Building a Reinforcement Learning A.I. for the Iterated Prisoner's Dilemma using Soar cognitive architecture

BY

KONSTANTINOS THOMAS

MASTER THESIS

Faculty of Philosophy and History of Science,

National and Kapodistrian University of Athens

2018

Athens, Greece

Supervisors:

Professor Aristides Hatzis
Dr. Yulie Foka Kavalieraki

META:

KONSTANTINOS THOMAS

Student ID: 008/15

Graduate Program on

History and Philosophy of Sciences and Technology

National and Kapodistrian University of Athens

Department of Natural Sciences

Faculty of Philosophy and History of Science

Thesis Committee:

Aristides Hatzis, Associate Professor

Chrysostomos Mantzavinos, Professor

Dr. Yulie Foka Kavalieraki

ABSTRACT

This thesis involves the creation of an Artificial Intelligence agent that uses Reinforcement Learning (Q-Learning) in order to discern an optimal strategy to the game-theoretic Iterated Prisoner's Dilemma game. The agent is built using the Soar cognitive architecture and starts with sole knowledge of its two possible moves - Cooperation and Defection. Exploring the problem space, blindly at first by executing random moves, the A.I. agent quickly develops an intuition as of how the game is played through the Rewards it receives after each move. As more rounds go by, the agent starts realizing ways to handle the different opponents in order to maximize its payoff. Once its training is complete we examine how it fares compared to the top deterministic IPD strategy, Tit for Tat, which was the winner of both Axelrod's tournaments. Eventually, we examine the aggregate preference that the agent heuristically developed, for each individual move and note how they contrast with the rules of known deterministic strategies.

Keywords: Soar, Cognitive Architecture, Iterated Prisoner's Dilemma, Artificial Intelligence, Machine learning, Reinforcement Learning

ABSTRACT in Greek language

Σε αυτή την εργασία φτιάχνουμε ένα πρόγραμμα Τεχνητής Νοημοσύνης το οποίο χρησιμοποιεί Ενισχυτική Μάθηση (Q-Learning) προκειμένου να καταλήξει σε μια αποτελεσματική λύση του επαναλαμβανόμενου Διλήμματος του Φυλακισμένου. Για την κατασκευή του προγράμματος χρησιμοποιούμε την Γνωστική Αρχιτεκτονική Soar. Το πρόγραμμα ξεκινάει το παιχνίδι με μόνη γνώση για τον κόσμο την ύπαρξη των δύο δυνατών κινήσεων του - Συνεργασία ή Προδοσία - και καμία περαιτέρω πληροφορία για τους κανόνες του παιχνιδιού. Καθώς το παιχνίδι εξελίσσεται, το πρόγραμμά μας, κάνοντας αρχικά τυχαίες κινήσεις, πολύ σύντομα αρχίζει να καταλαβαίνει πως να παίξει, μέσω των πόντων που λαμβάνει ως επιβράβευση. Καθώς οι γύροι εκτυλίσσονται, η κατανόηση του ξεπερνάει πια απλώς τους κανόνες του παιχνιδιού και αρχίζει να αναπτύσσει στρατηγικές για το πως να αντιμετωπίσει κάθε διαφορετική αντίπαλη στρατηγική, προκειμένου να αποσπάσει οσο το δυνατόν περισσότερους πόντους απο αυτήν. Αφού το πρόγραμμα μας παίξει μερικές δεκάδες χιλιάδες γύρους, συγκρίνουμε την απόδοσή του με αυτή της πιο αποτελεσματικής γνωστής στρατηγικής του επαναλαμβανόμενου Διλήμματος του Φυλακισμένου - της στρατηγικής Tit for Tat - νικήτρια και των δύο τουρνουά του καθηγητή Axelrod. Τέλος, αναλύουμε την ευριστική στρατηγική στην οποία το πρόγραμμα μας κατέληξε ως την πιο αποτελεσματική και την αντιπαραθέτουμε με τις γνωστές ντετερμινιστικές στρατηγικές του παιχνιδιού.

Λέξεις Κλειδιά: SOAR, Γνωστική Αρχιτεκτονική, Δίλημμα Φυλακισμένου, Τεχνητή Νοημοσύνη, Μηχανική Μάθηση, Ενισχυτική Μάθηση

1. Game Theory and the Prisoner's Dilemma

1.1. Game Theory

In Plato's *Symposium* and *Laches* dialogues, Socrates recalls an episode from the Battle of Delium. Consider a soldier at the front, waiting with his comrades to repulse an enemy attack. It may occur to him that if the defense is likely to be successful, then it isn't very probable that his own personal contribution will be essential. Furthermore, if he stays, he runs the risk of being killed or wounded - apparently for no point. On the other hand, if the enemy is going to win the battle, then his chances of death or injury are even higher, and quite clearly to no point. Based on this reasoning, it would appear that the soldier is better off running away regardless of who will win the battle. Of course, if each of the soldiers anticipate the same reasoning from their fellow soldiers, they will soon flee in panic and the battle will be lost without ever being fought. This is considered to be one of the first game-theoretic insights, going back to ancient times, way before game theory has been rendered mathematically systematic.

The study of such models of conflict and cooperation between intelligent decision-makers, within a competitive situation, came to be known as *Game theory* in the late 1920's. Following a paper published by John von Neumann titled *Theory of Parlor Games*. In some respects, Game theory is the science of strategy, or at least, that of the optimal decision-making by competing actors in a strategic setting. Each competitor has the same ultimate goal, to discover a dominant strategy and exploit it, securing as much of the available reward possible. There are numerous theoretical and real-world scenarios where one can apply Game theory, yet the *Prisoner's Dilemma* is probably its most iconic example, which is why we chose to focus on it.

1.2. Prisoner's Dilemma

The basis of the game-theoretical prisoner's dilemma model was originally framed in 1950 by two mathematicians working at the RAND Corporation, Merrill Flood and Melvin Dresher. It was only in later years that professor Albert William Tucker developed the model further, using it as a teaching tool for his graduate psychology students, giving it the interpretation of the prison

sentences and appropriately naming it *Prisoner's Dilemma* (henceforth PD) [1]. Tucker presented the game as follows:

> Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other. The prosecutors lack sufficient evidence to convict the pair on the principal charge. They hope to get both sentenced to a year in prison on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either to: betray the other by testifying that the other committed the crime, or to cooperate with the other by remaining silent. The offer is:
>
> - If A and B each betray the other, each of them serves 2 years in prison
> - If A betrays B but B remains silent, A will be set free and B will serve 3 years in prison (and vice versa)
> - If A and B both remain silent, both of them will only serve 1 year in prison (on the lesser charge)

It is assumed that both understand the nature of the game, and that despite being members of the same gang, they have no loyalty to each other and will have no opportunity for retribution or reward outside the game. They are practically two strangers who will never meet again.

Here is the game in matrix form:

Prisoner B **Prisoner A**	**Prisoner B stays silent (*cooperates*)**	**Prisoner B betrays (*defects*)**
Prisoner A stays silent (*cooperates*)	Each serves 1 year	Prisoner A: 3 years Prisoner B: goes free
Prisoner A betrays (*defects*)	Prisoner A: goes free Prisoner B: 3 years	Each serves 2 years

Table 1: Prisoner's dilemma payoff matrix

Examining the above table one thing becomes evident: regardless of what the other decides, each prisoner gets a higher reward by betraying the other ("defecting"). The reasoning involves an argument by dilemma: B will either cooperate or defect. If B cooperates, A has a choice. He can cooperate as well, receiving the 1 year sentence for mutual cooperation, or he can defect, receiving a zero (0) year sentence and serving no time at all. So it pays to defect if one thinks the other player will cooperate. Now suppose player B will defect, player's A choice is again between cooperating, which would give him the worst possible sentence of 3 years, and defecting, which would result in a mutual punishment that sentences both players with 2 years. Thus, it pays to defect in the scenario of the other player defecting, as well. This means that it is better to defect if one thinks that the other player will cooperate as it is also better to defect if one thinks the other player will defect. Ergo, player A should defect either way. Parallel reasoning will show that the exact same strategy is the optimal one for player B.

Because defection always results in a better payoff than cooperation regardless of the other player's choice, in game-theoretic terms, defecting is a *dominant strategy* of the game. Mutual defection is the only *dominant strategy equilibrium* in the game and therefore its only strong *Nash equilibrium* (i.e. the only outcome from which each player could only do worse by unilaterally changing strategy). The dilemma, then, is that mutual cooperation yields a cumulative better outcome than mutual defection but it is not the rational outcome because from a self-interested perspective, the choice to cooperate at the individual level is irrational.

The Prisoner's Dilemma is simply an abstract formulation of some very common and very interesting situations in which what is best for each person individually leads to mutual defection, whereas everyone would have been better off with mutual cooperation. Incidentally, Plato's Battle of Delium example can also be formulated as a PD game employing a matrix just like the above, using "Self, Others" as players and "Fight, Flee" as available moves.

1.3. Iterated Prisoner's Dilemma

In realistic situations, people often encounter each other more than once. Correspondingly, some social interactions can be modeled by repeated PD games. This super-game of the PD is called the *Iterated Prisoner's Dilemma* (IPD). An IPD game consists of multiple, successive plays with the same opponent, which introduces a big twist on the original standalone PD

game. Players may now frame their moves on their opponent's strategy, insofar as each can deduce it from the previous plays of each opponent. Much more like in real world situations, one can keep track in memory of someone else's past moves while they were associated and examine their historic moves in order to predict their future ones in analogous circumstances. Thus, labeling others as "trustworthy", "cooperative", "selfish" etc., for future reference. Unlike in the standard prisoner's dilemma, in the iterated version of the game, persistently defecting is counter-intuitive and fails to emulate the behavior of human players. In an IPD game, it may be beneficial even for a selfish agent to cooperate on some iterations in the hope of soliciting cooperation from its opponent.

It is noteworthy that, if the number of iterations in an IPD game is known, then the last iteration becomes the standalone PD game since the opponent will not have a chance to later retaliate. Hence, in the last iteration each agent is motivated to defect. Because both agents know that the opponent is going to defect on the last round, they have no motivation to cooperate on the second to last round either. This can inductively be carried out backwards all the way to the beginning of the interaction, which makes mutual defection the only possible Nash Equilibrium (and thus, paradoxically, some irrational agents will do better, under those conditions, than rational ones). Because fixed horizon IPD games have this characteristic, strategy testing on IPD games usually focuses on games with an indefinite horizon, i.e. the players are unaware of how many iterations are still to come. Thereby, 'always defect' may no longer be a strictly dominant strategy, but only a Nash equilibrium. Supportive of this is a result, amongst the many shown by Robert Aumann in a 1959 paper, according to which rational players repeatedly interacting for indefinitely long games, can sustain the cooperative outcome [23].

The structure of the traditional Prisoner's Dilemma can be generalized from its original prisoner setting in order to make it more permissive for general purpose testing. Instead of the payoffs being punishments (prison years), they are instead extrapolated to rewards, while keeping the same symmetry between each other. Ultimately, Table 1 becomes as follows:

Row Player Column Player	Cooperate	Defect
Cooperate	**R=3, R=3** Reward for mutual cooperation	**S=0, T=5** Sucker's payoff, and Temptation to defect
Defect	**T=5, S=0** Temptation to defect and Sucker's payoff	**P=1, P=1** Punishment for mutual defection

Table 2: Prisoner's dilemma payoff matrix

For a game to be considered a PD game the following condition must apply:

$$T > R > P > S$$

The Temptation payoff must be the highest payoff, followed by the Reward payoff, then the Punishment payoff and finally the Sucker's payoff.

Another rule that must hold is:

$$2R > T + S$$

This rule is enforced to ensure that constantly interchanging cooperation and defection with each other between rounds (namely, each getting the Temptation payoff on one round and the Sucker's payoff on the next) will not yield better results than steadily cooperating on both rounds. Two Rs must pay better than T + S.

1.4. IPD Strategies

Unlike in the standard PD games, it is generally assumed that in the IPD there exists no universal ultimate strategy whereby one player can enforce a claim to an unfair share of

rewards. The optimal strategy depends upon the strategies of likely opponents, and how they exchange defections and cooperations between them.

The iterated version of the PD was discussed from the time the game was devised, but interest accelerated after certain influential publications by Robert Axelrod [2][3] in the early eighties. Axelrod invited professional game theorists to submit computer programs for playing IPDs. All 14 programs, plus a 15th program that cooperated and defected at random, were entered into a tournament in which each program played every other (as well as a clone of itself) for 200 round IPDs. For example, if two strategies cooperated throughout the whole game, each would receive a 600 points score. If both constantly defected, they would each obtain 200 points. And if one always cooperated while the other always defected, the defecting strategy would earn the maximum possible payout of 1000 points, while its naive opponent would receive 0 points.

It is easy to see that in such a game no strategy is the “best” in the sense that its score would be highest among any possible group of competitors. If the rest of the group’s strategies do not take into account their opponent’s previous interactions in choosing their next move, it would be best to defect unconditionally since each round would be analogous to a standalone PD game. If the other strategies all begun by cooperating, however “punish” any defection against them by defecting on all subsequent rounds, then a policy of unconditional cooperation is the most effective. But playing that cooperating strategy against a group of more aggressive opponents will have a player winding up with the worst possible payout. Yet, some strategies have features that seem to allow them to do well in a variety of environments.

The strategy that scored highest in Axelrod's initial tournament, Tit for Tat (henceforth TFT), simply cooperates on the first round and imitates its opponent's previous move thereafter. More significant than TFT's initial victory, perhaps, is the fact that it won Axelrod's second tournament, whose sixty three entrants were all aware of the first tournament’s results.

Axelrod attributed the success of TFT to four properties. Firstly, it is ***nice***, meaning that it is never the first to defect. Interestingly, the eight nice entries in Axelrod's tournament were the eight highest ranking strategies. Secondly, it is ***retaliatory***, since it instantly punishes any defectors, making it difficult to be exploited by any non-nice strategies. Thirdly, it is ***forgiving***, in the sense of being willing to cooperate even with those who have defected against it (provided their defection wasn't in the immediately preceding round). An unforgiving rule is incapable of ever getting the Reward payoff after its opponent has defected once. And lastly, it

is ***clear***, presumably making it easier for other strategies to predict its behavior so as to facilitate mutually beneficial interaction [21].

In addition, these findings have interesting evolutionary interpretations, which were the main motivation behind Axelrod's research and also the theme of his book which, shortly followed the tournament in 1984, titled *The Evolution of Cooperation.* Nevertheless, we will not delve into such an interpretation here, as it stretches beyond the scope of this thesis.

1.4.1. Strategy Analysis

For our cause, we will produce an environment similar to Axelrod's tournament, to have our agent train on and compete against different strategies. The ground rules are going to be the same. Each IPD game consists of 200 rounds and scored using an identical payoff matrix, the one in Table 2 above. All contestants, including the Soar agent, will play against each other and ranked by the sum of the total points they received throughout the tournament.

Our intentions are twofold. Firstly, we need to determine the performance level of the machine learning agent that we created. Will it be a mediocre contestant or manage to do as well, maybe even surpass, the known deterministic strategies? Secondly, after letting our agent compete for thousands of iterations we will analyze the strategy that the agent worked out "by experience" and how does that heuristic decision scheme relates to the existing strategies. The presumption is that it should, to a degree, resemble the best known strategy of Tit for Tat.

