EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Combinatorial Optimization Research Group

Inverse Optimisation of Integer Programming Games

Master thesis Data Science and Artificial Intelligence

mr. R.I. (Robin) Verhoef

Supervisors:
dr. B.M.L. (Bart) Smeulders
dr. B.M.P. (Bart) Jansen
dr. L. (Layla) Martin

Final version

Eindhoven, June 2024

Abstract

This study investigates the possibilities for applying inverse optimization to an integer program and two integer programming games. The current inverse optimization methods for combinatorial optimization problems focus on learning the payoffs of instances. X extensions to these methods were developed. First, the current methods were adjusted to also learn with weights of instances. Secondly, existing methods were applied to new problems with new structures. Finally, all combinations of these applications were explored and compared. Numerical analyses demonstrate the resources required for different tasks and the impact of two different inverse optimization goals.

Keywords: Combinatorial optimization, Inverse optimization, Integer programming games, Knapsack problem, Knapsack packing game, Critical node game

Preface

This thesis completes my Master's program in Data Science and Artificial Intelligence at the Eindhoven University of Technology. I am very happy with the subject of my thesis, even if it is a bit niche for my master. I followed my first courses in optimization during my bachelor and especially combinatorial optimization is a subject that I find interesting. It also happens to be a quite practical field, where applying existing methods to new problems is a challenge and that is exactly what I did in this thesis.

I would like to thank a few people here, starting with the members of my assessment committee. First, my supervisor Bart Smeulders for guiding me through this process. After his course *Optimization for Data Science* and the *SPOR seminar*, I knew for sure that I wanted to do my thesis in this field. Bart gave me a lot of freedom to explore the literature and find my own topic and research question. During our meetings, Bart asked critical questions to ensure that I knew what I was doing and helped me when I was stuck. Second, I would like to thank Layla Martin for taking the time to discuss her paper [13], which is one of the foundational works of my thesis, and helped me decide on the focus of my thesis. I would also like the third member of my assessment committee, Bart Jansen, in advance for accepting to be a part of my thesis defence.

Further, I would like to thank two of my mathematically inclined friends, Noah Keuper and Luke Visser, for reading the draft versions of this thesis with a critical and detail-oriented eye. Without them, this thesis would certainly be less readable. Luke is the reason that most of the tables were banished to the Appendix in favour of graphs. Finally, I would like to thanks my friends and colleagues for helping me if I got stuck in the writing process and my parents for supporting me during studies and before.

With this (second) master thesis, my time as a student, both at the Eindhoven University of Technology and in general, is coming to an end. I am happy with what I was able to accomplish and the people I met and worked with while studying here. For now, I will start working in The Hague, but maybe I will return in the future for a PhD.

Robin Verhoef

Eindhoven, June 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of optimisation is generally concerned with trying to solve problems by modelling them and applying mathematical methods to solve those problems. These problems are very often connected to real-world problems related to resource allocation or planning. A general way to describe problems where you can only work with whole (integer) things are integer programs (IPs). Integer programs can be used to model the routes of garbage trucks and delivery trucks in order to minimise the driving time, determine the time tables of a university such that students do not have 8 hours of lectures in a single day or decide how much of each type of fruit a store should order for the next week in order to minimise waste.

Integer programming games (IPGs) extend (IPs) to problems with multiple independent decision makers, often called players. These players are assumed to be playing simultaneously while acting rationally, with complete information of the problem and trying to get the best result for themselves (non-cooperative) [27]. IPGs can be used to model the cat-and-mouse game of attackers and defenders of a server network where the goal of the defender is to keep the network running at minimal cost [17], project planning of competing companies where the choices of the competition impacts your own profitability [9] and coordinating international organ transplants by making it beneficial for all parties to share all possible patients [3, 4]. In all of these cases, the challenge is to try to find strategies for all the players with which they are all satisfied. Solving IPGs requires finding stable solutions where it is not feasible for players to change their strategy in order to gain an advantage. Finding stable solutions for the IPGs I will consider in this thesis currently requires solving individual problems of players repeatedly until either an equilibrium is found or no possible solutions are left over.

Inverse optimisation is a method for learning information about an optimisation problem by observing solutions to variations of a problem with some of its characteristics [23]. This can be useful to determine how a problem should be adjusted to obtain different results, or to determine the constraints of a problem or its objective function. For example, inverse optimisation can be used to change the usage of a road network. Imagine the roads in a major city and their usage. If you know their usage and the time it takes to drive certain routes, inverse optimisation can be used to look for the changes required to the driving times in order to change the usage of specific roads. For instance, the speed limit could be lowered or traffic lights could be added to make a certain route more costly to use.

Inverse optimisation has recently been applied to a location selection IPG [13], to determine a small number of parameters essential for the payoff function and there is also quite some literature on applying inverse optimisation toIPs ([36, 24, 32]). For IPGs, inverse optimisation could be used to learn the payoffs of players or their constraints when deciding their

strategies. Inverse optimisation can also be used to determine how payoffs or constraints should be changed in order to make non-cooperative players work for the common good by alligning it with their own goals.

1.1 Problem Definition

The work on applying inverse optimisation to IPGs has so far been limited to problems with a low number of parameters and a low number of observations. The most recent work looks at the Location Selection Game, which has 1 global parameter and 1 parameter per player [13], and aims to learn these payoff parameters. What happens in case of problems with a higher number of parameters per player, like the classical Knapsack Problem, the Knapsack Packing Game and the Critical Node Game? This is the focus of my research. I will be investigating how we can learn about these problems, what we can learn and what the limits are.

1.2 Research Questions

To guide my research into inverse optimisation of integer programming games and to determine the scope, the guiding research question for this thesis is:

RQ: How can inverse optimisation be applied to integer programming games with a high number of parameters?
There is already a method for inverse optimisation for integer programming games [13], but it focuses on instance parameters. There is also a method for learning player characteristics [36], but this is made for integer programs. Can these method be applied to learn player characteristics of integer programming games? This is the main question of my thesis. For structuring my research, I further specified the main research question with four subquestions.

SQ1: Can inverse optimisation be applied to integer programs with a high number of parameters?
Solving the problem of inverse optimising IPGs will require solving the same problem for a single player, which is an integer program. For a method to be successful for IPGs, it will also have to be able to solve integer programs. Thus, the first step is to see how well the method from [36] works if the number of parameters increases.

SQ2: Which inverse optimisation methods can be applied to integer programming games?
It might be the case that some methods from **SQ1** are somehow not suited for IPGs. I will apply the methods of [36] and [13] to see if they both work for (new types of) integer programming games.

SQ3: How can inverse optimisation be applied to specific integer programming games with a high number of parameters?
To see if the methods which result from **SQ2** work on integer programming games with a high number of parameters, I chose two problems with those characteristics. The first problem is the knapsack packing game which has a high number of parameters per player and is based on the classic knapsack problem. The second problem is the critical node game, where the two players are competing and the outcome for both players is intertwined with a complete interaction matrix.

SQ4: Are these methods effective on practical instances of the specific integer programming games?
Once the methods show promise, I will evaluate their performance, both in terms of accuracy as well as in computation time, to determine if these methods can be used on problems with a level of complexity that makes them interesting in practice.

1.3 Preliminaries

I will be using the following conventions in the equations and definitions, with vectors $a, b, x \in \mathbb{R}^n, c \in \mathbb{R}^m$ and matrix $A \in \mathbb{R}^{m \times n}$:

- $ab = a \cdot b \in \mathbb{R}$ denotes the dot product of two vectors, resulting in $\sum_{i=1}^{n} a_i b_i$;
- $a \odot b \in \mathbb{R}^n$ denotes the element-wise multiplication of the two vectors, resulting in $[a_1 * b_1, \ldots, a_n * b_n]$;
- $a+b \in \mathbb{R}^n$ denotes the element-wise addition of two vectors, resulting in $[a_1+b_1, \ldots, a_n+b_n]$;
- $\sum a = \sum_{i=1}^{n} a_i \in \mathbb{R}$ denotes the sum of the values of vector a;
- $Ax = A \cdot x \in \mathbb{R}^m$ denotes a standard matrix multiplication and results in a vector;
- A comparison between two vectors $Ax \in \mathbb{R}^m$ and $c \in \mathbb{R}^m$, like $Ax \leq c$, always compares the elements of the vectors, so $(Ax)_i \leq c_i, \forall i \in \{1, \ldots, m\}$.

1.3.1 Integer Programming

A linear program (LP) is a type of optimisation problem where the goal is to optimise a linear objective function $f(x)$ over a convex feasible set $\mathcal{X}$. Integer program (IPs) are a variant of LPs, where all variables are restricted to integers. It is also possible to apply combine integer and real valued variables in a single program, this is called a mixed integer program (MIP). In this thesis, I will focus on IP problems, but some of the solution methods add additional variables which turn the IP into a MIP.

Definition 1. Integer Program (IP)
An IP is an optimisation problem that can be described as

$$\max_x \{f(x) : Ax \leq b, x \in \mathbb{Z}^n\}$$

where $f(x) : \mathbb{Z}^n \to \mathbb{R}$ is the *objective* or *payoff* function and the matrix $A \in \mathbb{Z}^{m \times n}$ together with the vector $b \in \mathbb{Z}^m$ form the constraints.

Ideally, the result of solving an IP is an optimal solution with a finite value for the objective $f(x)$. However, depending on the constraints A and b, an IP can also have no valid solutions or an unbounded solution. Any IP with a bounded feasible set $\mathcal{X} = \{x \in \mathbb{Z}^n : Ax \leq b\}$ has a finite number of feasible solutions $x \in \mathcal{X}$. Since there is a finite number of solutions, the problem has an optimal solution $\hat{x} = \arg\max_{x \in \mathcal{X}} f(x)$ as long as the feasible set is not empty, i.e. $\mathcal{X} \neq \emptyset$ [26].

1.3.2 Solving Integer Programs

A simple method for finding the optimal solution is calculating $f(x)$ (usually of the form $px, p \in \mathbb{Z}^n$) for every solution in the feasible set $\mathcal{X}$ [26]. Unfortunately, the size of the feasible set generally increases exponentially with the number of variables of the IP, which means this method also takes exponentially more time to solve as the number of variables increases. In contrast, LPs can be solved in polynomial time using a randomised simplex algorithm [25]. The simplex algorithm [15] uses the fact that if an LP has an optimal solution, this solution will always be at an extreme vertex of the feasible set $\mathcal{Y} = \{x \in \mathbb{R}^n : Ax \leq b, \}$. Extreme vertices are vertices where some of the constraints of A are exactly satisfied, so $\mathbf{a}^i x = b_i$ for some of the rows $\{\mathbf{a}^1, \ldots, \mathbf{a}^m\}$ of the matrix A. The feasible set of an IP lacks this attribute, so the simplex algorithm can not be applied to solve IPs.

The general method for solving IPs is by relaxing the integrality constraint of the IP, which results in the LP relaxation [26]. This relaxed problem can be solved with the simplex algorithm. If the result is also a feasible solution for the IP, then the (sub-)problem is solved. If not, the result is used to separate the IP into sub-problems by excluding the infeasible solution. This usually means forcing one of the non-integer variables to either round up or down using a new constraint. The creation of sub-problems is usually implemented using a Branch-and-Bound algorithm which finds upper and lower bounds for sub-problems of the IP and uses this to find the optimal integer solution for the problem. A possible method to create sub-problems is by taking one of variables which did not have an integer value $\bar{x}_i \in \mathbb{R}$ and create one sub-problem with the additional constraint $x_i \leq \lfloor \bar{x}_i \rfloor$ and a second sub-problem with the additional constraint $x_i \geq \lceil \bar{x}_i \rceil$. More advanced methods add constraints on multiple variables at the same time. The state of the art methods for solving LPs and IPs are combined and implemented in solver programs, like Gurobi [21]. In this thesis, I will be using the gurobipy Python package of the Gurobi solver to model and solve all optimisation problems.

1.3.3 Knapsack Problem

The knapsack problem is one of the easiest NP-hard problems [26] and a very intuitive example of an IP. It can model scenarios where you have to select a subset of elements out a larger set under some constraint like capacity or time. Many problems related to planning or assignment of limited resources can be modelled using the knapsack problem [28]. The knapsack problem also often appears as a sub-problem of other NP-hard problems and is a foundation of combinatorial optimisation.

Definition 2. Binary Knapsack Problem (KP)
A KP is an IP that can be described as

$$\max_x \{px : wx \leq c, x \in \{0,1\}^n\}$$

where n is the number of items, the vector $p \in \mathbb{N}^n$ is the payoff vector, the vector $w \in \mathbb{N}^n$ is the weights vector and $c \in \mathbb{N}$ is the capacity, usually defined as a fraction of the total weight of all items $\sum w$.

KP is the standard version of the knapsack problem, see Table 1.1 for an example instance. Written out in the KP format used above it would look like

$$\max_x \{1x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 : \\ 2x_1 + 2x_2 + 3x_3 + 3x_4 + 4x_5 + 4x_6 \leq 5, x \in \{0,1\}^n\}$$

Table 1.1: Example of a KP. For $c = 5$, there are two optimal solutions: $\{0, 1, 0, 1, 0, 0\}$ and $\{0, 0, 0, 0, 0, 1\}$.

Item (i)	1	2	3	4	5	6
Weight (w_i)	2	2	3	3	4	4
Payoff (p_i)	1	2	3	4	5	6

. The length of this problem highlights the necessity of condensed notation for vector and matrix multiplications. As noted before, KP is an NP-hard problem, but not all instances are equally hard in practice. Instances where there is a high correlation between the payoff of an item and their weight are harder to solve then those where the two are less correlated [31].

There are many possible variations of the knapsack problem to model more complex problems [28]. For instance, the domain of x can be extended to allow the same item to be chosen multiple times. This is called the Multiple-Choice Knapsack Problem. If an item can be chosen an unlimited number of times, the problem turns into the Unbounded Knapsack Problem. KP can also be extended by adding more knapsacks with capacities c_k, for the Multiple Knapsack Problem.

1.4 Implementation

I implemented all problems and algorithms in this thesis using Python [19], the NumPy package [22] and the Gurobi Python package gurobipy [21]. All code can be found in the IPG-thesis Github repository. I will indicate relevant files for the implementation of problems and methods. For example, KP is implemented in `problems/knapsack_packing.py`. All files are assumed to be in the `src` folder unless stated otherwise. I used the R programming language [35] and the tidyverse package [37] to generate all the graphs in this text. The code to generate all the graphs can be found in `Plots.Rmd`.

1.5 Outline

The upcoming chapters will serve to answer my research questions. In the next chapter, I will review the literature on inverse optimisation and integer programming games to establish the building blocks for the later chapters. In Chapter 3, I will investigate inverse optimisation on integer programs with a high number of parameters and in Chapter 4 I will do the same for integer programming games. Chapter 5 contains my conclusions and summarised answers to my research questions together with my thoughts about future steps.

Chapter 2

Literature Review

In this chapter, I will introduce the two building blocks of my thesis: integer programming games (IPGs) and inverse optimisation. For each, I will introduce the concept together with a mathematical definition as well as concrete problems, which will come back in the next chapters.

2.1 Integer Programming Games

IPs have a single decision-maker solving a single problem. If a problem involves n decision-makers who have to make decisions which affect each other, we can still model this as an IP if the decision makers are willing to cooperate. Let the operator $(\cdot)^i$ represent the i-th component of $(\cdot)$. The operator $(\cdot)^{-i}$ is the inverse of $(\cdot)^i$ and represents the elements of $(\cdot)$ excluding the i-th component. Each decision maker $i \in \{1, \dots, n\}$ has a linear payoff function $f^i(x^i, x^{-i}) : \mathbb{Z}^m \times \mathbb{Z}^{(n-1)\times m} \to \mathbb{R}$ and a feasible set $\mathcal{X}^i = \{x^i \in \mathbb{Z}^m : A^i x^i \leq b^i\}$, where m is the number of decision variables and matrix $A^i \in \mathbb{Z}^{k_i \times m}$ together with vector $b^i \in \mathbb{Z}^{k_i}$ form k_i linear constraints for decision-maker i. Let $x = \{x^1, \dots, x^n\}$ be the combined strategies of all decision-makers, where $x \in \mathcal{X} = \{\mathcal{X}^1, \dots, \mathcal{X}^n\}$. Then, we can combine the problems of all decision-makers into a single IP

$$\max_x \{\sum_{i=1}^{n} f(x^i, x^{-i}) : x \in \mathcal{X}\}.$$

However, what if the decision-makers are not cooperative? This is where IPGs come in. IPGs extend IPs to allow for multiple independent decision-makers, often called *players*. IPGs were first introduced in [27] and can be used to solve games where each player only cares about their own outcome, which usually means maximising their own objective function. They have been used to model a wide range of problems, like kidney exchange programs [3, 4], knapsack packing games [18], network formation games [18] and critical node games [17]. Like in the example above, the payoff functions of players in an IPG are connected, which means that the optimal strategy of each player depends on the chosen strategies of others. In the example above, the goal was to optimise the collective payoff function, but in an IPG each player wants to have the best deal for themselves. In the IPGs I will consider, all players choose their strategy simultaneously, meaning that they cant observe the actions of their opponents [7]. Players also get complete information of payoffs and constraints [7]. The players in an IPG are:

- *non-cooperative*: players have payoff functions $f^i(x^i, x^{-i})$ which depend on the actions of other players with possible negative interactions; and
- *rational*: each players will maximise their own payoff function $f^i(x^i, x^{-i})$ and choose x^i accordingly [7].

Definition 3. Integer Programming Game (IPG)
An IPG is a simultaneous, non-cooperative game with complete information and rational players where each player $i \in \{1, \dots, n\}$ solves the optimisation problem

$$\max_{x^i} \quad \{f^i(x^i, x^{-i}) : x^i \in \mathcal{X}^i\}, \quad \mathcal{X}^i = \{x^i \in \mathbb{Z}^m : A^i x^i \leq b_i\},$$

where m is the number of decision variables and matrix $A^i \in \mathbb{Z}^{k_i \times m}$ together with vector $b^i \in \mathbb{Z}^{k_i}$ form k_i linear constraints for player i which define the feasible set $\mathcal{X}^i$. $\mathcal{X}^i$ is assumed to be finite, with lower and upper bounds for each variable. The payoff function $f^i(x^i, x^{-i}) : \mathbb{Z}^m \times \mathbb{Z}^{(n-1)\times m} \to \mathbb{R}$ is the payoff of player i if they play $x^i \in \mathcal{X}^i$ and their opponent(s) play(s) $x^{-i} \in \mathcal{X}^{-i}$.

The Nash equilibrium (NE) [30] is often used to define the solutions of IPGs [27]. In a NE, no player can change their strategy and also increase their payoff $f^i(x^i, x^{-i})$. In the simplest version of the NE, each player has to decide on a single strategy. This is called a Pure Nash Equilibrium.

Definition 4. Pure Nash equilibrium (PNE).
A PNE for an IPG is a vector of pure strategies $\bar{x} = \{\bar{x}^1, \dots, \bar{x}^n\}$ such that, for each player $i = 1, \dots, n$,

$$f^i(\bar{x}^i, \bar{x}^{-i}) \geq f^i(\hat{x}^i, \bar{x}^{-i}), \quad \forall \hat{x}^i \in \mathcal{X}^i.$$

The PNE is not the only version of the NE. It is also possible to define a NE for distributions of strategies, called a Mixed Nash Equilibrium (MNE). In an MNE, each player has to decide on a mixed strategy $\sigma^i \in \Delta(\mathcal{X}^i)$ which is a probability distribution over all possible strategies $\mathcal{X}^i$. Finite IPG instances, like the types of IPGs I will be covering in this thesis, are not guaranteed to have a PNE [8]. Calculating an MNE can only be done with specific methods and MNEa are not very straight-forward to interpret [13]. Instead of using MNEa when dealing with problem instances which lack a PNE, it is possible to relax the requirements of the PNE and find an approximate NE [17].

Definition 5. Approximate Nash Equilibrium (Φ-NE). A Φ-NE for an IPG is a vector of pure strategies $\bar{x} = \{\bar{x}^1, \dots, \bar{x}^n\}$, such that, for a given $\Phi \in \mathbb{R}_+$ and players $i = 1, \dots, n$,

$$f^i(\bar{x}^i, \bar{x}^{-i}) + \Phi \geq f^i(\hat{x}^i, \bar{x}^{-i}) \quad \forall \hat{x}^i \in \mathcal{X}^i.$$

The constant $\Phi \geq 0$ represents the maximum difference in payoffs between the chosen strategy and any alternative strategy. If $\Phi = 0$, then the Φ-NE is a PNE.

2.1.1 Solution Methods

Deciding if an IPGs has a PNE is $\sum_2^p$-complete[1] and if an equilibrium exists, computing it is at least PPAD-hard[2] [9]. This is not surprising, as solving an IPG requires computing the

[1] NP with NP oracle, meaning that an IPG is solvable in polynomial time using a nondeterministic Turing machine with a nondeterministic oracle running in polynomial time.

[2] Polynomial Parity Argument (Directed), a subclass of Total Function NP, which is a generalisation of NP using functions.

responses of players, which are generally NP-hard IP problems themselves. Thus, there are no truly efficient methods to find stable solutions. There are different approached possible and so different solving methods have been developed. An overview of six methods can be found in [7], and these are summarised below. Figure 2.1 shows the general structure which all IPG solving methods use. One of the main differences between methods is the types of equilibria which they can compute. Some methods can only compute MNEa while others only compute PNEa. A second difference lies is how they approach the set of feasible solutions. Some methods start with an under-estimation and expand that (inner approximation) while others start with an over-estimation and refine it (outer approximation).

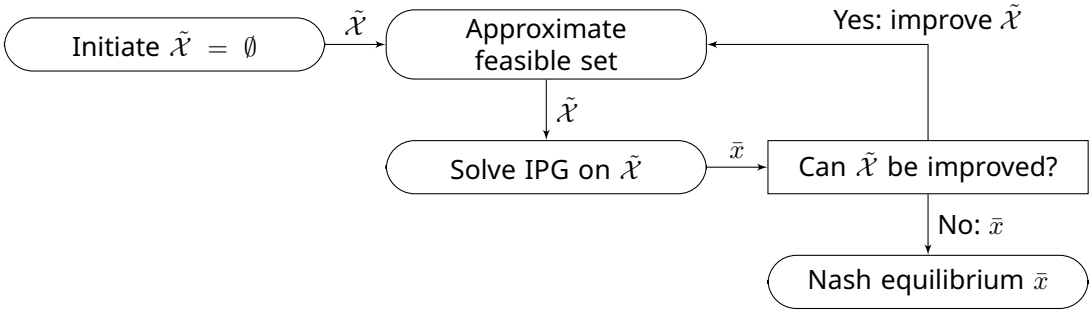

Figure 2.1: General three phase process to compute Nash equilibria in IPGs [7].

MNE methods

The three methods which compute MNEa are the Sample Generation Method (SGM), the Exhaustive Sample Generation Method (eSGM) and the Cut-and-Play (CnP) method [7].

- **SGM**: computes a MNE for IPGs with separable payoff functions and bounded feasible sets. SGM samples a subset of pure strategies for each player as an approximation of the IPG. In each iteration, the approximated IPG is solved and the approximate IPG is updated based on possible profitable deviations for the players. This is repeated until no player can deviate anymore [9].
- **eSGM**: extension of SGM which is capable of computing all possible solutions if the IPG only includes integer variables. eSGM will select the most probable equilibrium [14].
- **CnP**: computes a solution for non-convex IPGs with separable payoffs using outer approximation of the strategy sets of players in the form of a convex hull. Each iteration, the approximated IPG is solved and the solution is checked for feasibility and stability, the results of which are used to update the approximation. This is repeated until a feasible and stable solution is found [6].

PNE methods

The three methods which compute PNEa are the Branching Method (BM), the Branch-and-Prune (BnP) method and the Zero Regrets (ZR) method [7].

- **BM**: computes all PNEa in IPGs where the payoffs of players are continuously differentiable and concave and their strategy sets are convex and integral. BM uses a branching method with relaxation to find tentative equilibria and these are checked for fea-

sibility and stability. Results which are not feasible or stable are used to branch and create sub-problems to solve further [33].

- **BnP**: extends BM by releasing some restrictions on the payoff functions of players and by adding a pruning routine. BnP only needs feasible solutions to work, not tentative equilibria like BM [34].

- **Zero Regrets**: computes, selects and enumerate PNEa for IPGs with linearizable payoff functions. Zero Regrets uses a cutting planes method to create an outer approximation of the IPG by solving the IPG for all players and then computing the best response for each player. All profitable responses are added as cutting planes and this is repeated until no player has a profitable alternative response. See below for a more thorough explanation of Zero Regrets [18].

Zero Regrets

The Zero Regrets (ZR) algorithm computes PNEa or Φ-NEa for IPGs with the possibility to enumerate over all equilibria and optimise over them. Zero Regrets limits the feasible set $\mathcal{X}$, which should be bounded and linear, by generating a set of equilibrium inequalities Ω. The payoffs of players $f^i(x^i, x^{-i})$ should be linearizable, meaning that any interaction between players can be transformed into a linear interaction using additional variables.

