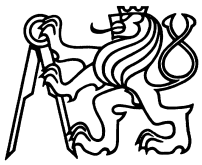
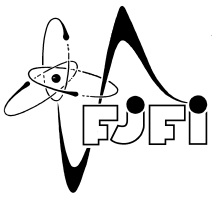


Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering



Continual resolving heuristics for the PAWS domain

Heuristiky algoritmu soustavně opakovaného řešení pro doménu PAWS

Research Project

Author: **Miroslav Kovář**

Supervisor: **Mgr. Viliam Lisý, MSc., Ph.D.**

Academic year: 2016/2017

- Zadání práce -

- Zadání práce (zadní strana) -

Acknowledgment:
I would like to express honest gratitude to my supervisor Viliam Lisý for his valuable advice and patience.

Author's declaration:
I declare that this research project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, February 6, 2018 Miroslav Kovář

Název práce:

Heuristiky algoritmu soustavně opakovaného řešení pro doménu PAWS

Autor: Miroslav Kovář

Obor: Aplikace přírodních věd

Zaměření: Matematická informatika

Druh práce: Výzkumný úkol

Vedoucí práce: Mgr. Viliam Lisý, MSc., Ph.D., Department of Computer Science, FEE, CTU Prague

Abstrakt: Mnoho situací reálného světa lze modelovat jako hry s nedokonalou informací. Nedávný úspěch algoritmu DeepStack proti lidským hráčům ve hře pokeru motivuje snahu tento algoritmus zobecnit pro jiné aplikace. Srdcem DeepStacku je heuristická funkce sloužící pro rychlou evaluaci herního stavu (podobně lidské intuici), původně tvořená předtrénovanou neuronovou sítí. V tomto článku zvažujeme algoritmus inspirovaný DeepStackem použitý jako nástroj pro ulehčení ochrany divoké zvěře a diskutujeme možnost vyhnutí se trénovací fáze řešením příslušného optimalizačního problému přímo.

Klíčová slova: heuristiky, hry s nedokonalou informací, optimalizace, teorie her, zelené bezpečnostní hry

Title:

Continual resolving heuristics for the PAWS domain

Author: Miroslav Kovář

Abstract: Many real-world situations can be modelled as imperfect information games. Recent success of DeepStack algorithm in the game of poker against human players motivates effort to generalize the algorithm to other applications. At the heart of DeepStack is a heuristic function serving as fast evaluation of the game state (akin to human intuition), originally performed by a pre-trained neural network. In this paper, we consider a DeepStack-inspired algorithm as a wildlife protection decision aid, and discuss the possibility of avoiding the training phase by solving the optimization problem task-specifically.

Key words: game theory, green security games, heuristics, imperfect information games, optimization

Contents

Introduction **11**

1 PAWS **13**
1.1 SSG game model . . . 13
1.2 Adversary's behavior model . . . 14
1.3 Terrain modelling . . . 14
1.4 Computing patrol strategy . . . 16

2 Counterfactual regret minimization **17**
2.1 Preliminaries . . . 17
2.2 Regret minimization . . . 19
2.3 Counterfactual regret . . . 19
2.4 Pseudocode . . . 21

3 DeepStack **23**
3.1 Abstractions . . . 23
3.2 Continual resolving . . . 24
3.2.1 Introduction . . . 24
3.2.2 CFR-D and strategy reconstruction . . . 26
3.2.3 Continual resolving in DeepStack . . . 27
3.3 Look-ahead heuristic . . . 28
3.4 Pseudocode . . . 28

4 Continual resolving in PAWS **31**
4.1 Game model . . . 31
4.2 Evaluation function heuristic . . . 33
4.3 Orienteering problem . . . 34
4.3.1 NP-completness . . . 36
4.3.2 Solution approaches . . . 36
4.3.3 Performance comparison . . . 41
4.3.4 Implementation . . . 41
4.4 Arc routing problem . . . 41
4.4.1 Approaches . . . 43
4.4.2 Performance comparison . . . 48

Conclusion **53**

Introduction

Protection of critical infrastructure such as airports, flights, ports, and metro trains have been the domain of game theory subfield called *security games*. These problems are usually modelled as Stackelberg games, i.e. as a competition for a limited resource where an agent called *leader* moves first, and other agents, called *followers* observe their action and then make their own decision. However, it can be observed that all of these problems represent game-like situations in which

- only incomplete information is available to the actors and
- the actors may have incentives to engage in deception,

which lends them to be modelled as imperfect information games.

One of the obstacles in this paradigm is that modelling these real-world situations requires large state, action and consequently strategy spaces to search for equilibria in. Moreover, reasoning about optimal strategies in imperfect information games differs from reasoning in perfect information games, as the optimal action at a state depends on the distribution over the information hidden to the currently acting player.

However, recent breakthrough in solving large imperfect information game of poker [16] - the DeepStack algorithm, combining the techniques of counterfactual regret minimization, limited look-ahead and continual resolving - remedies these issues and therefore lends itself to inspire a new approach to tackling the aforementioned problems in the area of security games.

PAWS (Protection Assistant for Wildlife Security) - anti-poaching route planning aid - is one of the first of a wave of proposed applications in a new subfield of security games called *green security games* and our aim is to propose a modification of the DeepStack algorithm to the PAWS domain.

At the heart of DeepStack is a heuristic function serving as fast evaluation of the game state performed by a pre-trained neural network. One of our objectives is to avoid the training phase by approximating the function's value directly on demand and task-specifically, instead of generalizing training examples using a neural network. This allows to use the same algorithm on many instances of the problem without compuationally expensive training phase, and provides the ability to change the output based on observations made after the initial computation, such as fallen trees, wet terrain or other route obstructions.

In this paper, we begin this endeavor by introducing the problem in detail, our approach to solution and comparison of viable heuristics needed as part of the final algorithm.

The first chapter presents PAWS and its current solutions. The second chapter sets out the basic theory of counterfactual regret minimization behind DeepStack. The third chapter briefly introduces DeepStack. The fourth chapter includes introduction to our approach of modelling PAWS domain and a resulting subproblem called the Orienteering Problem and comparison of its solution heuristics.

Chapter 1

PAWS

Combatting poaching is a decision-making task which requires efficient scheduling of limited resources - in particular, a timely and randomized allocation of park rangers (personnel or volunteers charged with protecting wildlife in a given area) to patrol routes such as to minimize the negative effects of illegal poaching activity. In this chapter, we introduce PAWS (Protection Assistant for Wildlife Security), an AI decision-aid system based on subfield of game theory called green security games, proposed in [36]. It models the decision-making task as a repeated Stackelberg security game.

1.1 SSG game model

In Stackelberg Security Games (SSG), one of the players takes the role of a *leader* and the rest of the players are *followers*. The leader acts first by commiting to a (mixed) strategy which is always known to be observed by the followers. Leader a players act sequentially, competing for resources. The leader's goal is to protect a set of targets using limited resources from being attacked by the adversary.

Formally, the attacker may choose to attack any one target from a finite set $T = \{t_1, t_2, \ldots, t_n\}$. The defender uses resources from a finite set $R = \{r_1, r_2, \ldots, r_K\}$. Each resource r_i is constrained by a set of schedules $S_i \subset \mathcal{P}(T)$, the set of sets of targets that the resource is able cover. The attacker's mixed strategy is a vector $(a_1, a_2, \ldots, a_n)$ where a_i is the probability of attacking target t_i. The defender's mixed strategy is a distribution over assignments of resources to targets compatible with their schedules, i.e. $(s_1, s_2, \ldots, s_K) \in S_1 \times S_2 \times \cdots \times S_K$. [12]

PAWS can be modelled as SSG as follows: The rangers take on the role of the leader, trying to protect animals by patroling on locations which are suspect of being attacked by the poachers, while the poachers are the followers, trying to poach by setting traps on those locations. The rangers execute a mixed strategy for a period of time (e.g. a month). The poachers conduct surveillance, (which gives them perfect knowledge of opponent's strategy)[1] and respond. New round begins with rangers picking a new mixed strategy.

The entire area is discretized into a grid of 1 km^2 grid cells. Let T represent the set of all grid cells. If the poacher attacks target $i \in T$ and it is protected by the defender, the attacker receives penalty $U^a_{p,i}$ and the defender receives reward $U^d_{r,i}$. Conversely, if it is not protected, the attacker receives reward $U^a_{r,i}$ and the defender receives penalty $U^d_{p,i}$. Obviously $U^a_{p,i} \leq U^a_{r,i}$ and $U^d_{p,i} \leq U^d_{r,i}$. Moreover, PAWS is zero-sum, i.e. we have $U^d_{r,i} = -U^a_{p,i}$ and $U^d_{p,i} = -U^a_{r,i}$. The payoffs are determined by animal density, as well as distance from poachers' and rangers' respective camps.

[1] This game model assumes that the surveillance is always performed. There are modifications of the SSG game model which allow for probabilistic determination of whether the attacker performed the surveillance. [12]

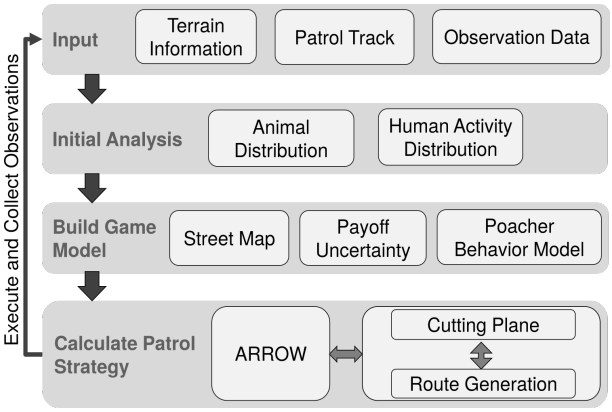


Figure 1.1: PAWS data flow diagram. [6]

Let the defender's strategy be a coverage vector $\langle c_i \rangle$, where c_i is the probability of defending a grid cell i. Then, the players' expected utilities are

$$U_i^a = c_i U_{p,i}^a + (1 - c_i) U_{r,i}^a \tag{1.1}$$

$$U_i^d = c_i U_{r,i}^d + (1 - c_i) U_{p,i}^d. \tag{1.2}$$

1.2 Adversary's behavior model

Previous work on SSG often assumed perfect rationality of the opponent [23]. PAWS is one of the first deployed applications in which this assumption was relaxed in favor of a boundedly rational model[2] called SUQR, which was shown to perform well against other models in experiments involving human subjects [20]. This model predicts the probability of attacking i as

$$q_i = \frac{e^{SU_i(\omega)}}{\sum_j e^{SU_j(\omega)}}, \tag{1.3}$$

where $SU_i(\omega) = w_1 c_i + w_2 U_{r,i}^a + w_3 U_{p,i}^a$ is subjective utility function and $\omega = (w_1, w_2, w_3)$ are parameters learned from past data of poaching activity. Experiments show that this models performs better when parameters ω are assumed to be normally distributed as opposed to fixed (see Fig. 1.2). [36]

1.3 Terrain modelling

The routes generated by the first version of PAWS which were occasionaly difficult to follow because of terrain constraints [6]. This prompted the second version to take terrain information into account. In particular, the input included countour lines describing elevation and locations of lakes. In addition, each-grid cell is further discretized into 50x50 m raster pieces (see Fig. 1.3). For each grid cell, a small subset of those raster pieces is then selected as *KAPs* (key access points), such as basecamps and mountaintops. Pairs of easily accessible raster pieces are chosen on grid cell boundaries. Then, local route segments connecting each pair of KAPs are found within each grid cell, effectively forming a graph with KAPs as nodes and route segments as edges. Therefore, in the following text, the words KAP and node, route segment and edge will be used interchangeably.

[2]Bounded rationality is a framework wherein the decision-makers' decisions are limited by their cognitive abilities, time constraints and complexity of the problem.

