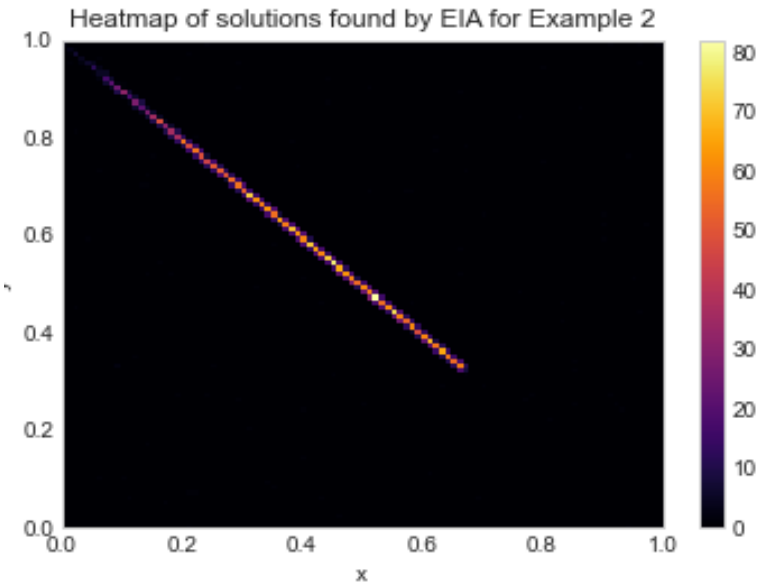

Figure 3.4: Heatmap of Solutions found by EIA for Example 2

close to the solution set that they maintain a penalty within this 0.1 threshold. As is clear from the heatmap, the algorithm has some bias towards finding solutions closer to the center of the search space, and does not find as many solutions close to the bounds. Similarly to question 1, this is due to the next generation function having a much higher chance to generate points in the direction opposite the bounds when near the bounds, as there is less space there. Overall, the dependence of the objective function on both player's decisions does not appear to affect the ability of this algorithm to find the solution set.

Example 3 introduces a key change in that the constraints are non-linear, resulting in a non-linear solution set. Since the EIA involves a linear regression, it is unable to solve this problem. Higher degree regression needs to be investigated in order to see whether this algorithm is capable of handling non-linear solution sets.

Recall from Section 1.3 that Example 4, while also used by Dr. Erin Wild [26], was formu-

lated by Harker to demonstrate the ability of Variational Inequalities and Quasi-Variational Inequalities to solve GNEPs [25]. It is interesting in that in addition to a line segment, the solution set contains a 'Rosen Point'; it contains a point that is disconnected from the rest. Another important detail for our methods to consider is that this Rosen point does not lie on the same line as the line segment, as is seen in Figure 3.5.

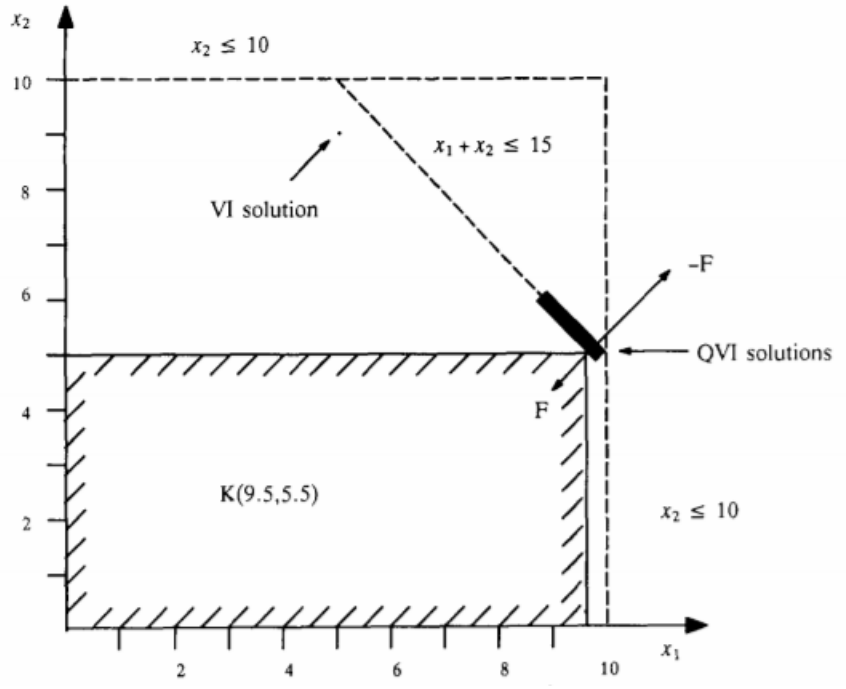

Figure 3.5: Solutions found by Dr. Patrick Harker for Example 4 [25]

This is problematic for the EIA method, as points near this Rosen point will have low penalty values. This causes the EIA to select points found near this solution in the selection step of the algorithm, where they are subsequently included in the linear regression. In this case, since the Rosen point is underneath the line created by the linear solution set, points found nearby this Rosen point 'pull' our linear regression down, skewing its ability to find solutions on the linear segment. Luckily, as demonstrated by Harker, this Rosen point can be computed relatively easily using Variational Inequalites [25]. After doing so, one can manually introduce an additional penalty for points found in the vicinity of the Rosen point,

lowering their fitness and making it less likely for these points to be selected. This allows the algorithm to find solutions on the linear set, with the user having the knowledge that the additional solution point exists as well. In this example, a penalty of 1.0 was added to any point within 0.5 of the Rosen point. The results of running the EIA with this manufactured penalty is show in Figure 3.6.

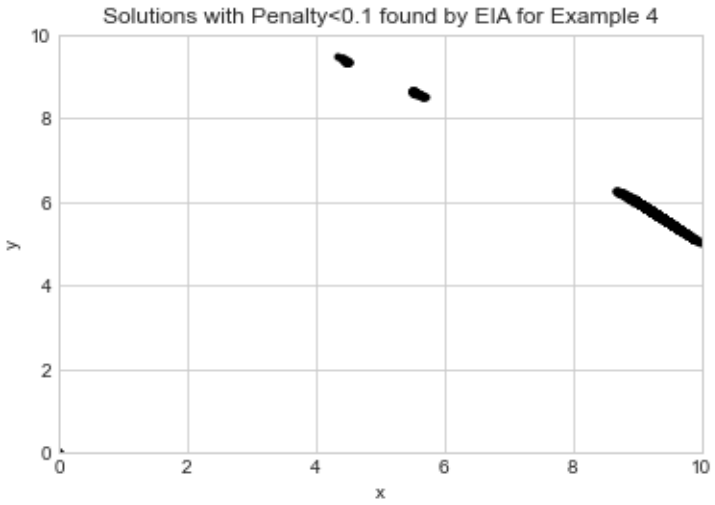

Figure 3.6: Solutions found by EIA for Example 4

In total, 4996 of the 5000 points (99.99%) converged onto points with penalty ≤ 0.1. Some of these points still surround the Rosen point of $(5, 9)$. These points are those who still have low penalty scores as they are near the Rosen Point, however are greater than 0.5 away and so manage to avoid the manufactured penalty. This can be remedied by increasing the region where we manufacture additional penalty. Fortunately, when observing the heatmap of solutions found be the EIA for example 4, we see that almost all points found are within the line segment solution set.

When observing the heatmap, we can see that the solutions found near the Rosen Point are so few that they cannot even be seen in the heatmap. As is the case for most of these examples, the points near the interior region of the search space are heavily favored over those near the bounds, for the same reason as previously.