For our simulation we selected 12 strategies out of the ones competed in the first and second tournaments, choosing evenly among the most successful, the least successful and. The competing strategies are the following:

1. Cooperate
2. Defect
3. Interchange (Hot-Cold)
4. Random
5. Tit for Tat (TFT)
6. Suspicious Tit for Tat (SuspTFT)
7. Tit for 2 Tats (TF2T)

8. Grofman
9. Grudger
10. Pavlov (win-stay, lose-switch)
11. Extort (ZD_Extort2)
12. Joss

In more detail, these strategies follow the rules described below:

Cooperate	Always cooperates.
Defect	Always defects.
Interchange	Interchanges repeatedly between defection and cooperation throughout the game.
Random	Selects randomly between defection and cooperation.
Tit for Tat	Starts by cooperating and then mimics the previous action of the opponent.
Suspicious Tit for Tat	Starts by defection and otherwise plays like Tit for Tat.
Tit for Two Tats	Starts by cooperating and then defects only after two consecutive defections by the opponent. Essentially forgiving each first defection.
Grofman	Cooperates on the first two rounds and returns the opponent's last action for the next 5. For the rest of the game Grofman cooperates if both players selected the same action in the previous round, and otherwise cooperates randomly with probability 2/7.

Grudger Starts by cooperating, however if at any point the opponent defects, it continuously plays defect for the rest of the game.

Pavlov Also known an *Win-Stay, Lose-Switch* (WSLS), starts by cooperation and keeps repeating the last move as long as it earned at least 3 points on the previous round, otherwise it switches action.

Extort This is zero-determinant strategy, also known as ZD_Extort2, which is defined by the following four conditional probabilities based on the last round of play:

- P(C|CC)=8/9
- P(C|CD)=1/2
- P(C|DC)=1/3
- P(C|DD)=0

Conditional probabilities, such as P(C|CC), are read “the probability that the strategy plays cooperation (C), after both players have played mutual cooperation (CC) on the previous round”.

Joss If the opponent cooperates, Joss cooperates on the next round with probability 0.9. Otherwise it plays Tit for Tat.

Next we need to write the code for all the strategies above as well as our A.I. agent’s, in order to run the competition simulation. There is a number of frameworks and programming languages that support Reinforcement Learning and we could use to program and run our project. Our choice is to use a Cognitive Architecture to do so, specifically Soar, which we will discuss on the next chapter.

2. The Cognitive Architecture of Soar

2.1. Artificial Intelligence

In the early 1950s, Herbert Simon, a political scientist who had already developed his theory of bounded rationality (for which he would later win a Nobel Prize) was working as a consultant at RAND Corporation. Anecdotally, he remembers seeing a machine printing out a map using ordinary letters and punctuation as symbols and realizing: a machine that could manipulate symbols could just as well simulate decision making and possibly even the process of human thought [4, pp. 41–44]. The program that printed the map had been written by Allen Newell, a RAND Corporation scientist in logistics and organization theory. In 1954, Newell, while attending a presentation on pattern matching, suddenly understood how the interaction of simple, programmable units could accomplish complex behavior including the intelligent behavior of human beings [4, p. 44]. Newell and Simon began to talk about the possibility of teaching machines to think. Their first project was a program that could prove mathematical theorems, like the ones used in Bertrand Russell and Alfred North Whitehead's *Principia Mathematica*. They enlisted the help of computer programmer Cliff Shaw, also from RAND, to develop the program and by 1956 the “Logic Theorist” was developed [16]. Eventually, Logic Theorist run on a computer at RAND's Santa Monica facility and soon proved 38 of the first 52 theorems in *Principia Mathematica*’s second chapter. The proof of theorem *2.85* was actually more elegant than the proof produced laboriously by Russell’s and Whitehead’s hand.

When Newell and Simon began to work on the Logic Theorist, in 1955, the field of artificial intelligence did not yet exist and would not be formed until the following summer, at a workshop at Dartmouth College in 1956 where the field of A.I. research was born. Attendees Allen Newell (CMU), Herbert Simon (CMU), John McCarthy (MIT), Marvin Minsky (MIT) and Arthur Samuel (IBM) became the founders and leaders of A.I. research [17]. The first generation of A.I. researchers was convinced that Artificial General Intelligence (AGI, aka “strong A.I.”) - namely, human-level “thinking machines” - was, not only possible, but would exist in just a few decades. As A.I. pioneer Herbert A. Simon wrote in 1965 "machines will be capable, within

twenty years, of doing any work a man can do." and in 1967 Marvin Minsky is quoted saying "Within a generation ... the problem of creating 'artificial intelligence' will substantially be solved." [4, p.109].

However, in the early 1970s, it became obvious that researchers had grossly underestimated the difficulty of the project. Funding agencies became skeptical of strong A.I. (AGI) and put researchers under increasing pressure to produce simpler, practical A.I. systems (aka “weak A.I.” or “classical A.I.”) [4, pp. 115-117]. As the 1980s began, the Japanese government revived interest in strong A.I., setting out a ten-year timeline that included strong A.I. goals like "carry on a casual conversation" [4, p. 211]. In response to this and to the success that weak A.I. systems already had, both industry and government pumped money back into the field [4, pp. 161–162, 197–203]. However, confidence in A.I. spectacularly collapsed in the late 1980s and the goal of AGI revolution did not get fulfilled once again. For the second time in 20 years, A.I. researchers who had predicted the imminent achievement of strong A.I. had been shown to be fundamentally mistaken. As a result, by the 1990s, the mainstream of A.I. research had again turned toward noarrower, domain-dependent and problem-specific solutions.

2.1.1. Weak and Strong A.I.

A.I. has worked across the full range of task complexity, from simple to very complex. However, for most AI systems the environment and the tasks are fixed and the system is structured to use knowledge for a limited set of tasks, specific algorithms and particular data structures. Most of our current Machine Learning approaches perform extremely well when trying to perform a certain task - like searching the internet, driving a car or playing a video game - but are unable to perform *all* of those tasks. When presented with any other task requirement besides the one they have been trained upon, their narrow A.I. is unable to comprehend and assess the new situation in order to respond appropriately. A good example of specialization is the famous chess-playing program *Deep Blue*. Deep Blue knows a great deal about chess and uses specialized knowledge (and hardware) to play chess at the highest levels however, this is the entirety of its abilities. Deep Blue would lose badly in a game of tic-tac-toe. We do not yet have computational systems that seamlessly integrate the many capabilities we associate with human intelligence. These capabilities include interacting with

dynamic complex environments, pursuing a wide variety of tasks, using large, diverse bodies of knowledge, planning, and continually learning from experience. It is a long and demanding list of items, yet one that AGI aspires to solve.

Artificial General Intelligence is an emerging field, dedicated to the development of “thinking machines” - i.e. general-purpose systems with intelligence comparable to that of the human mind (and perhaps ultimately well beyond human intelligence) - that could successfully perform *any* intellectual task that a human being can. Although *strong A.I.* was the original goal of the A.I. project during its inception, circumstances forced classical A.I. to steer away from the “Grand A.I. Dream”, creating the need for a seperate research field, namely AGI, to move along this direction. Common tests for AGI are:

- Alan Turing’s *Turing Test*, where a machine converses in text with a human evaluator, trying to fool him that it is a human that he is conversing with and
- Steve Woznik’s *Coffee Test,* where machine is required to enter an average American home and figure out how to make coffee: find the coffee machine, find the coffee, add water, find a mug, and brew the coffee by pushing the proper buttons.

Harder and more intricate tests exist as well, like having an A.I. passing university exams and earning a university degree, yet not even the simplest two tests, above, have been passed by any machine as of present time. Most mainstream A.I. researchers hope that AGI can be developed by the merging of A.I. programs that solve various sub-problems, using an integrated *Cognitive Architecture*. Due to this, cognitive architectures have seen a steady flow of research since their conceptual founding in the early 1960s.

2.2. Cognitive Architectures

A cognitive architecture can refer to a theory about the structure of the human mind. On the exterior, cognitive architectures look like a programmings languages, however, their computational constructs reflect assumptions about human cognition. One of the main goals of a cognitive architecture is to summarize the various results of cognitive psychology in a comprehensive computer model. However, the results need to be formalized so far as they can

be the basis of a computer program. The formalized models can be used to further refine a comprehensive theory of cognition, and more immediately, as a usable model.

The last decade has seen the emergence of a variety of cognitive architectures. This is good news, in general, for cognitive modeling because architectures provide a ready-made set of tools and theoretical constraints that can - according to architectural research methodology - assist the cognitive modeling enterprise by constraining the possible models of a set of phenomena or even making the “right” model an obvious consequence of the architectural constraints[5]. Just as special purpose programming tools, such as spreadsheets, give us the right language and tools for attacking specialized tasks, cognitive architectures are meant to give us the right constraints for building cognitive models. Although the various cognitive architectures often apply to overlapping cognitive capacities and share some similarities, they also make many different theoretical distinctions, which can of course have a major influence on the nature of cognitive models supported by each architecture. Two of the oldest, most developed and most studied architectures are Soar [5][6][7] and Act-R [8][9].

Soar and Act-Rare each based on a set of different theoretical assumptions reflecting, to a large extent, their different conceptual origins. Soar was developed by combining three main elements: 1) the heuristic search approach of knowledge-lean and difficult tasks; 2) the procedural view of routine problem solving and 3) a symbolic theory of *chunking* (creating memories of each chunk of knowledge discovered, after evaluating the outcome of a decision) designed to produce the power law of learning[6]. In contrast, Act-R grew out of detailed phenomena, well supported by empirical data, from memory, learning, and control [8][10]. Thus, the emphasis on Soar has been on general A.I. (functionality and efficiency), whereas the emphasis on ACT-R has always been on cognitive modeling (detailed modeling of human cognition). Due to the nature of our objective, which is heuristic problem solving, it seems appropriate to favour the former for our use case.

2.3. Soar

The goal of the Soar project is to develop the fixed computational building blocks necessary for general intelligent agents – agents that can perform a wide range of tasks and encode, use, and learn all types of knowledge to realize the full range of cognitive capabilities found in humans, such as decision making, problem solving, planning, and natural language understanding. It is both a theory of what cognition is and a computational implementation of that theory. Since its beginnings in 1983 as John Laird's thesis, it has been widely used by AI researchers to create intelligent agents and cognitive models of different aspects of human behavior.

2.3.1. Background

Soar was originally created in the 1980s by John Laird, Allen Newell, and Paul Rosenbloom at Carnegie Mellon University. It was an offshoot of the groundbreaking work done by Allen Newell and Herbert Simon from the mid 1950s through the mid 1970s, including pioneering work such as the aforementioned "Logic Theorist" (Newell and Simon 1956) [16], the "General Problem Solver" (Ernst and Newell 1969) [12], Heuristic Problem Solving (Simon and Newell 1958) [13] and on problem spaces (Newell 1991; Newell and Simon 1972) [14][15]. The "General Problem Solver" (GPS) created by George Ernst and Newell, used a single, general method called means-ends analysis to solve many different puzzle-style problems. Means-ends analysis uses knowledge about how operators (the means) reduce the difference between the current state and their goal state (the ends) and, moreover, it requires that there be an explicit representation of the available operators and the goal being pursued. Thus, GPS was limited in the types of knowledge it could use and the problems it could solve. But it was, nevertheless, the first AI system that could solve multiple problems.

The idea of Soar was born in 1981 by Laird and Newell in order to take a step beyond the GSP by creating a general cognitive architecture that could use, not just a single method such as means-ends analysis but whatever method was appropriate to the knowledge it had about a

task. Their goal was to make it possible for knowledge about problem solving to be decomposed into primitive (i.e. elementary) computational components that dynamically combine during problem solving. That approach, which led to the creation of Soar's first version, combined the idea of using *problem spaces* to organize behavior in solving a problem. In problem spaces, on which we will elaborate further in the following chapter, behavior is decomposed into selection of operators and their application to states to make progress toward a goal. States are representations of the current configuration and operators are the means by which a system can make deliberate changes on that configuration. For example, if the goal state consists of the configuration "empty water jug - full glass" and the current state is "full water jug - empty glass", the operator that would transform the current state to the goal state would be "pour water from jug to glass".

Soar was named for this basic cycle of State, Operator, And Result (SOAR), however, it is no longer regarded as an acronym.

2.3.2. A Theory of Cognition

Soar embodies multiple hypotheses about the computational structures underlying AGI. The original theory of cognition underlying Soar is the *Problem Space Hypothesis*, described in Allen Newell's book *Unified Theories of Cognition* [5] and dates back to the Logic Theorist, one of the first AI systems created as we discussed earlier. The Problem Space Hypothesis holds that all goal-oriented behavior can be cast as search through a space of possible states (a problem space) while attempting to achieve a goal. At each step, a single operator is selected and applied to the system's current state, which can lead to internal changes such as retrieval of knowledge from long-term memory, or external actions in the world. Inherent to the Problem Space Hypothesis is that all behavior, even a complex activity such as planning, is decomposable into a sequence of selection and application of primitive operators. When used to model human behavior, the execution of these operators corresponds to a 50 millisecond time interval for both Soar and ACT-R. Soar's real-time performance, of course, is hundreds to thousands of times faster [18, p.13].

A second hypothesis of Soar's theory is that although only a single operator can be selected at each step, forcing a serial bottleneck, the processes of selection and application are

implemented through *parallel* rule firings, which provide context-dependent retrieval of *procedural knowledge* (the knowledge needed in the performance of some task). Therefore, although the selection will decide on a single operator, the selection process is itself similar to internal reasoning, where new information arises by realizations happening as the thought evolves but without the input of any new knowledge. At the beginning of each cycle, the system uses the knowledge it has earned during the previous cycle, in order to elaborate as to which operator to select. Contemplating on this knowledge, the system draws some conclusions which conclusion will, in turn, be instantly used within the same selection process to elaborate and reach even further conclusions, until there are no more "new conclusions" to be found (Figure 1). At this point the best possible operator will be selected and the decision will pass to the operator application phase.

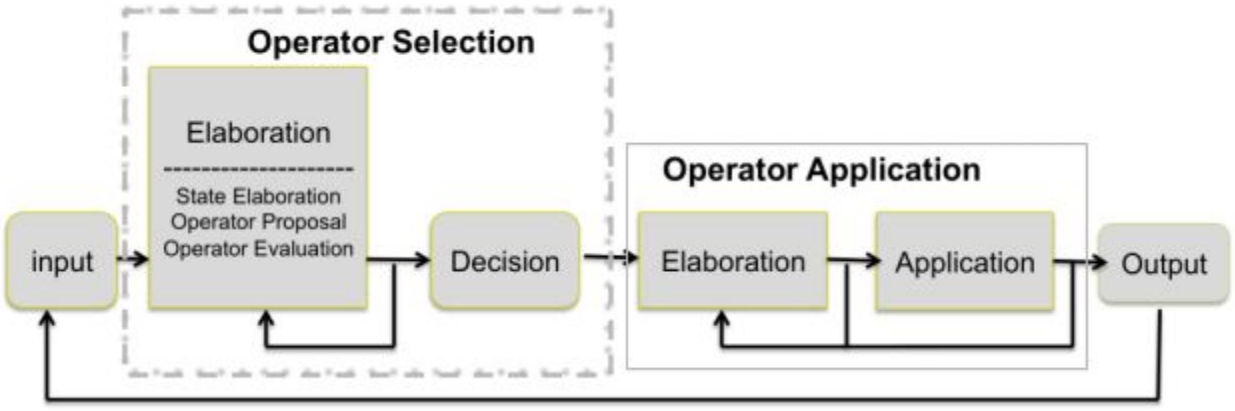


Figure 1 - The Soar Processing Cycle

This elaboration loop is essential since it escapes unnecessary and computationally costly processing cycles from taking place, simply to arrive at the same operator just a few full cycles later.