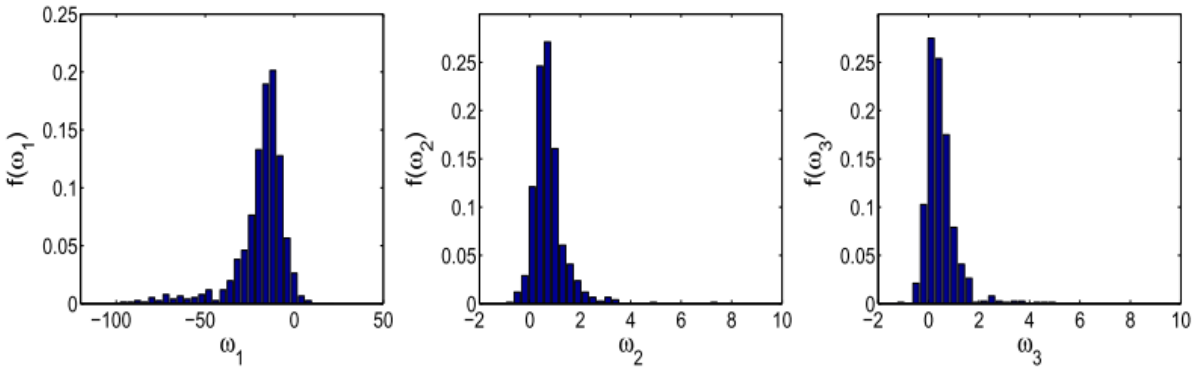


Figure 1.2: Measured distributions of the SUQR parameter.

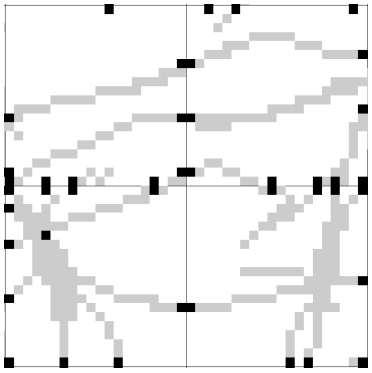


Figure 1.3: Four adjacent gridcells with highlighted KAPs. [6]

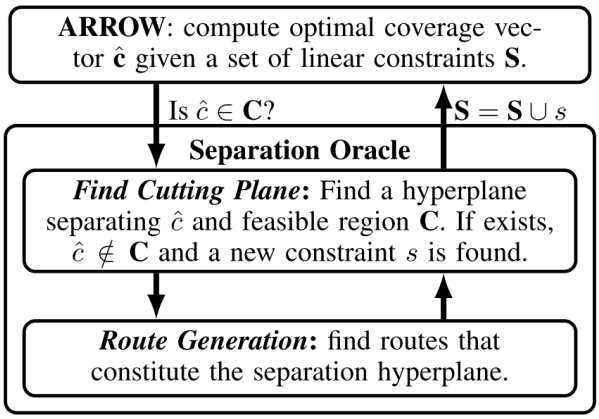


Figure 1.4: Algorithm integrating ARROW and BLADE. [6]

1.4 Computing patrol strategy

Animal densities are modelled from collected data using predictive probability raster for tigers [24], and these densities are then smoothed over raster pieces. It is necessary, however, to include uncertainty in the expected payoffs due to uncertainty in animal activity (such as migrations).

One technique of handling uncertainty in imperfect information games is called *regret minimization* (see Chapter 2). In particular, the ARROW algorithm (Algorithm for Reducing Regret to Oppose Wildife crime) [19], minimizes the maximum behavioral regret with respect to uncertainties in animal densities and adversary's behavior (given it's modelled by SUQR). It's output is a coverage vector of probabilities assigned to each grid cell, indicating ideal probability of defending the grid cell, without consideration of any routes. In order to generate defenders mixed strategy, i.e. distribution over a number of routes, the BLADE algorithm [37] is called to check if there are routes satisfying coverage probabilities given by ARROW, and either return the routes or a constraint used in another iteration of ARROW to refine its solution.

BLADE exploits the similarity with well-known *orienteering problem*, where the task is to, given a starting point, find a route on undirected vertex-valued graph maximizing the sum of values over visited verteces. Specifically, it uses similar approach to the one developed by Tsiligirides (1984) [30]. Starting at the base KAP, it iteratively chooses the next KAP to visit with probability proportional to the ratio of probability it should be visited (coverage probability of the grid cell which lies between it and the current KAP) and it's distance, and continues until the patrol distance limit is exceeded. In this manner, it generates a sample of routes and either pick the ones that successfully implemented the coverage vector with corresponding probabilities, or returns a new constraint.

Chapter 2

Counterfactual regret minimization

In 2000, Hart and Mas-Colell [10] introduced game-theoretic algorithm for normal form games called *regret matching*, whereby players reach equilibrium by repeatedly playing against each other, while continually adjusting their strategies based on accumulated quantities called *regrets*. In 2007, Bowling et al. [38] shown that regret matching in conjuction with newly introduced *counterfactual regret minimazation* can be used to compute arbitrarily close approximations to Nash equilibrium in large extensive games with imperfect information by self-play. Since then, these techniques found use in solutions to complex imperfect information games, such as poker.

In this chapter, we provide definitions of the basic terms used throughout this paper and a pseudocode of the CFR algorithm.

2.1 Preliminaries

Definition 1. *[[13]]* ***Finite extensive game with imperfect information*** *is 8-tuple* $\Gamma = (N, V, E, x^0, (V_i)_{i\in N}, \mathcal{I}, Z, O, u)$*, where*

N *is a finite set of players,*

(V, E, x^0) *is a finite rooted tree, where* V *is a set of vertices,* E *is a set of edges, and* $x_0 \in V$ *is the root,*

$(V_i)_{i\in N}$ *is a decomposition of* V *into disjunct subsets, where* V_i *is the set of vertices where player* i *acts,*

$\mathcal{I} = \{I_i^j\}_{i\in N}^{1,\dots k_i}$ *is a set of decompositions of each set* V_i *into* k_i *disjunct subsets called* ***information sets*** *of player* i*,*

$A(I_i^j)$ *are* ***action sets*** *for each player* $i \in N$*,* $j = 1, \dots k_i$*,*

$Z \subset V$ *is a set of* ***terminal (leaf) nodes****,*

O *is a set of* ***game outcomes*** *(e.g. real valued* $|N|$ *dimensional vectors of rewards for each player* $i \in N$*),*

$u : Z \to O$ *is a mapping of terminal nodes to outcomes,*

where ***information set*** *and* ***action set*** *is a tuple* $(I_i, A(I_i))$ *such that*

- $I_i = \{x_i^1, \dots, x_i^m\} \subseteq V_i$ *such that* $m \geq 2$ *and* $(\forall j \in \{1, \dots m\})\ |A(x_i^j)| = l_i$*,*
- $A(I_i)$ *is a decomposition of* $\cup_{j=1}^m A(x_i^j)$ *into* l_i *blocks, where each block contains exactly one element from each of the sets* $A(x_i^1), \dots, A(x_i^m)$*.*

Moreover, let

- *H denote the set of possible **histories**, finite walks on (V, E, x^0) starting on x^0,*
- *$Z(h)$, $Z(I)$ the set of terminal nodes reacheble from h, I,*
- *$I(h)$ the information set reached by history $h \in H$, and*
- *$A(h) = \{a \in E \mid (h, a) \in H\}$ for $h \in H$.*

Intuitively, extensive game is a game visualizable as a tree. Information set is a set of m nodes belonging to player i such that upon arriving to any of them, they cannot distinguish which one has been reached, only that it is a member of the particular information set - they have imperfect information about their position on the game tree. If any information set has more than one member, the game is called imperfect information game.

Each $h \in H$ can be mapped surjectively on an information set I. Therefore, in the following, we will sometimes use I in place of h.

Definition 2. ***Strategy** of player $i \in N$ is $\sigma_i : \mathcal{I}_i \to \bigcup_{j=1}^{k_i} A(I_i^j)$ such that $\sigma(I_i^j) \in A(I_i^j)$. **Strategy profile** $\sigma = (\sigma_1, \ldots, \sigma_{|N|})$ consists of strategies of all players. The set of all strategies of player i is denoted Σ_i.*

In the following, we will denote

- $\pi^\sigma(h) = \prod_{i \in N} \pi_i^\sigma(h)$ the probability of history $h \in H$ occuring given strategy profile σ,
- $\pi_{-i}^\sigma(h) = \prod_{i' \in N \setminus i} \pi_i^\sigma(h)$ the product of all players contributions except for player i,
- $\pi^\sigma(I) = \sum_{h \in I} \pi^\sigma(h)$ the probability of reaching information set I given strategy profile σ,
- $\pi_i^\sigma(I)$, $\pi_{-i}^\sigma(I)$ analogically, and
- $u_i(\sigma) = \sum_{h \in Z} u_i(h)\pi^\sigma(h)$ the value of strategy profile σ to player i.

Definition 3. ***A Nash equilibrium** of a two-player game is a strategy profile σ where*

$$u_1(\sigma) \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2) \qquad u_2(\sigma) \geq \max_{\sigma_2' \in \Sigma_2} u_2(\sigma_1, \sigma_2'). \tag{2.1}$$

*An ϵ approximation of Nash equilibrium or **ϵ-Nash equilibrium** is a strategy profile σ where*

$$u_1(\sigma) + \epsilon \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2) \qquad u_2(\sigma) + \epsilon \geq \max_{\sigma_2' \in \Sigma_2} u_2(\sigma_1, \sigma_2'). \tag{2.2}$$

Definition 4 ([3])**.** *Let C denote defender's mixed strategy in a SSG, and $g : C \to$ a the attacker's response function. **A strong Stackelberg equilibrium** is a strategy profile (C, g) such that*

1. *The defender plays a best response:*

$$U_d(C, g(C)) \geq U_d(C', g(C')), \ \forall C'$$

2. *The attacker plays a best response:*

$$U_a(C, g(C)) \geq U_a(C, g'(C)), \ \forall g'$$

3. *The attacker breaks ties optimally for the defender:*

$$U_d(C, g(C)) \geq U_d(C, a'), \ \forall a' \in \arg\max_a U_a(C, a).$$

2.2 Regret minimization

Let us consider playing an extensive game repeatedly. Let σ^t be the strategy profile at time t, $I \in \mathcal{I}$, $a \in A(I)$.

Definition 5 ([38]). ***Average overall regret*** *is defined as*

$$R_i^T = \frac{1}{T} \max_{\sigma_i^* \in \Sigma_i} \sum_{t=1}^{T} (u_i(\sigma_i^*, \sigma_{-i}^t) - u_i(\sigma^t))$$

Definition 6 ([38]). ***The average strategy*** *is defined as*

$$\overline{\sigma}^T(I)(a) = \frac{\sum_{t=1}^{T} \pi_i^{\sigma^t}(I)\sigma^t(I)(a)}{\sum_{t=1}^{T} \pi_i^{\sigma^t}(I)}$$

An algorithm for finding σ_i^t is called *regret minimizing* if and only if $\lim_{T\to+\infty} R_i^T = 0$ irrespective of $(\sigma_{-i}^t)_{t=1}^T$. The following theorem shows that regret minimizing algorithms (such as CFR) can be used to compute ϵ approximates of Nash equilibrium. Moreover, the average overall regret bounds the rate of convergence of the approximation.

Theorem 1. *In a zero-sum game at time T, if both player's average overall regret is less than ϵ, then $\overline{\sigma}^T$ is a 2ϵ equilibrium.*

Proof. See [35]. □

2.3 Counterfactual regret

In order to use regret minimization on large extensive games, the overall regret needs to be decomposed into independently minimizable, local, additive regret termscalled *counterfactual regrets*.