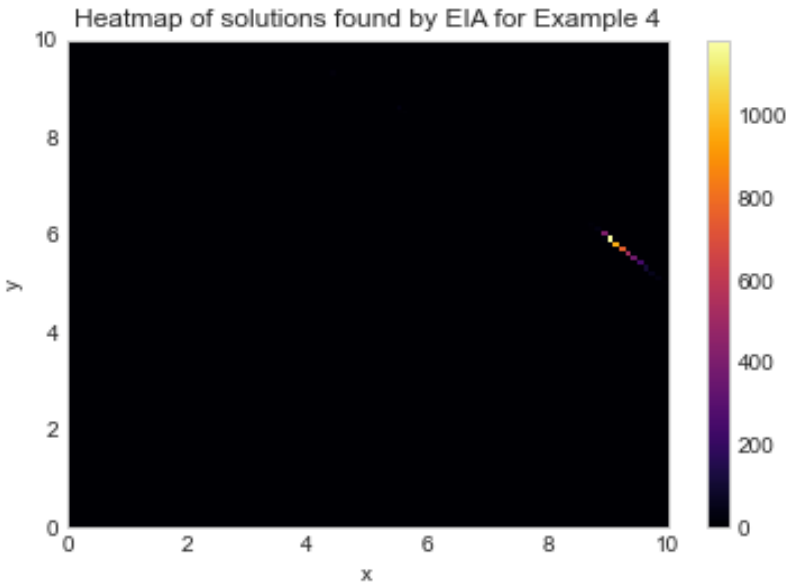

Figure 3.7: Heatmap of Solutions found by EIA for Example 4

Example 5 is a big step up in complexity, as it adds a third dimension, z. In this example, player 1 controls both variables x and y, and player 2 only has control over z. For this reason, we require a multi-variable optimizer. We utilize the SLSQP method in the optimize.minimize function in the *scipy* Python library. When doing so, we find the same solution set as found by [26], as well as an extension of it as shown in figures Figure 3.8 and Figure 3.9.

The solution set found by Dr. Wild is given by $(x, y, z) = (t, 11 - t, 8 - t)$ with $t \in [0, 2]$. Our solution extension is given by $(x, y, z) = (t, 13 - 2t, 12 - 3t)$ with $t \in (2, 3.4)$. It is possible that this line extends past $x = 3.4$, but the algorithm does not find points in this region. Points found in this newly found solution set were tested and confirmed to meet the definition of a GNE. As with other examples, solutions near the bounds are more sparse, and so more testing would be needed to get a better understanding of the extension of the solution set.

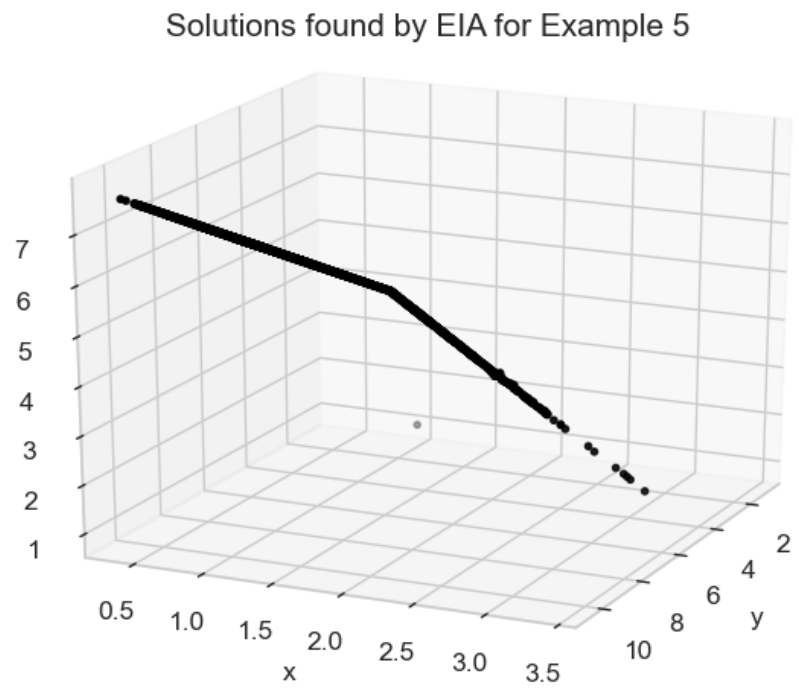

Figure 3.8: Solutions found by EIA for Example 5

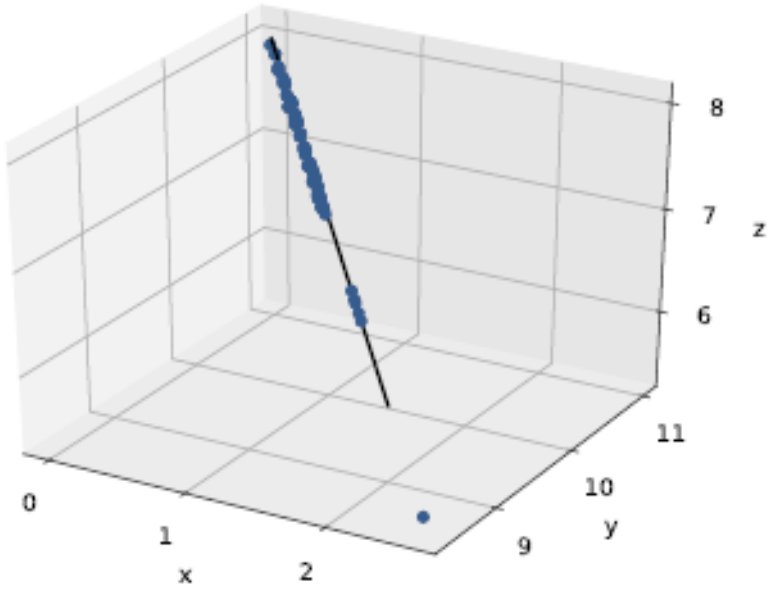

Figure 3.9: Solutions found by Dr. Erin Wild for Example 5 [26]

Recall that Example 6 represents a real world application. It represents a river basin pollution game introduced by Krawczyk, Jacek and Uryasev in [20]. Mathematically, example 6 is interesting in that it maintains the 3-dimensional challenge introduced in example 5, but instead of having one player control 2 variables, it introduces a third player. So here we have a 3 player game with one variable controlled by each player.

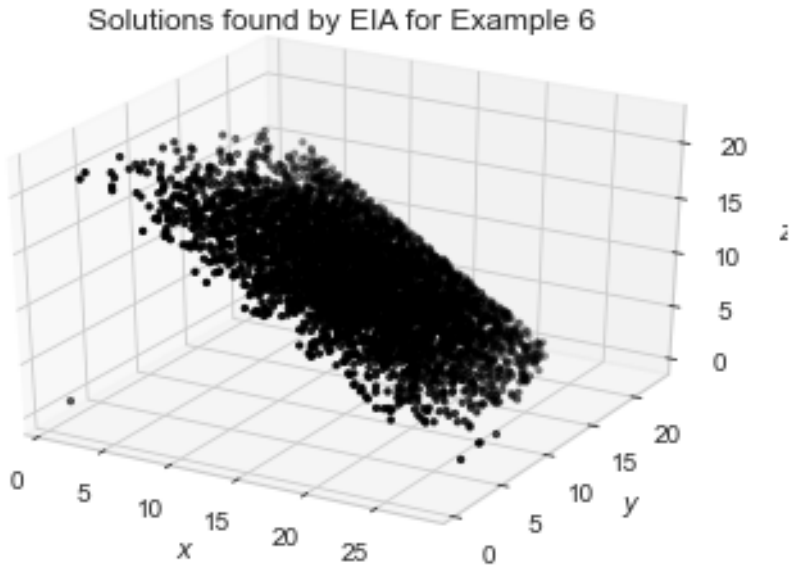

Figure 3.10: Solutions found by EIA for Example 6

Figure 3.10 shows the solutions found by the EIA method for Example 6. Each model made by the EIA is generally given by 10 points that lie on the same line. Since this solution set is given by a plane, there are many line segments that may be found each time the algorithm is run. Since we run 500 models and plot the results together, we observe here that the lines found by the EIA method are spread out enough that collectively they represent the majority of the true solution set. There are fewer points in the upper and lower regions of the solution set in the z direction, as well as in the lower end of the y direction. This is for the same reason as in the other examples; points are less likely to be initialized towards

the bounds of the search space. We do not observe this phenomenon in the upper limits of the y direction since it is not near the bounds of the search space. In this example, 4999 of the 5000 points found by the EIA algorithm had a penalty ≤ 0.1. The high success rate of the algorithm for this example can likely be attributed to the larger size of the solution set making it easier to locate, as well as the the fact that the set is located in the relative center of the search space.

