Program Equilibrium

Moshe Tennenholtz [*]
Faculty of Industrial Engineering and Management
Technion – Israel Institute of Technology
Haifa 32000, Israel

[*]This work has been carried out while the author was with the CS department at Stanford University.

Abstract

In a computerized setting, players' strategies can be implemented by computer programs, to be executed on a shared computational devise. This situation becomes typical to new Internet economies, where agent technologies play a major role. This allows the definition of a *program equilibrium*. Following the fundamental ideas introduced by von Neumann in the 40s (in parallel to his seminal contribution to game theory), a computer program can be used both as a set of instructions, as well as a file that can be read and compared with other files. We show that this idea implies that in a program equilibrium of the one-shot prisoners dilemma mutual cooperation is obtained. More generally, we show that the set of program equilibrium payoffs of a game coincides with the set of feasible and individually rational payoffs of it.
Journal of Economic Literature Numbers: C72 C70 C60

1 The basic setting and idea

The Internet creates a new economy, where existing theories and techniques are challenged. Many of the new theories have to do with extensions of existing theories due to possibilities emerging in the Internet. Examples include work on issues such as selfish routing [11] or incorporating distributed systems features into game-theoretic models [7]. In such contexts, when considering e.g. Internet auctions, the extended theories have to deal with issues such as the structure of the communication network and its effects. However, the fact we talk about computer programs rather than people plays a somewhat lesser role; the new medium suggests new problems, but the fact protocols are to run by computer programs do not play a significant role. Nevertheless, the Internet also allows sophisticated use of computer programs that should be studied very carefully. In particular, the new medium allows the use of software agents, programs (also called proxies) that are designed to behave on behalf of a user. Perhaps the most famous, although rather simple, use of such agent is in e-bay auctions (www.ebay.com), where agents are simple proxies that can be instructed how to bid on behalf of users. Such simple ideas may have huge ramifications. When several programs (agents [9], proxies [1]) are to run on a computer in service of different users, basic issues in the foundations of computer science should be taken into account as part of the economic theory.[1] One of them, exposed in this paper, is a program's (agent's) ability to refer to computations done in parallel by other programs. As it turns out, this leads to surprising phenomena, based on the very basic principles of computing introduced by Von Neumann. This paper exposes these phenomena and how they effect basic game-theoretic reasoning.

In [14], von Neumann introduced one of the most important concepts in computer science. In his work, termed "the first draft", von Neumann explains the dual role of computer programs:

1. A computer program can serve as a set of instructions.

[1]Indeed, the above references refer already to non-trivial basic economic activities, such as selling, buying, and auctioning.

2. A computer program can serve as a file, to be read and compared to, by itself or by other programs.

Today, this dual role is obvious to every programmer. One for example can easily write a computer program that reads itself (as a file), and prints its content. Similarly, two different computer programs can be instructed to read one another, and perform some action if the other program's content is of particular kind.

In this paper we show that the above simple (but fundamental) idea can lead to most interesting results in the context of game theory.

A computer can communicate with a set of players (or programmers, we will use these terms interchangeably) in the following way. Each programmer provides his/her own program in a given agreed-upon programming language. The computer will then perform the corresponding tuple of computer programs. A computer program can be designed in order to instruct the computer on some form of behavior. In one setting for example, a programmer needs to decide on whether he/she wishes to devote some private resources (e.g. disk space) for public use. A programmer, through his/her computer program, will need to instruct the computer (which deals with all related computer programs simultaneously) on a selected course of action. In this setting, if there are two programmers, we get an instance of the famous prisoners dilemma (PD): if both parties agree to allow the public use of their resources then this will be more beneficial for both of them than the case where they both disallow such use; however, the best for a programmer is that he/she will not allow the use of his/her resources while the other will allow such use. The situation where programmer A allows the public use of his/her resources, while the other program disallows such use, is the worst possible for A. Following that, we refer to the above actions as cooperate (allow) or defect (do not allow).

Naively, given the above setting, one may consider only programs that directly specify the selected action in the corresponding (PD) game: DO(coop) and DO(defect). However, computer programs may be more sophisticated than that. As long as each program implies a well defined decision about the course of action to be taken by the computer (i.e. allow/coop or do not allow/defect in the above mentioned setting), it can provide the computer with an elaborated and sophisticated program. In particular, each programmer can provide the computer with the following type of program:

- "if my-program=the-other-player-program then DO(coop) else DO(defect)".