To illustrate the functioning of Zero Regrets, I have included Figure 2.2, which gives a visual overview of the algorithm. To start, the algorithm needs the objective functions f^i of all players, the set of equilibrium inequalities Ω and the feasible set $\mathcal{X} = \{\mathcal{X}^1, \dots, \mathcal{X}^n\}$. Zero Regrets first solves the master problem

$$\max_x \{\sum_{i=1}^{n} f^i(x^i, x^{-i}) : x \in \mathcal{X} \text{ s.t. } \Omega\} \tag{2.1}$$

as if all players would be cooperative, by maximising $x \in \mathcal{X}$ subject to the inequalities of Ω. If Problem 2.1 is infeasible (there is no solution possible), the IPG instance being solved has no PNE [18]. If a candidate solution $\bar{x}$ is found, the equilibrium oracle then checks if there are players which would change strategies by solving

$$\max_{x^i} \{f^i(x^i, \bar{x}^{-i}) : x^i \in \mathcal{X}^i\}, \tag{2.2}$$

for each player i, resulting in n partial solutions $\hat{x}^i$. If $f^i(\hat{x}^i, \bar{x}^{-i}) > f^i(\bar{x}^i, \bar{x}^{-i})$, the partial solution improves the position of player i, thus $\bar{x}$ can not be a PNE. In that case, the inequality $f^i(\hat{x}^i, x^{-i}) \leq f^i(x^i, x^{-i})$ is added to the set ω. If $|\omega| = 0$ after checking all players, no player was able to improve their position and $\bar{x}$ is a PNE. If not, all inequalities from ω are added to Ω, limiting the solution space $\mathcal{X}$ s.t. Ω and Equation 2.1 is solved again. This is repeated until the joint problem becomes infeasible or $\bar{x}$ is concluded to be a PNE.

In order to compute Φ-NEa, the update step for $\tilde{\mathcal{X}}$ includes the bound Φ in the new constraints ($f^i(\hat{x}^i, x^{-i}) \leq f^i(x^i, x^{-i}) + \Phi$). If the problem becomes infeasible, increase Φ by 1 and retry to solve it. To compute all PNEa, add a constraint which excludes $\bar{x}$ to $\tilde{\mathcal{X}}$ after a PNE is found, note the PNE and repeat the algorithm. This will force Zero Regrets to find a new PNE or conclude that that does not exists, terminating the algorithm. I implemented Zero Regrets for the Knapsack Packing Game and the Critical Node Game. The implementation can be found in the files where the problems are implemented.

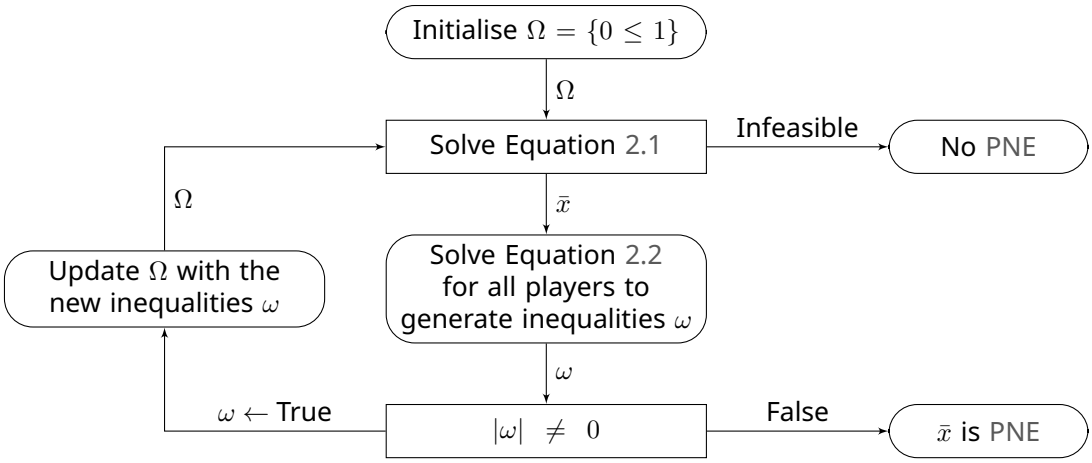

Figure 2.2: High-level representation of the Zero Regrets algorithm. This representation does not include variables needed to make the objective function f^i linear.

2.1.2 Knapsack Packing Game

The KP can be extended into an IPG, namely the Knapsack Packing Game (KPG) [9]. A smaller limited version of the KPG was also presented in [5]. In the KPG, n players have to decide which items they want to pick out of an offering of m items, represented by $x^i \in^m$ for player $i \in \{1, \ldots, n\}$. The payoff function $f(x^i, x^{-i})$ of the KPG introduces a relation between the choices of the players. If two players i and k pick the same item j, then they each get an additional payoff on top of p^i_j and p^k_j. This is called the interaction $I \in \mathbb{Z}^{n \times n \times m}$ and player i received $I^i_{k,j}$ while player k received $I^k_{i,j}$. A player i has the payoff function

$$f(x^i, x^{-i}) = p^i x^i + \sum_{k=1, k \neq i}^{n} (x^i \odot x^k) I^i_k$$

where $p^i \in \mathbb{N}^m$ is the payoff vector of player i and $\odot$ denotes the element-wise multiplication of two vectors, so for two vectors a, b. The constraints on the possible solutions x^i are determined by the weight vector $w^i \in \mathbb{N}^m$ and capacity $c^i \in \mathbb{N}$. Each player solves the optimisation problem

$$\max_{x^i} \{f(x^i, \bar{x}^{-i}) : w^i x^i \leq c^i, x^i \in \{0, 1\}^m\}$$

for a given $\bar{x}^{-i}$, but this already requires the strategies of other players.

Deciding if a KPG instance has a PNE is $\sum^p_2$-complete [18]. Instances with negative values in the interaction matrix I are most likely to not have a PNE. All KPGs have a Φ-NE, but without useful guarantees for Φ.

To solve a KPG instance using Zero Regrets, it is necessary to adjust the objective functions f^i. Zero Regrets requires that the summed objective function of all players is linear, and this is not the case of KPG. To do this, we can replace any reference to $x^i \odot x^k$ with $z^i_k \in \{0, 1\}^m$, where $z^i_{k,j} = x^i_j \wedge x^k_j, \forall i, k \in \{1, \ldots, n\}, i \neq k, \forall j \in \{1, \ldots, m\}$. Since x^i and x^k are both binary vectors, the $\wedge$ operator can be constructed using linear constraints and is included in solvers like Gurobi. See [2] for more information on how to make non-linear functions linear using additional variables. The payoff function then becomes

$$f(x^i, z) = p^i x^i + \sum_{k=1, k \neq i}^{n} z^i_k I^i_k$$

which is linear, as required by Zero Regrets. The implementation of KPG including Zero Regrets can be found in `problems/knapsack_packing_game.py`.

2.1.3 Critical Node Game

The Critical Node Game (CNG) is an IPG with two players and asymmetric roles [17]. This type of game can be used to model attack and defence strategies if there are distinct targets which can be attacked or defended at different costs to the attacker and defender. While the formulation of the Critical Node Game is focused on cloud networks of servers, the same framework can be used in the defence of physical locations.

To model an instance of the CNG with n nodes, the defender controls the binary variables x and the attacker controls the binary variables α. The defender defends a node $i \in \{1, \dots, n\}$ if $x_i = 1$ and the attacker attacks a node i if $\alpha_i = 1$. Every node has a weight associated with defending it (w_i^d) or attacking it (w_i^a) and the defender and attacker have resource limits $c^d \in \mathbb{N}$ and $c^a \in \mathbb{N}$. The defender has a payoff function $f^d(x, \alpha) : \mathbb{Z}^n \times \mathbb{Z}^n \to \mathbb{R}$ which represents the operational capacity of the nodes and the attacker has a payoff function $f^a(\alpha, x) : \mathbb{Z}^n \times \mathbb{Z}^n \to \mathbb{R}$ which represents the disruption of the network. The game consists of the defender solving

$$\max_x \{f^d(x, \alpha) : w^d x \leq c^d, x \in \{0, 1\}^n\},$$

while the attacker is solving

$$\max_\alpha \{f^a(\alpha, x) : w^a \alpha \leq c^a, \alpha \in \{0, 1\}^n\},$$

where $w^d \in \mathbb{N}^n$ and $w^a \in \mathbb{N}^n$.

What makes the CNG interesting to study next to the KPG is that the payoff functions of the defender and the attacker depend on the interactions between their strategies. The basis of the function is the payoff value p_i^d for the defender and p_i^a for the attacker for each node i. However, this value is adjusted based on whether a node is attacked or not and defended or not. For this, we will use four parameters δ, η, ϵ and γ, all in range $[0, 1]$ and $\delta < \eta < \epsilon$. The payoffs for an item i are as follows:

1. Normal operations ($x_i = 0 \wedge \alpha_i = 0$): The defender gets p_i^d and the attacker loses γp_i^a.
2. Successful attack ($x_i = 0 \wedge \alpha_i = 1$): The defender gets δp_i^d and the attacker gets p_i^a.
3. Mitigated attack ($x_i = 1 \wedge \alpha_i = 1$): The defender gets ηp_i^d and the attacker gets $(1 - \eta) p_i^a$.
4. Unchallenged defence ($x_i = 1 \wedge \alpha_i = 0$): The defender gets ϵp_i^d and the attacker gets 0.

Table 2.1: Rewards table for defender (left) and attacker (right) for all four possible combinations of strategies associated with each node i.

	$\alpha_i = 0$		$\alpha_i = 1$	
$x_i = 0$	p_i^d	$-\gamma p_i^a$	δp_i^d	p_i^a
$x_i = 1$	ϵp_i^d	0	ηp_i^d	$(1 - \eta) p_i^a$

See Table 2.1 for an interaction matrix between the different strategies and their resulting payoffs. The objective function of the defender is

$$f^d(x, \alpha) = \sum_{i=1}^{n} p_i^d((1 - x_i)(1 - \alpha_i) + \eta x_i \alpha_i + \epsilon x_i (1 - \alpha_i) + \delta (1 - x_i)\alpha_i) \tag{2.3}$$

while the objective of the attacker is

$$f^a(\alpha, x) = \sum_{i=1}^{n} p_i^a(-\gamma(1-x_i)(1-\alpha_i) + (1-x_i)\alpha_i + (1-\eta)x_i\alpha_i). \tag{2.4}$$

In order to solve CNG using Zero Regrets, the interactions between x and α need to be made linear. The clearest way to do this is to add four vectors of support variables, one for each type of interaction. $z_N = \neg x \wedge \neg\alpha$, $z_S = \neg x \wedge \alpha$, $z_M = x \wedge \alpha$ and $z_U = x \wedge \neg\alpha$, where each equality is checked on an element level in the same way as in the KPG subsection above. With these additional variables, the objective function for the defender becomes

$$f^d(z_N, z_S, z_M, z_U) = \sum_{i=1}^{n} p_i^d(z_N + \eta z_M + \epsilon z_U + \delta z_S)$$

and the function for the attacker

$$f^a(z_N, z_S, z_M, z_U) = \sum_{i=1}^{n} p_i^a(-\gamma z_N + z_S + (1-\eta)z_M).$$

These two functions are linear, as required by Zero Regrets. The implementation of CNG including Zero Regrets can be found in `problems/critical_node_game.py`.

2.2 Inverse optimisation

In integer programming, the goal is to solve the given problem by finding an optimal solution using optimisation methods. With inverse optimisation, this goal is inverted. The standard goal is, given an optimisation problem and a feasible, but not necessarily optimal, solution $\hat{x}$, to change the payoff vector p to make $\hat{x}$ an optimal solution to the optimisation problem [1]. Solving this problem can allow us to learn what drives decisions, for example to predict the movement of earthquakes, and can point to ways to adjust problems to make current solutions optimal, like adjusting road networks to relieve stress on a part of it [23]. For a recent overview of the literature on inverse optimisation, see [12].

I will look at two different types of inverse optimisation in this thesis. The first type is inverse optimisation on one problem instance with known payoffs, constraints and a target feasible solution, very similar to [1, 36, 32] when looking at payoffs and [11, 20] when looking at constraints. I will call this adjustment single-observation inverse optimisation, in the literature it is also called the *single feasible object inverse problem* [23]. The second type is inverse optimisation on a series of observations of similar problems where the parameter (vector) of interest is kept constant, but other attributes of the problem vary [13, 29]. Here, the goal is to reconstruct the parameter or vector of interest based on only very general information. I will call this type reconstruction multi-observation inverse optimisation. This is a variant on a problem which in the literature is called the *multiple feasible set inverse problem* [23].

In the two subsections below, I will use the basic payoff function $f(x)$, with px as concrete function if that is relevant for the inverse optimisation objective. I will assume that the number of decision variables is always n. For the constraints, I will assume that the constraint of interest is one row A_k in the constraint matrix A with a known b_k. A_{-k} then represents the matrix A without row k and b_{-k} represents the vector b without the k-th element.

2.2.1 Adjustment single-observation inverse optimisation

The adjustment single-observation inverse optimisation problem (ASOIOP) takes an instance of an IP with a given payoff vector p, constraints A and b and a target feasible solution $\hat{x}$. The targer solution is usually determined using a heuristic for the IP. The desired output is an adjusted version of either the payoff vector p or (part of) the constraints matrix A which differs as little as possible from the original, which makes the target solution $\hat{x}$ optimal.

When solving the ASOIOP on the payoff vector, the goal is to find a vector q which solves the optimisation problem

$$\min_q\{\mathbf{1}|p-q| : \hat{x} \in \arg\max_x\{qx : Ax \leq b, x \in \mathbb{Z}^n\}, q \in \mathbb{N}^n\}.$$

When solving the ASOIOP on the constraints matrix, there is usually one key constraint of interest and that constraints is the only one being optimised for. The goal is to find a vector d which solves the optimisation problem

$$\min_d\{\mathbf{1}|A_k-d| : \hat{x} \in \arg\max_x\{f(x) : A_{-k}x \leq b_{-k}, dx \leq b_k, x \in \mathbb{Z}^n\}, d \in \mathbb{N}^n\}.$$

In both of the formulations above, you can see that there are essentially two optimisation problems in one. The first problem aims to minimise the change compared to the initial vector/matrix, but solving this requires solving a secondary problem, namely checking if $\hat{x}$ is an optimal solution for these new parameters. Note also that the absolute value is taken of the difference between two vectors in the objective function. Strictly speaking, this is not a linear function. However, it can be made linear using a supporting vector.

2.2.2 Reconstruction multi-observation inverse optimisation

The reconstruction multi-observation inverse optimisation problem (RMOIOP) takes a set of observations $\mathcal{O}$ from related IP instances, which can be interpreted as a sequence of plays of a changing IP problem. Two possible changes are varying the payoffs and varying (a part of) the constraints. These IP instances share an unknown payoff vector p or (a part of) an unknown constraints matrix A and an optimal (PNE or Φ-NE) solution $\hat{x}^o$ is known for each instance $o \in \mathcal{O}$. The desired output is a vector or matrix for which all given solutions are optimal. Generally, it is beneficial to know the general magnitude M of the unknown parameter, to provide scale to the inverse optimisation.

When solving the RMOIOP on the payoff vector, the goal is to find a vector q which solves the optimisation problem

$$\min_q\{\sum_{o\in\mathcal{O}} qx^o - q\hat{x}^o : x^o \in \arg\max_x\{qx : A^o x \leq b^o, x \in \mathbb{Z}^n\}\forall o \in \mathcal{O}, q \in \mathbb{N}^n\}.$$

The main objective is to minimise the difference between the best possible solution for x^o and the known solution $\hat{x}^o$. Of course, once $\hat{x}^o$ is optimal, $qx^o = q\hat{x}^o$ so $qx^o - q\hat{x}^o = 0$. The secondary objective is to solve the underlying IP with the new payoff vector q to see what the optimal solution payoff is. [29] contains a cutting plane algorithm for solving this problem with two different objectives: minimising the worst case $\max qx^o - q\hat{x}^o \forall x^o \in \mathcal{S}^o \forall o \in \mathcal{O}$ or minimising the summed differences $\sum_{o\in\mathcal{O}} \max qx^o - q\hat{x}^o \forall x^o \in \mathcal{S}^o$.

When solving the RMOIOP on (a part of) the constraints matrix, the goal is to find a vector d which solves the optimisation problem

$$\min_d\{1 : \hat{x}^o \in \arg\max_{x^o}\{f^o(x^o) : A_{-k}x^o \leq b_{-k}, dx^o \leq b_k, x^o \in \mathbb{Z}^n\}\forall o \in \mathcal{O}, d \in \mathbb{N}^n\}.$$

The objective here is to find a constraint d which limits the feasible set $\mathcal{X}^o$ of each in such a way that $\hat{x}^o$ is one of the optimal solutions for each problem.

2.2.3 Methods

Most of the inverse optimisation literature is focused on the first type of problem, ASOIOP. [1] discusses the solutions to different problems, mostly for LP problems. [23] gives an overview of different (local) inverse combinatorial optimisation problems and their best known solutions. [24] shows that solving the ASOIOP for the knapsack problem with real-valued payoff values is pseudo-polynomial ($O(n^3S^2)$, where $S = \max\{|w_i| : \forall i \in \{1, \dots, n\}\}$) and so is the general ASOIOP of an IP with fixed number of constraints ($O(n^3S^2(mS)^{2m})$, where $S = \max\{|A_ij|, |b_i| : \forall i \in \{1, \dots, n\}, \forall j \in \{1, \dots, m\}\}$ and m is the number of constraints). Wang presents cutting plane algorithms for the ASOIOP on LPs and IPs, using the dual problem [36]. [13] adapt Wang's method for IPGs and RMOIOP, specifically for a location selection game.

[11] and [20] consider the problem of inverse optimisation of constraints for LPs. [11] focuses on the formulation of different problems without specific algorithms to solve them and depends on the duality formulations of specific problems to solve them. [20] also considers the case of dealing with multiple observations of different problems.

Wang's IP cutting plane method

Wang's cutting plane algorithm [36] for inverse optimisation of IPs solves the ASOIOP using the duality of the IP. The algorithm starts with an empty approximate solution set $\tilde{\mathcal{S}} = \emptyset$ solves the master problem

$$\min_{q,y}\{\mathbf{1}|p - q| : A^\top y \geq q, q\hat{x} \geq q\bar{x} \quad \forall \bar{x} \in \tilde{\mathcal{S}}, \quad y \geq 0, q \in \mathbb{R}^n, y \in \mathbb{R}^m\} \tag{2.5}$$

resulting in a candidate solution q. The constraint $A^\top y \geq q$ is used in [36] to prove that this problem can be converted to an equivalent linear program which terminates finitely with an optimal solution. In the linear problem, $A^\top y \geq q$ guarantees that the adverse problem is bounded from above and thus actually has an optimal solution. Experimentally, the constraint lowers the running time of the algorithm on KP instances. The adverse problem

$$\max_x\{qx : Ax \leq b, x \in S\} \tag{2.6}$$

is solved using the candidate solution q, which results in a new solution x from the set of feasible solutions $S = \{x \in \{0, 1\}^n : Ax \leq b\}$. If $q\hat{x} \geq qx$, then $\hat{x}$ has become an optimal solution and the problem is solved. q is the resulting payoff vector. If $q\hat{x} < qx$, then x is added to $\mathcal{S}$ and both problems are solved again. See Figure 2.3 for a visual representation.

Cronert's IPG cutting plane method

Cronert's IPG cutting plane algorithm [13] is inspired by the approach used in [36], specifically the approach for inverse optimisation under the weighted L_∞ norm. Like Wang's method, this method builds an approximation $\tilde{\mathcal{S}}_i^o$ in small steps instead of considering all possible solutions $\mathcal{S}_i^o$. The goal is to find the relevant parameters based on the choices of players $i \in I$ for $o \in O$ observations of similar IPGs where the payoffs are kept constant. All

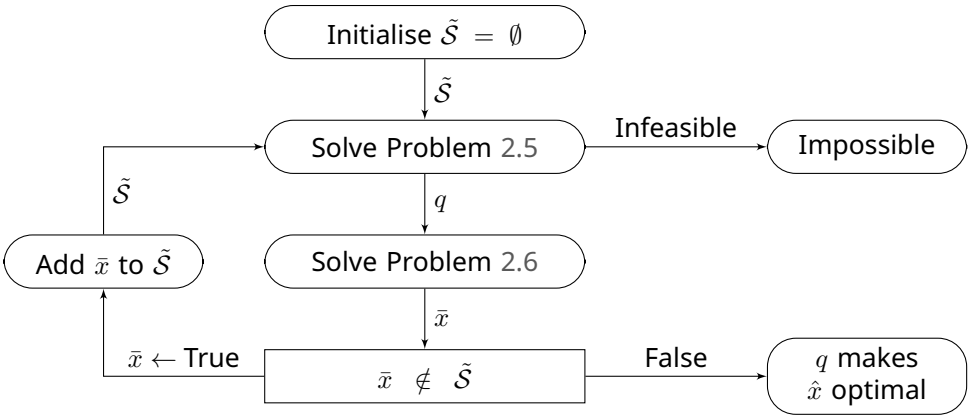

Figure 2.3: Visual representation of the algorithm proposed by Wang for inverse optimisation of IPs in [36].

solutions $\hat{x}_i^o$ are either PNEa or Φ-NEa. The master problem

$$\min_q \{\sum_{o\in\mathcal{O}}\sum_{i=1}^{n} \theta_i^o : \theta_i^o \geq f^i(\bar{x}_i^o, \hat{x}_{-i}^o) - f^i(\hat{x}_i^o, \hat{x}_{-i}^o)\ \bar{x}_i^o \in \tilde{\mathcal{S}}_i^o\ \ \forall i \in I\ o \in O\} \tag{2.7}$$

is solved on the given solutions, where the vector q^i is part of the objective function f^i. With the resulting vectors q^i for each player, the adverse problem

$$\max_{x_i^o} \{f^i(x_i^o, \hat{x}_i^o) : x_i^o \in \mathcal{S}_i^o\} \tag{2.8}$$

is solved for each player $i \in I$ and all observations $o \in \mathcal{O}$. The solutions to the subproblems are then checked to see if any of them are not yet part of the relevant set $\tilde{\mathcal{S}}_i^o$. If so, the relevant solution sets $\tilde{\mathcal{S}}_i^o$ are updated to include these new solutions. See Figure 2.4 for a visual representation.

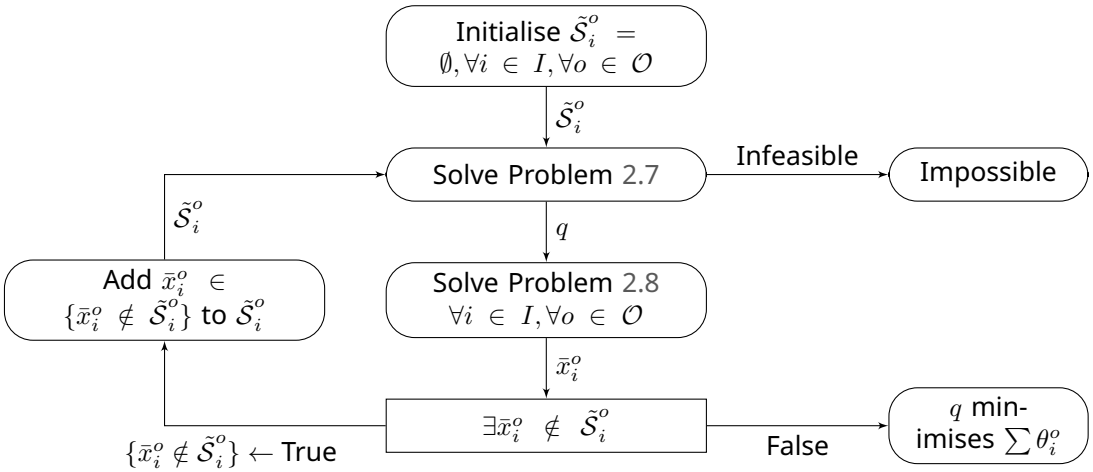

Figure 2.4: Visual representation of the algorithm proposed by Cronert for inverse optimisation of IPGs in [13].

2.3 Chapter conclusion

In this Chapter, I sketched the existing methods for inverse optimisation and the two problems I will be using to investigate inverse optimisation of IPGs. In the next Chapter, I will develop and test methods for adjustment inverse optimisation of payoffs and constraints and apply these to KP, KPG and CNG instances. I will be using and adapting Wang's and Cronert's methods to suit the challenges of the different problem types.

Chapter 3

Adjustment single-observation inverse optimisation

The goal of adjustment inverse optimisation is to adjust the payoffs or constraints of an optimisation problem in such a way that a given target feasible solution to the optimisation problem becomes the optimal solution. Wang's method gives a starting point for this, but it only deals with learning (parts of) the objective function [36]. Cronert's method extends it to IPGs, but again only for learning (parts of) the objective function and only on a problem with one parameter per player plus one for the game [13]. In this section, I will investigate the application of these two methods to the three problems I described earlier: KP, KPG and CNG. For each, I will try to adjust the weights and the payoffs to make the target solution optimal and see if there are instances where it is impossible to make the target solution optimal.