Example 7 is a 3-player generalization of Example 3. It has 2 players in players 1 and 2 whose objective functions only depend on the variables controlled by one another (x and y, and a third player in player 3 whose objective function only depends on their own controlled variable (z). x and z are each only constrained by y, though it is important to note that the constraint on x is nonlinear, while y is constrained by both x and z in separate functions. The combination of both linear and nonlinear constraints allows for a solution set that is partly linear and partly nonlinear. Unfortunately, for the same reason as in Example 3, the non-linearity of the solution set as a whole restricts the ability of the EIA to solve this problem.

Chapter 4

Stochastic Gradient Descent Algorithm

The second of the two heuristic algorithms used in this work is the Stochastic Gradient Descent algorithm. This algorithm generates 'children' points for each 'parent' point, and uses their penalty to estimate the gradient of the parent penalty, motivating the parent to move in the direction of steepest descent. Unlike the Evolutionary-Inspired Algorithm, the Stochastic Gradient Descent Algorithm operates on individual points rather than an entire generation. That is, the behaviour of 'parent' agents in this algorithm does not depend on the behaviour of any other parent agents. Because of this, the algorithm can be programmed such that it operates on large numbers of agents in parallel, allowing for a large number of solution points to be found despite only running the algorithm once. This is also known as an "Embarrassingly Parallel" algorithm, as it requires no communication between agents [31].

4.1 Method

The general process of the Stochastic Gradient Descent algorithm can be described by the repetition of 3 steps, with a 0-th step to initialize our first generation of points.

0. Initialize 'Parent' points (Initialization)
1. For each parent, generate a set of 'Children' in the local region of each point (Exploration)
2. Evaluate the fitness of the children (Evaluation)
3. Transform Parent according to a penalty-weighted average of Children's step size (Transformation)

Step 0 - Initialization: This algorithm requires an initial Parent point to operate on. We initialize these Parents in the same way as we do in Step 0 of our EIA in Chapter 3. That is, we initialize each dimension within the best-case-scenario global bounds of the constraint set for that dimension.

Step 1 - Exploration: In this step, each Parent P generates a series of m 'Children' in their local region. This is done by generating a m steps in each dimension, given by $\vec{\sigma} = (\sigma_1, \sigma_2, ..., \sigma_m)$, with each σ_i selected using a Gaussian distribution centered at 0 with standard deviation of $\alpha^{generation}$, using the random.normal function in the numpy python library. We then add them to the Parent P's coordinates to create a new set of points, the children, given by $P + \sigma_i$. This allows the Parent to search its local region without actually moving from its initial location. As the algorithm progresses, $\alpha^{generation}$ approaches 0, causing the children to search a more local area as confidence in the model grows.

Step 2 - Evaluation: In the second step, the penalty values of each of the children are calculated using the penalty function given by equation 2.1 described in Chapter 2. In order to calculate this penalty in parallel for every child of every parent, we use a vectorized version of the golden section Search algorithm in place of the optimize.minimize_scalar Scipy function used in the EIA method. This allows for far faster computation, allowing us to solve thousands of points at once instead of using an iterative approach, however it also constrains this method to single-variable optimization problems. As such, in order to generalize to problems in which a single player can control multiple variables at once, a different optimization algorithm must be used. That is left as future work.

Step 3 - Transformation: In the third and final step of the algorithm, the fitness of the children, along with their respective step sizes away from their parent, are used to generate a step size with which to transform the parent point for the next generation. For each parent P, β, the step for the next generation, is a fitness-weighted average of the children's steps, given by

$$\beta = \frac{1}{m} \sum_{i=1}^{m} \sigma_i (penalty(P) - penalty(P + \sigma_i)) \tag{4.1}$$

and thus the next generation of P is given by $P + \beta$.

Similar to the Evolutionary Inspired Algorithm, this process is repeated until threshold convergence, or until the maximum number of iterations are met. We show that this process approximates Gradient Descent, given a sufficient number of children generated.

Let us define our penalty function as $F(\vec{x})$. Assuming F is smooth, the Taylor Series expansion of F is given by

$$F(\vec{x} + \sigma g^i) \approx F(\vec{x}) + \sigma \langle g^i, \nabla F(\vec{x}) \rangle \tag{4.2}$$

The Gaussian vector g^i can be decomposed as:

$$\vec{g^i} = g^i_{\|} \widehat{\nabla F(\vec{x})} + \vec{g^i_{\perp}} \tag{4.3}$$

where $g_{\|} \sim \mathcal{N}(0,1)$ is a scalar Gaussian which gives the component of g^{-i} in the direction $\widehat{\nabla F(\vec{x})}$ and $g^{-i}_{\perp}$ is a Guassian that lives in the $d-1$ space, perpendicular to direction $\widehat{\nabla F(\vec{x})}$. In any orthonormal basis for this $d-1$ dimensional subspace, the entries of $g^{-i}_{\perp}$ are iid $\mathcal{N}(0,1)$ Gaussians too. With this decomposition, the inner product that appears is precisely:

$$\langle g^{-i}, \nabla F(\vec{x}) \rangle = g^i_{\|} ||\nabla F(\vec{x})|| \tag{4.4}$$

From this decomposition, we see that

$$\begin{aligned}
\frac{1}{n} \sum_{i=1}^{n} \sigma \vec{g}_i \left(F(\vec{x} + \sigma \vec{g^i}) - F(\vec{x}) \right) &\approx \frac{\sigma^2}{n} \sum_{i=1}^{n} \vec{g}_i \left\langle \vec{g^i}, \nabla F(x) \right\rangle \\
&= \frac{\sigma^2}{n} \sum_{i=1}^{n} \left(g^i_{\|} \widehat{\nabla F(x)} + \vec{g^i_{\perp}} \right) g^i_{\|} ||\nabla F|| \\
&= \left(\frac{1}{n} \sum_{i=1}^{n} \left(\sigma g^i_{\|} \right)^2 \right) \nabla F + \frac{\sigma^2}{n} ||\nabla F|| \sum_{i=1}^{n} g^i_{\|} \vec{g^i_{\perp}}
\end{aligned}$$

Note that coefficient in front of ∇F of the first term $\frac{1}{n} \sum_{i=1}^{n} \left(\sigma g^i_{\|} \right)^2 := \frac{^2}{n} \chi^2_n$ is a mean σ^2 random scalar that is concentrated around σ^2 (it has variance $\frac{2\sigma^4}{n}$). This can be seen since it is an average of n independent random variables which are mean σ^2.

On the other hand, the second term, which always lies in the subspace per perpendicular to ∇F, is $\frac{\sigma^2}{n} ||\nabla F|| \sum_{i=1}^{n} g^i_{\|} \vec{g^i_{\perp}}$ is mean 0. By independence of the terms in the sum, in

expectation, the magnitude of this vector is :

$$\mathbb{E}\left[||\frac{\sigma^2}{n}||\nabla F||\sum_{i=1}^{n} g_{\|}^{i}\vec{g}_{\perp}^{i}||^2\right] = \frac{\sigma^4}{n}||\nabla F||^2\mathbb{E}\left[\left(g_{\|}^{i}\right)^2\right]\mathbb{E}\left[||\vec{g}_{\perp}^{i}||^2\right]$$
$$= \frac{\sigma^4}{n}||\nabla F||^2 \cdot 1 \cdot (d-1)$$

which concentrates around 0 when choosing the number of children n to be high enough.

The following represents pseudo-code for the SGD:

```
Generate vector xParents of length numParents, each in
 (xMinConstraint,xMaxConstraint);
Generate vector yParents of length numParents, each in
 (yMinConstraint,yMaxConstraint);
while (iterations ≤ maxIterations) AND (mean(penalty) ≥ threshold) do
    xChildren = childrenGenerate(xParents, iteration, sigma, delta);
    yChildren = childrenGenerate(yParents, iteration, sigma, delta);
    xShadow = optimizeX(yChildren);
    yShadow = optimizeY(xChildren);
    childPenalty = findChildPenalty(xChildren, yChildren, xShadow, yShadow);
    parentPenalty = findParentPenalty(xParents, yParents);
    xParents = nextGen(xChildren, xParents, childPenalty, parentPenalty);
    yParents = nextGen(yChildren, yParents, childPenalty, parentPenalty);
    iterations += 1
end
```

4.2 Results

When applying the Stochastic Gradient Descent (SGD) model to example 1, we find similarly good solutions as we did with the EIA, as demonstrated by Figure 4.1. We only plot points with a penalty$\leq$ 0.1, allowing us to clearly see the solutions set. Out of the 10,000 points initialized in the set, 7561, or 75.61% converged onto the solution set with a penalty ≤ 0.1.