Although one may need to better define the exact syntax and semantics of the above program, its meaning can be easily understood by the reader. The program of a particular programmer instructs the computer to compare its content (as a file) to the content of the other programmer's program. If this (syntactic) comparison tells that the programs are the same (which is the case when both parties use the above program) then it calls for cooperation; otherwise, it calls for defection. The program uses the von-Neumann idea: when comparing my-program to the-other-player-program the programmer (player) asks for a syntactic comparison of the programs' texts (here the programs are treated as files), while based on this comparison a decision on the course of action is taken; the comparison itself is an instruction.

As the reader can verify, the above “tricky” program determines a *program equilibrium* for the (one-shot) prisoners dilemma. It is irrational for a player (programmer) to deviate from it, assuming that the other sticks to it. Moreover, as a result, mutual cooperation is obtained!

A more formal exposition of the above appears in the following sections. Moreover, we will show that the above result can be further extended. More specifically, we show that the set of program equilibrium payoffs of a game coincides with the set of feasible and individually rational payoffs of it.

2 Games and Programs

A game in strategic form is a tuple $\langle N, \{S_i\}_{i=1}^n, \{U_i\}_{i=1}^n \rangle$, where $N = \{1, 2, \ldots, n\}$ is a set of n players, S_i is a finite set of possible strategies for player i, and $U_i : \Pi_{i=1}^n S_i \rightarrow R$ is a utility function for player i.

The strategy of player i will be selected by his/her program. The players' programs can be specified in any agreed-upon known programming language. In this paper we use a simplified language with obvious semantics. This will not cause any loss of generality. Similar constructs can be found in any existing programming language. One of the properties of the programs represented in this language is that they are loop-free, and therefore each such program trivially halts.[2]

We need to define what a program for a player is. Notice that a program is a strategy in a the game where the player chooses from a set of available programs. As a result, one needs to carefully define the set of available programs. This is essential in order that the set of strategies in the corresponding meta-level game will be defined. The tool that computer science provides us in this regard is the theory of formal languages (and in particular the Backus Normal Form for specifying the syntax of the programs). This provides a well-defined meta-syntax in order to formally define the allowed computer programs, and as a result the set of available strategies. The exact formal language that defines the syntax of the programs used in this paper appears in the appendix. For ease of exposition we will refer in the main text only to the programs supported by this syntax that are used as part of the proofs. Indeed, the exact syntax of the programs is not that important as long as that the programming language supports the simple operators that we will use. These include operators like assignment of a value to a variable, IF THEN ELSE statements, and the comparison of two elements (where the elements can be files, variables, etc.)

[2]If one uses other languages, then we need to require that the program will halt. This issue can be tackled in a straightforward manner, namely by requiring that we will reach a ”DO” action (where the program stops) regardless of other players' programs.

3 Program equilibrium

Given a game G, let us denote the set of possible programs of player i with respect to G by $PROG_i(G)$, and the set of possible program profiles by $PROG(G)$. A tuple of programs $p = (p_1, \ldots, p_n) \in PROG(G)$ yields a probability distribution over the strategy profiles in $S = \Pi_{i=1}^{n} S_i$. We will denote by $U_i(p)$ the corresponding expected utility for player i.[3].

A tuple of programs $(p_1, \ldots, p_n) \in PROG(G)$ will be called a *program equilibrium* if for every i $(1 \leq i \leq n)$, we have that there is no $p'_i \neq p_i$, such that $U_i(p_1, \ldots, p_{i-1}, p'_i, p_{i+1}, \ldots, p_n) > U_i(p)$. Notice that a program equilibrium is a Nash equilibrium of the game where the players choose programs.[4]

Consider now the famous (one-shot) prisoners dilemma (PD). In the PD there are two possible strategies: 1 (coop) and 2 (defect). For every player i, the payoff for playing 1 is a if the other player cooperates, and b if he defects, and the payoff for playing 2 is c if the other cooperates, and d if he defects, where $c > a > d > b$. In our programming language the action to be selected is represented by a pair (x, y), where x (resp. y) is the probability for "coop" (resp. "defect").