In this chapter, I will sometimes refer to the relative absolute change (RAC), which is a measure of the numerical change between two vectors or matrices relative to their total value. If the weights w are being adjusted for instance, the adjusted values are v. The RAC is then defined as $\sum_{i=1}^{n}|w_i - v_i|/\sum_{i=1}^{n} w_i$. The mean RAC is just the mean of the RAC values of different samples and the RAC is presented like this in the results. In this chapter, this measure is used to show the impact of different problem characteristics on the feasibility of adjustment and the magnitude of the adjustments required to achieve an optimal or equilibrium solution.

3.1 Inversing the Knapsack Problem

As a preparation for the KPG and to provide a performance reference for the KPG and CNG, I decided to include KP as the first inverse optimisation problem. [32] considers the problem of payoff adjustment, so I will include their results in the next Subsection as a benchmark. For KP, the two vectors of interest are clear: the payoff vector $p \in \mathbb{N}^n$ and the weight vector $w \in \mathbb{N}^n$. In this section, I will apply and adapt Wang's method to solve the adjustment inverse problem for both.

3.1.1 Adjusting payoffs

How can we find a vector with minimal adjustment q of the payoff vector p of a KP instance to make a target solution $\hat{x}$ optimal? Wang's method [36] provides a general solution for all IPs. [32] looks at this problem and considers two different distance measurements: $L_1 : \sum_{i=1}^{n} |w_i - v_i|$ and $L_\infty : \max_i\{|w_i - v_i|\}$. For L_∞, they provide a pseudo-polynomial algorithm capable of dealing with instances of 100,000 items. For L_1, the authors consider a bi-level approach similar to Wang's, but reject it due to its theoretical computational complexity. They end up using a dynamic programming formulation of KP and implement it using integer programming. However, they only solve instances with 80 items without correlation, which leaves a lot of room for improvement.

I solved the payoff adjustment problem with n variables using Wang's method for a given target solution $\hat{x} \in \{0,1\}^n$, payoffs $p \in \mathbb{N}^n$, weights $w \in \mathbb{N}^n$ and capacity $c \in \mathbb{N}$ using the master problem

$$\min_{q,y}\{\sum_{i=1}^{n}|p_i - q_i| : q\hat{x} \geq qx \quad \forall x \in \tilde{\mathcal{S}}, \quad y \odot w \geq q, \quad q \in \mathbb{Z}_+^n, y \in \mathbb{R}_+^n\}.$$

The master problem uses a dual problem constraint $y \odot w \geq q$, which is also used in Wang's method [36]. It guarantees the algorithm terminates and experimentally it reduces the running time of the algorithm significantly. The proposed vector q is then tested using the adverse problem

$$\max_x\{qx : wx \leq c, \quad x \in \{0,1\}\}$$

and the resulting solution $\bar{x}$ is added to the approximate solutions set $\tilde{\mathcal{S}}$, if it was not already part of the solutions set. For a visual overview of the algorithm, see Figure 2.3. Once the adverse problem is unable to find a new solution $\bar{x}$ which is not yet part of $\tilde{\mathcal{S}}$, then q must make $\hat{x}$ optimal since $q\hat{x} \geq q\bar{x}$ based on the master problem. This same pattern is used in all payoff adjustment sections.

Implementation For the problems in this section, the payoffs $p \in \mathbb{N}^n$ are generated as an integer uniform random vector with entries in the range $[1, \ldots, r]$. The weights $w \in \mathbb{N}^n$ are generated as an integer uniform random vector conditional on p in the range $[\max(1, p - \lceil r/5 \rceil), \ldots, \min(r, p + \lfloor r/5 \rfloor)]$ so the maximum difference between the p_i and w_i of an item i is equal to $r/5$. The capacity $c = 0.5\sum_{i=1}^{n} w_i$ depends on the weights. These choices were inspired by the experiment design in [32]. The RAC is equal to $\sum_{i=1}^{n}|p_i - q_i| / \sum_{i=1}^{n} q_i$.

The target solutions $\hat{x}$ are generated using the greedy heuristic proposed in [16], where items are ordered by their relative value p_i/w_i. The item with the highest relative value is added each time until no items can be added anymore. The implementation of this section can be found in `methods/local_inverse_kp.py`. The results were generated with local_inverse_kp_results.py and can be found in Table A.1. Figure 3.1 is based on Table A.1.

Results Figure 3.1 contains the running times based on 30 samples for each combination of parameters with problems generated with correlation. You can see that the mean running times increase linearly with the number of items n and that the range of values r has almost no effect. We can measure the required change to make $\hat{x}$ optimal using the mean RAC. The mean RAC for each set of instances was less than or equal to 0.002 for all combinations. Thus, the vector p changed 0.2% or less to make $\hat{x}$ optimal.

To compare, I also implemented the dynamic algorithm proposed by [32]. Solving an instance with $n = 25$ and $r = 100$ took 15 seconds while my adaptation of Wang's method

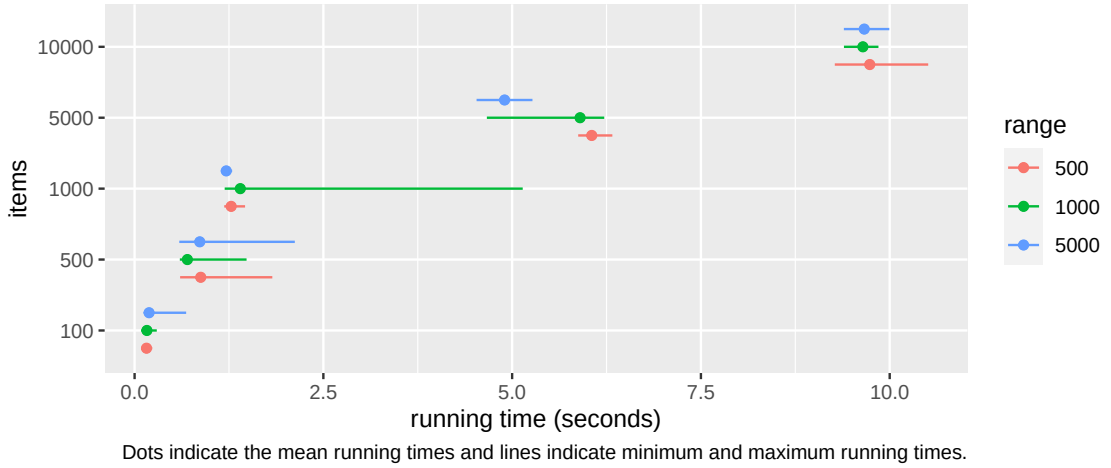

Figure 3.1: Running times of KP payoff adjustment over 30 samples.

solved it in 0.2 seconds. My implementation of their dynamic method is likely to be less efficient than theirs, since they solve instances with $n = 30$ and $r = 100$ with a similar approach for generating instances in about 5 seconds [32]. However, at $n = 50$, they report an average solution time of 209 seconds for similar instances, which is considerably slower than Wang's method. Their dynamic method has a complexity of $O(nc)$, which means its running time increases with the number of items, but also with the increase in the capacity as a result of increases in the number of items or range of values. Thus, I decided to not try to run a comparison with multiple samples like the results in Table A.1.

3.1.2 Adjusting weights

How can we find a vector with minimal adjustment v of the weights vector w of a KP instance to make a target solution $\hat{x}$ optimal? There is some literature on finding constraints of linear optimisation problems [10, 20], but it is focused on LPs. For this problem, the approach used by Wang of starting with an empty set of solutions is useful but the method for creating the constraints needs to be adapted. Instead of forcing that all feasible solutions $\bar{x}$ satisfy $f(\hat{x}) \geq f(\bar{x})$, we need to make all $\bar{x}$ infeasible for which $f(\bar{x}) > f(\hat{x})$. Any vector $\bar{v}$ for which $\hat{x}$ is sub-optimal is incorrect and the resulting optimal solution $\bar{x}$ should not be feasible. We can add the constraint $v\bar{x} > c$ to the master to enforce this. Note that this is not a strict inequality, which thus requires some care in the implementation. I solved the weight adjustment problem with n variables by adapting Wang's method for a given target solution $\hat{x} \in \{0,1\}^n$, given payoffs $p \in \mathbb{N}^n$, weights $w \in \mathbb{N}^n$ and capacity $c \in \mathbb{N}$ using the master problem

$$\min_{v}\{\sum_{i=1}^{n}|w_i - v_i| : v\hat{x} \leq c, v\bar{x} > c \; \forall \bar{x} \in \tilde{\mathcal{S}}, v \in \mathbb{N}^n\}, \tag{3.1}$$

which aims to find a valid vector v. The solutions are then tested using the adverse problem

$$\max_{x}\{px : vx \leq c, x \in \{0,1\}^n\}. \tag{3.2}$$

If $p\hat{x} \geq p\bar{x}$, then v makes $\hat{x}$ optimal and the algorithm is done. Otherwise $\bar{x}$ is added to $\tilde{\mathcal{S}}$, which then creates a new constraint in the master problem which makes $\bar{x}$ infeasible. See Figure 3.2 for a visual overview of this procedure. Just like in the previous section, if the

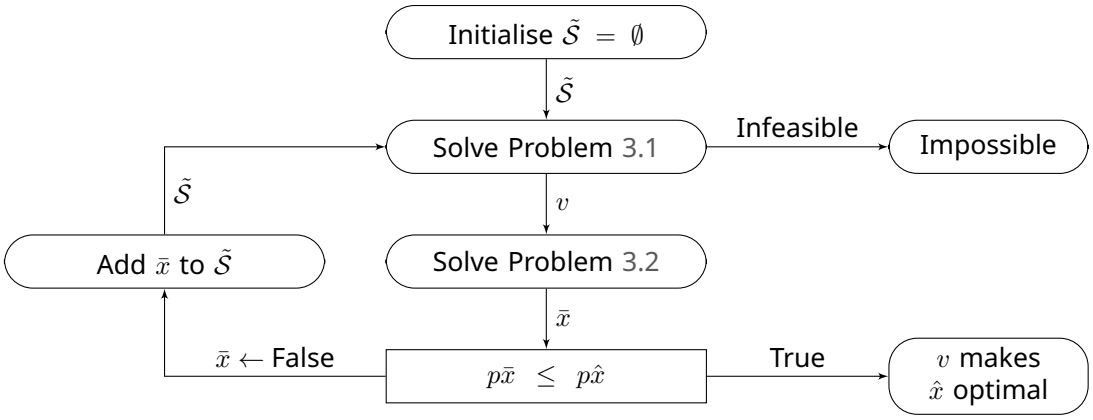

Figure 3.2: Visual representation of the weight adjustment algorithm for a KP instance.

adverse problem is unable to find a solution $p\bar{x} > p\hat{x}$, then the weights v must make $\hat{x}$ an optimal solution. This pattern is used in all weights adjustment sections.

Implementation The instances used in this subsection were generated in the same way as the instances in the previous section on payoff adjustment, with one major difference. Since the inequalities in this section are not strict, we need to adjust the implementations accordingly. For the comparison $f(\bar{x}) > f(\hat{x})$, I implemented it using a small error term $e = 0.001$ to define the strict inequality $f(\bar{x}) \geq f(\hat{x}) + e$. I used the same approach for the master problem constraints $v\bar{x} > c$. These are implemented as $v\bar{x} \geq c + e$. The RAC is equal to $\sum_{i=1}^{n} |w_i - v_i| / \sum_{i=1}^{n} w_i$.

The implementation of this section can be found in `methods/local_inverse_kp.py`. The results were generated with local_inverse_kp_results.py and can be found in Table A.2. Figure 3.3 is based on Table A.2.

Results Figure 3.3 contains the running times based on 30 samples for each combination of parameters with problems generated with correlation. You can see that the running times increase linearly with n and that the range of values r has little impact. If we compare the results of Figures 3.1 and 3.3, we can see that for all instances up to $n = 1000$, the min and max running times of the payoffs and weights overlap. The weights tend to vary more in this range. For the $n = 5000$ and $n = 10000$ instances, the variation drops. Both the payoff and weight run times become very regular and the standard deviations of both are almost identical (see Tables A.1 and A.4). For these instances, the weight method runs about 30% quicker than the payoffs method. The required adjustments to w were minimal, since the mean RAC was less than or equal to 0.001 for all combinations. So, the vector w changed by less than 0.1% to make $\hat{x}$ optimal.

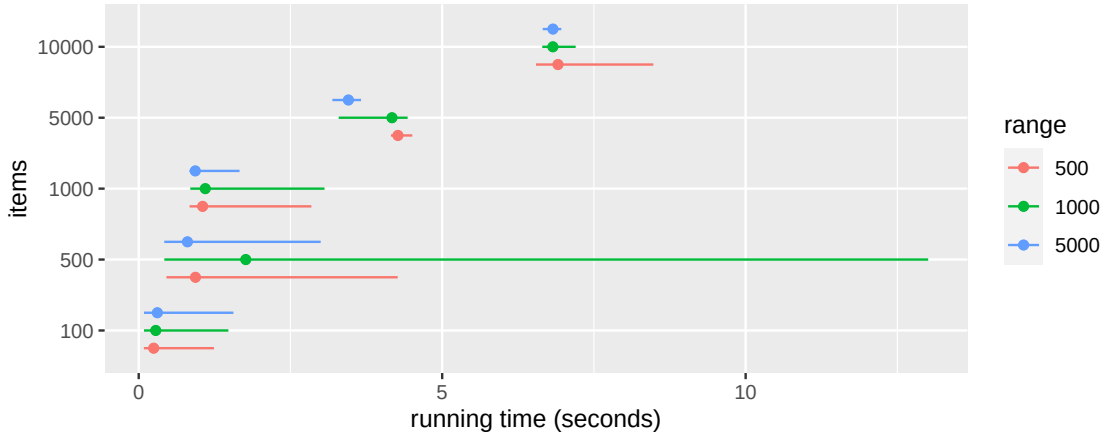

Figure 3.3: Running times of KP weight adjustment over 30 samples.

3.2 Inversing the Knapsack Packing Game

KPG is more complicated than KP due to the interactions between players. However, the inverse versions of this problem can be solved in similar ways to KP since players already know the actions of their opponents. This removes the need to solve a complete KPG instance. We only need to ensure that $\hat{x}$ is a PNE for the proposed payoff function which is defined by the payoff vectors $q \in \mathbb{N}^{n\times m}$ and the interaction matrix $J \in \mathbb{Z}^{n\times n\times m}$ or the proposed weights vectors $v \in \mathbb{N}^{n\times m}$. As long as none of the players are able to improve their position by switching from a given target solution $\hat{x}^i$ to a different solution $\bar{x}^i$, the target solution is a PNE. To do this, we need to adjust the algorithms from the previous section to accept multiple players.

3.2.1 Adjusting payoffs and interactions

Adjusting the payoffs of a KPG instance is similar to the problem of Subsection 3.1.1, because the strategies of all players are already fixed in the target solution $\hat{x}, \hat{\alpha}$, but with an important difference: the payoff function of KPG is more complicated. The function f^i of a player i consists of two parts: $p^i x^i$ and $\sum_{k=1,k\neq i}^{n}(x^i \odot x^k)I_k^i$. One possible approach is to try to only adjust the vectors p^i in order to make $\hat{x}^i$ optimal, however since a part of the payoff of each item would then be fixed based on the strategies $\hat{x}^{-i}$ and the lower bound of the payoffs is 1, this leads to infeasible problems. This means that we have to adjust both the payoffs and the interactions to make the target solution $\hat{x}$ into an optimal solution.

I solved the payoff adjustment problem with n players and m items by adapting Wang's method for a given set of target solutions $\hat{x} \in \{0,1\}^{n\times m} = \{\hat{x}^1, \ldots, \hat{x}^n\}$, payoffs $p \in \mathbb{N}^{n\times m} = \{p^1, \ldots, p^n\}$, interactions $I \in \mathbb{Z}^{n\times n\times m} = \{I^1, \ldots, I^n\}$, weights $w \in \mathbb{N}^{n\times m} = \{w^1, \ldots, w^n\}$ and capacities $c \in \mathbb{N}^n = \{c^1, \ldots, c^n\}$ using the master problem

$$\min_{q,J}\{\sum_{i=1}^{n}(\sum_{j=1}^{m}|p_j^i - q_j^i| + \sum_{k=1,k\neq i}^{n}\sum_{j=1}^{m}|I_{k,j}^i - J_{k,j}^i|) : f^i(\hat{x}^i, \hat{x}^{-i}) \geq f^i(\bar{x}^i, \hat{x}^i)$$

$$\forall \bar{x}^i \in \tilde{\mathcal{S}}^i \, \forall i \in \{1, \ldots, n\}, q \in \mathbb{N}^{n\times m}\}, \quad (3.3)$$

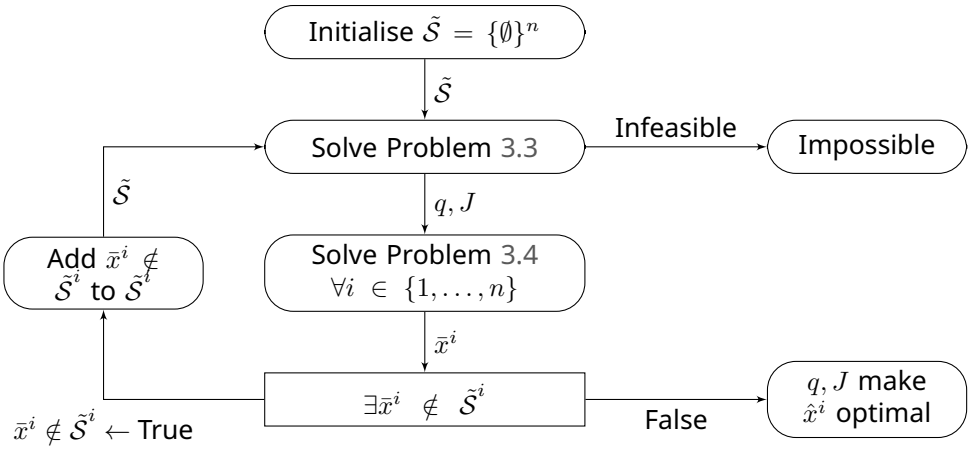

Figure 3.4: Visual representation of the payoff adjustment algorithm for a KPG instance.

where the payoff function f^i of player i is

$$f^i(x^i, x^{-i}) = q^i x^i + \sum_{k=1, k\neq i}^{n} (x^i \odot x^k) J^i_k.$$

The proposed payoffs and interactions are then tested with the adverse problem

$$\max_x \{f^i(x, \hat{x}^{-i}) : w^i x \leq c^i, x \in \{0,1\}^n\} \tag{3.4}$$

for all players $i \in \{1, \ldots, n\}$, which results in n new player solutions $\bar{x}^i$. Each of these solutions are checked against the set of solutions $\tilde{\mathcal{S}}^i$ and if none of solutions are not yet part of their respective set, then the solution is a PNE. Problem A.3 has a constraint for each solution in $\tilde{\mathcal{S}}^i$ preventing alternative solutions from having a higher payoff than the given solution $\hat{x}$. If there are no alternatives $\bar{x}$ which get a higher payoff than $\hat{x}$, then the current payoffs and interactions make $\hat{x}$ a PNE. See Figure 3.4 for a visual overview.

Implementation For the problems in this section, the payoffs $p \in \mathbb{N}^{n\times m}$ are generated as an integer uniform random matrix with entries in the range $[1, \ldots, r]$. The weights $w \in \mathbb{N}^{n\times m}$ are generated as an integer uniform random matrix conditional on p in the range $[\max(1, p - \lceil r/5 \rceil), \ldots, \min(r, p + \lfloor r/5 \rfloor)]$. Capacities $c^i = 0.5 \sum w^i \ \forall i \in \{1, \ldots, n\}$ depend on the weights w^i. KPG instances also have an interaction matrix $I \in \mathbb{Z}^{n\times n\times m}$. This matrix is generated as an integer uniform random matrix with entries in the range $[-\lceil \frac{r}{f} \rceil, \ldots, \lfloor \frac{r}{f} \rfloor]$ if negative interactions are allowed and $[0, \ldots, \lfloor \frac{r}{f} \rfloor]$ if negative interactions are not allowed. For these experiments, I choose $f = 3$. The time limit for all instances was 60 seconds. The RAC is equal to $(\sum_{i=1}^n \sum_{j=1}^m |p^i_j - q^i_j| + \sum_{k=1}^n |I^i_{j,k} - J^i_{j,k}|)/(\sum_{i=1}^n \sum_{j=1}^m p^i_j + \sum_{k=1}^n I^i_{j,k})$.

The target solutions for this problem are generated using a turn-based heuristic. Initially, all solution vectors are set to the zero-vector ($x^i = (0, \ldots,)^m$). Players then solve their individual problem and update x^i, with $x^{-i} = (0, \ldots, 0)^m$, resulting in $x = \{x^1, \ldots, x^n\}$. These individual solutions are stored in solutions sets $\mathcal{S}^i = \{x^i\}$. Then, players keep solving their individual problems using the most current x^{-i} to update their solutions x^i, adding x^i to $\mathcal{S}^i$ if $x^i \notin \mathcal{S}^i$, until a round where no player adds a new x^i to $\mathcal{S}^i$. Note that the strategies are updated in real-time, so player 2 can already use the strategies of player 1.

The implementations of this section can be found in `problems/local_inverse_kpg.py`. The results were generated with local_inverse_kpg_results.py and can be found in Tables A.3, 3.1, A.4 and 3.2. Figures 3.5 and 3.7 are based on Tables A.3 and A.4.

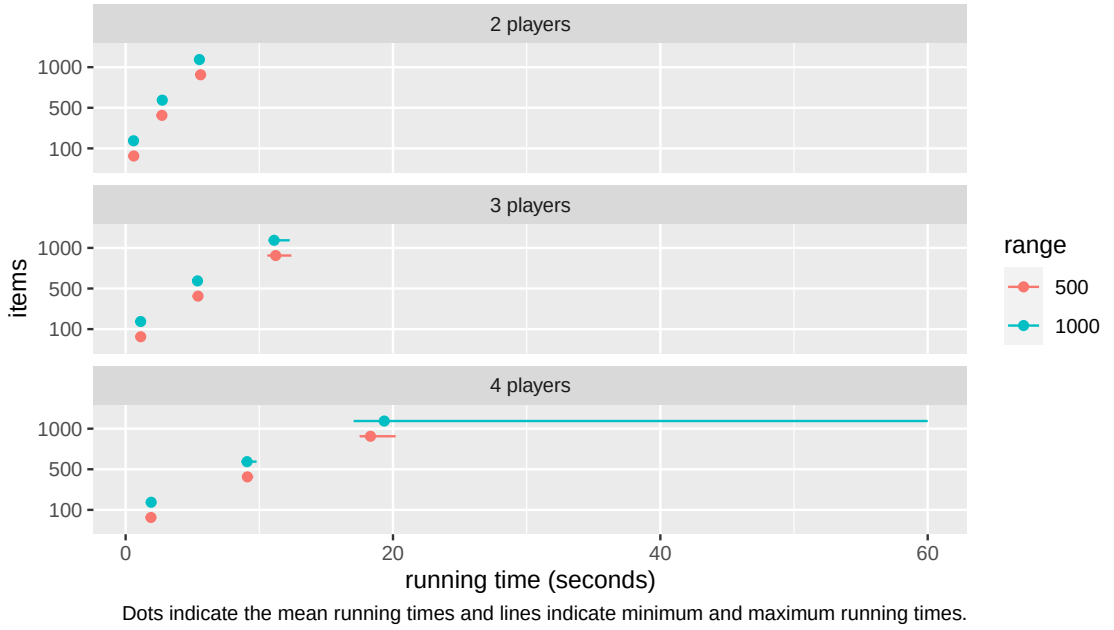

Figure 3.5: Running times of KPG payoff and interaction adjustment over 30 samples.

Results Figure 3.5 contains the running times for all combinations of players, ranges and items. It is clear that solving this problem for KPG is much more challenging than for KP. The range of the number of items m is much lower, since the mean running time for 500 items with 3 players is already similar to the average running time of adjusting the payoffs of a KP with 10000 items. We can see that doubling the number of items roughly doubles the running time, and adding an additional player doubles the running time. A doubling of the number of items m means the number of variables increases from $mn + mn^2 - mn = mn^2$ (payoffs + interactions - self-interactions) to $(2m)n^2 = 2mn^2$, so a doubling in the number of variables. Increasing the number of players by 1 leads to a doubling of the running time or more, and in terms of variables the number of variables increases from mn^2 to $m(n+1)^2 = m(n^2+2n+1) = mn^2 + 2mn + m$, which is an increase of $2mn + m$. If $n = 2$ and $m = 100$, this means the number of variables raises from $100 * 2^2 = 400$ to $100 * 2^2 + 2 * 100 * 2 + 100 = 900$, which is more than double the number of variables. The mean RAC was 0.001 or less for all instances, so I did not add those to the table. For the samples with $n = 2, m = 1000, r = 1000$, one problem was infeasible and timed out after 60 seconds.