The SGD model was not extremely accurate in terms of proportion of points that converged. This is likely due to the agents operating in isolation. In this model, agents initialized

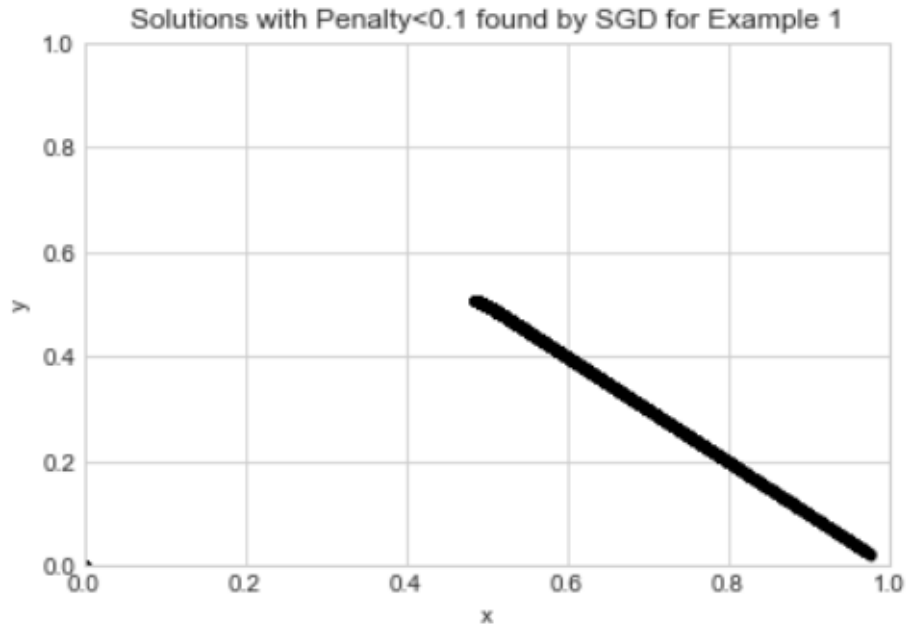

Figure 4.1: Solutions found by SGD for Example 1

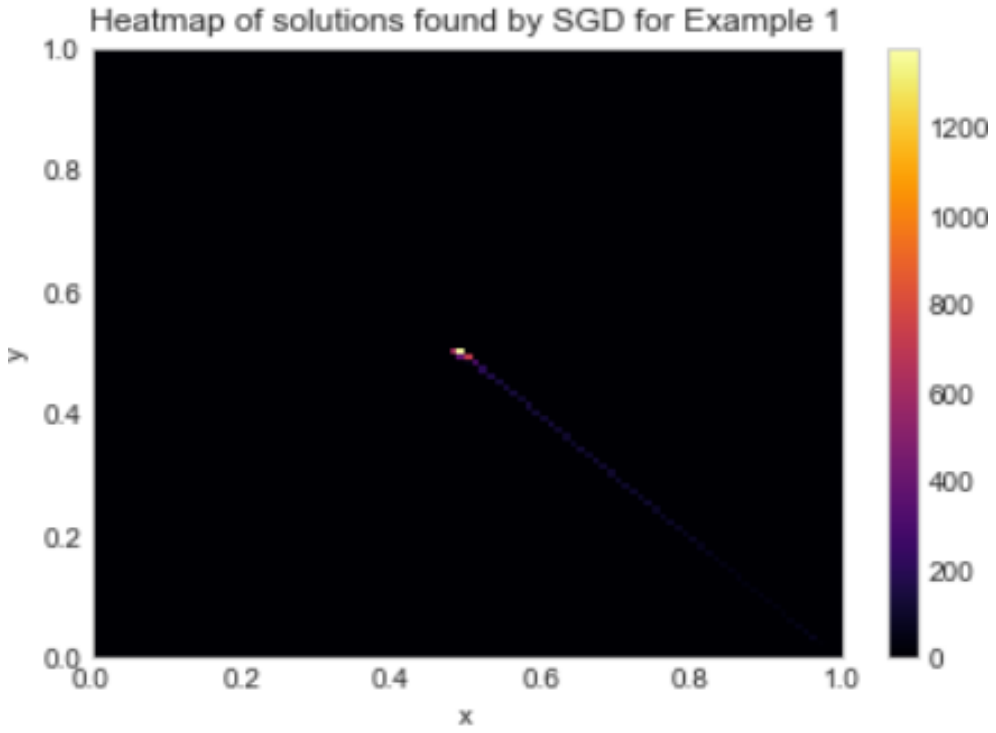

Figure 4.2: Heatmap of solutions found by SGD for Example 1

with poor fitness levels must travel long distances in order to converge onto the solution set, and their travel is capped by the increasingly strong dampening effect applied by $\alpha^{generation}$. As such, many of these agents do not make it all the way to the solution space before the $\alpha^{generation}$ becomes too low a value to move large enough distances. When observing the Heatmap shown in Figure 4.2, we see that the solutions are biased toward the center of the space. This is due to the uniformly random initialization generating many agents in the upper left, upper right, and lower left regions of the space. Agents will walk in the approximate direction that decreases their penalty the most, and so the nearest solution point for all agents in these spaces will be toward the center of the overall space. In order to converge near $(1, 0)$, an agent must be initialized nearby, otherwise there is likely to be a closer point on the solution set that is closer to $(0.5, 0.5)$.

Recall that Example 2 has the same shared constraints as Example 1, but with the objective functions depending on the strategies of both players. When applying the SGD model to example 2, we also achieve good results, as demonstrated in Figure 4.3 and Figure 4.4. Out of the 10,000 points found by the SGD algorithm, all but one (9999, 99.99%) had a penalty ≤ 0.1. The SGD model finds proportionally fewer solutions than the EIA model in example 2 for similar reasons as in example 1. The SGD algorithm is proportionally far more successful in example 2 than example 1, due to the fact that the solution set spans a greater proportion of the search space than in example 1. For this reason, we observe that the point $(0.66, 0.33)$ is by far the most commonly found solution, and that as you trend towards $(0, 1)$ the number of solutions found becomes more sparse. This is again due to the same reason as in example 1; the agents are designed to move towards the nearest solution, and so the all agents initialized to the bottom-right of $(0.66, 0.33)$ will converge onto that same point. Overall, the dependence of the objective function on both player's decisions does not appear to affect the ability of this algorithm to find the solution set.