A third hypothesis is that, if the knowledge to select or apply an operator is incomplete or uncertain, e.g. a selection between two operators that no one has advantage over the other, an *impasse* arises and the architecture automatically creates a sub-state. In the sub-state, the same process of problem solving is recursively used, but this time with the goal to retrieve or discover knowledge about which operator to select so that decision making can continue. This

can lead to a stack of sub-states where traditional problem methods, such as planning or hierarchical task decomposition, naturally arise. When the results that have been reached in the sub-state can resolve the impasse, the sub-state and its associated structures are removed and the operator selection is finally made. The overall approach is called *Universal Subgoaling*.

These assumptions lead to an architecture that supports three levels of processing. At the lowest level, is bottom-up, parallel, and automatic processing. The next level is the deliberative level, where knowledge from the first level is used to propose, select, and apply a single action. These two levels implement fast, skilled behavior, and roughly correspond to Daniel Kahneman's System 1 processing level, as defined in his book *Thinking, Fast and Slow* [19]. More complex behavior arises automatically when knowledge is incomplete or uncertain through a third level of processing using substates, roughly corresponding to System 2.

A fourth hypothesis in Soar is that the underlying structure is modular, not in terms of task or capability based modules such as planning or language, but instead as task independent modules (Figure 2). These modules include:

- Decision making module
- Memory modules
    - Working memories (the representation of the current situation)
    - Long-term memories
        - Procedural (information related to the performance of some task)
        - Declarative (memory of facts and events)
        - Episodic (memory of what happened to an entity - the system itself)
- Learning mechanisms associated with all long-term memories
- Perceptual and motor modules

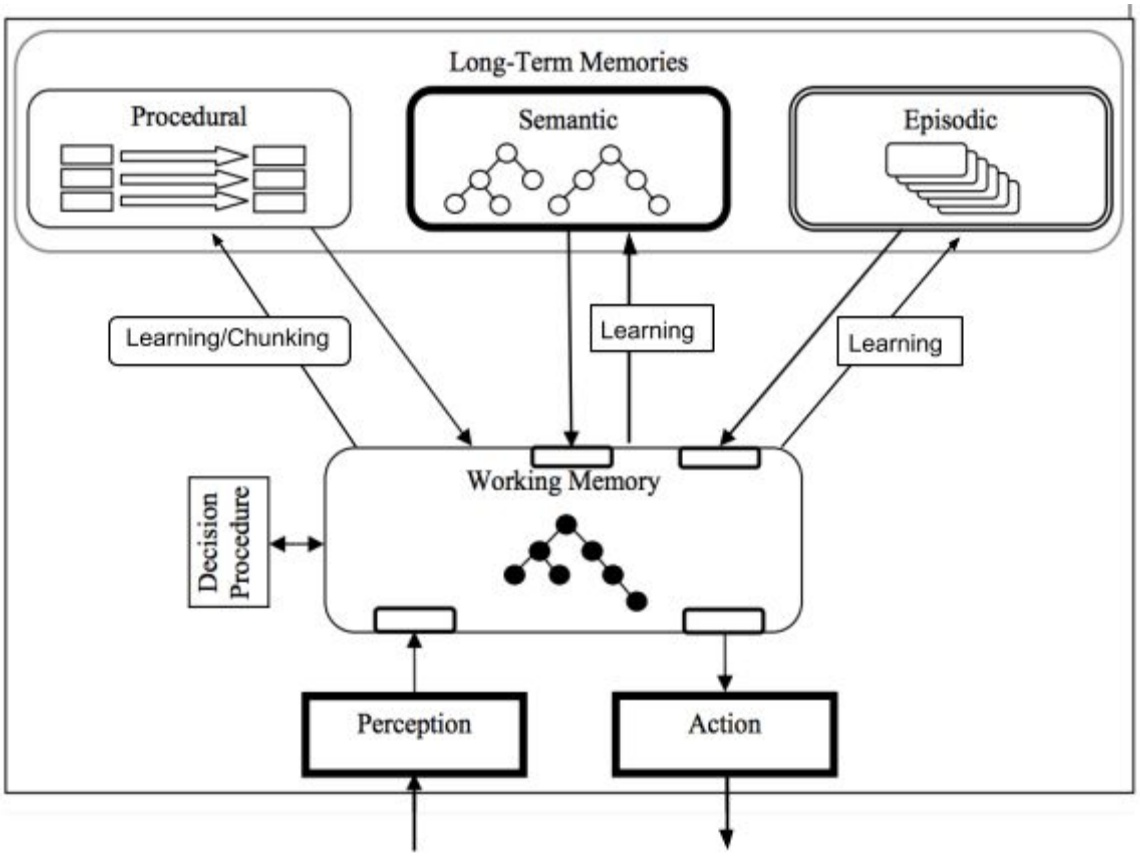


Figure 2 - Soar modules

A fifth hypothesis is that memory elements are represented as symbolic, relational structures. The hypothesis that a symbolic system is necessary for general intelligence is known as the physical symbol system hypothesis (PSSH) - a position in the philosophy of artificial intelligence formulated by Allen Newell and Herbert Simon [20]. An important evolution in Soar is that all symbolic structures have associated statistical metadata (such as information on recency and frequency of use, or expected future reward) that influences retrieval, maintenance, and learning of the symbolic structures.

2.3.3. Example of a Soar process

All of the above can be quite a handful for someone to grasp, especially anyone not familiar with cognitive architectures or, more generally, A.I. programming. Before delving into further technicalities I find it appropriate to explore an example, so as to cultivate a more intuitive feel of the entire Soar process. In the example we will demonstrate how Soar uses problem spaces, impasses and learning, to capture goal oriented behaviour. The problem selected for our example is a popular mathematical puzzle called *Tower of Hanoi.*

The Tower of Hanoi consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack on one of the rods. The disks are ordered by size with the smallest set at the top, thus making a conical shape represented by the *initial state* in *Figure 3* below. The objective of the puzzle (the *goal state*) is to move the entire stack of disks to one of the other rods while obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
3. No disk may be placed on top of a smaller disk.

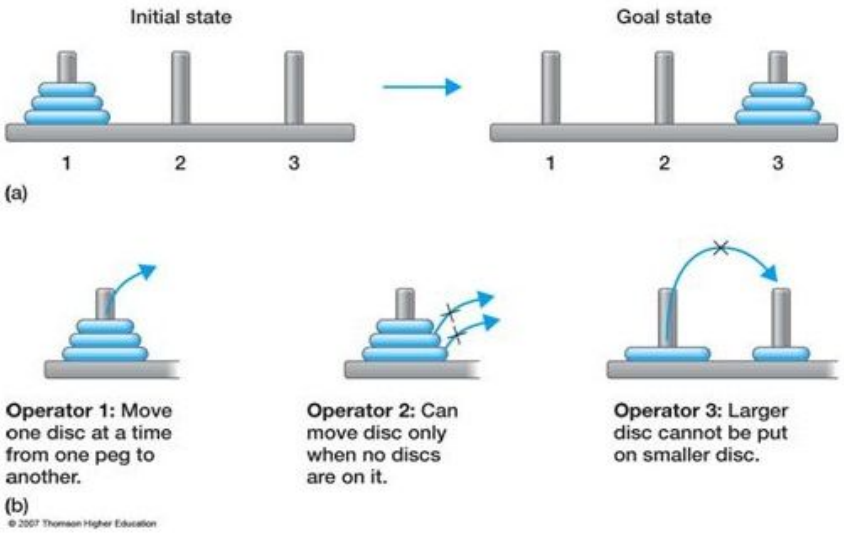


Figure 3 - The Tower of Hanoi

The puzzle can be played with any number of discs. In its simplest form, the game is played with 3 disks and the puzzle can be solved in a minimum of 7 moves, like so:

1) Small disk from rod 1 to rod 3
2) Medium disk from rod 1 to rod 2
3) Small disk from rod 3 to rod 2
4) Large disk from rod 1 to rod 3
5) Small disk from rod 2 to rod 1
6) Medium disk from rod 2 to rod 3
7) Small disk from rod 1 to rod 3

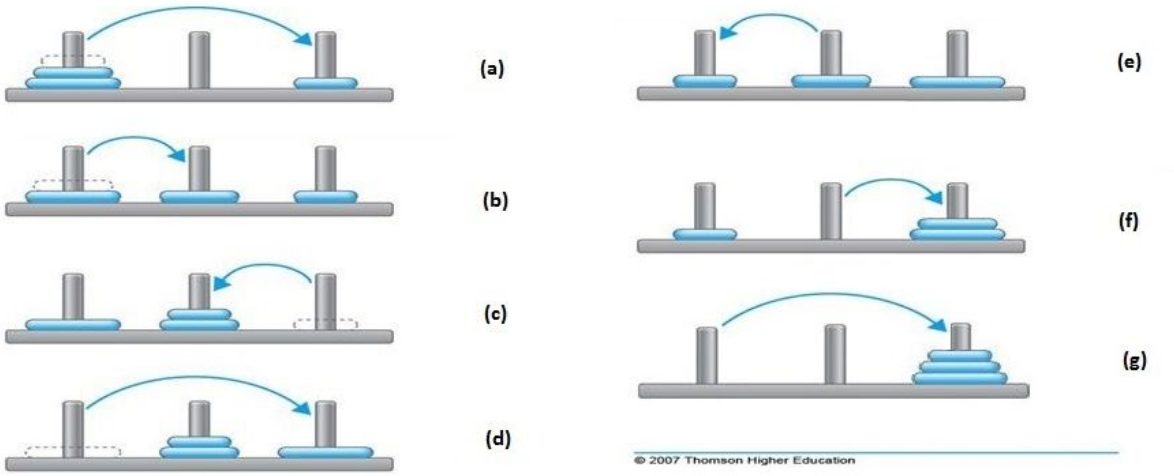


Figure 4 - Tower of Hanoi Solution Sequence

For Soar to solve this problem and others like it, the tasks must be represented within Soar's architectural framework. This means that the problem of reconfiguring discs on rods must be cast in terms of goal-oriented behavior in a *problem space*. To find a solution, Soar must transform its representation of the initial state of the discs into a representation of the desired state (goal state) of the discs. It transforms the state by applying an *operator* (a specific move, in this case) that changes some part of the current state to create a new current state. This process continues, operator by operator and current state to new current state, until the current

state becomes the desired state. When the desired state is reached the reconfiguration goal has been achieved and the Tower of Hanoi has been solved by *that* sequence of operators.

2.3.3.1. Problem Spaces

Problem space refers to the entire range of components that exist in the process of finding a solution to a problem, that is, the mental representation of a particular problem, including initial, final and all possible intermediate states (Figure 5). It is a way of organizing the system's knowledge and one of the foundational building blocks of Soar's theory. In general, Soar systems have their knowledge organized across multiple problems spaces and the relationships between the problem spaces are defined by *impasses.*

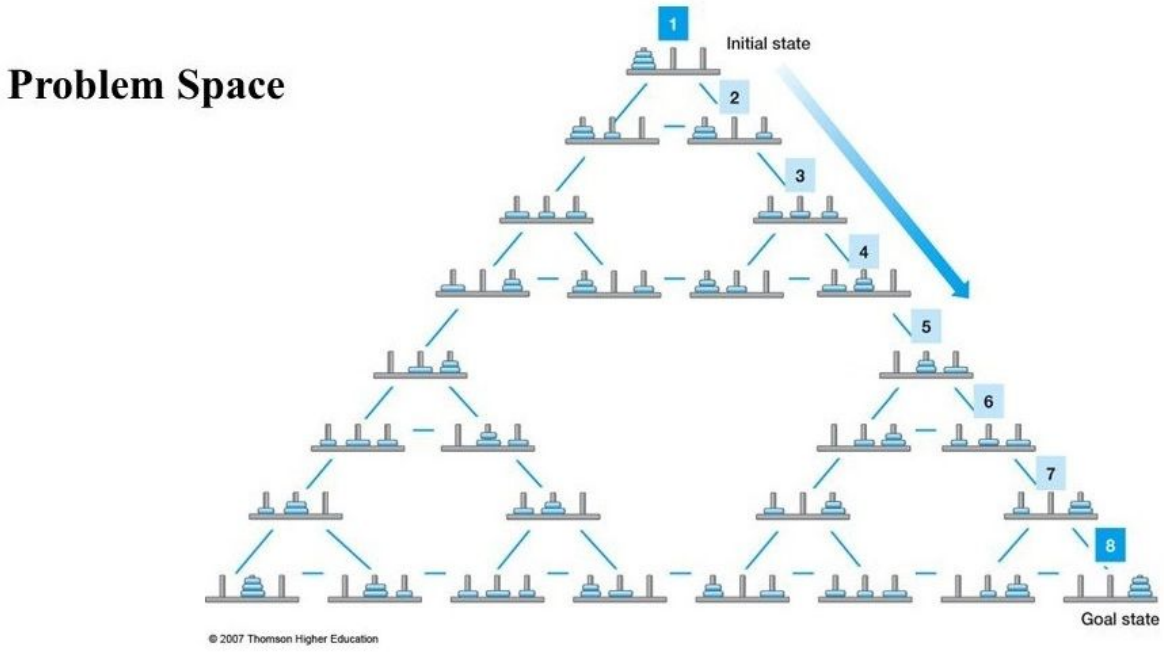


Figure 5 - A Problem Space with its solution path

2.3.3.2. Impasses

Soar begins to solve the disk reconfiguration problem in the *Switch* problem space, at the initial state [Figure 6]. *Switch* is just a term used to differentiate between the two problem spaces that

Soar employs - Switch & Selection problem spaces - in order to solve the problem. In Switch, two move operators can be applied to this configuration. The small disc can move either to the second rod, or the third rod. Although two move operators are possible, Soar may choose only one of them, but has currently no information as to which is the better one. This tie among operators is an *impasse* that arises because Soar doesn't know which operator leads, in the long run, to the desired reconfiguration. In general an impasse arises whenever the system lacks immediately available knowledge about how to proceed, so it responds with an impasse and setting itself the new goal of finding knowledge that can resolve that impasse.