Definition 7 ([18]). *Player i's **counterfactual value** at non-terminal history $h \in V \setminus Z$ is defined as expected utility given that history h is reached and all players play strategy σ except player i plays to reach h, formally*

$$v_i(\sigma, h) = v_i^{\sigma}(h) = \sum_{z \in Z(h)} \pi_{-i}^{\sigma}(h)\pi^{\sigma}(h, z)u_i(z).$$

*Player i's **counterfactual value** at information set $I \in \mathcal{I}$ is defined as [38]*

$$v_i(\sigma, I) = v_i^{\sigma}(I) = \frac{1}{\pi_{-i}^{\sigma}(I)} \sum_{h \in I, z \in Z(h)} \pi_{-i}^{\sigma}(h)\pi^{\sigma}(h, z)u_i(z).$$

Intuitively, counterfactual value can be understood as the expected utility of reaching $h \in H$ or $I \in \mathcal{I}$ given σ with player i actively and deliberately attempting to reach h or I.

Definition 8 ([18]). *Player i's **expected utility (payoff)** at history h, information set I, or **net payoff** given strategy σ is respectively defined as*

$$u_i(\sigma, h) = u_i^{\sigma}(h) = \sum_{h' \in Z(h)} u_i(h')\pi^{\sigma}(h')$$

$$u_i(\sigma, I) = u_i^{\sigma}(I) = \sum_{z \in Z(I)} u_i(z)\pi^{\sigma}(z)$$

$$u_i(\sigma) = u_i^{\sigma} = \sum_{z \in Z} u_i(z)\pi^{\sigma}(z)$$

Definition 9 ([38]). *Player i's **conterfactual regret** of taking action $a \in A(I)$ on information set $I \in \mathcal{I}_i$ is*

$$R_i^T(I,a) = \frac{1}{T}\sum_{t=1}^{T} \pi_{-i}^{\sigma^t}(I)(v_i(\sigma^t|_{I\to a}, I) - v_i(\sigma^t, I)),$$

where $\sigma^t|_{I\to a}$ is a strategy profile identical to σ except that player i always plays $a \in A(I)$ on $I \in \mathcal{I}_i$.

We define $R_i^T(I) = \max_{a\in A(I)} R_i^T(I,a)$ and $R_i^{T,+} = \max(R_i^T, 0)$.

Intuitively, counterfactual regret can be understood as a cumulative statistic of how much better player i would have done if he had played action a on information set I instead of action dictated by σ, weighted by the probability that I would be reached given σ with player i actively attempting to reach I.

The following theorem shows that any algorithm minimizing the sum of positive counterfactual regrets over all information sets is a regret minimizing algorithm and therefore enables computation of approximate Nash equilibria.

Theorem 2. $R_i^t \le \sum_{I\in\mathcal{I}_i} R_i^{T,+}(I)$

Proof. See [38]. □

It can be shown that if actions of all players during self-play are selected in proportion to the amount of positive counterfactual regret of not choosing them and randomly if no actions have positive counterfactual regret, the resulting strategy will converge to Nash equilibrium. This very important result is formalized as follows. [18]

Definition 10 (Regret matching)**.**

$$\sigma_i^{T+1}(I,a) = \begin{cases} \frac{R_i^{T,+}(I,a)}{\sum_{a\in A(I)} R_i^{T,+}(I,a)} & \text{if } \sum_{a\in A(I)} R_i^{T,+}(I,a) > 0 \\ \frac{1}{|A(I)|} & \text{otherwise} \end{cases} \tag{2.3}$$

Theorem 3. *If player i's strategy is given by (2.3), then*

$$R_i^T(I) \le (\max_z u_i(Z) - \min_z u_i(z))\sqrt{\frac{\max_{I\in\mathcal{I}_i}|A(I)|}{T}}.$$

Proof. See [38]. □

Definition 11 ([17]). ***Best response** strategy of player i to σ_{-i} is a strategy that maximizes i's expected utility given σ_{-i}, i.e.*

$$\mathrm{BR}_i(\sigma_{-i}) = \arg\max_{\sigma_i'} u_i(\sigma_i', \sigma_{-i}).$$

Definition 12 ([17]). ***Counterfactual best response** of player i to σ is a strategy which maximizes the counterfactual value at every information set, i.e.*

$$\mathrm{CBR}_i(\sigma_{-i})(I,a) > 0 \iff v_i^{\langle CBR_i(\sigma_{-i}),\sigma_{-i}\rangle}(I) = \max_{a'\in A(I)} v_i^{\sigma|_{I\to a'}}(I).$$

Intuitively, while BR_i maximizes expected value, CBR_i maximizes counterfactual value at every information set. Unlike BR_i, CBR_i maximizes counterfactual value even at every information set, including those that it does not reach due to its earlier actions. Therefore, it is the case that $\mathrm{CBR}_i(\sigma_{-i})$ is always $\mathrm{BR}_i(\sigma_{-i})$, but not otherwise, since BR_i can choose arbitrary actions at states I where $\pi_i(I) = 0$.

Definition 13 ([17]). ***Counterfactual best response value*** *of player i is the counterfactual value of playing* $\mathrm{CBR}_i(\sigma_{-i})$*, i.e.*

$$\mathrm{CBV}_i^{\sigma_{-i}} = v_i^{\langle \mathrm{CBR}_i(\sigma_{-i}), \sigma_{-i} \rangle}.$$

Definition 14 ([7]). *Let* σ *be a Nash equilibrium. The term* ***exploitability*** *of a strategy* σ'_i *is*

$$\mathrm{expl}(\sigma'_i) = u_i^{\sigma} - u_i^{\langle \sigma'_i, CBR_{-i}(\sigma'_i) \rangle}.$$

Intuitively, exploitability of σ'_i is the loss player i incurs by switching from Nash equilibrium to σ'_i. The notion of BR allows for an alternative definition of Nash equilibrium:

Definition 15. *Nash equilibrium is a strategy profile* σ *where all players are simulataneusly playing best response strategies, i.e.*

$$(\forall i \in N)\ \sigma_i = \mathrm{BR}_i(\sigma_{-i}).$$

2.4 Pseudocode

A version of the basic CFR algorithm for two players is provided in Algorithm 1. Recursion on the `CFR` function is used for depth-first traversal the game tree. It takes the history of already visited nodes `h`, the number of player whose optimal strategy being learned `i`, the number of current iteration `t` and the respective players' probabilities of reaching the current node given learned strategy π_1, π_2. First, the regret tables r_I and cumulative strategy s_I (which is used to store the solution - approximation of Nash equilibrium) are initialized to zero, and the initial strategy profile is set to uniform.

If the reached node is a leaf, its counterfactual value is returned. If the reached node is a chance node, an action is sampled from the distribution σ_c associated with the chance node. Else, counterfactual values of the child nodes are computed recursively. Lines 17, 19 and 21 are used to find the terms in the sum from Definition 9, accumulated over the iterations on line 25. Finally, strategy for the next iteration is calculated by regret matching from Definition 10 and the counterfactual values are returned.

The `CFR` function is called and the optimal strategy is computed separately for each player.

Algorithm 1 Counterfactual Regret Minimization with chance sampling. [18]

Initialize cumulative regret tables: $\forall I, r_I[a] \leftarrow 0$
Initialize cumulative strategy tables: $\forall I, s_I[a] \leftarrow 0$
Initialize initial profile: $\sigma^1(I, a) \leftarrow 1/|A(I)|$
function CFR(h,i,t, π_1, π_2)
    if h is terminal **then**
      return $u_i(h)$
    end if
    if h is a chance node **then**
        Sample a single outcome $a \sim \sigma_c(h, a)$
      return CFR(ha, i, t, π_1, π_2)
    end if
    Let I be the information set containing h
    $v_\sigma \leftarrow 0 \; \forall a \in A(I)$
    for $a \in A(I)$ **do**
        if $P(h) = 1$ **then**
            $v_{\sigma_{I \to a}}[a] \leftarrow CFR(ha, i, t, \sigma^t(I, a) \cdot \pi_1, \pi_2)$
        end if
        if $P(h) = 2$ **then**
            $v_{\sigma_{I \to a}}[a] \leftarrow CFR(ha, i, t, \pi_1, \sigma^t(I, a) \cdot \pi_2)$
        end if
        $v_\sigma \leftarrow v_\sigma + \sigma^t(I, a) \cdot v_{\sigma_{I \to a}}[a]$
    end for
    if $P(h) = i$ **then**
        for $a \in A(I)$ **do**
            $r_I[a] + \pi_{-i} \cdot (v_{\sigma_{I \to a}}[a] - v_\sigma)$
            $s_I[a] \leftarrow s_I[a] + \pi_i \cdot \sigma^t(I, a)$
        end for
        $\sigma^{t+1}(I) \leftarrow$ regret-matching values computed using (2.3)
    end if
      return v_σ
end function

procedure SOLVE
    for $T = \{1, 2, \ldots, T\}$ **do**
        for $i \in \{1, 2\}$ **do**
            CFR($\emptyset$, i, t, 1, 1)
        end for
    end for
end procedure

Chapter 3

DeepStack

One of the main obstacles to finding quality strategies in large imperfect information games is intractability of big-sized game-trees. In Heads-Up No-Limit Texas Hold'em poker (HUNL), the number of possible game states reaches the order of 10^{160}. DeepStack combines the following three ingredients in order to address this issue:

- reducing dimensionality of state and action spaces using *abstractions*,
- minimizing the amount of memory required to remember the action histories and generated strategy via *continual resolving*,
- depth-limited *look-ahead heuristic* (i.e. a neural network).

These ingredients have been used with success in perfect information games, however, DeepStack is the first algorithm to combine all of them in theoretically sound way on imperfect information games [16].

In the following text, we will desribe each of those points in more detail. The reader is expected to have a very basic and intuitive understanding of the rules of HUNL and be familiar with commonly used poker terms.

3.1 Abstractions

A traditional approach of reducing game-tree size is by grouping actions and states, creating a "coarser game". Then, an ϵ-equilibrium is computed for the abstracted game and used for the gameplay of the original game by translating the state in the original game to the group in the coarser game,

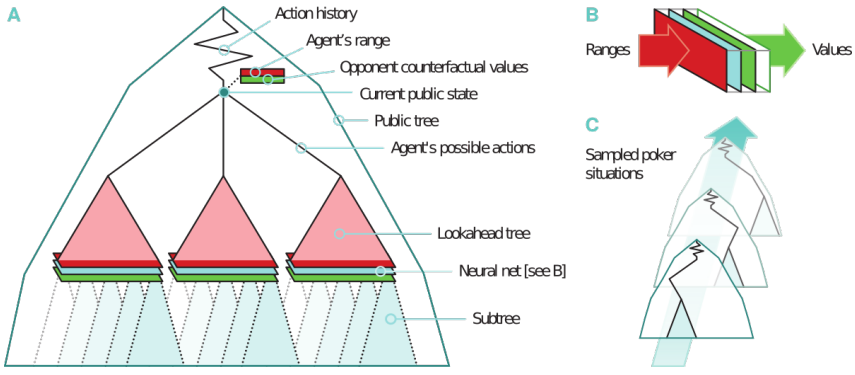


Figure 3.1: Overview of DeepStack's continual resolving [16]

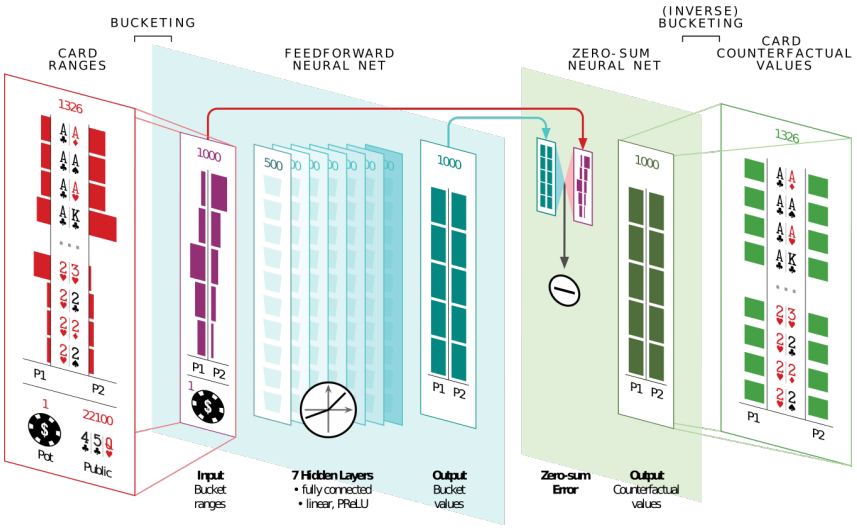


Figure 3.2: DeepStack's counterfactual value deep neural network design. Before entering the feedforward network, the input - players' ranges and public state - is pooled into 1000 clusters. The feedforward network itself consists of seven fully connected hidden layers with 500 nodes overall, and is embedded in another neural network, which forces the counterfactual values returned by the network to satisfy the zero-sum property (for the case that counterfactual values returned for each player's ranges don't satisfy it). [16]

reading the action group from the computed strategy, and translating to an action in the original game. A drawback of grouping and translation is the possibility of information loss in case the original action or state and its translation differ in an aspect important for the game. [17]

A reasonable compromise between solving the entire game and saving space with abstractions is computing a strategy for largest handleable game and continually refining the strategy as the remainder of the game becomes increasingly tractable.