The results in Figure 3.5 are for problems without negative interactions. Table 3.1 contains the results for some smaller instances with negative interactions. It is clear that allowing the values of the interaction matrix I to be negative is a big burden on the algorithm adding another source of variance. Some instances were infeasible due to hitting the time limit, while others were solved in less than 2 seconds. The mean RAC of these instances is also much higher than for the instances without negative interactions. Interestingly, the instances with 100 items were solved quicker than those with 50 items.

3.2.2 Adjusting weights

Just like in the previous subsection, adjusting the weights for a KPG instance is related to the problem of Subsection 3.1.2. With the payoffs fixed, now we need to adjust the weights to make the target solution $\hat{x}$ a PNE. Since weight adjustment on KP instances was quicker

Table 3.1: KPG payoff and interaction adjustment running time statistics in seconds, number of infeasible instances and mean RAC on instances with 2 players and negative interactions over 30 samples.

items	range	mean	SD	min	max	inf	RAC
50	500	24.252	29.188	0.316	60.000	12	0.194
	1000	18.366	27.256	0.334	60.000	9	0.193
100	500	7.586	17.511	0.898	60.000	3	0.180
	1000	15.233	24.699	0.911	60.000	7	0.193
200	500	23.899	22.451	3.775	60.000	7	0.184
	1000	18.348	20.229	3.341	60.000	5	0.187

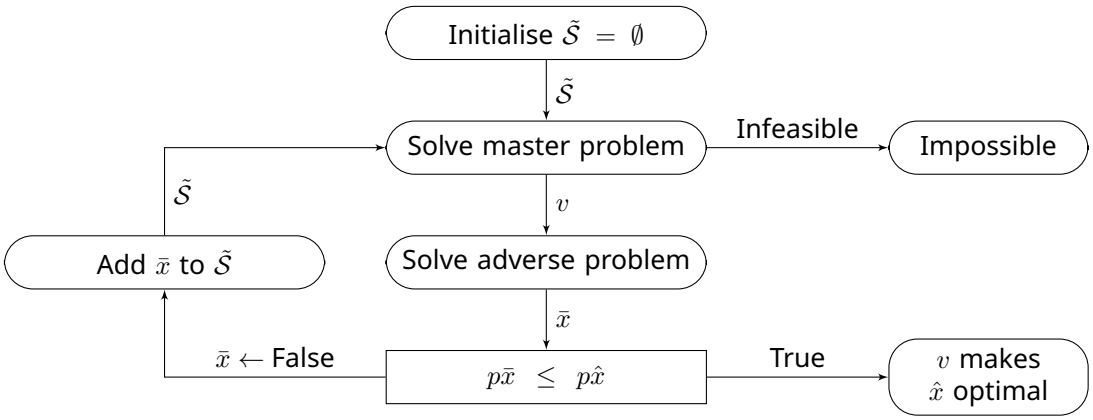

Figure 3.6: Visual representation of the weight adjustment algorithm for a KPG instance.

than payoff adjustment, it is reasonable to expect the same from KPG instances, since each player is essentially solving a KP-like problem for themselves.

I solved the weights adjustment problem with n players and m items by extending the algorithm from Subsection 3.1.2 for a given set of target solutions $\hat{x} \in \{0,1\}^{n\times m} = \{\hat{x}^1, \dots, \hat{x}^n\}$, a given set of payoffs $p \in \mathbb{N}^{n\times m} = \{p^1, \dots, p^n\}$, a given set of interactions $I \in \mathbb{Z}^{n\times n\times m} = \{I^1, \dots, I^n\}$, a given set of weights $w \in \mathbb{N}^{n\times m} = \{w^1, \dots, w^n\}$ and a given set of capacities $c \in \mathbb{N}^n = \{c^1, \dots, c^n\}$. Using the target solution $\hat{x} = \{x^1, \dots, x^n\}$, we can redefine the payoff function $f^i(x^i)$ for each player as

$$f^i(x^i) = (p^i + \sum_{k=1,k\neq i}^{n} I^i_k \odot \hat{x}^k)x^i,$$

which is just a KP with a different payoff vector $p^i + \sum_{k=1,k\neq i}^{n} I^i_k \odot \hat{x}^k$. Thus, we can apply the method from Subsection 3.1.2 to this problem, with some slight modifications on the variables. See Figure 3.6.

Implementation The instances used in this section were generated in the same way as the instances in the previous section on payoff adjustment and implemented in the same files. The time limit of the adjustment was 60 seconds for all instances. The RAC is equal to $\sum_{i=1}^{n}\sum_{j=1}^{m}|w^i_j - v^i_j| / \sum_{i=1}^{n}\sum_{j=1}^{m} w^i_j$.

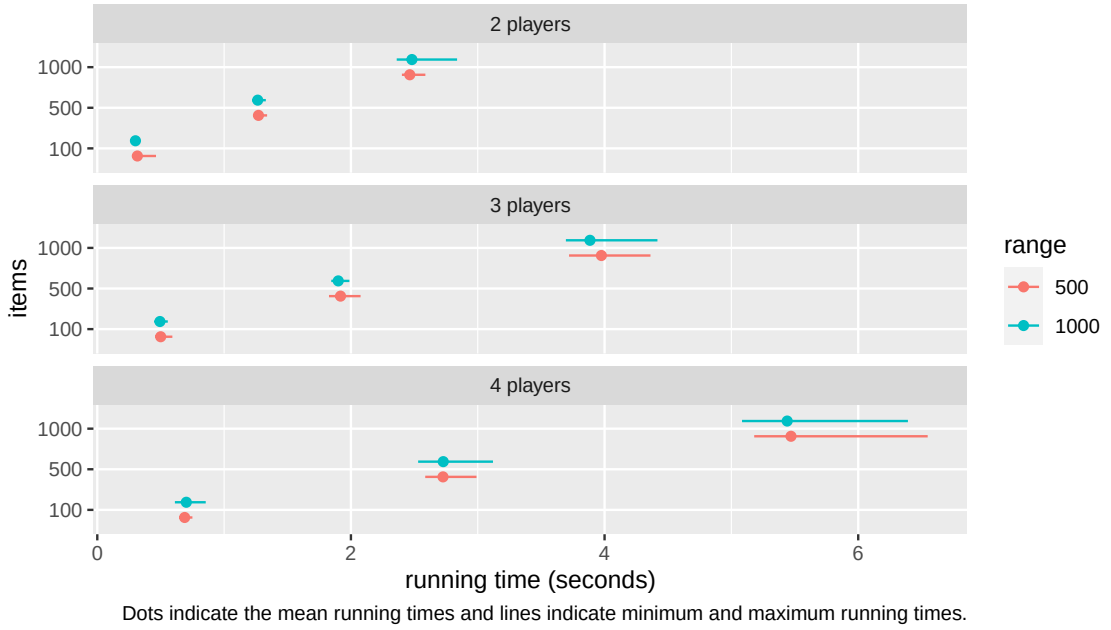

Figure 3.7: Running times of KPG weight adjustment over 30 samples.

Table 3.2: Running time statistics in seconds, number of infeasible instances and mean RAC of the weights adjustment algorithm on instances with 2 players, negative interactions and 30 samples per combination.

items	range	mean	SD	min	max	inf	RAC
50	500	0.186	0.034	0.154	0.333	0	0.011
	1000	0.195	0.019	0.171	0.253	0	0.012
100	500	0.342	0.037	0.291	0.436	0	0.009
	1000	0.345	0.033	0.301	0.425	0	0.006
200	500	0.418	0.017	0.395	0.458	0	0.000
	1000	0.444	0.035	0.400	0.530	0	0.000

Results Figure 3.7 contains an overview of the running times of the different combinations. It is clear that adding an additional player increases solution time, with the increase being a little larger than the time per player at two players. Doubling the number of items also leads to roughly double the running time. The range r also does not seem to have any effect on the running times. Making $\hat{x}^i$ optimal requires similar changes as for the payoffs, the mean RAC was less than 0.001 for all combinations.

The results in Figure 3.7 do not contain any instances with negative interactions. Table 3.2 contains the results of adjusting weights of instances with negative interactions. These instances are all for just 2 players and the running times and number of infeasible instances show that adjusting weights for KPG instances with negative interactions is not significantly harder than without these interactions. The changes required also decrease when increasing the number of items. These results are much better than those of the payoffs, where major changes were required with long running times to achieve the same effect.

3.3 Inversing the Critical Node Game

A big difference between KPG and CNG instances is that PNE solutions are much rarer for CNG instances. Most CNG instances only have a Φ-NE. This lead me to formulate a double question for this section: can a CNG instance with a target solution $\hat{x}, \hat{\alpha}$ be adjusted to make $\hat{x}, \hat{\alpha}$ a PNE and can a Φ-NE solution $\hat{x}, \hat{\alpha}$ be adjusted to reduce Φ? The first question is more in line with the previous section, but the second question might also give useful insights about the nature of Φ in the Φ-NE.

3.3.1 Adjusting payoffs

Adjusting the payoffs of a CNG instances is similar to the problem of Subsection 3.1.1, but with a more complicated payoff function. The payoff function of the defender $f^d(x, \alpha)$ and the attacker $f^a(\alpha, x)$ are asymmetric, both in terms of parameters and in structure.

I solved the Φ payoff adjustment problem by extending Wang's method with the Φ-NE approach used in the Zero Regrets implementation proposed in [17] in order to find the minimal change of the payoff to get the Φ-NE with the lowest possible Φ.

The master problem being used is

$$\min_{q^d, q^a} \{\sum_{i=1}^{n} |p_i^d - q_i^d| + |p_i^a - q_i^a| : f^d(\hat{x}, \hat{\alpha}) + \Phi \geq f^d(\bar{x}, \hat{\alpha}) \forall \bar{x} \in \tilde{\mathcal{S}}^d, f^a(\hat{\alpha}, \hat{x}) + \Phi \geq f^a(\bar{\alpha}, \hat{x}) \\ \forall \bar{\alpha} \in \tilde{\mathcal{S}}^a, q^d, q^a \in \mathbb{N}^n, \Phi \leq \Phi_{UB}\}, \quad (3.5)$$

where the objective functions f^d and f^a use the adjusted payoffs q^d and q^a instead of p^d and p^a, so the payoff functions look like

$$f^d(x, \alpha) = \sum_{i=1}^{n} q_i^d((1 - x_i)(1 - \alpha_i) + \eta x_i \alpha_i + \epsilon x_i(1 - \alpha_i) + \delta(1 - x_i)\alpha_i)$$

and

$$f^a(\alpha, x) = \sum_{i=1}^{n} q_i^a(-\gamma(1 - x_i)(1 - \alpha_i) + (1 - x_i)\alpha_i + (1 - \eta)x_i \alpha_i).$$

Φ starts at 0 and is increased by 1 each time the master problem becomes infeasible. The proposed payoffs q^d and q^a are tested with the adverse problems

$$\max_x \{f^d(x, \hat{\alpha}) : w^d x \leq D, x \in \{0, 1\}^n\} \quad (3.6)$$

and

$$\max_\alpha \{f^a(\alpha, \hat{x}) : w^a \alpha \leq A, \alpha \in \{0, 1\}^n\}. \quad (3.7)$$

The new solutions $\bar{x}$ and $\bar{\alpha}$ are checked against the sets of solutions $\tilde{\mathcal{S}}^d$ and $\tilde{\mathcal{S}}^a$. If they do not appear in them, then they are added to the sets of solutions. If they both already appear in their respective sets, the solution is a PNE if $\Phi_{UB} = 0$ and a Φ-NE otherwise with Φ_{UB} as Φ. See Figure 3.8 for a visual overview of the procedures.

Implementation For the problems in this section, the settings for the payoffs $p \in \mathbb{N}^{2 \times n}$, weights $w \in \mathbb{N}^{2 \times n}$ and parameters are similar to those used in [17]. The weights w are

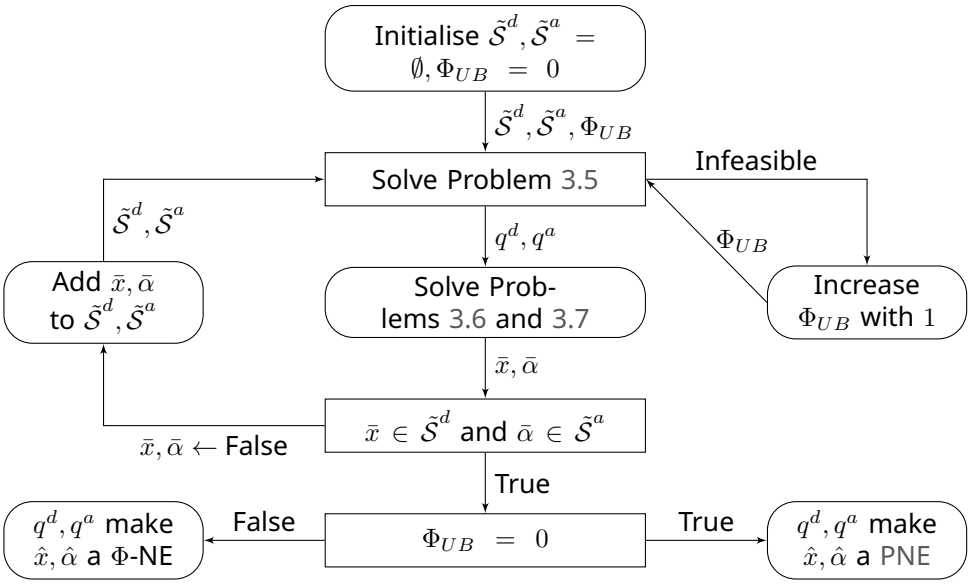

Figure 3.8: Visual representation of the payoff adjustment algorithm for a CNG instance.

generated as an integer uniform random matrix with entries in the range $[1, \ldots, 25]$ and the payoffs p are generated as the sum of w and an integer uniform random matrix with entries in the range $[1, \ldots, 25]$. The capacities are fixed at $D = 0.3 * \mathbf{1}w^d$ for the defender and $A = 0.03 * \mathbf{1}w^a$ for the attacker. The parameters $\epsilon = 1.25\eta, \delta = 0.8\eta$ depend on η. The time limit for adjustment was n seconds. The RAC is equal to $\sum_{i=1}^{n} |p_i^d - q_i^d| + |p_i^a - q_i^a| / \sum_{i=1}^{n} p_i^d + p_i^a$. The change in Φ is called $\Delta\Phi_{UB} = (\Phi_{UB} - \Phi_{new})/\Phi_{UB}$

I used two different methods to find the target solutions $\hat{x}, \hat{\alpha}$ for a given CNG instance. The first method uses the same heuristic as in the KPG section for finding a target solution. Each player takes turns to determine their best strategy and all unique strategies are recorded. This process finishes once both players repeat a previously played strategy. I will call instances with these type of solutions heuristic instances.

To find Φ-NE solutions, I used Zero Regrets to solve the problem twice, from the perspective of the defender using $f^d(x, \alpha)$ as objective and from the perspective of the attacker using $f^a(\alpha, x)$ as objective. This can lead to two different Φ-NE solutions $\hat{x}^d, \hat{\alpha}^d$ and $\hat{x}^a, \hat{\alpha}^a$, both of which were used as separate problems in the results section. If $\hat{x}^d, \hat{\alpha}^d = \hat{x}^a, \hat{\alpha}^a$, the results for adjusting $\hat{x}^d$ were duplicated to save computing time. Instances which had a PNE solution were rejected, since no adjustment is needed to achieve a PNE for those instances. After solving for 200 seconds, the Zero Regrets algorithm was stopped and its results were checked for viability and $\Phi_{UB} >= 1$. If they were not viable or $\Phi_{UB} < 1$, then they were also rejected.

The implementation of this section can be found in `methods/local_inverse_cng.py`. The results were generated using local_inverse_cng_phi_results.py and local_inverse_cng_greedy_results.py. These can be found in Tables A.6 and A.8 (Φ-NE results) and A.5 and A.7 (greedy results). Figures 3.10, 3.13, 3.9 and 3.12 were based on these tables.

Results Figure 3.9 contains the mean running times, mean RAC and mean Φ values for heuristic instances. The first thing we can see is the increase in mean running time. Going from 50 to 100 items already shows the effect of more variables on the running time, with

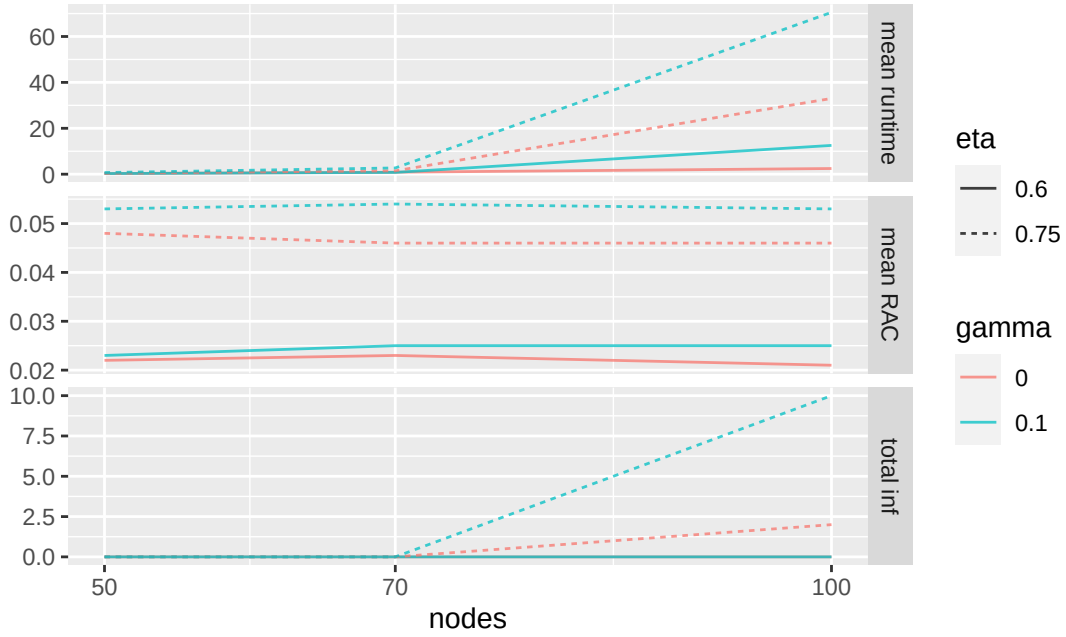

Figure 3.9: Mean running times, mean RAC, and total infeasible instances of payoff adjustment on heuristic CNG instances with 20 samples per combination.

a tenfold increase. The average running time of the $n = 100, \gamma = 0.75, \eta = 0.1$ instances was mostly influenced by the time limit, since it was unable to finish and find a PNE within the given time in 50% of the instances. Interestingly, in all other cases a PNE was found, with RAC values similar to those for the Φ-NE instances.

Figure 3.10 contains mean running times, mean RAC and mean relative changes in Φ. As you can see, the running times of the algorithm adjusting the payoffs are very low. However, the algorithm generating the Φ-NE solutions is very time consuming. For an instance with 50 items, obtaining a Φ-NE solution can take minutes while the inverse method takes fractions of seconds. Thus, the size of the experiments was mostly limited by generating the target solutions and not the method itself. We can see that the running times roughly double going from $n = 20$ to $n = 40$, which is a smaller increase than for the heuristic instances. We can also see that for the largest problems, the Φ_{UB} was decreased on average by around 85% and for the smallest problems the reduction was around 70%. Out of all instances, only in one case it was possible to turn a Φ-NE solution into a PNE, which shows how difficult this is.

3.3.2 Adjusting weights

Adjusting the weights of CNG instances is similar to the problem in Subsection 3.1.2, but with an important difference. CNG instances are difficult to transform to a PNE, especially when starting with Φ-NE solutions. Also, the Φ approach from the previous subsection where constraints can be relaxed by increasing Φ_{UB} does not work with the methods for adjusting the weights which I have used so far. These methods all reduce the size of a set, but this depends on the assumption from PNE that all possible vectors v^d, v^a which lead to solutions x, α with a higher payoff than the given solutions $\hat{x}, \hat{\alpha}$ can be rejected. If we have to also consider Φ-NEa, then this does not hold. Instead, only solutions for which $f(x, \alpha) > f(\hat{x}, \hat{\alpha}) +$

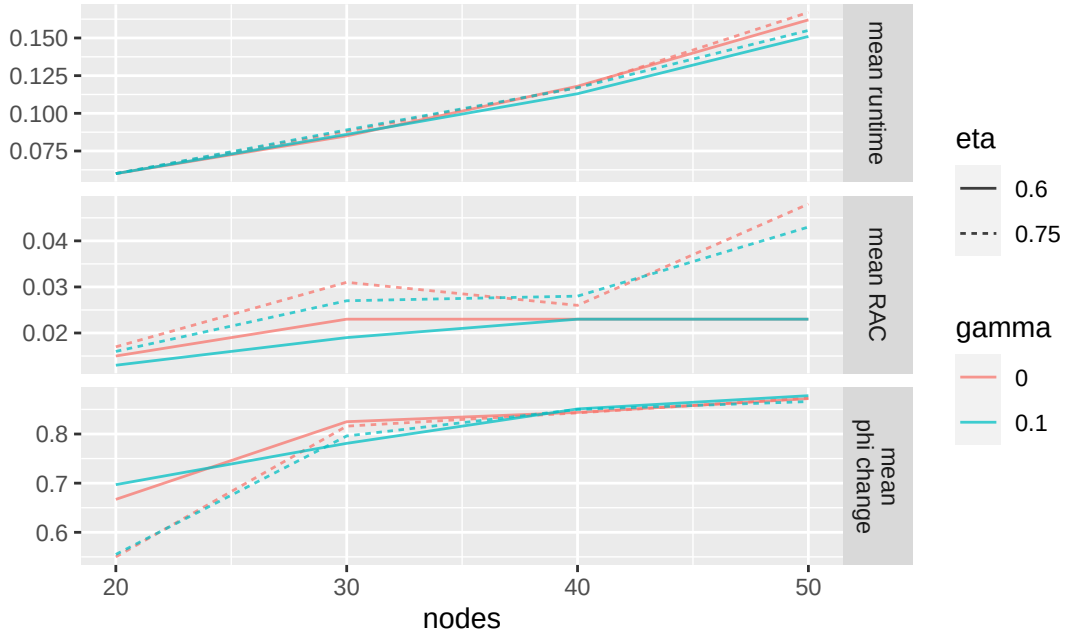

Figure 3.10: Mean running times, mean RAC and mean phi change of payoff adjustment on Phi-NE CNG instances with 10 samples per combination.

Φ is true can be rejected as impossible. However, we do not know Φ beforehand, so we need to find it together with the adjusted vectors v^d, v^a. To achieve this, we can start with the most restrictive $\Phi_{UB} = 0$ and increase Φ_{UB} by 1 if we determine the problem to be infeasible. Once the problem becomes feasible, we have found an upper bound integer Φ and we can adjust v^d, v^a. It is possible to further refine Φ since the payoff results of CNG players can be real valued. However, since the authors of the CNG also used integer Φ values [17], I decided to do the same here.

The master problem being solved is

$$\min_{v^d, v^a} \{\sum_{i=1}^{n} |w_i^d - v_i^d| + |w_i^a - v_i^a| : v^d \bar{x} > D \; \forall \bar{x} \in \tilde{\mathcal{S}}^d, v^a \bar{\alpha} > A \; \forall \bar{\alpha} \in \tilde{\mathcal{S}}^a, \\ v^d \hat{x}^d \leq D, v^a \hat{\alpha}^a \leq A, v^d \in \tilde{\mathcal{V}}^d, v^a \in \tilde{\mathcal{V}}^a\} \quad (3.8)$$

which aims to find two valid vectors v^d, v^a which are then tested using the adverse problems

$$\max_x \{f^d(x, \hat{\alpha}^d) : v^d x \leq D, x \in \{0,1\}^n\} \quad (3.9)$$

and

$$\max_\alpha \{f^a(\alpha, \hat{x}^a) : v^a \alpha \leq A, \alpha \in \{0,1\}^n\}. \quad (3.10)$$

The payoffs of the new solutions $\bar{x}$ and $\bar{\alpha}$ are checked against the payoffs of the given solutions. If $f^d(\bar{x}, \hat{\alpha}) \geq f^d(\hat{x}, \hat{\alpha}) + \Phi$, $\bar{x}$ can be rejected as impossible. By being added to $\tilde{\mathcal{S}}^d$, the constraint $v^d \bar{x} > D$ is added to Problem 3.8 and $\bar{x}$ becomes infeasible. The same applies to $\bar{\alpha}$. If the master problem is infeasible, Φ is increased by 1 and the procedure is repeated. See Figure 3.11 for a visual representation of the method.