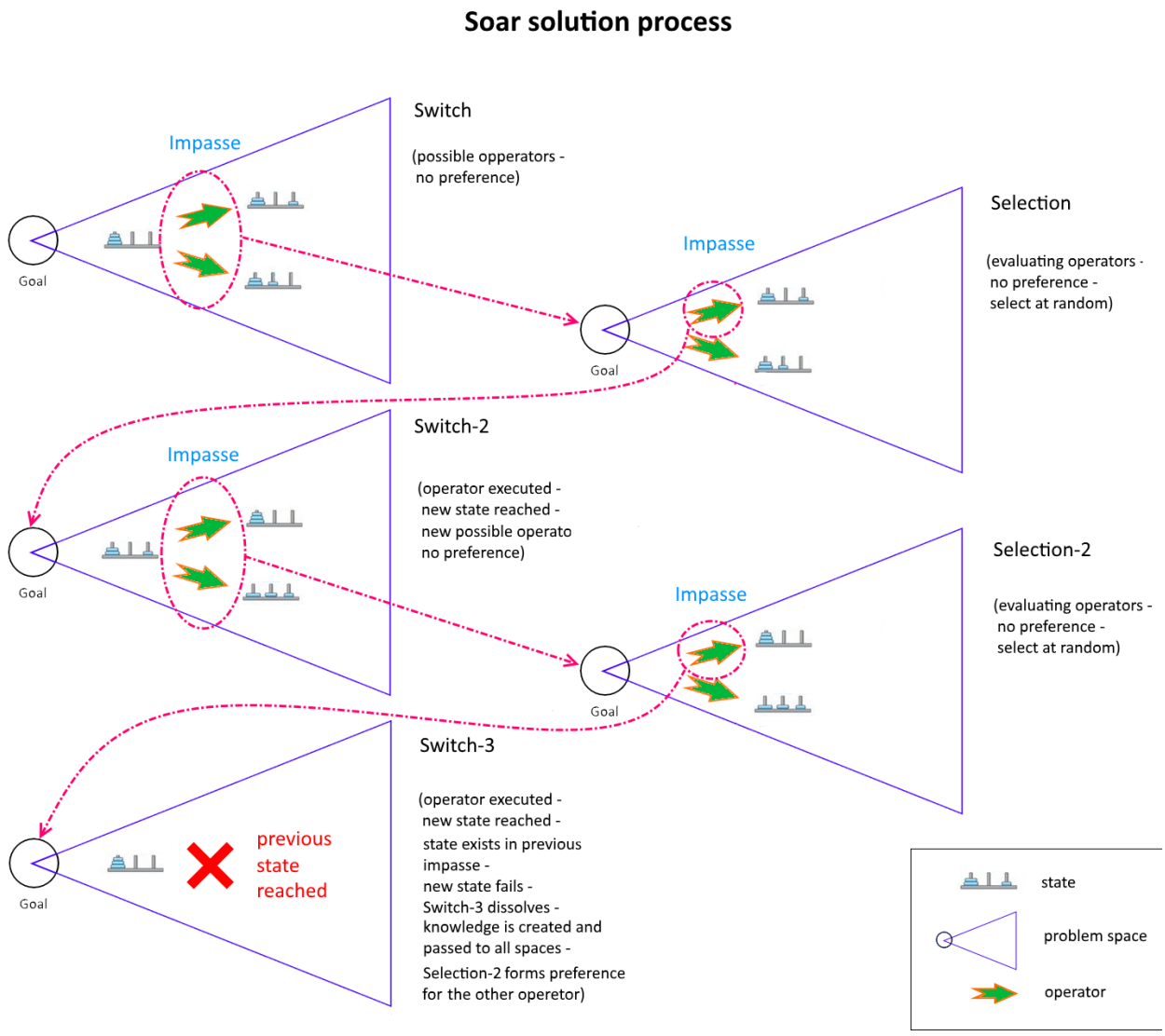


Figure 6 - Soar solution process example

Soar's knowledge is spread across two types of problem spaces, Switch and Selection. Switch spaces provide knowledge about how discs move, with states that represent disc configurations and operators the move discs from one configuration to the next. Selection spaces, on the other hand, provide knowledge about which paths are most likely to lead quickly to a goal state (Search Control Knowledge), with states that represent information about the tied moves in Switch space, and operators that evaluate those moves. Thus, Selection's initial state contains the two tied, unevaluated operators of Switch. A visual representation of this whole process can be seen in Figure 6.

Selection, then, chooses one of Switch's two operators at random and applies an evaluation operator to it in order to assess them. But because Selection doesn't yet know if this move will lead to success or failure, a new impasse arises and Soar crates for itself the new goal of attaining the knowledge to resolve it. Choosing a new instance of Switch (Switch-2) as the place to find that knowledge. In Switch-2, Soar applies the selected operator to the initial state and moves the small disk to the third rod. This new state is still not a desired state however, so Soar continues its look-ahead search.

Before we continue, it is important to note that Switch-2 is, in a sense, an imaginary state which Soar uses to explore different possibilities. Changes in this Switch-2 state do not actually occur to the initial Switch-1 state. The initial state will be changed once this internal thought process has concluded as to which operator is the optimal one as the first move.

Continuing, now, to the solution of the first impasse. Assuming, for simplicity, that only two operators can be applied to Switch-2's current state - moving the medium disk to the second rod or returning the small disc to the first rod - soar must again choose which path to follow. It does so by using the same impasse structure we have just seen. The tie impasse leads to a new Selection space (Selection-2) for choosing and evaluating each path, followed by a new instance of the Switch space (Switch-3) to try out the selected move.

If the random choice made in Selection-2 was to move the small disc back to the first rod, Soar notices that this choice returns to a state that already exists in an impasse problem space (in this case, it returns to the initial state). Returning to a previous state causes the current state to be marked as a failure. This evaluation serves to resolve the impasse in Selection-2 and the

resolution of this impasse pushes the look-ahead search down the only available alternate path. This gives us our first opportunity to look at how Soar learns.

2.3.3.2. Learning

Remember that an impasse is caused by a lack of knowledge and notice that when the impasse is resolved, knowledge has become available. This knowledge relates conditions before the impasse, to actions that occurred after the impasse. In this instance, Soar examines: 1) the previous configuration of the discs [Figure 5 - configuration state: 2], 2) the operator that has been applied to that state [move the small disc from rod 3 to rod 1] and 3) the current configuration of the discs [Figure 5 - configuration state: 1], in order to find an evaluation. In this case, because the selected operator backtracks to an unwanted previous state, the operator is evaluated as undesirable. Whenever an impasse is resolved, Soar's learning mechanism automatically stores away a pattern of conditions and their actions. This process is called *chunking* and once the pattern has been chunked, in the form of [previous state, current state, operator, evaluation of resulting state], its knowledge is available in every Selection space. So the next time Soar is in a similar situation in a Selection space and wants to evaluate an operator that moves a disc to a rod that represents the configuration of a past state, this chunk will fire, the evaluation will immediately appear and the prefered operator will be selected right away - thus no impasse will occur.

2.3.3.3. The Solution Path

We've just seen how Soar uses problem spaces, impasses and learning to explore a possible sequence of move operators which leads from the initial state to the desired state. The system continues to problem-solve in this way. Its search among sequences of operators is taking the form of a constantly changing hierarchy of alternating Switch and Selection spaces. Remember that each space has access to all the chunks created during previous impasse resolutions. Eventually Soar reaches the point at which the operators that have impassed in each of the Selection spaces have now been solved and each one corresponds to a move along the solution path.

Reaching the desired state in the final Switch problem space, starts a chain of impasse resolutions and chunk creations in order to bring the look-ahead search to a successful conclusion. As each impasse resolves chunks are added to Soar's long-term memory. These chunks capture the knowledge needed to prevent an impasse in analogous situations.

At last Soar has reached the original impasse in the hierarchy. This impasse occurred because Soar did not know which of the two operators that could be applied to the initial state would lead to the desired configuration. The look-ahead search has shown that the operator that moves the small disk to rod no. 3 will lead to a solution. Furthermore, as the original impasse resolves, a chunk is built that captures the preference for that operator, and the operator itself is applied to Switch's initial state to create the next state [state 2 of Figure 5].

Now, it might seem that after taking the first step down the solution path, Soar would have to repeat the whole look-ahead search to find the next step. But having found the whole solution path during the look-ahead means that, each operator on that path has been once applied and evaluated in response to an impasse that arose previously. Hence, there already exists a chunk that recognizes the state and selects the operator to make the state transition, without any new search. In other words, Soar now knows the intermediate states that exist along the path towards the goal state and the operators that lead to said states. From this point on, every time such an operator is between the available operators in a Switch space, Soar will automatically have a preference for it without having to call a Selection space in order to re-evaluate them. In fact, because it learned a pattern for the resolution of each impasse, Soar can now complete the reconfiguration sequence, step by step, to the desired state without further search.

In this elementary example, Soar solved the Tower of Hanoi puzzle using the Switch and Selection spaces to organize knowledge into a look-ahead search. Impasses in Switch spaces occurred due to lack of knowledge regarding which operators to select. Impasses in Selection spaces occurred due to lack of knowledge regarding how to evaluate candidate operators in Switch. These impasses imposed a hierarchical relationship between Switch and Selection which enabled chunking to transfer the knowledge found during look-ahead, toward the final solution sequence. Although they may use different problem spaces, operators and search control, other Soar systems work in much the same way when they lack direct knowledge about how to select and evaluate operators. They all rely in multiple problem spaces, impasses and learning, which characterizes the Soar architecture.

3. Building a Soar agent

Our journey will begin with the creation of an agent able to play a single round PD by selecting randomly between Cooperation and Defection. This basic form of the game will provide us with the a structure which we will improve on - the states, the proposal rules, the operators, the opponents and our own agent. After our elementary program is prepared and functional, reinforcement learning will be added to the agent in order for it to maximize the expected value of his moves' payoff. We expect that learning will lead the agent into aligning his operator selection preferences with those of the dominant PD strategy of always defecting. If everything goes to plan we will, eventually, upgrade our agent to be able to play the iterated version of prisoner's dilemma while still learning along the way. Hopefully it will be able to successfully handle all the different opponent strategies that we put against it as we inspect the empirical strategy it developed along the way.

For those interested getting into the programming aspect of this, the code for one for the agents coded for this thesis, is pasted in the Appendix. Specifically, the standalone-PD agent with reinforcement learning enabled, which is one of the more compact programs. The IPD agent is more than 800 lines of code, something no one would enjoy reading on paper, but the ***entire code*** for all the agents can be found, downloaded and fiddled with, on my GitHub[1] page.

3.1. The Building Blocks

As we have seen, Soar's architecture consists of *states* and *operators*. Operators act on the states and alter them progressively until the desired state is reached. Here, we will give the reader a glimpse of how those elements are represented within Soar's language and get an idea of Soar's internal logical structures.

[1] https://github.com/konstantinosftw/Soar-IPD-Thesis-code

3.1.1. States

States live in the working memory and are organized as graph structures. Those structures are made of *attributes* and *identifiers* [see Figure 7]. Below is a simple example of the working memory's structure for a state representing two blocks of different colors, one on top of the other and both on top of a table.

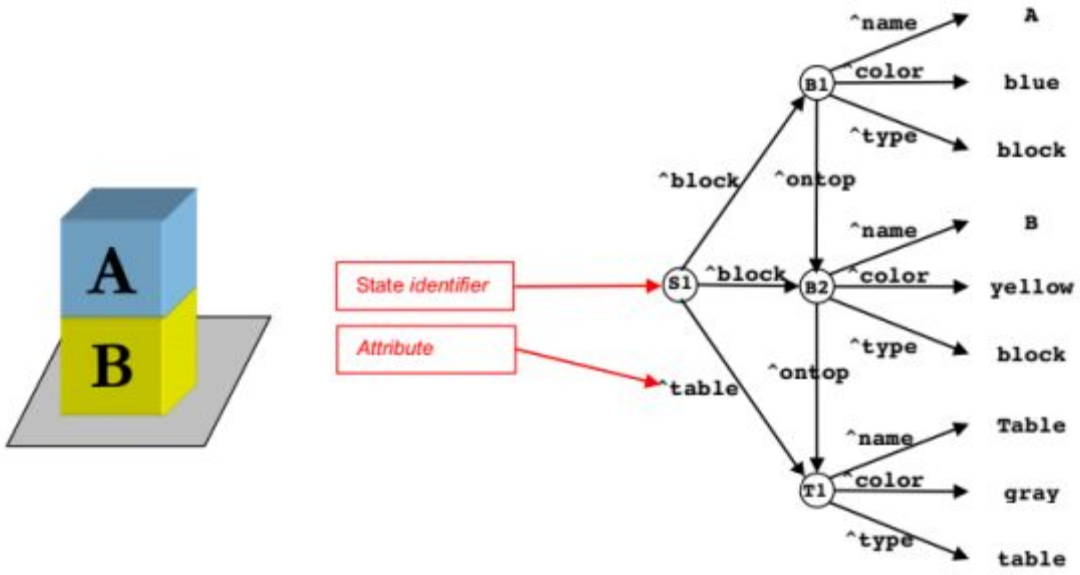


Figure 7 - A Soar state

In the example above, S1 is the identifier for the state, B1, B2 and T1 are the identifiers for the two blocks and the table, respectively. All of the identifier symbols are created automatically by Soar and consist of a single letter followed by a number. Attributes are prefaced by the "^" character, called a caret, and can point either towards an identifier, e.g. "B1", or towards a constant, e.g. "yellow" or "table". Notice the ^*ontop* attributes between B1, B2 and T1. That attribute creates the spatial relation between the structures - B1 on top of B2, B2 on top of T1. Of course, Soar does not know what ^*ontop* means, as it also does not know what a *table* is, or the color *yellow*. These variables mean something only to us and we could have named them however it made sense to us. Soar, as any programming language, is only interested with the logical relationships of those attributes and how they can be manipulated when applying operators on them.

3.1.2. Operators

Operators perform actions, either in the world or internally in the "mind" of an agent. In our Tower of Hanoi example, the operators were moving discs between rods. In the Blocks example above, we could, for example, write an operator that changed the color of a block, or one that moves a block from the bottom to the top. Because operators perform actions, they are the locus of decision making. They are the place where knowledge is used in order to decide what to do (or not to do). Thus, Soar's basic operation is a cycle in which operators are continually proposed, selected, and applied.

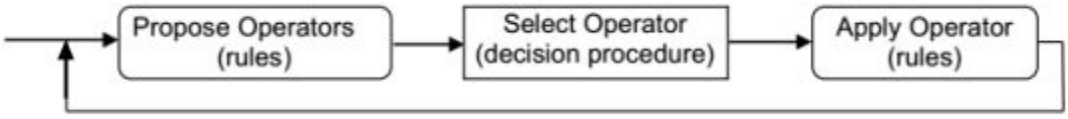


Figure 8 - The operator process

When a fresh state appears and a new decision cycle starts, Soar examines the state and proposes the relevant operators to the working memory. Proposal rules contain "if-then" conditions that test features of the state to ensure that each selected operator is appropriate. For example, when all the disks are on the leftmost rod in the initial state of the Tower of Hanoi, no operator that moved the medium or the large disk could be proposed as it would be against the rules of the game (which are encoded in the Soar program). When the proposal rules fire, a representation of each proposed operator is created in the working memory along with an *acceptable preference* for that operator. This preference, that we will see in more detail during the learning process, is a way of telling the selection process which operator is more useful for the current situation. After the proposal of all the appropriate operators the selection process comes along, makes a decision (taking into consideration the operators' acceptable preferences, among other things) and passes the chosen operator to the application process. The application of the operator takes place and the appropriate changes, that the operator dictates, are made on the current state, giving rise to a new current state. In the Blocks

example above, an operator that would take block A and move it below the table, would consist of two rules. A rule that removes B1's attribute ^*ontop B2* and a rule that adds to the table T1 an attribute ^*ontop B1*.