DeepStack's use of abstractions is very limited. It does not abstract the entire game. Instead, it employs continual resolving (see Section 3.2), while limiting even the depth of resolved subgame further by using look-ahead heuristic (see Section 3.3). The abstraction framework is used only in that the dimensionality of the input to the neural network used for evaluation of the heuristic function, i.e. the players' ranges, public cards and betting sizes, is reduced into 1000 clusters (see Fig. 3.2).

Although it does restrict the number of actions by grouping betting sizes, since DeepStack does not need to explicitly remember action histories, no information loss occurs due to the action grouping - since opponent's action is not required in order to compute opponent's counterfactual values, there is no need to translate their action into corresponding group.

3.2 Continual resolving

3.2.1 Introduction

At any moment, the CFR algorithm mentioned in Chapter 2 stores mixed strategy for the entire game. Because it is impossible to store the entire mixed strategy of a HUNL game, DeepStack uses an approach of *continual resolving*, which is a process of forgetting the strategy used to reach particular game-state after taking an action and then reconstructing (the relevant part of) the strategy from information of

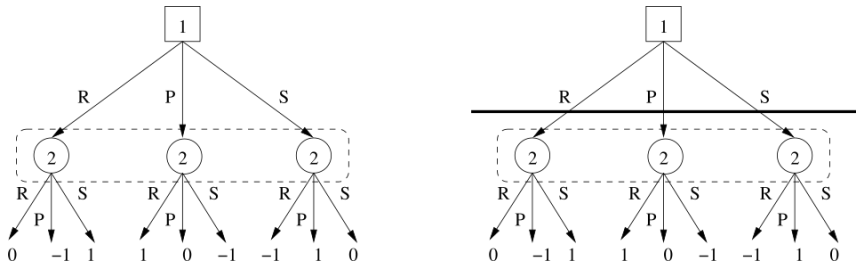


Figure 3.3: On the left, the the game of rock paper scissors in extensive form. On the right, the game of rock paper scissors separated into trunk and a subgame. [4]

constant size to the degree that allows computation of ϵ-Nash equilibrium using this process (see Fig. 3.1).

In games with perfect information, game state alone is, in principle, sufficient to find the ideal strategy. In imperfect information games, however, isolating subgames is not as trivial, because game history and opponent's moves in particular may provide valuable insight into their hidden information.

Furthemore, in a perfect information game, a subgame is defined simply as a subtree rooted at any node. In perfect information games, such definition is not possible, since it could exclude a part of an information set. Let us therefore introduce a definition for a subgame in imperfect information games.

Definition 16 ([4])**.** *Let $H_{p'}(h)$ be the sequence of p''s information sets player $p' \neq p$ encounters during game history h, along with p''s actions. An augmented information set $I_{p'}$ is given by*

$$I_{p'}(h) = I_{p'}(j) \iff H_{p'}(h) = H_{p'}(j)$$

for histories h, j, where $P(h) = P(j) = p$.

Intuitively, this means that two histories (at the end of which player p acts) belong to the augmented information set of p' if and only if they can not be distinguished by p'.

Definition 17 ([17])**.** *An imperfect information subgame is a forest of trees, closed under both the descendant relation and membership within augmented information sets for any player.*

Using this definition, all the states that can be reached with the other players observing the history that was actually observed - the augmented information sets - are included at the root of the subgame.

To illustrate the issue of subgame isolation further, consider a game of rock-paper-scissors (see Fig. 3.3). Suppose the p_1's strategy was solved for Nash equilibrium, so all their actions have probability of $\frac{1}{3}$. In p_2's subgame, all actions have expected utility of 0. Choosing any one of them, however, will lead to increased exploitability of the combined strategies. Suppose p_2 chooses to play rock with value 0 in equilibrium - p_1's counterfactual values for rock, paper and scissors are 0, 1, -1 in the trunk game for rock, paper and scissors respectively - so they will get value of -1 if player one decides to play paper.

This suggests what information p_2 needs in order to resolve their subgame properly. They have to find a strategy, where the opponent's best response counterfactual values for the original strategy are lower than or equal to their values in the combined strategy. Therefore, best response counterfactual values need to be maintained for all information sets at the root of the resolved subgames. [4]

In other words, let a whole game strategy σ, subgame S and information sets in its root R, a new resolved strategy for the subgame σ^S and combined strategy σ' be given, and assume, without loss of generality, that p_1's subgame is being resolved. Then, if p_2's expected payoff in σ' does not increase

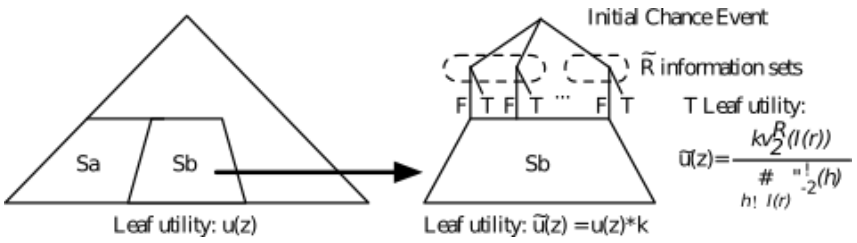


Figure 3.4: Subgame gadget construction. [4]

relative to σ, the exploitability of σ' is no higher than the exploitability of σ. The following implication can be used in order to show this sufficient condition for non-increasing exploitability is satisfied:

$$(\forall I \in R)\, v_2^{\langle \sigma'_1, CBR(\sigma'_1)\rangle}(I) \leq v_2^{\langle \sigma_1, CBR(\sigma_1)\rangle}(I) \implies (\forall I \in R)\, u_2^{\langle \sigma'_1, CBR(\sigma'_1)\rangle}(I) \leq u_2^{\langle \sigma_1, CBR(\sigma_1)\rangle}(I). \tag{3.1}$$

Proof. [4] □

DeepStack uses a variant of the vanilla CFR algorithm called CFR-D (where D stands for decomposition), which is, historically, the first method of subgame refinement which achieves this. It was one of the first algorithms successfully applying decomposition - technique famously useful in perfect information games - to the domain of imperfect information games while providing theoretical quarantees about quality of the solution. [4]

3.2.2 CFR-D and strategy reconstruction

In CFR-D, the game tree is first split into a trunk and subgames, and the goal is to find an approximation of Nash equilibrium for the trunk strategy, which is, at the beginning, initialized to random. In each iteration, subgames are solved using CFR, one after another, given the fixed trunk strategy, which is updated after each solution using the found counterfactual value at the root of the subgame. During this process, average counterfactual values are maintained at the roots of the subgames to be used for solving the subgame. The average of strategies found across iterations converges to Nash equilibrium. [4]

CFR-D saves space otherwise necessary for storing the mixed strategy. It comes at the price of increase in CPU time [4], which is a viable tradeoff in cases like HUNL, where it is infeasible to store the complete strategy.

In order to resolve subgame S, the *subgame gadget* meant to resolve for p_1's strategy, is constructed as follows.

A new initial chance node which leads to another new node corresponding to each state in each information set of p_2 in the original game is put at the beginning of the new subgame (see Fig. 3.4). In each of these nodes, p_2 has a binary choice of either F (follow through) or T (terminate). After T, the game ends, after F, the game continues as in the original subgame. In other words, each state $r \in R$ is transformed into three states from $\tilde{R}$, the subgame gadget:

- a chance node $\tilde{r} \in \tilde{R}$,
- a terminal state (leaf node) $\tilde{r} \cdot T \in \tilde{R}$,
- a follow through state $\tilde{r} \cdot F \in \tilde{R}$ identical to $r \in R$.

The chance node's probabilities are set to

$$\pi(\tilde{r}) = \pi_{-2}^{\sigma_1}(r) / \sum_{r' \in R} \pi_{-2}^{\sigma_1}(r')$$

and leaf utilities are normalized accordingly as

$$\tilde{u}_2(\tilde{z}) = u_2(z) \sum_{r' \in R} \pi^{\sigma_1}_{-2}(r'),$$

which implies that

$$\tilde{u}_2(I \cdot T) = v_2^{\langle \sigma_1, CBR_2(\sigma_1) \rangle}(I).$$

Intuitively, an option to continue playing, or terminating and earning the trunk counterfactual value is offered to p_2. This means that

$$\tilde{u}_2(I) \geq v_2^{\langle \sigma_1, CBR_2(\sigma_1) \rangle}(I).$$

Then, because values $v_2^{\langle \sigma_1, CBR_2(\sigma_1) \rangle}$ were constructed as best response values for p_2, we have

$$\tilde{u}_2(I) \leq v_2^{\langle \sigma_1, CBR_2(\sigma_1) \rangle}(I),$$

so $v_2^R = \tilde{u_2}(I)$ in an equilibrium and therefore, for a Nash strategy of the subgame $\tilde{\sigma}'$ (recall Definition 15),

$$\tilde{u}_2^{\langle \tilde{\sigma}'_1, CBR_2(\tilde{\sigma}'_1) \rangle}(I \cdot F) \leq v_2^{\langle \sigma_1, CBR_2(\sigma_1) \rangle}(I),$$

which, translated to the original game strategy σ', means

$$v_2^{\langle \sigma'_1, CBR_2(\sigma'_1) \rangle}(I \cdot F) \leq v_2^{\langle \sigma_1, CBR_2(\sigma_1) \rangle}(I).$$

Hence, if this subgame gadget is solved, for example using CFR, then the antecedent of the implication (3.1) is satisfied. [4]

3.2.3 Continual resolving in DeepStack

On any given player's turn, a local search procedure is invoked, which reconstructs the future strategy, updates it via regret matching (see Chapter 2), and selects the optimal action (and discards the found strategy) using only the following two pieces of information:

- acting player's *range* at the public state (probability distribution over possible hands given that the public state is reached),
- opponent's *counterfactual values* (upper bound on reward achievable by the opponent from the current public state).

The reason for maintaining the counterfactual values was explained in 3.2.1. Acting player's range is used to summarize their hidden information and, consequently, their expected utility at the and of the game. In other words, the range plays similar values to the probabilities of reaching given state π in the CFR algorithm in Algorithm 1. Opponent's range does not need to be maintained - it is recomputed during each resolve (see 3.2.2).

This information is sufficient to compute a strategy at any given public state using CFR. After taking an action, these values are updated based on the acting player as follows:

Own action: replace the opponent counterfactual values with the average of values achieved by the opponent during the re-solve after the action was taken, and update own range using the re-solved strategy and Bayes rule.

Opponent's action: no actual change is required to either the opponent's counterfactual values or own range.

New cards dealt: replace the opponent's counterfactual values with those computed for this chance action in the last re-solve. Our range is updated by giving zero to hands in the range which are not achiavable given the new cards.

Note that opponent's action is not used in this update and is therefore irrelevant for the computation of Nash equilibrium. Although the action history is relevant for computing strategy in imperfect information games as it might reveal information about the opponents strategy, it is not necessary to store it explicitly for computation of Nash equilibrium. This is an important step which makes solving large imperfect information games possible.