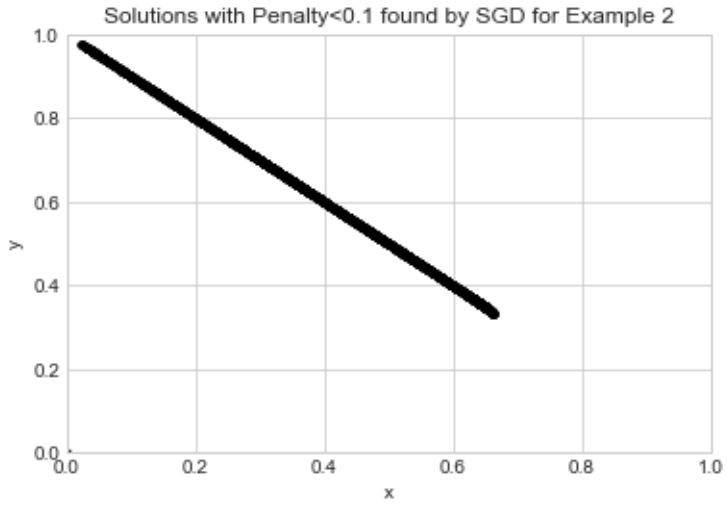

Figure 4.3: Solutions found by SGD for Example 2

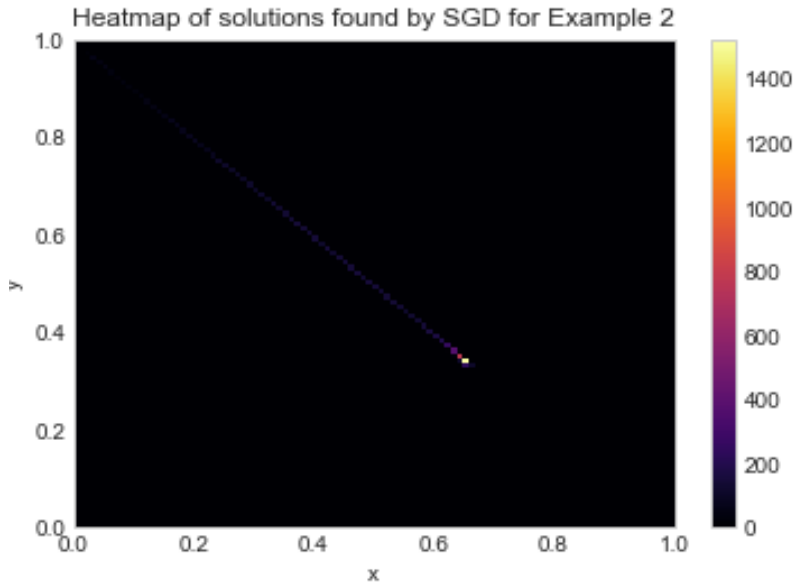

Figure 4.4: Heatmap of solutions found by SGD for Example 2

Recall that Example 3 introduces a key change in that the constraints are non-linear, resulting in a non-linear solution set. Despite this change, the SGD algorithm is very successful when solving this problem, as demonstrated by Figure 4.5 and Figure 4.6. The algorithm does not rely on any linearity in its method, and so this change does not affect the quality of the results. Out of 50,000 points, 47,342 (94.68%) converged onto the solution set within

a penalty ≤ 0.1. This is because despite the fact the solution set is much larger than in example 1, the set is much closer to the border, and so there are still some agents initialized who will be unable to reach the solution set before the $\delta^{generations}$ term becomes too powerful to take large enough steps reach the solution. This is especially true for points close to $(0, 0)$.

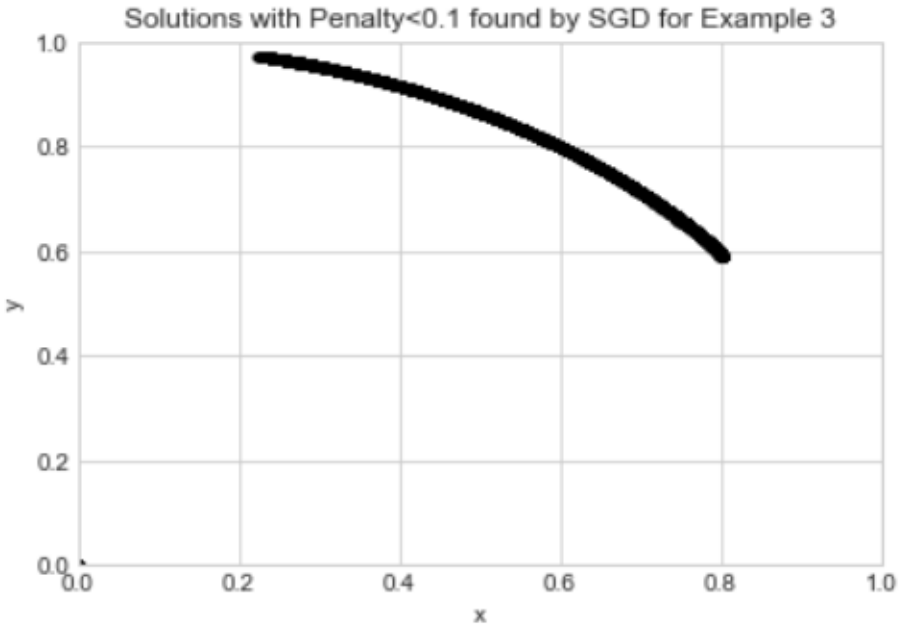

Figure 4.5: Solutions found by SGD for Example 3

For Example 4, the River basin example, recall that we have a Rosen point in addition to the linear solution segment. We treat this as we would any other question, allowing the algorithm to find the Rosen Point on its own. We see in Figure 4.7 and Figure 4.8 that the algorithm has no trouble finding both the linear solution set as well as the Rosen point. This is simply due to the fact that agents are simply attracted to the nearest solution, and so their behaviour is determined largely by where they are initialized and by the location of the nearest solution, and they remain largely unaffected by solutions farther away. Unfortunately, out of the 10,000 agents initialized, only 376 (3.76%) converged onto a points with penalty ≤ 0.1. This is likely due to both the small size of the solution set relative to other problems, and due to the location of the set being so near the global bounds. Both of these factors

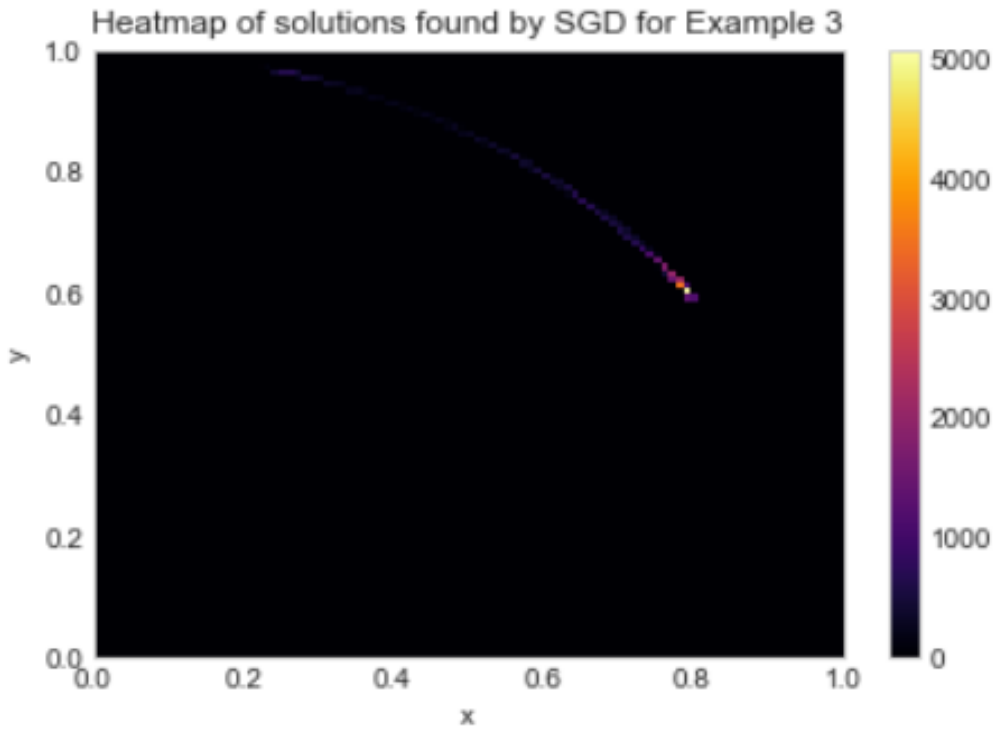

Figure 4.6: Solutions found by SGD for Example 3

make it less likely that agents will be initialized near enough to a solution to have the time to converge before the $\alpha^{generation}$ term inhibits their movement too much. In this problem, the centering bias in solutions appears less obvious than in others. This may be due to the overall low number of total points found affecting the heatmap scale, as well as the overall small size of the set and proximity to the border making them all similarly difficult for agents to locate.