3.2. Single-PD agent

In order to build an AI agent that plays multiple games of IPD, we need to start by building a simple agent that plays a single round of the prisoner's dilemma game. At this point, both the agent and the opponent will select their moves randomly, although each does it through a different process. The opponent is a part of the deterministic environment, in the sense that it always follows a pre-coded strategy and is not a decision making agent. It is bound to move, and react to moves, in a pre-programmed way, i.e. always defect, always cooperate or choose randomly between the two. Our Soar agent, on the other hand, does not have any pre-coded behaviour like that. In fact, the agent does not even know how the game is played, merely that it can either select a move named "cooperate" or one named "defect", not knowing what each does or how much each pays off. It has to learn all this information empirically by repeatedly playing the game. This way we can observe the agent's learning process from scratch, allowing him access to the least possible information.

Back to the selection process, the opponent reaches its random move by a randomized algorithm that outputs an integer of either 0 or 1. If the result is 0, it defects and if result is 1, it cooperates. In contrast, our Soar agent has a different approach to his, eventually, random decision. When faced with the decision, the agent proposes both the cooperate and the defect operators to the selection process, as they are both valid candidates for selection. Due to the fact that there exists no prior knowledge at this point, the operators are proposed without any preference and thus they are deemed equal. In turn, the selection process, being unable to figure out which operator is the better one, reaches an impasse. But Soar, realizing that it lacks any information that could eventually lead to a solution (which would drive the program in an infinite look-ahead search) forces the selection process to make a decision at random and the selected operator gets applied on the application phase. Therefore, although at this point the agent is no better in playing the PD game than its random-playing opponent, its moves have a fundamental difference - they are the product of decision making. Total lack of information

results to its decisions being analogous to “I have no clue, let’s flip a coin to decide”, but this will soon change once learning is enabled.

The program runs successfully and after each game the score is calculated and printed, along with the winner of each game. The output looks like Figure 9 - first the agent is initialized and the scores are both set to zero, then we print the operator that got selected by the agent (in this case *O2* - a cooperation) while the opponent selects its random move. On the next line, the round’s number is printed, which will become useful on the iterated version as the single PD game consists of only one round, followed by both the agent’s move and the opponent’s move, in that order (here, “coop-coop”). Eventually, we print the winner of the game followed by the exact score for each of the players and the Soar program halts. Note that, for our program we are using the payoff Table 2 from chapter one - 5 points as the Temptation payoff, 3 points as the Reward payoff, 1 point as the Punishment payoff and 0 points as the Suckers payoff.

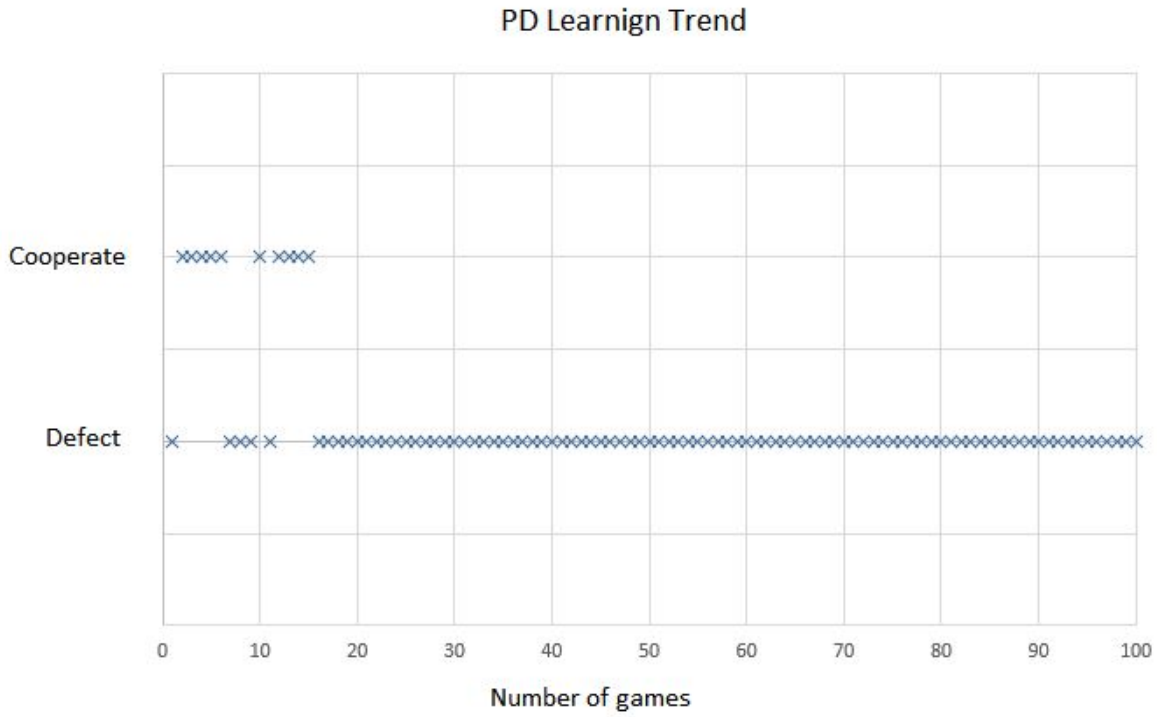
```
Agent reinitialized.
run
     1: O: O1 (initialize-prisoners-dilemma)
Agent: 0 / Opponent: 0
     2: O: O2 (coop)
Round 1: coop-coop
--- THE GAME IS DRAW ---
Agent: 3 / Opponent: 3
Interrupt received.
This Agent halted.
```

Figure 9 - Soar’s output

Now that our program is able execute moves and calculate payoffs correctly, it’s time to input these payoffs as rewards to the agent in order for it to evaluate its behavior. We will get into the details of the learning algorithm on the following section, since it needs a subchapter of its own. Currently, it is enough to understand the learning process qualitatively, namely, that the agent gets a reward analogous to the payoff of its move. The sum of those rewards for each move are then passed as *preference* for that move’s operator, at the operator proposal phase - which are taken into account during the operator selection phase. So, the operator that generates the highest average rewards will get selected with a higher preference. The game-theoretical solution to the PD shows that defection is the dominant strategy, since it has

the highest expected value. Ergo, we would expect our agent to start developing a preference towards defection as games go by.

Let's finally run the agent.



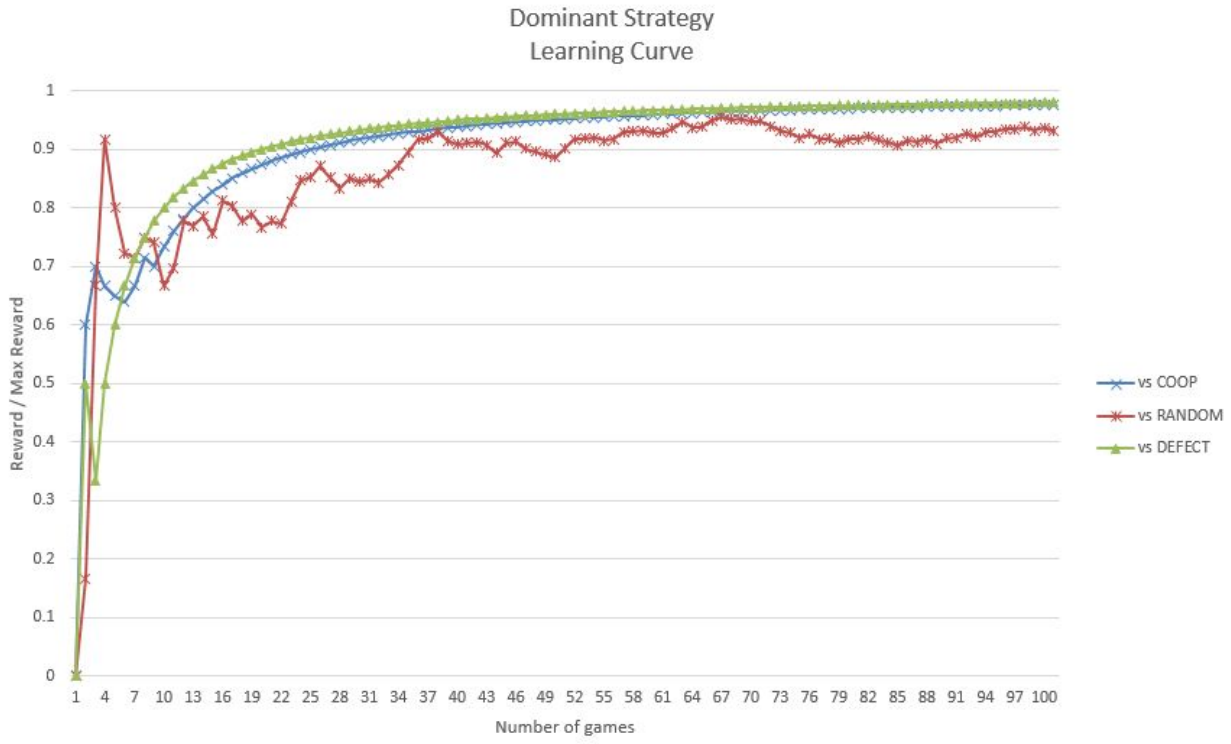
Graph 1 - PD Learning Trend

Indeed, the Soar agent very soon started to realize that defecting has a higher aggregate payoff than cooperating. Figure 1 is a sample of 100 games against the random opponent (RANDOM). The same behaviour was developed against the always-defect (DEFECT) and always-coop (COOP) opponents. In fact, learning was faster against the two persistent strategies - especially against the defector - as those yielded more consistent payoffs, solely dependant on the agent's moves. On the above series of 100 games, for example, the agent started with defection, but RANDOM also defected on that first round, rewarding our agent with only a single point. On the next move the agent cooperated and so did RANDOM, giving a higher, three point reward. The agent saw that cooperation pays better, so it kept cooperating until RANDOM betrayed our agent twice in a row, at which point our agent tried defecting again -

and so on. Eventually, after sixteen moves, the agent had gone through all possible payoffs and found that defecting pays out better.

Against the persistent DEFECT, on the other hand, our agent earned zero points when it first cooperated and one point on its first defection. Consequently, in only a few rounds it learned that is should continuously defect, chasing the sole reward. For the same reason, the learning rate against COOP was faster then RANDOM as well, still not as fast as against DEFECT due to the fact that the agent was gaining a minimum of 3 reward points for its cooperations - enough to provide some encouragement for selecting that move again.

The agent's learning curves towards heuristically determining the dominant strategy, against each of the three opponents, is plotted on Graph 2 below. We see that the strategy stabilizes faster vs DEFECT, followed closely by that vs COOP and lastly by that vs RANDOM.



Graph 2 - Soar agent's learning curve towards PD's dominant strategy

The horizontal axis represents the number of games and the vertical axis represents the *effectiveness* of each move, described by the this equation:

$$f(x) = \frac{\sum_{n=0}^{x} Reward(x)}{x * MaxReward}$$

The sum of all the reward points that the agent had received towards each step, divided by the maximum points it would have received up to that point, if it was playing the dominant strategy all along.

3.3. Reinforcement Learning

On chapter two, while introducing the inner workings of Soar, we went through chunking, Soar's "signature" learning mechanism. Chunking is a great way for summarizing sub-goal results in a way that prevents the need of rediscovering the best possible path towards a goal state on every step of the way. After a chunk is created, Soar knows a shorthand of achieving the desired state with a minimal number of moves. E.g. moving the whole Tower of Hanoi from the left rod to the right rod, thus solving the game, can happen in a minimum of seven moves but it can also happen in eight, nine or a hundred moves. Chunking will make sure that, after discovering the path to the goal state as a result of a long number of random permutations, the next time Soar tries to solve the same game, it will follow the shortest path to the solution instead of randomly wandering through the whole of the problem space again. Though, what is the goal state of the Prisoner's Dilemma?

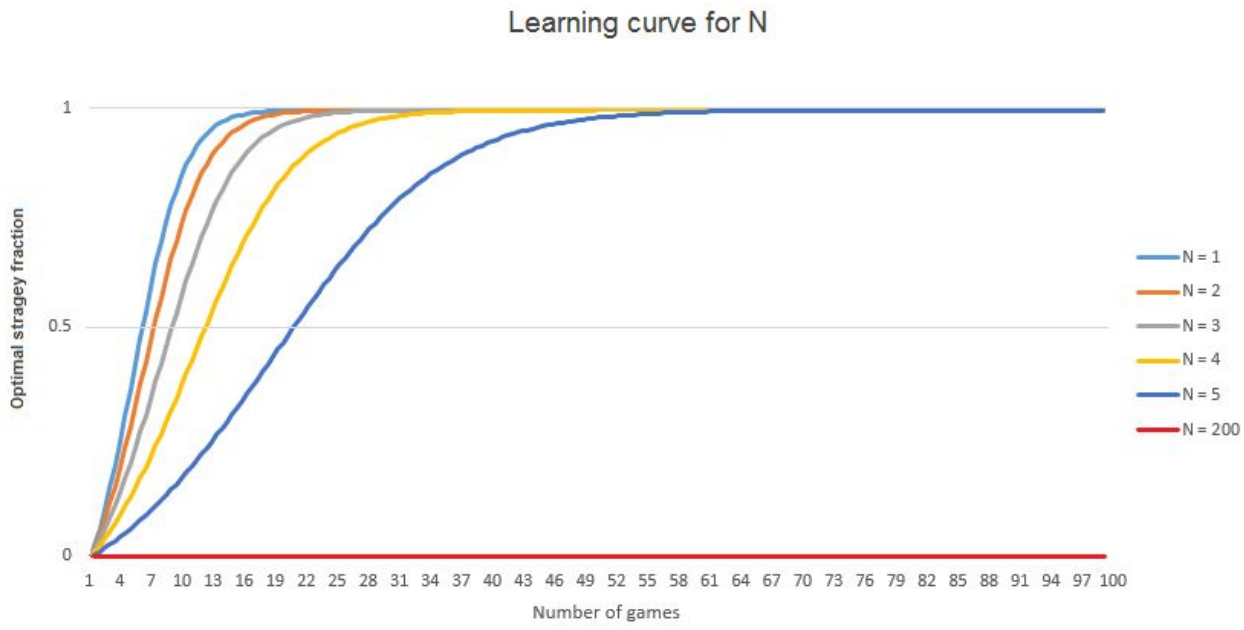
In multiple opponent games, unlike in single player puzzles, winning in not simply determined by discovering the path to the solution state, but outthinking the opponent. Forming a rigid memory (chunk) of how an action led to a certain result, in a dynamic environment, will result in creating a permanent, suboptimal rule. The agent, recalling that chunk, will constantly repeat the same move that paid out well the first time, instead of reevaluating the patterns of its opponent's moves - something that can hurt the performance of the agent very fast. Instead of a process that creates inner rules and retrieves actions, we need to use one that creates rewards and retrieves numeric preference of actions. That, is the function of Soar's second learning mechanism, reinforcement learning.