3.3 Look-ahead heuristic

Another important part of reducing the size of the game state is limiting the depth of any considered sub-game. Because, in perfect information games, the value of any game node is fully determined by subtree rooted at the node, an equivalent game can be constructed by replacing such node by a leaf node with value found by a subtree search. In imperfect information games, however, such replacement using a local search is not possible, since node's value may generally depend on values of nodes outside the subtree. In other words, a decision can not be made without considering other, even unmade, decisions, because probabilities of the current game state are affected by those decisions.

In DeepStack, this challenge of non-locality is overcome by a counterfactual value approximation function, substituting player's "intuition", learned using a deep neural network (see Fig. 3.2). Three neural networks are trained separately for first three (pre-flop, flop and turn) turns. For the last (river) turn, the game sub-tree is small enough to be solved entirely. Each of the networks takes the current public information (i.e. the revealed cards and current pot size) and both players' estimated ranges, i.e. 1326 dimensional (52*51/2) vectors and outputs vectors of counterfactual values for each hand.

In other words, the input itself is a representation of a poker game, consisting of

- public game state - revealed public cards and betting sizes,
- both players' ranges - probability distributions over possible cards dealt to both players,

The output is a vector of approximate values of holding certain pairs of cards.

The networks were trained by solving randomly generated game instances. The turn (last turn) network was trained using 10 million examples solved entirely. The flop (second turn) network was trained using 1 million examples and solved with depth-limit using the turn network.

3.4 Pseudocode

Pseudocode Algorithm 2 provides concise summary of the DeepStack algorithm. The `Resolve()` method performs a single step of continual resolving. In comparison to Algorithm 1, we can see that the `Values()` method is the analogue of the loop computing the vectors of counterfactual values v_1^t, v_2^t on lines 15 to 21 in Algorithm 1, whereas the `UpdateSubtreeStrategies()` method does the regret matching on lines 23 to 29. The probabilities π_1, π_2 partly play the role of the range vectors in the computation of counterfactual values.

DeepStack does not keep track of opponent's ranges. Instead, `RangeGadget()` method is used to reconstruct opponent's range r_2^t over T CFR iterations based on average counterfactual values v_2 passed

from previous CR iteration and counterfactual values of the subgame $v_2^t(S)$ computed in the current CFR iteration. This reconstruction of r_2^t is based on the technique of creating a subgame gadget from the CFR-D algorithm described in 3.2.2, and is one of the reasons for maintaining opponent's counterfactual values.

The range reconstruction requires larger number of iterations T than CFR alone would have. In order to raise precision of the result, only the last values are accounted for in the averages. [16]

Algorithm 2 Depth-limited continual re-solving. [16]

INPUT: Public state S, player range $\boldsymbol{r}_1$ over our information sets in S, opponent counterfactual values $\boldsymbol{v}_2$ over their information sets in S, and player information set $I \in S$
OUTPUT: Chosen action a, and updated representation after the action $(S(a), \boldsymbol{r}_1(a), \boldsymbol{v}_2(a))$

function RE-SOLVE($S, \boldsymbol{r}_1, \boldsymbol{v}_2, I$)
  $\sigma^0 \leftarrow$ arbitrary initial strategy profile
  $\boldsymbol{r}_2^0 \leftarrow$ arbitrary initial opponent range
  $R_G^0, R^0 \leftarrow \mathbf{0}$ ▷ Initial regrets for gadget game and subtree
  for $t = 1$ to T **do**
    $\boldsymbol{v}_1^t, \boldsymbol{v}_2^t \leftarrow$ VALUES($S, \sigma^{t-1}, \boldsymbol{r}_1, \boldsymbol{r}_2^{t-1}, 0$)
    $\sigma^t, R^t \leftarrow$ UPDATESUBTREESTRATEGIES($S, \boldsymbol{v}_1^t, \boldsymbol{v}_2^t, R^{t-1}$)
    $\boldsymbol{r}_2^t, R_G^t \leftarrow$ RANGEGADGET($\boldsymbol{v}_2, \boldsymbol{v}_2^t(S), R_G^{t-1}$)
  end for
  $\overline{\sigma}^T \leftarrow \frac{1}{T}\sum_{t=1}^{T}\sigma^t$ ▷ Average the strategies
  $a \sim \overline{\sigma}^T(\cdot|I)$ ▷ Sample an action
  $\boldsymbol{r}_1(a) \leftarrow \langle \boldsymbol{r}_1, \sigma(a|\cdot)\rangle$ ▷ Update the range based on the chosen action
  $\boldsymbol{r}_1(a) \leftarrow \boldsymbol{r}_1(a)/\|\boldsymbol{r}_1(a)\|_1$ ▷ Normalize the range
  $\boldsymbol{v}_2(a) \leftarrow \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{v}_2^t(a)$ ▷ Average of counterfactual values after action a
  return $a, S(a), \boldsymbol{r}_1(a), \boldsymbol{v}_2(a)$
end function

function VALUES($S, \sigma, \boldsymbol{r}_1, \boldsymbol{r}_2, d$) ▷ Gives the counterfactual values of the subtree S under σ, computed with a depth-limited lookahead.
  if S is terminal **then**
    $\boldsymbol{v}_1(S) \leftarrow U_S\boldsymbol{r}_2$ ▷ Where U_S is the matrix of the bilinear utility function at S,
    $\boldsymbol{v}_2(S) \leftarrow \boldsymbol{r}_1^{\mathsf{T}}U_S$ $U(S) = \boldsymbol{r}_1^{\mathsf{T}}U_S\boldsymbol{r}_2$, thus giving vectors of counterfactual values
    return $\boldsymbol{v}_1(S), \boldsymbol{v}_2(S)$
  else if d = MAX-DEPTH **then**
    return NEURALNETEVALUATE($S, \boldsymbol{r}_1, \boldsymbol{r}_2$)
  end if
  $\boldsymbol{v}_1(S), \boldsymbol{v}_2(S) \leftarrow \mathbf{0}$
  for action $a \in S$ **do**
    $\boldsymbol{r}_{\text{Player}(S)}(a) \leftarrow \langle \boldsymbol{r}_{\text{Player}(S)}, \sigma(a|\cdot)\rangle$ ▷ Update range of acting player based on strategy
    $\boldsymbol{r}_{\text{Opponent}(S)}(a) \leftarrow \boldsymbol{r}_{\text{Opponent}(S)}$
    $\boldsymbol{v}_1(S(a)), \boldsymbol{v}_2(S(a)) \leftarrow$ VALUE($S(a), \sigma, \boldsymbol{r}_1(a), \boldsymbol{r}_2(a), d+1$)
    $\boldsymbol{v}_{\text{Player}(S)}(S) \leftarrow \boldsymbol{v}_{\text{Player}(S)}(S) + \sigma(a|\cdot)\boldsymbol{v}_{\text{Player(S)}}(S(a))$ ▷ Weighted average
    $\boldsymbol{v}_{\text{Opponent}(S)}(S) \leftarrow \boldsymbol{v}_{\text{Player}(S)}(S) + \boldsymbol{v}_{\text{Opponent(S)}}(S(a))$
    ▷ Unweighted sum, as our strategy is already included in opponent counterfactual values
  end for
  return $\boldsymbol{v}_1, \boldsymbol{v}_2$
end function

function UpdateSubtreeStrategies($S, \boldsymbol{v}_1, \boldsymbol{v}_2, R^{t-1}$)
  for $S' \in \{S\} \cup$ SubtreeDescendants(S) with Depth(S') < MAX-DEPTH **do**
    for action $a \in S'$ **do**
      $R^t(a|\cdot) \leftarrow R^{t-1}(a|\cdot) + \boldsymbol{v}_{\text{Player}(S')}(S'(a)) - \boldsymbol{v}_{\text{Player}(S')}(S')$
      ▷ Update acting player's regrets
    end for
    for information set $I \in S'$ **do**
      $\sigma^t(\cdot|I) \leftarrow \frac{R^t(\cdot|I)^+}{\sum_a R^t(a|I)^+}$ ▷ Update strategy with regret matching
    end for
  end for
  return σ^t, R^t
end function

function RangeGadget($\boldsymbol{v}_2, \boldsymbol{v}_2^t, R_G^{t-1}$) ▷ Let opponent choose to play in the subtree or receive the input value with each hand
  $\sigma_G(\text{F}|\cdot) \leftarrow \frac{R_G^{t-1}(\text{F}|\cdot)^+}{R_G^{t-1}(\text{F}|\cdot)^+ + R_G^{t-1}(\text{T}|\cdot)^+}$ ▷ F is Follow action, T is Terminate
  $\boldsymbol{r}_2^t \leftarrow \sigma_G(\text{F}|\cdot)$
  $\boldsymbol{v}_G^t \leftarrow \sigma_G(\text{F}|\cdot)\boldsymbol{v}_2^{t-1} + (1 - \sigma_G(\text{F}|\cdot))\boldsymbol{v}_2$ ▷ Expected value of gadget strategy
  $R_G^t(\text{T}|\cdot) \leftarrow R_G^{t-1}(\text{T}|\cdot) + \boldsymbol{v}_2 - v_G^{t-1}$ ▷ Update regrets
  $R_G^t(\text{F}|\cdot) \leftarrow R_G^{t-1}(\text{F}|\cdot) + \boldsymbol{v}_2^t - v_G^t$
  return $\boldsymbol{r}_2^t, R_G^t$
end function

Chapter 4

Continual resolving in PAWS

We decided to take fundamentally new approach to PAWS. In order to exploit the combination of counterfactual regret minimization (Chapter 2) and continual resolving (Chapter 3), the PAWS domain must be modelled as an extensive game, not as Stackelberg security game. In other words, it is not the case anymore that the attacker first observes defender's strategy and then acts. Instead, the attacker attacks a target without the knowledge of defender's strategy.

Our goal, however, is to construct game model with solutions equivalent to those of the original PAWS with fully rational attacker and no payoff uncertainties. Fortunately, it is the case that in zero-sum two player security games[1], all Nash and strong Stackelberg equilibria are mutually equivalent, with the additional property that that any player's equilibrium strategy can be paired with any other player's equilibrium strategy and achieve the same payoffs for both players. [12]

4.1 Game model

The input to our algorithm is

- directed weighted game graph with
  - edges, where
    - nodes represent KAPs, and
    - edges represent routes between them, weighted by animal density on the route and distance between corresponding KAPs.
  - one of the nodes is marked base,
- grid, in which the conservation area is located (so each edge lies in a single grid cell),
- precomputed distance matrix for evaluation heuristics (see Section 4.2), and
- parameters, such as number of CFR iterations, maximum subtree depth, and route length limit.

In compliance with the general SSG model described in Section 1.1, the attacker begins the game by choosing a single grid cell as a point of attack. Corresponding grid cell coordinates, however, are unknown to the defender. Starting at the base node, the defender then traverses the graph, being presented with a choice at each node of which neighboring node to visit next. Each node visit is considered a turn

[1] Generally, for all security games with the property that $(\forall s \in S_i)\ s' \subset s \implies s' \in S_i$, i.e. any subset of a schedule is also a possible schedule, it is the case that any defender's SSE strategy is also a NE strategy. [12]

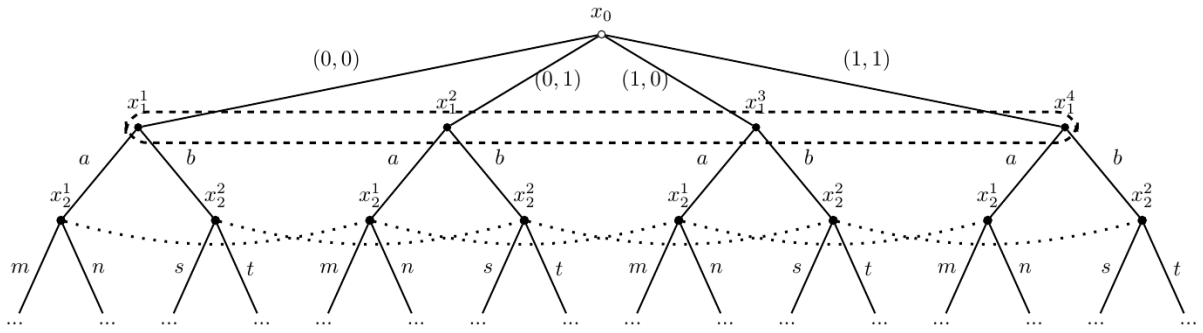


Figure 4.1: Our game model depicted as extensive game with 4 grid cells. The dotted lines delineate information sets.

in the game. By traversing a node of animal density r on grid cell not chosen by the attacker, they receive reward $-r$ and the attacker receives reward r. By traversing a node on grid cell chosen by the attacker, they receive reward r and the attacker receives reward $-r$.[2] No reward is received when an edge is traversed more than once, the total distance covered by the defender is limited by predefined route length limit, the route has to begin and end at the base node and the distribution of animal densities is known to both players.[3] Both players' goals are to accumulate maximum reward possible.