Recall that Example 5 is much more complex, in that not only does it add a third dimension z, but it also remains a 2-player game, resulting in player 1 controlling both x and y. This requires 2-dimensional optimization to evaluate the shadow point coordinates for player 1. Our golden section search algorithm used by the SGD model can only handle single-variable optimization, and the ability to run our code in parallel is essential to our ability to evaluate a large enough number of points to have an understanding of our solution

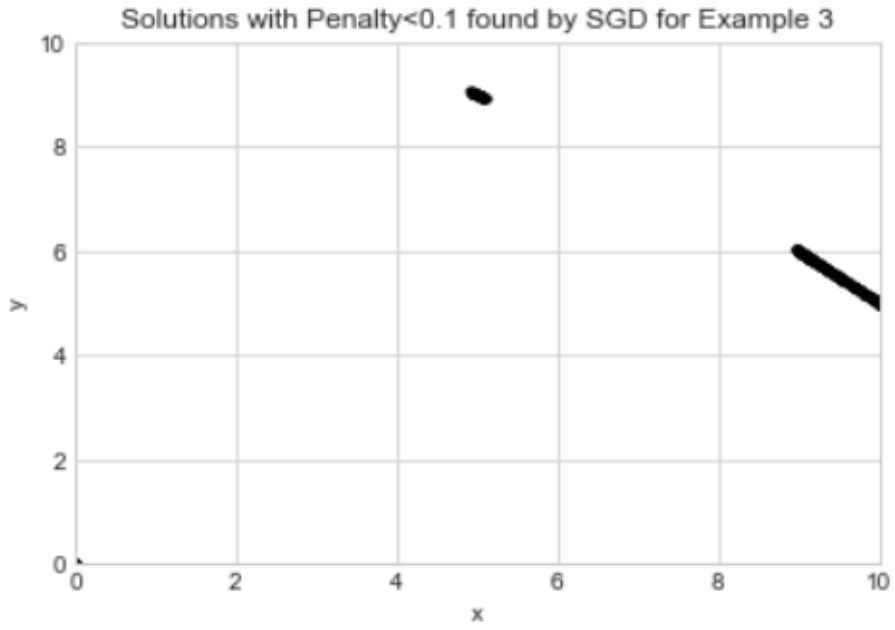

Figure 4.7: Solutions found by SGD for Example 4

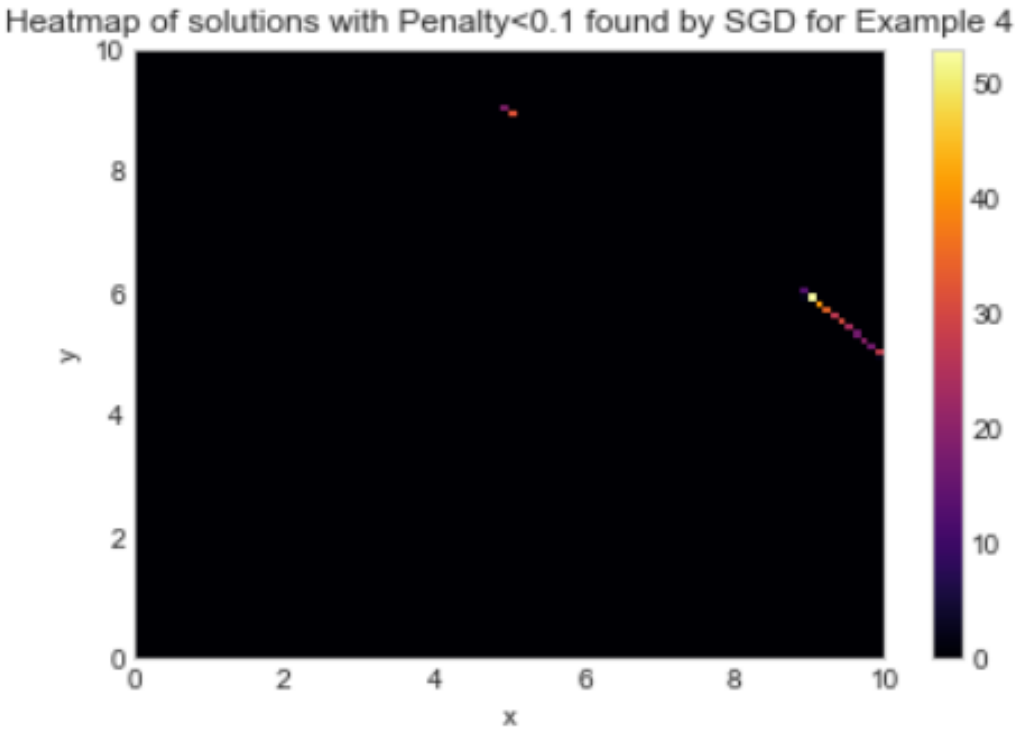

Figure 4.8: Solutions found by SGD for Example 4

set. For this reason, the SGD model has not been applied to this example. With the implementation of a 2-dimensional optimizer that runs in parallel, we would expect the SGD model to perform well on this example.

In Example 6, the river basin problem, we now have 3 players to accompany our 3 variables. Since each player controls only a single variable, we no longer require multi-variable optimizers, and our golden section search algorithm is thus viable once again. Unfortunately, the SGD algorithm still struggled at finding our solution set for this problem. In total, of the 10,000 agents placed in the space, only 109 converged onto a point with penalty ≤ 0.1. Recall from Section 3.2 that our solution set is given by a plane. The SGD algorithm was only able to find the lower y and bottom z boundaries of the solution set, without finding almost any interior points. The lack of interior points may be due to the behaviour of the search algorithm. Parents search their vicinity for points that lower their penalty, however if a point already lies on a solution, there is no benefit to moving. For this reason, points that find a boundary have no motivation to move, and thus are likely to stay on the boundary, restricting them from finding the interior of the set. It is strange that the algorithm does not find the upper y or z boundaries of the solution set. More investigation would need to be done to find out why this is.

Recall that Example 7 is a 3-player generalization of example 3. It features objective functions depednant on other player's strategies, and objective functions dependant only on the player's own strategy. It also features a combination of both linear and non-linear constraints, leading to both linear and non-linear segments in the solution set.

In Figure 4.10 we observe that the SGD algorithm was very successful at finding solutions to this GNEP, but still lacked consistency. Of the 10,000 agents initialized in the search

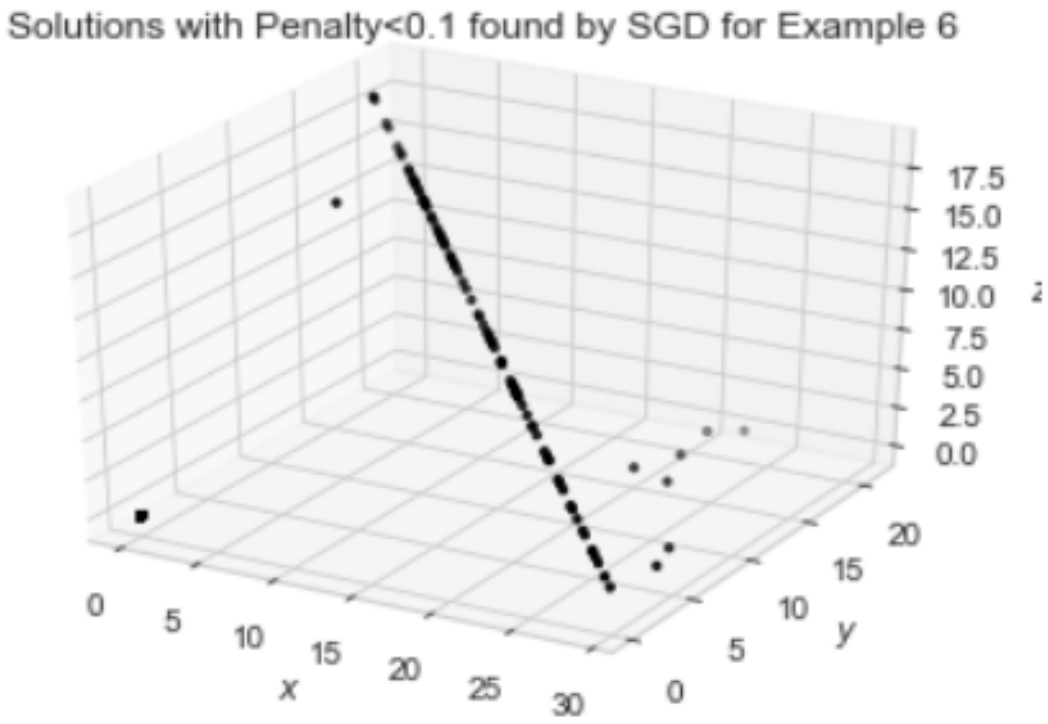

Figure 4.9: Solutions found by SGD for Example 6

space, only 1959 (19.59%) converged onto points with penalty ≤ 0.1. While proportionally few of the agents found solutions, the entire solution set is well represented by the successful agents. This is due to only a small proportion of the solution set being near the bounds. The lower proportional success rate of this algorithm as compared to other GNEPs studied, example 3 in particular, may be attributed to the fact that example 7 is in 3 dimensions, vastly increasing the size of the space needed to be searched to find solutions.