Reinforcement learning (henceforth RL) in Soar allows agents to alter behavior over time by dynamically changing numerical preferences in procedural memory, in response to a reward signal. The reinforcement learning agent has to decide how to act to perform its task by collecting training examples ("this action was good, that action was bad"), through trial-and-error, as it attempts its task with the goal of maximizing long-term reward. This is how our PD agent discovered the dominant strategy whilst playing the game. Its RL mechanism created a memory of the payoff (reward) which the agent received at the end of each game, associated with the move that the agent played. This provided a numeric preference for that specific move on each subsequent game.

Nevertheless, as we wish to move forward towards the iterated version of the prisoner's dilemma, we are presented with a problem. If we reward the agent after each single iteration of the IPD game with a reward relative to the payoff it just received on that iteration, the agent will try to maximize exactly this - the payoff of every single round. This will drive the agent towards persistent defection, as every round will essentially be a standalone game of PD. What we need to do, is to make the agent realize that it needs to maximize the *sum* of the payoffs it will receive throughout the entire 200-round game.

This seems as an easy fix, we can simply reward the agent at the end of the whole game instead of each individual round. This way, if the agent constantly defects for 200 rounds, let's say against Tit for Tat, it will receive 204 points as a payoff (and as a reward), while, if it constantly cooperates it will receive a reward of 600. What will happen, though, if we put this agent against a slightly more aggressive opponent, for example one that interchanges cooperation and defection on each round? Not only will the agent need to complete a whole game of 200 rounds before realizing that it received a terrible reward but, even worse, it would not even be able to understand which of its moves should be changed in order to achieve a better reward. It is like providing a student with the bare grade of his 200-question True/False exam, without disclosing which of his answers were correct and which were erroneous. Imagine, how many times would that student need to retake the exam, cross-checking along the way, in order to find out with certainty which of his answers are correct and which aren't. The amount of possible True/False, or Coop/Defect, permutations grow exponentially with each question, or round, being added. An IPD game of 3 rounds against the interchanging opponent would be solved optimally after a maximum of 9 games. But for an IPD game of 200 rounds,

the agent would need to play a maximum number of games sixty orders of magnitude higher - a computationally absurd number. And that would be for each single opponent strategy. In Graph 3 below, we can develop a sense of how, delaying the rewards for *N* iterations, can affect the search for an optimal strategy.



Graph 3 - Learning curves for different number of iterations (N)

What we need is a way of rewarding the agent after each single move so that we maximize its learning speed and make it quick to respond to changes in its opponent's strategy while, at the same time, makes the agent sensitive to its possible future rewards, so that it can move towards a long-term payoff maximization. Thankfully, there is an RL algorithm that precisely fits our needs.

3.3.1. Q-Learning

Q-learning is a reinforcement learning technique that Soar can incorporate in its RL mechanism, making it able to compare the *expected* utility of the available actions. The way

Q-learning works is that it calculates, not only the immediate reward of a specific action applied on certain state, but also the expected value of all future rewards that this action could lead to. This blend, of present and future rewards, is called the *Q function* of a state-action pair and is symbolized as:

$$Q(s_t, a_t)$$

The function ***Q*** of an action ***a*** acting on a state ***s***, both during time ***t***. And the calculation of that function happens as follows:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

At each step, we update the Q function of a certain state-action pair, by increasing or decreasing the old Q-value by the difference it has with the new Q-value (multiplied by a learning rate ***α***).

To make this more readable, let's go through the equation from left to right.

1. a is the *learning rate*, which is a number between 0 and 1. The higher the the learning rate, the more aggressive the learning process. With a learning rate of 1, the agent will be totally replacing its old Q-value with the new one, after every move. With a lower learning rate the agent will take into account the reward of its latest move (a fraction of it), but within the scope if a bigger number of moves - averaging the rewards *that* action on *this* state. The default learning rate is 0.3.
2. The *reward* r_t represents the reward value that the agent just received by executing action a_t on state s_t.
3. The *discount factor γ* is a number between 0 and 1, which represents the weight that we assign to future rewards, compared to the present reward. If the discount factor is 0, the agent does not take into account any future rewards its move can have, while selecting an action - only the immediate reward. A discount factor of 1 means that the future reward is going to be taken into account exactly as much as the current reward. The default *γ* is 0.9. We will set our discount factor to 1 because we want our agent to

be able to be able to even make short term sacrifices in order to achieve long term goals. For example if the players gets stuck in a defect-defect loop, our agent should be able to sacrifice 1 point on the current round and cooperate once, for the possibility of a future 3 point reward.

4. The *estimate of optimal future value* is exactly that, the Q-value of the best possible action that the agent will be able to make on the next round (s_{t+1} represents the next state).
5. Finally, we subtract the old value of Q to make sure that we're only increasing or decreasing by the *difference* in the estimate (multiplied by alpha of course).

This dynamic way of updating the utility value for each action, directly after each action while at the same time taking into account the possible future rewards the agent might receive, is exactly what we were hoping for.

3.3.2. Exploration vs Exploitation

An additional difficulty that one is faced when dealing with RL (and many times when dealing with real life), is the *exploration vs exploitation* problem. Imagine, our learning agent against an opponent where they mutually defected on the first round, which just granted our agent a 1-point reward for his defection. What keeps our agent from constantly defecting from this point onward, exploiting the single point reward it receives again and again, not exploring any other possibilities that could eventually prove more profitable? Or imagine the opposite scenario. Out agent plays against persistent COOP opponent and, although it receives a 5 point payoff everytime it defects, the agent keeps exploring different combinations of cooperations and defections in a constant effort to receive a possibly higher reward. At which point, then, should the agent stop using its time and resources in order to search for new, possibly more efficient, ways to solve a problem (Explore) and should start capitalizing on the best methods it already knows (Exploit)?

This brings up the *exploration/exploitation* tradeoff. One simple strategy for exploitation would be for the agent to take the best known action most of the time (say, 80% of the time), but

occasionally explore a new, randomly selected direction even though it might be walking away from known reward. This strategy is called the *epsilon-greedy* strategy, where *epsilon* (symbolized with ε) is a 0 to 1 probability that the agent will prefer a random action, over the best known action (in this case, 20% of the time). We usually start with a lot of exploration (i.e. a higher value for epsilon). Over time, as the agent learns more about the game and which actions yield the most long-term reward, it would make sense to steadily reduce epsilon to 5% or even lower, as it settles into exploiting the best moves it has learned so far. The epsilon-greedy approach is intuitive, efficient and it can be readily applied in Soar, so this is exactly what we will use to handle the explore/exploit problem.

3.4. IPD agent

At this point we have managed to build a PD agent that can play a series of standalone PD games and, through reinforcement learning, manage to work out the dominant strategy after just a few games. By exploring the Q-learning algorithm, we have also managed to solve the problem of reward intervals - rewarding after each move or at the end of each game. It is time now to modify out PD agent so that it can handle multiple iterations, and code, in Soar's language, all the opponent strategies that we specified on chapter 1.4.1.

There is one more ingredient we need to add to the agent, and that is the ability to remember, and connect, the previous moves of both the its opponent and itself. If we deploy the RL mechanism solely monitoring the agent's current move, like we did in the standalone PD game, the agent will always converge towards a persistent strategy, either constant cooperation or constant defection. It will have no way of discerning how its actions impact the opponent's *next* action and its learning rules will be analogous to "cooperate = good / defect = bad". We need the agent to have the capacity for a more complicated thought process, such as "whenever I cooperate, my opponent tends to also cooperate on the next round". For this reason, we need to make the agent able to look back into the moves' history. Obviously, the further back the agent can look at, the more complicated the patterns it will be able to detect and the more refined its strategy will be. But this can run again on the same tradeoff of learning speed versus precision. As the number of rounds that the agent can look back into tend to increase, the learning curve becomes flatter (and the learning rate decreases) exponentially fast. Considering

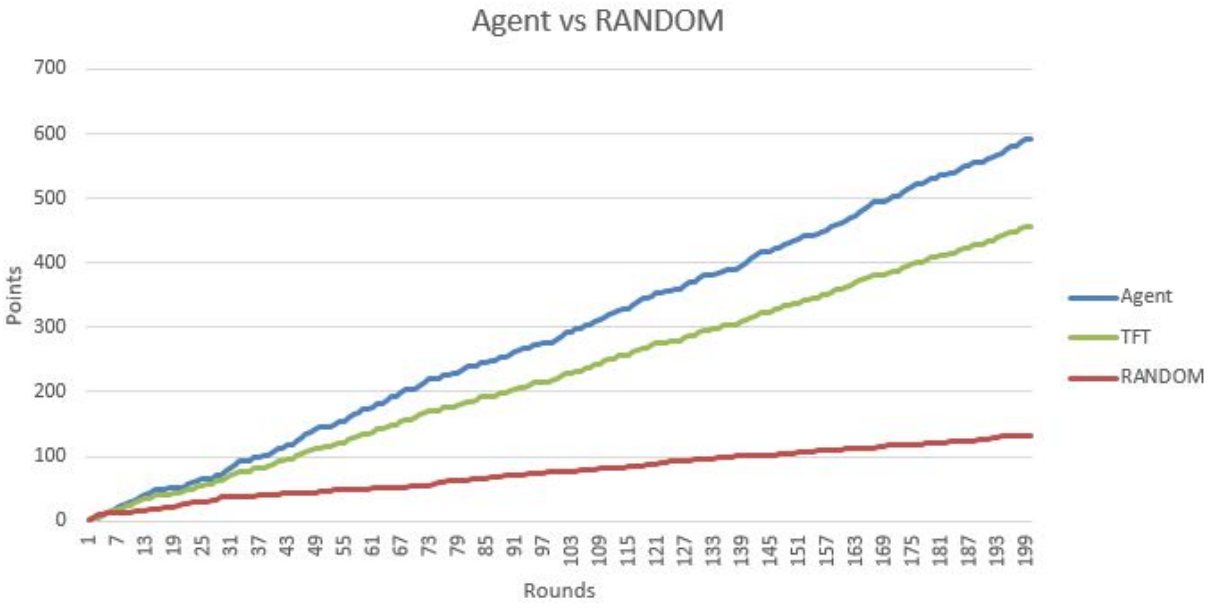
that Axelrod's original tournaments have lead to a belief that simple strategies do just as well, if not better, than more complex ones, and that the winning strategy had a look-back of one round, we will assume that programming our agent look back to only the previous round, will suffice. After all, our agent has the uneven advantage of actively learning throughout the game.

As mentioned on chapter 2, if the number of PD iterations in an IPD game is known, then the last iteration becomes a standalone PD game and the agent is motivated to defect. For this reason, our agent is not given any learning tools that might have it figuring out that the 200th iteration is always the last one. It will, therefore, try to maximize its payoffs for an infinite horizon of rounds, just like the rest of the strategies.

Let's have a look at a detailed example of our agent against a simple strategy, before we match it against the full stack of all 12 opponent strategies.

3.4.1. Agent vs Random

Putting our Soar agent against RANDOM, we immediately see that the it can consistently beat RANDOM by a huge margin, starting from the first game. The agent averages close to 600 points on each game, while RANDOM averages around 100 points, as shown in Graph 4.



Graph 4 - Agent vs RANDOM

We have added the average score of TFT against RANDOM in the graph for measure. TFT's strategy has not advantage over RANDOM, but no disadvantage either, as they both tie on an average of 450 points on each game (RANDOM's score against TFT is not placed on the graph). Our agent on the other hand manages to get an edge over RANDOM and that is because it plays constant defection against it, as we see on Figure 10.

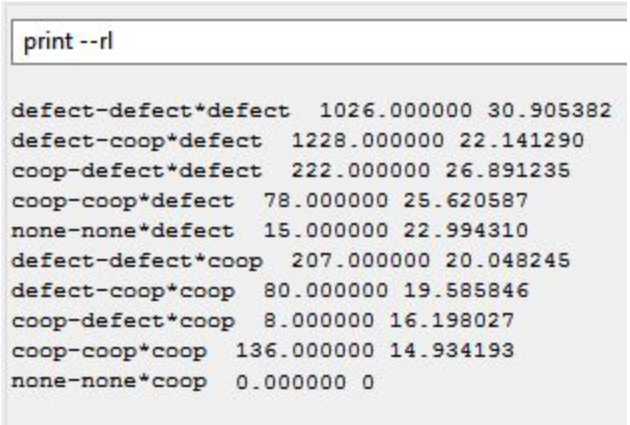


Figure 10 - Agent's Q-values vs RANDOM

The above image is the printout of Soar's debugger console, which keeps track of all the states, the operators, their preference and everything related to the agent in general. In this instance we requested the Q-values with the command *print --rl*. On the first column, the table contains the pair of moves that the "agent-opponent" pair played on the previous round, followed by an asterisk (*) and the agent's current move. Next column is the number of times, this action was selected and the last number in the row is the Q-value of that action. For example, the first line writes: the agent defecting, after a mutual defection on the previous round, happened 1026 times on this sample of 15 games, i.e. 3000 rounds, and this move has a Q-value of 30.905382. Contrast to this, is the *defect-defect*coop* line, which shows how many times the agent choswe to cooperate after a mutual defection, which is only 207 times with a Q-value of 20.048245. Examining the printout of the agent's Q-values shows us that it prefers playing defection over cooperation, on any given occasion.

TFT on the other hand, after a defect-coop round, reacts with a cooperation, having a 50/50 chance of receiving a 0 or 3 point reward on the current round. This is the reason our agent

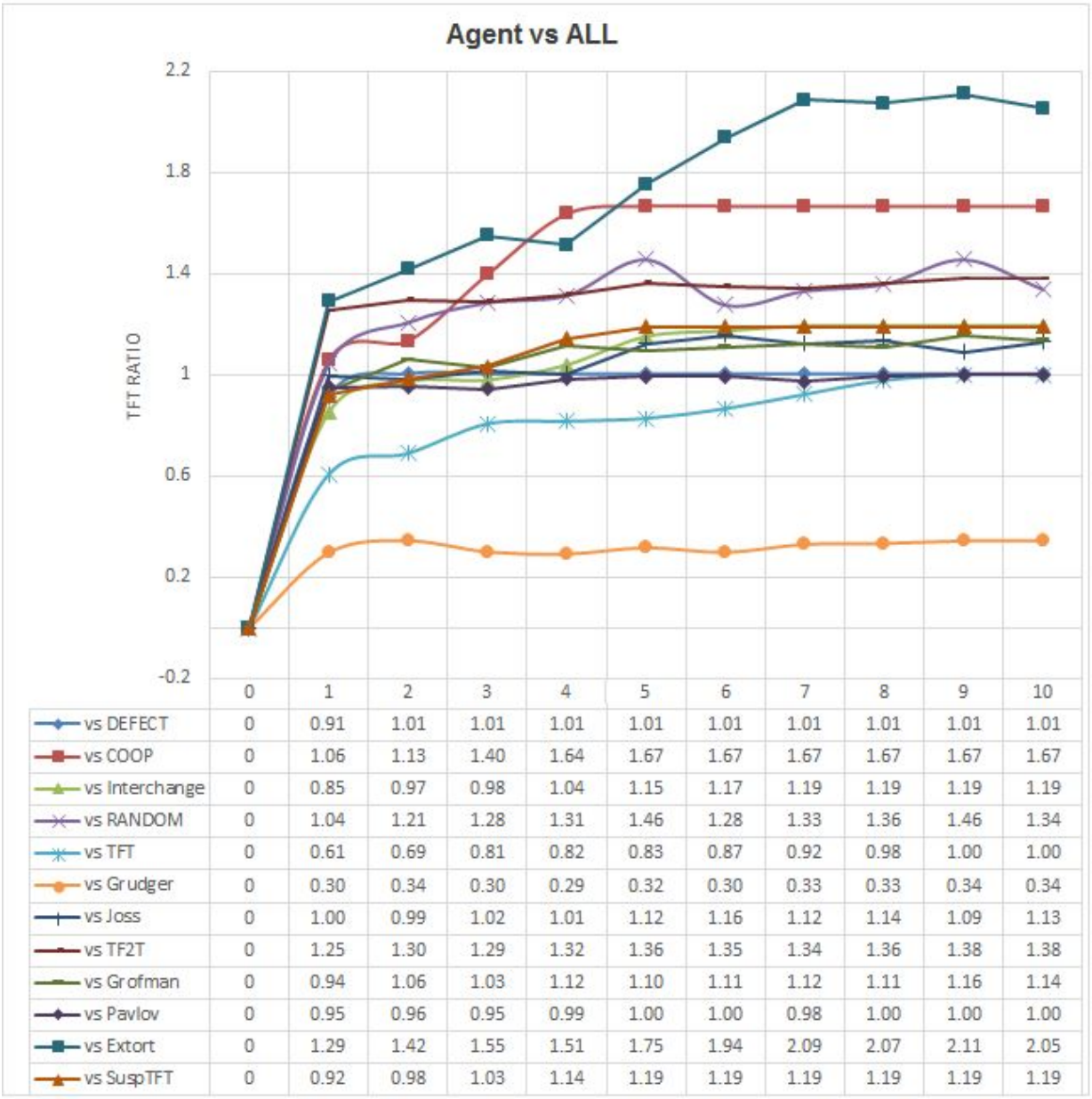
was able to manage the RANDOM opponent better than TFT did, because it learned to defect even after the opponent's cooperations. Due to the fact that random plays are each, practically, a standalone PD game, PD's dominant strategy is the best possible plan of action.