In terms of Definition 1, this indeed is an extensive game with imperfect information (see Fig. 4.1). Root node $I_1 = \{x_0\}$ represents attacker's choice of a grid cell, all its child nodes $I_1^1 = \{x_1^1, x_1^2, \dots\}$ represent the information set of defender's choices at the base node. Any path in any subtrees corresponding to nodes in I_1^1 (further referred to as the defender's subgames) from the root node to the leaf node represents a route in the original game graph with no repeated (oriented) edges. Nodes of depth d in each of defender's subgames are exactly the nodes reachable in d steps from the base node. Therefore, each information set contains at least k nodes, where k is the number of grid cells or more, if it is possible to reach corresponding node via multiple routes with the same sets of previously visited nodes.[4]

Since the attacker never responds to defender's actions, defender has no hidden information. The attacked grid cell is attacker's hidden information.

The output of our algorithm is a generated route sampled from the distribution over all possible routes given by ϵ-Nash equilibrium between attacker and defender strategies for the particular animal densities.

With the input, game mechanics and desired output in mind, let us informally draw the analogue to DeepStack terminology used in Chapter 3:

Node history: unordered list of visited nodes.

Route / Patrol route: ordered list of visited nodes, defining a trail in the game graph.

(Public) game state: patrol route (including the current node)

Defender's action: an edge between two KAPs.

Attacker's action: grid cell coordinates.

Defender's strategy: a probability distribution over neighboring edges for all information states.

[2] Extending the game rules to include reward uncertainty or reward assymetry may be focus of future work.

[3] If no such route exists, the corresponding game instance is considered invalid.

[4] This may be considered a form of imperfect recall which, however, has no effect on strategy and serves as a way to reduce the number of information sets.

Attacker's strategy: a probability distribution over grid cells.

Defender's counterfactual value: the highest (cumulative) payoff achievable by the defender from a game state given attacker's strategy, i.e. their couterfactual best response value (see Definition 13).

Attacker's counterfactual value: the highest payoff achievable by the attacker for all attacker's actions given a patrol route.

Defender's range: not necessary, since the attacker never responds to any defender's action.

Attacker's range: distribution over grid cells indicating defender's belief that the attacker attacked the particular grid cell.

Note that in Section 1.1 we mentioned that in the original PAWS game model, both defender and attacker strategies were distributions over grid cells, and the routes were computed retroactively. Our approach is computing the route directly. On the other hand, it does not consider payoff uncertainties and boundedly rational attacker (it assumes fully rational attacker).

Since attacker's only move is the first action of the game and they never respond to defender's actions, there is no reason for keeping defender's hidden information, i.e. their current position, as a distribution in a range vector variable. Therefore, this information is kept as a part of the public game state, simplifying the game model considerably.

As mentioned earlier, our goal is to emulate the original PAWS game, where the defender's observations are not taken into account for strategy computation. The nature of continual resolving, however, offers the possibility of updating attacker's range at each resolve step based on evidence of attack on the particular grid cell or lack thereof, and consider it for computing defender's strategy. A speculation of one way of realizing this in a another version of the algorithm could be as follows. Unlike in DeepStack, attacker's range is not be reconstructed after each resolve step, but maintained and updated based on observations made by the defender averaged over CFR iterations. Defenders knowledge of total animal density distribution will be used for the update. The distribution is initialized to proportional to the net animal density per grid cell. Then, if it revealed that the attacker did not attack a grid cell, the corresponding probability is set to 0 and the rest is normalized. If it is revealed the attacker did attack the grid cell, its probability is set to 1 and the rest is set to 0.

A pseudocode of the proposed algorithm incorporating CFR and continual resolving into PAWS with single attacked grid cell can be seen in Algorithm 9.

4.2 Evaluation function heuristic

A real-world test graph which we used for benchmarking consists of 553 nodes with mean distance of 986 m ranging from 100 m to 2.6 km, laid out on grid of 26x26 cells. The average shortest path between two nodes is 4.9 km with the maximum of 11.4 km. Because of such scope, it is impossible to store the entire defender's strategy - a distribution over all possible paths starting and ending at the base node - in memory.

To combat this issue, we will use continual resolving, described in Chapter 3, combined with a local approximation of the global strategy - instead of representing the strategy as a distribution over all possible routes, we remember and compute only distribution over possible actions at each resolve step.[5]

To find the strategy, we use the CFR algorithm described in Chapter 2 to iterate over unvisited (oriented) edges around the current node forming a game subtree, keeping in mind that

[5]Computing the distribution over all possible routes in a subtree is a possibility, but even this would, presumably, prove impractical at higher subtree depths.

1. rewards should not be awarded after second visiting of the same (unoriented) edge,

2. edges from which it is not possible to return to base due to remaining patrol distance should not be included, and

3. this distance should be computed with regards to edges already visited (and thus effectively removed from the graph) both as part of the route generated until the current node, and as part of the current CFR iteration.

In order to limit the depth of the searched subtree on any node, however, it is necessary to use an evaluation heuristic, which returns an approximation of the counterfactual values of any node given attacker's strategy.

Given the model explained in Section 4.1, it is obvious our evaluation function heuristic plays role similar to the one used in DeepStack. In our case, however, the values returned are not going to be learned over time by a neural network simulating problem instances, but computed by an algorithm attempting to approximate the solution quickly. To our knowledge, this is the first attempt considering this approach, and one of our ultimate goals is to compare the performance of learned and hand-crafted evaluators, or create such a neural network evaluator that will be able to generalize from instances evaluated by our hand-crafted evaluator.

In analogy to the DeepStack evaluation heuristic mentioned in Section 3.3, the heuristic considers the following input:

1. public game state - patrol route generated until the current node,

2. attacker's range - distribution over grid cells

and outputs a vector of approximate values over actions for each player. Our case is simplified by the fact that, unlike in poker, there is no defender's hidden information and no betting.

At the end of the game, the patrol route is closed (starting and ending in base node). The end values of the game are easily computed post-facto by weighting animal densities by attacker's range, traversing the route and accumulating resulting edge rewards.

Very important consideration is that the heuristic should be fast. Let d be the depth at which the evaluation heuristic is invoked, e be the average number of edges per node and T number of CFR iterations, N route length and t the average time spent in computation of a single run of the heuristic. Then the overall net time t_o spent computing the heuristic per single run of the algorithm is

$$t_o \approx N * T * t * e^d.$$

Even for relatively conservative values of $N = 10$, $T = 4$, $t = 1$ s, $e = 4$ and $d = 3$, we get $t_o \approx 2560$ s = 42.67 min.

This also suggests that the heuristic should perform well even on invokations at lower depth.

4.3 Orienteering problem

A well-known problem similar to the one our heuristic should solve, which has been attracting large amount of research interest due to its many applications, exists.

The name *orienteering problem* (OP) originates from the sport game of orienteering, in which individual competitors start at a base and then visit specified checkpoints, accrueing some amount of points at each, with the goal of maximizing the amount of points collected in a given time limit.

OP can seen as a loose combination beteen the Knapsack Problem and Travelling Salesman Problem, where not all vertices have to be visited (but still each at most once) and the goal is to maximize total collected score, instead of minimizing the travel distance. In the following, we will formalize the problem and prove NP completeness.

Let the following be given:

$\{1, \dots N\}$ a set of N nodes (vertices), where two nodes are marked as starting and ending node,

$\{S_i\}_{i=1}^{N}$ set scores for each node,

$\{t_{ij}\}_{i,j=1}^{N}$ set of euclidean distances between nodes satisfying triangle inequality (verify, prob. not necessary) and $t_{ij} = t_{ji}$,

T_{max} the time limit.

It is easy to see that the above corresponds to a weighted undirected complete graph without loops. Using this notation, the OP can be stated as the following integer problem [32]

$$\max \sum_{i=2}^{N-1} \sum_{j=2}^{N} S_i x_{ij}, \tag{4.1}$$

$$\sum_{j=2}^{N} x_{1j} = \sum_{i=1}^{N-1} x_{iN} = 1, \tag{4.2}$$

$$\sum_{i=1}^{N-1} x_{ik} = \sum_{j=2}^{N} x_{kj} \leq 1, \tag{4.3}$$

$$\sum_{i=1}^{N-1} \sum_{j=2}^{N} t_{ij} x_{ij} \leq T_{max}, \tag{4.4}$$

$$2 \leq u_i \leq N; \qquad \forall i = 2, \dots, N, \tag{4.5}$$

$$u_i - u_j + 1 \leq (N-1)(1 - x_{ij}); \qquad \forall i, j = 2, \dots, N, \tag{4.6}$$

$$x_{ij} \in \{0, 1\}; \qquad \forall i, j = 1, \dots, N, \tag{4.7}$$

where $x_{ij} = 1$ if visit to node i is followed by visit to node j, $x_{ij} = 0$ otherwise, and u_i is the position of node i on the path.

The objective function (4.1) is to maximize the total collected score. Equation (4.2) ensures that the path starts at node 1 and ends at node N. Bounds (4.3) guarantee connectivity of the graph and that each node is visited at most once, while (4.4) represents the limited time budget. The last two are special constraints which are necessary to prevent so called subtours [15].

The definition mentioned above describes the vanilla version of the OP, however, there are multiple variations. In the following, only a few most notable examples of those are given - an exhaustive list would have been much longer.

In *team orienteering problem*, the goal is to find multiple tours maximizing collective score. In *orienteering problem with time windows*, each location is assigned a time interval in which it has to be visited. Combination of those two, *team orienteering problem with time windows* is discussed further in Section 4.3.2.3. *Generalized orienteering problem* considers a nonlinear objective function, meaning that certain combinations of visited locations can be rated higher or lower than the sum of their scores individually - inspired by the idea that in a tourist trip, some attractions should be visited together to be fully appreciated. In *capacitated team orienteering problem*, each location has associated demand, and the cumulative demand for each route must not exceed a limit.

4.3.1 NP-completness

Theorem 4 ([8])**.** *The OP is NP-hard.*

Proof. A problem H is NP-hard if and only if every problem L in NP can be reduced in polynomial time to H, i.e. if time complexity of H is constant, L can be solved using H in polynomial time. Formally, H is NP-hard if and only if $\forall H \in NP, L \propto H$. Therefore, to prove the OP is NP-hard, we have to show that the existence of a polynomially bounded algorithm for OP would imply existence of polynomially bounded algorithm for any (and therefore all) NP-complete problems. We will choose Travelling Salesman Problem (TSP) [15] for this purpose.