Implementation The instances used in this section were generated in the same way as the instances in the previous section on payoff adjustment and implemented in the same

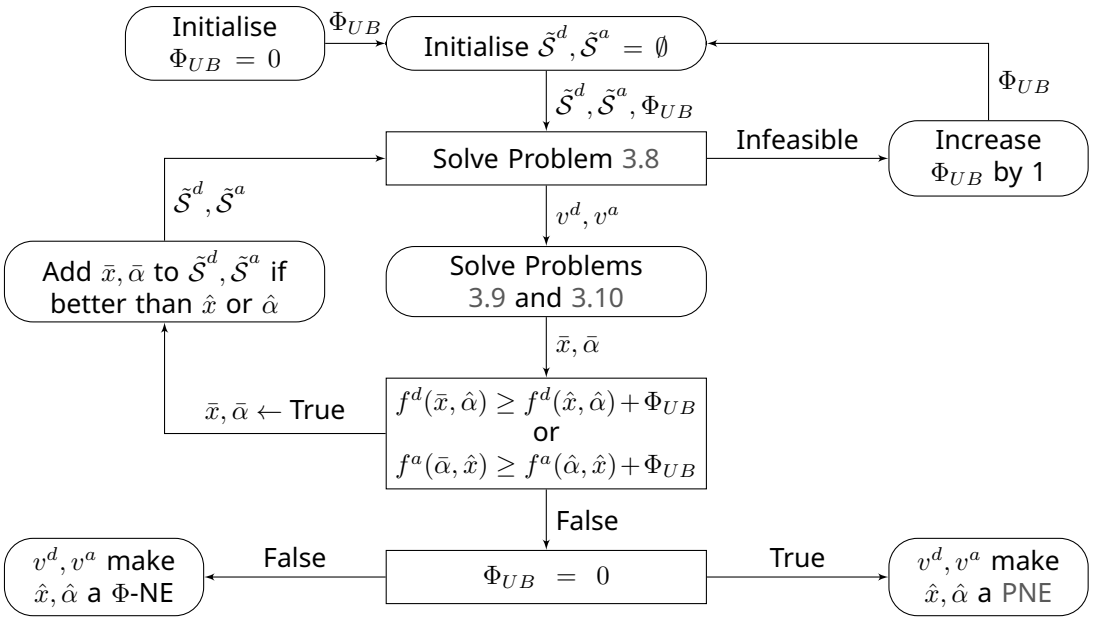

Figure 3.11: Visual representation of the weights adjustment algorithm for a KP instance.

files. The time limit for adjustment was n seconds. The RAC is equal to $\sum_{i=1}^{n} |w_i^d - v_i^d| + |w_i^a - v_i^a| / \sum_{i=1}^{n} w_i^d = w_i^a$. The change in Φ is called $\Delta\Phi_{UB} = (\Phi_{UB} - \Phi_{new})/\Phi_{UB}$

Results Figure 3.12 contains the mean running times, mean RAC and mean Φ values for heuristic instances. The first thing we can see is the increase in mean running time. Going from 100 to 200 items already shows the effect of more variables on the running time, with a more than tenfold increase. A second observation is that adjusting a Φ-NE is much more demanding than adjusting a greedy solution. Both the running times and the mean RAC for $n = 50$ are much higher in Figure 3.13 than in Figure 3.12.

Figure 3.13 contains mean running times, mean RAC, mean relative changes in Φ and the number of PNEa for Φ-NE instances. As you can see, the running time of adjusting weights is very low. However, the algorithm generating the Φ-NE solutions is very time consuming. When looking at these results, remember that for each instance, two solutions were generated, one maximising f^d and one maximising f^a. Thus, the 10 samples per combination resulted in 20 pairs of given solutions $\hat{x}, \hat{\alpha}$ for which the weights were adjusted.

What is interesting to note is that adjusting weights is clearly slower than adjusting payoffs (Figure 3.10). Both methods are running on the same CNG instances and at $n = 50$, you can see that the running time of adjusting weights is about ten times the running time of adjusting payoffs. Note also that the running times of adjusting weights do not strictly increase for all parameter combinations, there seems to be a lot of variance in the running depending on the exact characteristics of the problem. Another big difference between adjusting weights and payoffs is that the weights method is able to turn the solutions into PNEa in many cases, while for payoffs this is really an exception.

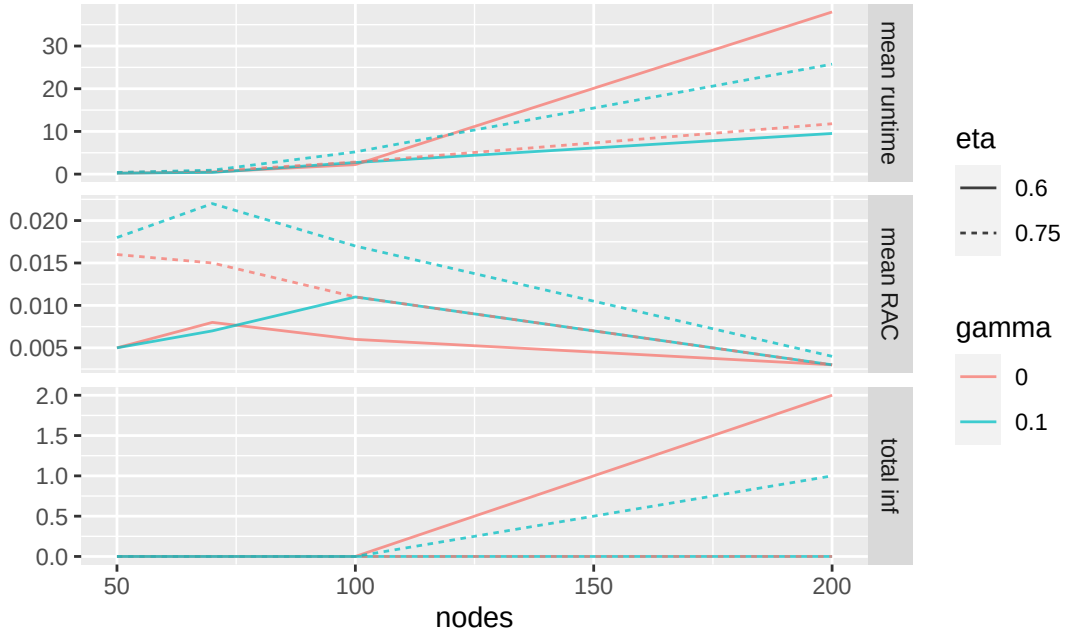

Figure 3.12: Mean running times, mean RAC and total infeasible instances of weight adjustment on heuristic CNG instances with 20 samples per combination.

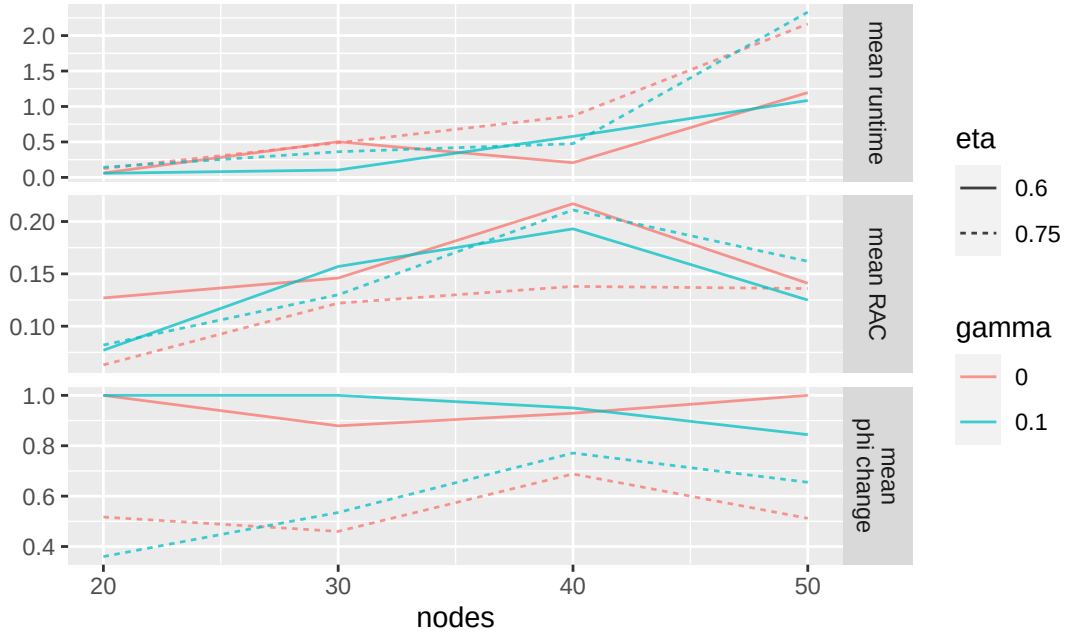

Figure 3.13: Mean running times, mean RAC and mean Phi change of weights adjustment on Phi-NE CNG instances with 10 samples per combination.

3.4 Chapter conclusion

In this chapter, I looked at adjustment inverse optimisation on the KP, KPG and CNG. The results of this section provide initial answers to my research questions. The results from Section 3.1 show that adjustment inverse optimisation of both the objective function and constraints can be applied to a classic IP with thousands of variables, which means that the answer to **SQ1** is at least partially yes. In the next Chapter, we will see if this also holds true for reconstruction inverse optimisation.

The results from Sections 3.2 and 3.3 show that adjustment inverse optimisation with the goal of turning a given target feasible solution $\hat{x}$ into a PNE solution is possible for both the KPG and the CNG, but that especially for the CNG, there is no guarantee that it is possible to achieve a PNE. In this Chapter, I adapted both Wang's and Cronert's methods and these are both suitable for adjustment inverse optimisation on IPGs, thus providing a partial answer to **SQ2**.

The results from the previous sections also show that adjustment inverse optimisation can be done on both the constraints and the payoffs of both KPG and CNG instances. For KPG instances, adjusting the weights and adjusting the payoffs for instances with $n = 4$ and $m = 1000$ can be done in less than 30 seconds if all the interactions are positive. If the values of I can also be negative, then the running times quickly increase. Instances with $n = 2$ and $m = 100$ often hit the time limit of 60 seconds and the minimum running times were still three times higher than of comparable instances without negative interactions.

For CNG instances, I used two different sources for target solutions, heuristic and Φ-NE. For the Φ-NE solution, only adjusting the weights makes it possible to turn some of the target solutions into PNEa. The increases in running time were a bit higher for the weights adjustment compared to the payoffs adjustment. Unfortunately, due to the time requirements of computing the Φ-NE, I was unable to test instances with more than $n = 50$. This was possible for the heuristic instances and here we can see that adjusting weights and payoffs both can often find PNEa. The weights adjustment dealt better with adding more nodes, but at $n = 200$ sometimes the algorithm ran into a time limit. In terms of the required adjustments, the payoffs adjustment is similar for Φ-NE and heuristic instances, but for the weights adjustment it is clear that the Φ-NE instances require much more changes.

Thus, to partially answer **SQ3**, adjustment inverse optimisation can be applied to non-trivial instances of IPGs like KPG and CNG. The size of the instances, in terms of the number of items and/or players, which can be solved within a given time depends on the type of adjustment being performed, the problem and the other parameters of the problem instance.

The methods in this chapter can adjust non-trivial KPG and CNG instances with given solutions to PNE and Φ-NE solutions. The size of the instances which can be solved in a given time depends on the problem. For the KPG, adjustment of instances with $n = 4, m = 1000$ can be done less than 30 seconds on average if the interactions are positive. These instances are larger than the KPG instances in [18]. For the CNG, payoff adjustment of instances with $n = 100$ and weight adjustment of instances with $n = 200$ can be solved in similar times. These instances are comparable the synthetic instances in [17]. Thus, the partial answer to **SQ4** is that these methods work on the instance sizes mentioned in the literature.

In the next Chapter, I will move to reconstruction inverse optimisation, where the goal is to determine the payoffs or weights of a certain problem based on repeated plays of that problem where the other attributes of the problem vary.

Chapter 4

Reconstruction multi-observation inverse optimisation

In the previous chapter, I focused on adjusting payoffs and constraints based on a single observation. However, what if we do not have a starting point and instead get a set of observations of a problem? In this scenario, we can try to reconstruct the payoff or constraints with the goal of making all the observed solutions as close to optimal as possible. Cronert's method is suitable to learn the payoffs for IPGs and can deal with Φ-NE and PNE solutions, but it needs to be adjusted to learn constraints and to deal with Φ-NE when learning constraints. In this section, I will investigate the application of Cronert's method to KP, KPG and CGN.

In this chapter, problem instances will always be a set of observations of size o. All observations in a set share either their payoffs or their weights and the aim is to reconstruct one of these two. The other attributes of the instances in a set of observations will vary but are considered known and available for solving the problem. I will be used the same value, RAC, to evaluate the results in this chapter. However, in this chapter the RAC measures how close the results of a specific experiment were to the true value and not how much change was required to make an instance optimal.

4.1 Inversing the Knapsack Problem

While KP is not really an IPG, we can treat it as an IPG with only one player. Thus, we can apply Cronert's method to it. Instead of PNE solutions, we can use optimal solutions and the goal is to find the payoffs $\hat{p}$ or the weights $\hat{w}$ which makes all observations optimal.

4.1.1 Reconstructing payoffs

How can we find the payoffs $\hat{p}$ which makes a set of observations of KP instances optimal? This can be done by applying Cronert's method as an extension of the approach in Subsection 3.1.1. Given a set of observations $\mathcal{O}$ of related KP problems with n items where for each observation $o \in \mathcal{O}$ the optimal solution $\hat{x}^o \in \{0,1\}^n$, varying weights $w^o \in \mathbb{N}^n$ and varying capacity $c^o \in \mathbb{N}$ are known and the magnitude of the payoffs $P = \sum \hat{p}$ is also known, the goal

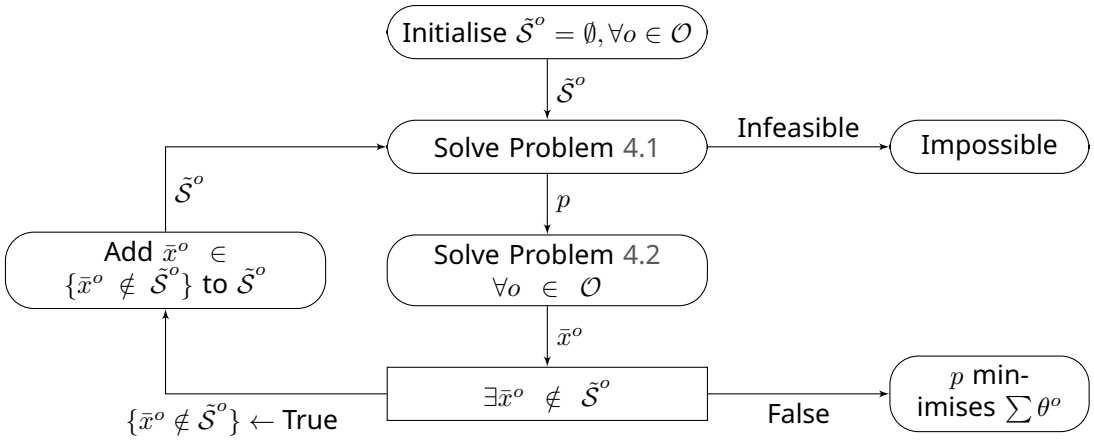

Figure 4.1: Visual representation of payoff reconstruction algorithm for a KP instance.

is to find unknown payoffs $\hat{p} \in \mathbb{N}^n$ by solving the master problem

$$\min_p \{\sum_{o \in \mathcal{O}} \theta^o : \theta^o \geq \bar{x}^o p - \hat{x}^o p \quad \forall \bar{x}^o \in \tilde{\mathcal{S}}^o \, \forall o \in \mathcal{O}, \quad \sum p = P, p \in \mathbb{N}^n\} \tag{4.1}$$

where the proposed payoffs p are tested by solving

$$\max_x \{px : w^o x \leq c^o\} \tag{4.2}$$

for each observation $o \in \mathcal{O}$ resulting in k solutions $\bar{x}^o$. All solutions are checked against the set of solutions $\tilde{\mathcal{S}}^o$ of their observation and added if they are new. If none of the solutions are new, then the current payoffs p is optimal. This pattern is similar to the previous chapter, but now all observations have to be checked. For a visual overview, see Figure 4.1.

Implementation For the problems in this section, the payoffs $\hat{p} \in \mathbb{N}^n$ are generated as an integer uniform vector with entries in the range $[1, \ldots, 100]$ and are constant for all observations. The magnitude $P = \sum_{i=1}^n \hat{p}_i$ is considered known. For each observation o weights w^o are generated as an integer uniform vector dependent on $\hat{p}$ with entries in the range $[\max(1, \hat{p} - \lceil r/5 \rceil), \ldots, \min(r, \hat{p} + \lfloor r/5 \rfloor)]$. The capacity c^o depends on the weights w^o. The time limit for reconstruction was set at $|\mathcal{O}|/2$ seconds. The RAC is equal to $\sum_{i=1}^n |\hat{p}^i - p^i|/P$

The solutions $\hat{x}$ are generated by solving the KP instance $\max_x \{px : w^o x \leq c_0, x \in \{0,1\}^n\}$. These choices were inspired by [32].

The implementation of this section can be found in `methods/inverse_kp.py`. The results were generated with inverse_kp_results.py and inverse_kp_results_rc.py. These can be found in Tables B.1 and B.2. Figure 4.2 is based on Tables B.1 and B.2.

Results Figure 4.2 contains the results of reconstructing the payoffs on problems with 20, 40 and 60 items and. For each value of n, $8n$ observations were generated and the table shows the effect of increasing the number of observations. We can see two clear trends in the table: more observations increases the average running time and more observation decreases the mean RAC. Even the highest RAC values are below 0.2, meaning the absolute error at $\frac{1}{2}n$ observations is already less than 20%. Increasing the number of observations leads to a measurable improvement of the result, but the highest improvements can be

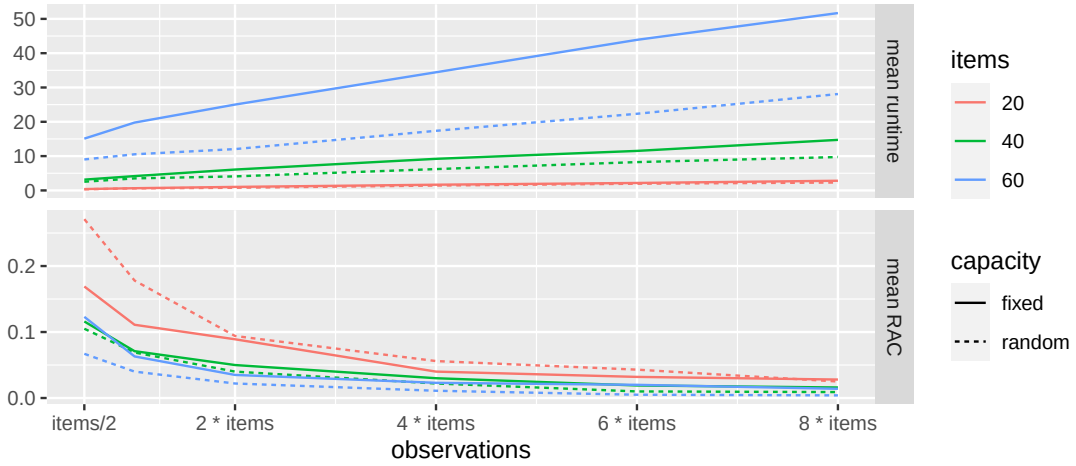

Figure 4.2: KP payoff mean reconstruction running times (in seconds) and mean RAC over 5 samples.

seen when going from $\frac{1}{2}n$ to n observations, halving the RAC for $n = 60$. Going to $2n$ observations also decreases the RAC again but the absolute change goes down quickly. For these experiments, 5 batches of $\max\{|\mathcal{O}|\}$ were generated for each value of n and the first $|\mathcal{O}|$ observations were used for each set of results.

The dashed lines in Figure 4.2 contains the results of reconstructing payoffs if the capacities vary. This means that instead of $c^o = 0.5 \sum w^o$, each observation has a random capacity fraction h^o which is generated as a uniform random value in the range $[0.2, 0.8]$, so $c^o = h^o \sum w^o$. This variation leads to more varied solutions, since some solutions can only include the most valuable items while others can also included the more generic items. If we compare the results for fixed and random capacities, we can see a clear decrease in the running time, but not necessarily lower errors, for random capacities. The errors for $n = 20$ are higher until $8n$ observations, but for the $n = 40$ problems we can see a clear improvement across the board and this trend continues for $n = 60$.

4.1.2 Reconstructing weights

How can we find weights w which makes a set of observations of KP instances optimal? We can extend the approach from Subsection 3.1.2 by extending the algorithm to optimising multiple observations.

Given a set of observations $\mathcal{O}$ of a KP problem with n items where for each observations $o \in \mathcal{O}$ an optimal solution $\hat{x}^o \in \{0,1\}^n$, varying payoffs $p^o \in \mathbb{N}^n$, the magnitude $W = \sum w$ and capacities $c^o \in \mathbb{N}^{|\mathcal{O}|}$ are known, the goal is to find the unknown weights $\hat{w} \in \mathbb{N}^n$ by solving the master problem

$$\min_w \{1 : \bar{x}^o w > c^o \quad \forall \bar{x}^o \in \tilde{\mathcal{S}}^o \forall o \in \mathcal{O}, \quad \sum_{i=1}^{n} w_i = W, w \in \mathbb{N}^n\} \tag{4.3}$$

where the proposed weights w are tested by solving

$$\max_x \{p^o x : wx \leq c^o\} \tag{4.4}$$

for each observation $o \in \mathcal{O}$ resulting in $|\mathcal{O}|$ solutions $\bar{x}^o$. All solutions are checked against the set of solutions $\tilde{\mathcal{S}}^o$ of their observation and added if they are new. If none of the solutions

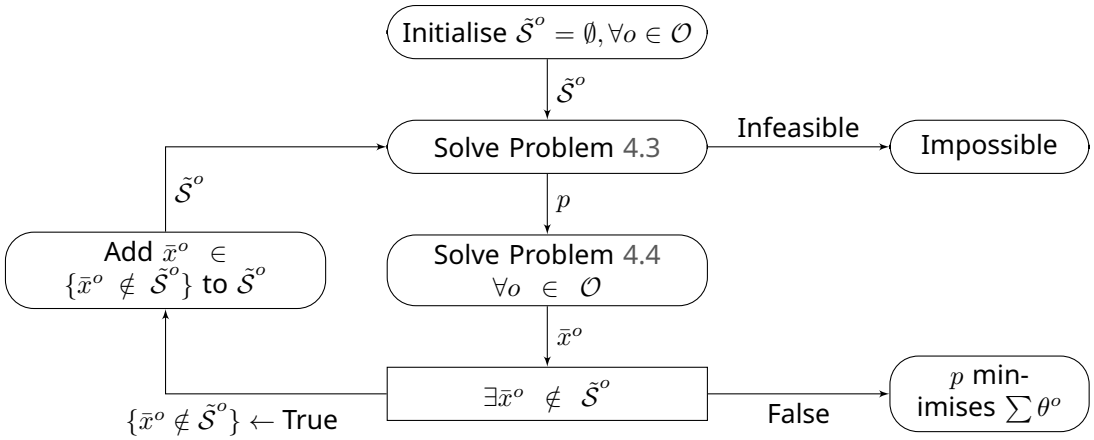

Figure 4.3: Visual representation of payoff reconstruction algorithm for a KP instance.

are new, then the current weights w are optimal. This pattern is similar to that of last chapter, but now all observations have to be checked. For a visual overview, see Figure 4.3.

Implementation The instances used in this section were generated in a similar way as the instances in the previous section, with an important difference. Now the weights $\hat{w} \in \mathbb{N}^n$ are fixed and generated as an integer uniform random vector with entries in the range $[1, \ldots, 100]$. The magnitude $W = \sum_{i=1}^{n} \hat{w}_i$ is considered known. For each observation o, new payoffs p^o are generated as an integer uniform vector dependent on $\hat{w}$ with entries in the range $[\max(1, \hat{w} - \lceil r/5 \rceil), \ldots, \min(r, \hat{w} + \lfloor r/5 \rfloor)]$. The capacities c^o are related to w. The time limit for reconstruction was set at $|\mathcal{O}|/2$ seconds. The RAC is equal to $\sum_{i=1}^{n} |\hat{w}_i - w_i|/W$

The implementation of this section can be found in `methods/inverse_kp.py`. The results were generated with inverse_kp_results.py and inverse_kp_results_rc.py. These can be found in Tables B.3 and B.4. Figure 4.4 is based on Tables B.3 and B.4.

Results Table B.3 contains the results for KP problems with 20, 40 and 60 items. If we compare the results for reconstructing the weights to those of Figure 4.2, we can see some differences. The range of values for $|\mathcal{O}|$ is smaller than for payoffs, but the range of results is larger. For $n = 20$, the reconstruction is almost perfect and the running times are going down when more observations are added. The same happens for $n = 40$, perfect reconstruction with decreasing running times. Once we reach $n = 60$ however, the perfection has run out. Now, the first result is almost worse than a vector with only 0. If we increase the number of observations, the error goes down rapidly, but even with the highest number of observations, the error is still larger than the highest payoff error or the starting error of $n = 40$. Based on these findings, it is unclear what amount of observations would be required to bring the mean RAC for $n = 60$ close to the perfection of the lower two item counts.