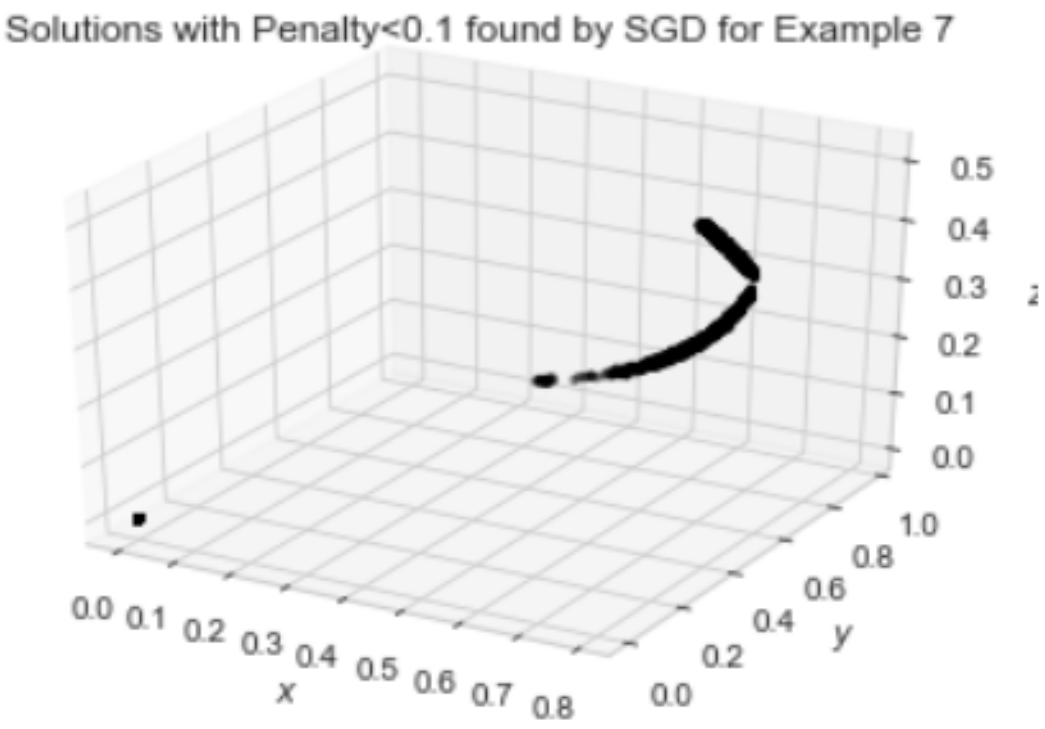

Figure 4.10: Solutions found by SGD for Example 7

Chapter 5

Comparison, conclusion and future work

In this chapter we will discuss and compare the strengths and weaknesses of each algorithm, and where each may be used more appropriately. We will also discuss the limitations of each algorithm, and potential ways to remedy these. Lastly, we will discuss ways to further optimize the performance of these algorithms.

5.1 Comparison and Limitations

As seen throughout Chapters 2 and 3, neither of these algorithms act as a 'jack of all trades'. The evolutionary inspired algorithm performs well when there are linear constraints, but cannot handle problems with nonlinear constraints and solution sets due to the linear regression step in the algorithm. This can potentially be remedied by introducing non-linear regression in place of the linear one, but more trials would need to be conducted to evaluate if this is possible, and whether it can be done efficiently. Alternatively, the SGD model performs equally well on both linear and non-linear constraints and solution sets without further work

needed.

Both the EIA and the SGD algorithm struggle to find solutions close to the boundary of the feasible set. One potential remedy for the EIA to employ would be to alter the new point generation function to first make a binary choice in each dimension about whether to increase or decrease the value, and only afterwards choose a value within the search space. For the SGD algorithm, the centering bias is due to the uniform initialization and the location of the solution set. While we do not suggest a specific way to remedy this, we suggest utilizing very large numbers of agents so as to more completely sample the feasible set, and thus while the majority of agents may not converge, enough agents do converge that the user has a full understanding of the solution set regardless of the number of discarded points.

The SGD algorithm struggled to find interior solution points to larger solution sets such as planes. In these examples, the algorithm stops on the border of the set, and has no motivation to move since the penalty is already minimized at 0. This would likely be replicated in solution sets such as these in higher dimensions as well, such as cubes. In order to get an understanding of whether the solutions found represent the full span of the solution set or just the borders, the user would have to manually initialize agents within the borders of the solutions found and observe their behaviour. Conversely, due to the large number of models generated and presented in union by the EIA, it has no problem capturing planes. It would be interesting to observe the behaviour of the EIA when solving for a higher dimensional set such as a cube.

When handling segmented solution sets that do not lie on the same line, such as that demonstrated in Example 4, the EIA algorithm struggled. This was due to the core assumption of the algorithm that all solutions lie on the same function. In the current version of the model, we assume all solutions lie on the same line, however this same problem exists when assuming higher degree curves as well. This can only be remedied by manually penalizing solution segments such that only a single curve remains, allowing the algorithm to solve for

that curve. Obviously this is much more difficult to do if one does not know the solution set to begin with, thus this is a large scale problem when encountering problems with these types of segmented solution sets. The SGD model utilizes independent agents each looking for any single solution. As such, it is irrelevant what the shape or relationship between solutions are, simply their location relative to each agent. For this reason, the SGD model performs equally well on these types of problems as on ones with strong structured solutions.
The SGD algorithm is currently unable to handle problems with multivariable optimization, due to the golden section search. This can be remedied with the implementation of a multi-variable optimizer that can run in parallel. Since the EIA does not run in parallel, it is easily able to make use of built in python libraries to perform these types of optimization.

Taking these factors into mind, under the current versions of the algorithms, the EIA is more desirable in cases where the solution set is linear, continuous and in higher dimensions, such as in Example 6, or in problems like Example 5 where multivariable optimization is required. The SGD algorithm is more desirable in cases where the solution set is non-linear such as Examples 3 and 7, or discontinuous such as in Example 4. In simple cases where the solution set is a continuous line segment requiring only single-variable optimization, either algorithm may be used to achieve similar results (Examples 1 and 2).

5.2 Future Work

In addition to addressing the limitations of the algorithms, there are ways for us to further optimize the performance of both algorithms. In order to further optimize the EIA, two significant factors must be investigated. We must observe the effect of both number of points in each generation, and the number of points selected in each generation. In this paper, we use 20 points per generation, selecting the top 10 with each iteration of the algorithm. It

is likely that when increasing the number of search points, we also increase the accuracy of the algorithm, as the algorithm has more points to choose from. This has the unwanted side effect of increasing the computational power required to solve each generation. It is possible that when doing so however, it also decreases the number of iterations required to converge onto a solution set. For this reason, it would be interesting to find the point of 'diminishing returns', and to optimize the number of points for maximum computational efficiency. This value would likely be different depending on the complexity of the problem. Additionally, raising the number of selected points decreases the 'power' of any individual selected point in the linear regression. This would improve the algorithm's ability to work through outlier points and may improve the accuracy of the regression. This too would need to be optimized to the specific question the algorithm is used for, as well as the number of generated points. It would be interesting to investigate a possible optimal ratio of generated points to selected points. Other variables that can be changed are the δ values we utilize to decrease the size of the search space over time, similar to a step size. δ values closer to 1 slow down the algorithm, requiring more iterations to converge onto the solution set, but make it much more likely the algorithm will find good solutions along the way. Lower δ values restrict the search space much faster, resulting in a much faster algorithm, but one that risks converging too quickly and not finding the solution set before restricting the search space to a region that does not contain the solution.