Consider now the following program (to be adopted by both players):

```
• IF $P_1 = P_2$ then DO (1,0);
  ELSE DO (0,1);
  STOP;
```

In the above program P_1 refers to the program of player 1 and P_2 refers to program of player 2. $DO(a, b)$ is a call for action: play the first strategy (coop) with probability a and the other one (defect) will probability b. This fits the general syntax of programs discussed in the appendix.

Notice that the (potentially mixed) strategy of a player in a given game, to be called by the program using a DO action, is an output of the program. The players' strategies

[3]In order to discuss settings where participants might submit buggy programs, we will assume that programs that are syntactically incorrect lead to some highly negative payoff for a player who submits an incorrect program, and some highly positive payoffs for all other players. Formally, programs that are syntactically incorrect can be associated with a distinguished program *false*. We can then assume that $U_i(p) > U_i(s)$ for every program profile p and strategy profile s, if player i's program is the *false* program (i.e. it is not syntactically correct and does not always determine a unique action to be performed by the computer, and therefore will not be executed). In addition, in this case $U_j(p) > U_j(s)$ for every agent j who does not submit the false program in p, and for every strategy profile s. Given this approach the existence of *false* programs will not affect the results. We will therefore assume the strategies are syntactically correct.

[4]One issue that deserves further attention is the fact that computer programs are finite, and can not represent real numbers up to arbitrary precision. Technically speaking, this will imply that if we wish to refer to arbitrary probability distributions, we can represent them up to a precision of some $\epsilon > 0$, where this ϵ is determined by the computer architecture. For example, if we can represent up to k binary digits then we can represent the probability of choosing a strategy up to a precision of 2^{-k}. The set of possible programs will be always restricted by the machine architecture, i.e. the machine architecture determines the corresponding ϵ. Our setting and results can be adapted to that setting, but for ease of exposition we omit detailed discussion of these topics.

in the strategy profile that the programs output are assumed to be executed in parallel, as in classical work on one-shot games. Indeed, the fact that a computer runs the players' programs does not necessarily mean it is the authority to execute the selected strategies; the computer is only the tool to run the players' programs that determine the identity of the strategies to be selected (this is standard in work that involves computational models in game-theoretic settings; see e.g. [6]).

We can now prove:

Theorem 1 *There exists a program equilibrium for the PD, which yields cooperation by both parties.*

Proof:

If both players adopt the above program, then the players have programs with the same syntax (they are equivalent as "files"), and therefore the condition $IF \;\; P_1 = P_2$ holds, and $DO(1, 0)$ (i.e. cooperate) will be selected by both players. If one player adopts the specified program and the other deviates from that program, then the above condition does not hold any more. Since in this case the program instructs $DO(0, 1)$ the payoff for the deviator is at most $d < a$, which yields the desired result. ▮

4 Program equilibrium may yield any feasible individually rational payoff vector

Given a game G, denote the possible probability distributions over the strategies available to player i (aka mixed strategies for player i) by $\Delta(S_i)$. The threat point for player i is $v_i = min_{s_{-i} \in \Pi_{j \neq i} \Delta(S_j)} max_{s \in S_i} u_i(s, s_{-i})$. Let $minimax(i, j)$ denote player j's strategy in some arbitrary $s_{-i} \in \Pi_{j \neq i} \Delta(S_j)$ such that $max_{s \in S_i}(u_i(s, s_{-i})) = v_i$ (notice that $(minimax(i, j))_{j \neq i}$ equals s_{-i}, and guarantees that player i will not obtain a value greater than its threat point). A feasible individually rational payoff vector for a game G is one that is obtained by some mixed strategy of G, where the payoff for player i is greater than or equal to v_i (for every player i).

Below we will use $feasible$ to denote a strategy profile for the players, where $feasible_i$ is player i's strategy in this strategy profile.

Consider now the following program. As we will later show this program can be used in order to lead the players to use (as a program equilibrium) the feasible individually rational program $feasible$. Since such a program is available for any given feasible strategy profile $feasible$, and lead the players to play it in a program equilibrium if $feasible$ is individually rational, then we get that any feasible individually rational payoff can be obtained in a program equilibrium. This program obeys standard syntactic rules; for the more general definition of the sets of possible program the reader may consult the appendix. We discuss the parts of the syntax relevant to the mentioned program, soon after its presentation.