The dashed lines in Figure 4.4 represent the results for KP problems where each observation has a random capacity fraction h^o which is generated as a uniform random value in the range $[0.2, 0.8]$, so $c^o = h^o \sum w$. If we compare these results to their solid counterparts, there are no major differences for $n = 20$, but for $n = 40$, we can see the running times going down much quicker and the error reaching 0 earlier. When the number of items is increased to $n = 60$, we can see that the tipping point for the running time is crossed at $4n$, with a large drop in running time and perfect reconstruction. This suggests that for the results with fixed

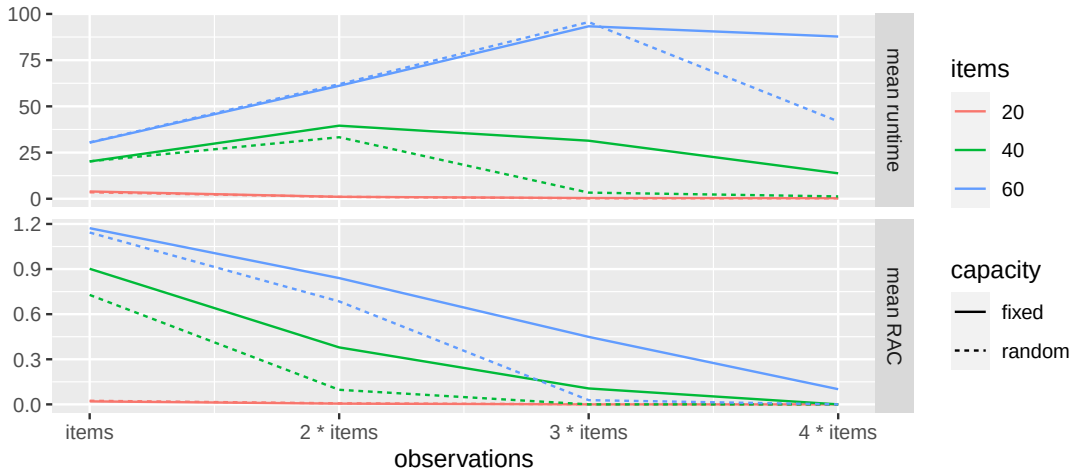

Figure 4.4: KP weight reconstruction running times (in seconds) and RAC over 5 samples.

capacities, there should be an observation count for $n = 60$ where the error is also 0.

4.2 Inversing the Knapsack Packing Game

Inversing the KPG extends the methods from the previous section. The KPG setting introduces two complicating factors: the solutions are now PNEa or Φ-NEa, instead of the optimal solutions of the previous section, and since each KPG has multiple players, these all have to solve their problem individually which increases the complexity and solving time of the problem. The goal of the next two subsections is to find the payoffs $\hat{p}$ or weights $\hat{w}$ which ensure that the observations $\hat{x}^o$ are PNEa or Φ-NEa.

4.2.1 Reconstructing payoffs

How can we find the payoffs $\hat{p}$ of a group of related KPG problems using a set of observations $\mathcal{O}$ with PNEa or Φ-NEa solutions $\hat{x}^o$. We can use Cronert's method for this, which can deal with both PNEa and Φ-NEa solutions, and expand the implementation in Subsection 4.1.1 to allow for multiple players.

Given a set of observations $\mathcal{O}$ of a KPG problem with n players and m items where for each observation $o \in \mathcal{O}$ a solution $\hat{x}^o \in \{0,1\}^{n\times m}$, interactions $I^o \in \mathbb{N}^{n\times n\times m}$, weights $w^o \in \mathbb{N}^{n\times m}$ and capacities $c^o \in \mathbb{N}^{|\mathcal{O}|\times n}$ are known, the goal is to find the unknown payoffs $\hat{p} \in \mathbb{N}^{n\times m}$ by solving the master problem

$$\min_p \{\sum_{o\in\mathcal{O}}\sum_{i=1}^{n}\theta_i^o : \theta_i^o \geq f^i(\bar{x}_i^o, \hat{x}_{-i}^o) - f^i(\hat{x}_i^o, \hat{x}_{-i}^o) \quad \forall \bar{x}_i^o \in \tilde{\mathcal{S}}_i^o \, \forall i \in \{1,\dots,n\} \forall o \in \mathcal{O}, \\ p \in \mathbb{N}^{n\times m}\}, \quad (4.5)$$

where the payoff function f^i of player i looks like

$$f^i(x^i, x^{-i}) = p^i x^i + \sum_{k=1,k\neq i}^{n} (x^i \odot x^k) I_k^i,$$

and the proposed payoffs p are tested by solving

$$\max_x \{f^i(x, \hat{x}_i^o) : w_i^o x \leq c_i^o, x \in \{0,1\}^n\},$$

for all observations $o \in \mathcal{O}$ and all players $i \in \{1,\dots,n\}$ resulting in $n|\mathcal{O}|$ new solutions $\bar{x}_i^o$. Each solution $\bar{x}_i^o$ is checked against the approximate solution set $\tilde{\mathcal{S}}_i^o$ and added if it is a new solution. If none of the solutions were added to their approximate solution set, the algorithm finishes with the current p. Otherwise, the master problem is solved again and the process is repeated. If all instances have PNE solutions, it should be possible to reduce the sum of all θ_i^o to 0 in the master problem. However, if some of the observations have Φ-NE solutions, these can add a lower bound of Φ to the corresponding value of θ_i^o. In that case, minimising $\sum \theta_i^o$ to that lower bound still allows us to reconstruct p.

Implementation For the problems in this section, the payoffs $\hat{p} \in \mathbb{N}^{n\times m}$ are the same for all observations and generated as an integer uniform random matrix with entries in the range $[1,\dots,100]$. For each observation o, new weights $w^o \in \mathbb{N}^{n\times m}$ are generated as an integer uniform random matrix conditional on p with entries in the range $[\max(1, \hat{p} - 20),\dots,\min(100, \hat{p} + 20)]$. The capacities c_i^o depend on w_i^o and are either fixed at 50%, so $c_i^o = 0.5\sum w_i^o$ or the fractions $h \in [0,1]^{|\mathcal{O}|\times n}$ are randomly generated as an uniform random matrix with entries in the range $[0.2,\dots,0.8]$, resulting in $c_i^o = h_i^o \sum w_i^o$. The interactions

$I \in \mathbb{N}^{n \times n \times m}$ are generated as a random uniform matrix with entries in the range $[0, \ldots, 33]$. The time limit for reconstruction was $n * |\mathcal{O}|/2$ seconds. The RAC is equal to $\sum_{i=1}^{n} \sum_{j=1}^{m} |\hat{p}_j^i - p_j^i| / \sum_{i=1}^{n} \sum_{j=1}^{m} \hat{p}_j^i$.

For the experiments, $8m$ valid observations are generated and solved using Zero Regrets. The experiments with lower $|\mathcal{O}|$ values use the first $|\mathcal{O}|$ of the list of observations. The generation method limits the Zero Regrets solution time to m seconds per instance, and if no stable solution is found then a new problem is generated. In the first set of results, only problems with PNE solutions were accepted. The second table contains the results when Φ-NE solutions are also allowed. These two sets of results were generated using different seeds and thus are independent.

The implementations of this section can be found in `problems/inverse_kpg.py`. The results in Tables B.5 and B.6 were generated with inverse_kpg_results.py. The results in Table B.7 were generated with inverse_kpg_results_rc.py. Figures 4.5, 4.6 and 4.7 are based on Tables B.5, B.6 and B.7.

Results Figure 4.5 contains the results for KPG problems with 10, 20 and 25 items and 2 or 3 players where the solutions of all problems are PNEa. The solutions for the problems in Figure 4.6 are either Φ-NEa or PNEa. The relative absolute change RAC. When comparing the two tables, there does not seem to be a big difference between using only PNE solutions or also allowing Φ-NE solutions. A general trend in both cases is that the additional observations due to increasing the number of items is well used. There almost seems to be a relation between the absolute number of observations and the mean RAC. For instance, the mean RAC for $m = 10, |\mathcal{O}| = 2m$ is very similar to the mean RAC for $m = 20, |\mathcal{O}| = m$ in both tables. The same holds when doubling $|\mathcal{O}|$ and when tripling. The problems with more items get closer to $\hat{p}$ with the same absolute number of observations.

The mean running time increases as $|\mathcal{O}|$, which is to be expected. Increasing the number of observations $|\mathcal{O}|$ leads to a linear increase in running time. Adding a third player leads to bigger differences as the number of observations increases, from a 50% increase to almost 100% when going from m to $8m$ observations.

Figure 4.7 shows what happens if the capacities of problems are random instead of fixed and Φ-NE solutions are also allowed. The mean running times are slightly lower, but the difference in mean RAC is quite considerable, around 0.04 at the highest observations and even higher before that. This makes sense, since the varying capacity gives more information about the items with the highest payoffs.

The time required for solving the underlying KPG problems was a big limiting factor in the size of these experiments, that is the main reason that all instances are quite small.

4.2.2 Reconstructing weights

How can we find the weights $\hat{w}$ of a group of related KPG problems using a set of observations $\mathcal{O}$ with PNEa solutions $\hat{x}^o$? We can use the adaptation of Cronert's method of Subsection 4.1.2 and expand it to allow for multiple players.

Given a set of observations $\mathcal{O}$ of a KPG problem with n players and m items where for each observation $o \in \mathcal{O}$ a solution $\hat{x}^o \in \{0,1\}^{n \times m}$, payoffs $p^o \in \mathbb{N}^{n \times m}$, interactions $I^o \in \mathbb{N}^{n \times n \times m}$, magnitudes $W^i = \sum_{j=1}^{m} \hat{w}_j^i \forall i \in \{1, \ldots, n\}$ and capacities $c \in \mathbb{N}^{|\mathcal{O}| \times n}$ are known, the goal is to find unknown weights $\hat{w} \in \mathbb{N}^{n \times m}$ by solving the master problem

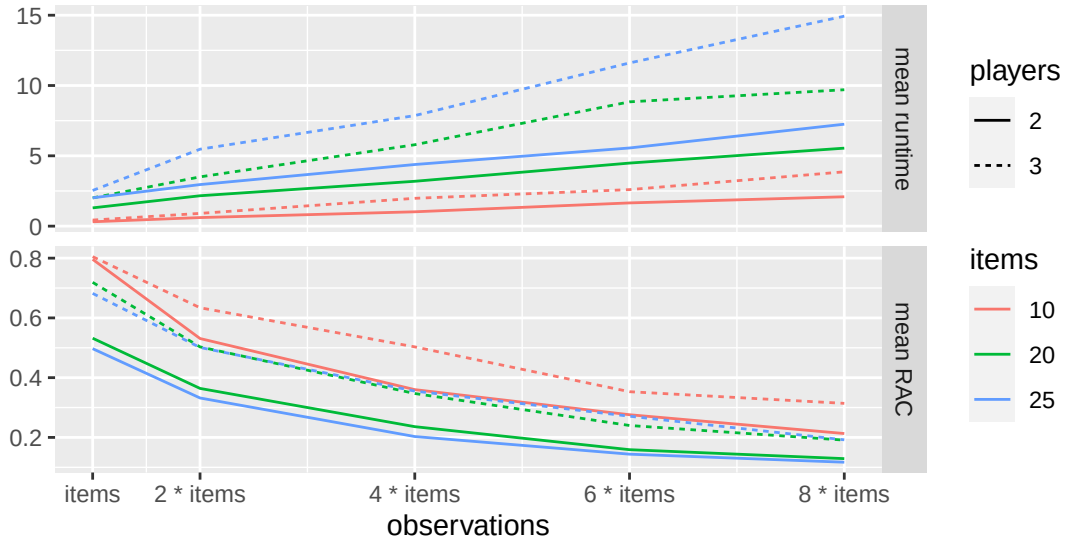

Figure 4.5: Mean running times and mean RAC of payoff reconstruction on KPG problems with PNE solutions, fixed capacities and 5 repeats per combination.

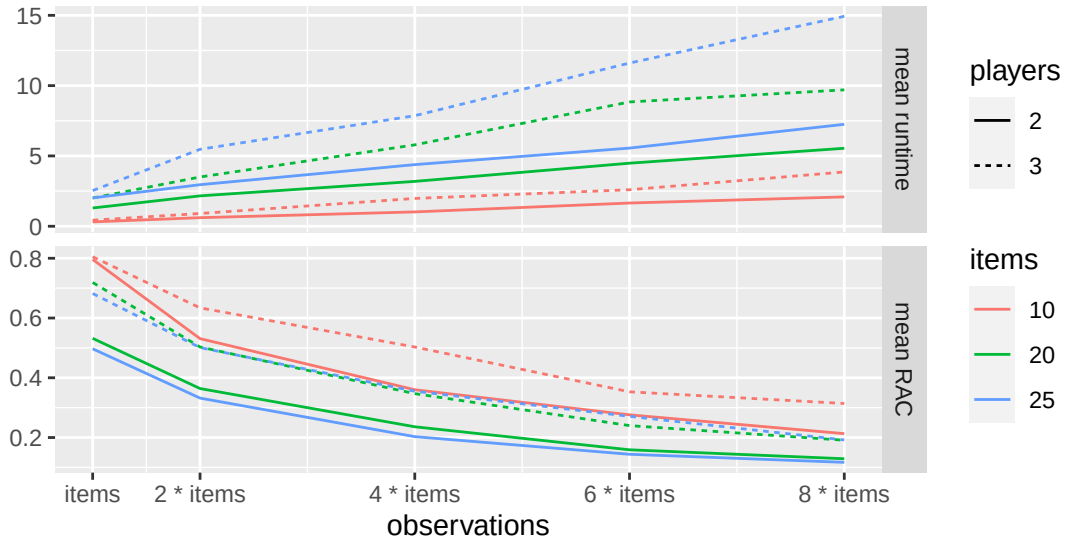

Figure 4.6: Mean running times and mean RAC of payoff reconstruction on KPG problems with approximate-NE and PNE solutions, fixed capacities and 5 repeats per combination.

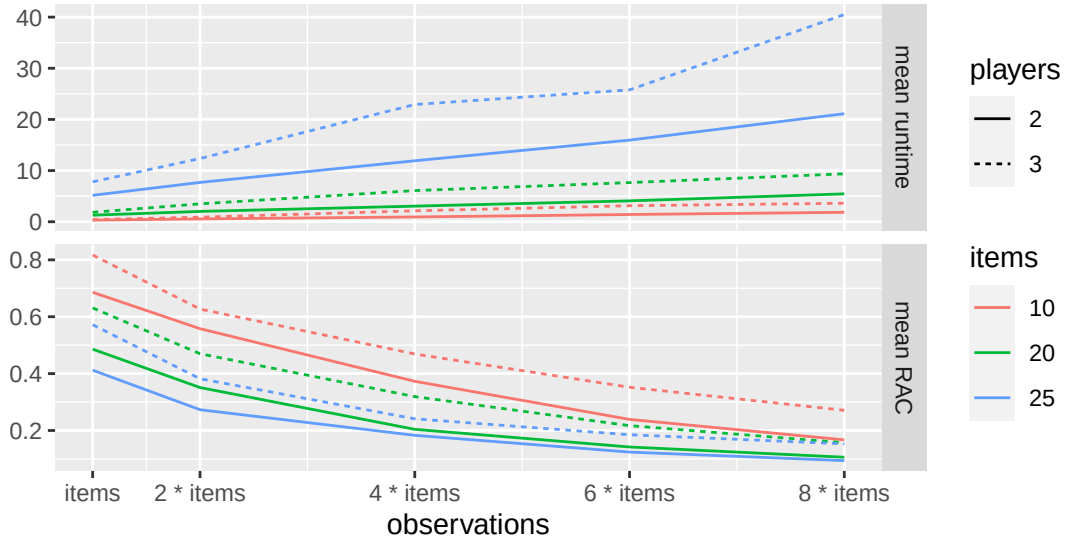

Figure 4.7: Mean running times and mean RAC of payoff reconstruction on KPG problems with approximate-NE and PNE solutions, random capacities and 5 repeats per combination.

$$\min_{w}\{1 : w^i \bar{x}_i^o > c_i^o \quad \forall \bar{x}_i^o \in \tilde{\mathcal{S}}_i^o \forall i \in \{1,\dots,n\}\ \forall o \in \mathcal{O}, \quad w^i x_i^o \le c_i^o \quad \forall i \in \{1,\dots,n\} \forall o \in \mathcal{O},$$
$$\sum_{j=1}^{m} w_j^i = W^i \quad \forall i \in \{1,\dots,n\}, \quad w \in \mathbb{N}^{n\times m}\}, \quad (4.6)$$

where the proposed weights w are tested by solving

$$\max_{x}\{f_i^o(x, \hat{x}_i^o) : w^i x \le c_i^o, x \in \{0,1\}^n\},$$

for all observations $o \in \mathcal{O}$ and all players $i \in \{1,\dots,n\}$ resulting in $n|\mathcal{O}|$ new solutions $\bar{x}_i^o$. These new solutions are compared against the known solutions $\hat{x}_i^o$ using

$$f_i^o(\bar{x}_i^o, \hat{x}_{-i}^o) > f_i^o(\hat{x}_i^o, x_{-i}^o).$$

If this is true, then $\bar{x}_i^o$ is added to $\tilde{\mathcal{S}}_i^o$ and made infeasible when solving the master problem in the next iteration, since all solutions are PNEa. If none of the solutions were added to their $\tilde{\mathcal{S}}$, the algorithm finishes with the current w. Otherwise, the master problem is solved again and the process is repeated.

Implementation For the problems in this section, the weights $\hat{w} \in \mathbb{N}^{n\times m}$ are the same for all observations and generated as an integer uniform random matrix with entries in the range $[1,\dots,100]$. The capacities c^i are either fixed at 0.5 times the total weight ($c_i^o = 0.5\sum w^i, \forall i \in \{1,\dots,n\}$) and the same for all observations or the fractions $h \in [0,1]^{|\mathcal{O}|\times n}$ are randomly generated as an uniform random matrix with entries in the range $[0.2,\dots,0.8]$, resulting in $c_i^o = h_i^o \sum w^i$. For each observation o, new payoffs $p^o \in \mathbb{N}^{n\times m}$ are generated as an integer uniform random matrix conditional on w with entries in the range $[\max(1, \hat{w} - 20),\dots,\min(100, \hat{w} + 20)]$. The interactions $I \in \mathbb{N}^{n\times n\times m}$ are generated the same as in the previous section. The time limit for reconstructing was $n * |\mathcal{O}|/2$ seconds. The RAC is equal to $\sum_{i=1}^{n}\sum_{j=1}^{m}|\hat{w}_j^i - w_j^i| / \sum_{i=1}^{n}\sum_{j=1}^{m}\hat{w}_j^i$.

For the experiments, $6m$ valid observations are generated and solved using Zero Regrets. The experiments with lower $|\mathcal{O}|$ values use the first $|\mathcal{O}|$ of the list of observations. The generation method limits the Zero Regrets solution time to m seconds per instance, and if no

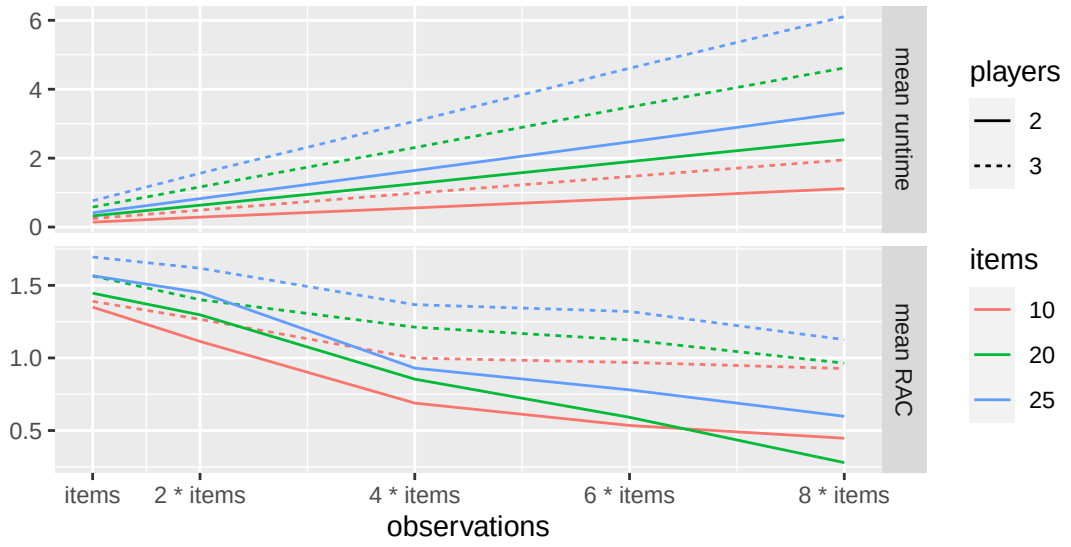

Figure 4.8: Mean running times and mean RAC of weight reconstruction on KPG problems with PNE solutions, fixed capacities and 5 repeats per combination.

stable solution is found then a new problem is generated. Only problems with PNE solutions were accepted, since Φ-NE solutions do not have the characteristics such that solutions with a higher payoff than the known solution can be rejected, which is the basis for this method.

The implementations of this section can be found in `problems/inverse_kpg.py`. The results in Table B.8 were generated with inverse_kpg_results.py and the results in Table B.9 with inverse_kpg_results_rc.py. Figures 4.8 and 4.9 are based on Tables B.8 and B.9.

Results Figure 4.8 shows the mean running times and mean RAC for different player counts, item counts and with fixed capacities. Looking at the running times first, the effect of more items, more players and more observations is in line with expectations. Doubling the items doubles the running time, doubling the observations doubles the running time and adding a third player increases the running time by between 70 and 80 percent.

Looking at the mean RAC, we can see that it decreases, with the higher item counts starting higher but ending lower when more observations can be used. For $n = 2$, we can see this happening at $8m$ items for fixed capacities and $4n$ items for random capacities. For $n = 3$, more observations would be necessary to see the same crossover and in case of the random capacities, it is the question where the crossover will be. Adding a third player has a clear effect on the RAC, with higher starting values at m observations and less improvements when more items are added.

If we compare the weight results against the payoff results, the first thing to note is the differences in running times. Reconstructing the weights is about twice as fast as reconstructing the payoffs. However, the errors of reconstructing the payoffs are considerably lower than those of weights instances with similar parameters.

Figure 4.9 shows what happens if the capacities are random instead of fixed. The running times stay roughly the same as in Figure 4.8, but the errors decrease considerably. The effect is much stronger here compared to the difference between the fixed and random capacities for the payoffs.

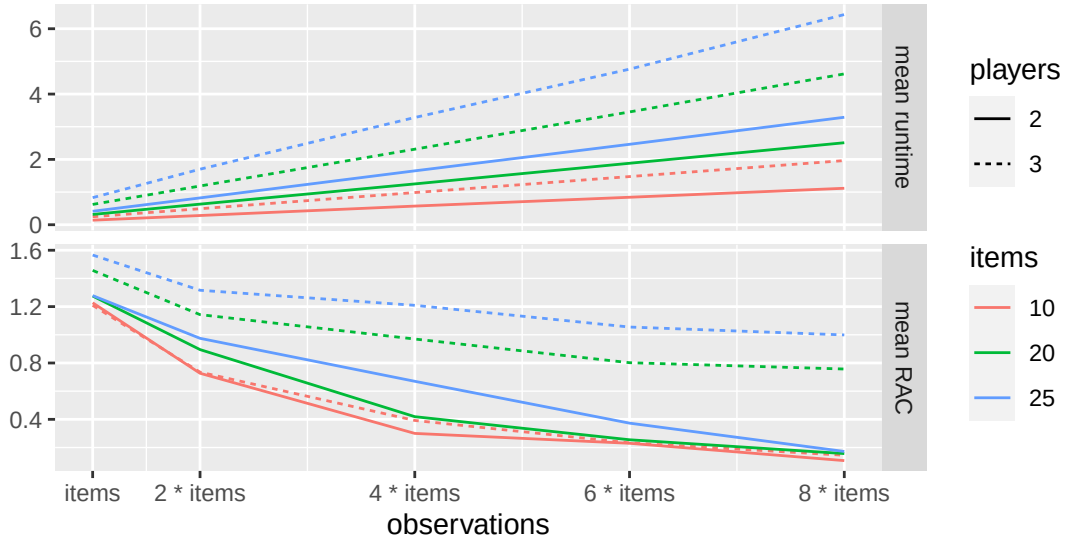

Figure 4.9: Mean running times and mean RAC of weight reconstruction on KPG problems with PNE solutions, random capacities and 5 repeats per combination.

Just like in the previous section was the time required to generate the observations and solutions a major limitation in scaling these experiments. The weights reconstruction process requires much more observations than the payoffs reconstruction process, especially with $m = 3$.

4.3 Inversing the Critical Node Game

Just like in the previous chapter, I will try to apply the techniques of the previous two section to the CNG. At a first glance, the CNG seems comparable to the KPG. Two players each have a set of nodes with a corresponding weight and payoff. Each player also has a limited capacity to select the nodes they want to defend and the payoff function is based on the choices of the two players. This is where the similarities stop, since the payoff functions of the two players are very different from the payoff functions used in the KPG. The functions are not symmetric in their construction and each combination of action and inaction of the two players has a different outcome. For instance, the defensive player gets a reward if they do not defend a node that is not being attacked. See Table 4.1 for the payoffs of the different interactions.