3.4.2. Agent vs ALL

3.4.2.1. Trained

The A.I. agent was fairly successful against RANDOM, even topping TFT's performance. It was intelligent enough to figure out how to best deal with this specific opponent and exploited that knowledge. Nevertheless, RANDOM is not a very sophisticated strategy, after all it was the least successful strategy on Axelrod's tournament. The fact, though, that the agent was not simply able to beat it but outperform, against it, the champion strategy, is a promising first step.

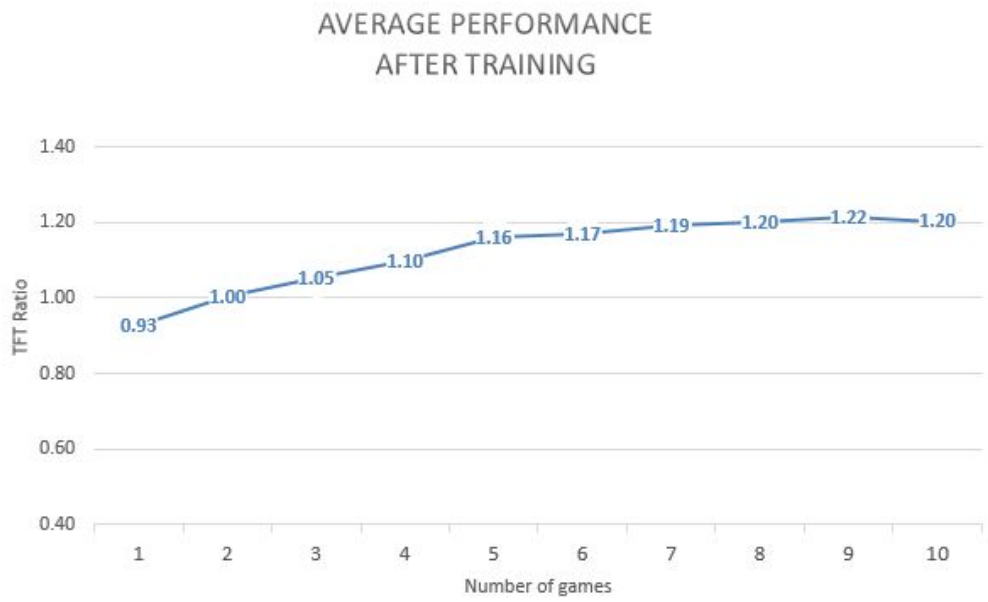
At last, the time has come to run our agent against all 12 strategies, one by one, for 10 games each, and see how its A.I. fairs against them. Our benchmark will be the best known strategy, TFT. Each result will be divided by the amount of points that TFT was able to earn against that specific opponent and plotted on Graph 5, below. For example, our score against RANDOM would be 1.33 (600 divided by 450), namely, surpassing TFT's performance by 33%.



	0	1	2	3	4	5	6	7	8	9	10
vs DEFECT	0	0.91	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01
vs COOP	0	1.06	1.13	1.40	1.64	1.67	1.67	1.67	1.67	1.67	1.67
vs Interchange	0	0.85	0.97	0.98	1.04	1.15	1.17	1.19	1.19	1.19	1.19
vs RANDOM	0	1.04	1.21	1.28	1.31	1.46	1.28	1.33	1.36	1.46	1.34
vs TFT	0	0.61	0.69	0.81	0.82	0.83	0.87	0.92	0.98	1.00	1.00
vs Grudger	0	0.30	0.34	0.30	0.29	0.32	0.30	0.33	0.33	0.34	0.34
vs Joss	0	1.00	0.99	1.02	1.01	1.12	1.16	1.12	1.14	1.09	1.13
vs TF2T	0	1.25	1.30	1.29	1.32	1.36	1.35	1.34	1.36	1.38	1.38
vs Grofman	0	0.94	1.06	1.03	1.12	1.10	1.11	1.12	1.11	1.16	1.14
vs Pavlov	0	0.95	0.96	0.95	0.99	1.00	1.00	0.98	1.00	1.00	1.00
vs Extort	0	1.29	1.42	1.55	1.51	1.75	1.94	2.09	2.07	2.11	2.05
vs SuspTFT	0	0.92	0.98	1.03	1.14	1.19	1.19	1.19	1.19	1.19	1.19

Graph 5 - Agent vs all 12 strategies

After 10 games, the agent was able to do at least as good as TFT against all, but one, strategies - with an average payoff of 1.20 between all 12 strategies, on the 10th game [Graph 6]. Outperforming the best IPD strategy by such margin may seem surprising, yet keep in mind that we are using machine learning to compete against deterministic strategies. This gives our agent the ability to adapt to its opponents, instead of following a blueprint of “if-then” rules.



Graph 6 - Average Trained Performance of the Agent

Tit for Tat, although the fittest strategy, has a distinctive counterintuitive characteristic - it can never win over an opponent on a one-to-one game. The best possible outcome for TFT is a draw, specifically one after a chain of 200 cooperations. This is the highest score TFT can achieve on a single game, 600 points, since it will never be the one to betray its opponent, first. However, when this strategy is averaged out against multiple opponents it yields very high results, especially in an environment that consists of more *nice* strategies. Our agent, on the other hand, can either be nice, like TFT, or take advantage of a naive opponent which cooperates unconditionally, extorting an uneven number points. Let's analyze some of the games from chart 5 to explore how this happens in detail.

Starting from one of the most successful plays, the one against **Cooperate**. TFT received 600 points against it as they both cooperated from start to finish. The agent, on the other hand, after realizing that defecting against **Cooperate** had no drawback, switched to constant defection, receiving the highest possible payoff of 1000 points. Similarly, against **TF2T**, the agent was able to figure out that defecting, but only once at a time, yielded a higher total payoff and so it started interchanging cooperations and defections, in a way that kept naive **TF2T**

constantly cooperating. Against **Random** and **Interchange**, while TFT was trapped in a loop of eternally mismatching coops and defects, the agent played pure defection and maximized its payoff. In fact, the agent received the theoretical maximal payoff, that each specific opponent could offer, from most of the opponents it faced. Intelligently crafting a tailor made strategy against them, including its game against Tit for Tat itself.

The only opponent the agent could not handle better than TFT, and in fact barely handle at all, was the **Grudger**. While TFT consistently received the optimal 600 points payoff against the Grudger, our agent never managed to get much more than 200 points. The Grudger's strategy simply seemed impossible for the agent's intelligence to grasp. This is due to the fact that this is the most agressively retaliating strategy in the game - after only a single defection by its opponent, the Grudger will "flip the switch" and go into a persistent defect mode. Considering that RL needs, by nature, some exploration in order to evaluate possible moves before it starts exploiting them, the agent stood no chance. In most of the games, our agent had tried out its first defection before even the third round, spiraling onto an unforgiving defection sequence all the way to the last round. Trying to cooperate once that line was crossed, never payed out. Eventually, defecting accumulated so many reward points for the agent that, soon, its prefered strategy became persistent defection, from the first round. The only opponents that do well against Grudger are the ones that can show blind trust from start to finish.

An other thing to notice is that, those high payoffs that overwhelm TFT's scores, are a product of training against each opponent for 10 rounds. As we notice on Graph 5, the payoffs climb, game after game, before reaching their maximum. Some opponents are easier to "read", which permitted the agent to reach the optimal strategy against them only after a couple of games, while other strategies need 7 or 8 rounds to be optimally solved. It would be interesting to see how would the agent fare against each strategy, without any prior knowledge of their behaviour.

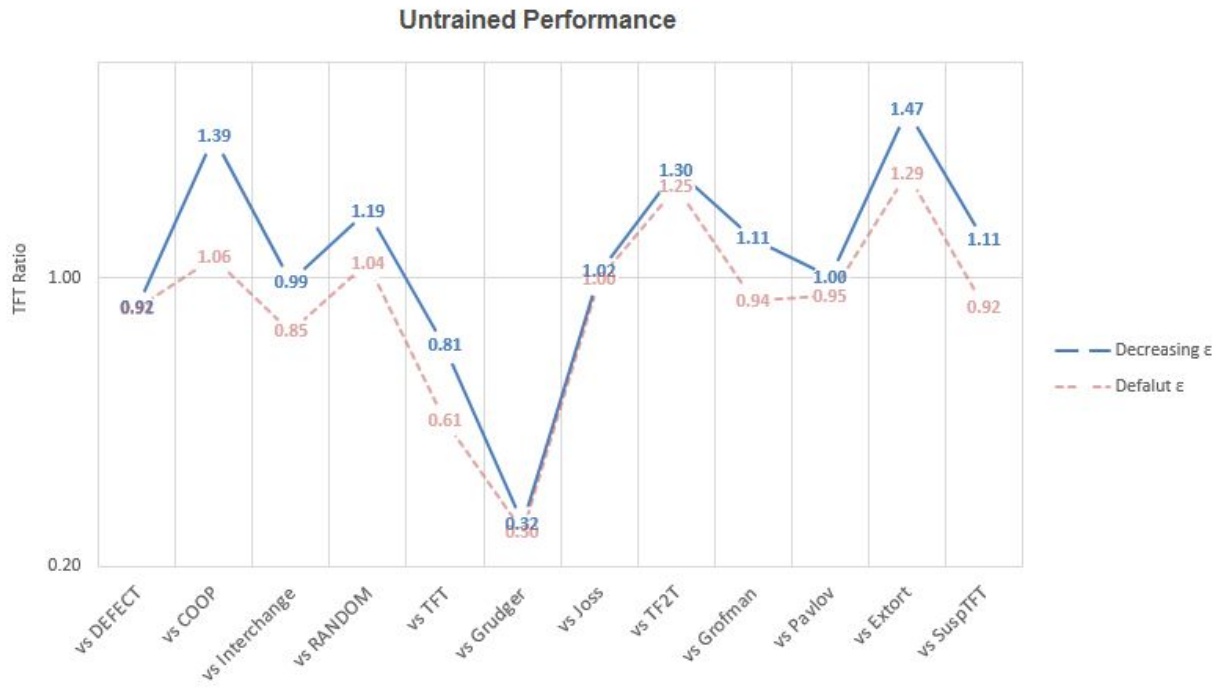
3.4.2.2. Untrained

It seems only fair to disallow our agent from playing multiple games with each strategy in order for it to train against them. After all, none of the programmers that sent their strategy to Axelrod's tournament had any prior knowledge towards the rest of the other strategies being

entered. Therefore we, also, should simulate the same conditions. Putting our A.I. agent against all of the opponents again, but this time without giving it any head start to observe its opponents. Each game will be a one-off game and the agent will reset between each game. This time, the agent will have to be quick enough to anticipate its opponents moves and exploit their weak spots, within the timeframe of 200 rounds.

You may have noticed that this kind of information is already available to us as the first games' column of Graph 5. After all, on the first round against each opponent, the agent was a blank slate. Graph 6 shows that, on the first game, the agent managed to gather the average of 0.93 of TFT's total points against all the strategies. It was only after having gained some experience from the first game that it managed to attain a score similar to that of TFT, and only on the third round that it managed to surpass it. Nevertheless, most of Soar's learning parameters were left at their default values. For example, on the first few games against any of the opponents, the agent had an epsilon of 0.3, then for a few more games an epsilon of 0.2 and so fourth, until it dropped down to 0 on the last games. Which means that, for most of the games, the agent was playing its assumed optimal move only 70% to 80% of the time, while exploring new moves during the rest 20-30% of that time. It seems that this gives us a margin for optimization, and this area is where we will focus.

Instead of letting the agent explore for whole games at a time, we will try to progressively lower its epsilon value in the span of a single IPD game. This is achieved by setting an exponential *reduction-policy* in Soar, which means that the epsilon is reduced after every decision making step, following an exponential function - that is, very slowly in the beginning but rapidly accelerating after a point, exactly like an exponential curve. Doing this 10 times for each of the opponents, while resetting the Q-values between games so that we have a fresh agent every time, and taking the average of those 10 scores, evens out the chance of the agent hitting a lucky sequence of moves in one of the games.



Graph 7 - Agent's Untrained Performance

The results of the exponentially decreasing epsilon are on Graph 7 above, including the previous first-game scores, with a default epsilon, for good measure. Indeed, the agent performed fairly better with this method - 13% better. In fact, the average of its payoffs amounts to 1.05 of TFT's scores, beating TFT for a short margin of 5%. Reducing the random variance (epsilon) of its moves gradually, within each single game, worked better that reducing it after whole games at a time. Evidently, our agent can figure out a good part of the opponents' strategies fast enough, on average within the first 30 rounds, so that it can exploit its findings for the rest of each game. Of course, it leaves a lot of useful information behind, being in such a hurry. There was still a considerable distance to be covered towards the 1.20 average score of its trained counterpart.

3.4.2. Heuristic Strategy Approximation

Finally, there is one last topic we'd like to explore. Which moves are the ones that our agent statistically prefers? Do they converge to a known strategy or are they evenly spread throughout the spectrum of all the possible moves? In order to find that out, we will modify our Soar program, so that it puts all 12 opponent strategies against our agent in a row, and run it for 10 repetitions. After playing all 24 000 rounds, we will print Soar's output, like we did before, in order to see which moves are prefered by the agent's A.I. the most, and after which situations each. We cannot do this, though, by reading the Q-values which the agent acquired after all those games, since they will not tell us the whole story. Q-values are very dynamic and sensitive to change. After just a few rounds an action's Q-value may have completely changed. Because of this, the final Q-values will heavily, if not solely, reflect the agent's action preferences against only the last opponent. The value that discloses the information we seek, in a more objective manner, is the number of times each move has been preferred and executed by the agent. With this method we can assume which actions had the highest Q-values at the time they were selected. The moves that got selected more times were, simply, the most effective ones for the most part of the game.