Suppose a polynomial algorithm for solving OP exists. That is, given $\{1, \dots N\}$, $\{S_i\}_{i=1}^{N}$, $\{t_{ij}\}_{i,j=1}^{N}$ and T_{max}, this algorithm would find the sequence of nodes maximizing total score in polynomial time. Consider an instance of TSP, asking the question: Given this specific set of nodes $\{1, \dots N\}$ and distances $\{t_{ij}\}_{i,j=1}^{N}$, is there a tour of length less than T through all the nodes? Set $S_i = 1 \forall i \in N$, $T_{max} = T$ and any node as both starting and ending nodes and solve as an instance of OP using our polynomial algorithm. If $\max \sum_{i=2}^{N-1} \sum_{j=2}^{N} S_i x_{ij} = N$, output yes, otherwise output no. □

This result suggests that the exact algorithm for solving the OP in polynomial does not exist, and indeed none was found. On the other hand, many fast and effective heuristics exist, giving accurate results in reasonable amount of time. These are the topic of the next subsection.

4.3.2 Solution approaches

Finding fast and accurate heuristics to solving OP (and its variants) is an active research area, coming to prominance lately due to the increase in computing power and many applications requiring low computation time, including military, routing of technicians, service delivery, and tourist trip planning [31]. Tens of approaches have been proposed, dating back to 1960's. In the following text are presented the ones we were able to acquire for use in our work.

4.3.2.1 S-Algorithm

Tsiligirides proposed the stochastic S-Algorihm in 1984 [30]. It is based on Monte Carlo random sampling method of generating a large number of paths and picking the one with maximum score. Each path is generated by inserting the initial node to the solution and subsequently iteratively selecting the next node to be added to the path from a list of nodes reachable with the remaining distance with probability

$$P_j = \frac{A_j}{\sum_{i=1}^{N} A_i},$$

where

$$A_j = \left(\frac{S_j}{t_{curr,j}}\right)^4 \qquad \forall j \in \text{unvisited nodes.}$$

The number of generated routes is suggested to be around 3000.

4.3.2.2 Evolutionary algorithm

Genetic algorithms (GA) are a family of optimization heuristics isnpired by the biological process of natural selection and evolution. In these algorithms, viable solutions are encoded into chromosomes representing a population to be evolved through generations. At each generation, parents are selected to

produce offspring using a crossover operator. The offspring is consequently mutated and pooled togother to select the following generation.

We will describe a genetic algorithm due to Tasgetiren [29], where chromosomes in encoded as ordered list of variable length of nodes visited in the route the chromosome represents.

First, a random initial population of size λ of routes is generated such that they satisfy T_{max} constraint. For each generation, two parents are selected - first by tournament selection of size two and second randomly - to produce offspring through process of *order crossover*. First parent selects and insertion point in its chromosome, second parent chooses a sublist from its chromosome and inserts it into first parent's chromosome. Then, duplicate nodes are removed and the size of the resulting chromosome is fitted to the size of the first parent. This is repeated until μ amount of offspring is produced. Therefore, the size of population is increased to $(\lambda + \mu)$ at the end of each generation.

Mutations consisting of add, omit, replace and swap operations are applied to randomly selected individuals from the offspring and tournament selection of size of two is applied in order to maintain population size of λ. This procedure is repeated until the stopping criterion is achieved.

Algorithm 3 Pseudocode for the genetic algorithm. [29]

```
Initialize population P of size λ
Evaluate λ individuals in P
while not termination do
    Select 2μ individuals from P
    Crossover individuals to produce μ offspirng
    Mutate some individuals in μ
    Add μ offspring to λ individuals in P
    Evaluate (λ + μ) individuals in P
    Select λ individuals from (λ + μ) individuals in P
end while
```

4.3.2.3 GRILS

Team Orienteering Problem (TOP) and OP with Time Windows (OPTW) are two well-known extensions of the OP. The former searches for multiple routes maximizing the total score, the latter allows for defining a time interval $[\text{Opening}_i, \text{Closing}_i$ for each node i in which its visit has to take place and visiting time T_i. Moreover, a variant which allows for multiple constraints aside from the one imposed by limited time budget is called Multi-Constraint OP.

GRILS (or ILS-GRASP) is an algorithm for solving Multi-Constraint Team Orienteering Problem with Multiple Time Windows (MCTOPMTW). It is very well performing algorithm with envisioned application in tourist tour planning [28]. It combines iterated local search [33] (ILS) and Greedy Adaptive Search Procedure [27] (GRASP) algorithms, both of which outperform long-standing best performing 5-step algorithm by Chao (1994) [5].

The **ILS** combines two steps: an insertion step and a shaking step.

The *insertion step* tries to add, one by one, new nodes to the route. Because of the time window constraints, it has to be verified, before inserting a new node to the tour, that visits scheduled after it will still satisfy their time window. To this end, ILS records MaxShift and Wait variables for each location already visited.

Wait is defined as the waiting time in case the arrival takes place before the opening of the time window:

$$\text{Wait}_i = \max(0, \text{Opening}_i - t_{curr,i})$$

MaxShift is defined as the maximum amount of time for which the arrival can be delayed without making the visit infeasible:

$$\text{MaxShift}_i = \min(\text{Closing}_i - \text{TimeVisited}_i, \text{Wait}_{i+1} + \text{MaxShift}_{i+1})$$

Shift, the total time consumption of adding an extra visit between two nodes, is limited by the sum of Wait and MaxShift of the end node:

$$\text{Shift}_j = t_{ij} + \text{Wait}_i + T_j + t_{jk} - t_{ik} \leq \text{Wait}_k + \text{MaxShift}_k$$

The unvisited candidate with the highest ratio S_i^2/Shift_i is selected for insertion. After the insertion, variables for unvisited nodes are updated.

The *shake step* is used to escape local optima. It takes in two parameters - how many consecutive visits are removed and starting node of the removal, and is called with different values for each tour, depending on its length. After the removal, the visit times are shifted to the beginning of the tour, if the time window allows it. If it does not, the scheduled visit times for the node and all consequent nodes stay the same.

The algorithm starts with empty tour and shake step parameters initialized to one. In a loop, first, the insertion step is applied until a local optima is reached. If it is the highest optima found, it is recorded and, subsequently, shake step is applied with varying parameters such that, during the entire procedure, every visit is removed at least once with high probability. This loop is repeated until no improvement is found for a certain number of times in a row - this number is also the only parameter of the algorithm.

The authors noted, as same as we do in Section 4.3.3, that 'It appeared that further increasing the maximum number of iterations without improvement does not significantly enhance the obtained results and only causes longer computation times' [33].

The **GRASP** (Greedy Randomized Search Procedure) [27] [26] performs a number of independent iterations, each constructing a randomized path and then modifying it in a process called path relinking, searching for local optima. The best route found is then returned as a solution.

Specifically, in the loop, it executes four proceudres: construct, local search, link to elites and update elite pool.

Algorithm 4 GRASP algorithm outline. [27]

```
1: while num. of iterations wihout improvement not exceeded do
2:     construct
3:     local search
4:     link to elites
5:     update elite pool
6: end while
7: return best found
```

The *construct procedure* takes in a single greediness parameter drawn on random from the interval between 0 and 1, representing the ratio of greediness to randomness. Similar to the the insertion step in ILS, starting from and empty solution, it iteratively inserts an unvisited location based on the ratio between its score and increase in duration of the route resulting by inserting the candidate node. Only candidates whose increase in duration is less than the remaining time are considered. The difference between the maximum and minimum ratios is multiplied by the greediness parameter, and only candidates whose ratio is higher than this threshold value are left, one is randomly picked and inserted into the constructed solution.

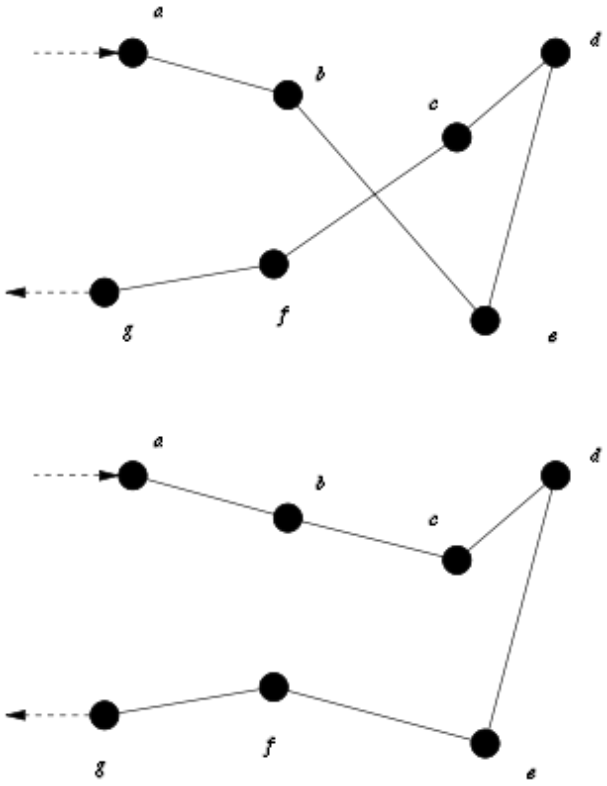


Figure 4.2: A visualization of the 2-opt procedure. (Courtesy: Pierre Selim)

The *local search procedure* alternates between reducing the total time of a solution and increasing its total score. Depending on the amount of remaining time, it either tries to exchange locations included in the solution with the lowest score with those not included by considering the cheapest insertions in terms of the ratio described in construct procedure, or call the regular insertion procedure. The move with the highest score increase is executed. In this way, it iteratively searches four different neighborhoods constructed using the 2-opt procedure [14] and stops when the found solution is optimal with respect to all the neighborhoods.

Local search is a process of iteratively modifying a solution to an optimization problem by exchanging it for another, more optimal solution in its search space neighborhood. In this case, the *local search procedure* first makes use of an algorithm called `2-opt`, which itself is well-known local search procedure which found its use in Travelling Salesman Problem heuristics. [14] Given a route, tt repeatedly applies a particular route altering procedure and stops when no possible alteration improves to route length - it found locally optimal solution. This procedure can be intuitively understood to be designed to make a self-crossing route not cross itself (see Fig. 4.2. However, it can shorten a route which does not cross over itself. [11]

Then, it applies the `swapping` procedure, which considers pairs of locations between tours and swaps them if an improvement in solution time is achieved.

Afterwards, `replace` procedure either inserts a new location to the tour (if enough time resources are available), or attempts to replace a low scoring locations in the current route with locations improving the overall score (or solution time, in case of a tie).

Finally, the `insert` procedure attempts to increase the score in the manner described in construct procedure description.

Algorithm 5 Local search procedure. [27]

```
1: while improvement do
2:     2-opt
3:     swap
4:     replace
5:     insert
6: end while
7: return local optimum
```

Link to elites procedure is used to remedy the independence of the different iterations by introducing a long-term memory component. It takes two solutions and visits the solutions on a path in the search space in between the two solutions and returns the best solution found on the path. It is performed each iteration for all pairs of combinations of

1. the solution constructed and improved in the current iteration, and

2. the elite pool of best solutions found so far,

except for a situation where a measure of their similarity $2n_{X\cap Y}/(n_X + n_Y)$, where $n_{X\cap Y}$ is the number of locations solutions X and Y have in common and n_X, n_Y are number of locations in solution X and Y respectively, is below a threshold.

Update elite pool procedure adds the best solution found by link to elites procedure into the pool of elites and removes the worst elite solution if the pool size limit is exceeded.

The way GRILS combines the ILS and GRASP algorithms is outlined in the following pseudocode:

Algorithm 6 The GRILS algorithm. [28]

```
1: for greed=0.89; greed>0.59; greed-=0.01 do
2:     while num. of iterations with no improvement < 100 do
3:         solution = GRASP(greed)
4:         if solution > best found then
5:             best found = solution
6:             num. of iterations with no improvement = 0
7:         else
8:             num. of iterations with no improvement += 1
9:         end if
10:        shake
11:    end while
12: end for
13: return best found
```

On 2.5 GHz Intel Xeon processor with 4 GB of RAM, the error against the optimal solution is 3.26% in under 1 second of computation time. More extensive testing suggets that hybridizing ILS with GRASP results in faster and more accurate algorithm than any of them in isolation.