- $X(1,2) := minimax(1,2);$
- $X(1,3) := minimax(1,3);$
- ...
- ...
- $X(1,n) := minimax(1,n);$
- ...
- ...
- ...
- ...
- $X(n,1) := minimax(n,1);$
- $X(n,2) := minimax(n,2);$
- ...
- ...
- $X(n,n-1) := minimax(n,n-1);$
- $X(1,1) := feasible_1;$
- ...
- ...
- $X(n,n) := feasible_n;$
- $IF \;\; MYINDEX \neq 1$
  $THEN$
  $IF \;\; P_1 \neq MYPROGRAM \;\; THEN \; \{DO(X(1, MYINDEX)); STOP;\}$
  $ELSE;$
  $ELSE;$
- ...
- ...
- ...
- $IF \;\; MYINDEX \neq n$
  $THEN$
  $IF \;\; P_n \neq MYPROGRAM \;\; THEN \; \{DO(X(n, MYINDEX)); STOP;\}$
  $ELSE;$
  $ELSE;$

- $DO(X(MYINDEX, MYINDEX))$;
- $STOP$;

Elements of the form $X(i, j)$ denote variables. Variables can be assigned values (constants). The values/constants themselves can be (mixed) strategies. For example, $X(i, i) := feasible_i$ assigns to the variable $X(i, i)$ the strategy of player i in the strategy profile $feasible$, and $X(i, j) := minimax(i, j)$ assigns to $X(i, j)$ the strategy of player j in the corresponding s_{-i} leading to the threat value v_i (as described above). Each program has an index, which corresponds to the player who uses it, and MYINDEX refers to the index of the player who uses the program (so if player i is using the program then MYINDEX is i). MYPROGRAM refers to the syntax (as a file) of the program of the player who uses it, and P_i refers to the program of player i.

Given a game G, and considering a strategy profile, $feasible$, we get an instantiation of the above-mentioned program, by replacing the $feasible_i$'s and the $minimax(i, j)$'s, by appropriate constants (mixed strategies appropriate for the game in question). The idea of the above is that if all programmers adopt the above program then the desired individually rational strategy profile will be played. Any deviation by a single player will cause the deviator to be punished. As in the case of the prisoners dilemma, this kind of desired behavior can be obtained in the context of one-shot games using the ideas developed by von Neumann. The computer which executes the related programs can refer to them both as syntactic files as well as a set of instructions. We can show:

Theorem 2 *Given a game G, any individually rational strategy profile of it is played in some program equilibrium. As a result, the set of program equilibrium payoffs coincides with the set of feasible and individually rational payoffs of G.*

Proof: Assume that all players use the previously prescribed program (where $feasible_i$ and $minimax(i, j)$ are replaced by appropriate constants). Notice that in this case, syntactically, all the players have the same program. On the other hand, the execution of the program may be different for different players. When all players adopt the prescribed program then player i will instruct the computer to perform $feasible_i$, and its expected payoff will be according to $feasible$. This is implied by the syntactic comparison of the programs' content, as prescribed by the programs. If player i deviates from the prescribed program, while the others stick to it, then the syntactic comparison will detect this deviation, and will cause each player $j \neq i$ to call for the execution of $minimax(i, j)$. Given that $feasible$ is individually rational, this will imply that a deviation is irrational. ∎

5 Discussion

This paper exposes a connection between the foundations of computer programming and economic theory. This connection is highly motivated by emerging markets, and by agent technology, which is becoming a central aspect of economic activity in computational settings.

There are several lines of research that have some (although mostly superficial) similarity with the work reported in this paper. Previous work dealt with correlation devices that support communication, as discussed for example in work on communication equilibrium [2]. In program equilibrium one may view the computer as a mediator, but the interaction between players is by having one player's program reason computationally about other players' programs. This suggests a distinguished concept, which builds explicitly on the structure of computer programs. Previous work has also dealt with CS metaphors in game theory. Of particular interest is the work on the effects of memory bounds on the evolution of cooperation [8, 12]. This work has been the first to tightly connect game theory and computer science. This has been later extended to yield further understanding of the effects of resource bounds (see e.g. [10]), as well as to incorporate other features of computing, such as communication bounds (see e.g. [16, 13]) and asynchronous interactions (see e.g. [7]). Our paper introduces a totally different connection, exploiting basic ideas in the architecture of computer programs. The reader may also wish to notice the similarity between our work and folk theorems in economics (see e.g. [3] and the extended discussion in [4]). In fact, programs allow the implementation of the ideas behind folk theorems in the context of one-shot games, by exploiting von Neumann's fundamental idea.[5] Finally, program equilibrium provides a rigorous and actual approach for implementing conditional moves, as discussed by e.g. Howard and Danielson in the context of the prisoners dilemma.[6] It provides semantics grounded in programming languages to instructions of the form "I will cooperate if you will cooperate". Given this semantics, which is based on one of the fundamental ideas in computer science, we can address general n-person games.