The stochastic gradient descent method can also be further improved and optimized, beyond the limitations mentioned previously. One major variable that can be optimized according to the problem is the step size α. A small α value results in smaller steps. The advantage of this is that the algorithm is very unlikely to 'jump over' a solution and miss it. Unfortunately, since the α dampened by $0.99^{generation}$, an alpha too small will impede some agents from being able to travel all the way to a solution point before the steps become too

small to move there. Alternatively, a large value of α allows the agents to travel further distances before their movement is significantly impeded, however it also runs the risk of taking steps past the solution points and missing them. This value must be varied according to the problem set, and is especially dependent on the size of the space needed to be searched. Since we have so many agents working in parallel, it may be ok to the user if many or most of the agents never converge onto solutions, as long as enough agents do that the user gets a good understanding of the set. Additionally, one can change the rate at which *alpha* is dampened by changing the dampening factor to another value. A value closer to 1 will slow down the rate at which α is dampened, allowing for more of the space to be searched, but slowing down the time until convergence. A lower value will force α to zero much faster, but may not give the agent a chance to find the set before it stops moving. Once again, the optimal value will depend on the specific problem for which the algorithm is utilized. Similar to the number of generated and selected points being optimized in the EIA, one could find the optimal number of children to generate to maximize the speed of the algorithm while minimizing the computational load. It would also be interesting to investigate the different ways to decide the step size and direction based on the 'children'. For example, one could select the best child, or take a weighted average of the best children instead of a weighted average of them all.

5.3 Conclusion

In this paper we introduced a novel penalty function, similar to the Nikaidô-Isoda function [1]. We utilized this function to motivate the behaviour of two heuristic models. The evolutionary-inspired algorithm selects the top points found in each iteration, performs a linear regression, and uses that regression to generate replacement points within a space dependent on the current iteration count and the proximity to the global bounds of each

dimension. The stochastic gradient descent algorithm initialized agents randomly in the space, and utilized the penalty function and stochastic gradient descent to find solutions. The evolutionary inspired algorithm (EIA) was very successful at finding linear solution sets, including those in higher dimensions such as planes. It struggled at finding points in the solution set that are near a dimension's global bounds due to the way it generates replacement points. Due to the linear regression, the EIA model was unable to find nonlinear solution sets, a problem which may be remedied by using higher-order regression. The stochastic gradient descent (SGD) algorithm performed very well on both linear and non-linear solution sets when those sets were simply lines or curves, but struggled when finding higher dimensional surfaces. In these problems, the algorithm got 'stuck' on the border of the solution set, as it had no 'motivation' to enter the interior of the set. Both of these algorithms show potential to be used successfully in a much larger class of problems than demonstrated here, but require further optimization in their parameters such as step size or number of points initialized in order to improve the results. These parameters are likely going to have different optimum values dependant on the problem the algorithms are used on.

REFERENCES

[1] H. Nikaidô and K. Isoda, "Note on non-cooperative convex games," *Pacific Journal of Mathematics*, vol. 5, no. 5, 1955.

[2] R. B. Myerson, *Game Theory - Analysis of Conflict*. Harvard University Press, 1997.

[3] G. Debreu, "A social equilibrium existence theorem," *Proceedings of the National Academy of Sciences*, vol. 38, no. 10, pp. 886–893, 1952.

[4] J. F. Nash, "Equilibrium points in n-person games," *Proceedings of the National Academy of Sciences*, vol. 36, no. 1, pp. 48–49, 1950.

[5] C. A. Holt and A. E. Roth, "The nash equilibrium: A perspective," *Proceedings of the National Academy of Sciences*, vol. 101, no. 12, pp. 3999–4002, 2004.

[6] J. F. Nash, "The bargaining problem," *Econometrica*, January 1950.

[7] A. R. Ofer and A. V. Thakor, "A theory of stock price responses to alternative corporate cash disbursement methods: Stock repurchases and dividends," *The Journal of Finance*, 1987.

[8] T. Karabiyik, O. Akal, and E. Aktas, "Sustainable equilibirum in a stock market: Agent-based modeling with evolutionary game theory applied to traders," *International Journal of Business, Accounting and Finance*, 2017.

[9] R. Bshary and R. F. Oliveira, "Cooperation in animals: toward a game theory within the framework of social competence," *Current Opinion in Behavioural Sciences*, vol. 3, pp. 31–37, 2015.

[10] C. Hadjichrysanthou and M. Broom, "When should animals share food? Game theory applied to kleptoparasitic populations with food sharing," *Behavioral Ecology*, vol. 23, pp. 977–991, 05 2012.

[11] J. M. Smith, "Game theory and the evolution of behaviour," *Behavioral and Brain Sciences*, vol. 7, no. 1, p. 95–101, 1984.

[12] G. Hardin, "The tragedy of the commons," *Journal of Natural Reources Policy Research*, pp. 243–253, 2009.

[13] I. Parrachnio, A. Dinal, and P. Fioravante, "Cooperative game theory and its application to natural, environmental and water resource issues: 3. application to water issues," *World Bank*, 2006.

[14] E. Oftadeh, M. Shourian, and B. Saghafian, "An ultimate game theory based approach for basin scale water allocation conflict resolution," *Water Resources Management*, vol. 31, no. 1, pp. 4293–4308, 2017.

[15] Q. Han, G. Tan, X. Fu, Y. Mei, and Z. Yang, "Water resource optimal allocation based on multi agent game theory of hanjiang river basin," *Water*, 2018.

[16] L. Lin, C. Huang, and L. Zhao, "Application of cooperation game theory in ractive power market of hydraulic-power plant," *Journal of Coastal Research Special Issue No. 93: Advances in Water Resources and Exploration*, pp. 578–581, 2019.

[17] K. Yamamoto, "A comprehensive survey of potential game approaches to wireless networks," *IECE Transactions on communications*, vol. E98-B, pp. 1804–1823, september 2015.

[18] L. R. Foulds, "The heuristic problem-solving approach," *The Journal of the Operational Research Society*, vol. 34, no. 10, pp. 927–934, 1983.

[19] J. B. Rosen, "Existence and uniquness of equilibirum points for concave n-person games," *Econometrica*, vol. 33, pp. 520–534, 1965.

[20] J. B. Krawczyk and S. Uryasev, "Relaxation algorithms to find nash equilibria with economic applications," *Environmental Modeling and Assessment*, vol. 5, pp. 63–73, 2000.

[21] D. Aussel and J. Dutta, "Generalized nash equilibrium problem, variational inequality and quasiconvexity," *Operations Research Letters*, vol. 36, no. 4, pp. 461–464, 2008.

[22] F. Facchinei, A. Fischer, and V. Piccialli, "On generalized nash games and variational inequalities," *Operations Research Letters*, vol. 35, no. 2, pp. 159–164, 2007.

[23] T. Migot and M.-G. Cojocaru, "A decomposition method for a class of convex generalized nash equilibrium problems," *Optimization and Engineering*, pp. 1–27, 2020.

[24] T. Migot and M.-G. Cojocaru, "A parametrized variational inequality approach to track the solution set of a generalized nash equilibrium problem," *European Journal of Operational Research*, vol. 283, no. 3, pp. 1136–1147, 2020.

[25] P. T. Harker, "Generalized nash games and quasi-variational inequalities," *Theory and Methodology*, 1988.

[26] E. Wild, *A Study of Heuristic Approaches for Solving Generalized Nash Equilibrium Problems and Related Games*. PhD thesis, University of Guelph, August 2017.

[27] J. F. Nash, "Non-cooperative games," *Annals of Mathematics*, vol. 54, no. 2, pp. 286–295, 1951.

[28] K. J. Arrow and G. Debreu, "Existence of an equilibrium for a competitive economy," *Econometrica*, vol. 22, no. 3, pp. 265–290, 1954.

[29] K. Nabetani, "Variational inequality approaches to generalized nash equilibrium problems," tech. rep., Department of Applied Mathematics, School of Informatics, Kyoto University, 2008.

[30] M. Hintermuller, T. Surowiec, and A. Kammler, "Generalized nash equilibrium problems in banach spaces: Theory, nikaido-isoda-based path-following methods, and applications," *SIAM Journal on Optimization*, vol. 25, no. 3, pp. 1826–1856, 2015.

[31] F. A. B. da Silva and H. Senger, "Scalability analysis of embarassingly parallel applications on large clusters," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, 2010.