Table 4.1: Rewards table for defender (left) and attacker (right) for all four possible combinations of strategies associated with each node i, where $\delta < \eta < \epsilon$. Same as Table 2.1.

	$\alpha_i = 0$		$\alpha_i = 1$	
$x_i = 0$	p_i^d	$-\gamma p_i^a$	δp_i^d	p_i^a
$x_i = 1$	ϵp_i^d	0	ηp_i^d	$(1-\eta)p_i^a$

A second challenge of the CNG when compared to KPG instances is the lack of PNE solutions. While KPG instances without negative interactions usually have a PNE, or otherwise a Φ-NE with Φ-values of 1 or 2, CNG instances rarely have a PNE solution and in most cases, their Φ-NE solutions have values of Φ much higher than of KPG instances. This means that for a

given solution $\hat{x}, \hat{\alpha}$, there are many possible alternative solutions for both the attacker and the defender which give a better result than the given solution $\hat{\alpha}$.

In this section, I will attempt to reconstructing the payoffs, weights and payoff parameters of CNG instances. The last task is the most similar to the original task in [13] and a new addition over the tasks in the last chapter.

4.3.1 Reconstructing payoffs

To reconstruct the payoffs p^d, p^a, I tried approaching it like the KPG in Section 4.2. I implemented Cronert's method with the custom payoff functions f^d and f^a. I also added the magnitude constraint $P^d = \sum \hat{p}^d, P^a = \sum \hat{p}^a$, like in the weights subsection of Section 4.2. Given a set of observations $\mathcal{O}$ of a CNG problem with n items where for each observation $o \in \mathcal{O}$ two solutions $\hat{x}_d^o, \hat{\alpha}_d^o \in \{0,1\}^n$ and $\hat{x}_a^o, \hat{\alpha}_a^o \in \{0,1\}^n$, weights $w_d^o, w_a^o \in \mathbb{N}^n$, capacities $c_d^o, c_a^o \in \mathbb{N}$, magnitudes $P^d = \sum \hat{p}^d, P^a = \sum \hat{p}^a$ and parameters $\delta, \eta, \epsilon, \gamma \in [0,1]$ are known, the goal is to find the unknown payoffs $\hat{p}^d, \hat{p}^a \in \mathbb{N}^n$ by solving the master problem

$$\begin{aligned}\min_{p^d,p^a}\{\sum_{o\in\mathcal{O}} \theta_d^o + \theta_a^o : \theta_d^o \geq f^d(\bar{x}^o, \hat{\alpha}_d^o) - f^d(\hat{x}_d^o, \hat{\alpha}_d^o) \quad \forall \bar{x}^o \in \tilde{\mathcal{S}}_d^o \forall o \in \mathcal{O},\\ \theta_a^o \geq f^d(\bar{\alpha}^o, \hat{x}_d^o) - f^a(\hat{\alpha}_a^o, \hat{x}_a^o) \quad \forall \bar{\alpha}^o \in \tilde{\mathcal{S}}_a^o \forall o \in \mathcal{O},\\ \sum_{i=1}^n p_i^d = P^d, \sum_{i=1}^n p_i^a = P^a, \quad p^d, p^a \in \mathbb{N}^n\}. \quad (4.7)\end{aligned}$$

where the proposed payoff vectors p^d, p^a are tested by solving

$$\max_x\{f^d(x, \hat{\alpha}_d^o) : w_d^o x \leq c_d^o, x \in \{0,1\}^n\}$$

and

$$\max_\alpha\{f^a(\alpha, \hat{x}_a^o) : w_a^o \alpha \leq c_a^o, \alpha \in \{0,1\}^n\}$$

for all observations $o \in \mathcal{O}$. The two resulting strategies are checked against the sets $\tilde{\mathcal{S}}^d$ and $\tilde{\mathcal{S}}^a$ and added if they are not in them yet. If both the strategies are not new, then the algorithm is finished.

Implementation To test the method, I generated samples with fixed payoff vectors $\hat{p}^d, \hat{p}^a$ and correlated vectors w^d, w^a. These weights vectors varied from observation to observation. The payoffs $\hat{p}$ are generated as an integer uniform random vectors with entries in the range $[1, \ldots, 25]$. The weights w^o are generated as integer uniform random vectors conditional on $\hat{p}$ with entries in the range $[\hat{p}+1, \ldots, \hat{p}+25]$. The parameters were always constant on all observations, with the same relations between η, δ and ϵ: $\delta = 0.8\eta$ and $\epsilon = 1.25\eta$. The RAC is equal to $\sum_{i=1}^n |\hat{p}_i^d - p_i^d| + |\hat{p}_i^d - p_i^d| / \sum_{i=1}^n \hat{p}_i^d + \hat{p}_i^a$.

For each sample, I generated $10n$ observations, where n is the number of items. The observations were all solved using Zero Regrets from the perspective of both the defender and the attacker, resulting in two solution pairs $\hat{x}^d, \hat{\alpha}^d$ and $\hat{x}^a, \hat{\alpha}^a$, both Φ-NE (or very rarely PNE) solutions. These might be identical, but this is not always the case.

The implementations of this section can be found in `problems/inverse_cng.py` and `inverse_cng_test.py`.

Results When I tried to generate the initial tests for this section, starting with $n = 10, \eta = 0.75, \gamma = 0.1$, $c^d = 0.3 \sum w^d, c^a = 0.1 \sum w^a$, I ran into a big issue: the reconstructed payoffs p^d were never even close to $\hat{p}^d$ and usually almost uniform with one high outlier to satisfy the magnitude constraint. On the other hand, the reconstructed payoffs p^a were quite accurate (RAC of 0.06 at $10n$ observations), but decreasing the number of observations down to $5n$ already showed a big decrease in accuracy (RAC of 0.38). I tried varying the capacities by increasing c^d or c^a, varying η up and down and changing η. Nothing changed the result that p^a was quite accurate but p^d was worse than using P^d/n as payoff for each item. Increasing the number of observations to $25n$ did also not change much but at $50n$ the quality of p^d improved somewhat. However, at $100n$ this effect had disappeared again.

I have two ideas what could be causing this. First, it might be related to the fact that for the defender, the payoff of defending a node ($x_i = 1$) is either $\epsilon p_i^d = 1.25\eta p_i^d$ or ηp_i^d, while the payoff of not defending a node ($x_i = 0$) is either p_i^d or $\delta p_i^d = 0.8\eta p_i^d$. These two sets of outcomes are very similar. In the KPG, the payoff is either 0 or $p_j^i + \sum I_j^i$ for an item, a much clearer image. In constrast, the payoffs of attacking a node ($\alpha_i = 1$) are either p_i^a or $(1-\eta)p_i^a$ while the payoffs of not attacking a node ($\alpha_i = 0$) are either $-\gamma p_i^\alpha$ or 0. Thus, the difference between the two choices of the defender exists, but both are factors of p_i^d while for the attacker there is a small penalty for not attacking undefended nodes, but there is a clear reward for attacking. This difference in effect might explain the lack of results for the defender.

A second factor which might make CNG harder than KPG is the fact that (almost) all solutions are Φ-NE. This means that it is always possible to find better solutions in the adverse step, since there is always an alternative solution $\hat{x}$ or $\hat{\alpha}$ which is better than $\hat{x}$ or $\hat{\alpha}$. These different solutions have to be balanced against each other by the solver, since the difference between them and the target solution has to be minimised.

4.3.2 Reconstructing weights

To reconstruct the weights w^d, w^a, I tried approaching it like the KPG in Section 4.2. I implemented Cronert's method in the same way as Subsection 4.2.2, with the magnitude constraint $W^d = \sum \hat{w}^d, W^a = \sum \hat{w}^a$. Given a set of observations $\mathcal{O}$ of a CNG problem with n items where for each observation $o \in \mathcal{O}$ two NE solutions $\hat{x}_d^o, \hat{\alpha}_d^o \in \{0,1\}^n$ (defender perspective) and $\hat{x}_a^o, \hat{\alpha}_a^o \in \{0,1\}^n$ (attacker perspective) with their $\Phi^o \in \mathbb{Z}, \geq 0$, payoffs $p_d^o, p_a^o \in \mathbb{N}^n$, capacities $c_d^o, c_a^o \in \mathbb{N}$, weights magnitudes $W^d = \sum_{i=1}^n \hat{w}_i^d, W^a = \sum_{i=1}^n \hat{w}_i^a$ and parameters $\delta, \eta, \epsilon, \gamma \in [0,1]$ are known, the goal is to find the unknown weights $\hat{w}^d, \hat{w}^a \in \mathbb{N}^n$ by solving the master problem

$$\begin{aligned}\min_{p^d, p^a} \{1 : w^d \bar{x}^o > c_d^o \quad \forall \bar{x}^o \in \tilde{\mathcal{S}}_d^o \forall o \in \mathcal{O}, \quad w^a \bar{\alpha}^o > c_a^o \quad \forall \bar{\alpha}^o \in \tilde{\mathcal{S}}_a^o \forall o \in \mathcal{O}, \\ w^d \hat{x}_d^o \leq c_d^o,\ w^d \hat{x}_a^o \leq c_d^o,\ w^a \hat{\alpha}_d^o \leq c_a^o,\ w^a \hat{\alpha}_a^o \leq c_a^o \quad \forall o \in \mathcal{O}, \\ \sum_{i=1}^n w_i^d = W^d, \sum_{i=1}^n w_i^a = W^a, \quad w^d, w^a \in \mathbb{N}^n\}, \end{aligned} \tag{4.8}$$

which makes sure that both solution pairs remain feasible. The proposed weights vectors w^d, w^a are tested by solving

$$\max_x \{f_d^o(x, \hat{\alpha}_d^o) : w^d x \leq c_d^o, x \in \{0,1\}^n\}$$

and

$$\max_\alpha \{f_a^o(\alpha, \hat{x}_a^o) : w^a \alpha \leq c_a^o, \alpha \in \{0,1\}^n\}$$

for all observations $o \in \mathcal{O}$. In Subsection 4.2.2, it was possible to reject solutions with a higher payoff than the given solution since this is not possible in a PNE. However, here we are dealing with Φ-NE, meaning that each player is capable to achieve a Φ improvement by changing strategy. Thus, when we compare the payoff $f_d^o(x^o, \hat{\alpha}_d^o)$ are compared against the known payoff $f_d^o(\hat{x}_d^o, \hat{\alpha}_d^o)$, we can only reject x^o if $f_d^o(x^o, \hat{\alpha}_d^o) > f_d^o(\hat{x}_d^o, \hat{\alpha}_d^o) + \Phi_d^o$. Only then, x^o is added to $\tilde{\mathcal{S}}_d^o$. The same happens for the attacker strategies α^o. This limitation makes all impossible improvements infeasible in the updated master problem. If no new strategies were added to $\tilde{\mathcal{S}}^d$ or $\tilde{\mathcal{S}}^a$, then the algorithm finishes. Otherwise, the master problem is solved with the newly added constraints.

Implementation To test the method, I generated samples where the weights $\hat{w}^d, \hat{w}^a$ are fixed and the payoffs p_d^o, p_a^o are correlated and varied between observations. The weights $\hat{w}$ are generated as integer uniform random vectors with entries in the range $[1, \dots, 25]$. The payoffs p^o are generated as integer uniform random vectors conditional on $\hat{w}$ with entries in the range $[\hat{w} + 1, \dots, \hat{w} + 25]$. The parameters were always constant on all observations, with the same relations between η, δ and ϵ: $\delta = 0.8\eta$ and $\epsilon = 1.25\eta$. The RAC is equal to $\sum_{i=1}^{n} |\hat{w}_i^d - w_i^d| + |\hat{w}_i^d - w_i^d| / \sum_{i=1}^{n} \hat{w}_i^d + \hat{w}_i^a$.

For each sample, I generated $10n$ observations, where n is the number of items. These observations were solved using Zero Regrets from the perspective of both the defender and the attacker, resulting in two solution pairs $\hat{x}^d, \hat{\alpha}^d$ and $\hat{x}^a, \hat{\alpha}^a$, both Φ-NE (or very rarely PNE) solutions. These might be identical, but this is not always the case.

The implementations of this section can be found in `problems/inverse_cng.py` and `inverse_cng_test.py`.

Results Reconstructing the weights of a problem with only Φ-NE using the method from Section 4.2 was a challenge. Cronert's method relies on the fact that strategies with a higher payoff than the given solutions $\hat{x}, \hat{\alpha}$ should be very unlikely and aims to minimise the difference between the target solution and the best alternative. When applying Wang's method in Section 3.3, a second optimisation layer was necessary to optimise for Φ to make the problem feasible. For reconstructing w^d, w^a, there is not one value of Φ, but there are possibly two values of Φ per observation, meaning that to correct for the fact that solutions are allowed to be Φ better than the given solutions, we would either need to consider all values of Φ known or optimise over many different Φ^o values to approximate the original values. Thus, I decided to assume that the values Φ^o are given to see if this gave any initial results.

I started my tests with the same settings as the previous subsection: $n = 10, \eta = 0.75, \gamma = 0.1, c^d = 0.3 \sum w_d^o, c^a = 0.1 \sum w_a^o$. While I anticipated having to use the magnitude of the weights W^d and W^a and the Phi-values of all solutions, the results were very poor. I tried random capacity fractions for the different observations as well as increasing the capacity of the defender, but the results remained poor. Increasing the number of observations to $20n$ made the results a little better, but still, a vector with W^d/n would have a lower error than the results I was getting.

4.3.3 Reconstructing parameters

Since the CNG has payoff parameters, just like the location selection game of [13], I wanted to see if I could use Cronert's method for the task it was originally designed for: estimating parameters of payoff functions. Given a set of observations $\mathcal{O}$ of a CNG problem with n items where for each observation $o \in \mathcal{O}$ two solutions $\hat{x}_d^o, \hat{\alpha}_d^o \in \{0, 1\}^n$ and $\hat{x}_a^o, \hat{\alpha}_a^o \in \{0, 1\}^n$,

payoffs $p_d^o, p_a^o \in \mathbb{N}^n$, weights $w_d^o, w_a^o \in \mathbb{N}^n$ and capacities $c_d^o, c_a^o \in \mathbb{N}$ are known, the goal is to find the parameters $\hat{\delta}, \hat{\eta}, \hat{\epsilon}, \hat{\gamma} \in [0,1]$ by solving the master problem

$$\min_{\delta,\eta,\epsilon,\gamma} \{\sum_{o\in\mathcal{O}} \theta_d^o + \theta_a^o : \theta_d^o \geq f_d^o(\bar{x}^o, \hat{\alpha}_d^o) - f_d^o(\hat{x}_d^o, \hat{\alpha}_d^o) \quad \forall \bar{x}^o \in \tilde{\mathcal{S}}_d^o \forall o \in \mathcal{O},$$
$$\theta_a^o \geq f_a^o(\bar{\alpha}^o, \hat{x}_a^o) - f_a^o(\hat{\alpha}_a^o, \hat{x}_a^o) \quad \forall \bar{\alpha}^o \in \tilde{\mathcal{S}}_a^o \forall o \in \mathcal{O}, \quad \delta < \eta < \epsilon, \quad \delta, \eta, \epsilon, \gamma \in [0,1]\}, \quad (4.9)$$

where the parameters $\delta, \eta, \epsilon, \gamma$ are part of the payoff functions f_d^o and f_a^o (Equations 2.3 and 2.4). The effects of the parameters on the different possible interactions between the defender and the attacker can also be seen in Table 4.1. The parameters are tested by solving

$$\max_x \{f_d^o(x, \hat{\alpha}_d^o) : w_d^o x \leq c_d^o, x \in \{0,1\}^n\}$$

and

$$\max_\alpha \{f_a^o(\alpha, \hat{x}_a^o) : w_a^o \alpha \leq c_a^o, \alpha \in \{0,1\}^n\}$$

for all observations $o \in \mathcal{O}$, where the parameters are again part of the function f_d^o and f_a^o. The resulting solutions x^o, α^o are checked against the sets of known solutions $\tilde{\mathcal{S}}_d^o, \tilde{\mathcal{S}}_a^o$. New solutions are added to their respective sets. If no new solutions are found for any observation, the algorithm finishes. Otherwise, the master problem is solved again with the newly added constraints.

Implementation To test the method, I generated samples where the weights $\hat{w}_d^o, \hat{w}_a^o$ vary and the payoffs p_d^o, p_a^o are correlated to the weights. The weights w^o are generated as integer uniform random vectors with entries in the range $[1, \ldots, 25]$. The payoffs p^o are generated as integer uniform random vectors conditional on w^o with entries in the range $[\hat{w}^o + 1, \ldots, w^o + 25]$. The parameters were constant, with fixed relations between η, δ and ϵ: $\delta = 0.8\eta$ and $\epsilon = 1.25\eta$. I tested with $\eta = 0.75$.

For each sample, I generated $10n$ observations, where n is the number of items. The observations were all solved using Zero Regrets from the perspective of both the defender and the attacker, resulting in two solution pairs $\hat{x}^d, \hat{\alpha}^d$ and $\hat{x}^a, \hat{\alpha}^a$, both Φ-NE (or very rarely PNE) solutions. These might be identical, but this is not always the case.

The implementations of this section can be found in `problems/inverse_cng.py` and `inverse_cng_test.py`.

Results When testing this method, I started with $n = 10$ items and ran into the same problems mentioned in the last subsection. Different parameters for the payoffs can have a big impact on the optimal strategies for the players, resulting in strategies which are big improvements over the original strategies. In order to minimise the differences between the results of different strategies, the optimiser finds extreme values, like $\delta = 0.01, \eta = 0.01, \epsilon = 0.99$ and $\gamma = 0.01$, which achieve this goal. The resulting parameters depend on the capacities used. When capacities are fixed, for instance $c_d^o = 0.3 \sum w_d^o, c_a^o = 0.03 \sum w_a^o$, then δ and η are close to 0, but when the capacity of the attacker increases to $0.1 \sum w_a^o$, suddenly δ and η are close to 0.5. The true values are: $\delta = 0.8 * 0.75 = 0.6, \eta = 0.75, \epsilon = 0.9375$ and $\gamma = 0.1$. Randomised capacities with factions of $\sum w_d^o$ and $\sum w_a^o$ also gives similar results.

4.4 Chapter conclusion

In this chapter, I investigates reconstruction based inverse optimisation on sets of observations for KP, KPG and CNG problems. For these problems, I looked at reconstructing payoffs,

weights and, in the case of CNG, problem parameters. If we start with the KP, the reconstruction worked, but at a significant cost in performance. Reconstruction is much more resource intensive than adjustment and this is immediately clear from the number of items used in Section 4.1. Thus, the answer to **SQ1** is that reconstruction inverse optimisation is more limited than adjustment inverse optimisation. The complexity of this problem is considerably higher than of its adjustment counterpart.

Looking at the KPG and CNG, Cronert's method worked well on the KPG instances, but less so on the CNG. The structure of the CNG problem make the approach of Cronert's hard to execute. Thus, to answer **SQ2**, Cronert's method works well on the KPG instances, but not on CNG. For the payoffs, the defender payoffs are problematic. Reconstructing the weights suffers from more general problems, where almost the entire budget is dumped on a single low-value item, thus providing almost no useful information.

If we look at the sizes of the problems in this chapter, it is clear that the reconstruction problem takes its toll in computation time and complexity. A secondary problem for the generation of the results in this chapter was the fact that the Zero Regrets method is also quite complex and takes time on the larger instances. The problems in this chapter are much smaller than the instances in the adjustment chapter, so to answer **SQ3**, the number of parameters which can be handled depends on the type inverse optimisation but reconstruction has a lower limit than adjustment.

SQ4 asks if the methods are effective on practical instances. While I think this is the case in the previous chapter, for this chapter I have less comparison material on instances sizes. The instances in this section were much smaller than those in the previous chapter, but the reconstruction problem is also different. Cronert's method was developed for estimating parameters which are essential for the complex payoff function, with one parameter per player plus one global parameter [13]. They reported results for two players with 30 locations (similar to items) and running times in the order of hours. I was not able to dedicate the computer resources to perform experiments with individual runs which take that long, but compared to that, solving problems with 3 players and 25 items in seconds is quite okay.

Chapter 5

Conclusions and discussion

In this chapter, I will summarise the conclusions of the previous two chapters and provide a summary of my contributions to the literature. In addition, I will discuss the limitations of my research and highlight possible future steps for investigating these limitations.

5.1 Conclusions

In the first chapter, I formulated a main research question with four subquestions. In the previous two chapters, I already provided partial answers to the subquestions. I will collect the answers to the subquestions first and then provide the final answer to the main research question.

SQ1: Can inverse optimisation be applied to integer programs with a high number of parameters?
To determine if inverse optimisation worked on integer programs, I started with a simple IP, the knapsack problem. The main reason for choosing this specific problem is the fact that it has a related IPG, the knapsack packing game. I applied adjustment inverse optimisation to the KP, both on the weights and the payoffs and was able to solve instances with 10000 items. These instances are 2-3 orders of magnitude larger than the instances reported in [32]. I was not able to find literature dealing specifically with adjusting the constraints of KPs, only for general LPs. Thus, I do not have direct comparison material for this. 10000 items is large enough to model a range of real world problems, so adjustment inverse optimisation can be applied to knapsack problems with a high number of items.

If we look at reconstruction inverse optimisation applied to the KP, this is a new application of existing techniques when reconstructing payoffs and an adaptation of existing techniques when reconstructing weights. The drop in number of items is significant, with the largest instances having just 60 items. While this means we are learning 60 parameters, the difference with the adjustment setting is considerable. For the payoffs, the trend of the running time suggests that solving larger problems will come at a significant cost, but all mean running times were less than 60 seconds, so there is some room for larger problems. When considering reconstructing weights, the instances were similar but the running time trends were not. When reconstructing the weights, there was a turning point after which the running times started to decrease. Where this happened depended on the number of items and observations. There might a certain upper bound of information after which weight reconstruction becomes significantly easier. However, time limitations prevented me from investigating this. Reconstruction inverse optimisation is more limited than adjustment and

can be applied to knapsack problems with significantly less parameters.

SQ2: Which inverse optimisation methods can be applied to integer programming games?
In the literature review in Chapter 2, I highlight two inverse optimisation method. Wang's method was developed for adjustment inverse optimisation of IPs [36] and Cronert's method was developed for reconstruction inverse optimisation of IPGs [13]. In Chapter 3, I adjusted Wang's method to perform adjustment inverse optimisation on IPs and IPGs, both of payoffs and weights. This worked well for KPG when applied to both payoffs and weights. CNG was the most challenging problem for adjustment, with large changes required to achieve the goal of finding PNE solutions or improving Φ-NE.

In Chapter 4, I adjusted Cronert's method to perform reconstruction inverse optimisation on IPs and IPGs. I had to decrease the size of problems considerably since generating the observations and solutions for this chapter took a long time. For KP, the method worked for both weights and payoffs. For KPG, the method also worked for both, but considerably better for the payoffs. Here, there was also a big impact of having instances with varying capacities to encourage different strategies. The CNG was hard to inverse. I tried to inverse the payoffs, weight and parameters but I was not able to gain any useful results using Cronert's method. The best results was that the payoffs of the attacker were partially reconstructed, but then the payoffs of the defender were still useless.

SQ3: How can inverse optimisation be applied to specific integer programming games with a high number of parameters?
Adjustment inverse optimisation using Wang's method can be applied to both KPG and CNG, suggesting that it also should work for other IPGs. It can deal with IPGs where there is no clear link between the reward and choosing a certain item and it can also generate PNEa there, although these are most likely not very stable. Adjustment inverse optimisation of payoffs or weights can be applied to KPG instances with 4 or more players and 1000 items or more and still be solved in minutes. Adjustment of CNG instances was possible up to 100 nodes for payoffs and 200 nodes for weights when trying to turn heuristic solutions into PNE solutions. Φ-NE solutions were more time consuming to generate and also to solve in the case of weights. For payoffs, the difference is not immediately clear.

Reconstruction inverse optimisation using Cronert's method is demanding to test since it requires many observations for each result and these all need to be solved to a PNE or Φ-NE result. This limited the problem sizes which I was able to test. For the KPG, I was able to reconstruct the weights of problems with three players and 25 items in less than 10 seconds and the payoffs in less than 60 seconds. For the CNG, my attempts at adapting Cronert's method were not successful. I was not able to reconstruct the payoffs, weights or parameters of the CNG in a way that the results were useful for both players. The only thing I managed to accomplish was a partial reconstruction of the payoffs of the attacker.

SQ4: Are these methods effective on practical instances of the specific integer programming games?
The adjustment method work on IPGs of considerable sizes and the running times of these method and especially for the KPG, both the payoff and weights adjustment could be scaled to instances with four player and 1000 items. The CNG results were a little more limited, but still managed 100 items for the payoffs and 200 items for the weights. All these results take seconds or minutes to complete.

The reconstruction method works on the KPG, but is much more limited than the adjustment method. I was not able to test larger instances than 25 items due to the time required to generate the underlying observations. However, the increases in the running time of the weights reconstruction do not suggest that the method would scale poorly to larger instances. For the CNG, reconstruction was not possible on any instances. This is unfortunate,

but also provides questions for new research.