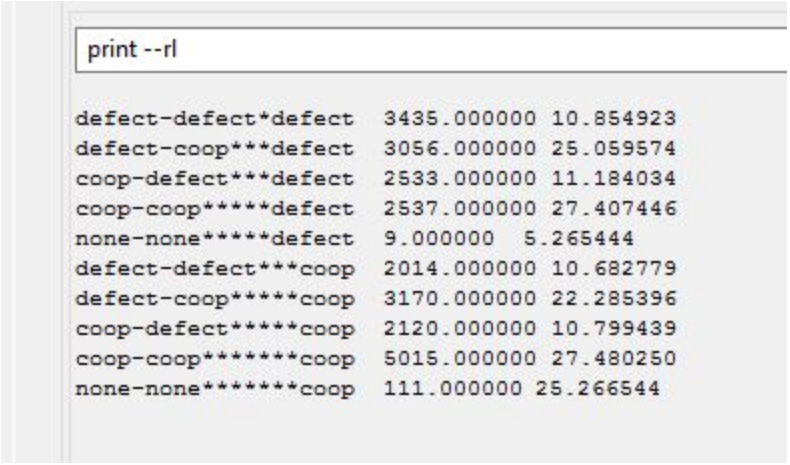


Figure 11 - Agent's Move Preferences

As with the previous terminal printout, the first number column is the one representing how many times each move has been executed and "*none-none*" represents the first move of each

IPD game. Examining the terminal's printout we can see that, after 24 000 iterations, the agent prefers to:

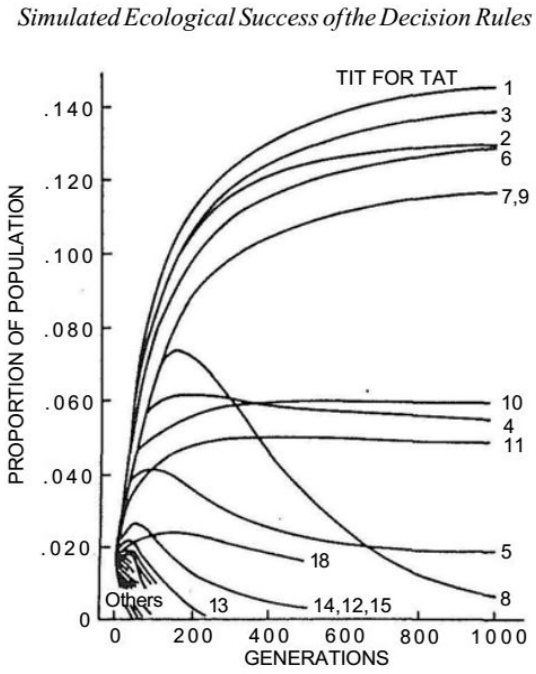
I. Start by cooperating (being *nice*)
II. Defect after mutual defection (D after DD)
III. Defect after the opponent defected on an agent's cooperation (D after CD)
IV. Cooperate after the opponent cooperated on an agent's defection (C after DC)
V. Cooperate after mutual cooperation (C after CC)

Seems familiar? Those 5 rules can be further simplified to the following 3:

I. Start by cooperating
II. Cooperate after the opponent cooperates
III. Defect after the opponent defects

which match exactly Tit for Tat's three core rules. Of course, our agent also played thousands of moves that TFT would have never been able to play, like defecting after mutual cooperation. It is, nevertheless, interesting to see that TFT can be approximated heuristically.

Given that the A.I. agent's moves are adaptive and depend greatly on its environment, it would be the case that, if most opponent strategies were aggressive defectors, the agent would have instead defected most of the time - but so would TFT. In a scenario like this, the pattern that connects the A.I. agent with TFT would have been much more subtle to discern, since we would mostly experience the defecting sides of both strategies. The connection would, however, still exist, just in a hibernated state. Evolutionary studies on the IPD have shown that cooperating strategies, like TFT, reproduce considerably more in most environments.



Graph 8 - Ecological Success of IPD Strategies[2]

Even in an environment comprising of 20 Defectors and only 2 TFTs, the TFT strategies will receive such a boost in points when facing each other that will yield them a higher average payoff than the Defectors'. Allowing TFT to reproduce more that the Defectors towards the next generation and eventually overtaking the "world". The really successful strategies (top of Graph 8) were ones that could work well with other successful strategies, basically nice or otherwise cooperative strategies. They supported one another and were able to continue to reproduce, overwhelming in population the more selfish strategies (no. 4, 10, 11). Exploitative strategies like HARRINGTON (no. 8), did well at the start but as its victims went "extinct" (no. 12, 13, 14, 15, 18, others), its population declined as well [21]. Because cooperative strategies that strike back when provoked have such an evolutionary success, this is exactly the behaviour that we would expect our A.I. agent to evolve over time - as it eventually did.

[2] Axelrod, 1984, *The Evolution of Cooperation*, p.51

Conclusion

After a long journey through Game Theory, Prisoner Dilemmas, learning the Soar Cognitive Architecture and how to train a machine using Reinforcement Learning, we managed to build our Artificial Intelligence agent, that was eventually able to marginally outperform the elite IPD strategy of *Tit for Tat*. The strategy that our agent heuristically deduced as the best strategy was a *nice*, generally cooperating strategy, very much like TFT. Examining evolutionary data for IPD strategies, we identified that the evolutionarily most successful strategies share those same characteristics - starting *nice*, i.e. cooperating on the first round, being generally cooperative and at the same time ready to strike back when their opponents takes advantage of them. Taking this evidence into account, it is not surprising that our RL agent, through trial and error, developed a strategy based on these rules. Our findings are in alignment Axelrod's evolutionary study of cooperation and act as a heuristic support of the theory.

Appendix

Below is the code of the reinforcement learning, standalone PD agent.

The code in its entirety, including all agents and opponents, can be found on my GitHub page: https://github.com/konstantinosftw/Soar-IPD-Thesis-code

Filename: *pd-standalone-rl.soar*

```
# Standalone PD agent - reinforcement learning enabled
# p1 is our Agent
# p2 is the opponent

sp {propose*initialize-prisoners-dilemma
   (state <s> ^superstate nil
              -^name)
-->
   (<s> ^operator <op> +)
   (<op> ^name initialize-prisoners-dilemma)
}

sp {apply*initialize-prisoners-dilemma
   (state <s> ^operator <op>)
   (<op> ^name initialize-prisoners-dilemma)
-->
   (<s> ^name prisoners-dilemma
        ^player <p1> <p2>
        ^rounds <rounds>)
   (<p1> ^name agent
         ^score 0)
   (<p2> ^name opponent
         ^score 0)
   (<rounds> ^current 0
             ^wanted 1)
}

# Propose Agent's coop and defect moves

sp {prisoners-dilemma*propose*cooperate
   (state <s> ^name prisoners-dilemma
              ^player <p1> <p2>
              )
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score <s2>)
-->
   (<s> ^operator <op> +)
   (<op> ^name coop)
}

sp {prisoners-dilemma*propose*defect
   (state <s> ^name prisoners-dilemma
              ^player <p1> <p2>)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score <s2>)
-->
   (<s> ^operator <op> +)
```

```
   (<op> ^name defect)
}

# Opponent's random move - when Agent proposes COOP (so,
# every time) an elaboration choses opponent's random move

sp {prisoners-dilemma*elaborate*opponent-move
   (state <s> ^name prisoners-dilemma
              ^operator <o> +
              # ^operator { <> <o1> } +
              ^player <p2>)
   (<o> ^name coop)
   (<p2> ^name opponent)
-->
   (<p2> ^move (rand-int 1)) # 0 for defect, 1 for coop
}

# Apply Agent's coop and defect moves

sp {prisoners-dilemma*apply*coop-coop
   (state <s> ^operator <op>
              ^player <p1> <p2>
              ^rounds <rou>)
   (<rou> ^current <cu>)
   (<op> ^name coop)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score <s2>
         ^move 1)
-->
   (<p1> ^score <s1> - (+ <s1> 3))
   (<p2> ^score <s2> - (+ <s2> 3))
   (<rou> ^current <cu> - (+ <cu> 1))
   (write (crlf) |Round |(- (dc) 1) |: coop-coop|)

}

sp {prisoners-dilemma*apply*coop-defect
   (state <s> ^operator <op>
              ^player <p1> <p2>
              ^rounds <rou>)
   (<rou> ^current <cu>)
   (<op> ^name coop)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score <s2>
         ^move 0)
-->
   (<p2> ^score <s2> - (+ <s2> 5))
   (<rou> ^current <cu> - (+ <cu> 1))
   (write (crlf) |Round |(- (dc) 1) |: coop-defect|)
}

sp {prisoners-dilemma*apply*defect-coop
   (state <s> ^operator <op>
              ^player <p1> <p2>
              ^rounds <rou>)
   (<rou> ^current <cu>)
   (<op> ^name defect)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score <s2>
         ^move 1)
```

```
-->
   (<p1> ^score <s1> - (+ <s1> 5))
   (<rou> ^current <cu> - (+ <cu> 1))
   (write (crlf) |Round |(- (dc) 1) |: defect-coop|)
}

sp {prisoners-dilemma*apply*defect-defect
   (state <s> ^operator <op>
              ^player <p1> <p2>
              ^rounds <rou>)
   (<rou> ^current <cu>)
   (<op> ^name defect)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score <s2>
         ^move 0)
-->
   (<p1> ^score <s1> - (+ <s1> 1))
   (<p2> ^score <s2> - (+ <s2> 1))
   (<rou> ^current <cu> - (+ <cu> 1))
   (write (crlf) |Round |(- (dc) 1) |: defect-defect|)
}

# Elaborate score after each operator application

sp {prisoners-dilemma*monitor*operator-application
   (state <s> ^name prisoners-dilemma
              ^player <p1> <p2>)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score <s2>)
-->
   (write (crlf) |Agent: | <s1> | / Opponent: | <s2> )
}

# Elaborate who the winner is

sp {prisoners-dilemma*elaborate*winner-agent
   (state <s> ^name prisoners-dilemma
              ^player <p1> <p2>
              ^rounds.current <cu>
              ^rounds.wanted <cu>)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score { < <s1> <s2>})
-->
   (write (crlf) |--- WINNER IS: AGENT ---| )
   (halt)
}

sp {prisoners-dilemma*elaborate*winner-opponent
   (state <s> ^name prisoners-dilemma
              ^player <p1> <p2>
              ^rounds.current <cu>
              ^rounds.wanted <cu>)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score { > <s1> <s2>})
-->
   (write (crlf) |--- WINNER IS: OPPONENT ---| )
   (halt)
}
```

```
sp {prisoners-dilemma*elaborate*winner-draw
   (state <s> ^name prisoners-dilemma
              ^player <p1> <p2>
              ^rounds.current <cu>
              ^rounds.wanted <cu>)
   (<p1> ^name agent
         ^score <s1>)
   (<p2> ^name opponent
         ^score <s1>)
-->
   (write (crlf) |--- THE GAME IS DRAW ---| )
   (halt)
}

# Learning Rules

sp {prisoners-dilemma*rl*coop
   (state <s> ^name prisoners-dilemma
              ^operator <op> +)
   (<op> ^name coop)
-->
   (<s> ^operator <op> = 0)
}

sp {prisoners-dilemma*rl*defect
   (state <s> ^name prisoners-dilemma
              ^operator <op> +)
   (<op> ^name defect)
-->
   (<s> ^operator <op> = 0)
}

# Reward (once at the end of all iterations)

sp {elaborate*reward*end-game
   (state <s> ^name prisoners-dilemma
              ^reward-link <r>
              ^player (^name agent
                       ^score <score>)
              ^rounds.current <cu>
              ^rounds.wanted <cu>)
-->
   (<r> ^reward.value <score>)
}


# -------- END -------- #
```

References

1. Poundstone, William, 1992, “Prisoner's Dilemma: John von Neumann, Game Theory, and the Puzzle of the Bomb”, Anchor Publishing
2. Axelrod, Robert and William Hamilton, 1981, “The Evolution of Cooperation,” Science, 211 (March 27): 1390–1396.
3. Axelrod, Robert, 1981, “The Emergence of Cooperation Among Egoists,” The American Political Science Review, 75: 306–318.
4. Crevier, Daniel, 1993, “AI: The Tumultuous Search for Artificial Intelligence”, New York, BasicBooks
5. Newell, Allen, 1990, “Unified Theories of Cognition”, Harvard University Press
6. Laird, J., Rosenbloom, P., & Newell, A., 1986, “Universal Subgoaling and Chunking”, Kluwer Academic Publishers
7. Laird, J. E., Newell, A., & Rosenbloom, P. , 1987, “SOAR: An architecture for general intelligence”, Artificial Intelligence, 33, 1-64
8. Anderson, J. R., 1993, “Rules of the Mind”, Lawrence Erlbaum Associates
9. Lebiere, C., 1996, “Act-R: A Users Manual”, [On-line] Available: ftp://ftp.andrew.cmu.edu/pub/actr/ftp/release/beta/ACTR3TXT/Manual.rtf
10. Anderson, J. R., 1990, “ The Adaptive Character of Thought”, Lawrence Erlbaum Associates
11. Johnson, T. R., 1997, Control in Act-R and Soar. In M. Shafto & P. Langley (Eds.), Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society (pp. 343-348), Lawrence Erlbaum Associates
12. Ernst, G.W. and Newell, A., 1969, “GPS: a case study in generality and problem solving”, Academic Press
13. Simon, Herbert A., and Allen Newell, 1958, “Heuristic Problem Solving: The Next Advance in Operations Research”, Operations Research, vol. 6, no. 1, pp. 1–10.
14. Newell, A. 1991, “Reasoning, problem solving and decision processes: The problem space as a fundamental category”, In Attention and Performance VIII, ed. N. Nickerson. Erlbaum.
15. Newell, A., and Simon, H. A., 1972, “Human Problem Solving”, Prentice-Hall

16. Newell, A., and Simon, H. A., 1956, “The logic theory machine: A complex information processing system”, Institute of Radio Engineers Transactions on Information Theory 2 vol. 3, pp. 61 – 79
17. Russell, Stuart J., Norvig, Peter, 2003, “Artificial Intelligence: A Modern Approach” (2nd ed.), Prentice Hall
18. Laird, J. E., 2012, “The Soar Cognitive Architecture”, MIT Press
19. Kahneman, Daniel, 2011, “Thinking, Fast and Slow”, Farrar, Straus and Giroux
20. Newell, Allen; Simon, H. A., 1976, "Computer Science as Empirical Inquiry: Symbols and Search", Communications of the ACM, 19, vol. 3, pp. 113–126
21. Axelrod, Robert, 1984, “The Evolution of Cooperation”, Basic Books
22. Tucker, A. W., Luce, R. D., 1959, "Acceptable Points in General Cooperative n-Person Games", Contributions to the Theory of Games IV, Annals of Mathematics Study 40, Princeton University Press, pp. 287-324.