It is hard to ignore the time overlap between the introduction of the seminal CS ideas by von Neumann, and the introduction of game theory in the book by von Neumann and Morgenstern [15]. However, von Neumann's ideas are exploited in this paper in the context of equilibrium analysis in non-cooperative games, a concept that has been introduced only in the 50s.

Appendix: A formal language defining the allowed programs

For completeness we now present in this appendix a formal language providing formal definition of the set of allowed programs we consider. For a general introduction to the theory of formal languages the reader may consult [5].

Although the above meta-language we will use is standard, and it is not our aim to teach the related formal languages theory, we now mention some conventions that will allow the reader to better understand the notation.

In the meta-language we have variables and constants. The set of allowed programs is the set of strings of constants that can be generated by finitely many applications of derivation

[5]Notice that although we obtain any feasible and individually rational payoffs, there is no notion of "convex hull" of programs which is used here. Mixture of programs may be a subject for further study.

[6]See http://plato.stanford.edu/entries/prisoner-dilemma/#Transp for an overview.

rules, where a derivation rule assigns a value to a variable (in the meta-language). The symbol ::= is used as part of the meta-language for defining a derivation rule. A variable name appears in between < and >. Notice that we will have here an inductive definition. The symbol || stands for concatenation, and the symbol | separates between possibilities. The following very simple example will illustrate the above conventions. Assume that a player has two strategies, 1 and 2, that stands for "print 1" and "print 2". Assume that $DO(x, y)$ instructs the computer to print 1 with probability x and 2 with probability y. We are interested in formally defining a set of programs that consist of printing sequences of 1's and 2's (notice that in this very simplified example we do not have conditions or inputs, etc.) We can now introduce a formal language defining a set of allowed programs. In the formalism below Λ stands for the empty program:

- $< program >::= \Lambda| < program > || < program > |DO(< action >)$
- $< action >::= (0, 1)|(1, 0)$

In words, a program is the empty program or concatenation of two programs, or have the form $DO(< action >)$ where $< action >$ can be replaced by a probability distribution over the available strategies. In our example the allowed actions are only the deterministic strategies. The variable $< action >$ can be replaced by an appropriate derivation rule with (1,0) or (0,1) (that stands for "printing 1" and "printing 2" respectively). Hence, the program that prints 1,2,1 is defined (and allowed) by using the following derivation: $< program > \to < program >< program > \to < program >< program >< program > \to DO(< action >) < program >< program > \to DO(< action >)DO(< action >) < program > \to DO(< action >)DO(< action >)DO(< action >) \to DO(1, 0)DO(< action >)DO(< action >) \to DO(1, 0)DO(0, 1)DO(< action >) \to DO(0, 1)DO(1, 0)DO(0, 1)$

We now describe the formal language defining the syntax of allowed programs that we consider in the paper. We first present the definition and then further expand on the syntax and semantics.

- $< program >::= \Lambda|$
  $< program >; |$
  $< program > || < program > |$
  $\{< program >< program >\}|$
  $STOP|$
  $< variable >:=< action > |$
  $DO(< action >)|$
  $DO(< variable >)|$
  $IF < condition > THEN < program > ELSE < program >$
- $< condition >::=$
  $(< program - index > | < variable > | < action > | < player - index >)||$
  $(= | \neq)||$
  $(< program - index > | < variable > | < action > | < player - index >)$

- $< action >$::= a tuple of real numbers (to denote a probability distribution)
- $< program - index >$::= $P_1|P_2|\cdots|P_n|MYPROGRAM$
- $< player - index >$::= $1|2|\cdots|n|MYINDEX$
- $< variable >$::= $X(integer, integer)$

Notice the difference between ::=, :=, and =. As we mentioned, ::= is part of the meta-language that defines the derivation rules. The symbol := stands for the assignment operator in programming languages, when we assign a value to a variable, and the symbol = stands for the standard equality condition when we compare terms. The symbol ";" is a standard sign for "end of command".