RQ: How can inverse optimisation be applied to integer programming games with a high number of parameters?
Inverse optimisation can be applied to both the KPG and CNG to make a given solution optimal by adjusting their weights or payoffs. This is possible for instances of considerable size and the results leave room for investigation of larger instances.

Inverse optimisation can also be applied to learn the payoffs or weights of players in a KPG with repeated observations. Here, the accuracy increases with the number of available observations. The results so far leave room for investigation of larger instances, since most individual results took seconds or minutes.

5.2 Scientific contributions

In this thesis, I investigated adjustment and reconstruction inverse optimisation methods for integer programs and integer programming games. I applied Wang's adjustment method to KP instances much larger than those reported in the literature and adjusted it to allow for the adjustment of weights. I adapted Wang's method to make it compatible with IPGs, specifically KPG and CNG, and added the adjustment of weights for those too. Adjustment inverse optimisation of IPGs and especially adjustment of constraints of IPGs is a new area for inverse optimisation research.

I applied Cronert's reconstruction method to KP instances and adjusted the method to allow for constrain reconstruction. This is a new application for IPs. I also adapted Cronert's method to two new IPG problems, the KPG and CNG. For the KPG, I was able to scale the method to many observations and parameters, both in the payoffs and constraints. The use of this method for constraint inverse optimisation is new. For the CNG, I tried to reconstruct the payoffs, weights and parameters. Unfortunately, the nature of this problem was not suited for the method used by Cronert's method and the results were almost all useless.

5.3 Discussion

The first obvious limitation of my research is that **SQ1** mentions integer programs in general, but I only investigated one specific type. This is a limitation, but IPs were not the focus of my research. I used the KP to try out all the methods and see what worked. While there is no reason to assume these results might not extend to other types of problems, [36] is formulated generally, and worked for all three problems, even the CNG which has some negative values and interactions.

The second limitation is very similar. I only investigated two types of IPGs, the KPG and the CNG, and especially for reconstruction the results were not comparable between the two. This is an indication that more research is needed since in [13], the method is described as a general Inverse IPG method. To also work for problems like the CNG, the adverse step probably needs to be made more sophisticated. Currently, the Φ-NE nature of the CNG solutions was too complex to solve.

A third limitation of my results ares the sizes of the problems due to their running times. The adjustment results are all taken over 30 samples to get an idea of the variance in those results and combine that with the combinations of parameters means that the individual problems can not be very time consuming. The reconstruction results required large sets

of observations, which all had to be solved by Zero Regrets or like an IP. The large number of problems to be solved forced me to keep the individual problems to running times in the order of seconds or minutes. Thus, a big open question for the future is what the scaling of these problems is like. The tests of the reconstruction method required a lot of preparation work to generate all the observations, limiting the size of the problem. One thing I was able to observe is that increasing the number of items requires more than a linear increase in the number of observations to achieve similar accuracy results. How does this work for instances with 50 or 100 items? The running time is also affected by the number of observations, so what running times can we expect for larger instances if we want results with close to perfect reconstruction results? There were also cases where the relation between number of observations and running time broke down. When reconstructing the weights of the KP, the running time starts to decrease after a time, so the payoff reconstruction and weight reconstruction problems might have different characteristics. All of this needs further research with larger experiments to get a better idea of the scaling characteristics of these problems.

A final limitation of my research are the (lack of) results when applying reconstruction inverse optimisation to the CNG. I tried to apply Cronert's method to them, expecting similar results to those of the adjustment inverse optimisation, but was unable to do so. While I have ideas as to why Cronert's method did not work, analysing the exact nature of CNG and its characteristics to find out what was going wrong was beyond the scope of my research. A more thorough analysis of this should reveal the reasons why the method did not work and also give suggestions for possible changes or improvements to make Cronert's method more robust to problems like CNG with (only) Φ-NE solutions and zero-sum elements.

Bibliography

[1] Ravindra K. Ahuja and James B. Orlin. Inverse Optimization. *Operations Research*, 49(5):771–783, 2001. Publisher: INFORMS. 12, 14

[2] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997. 10

[3] Danny Blom, Christopher Hojny, and Bart Smeulders. Cutting Plane Approaches for the Robust Kidney Exchange Problem. *Computers & Operations Research*, 162:106470, February 2024. 1, 6

[4] Danny Blom, Bart Smeulders, and Frits Spieksma. Rejection-proof mechanisms for multi-agent kidney exchange. *Games and Economic Behavior*, 143:25–50, January 2024. 1, 6

[5] Margarida Carvalho. *Computation of equilibria on integer programming games*. PhD thesis, Universidade do Porto, Porto, 2016. 10

[6] Margarida Carvalho, Gabriele Dragotto, Andrea Lodi, and Sriram Sankaranarayanan. The Cut-and-Play Algorithm: Computing Nash Equilibria via Outer Approximations, June 2023. arXiv:2111.05726 [cs, math]. 8

[7] Margarida Carvalho, Gabriele Dragotto, Andrea Lodi, and Sriram Sankaranarayanan. Integer Programming Games: A Gentle Computational Overview, June 2023. arXiv:2306.02817 [cs, math]. vii, 6, 7, 8

[8] Margarida Carvalho, Andrea Lodi, and João Pedro Pedroso. Existence of Nash Equilibria on Integer Programming Games. In A. Ismael F. Vaz, João Paulo Almeida, José Fernando Oliveira, and Alberto Adrego Pinto, editors, *Operational Research*, volume 223, pages 11–23. Springer International Publishing, Cham, 2018. Series Title: Springer Proceedings in Mathematics & Statistics. 7

[9] Margarida Carvalho, Andrea Lodi, and João.P. Pedroso. Computing equilibria for integer programming games. *European Journal of Operational Research*, 303(3):1057–1070, December 2022. 1, 7, 8, 10

[10] Timothy C. Y. Chan and Neal Kaw. Inverse optimization for the recovery of constraint parameters, July 2019. arXiv:1811.00726 [math] version: 2. 19

[11] Timothy C. Y. Chan and Neal Kaw. Inverse optimization for the recovery of constraint parameters. *European Journal of Operational Research*, 282(2):415–427, April 2020. 12, 14

[12] Timothy C. Y. Chan, Rafid Mahmood, and Ian Yihang Zhu. Inverse Optimization: Theory and Applications. *Operations Research*, page opre.2022.0382, December 2023. 12

[13] Tobias Crönert, Layla Martin, Stefan Minner, and Christopher S. Tang. Inverse optimization of integer programming games for parameter estimation arising from competitive retail location selection. *European Journal of Operational Research*, 312(3):938–953, February 2024. iii, vii, 1, 2, 7, 12, 14, 15, 17, 44, 46, 48, 50, 51

[14] Tobias Crönert and Stefan Minner. Equilibrium Identification and Selection in Finite Games. *Operations Research*, page opre.2022.2413, December 2022. 8

[15] George Dantzig, Alexander Orden, and Philip Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, June 1955. 4

[16] George B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research*, 5(2):266–277, 1957. Publisher: INFORMS. 18

[17] Gabriele Dragotto, Amine Boukhtouta, Andrea Lodi, and Mehdi Taobane. The Critical Node Game, March 2023. arXiv:2303.05961 [cs, math]. 1, 6, 7, 11, 26, 29, 32

[18] Gabriele Dragotto and Rosario Scatamacchia. The Zero Regrets Algorithm: Optimizing over Pure Nash Equilibria via Integer Programming. *INFORMS Journal on Computing*, 35(5):1143–1160, September 2023. 6, 9, 10, 32

[19] Python Software Foundation. The Python Language Reference. Version 3.10.5. 5

[20] Kimia Ghobadi and Houra Mahmoudzadeh. Inferring linear feasible regions using inverse optimization. *European Journal of Operational Research*, 290(3):829–843, May 2021. 12, 14, 19

[21] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. 4, 5

[22] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. Publisher: Springer Science and Business Media LLC. 5

[23] Clemens Heuberger. Inverse Combinatorial Optimization: A Survey on Problems, Methods, and Results. *Journal of Combinatorial Optimization*, 8(3):329–361, September 2004. 1, 12, 14

[24] Siming Huang. Inverse Problems of Some NP-Complete Problems. In *Algorithmic Applications in Management*, volume 3521, pages 422–426. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. Series Title: Lecture Notes in Computer Science. 1, 14

[25] Jonathan A. Kelner and Daniel A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing*, pages 51–60, Seattle WA USA, May 2006. ACM. 4

[26] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 3, 4

[27] Matthias Köppe, Christopher Thomas Ryan, and Maurice Queyranne. Rational Generating Functions and Integer Programming Games. *Operations Research*, 59(6):1445–1460, December 2011. 1, 6, 7

[28] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. Wiley-Interscience series in discrete mathematics and optimization. J. Wiley & Sons, Chichester ; New York, 1990. 4, 5

[29] Mahsa Moghaddass and Daria Terekhov. Inverse integer optimization with multiple observations. *Optimization Letters*, 15(4):1061–1079, June 2021. 12, 13

[30] J.F. Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951. 7

[31] David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284, September 2005. 5

[32] Julien Roland, José Rui Figueira, and Yves De Smet. The inverse { 0 , 1 } -knapsack problem: Theory, algorithms and computational experiments. *Discrete Optimization*, 10(2):181–192, May 2013. 1, 12, 17, 18, 19, 34, 49

[33] Simone Sagratella. Computing All Solutions of Nash Equilibrium Problems with Discrete Strategy Sets. *SIAM Journal on Optimization*, 26(4):2190–2218, January 2016. 9

[34] Stefan Schwarze and Oliver Stein. A branch-and-prune algorithm for discrete Nash equilibrium problems. *Computational Optimization and Applications*, 86(2):491–519, November 2023. 9

[35] R Core Team. R: A language and environment for statistical computing, 2022. 5

[36] Lizhi Wang. Cutting plane algorithms for the inverse mixed integer linear programming problem. *Operations Research Letters*, 37(2):114–116, March 2009. vii, 1, 2, 12, 14, 15, 17, 18, 50, 51

[37] Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, Max Kuhn, Thomas Lin Pedersen, Evan Miller, Stephan Milton Bache, Kirill Müller, Jeroen Ooms, David Robinson, Dana Paige Seidel, Vitalie Spinu, Kohske Takahashi, Davis Vaughan, Claus Wilke, Kara Woo, and Hiroaki Yutani. Welcome to the Tidyverse. *Journal of Open Source Software*, 4(43):1686, November 2019. 5

Appendix A

Adjustment single-observation inverse optimisation

A.1 Knapsack Packing

Table A.1: Running times in seconds of the inverse payoffs algorithm on instances with 30 samples per combination.

range	$r = 500$				$r = 1000$				$r = 5000$			
items	mean	SD	min	max	mean	SD	min	max	mean	SD	min	max
100	0.158	0.036	0.123	0.234	0.163	0.040	0.123	0.294	0.192	0.105	0.124	0.683
500	0.876	0.340	0.603	1.824	0.699	0.164	0.599	1.483	0.863	0.407	0.592	2.122
1000	1.279	0.076	1.187	1.463	1.400	0.700	1.193	5.140	1.214	0.029	1.182	1.292
5000	6.054	0.148	5.875	6.327	5.900	0.480	4.665	6.220	4.900	0.188	4.529	5.269
10000	9.736	0.301	9.273	10.510	9.645	0.141	9.394	9.851	9.663	0.144	9.393	9.995

Table A.2: Running times in seconds of the inverse weights algorithm on instances with 30 samples per combination.

range	$r = 500$				$r = 1000$				$r = 5000$			
items	mean	SD	min	max	mean	SD	min	max	mean	SD	min	max
100	0.247	0.238	0.085	1.241	0.281	0.333	0.087	1.478	0.307	0.356	0.087	1.560
500	0.935	0.755	0.457	4.267	1.764	2.614	0.422	13.008	0.803	0.611	0.421	2.998
1000	1.053	0.394	0.838	2.845	1.097	0.426	0.850	3.061	0.932	0.177	0.837	1.661
5000	4.271	0.095	4.156	4.508	4.173	0.337	3.295	4.431	3.455	0.117	3.191	3.663
10000	6.908	0.380	6.546	8.479	6.824	0.118	6.649	7.199	6.825	0.085	6.657	6.962

A.2 Knapsack Packing Game

Table A.3: Running time statistics in seconds of the payoff adjustment algorithm on instances with no negative interactions and 30 samples per combination.

	range	$r = 500$				$r = 1000$			
players	items	mean	SD	min	max	mean	SD	min	max
2	100	0.608	0.050	0.574	0.871	0.591	0.019	0.564	0.664
	500	2.716	0.052	2.657	2.851	2.742	0.054	2.663	2.858
	1000	5.609	0.096	5.417	5.843	5.520	0.098	5.288	5.645
3	100	1.130	0.034	1.094	1.268	1.123	0.023	1.089	1.173
	500	5.414	0.118	5.248	5.670	5.377	0.117	5.255	5.757
	1000	11.226	0.449	10.600	12.400	11.109	0.401	10.707	12.273
4	100	1.897	0.034	1.826	1.976	1.910	0.051	1.826	2.009
	500	9.109	0.169	8.857	9.545	9.081	0.190	8.835	9.798
	1000	18.317	0.643	17.503	20.200	19.345	7.568	17.054	60.000

Table A.4: Running time statistics in seconds of the weights adjustment algorithm on KPG instances with no negative interactions and 30 samples per combination.

	range	$r = 500$				$r = 1000$			
players	items	mean	SD	min	max	mean	SD	min	max
2	100	0.316	0.033	0.284	0.463	0.301	0.011	0.283	0.321
	500	1.271	0.029	1.230	1.339	1.265	0.021	1.232	1.329
	1000	2.464	0.038	2.402	2.587	2.481	0.094	2.361	2.837
3	100	0.500	0.029	0.462	0.592	0.494	0.026	0.456	0.556
	500	1.918	0.055	1.827	2.076	1.900	0.041	1.845	1.988
	1000	3.975	0.190	3.720	4.362	3.885	0.173	3.695	4.417
4	100	0.689	0.030	0.649	0.750	0.702	0.055	0.612	0.855
	500	2.726	0.095	2.586	2.990	2.728	0.110	2.530	3.120
	1000	5.470	0.276	5.180	6.548	5.440	0.270	5.084	6.392

A.3 Critical Node Game

Table A.5: Mean running times, mean RAC, number of PNE solutions and infeasible instances of payoff adjustment on heuristic instances with 20 samples per combination.

		$\gamma = 0$				$\gamma = 0.1$			
items	η	mean runtime	mean RAC	PNEa	inf	mean runtime	mean RAC	PNEa	inf
50	0.60	0.266	0.022	20	0	0.288	0.023	20	0
	0.75	0.377	0.048	20	0	0.721	0.053	20	0
70	0.60	0.864	0.023	20	0	0.770	0.025	20	0
	0.75	1.480	0.046	20	0	2.717	0.054	20	0
100	0.60	2.471	0.021	20	0	12.551	0.025	20	0
	0.75	33.006	0.046	18	2	70.515	0.053	10	10

Table A.6: Mean running times, mean RAC, mean $\Delta\Phi_{UB}$ and number of PNE solutions of payoff adjustment on Φ-NE instances with 10 samples per combination.

		$\gamma = 0$				$\gamma = 0.1$			
items	η	mean runtime	mean RAC	mean $\Delta\Phi_{UB}$	PNEa	mean runtime	mean RAC	mean $\Delta\Phi_{UB}$	PNEa
20	0.60	0.060	0.015	0.667	0	0.060	0.013	0.697	4
	0.75	0.060	0.017	0.550	0	0.060	0.016	0.555	0
30	0.60	0.085	0.023	0.825	0	0.086	0.019	0.781	0
	0.75	0.088	0.031	0.816	0	0.089	0.027	0.796	0
40	0.60	0.118	0.023	0.844	0	0.113	0.023	0.851	0
	0.75	0.117	0.026	0.843	0	0.117	0.028	0.850	0
50	0.60	0.162	0.023	0.872	0	0.151	0.023	0.878	0
	0.75	0.167	0.048	0.873	0	0.155	0.043	0.866	0

Table A.7: Mean running times, mean RAC, number of PNE solutions and infeasible instances of weight adjustment on heuristic instances with 20 samples per combination.

		$\gamma = 0$				$\gamma = 0.1$			
items	η	mean runtime	mean RAC	PNEa	inf	mean runtime	mean RAC	PNEa	inf
50	0.60	0.230	0.005	20	0	0.383	0.016	20	0
	0.75	0.223	0.005	20	0	0.395	0.018	20	0
100	0.60	0.503	0.008	20	0	0.720	0.015	20	0
	0.75	0.397	0.007	20	0	0.922	0.022	20	0
150	0.60	2.223	0.006	20	0	2.716	0.011	20	0
	0.75	2.834	0.011	20	0	5.202	0.017	20	0
200	0.60	37.982	0.003	18	2	9.522	0.003	20	0
	0.75	11.797	0.003	20	0	25.753	0.004	19	1

Table A.8: Mean running times, mean RAC, mean $\Delta\Phi_{UB}$ and number of PNE solutions of weights adjustment on Φ-NE instances with 10 samples per combination.

		$\gamma = 0$					$\gamma = 0.1$			
items	η	mean runtime	mean RAC	mean $\Delta\Phi_{UB}$	PNEa	inf	mean runtime	mean RAC	mean $\Delta\Phi_{UB}$	PNEa
20	0.60	0.062	0.127	1.000	20	0	0.058	0.077	1.000	20
	0.75	0.124	0.063	0.517	8	0	0.143	0.082	0.360	4
30	0.60	0.503	0.146	0.879	16	0	0.104	0.157	1.000	20
	0.75	0.488	0.122	0.460	2	0	0.362	0.130	0.535	2
40	0.60	0.207	0.217	0.929	18	0	0.578	0.193	0.950	18
	0.75	0.867	0.138	0.688	6	0	0.475	0.211	0.771	6
50	0.60	1.196	0.141	1.000	20	0	1.085	0.125	0.844	16
	0.75	2.164	0.136	0.512	4	4	2.332	0.162	0.655	8

Appendix B

Reconstruction multi-observation inverse optimisation

B.1 Knapsack Packing

Table B.1: Mean running times and mean RAC of payoffs reconstruction on KP problems with $c^o = 0.5 \sum w^o$ and 5 repeats per combination.

items	$n = 20$		$n = 40$		$n = 60$	
obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
$\frac{1}{2}n$	0.390	0.169	3.145	0.116	15.092	0.123
n	0.640	0.111	4.176	0.071	19.775	0.063
$2n$	1.020	0.089	6.083	0.050	25.021	0.035
$4n$	1.663	0.040	9.205	0.030	34.442	0.023
$6n$	2.170	0.032	11.518	0.019	43.893	0.020
$8n$	2.815	0.028	14.732	0.016	51.694	0.014

Table B.2: Mean running times and mean RAC of payoffs reconstruction on KP problems with random capacities c^o and 5 repeats per combination.

items	$n = 20$		$n = 40$		$n = 60$	
obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
$\frac{1}{2}n$	0.330	0.271	2.532	0.105	9.023	0.067
n	0.601	0.178	3.503	0.069	10.529	0.040
$2n$	0.775	0.094	4.088	0.040	12.045	0.022
$4n$	1.421	0.056	6.218	0.022	17.373	0.011
$6n$	1.950	0.043	8.252	0.010	22.339	0.005
$8n$	2.298	0.025	9.728	0.009	28.094	0.004

Table B.3: Mean running times and mean RAC of weights reconstruction on KP problems with $c = 0.5 * \sum w$ and 5 repeats per combination.

items	$n = 20$		$n = 40$		$n = 60$	
obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
n	3.944	0.020	20.180	0.902	30.355	1.172
$2n$	1.043	0.005	39.547	0.379	61.126	0.840
$3n$	0.396	0.000	31.412	0.106	93.323	0.449
$4n$	0.294	0.000	13.764	0.000	87.772	0.101

Table B.4: Mean running times and mean RAC of weights reconstruction on KP problems with random capacities c^o and 5 repeats per combination.

items	$n = 20$		$n = 40$		$n = 60$	
obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
n	3.502	0.024	20.249	0.728	30.523	1.143
$2n$	1.034	0.007	33.304	0.097	62.011	0.685
$3n$	0.260	0.001	3.330	0.000	95.602	0.028
$4n$	0.261	0.000	1.281	0.000	41.879	0.000

B.2 Knapsack Packing Game

Table B.5: Mean running times and mean RAC of payoff reconstruction on KPG problems with only PNE solutions, $c_i^o = 0.5 \sum w_i^o$ and 5 repeats per combination.

items		$m = 10$		$m = 20$		$m = 25$	
players	obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
2	m	0.309	0.689	1.348	0.529	2.004	0.440
	$2m$	0.605	0.517	2.106	0.351	3.100	0.304
	$4m$	1.178	0.322	2.857	0.224	4.517	0.177
	$6m$	1.545	0.270	4.663	0.186	5.808	0.134
	$8m$	2.203	0.173	5.464	0.122	7.636	0.096
3	m	0.450	0.856	1.666	0.724	2.503	0.737
	$2m$	1.034	0.705	3.372	0.583	5.367	0.553
	$4m$	1.716	0.516	6.525	0.357	8.736	0.351
	$6m$	2.938	0.377	7.817	0.287	10.834	0.273
	$8m$	4.166	0.258	10.761	0.207	14.140	0.217

Table B.6: Mean running times and mean RAC of payoff reconstruction on KPG problems with Φ-NE and PNE solutions, $c_i^o = 0.5 \sum w_i^o$ and 5 repeats per combination.

	items	$m = 10$		$m = 20$		$m = 25$	
players	obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
2	m	0.310	0.796	1.298	0.532	2.013	0.497
	$2m$	0.610	0.531	2.160	0.364	2.955	0.332
	$4m$	1.020	0.360	3.190	0.236	4.383	0.203
	$6m$	1.651	0.276	4.484	0.159	5.559	0.144
	$8m$	2.088	0.213	5.546	0.129	7.249	0.117
3	m	0.431	0.805	2.007	0.719	2.537	0.682
	$2m$	0.907	0.634	3.504	0.503	5.471	0.501
	$4m$	1.979	0.503	5.789	0.347	7.861	0.355
	$6m$	2.597	0.353	8.840	0.240	11.605	0.271
	$8m$	3.863	0.314	9.699	0.191	14.930	0.192

Table B.7: Mean running times and mean RAC of payoff reconstruction on KPG problems with Φ-NE and PNE solutions, random capacities and 5 repeats per combination.

	items	$m = 10$		$m = 20$		$m = 25$	
players	obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
2	m	0.309	0.686	1.284	0.486	5.164	0.412
	$2m$	0.553	0.558	2.021	0.351	7.689	0.273
	$4m$	0.935	0.373	3.049	0.204	11.919	0.183
	$6m$	1.416	0.239	4.082	0.142	15.956	0.124
	$8m$	1.848	0.167	5.452	0.106	21.117	0.094
3	m	0.419	0.817	1.844	0.631	7.798	0.572
	$2m$	0.893	0.627	3.501	0.470	12.358	0.382
	$4m$	2.141	0.469	6.089	0.319	22.916	0.241
	$6m$	3.132	0.352	7.653	0.217	25.780	0.185
	$8m$	3.616	0.271	9.370	0.157	40.494	0.153

Table B.8: Mean running times and mean RAC of weight reconstruction on KPG problems with PNE solutions, fixed capacities and 5 repeats per combination.

	items	$m = 10$		$m = 20$		$m = 25$	
players	obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
2	m	0.142	1.350	0.324	1.446	0.415	1.566
	$2m$	0.288	1.114	0.634	1.297	0.826	1.451
	$4m$	0.555	0.689	1.258	0.854	1.642	0.930
	$6m$	0.831	0.535	1.899	0.591	2.471	0.780
	$8m$	1.115	0.447	2.532	0.279	3.315	0.598
3	m	0.246	1.390	0.579	1.565	0.763	1.695
	$2m$	0.491	1.266	1.165	1.401	1.561	1.618
	$4m$	0.984	0.999	2.307	1.212	3.071	1.367
	$6m$	1.468	0.969	3.483	1.124	4.609	1.320
	$8m$	1.950	0.927	4.617	0.964	6.108	1.126

Table B.9: Mean running times and mean RAC of weight reconstruction on KPG problems with PNE solutions, random capacities and 5 repeats per combination.

		$m = 10$		$m = 20$		$m = 25$	
players	obs	mean runtime	mean RAC	mean runtime	mean RAC	mean runtime	mean RAC
2	m	0.140	1.227	0.315	1.275	0.412	1.278
	$2m$	0.282	0.727	0.631	0.895	0.821	0.975
	$4m$	0.571	0.300	1.250	0.419	1.650	0.670
	$6m$	0.841	0.230	1.882	0.255	2.460	0.373
	$8m$	1.116	0.107	2.510	0.157	3.290	0.172
3	m	0.245	1.209	0.622	1.457	0.829	1.566
	$2m$	0.490	0.735	1.186	1.143	1.700	1.316
	$4m$	0.985	0.392	2.313	0.970	3.283	1.209
	$6m$	1.475	0.233	3.453	0.802	4.763	1.055
	$8m$	1.964	0.145	4.617	0.757	6.433	0.999