The type $< integer >$ is a "standard type" and refers to the standard integers. A special element is $< action >$ which denotes a mixed strategy (probability distribution on the pure strategies) in the given game. Notice also the use of "{" and "}": the use of these symbols is to instruct that any set of instructions in between "{" and "}" should be executed without interruption; notice that this is different from concatenation in the meta-language that defines the syntax. The symbols "{" and "}" are typically used when we write constructs of the form $IF < condition > THEN < program >$, in order to signal where the set of instructions that should be executed, if the condition holds, ends. For example, IF *cond* $THEN\{program1\ program2\}ELSE\ program3$ will instruct the computer in the case that the condition *cond* holds, to perform *program*1, to be followed by the execution of *program*2. In the case where *cond* does not hold, then *program*3 will be executed.

P_i refers to the program of player i (i.e. the content, as a file, of that program). In addition, MYPROGRAM refers to the program where it is mentioned. Similarly, each player is associated with an appropriate index. The term MYINDEX refers to the index of the player who submitted the program where it is mentioned. An action defines a probability distribution over the strategies of a player. During execution, when an action is called by the operator DO, the computer will select a strategy based on the probability distribution associated with that action, and halt. There are also variables, that may contain the description of actions as their values. When applied to a variable, the operator DO will call for the execution of the action assigned to the corresponding variable. DO can not be applied to a variable that its value is undetermined. Variables are taken to be two-dimensional arrays of the form $X(i, j)$. The command STOP calls the program to stop.

Constructs that are similar to the above can be found in any programming language. A program will be executed only if it uniquely determines a valid action (i.e. probability distribution over strategies) for the player (given any set of programs of the other players) . The verification of this is trivial for the style of programs presented above. All that we need to check is that any branching in the program leads to a DO action, where DO refers to an action associated with a probability distribution over strategies, or to a variable that has been assigned a corresponding value. Notice that STOP is used therefore just for syntactic convenience/clarity, since a program always stops soon after it faces and performs a DO action.

References

[1] L.M. Ausubel. Computer implemented methods and apparatus for auction. U.S. Patent No. 5,905,975, 1999.

[2] F. Forges. An approach to communication equilibrium. *Econometrica*, 54(6):1375–1385, 1986.

[3] D. Fudenberg and E. Maskin. The folk theorem in repeated games with discounting or with incomplete information. *Econometrica*, 52:533–554, 1986.

[4] D. Fudenberg and J. Tirole. *Game Theory*. MIT Press, 1991.

[5] J. E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[6] N. Megiddo and A. Wigderson. On Play by Means of Computing Machines. In *The 1st conference on Theoretical Aspects of Reasoning About Knowledge*, 1986.

[7] D. Monderer and M. Tennenholtz. Distributed games. *Games and Economic Behavior*, 27:55–72, 1999.

[8] A. Neyman. Bounded complexity justifies cooperation in the infinitely repeated prisoner's dilemma. *Econ. Lett.*, 19:227–229, 1985.

[9] R. Guttman P. Maes and A. Moukas. Agents That Buy and Sell. *Communications of the ACM*, 42(3):81–91, 1999.

[10] C. H. Papadimitriou. On players with a bounded number of states. *Games and Economic Behavior*, 4:122–131, 1992.

[11] T. Roughgarden. The price of anarchy is independent of the network topology. In *Proceedings of the 34th Annual ACM Symposium on the Theory of Computing*, pages 428–437, 2001.

[12] A. Rubinstein. Finite automata play the repeated prisoner's dilemma. *Journal of Economic Theory*, 39:83–96, 1986.

[13] Y. Shoham and M. Tennenholtz. On rational computability and communication complexity. *Games and Economic Behavior*, 35:197–211, 2001.

[14] J. von Neumann. First draft of a report on the edvac, contract no. w-670-ord-402 moore school of electrical engineering, univ. of penn., philadelphia. Reprinted (in part) in Randell, Brian. 1982. Origins of Digital Computers: Selected Papers, Springer-Verlag, Berlin Heidelberg, pp. 383–392., 1945.

[15] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, 1944.

[16] E. Zemel. Small talk and cooperation: A note on bounded rationality. *Journal of Economic Theory*, 49:1–9, 1989.