4.3.3 Performance comparison

For our purpose, we were able to acquire and adapt the algorithms described in Subsection 4.3.2. The evolutionary algorihm is curtesy to Wes Carlson[6], the GRILS algorithm can be found on [1]. Both were used with the permission of the author. The Tsiligirides' algorithm is our custom implementation based on the paper [30].

In order to test performance, we converted the real world dataset described in Section 4.1 using data manipulation tools to the format suitable for each implementation and assigned a random score ranging from 0 to 100 to each vertex. The testing hardware was Intel Core i3 2.10GHz with 4 cores and 4 GB of RAM.

All implementations are in Python, which does not support multi-threaded parallelism[7]. The GRILS algorithm supports Message Passing Interface (MPI) and therefore allows for concurrency by spawning multiple processes. For the purpose of our testing, we decided to use only one process to not skew the results based on this advantage.

Different runtimes were achieved by manipulating the algorithm's free parameters, i.e. number of iterations for GRILS, population size and number of generations for evolutionary algorithm and number of randomly generated routes for Tsiligirides' algorithm.

Figure 4.3 shows the relationship between runtime (in seconds) and average achieved score of the three algorithms tested. Each algorithm was run 200 times, with parameters picked randomly from appropriate ranges. Vertical bars depict the standard deviation.

Although, all algorithms show similar performance, Tsiligirides seems to find slightly better solutions in the first two seconds of computation. Moreover, variance of its solutions' scores is consistently smaller. Since our purpose is to use the algorithm as a heuristic evaluation in continual resolving, and we are, therefore, concerned primarily with speed performance and the quality of solutions found in the first seconds of computation, Tsiligirides algorithm seems to be the best choice out of the algorithms tested.

4.3.4 Implementation

Because the definition of OP and our game model consider scores on graph nodes and animal densities on graph edges respectively, a modification has to be made if the OP is to be adopted as an evaluation heuristic.

Our solution is to modify the input game graph by inserting a new node in the middle of each edge, adding a score equivalent to the edge animal density in the original game graph to the middle node, and zero score to the remaining two, while halving the distance weight of the two new edges relative to the distance weight of the original edge (see Fig. 4.4).

Also, as mentioned in 4.2, the animal densities are multiplied by corresponding attacker's range before invocation of the heuristic.

4.4 Arc routing problem

The fact that in our game model, not each node, but each *edge*, is associated with reward (see Section 4.1) lends itself to a a different optimization problems called *arc routing problem* (ARP) and its variant *cycle trip planning problem* (CTPP). Both ARP and CTPP are similar to the OP, and can also be formulated as integer problems. In their case, however, the network is modelled as oriented graph, where

[6] carlsonwes@gmail.com

[7] This is due to a mechanism aimed to simplify Python interpretation so that only one thread can execute the script at any time, called Global Interpreter Lock (GIL). This limitation can be circumvented by spawning multiple system processes, each running its own interpreter.

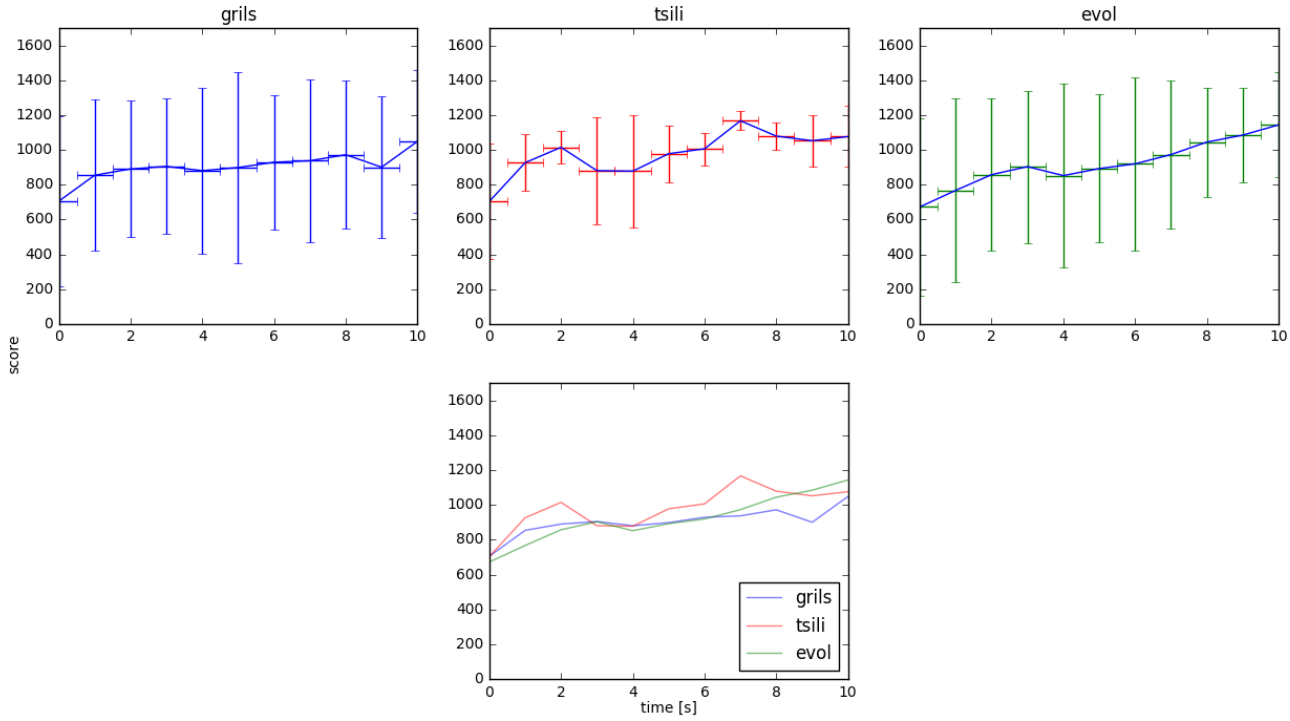


Figure 4.3: OP heuristics performance comparison.

Figure 4.4: On the left, an edge of the original graph. On the right, the input graph to the heuristic.

each arc a (oriented edge) has its inversely oriented counterpart $\overline{a}$. Moreover, cost c_a and profit p_a are associated with each arc a. Tour length is constrained by upper and lower bounds C_{max} and C_{min}. In case of ARP, no location can be visited twice, while in CTTP, no arc or its complement or starting position can be visited twice.

Although very well performing commercial algorithms tackling the ARP [9] and CTTP [34] exist, such as [2] using a variant of the GRASP algorithm mentioned in Section 4.3.2.3, or [34] using branch-and-cut method extended by a metaheuristic, the literature on ARP is relatively limited in comparison to the OP.

4.4.1 Approaches

Because converting edge profits into vertex profits as described Section 4.3.4 doubles the size of the input and presumably negatively impacts peformance, we decided to implement custom algorithms inspired by the approaches mentioned in 4.3.2 and compare their relative performance on real world dataset.

For our purpuses, we modified the definition of the AOP so that both locations and arcs can be visited multiple times, but score for corresponding edge visit is awarded only once.

4.4.1.1 S-algorithm

The original Tsiligirides' algorithm [30] from Section 4.3.2.1 can be easily modified to solve AOP instances by instead of iteratively constructing a solution by appending nodes from a list of reachable unvisited nodes with probability given by equation 4.3.2.1, creating a list of arcs and their counterparts and then appending reachable unvisited arcs a from the list with probability

$$P_a = \frac{a_{\text{score}}}{\text{dist}(\text{curr\_node}, a[0])},$$

along with a shortest path to the first node of the arc, to the solution. After each such insertion, all the arcs (along with their counterparts) encountered on the chosen shortest path are removed from the list of unvisited arcs. Although only unvisited reachable candidates are considered for insertion, the shortest path included with the insertion may contain already visited arcs and, therefore, the final solution may contain repeated arcs. Naturally, when computing score of the final solution, each visit is accounted for only once.

In the implementation, all reachable arcs are considered for insertion, but those already visited are assigned very small probability ϵ. This is to avoid situations where all edges necessary to complete the route have been visited.

We used Python and NumPy library to optimize vector operations. Rewriting the code into Cython did not result in substantial improvement in performance.[8] The algorithm is, however, inherently easily parallelizable, so significant speedup is expectable with parallelization.

4.4.1.2 GRASP

Because of its success in our performance comparison in Section 4.3.3 and inspired by its application to AOP in [2], we decided to implement a simplified version of GRASP algorithm [26] (described in Section 4.3.2.3) for AOP.

For a number of iterations, the algorithm starts by creating an empty solution containing only first and last nodes and picking a greediness parameter λ. Different ways of doing so are described later. The

[8] This may be because (most of) NumPy backend is already written in optimized C.

Algorithm 7 Tsiligirides' S-algorithm for the AOP.

```
1: function TSILI(iters)
2:     for iters do
3:         total_score ← 0
4:         rem_dist ← distance limit
5:         curr_node ← starting node
6:         unvisited ← all_arcs
7:         while True do
8:             feasible ← {a ∈ all_arcs | dist[curr_node, a[0]] + a_length + dist[a[1], end_node] <= rem_dist}
9:             if feasible = {∅} then
10:                break
11:            end if
12:            for a in feasible do
13:                if a ∈ unvisited then
14:                    P_a ~ (a_score / dist(curr_node, a[0]))^4
15:                else
16:                    P_a = ϵ
17:                end if
18:            end for
19:            next_arc ← select from feasible randomly with distribution P_a
20:            total_score += REWARD_FROM_PATH(curr_node, next_arc[0]) + a_score
21:            unvisited \ {(a[0],a[1]), (a[1],a[0]), path(curr_node, arc[0])}
22:            rem_dist -= dist(curr_node, next_arc[0]) + next_arc_length
23:            curr_node ← next_arc[1]
24:        end while
25:        if total_score > best_score then
26:            best_score ← total_score
27:        end if
28:    end for
29:    return best_score
30: end function
```

Algorithm 8 Simplified GRASP algorithm for AOP.

```
1:  function SEARCH(iters)
2:      best_solution ← empty solution
3:      for iters do
4:          solution ← empty solution
5:          λ ← SELECT_GREEDINESS()
6:          candidate_list, hmin, hmax ← GENERATE_NEIGHBORHOOD(solution)
7:          while candidate_list ≠ {∅} and solution.rem_dist > 0 do
8:              threshold ← hmin + λ*(hmax - hmin)
9:              restrict candidate_list by threshold
10:             insert random insertion from candidate_list to solution
11:             candidate_list, hmin, hmax ← GENERATE_NEIGHBORHOOD(solution)
12:             if solution.score > best_solution.score then
13:                 best_solution ← solution
14:             end if
15:         end while
16:     end for
17:     return best_solution
18: end function
```

function `generate_neighborhood(solution)` creates a list of candidate insertions (i.e. arcs) based on the current state of `solution` (already inserted arcs and remaining distance), computes a heuristic value associated with each candidate insertion as

$$\text{hval} = \frac{\text{arc score}}{\text{increase in solution cost after insertion}},$$

and outputs the list, along with the extremal heuristic values `hmin`, `hmax` in this list. Note that those extremal values are available only after constructing the whole list. A threshold is computed based on the extremal values `hmin, hmax` and greediness parameter λ as

$$\text{threshold} = \text{hmin} + \lambda * (\text{hmax} - \text{hmin}).$$

Then, only cadidates with heuristic value larger than `threshold` are left in the `candidate_list`. A random candidate from the restricted candidate list is inserted into the solution, and a new candidate list is created based on the updated solution. This process continues until either no valid insertions are generated (the candidate list is empty), or the solution exceeded its overall travelled distance (i.e. net cost) limit.

This formulation of the algorithm leaves open two questions:

1. How to genereate neighborhood of a solution?
2. How to select λ?

The canonical version of GRASP, when creating insertion candidate list, considers inserting new arcs between each two nodes in the current solution. This relatively broad definition of neighborhood means larger variance of found solutions and higher probability of escaping local optima. Because, for our purpose, we are concered primarily with the speed performance of the algorithm, we decided to modify the `generate_neighborhood()` method to pick random two nodes in the current solution and consider

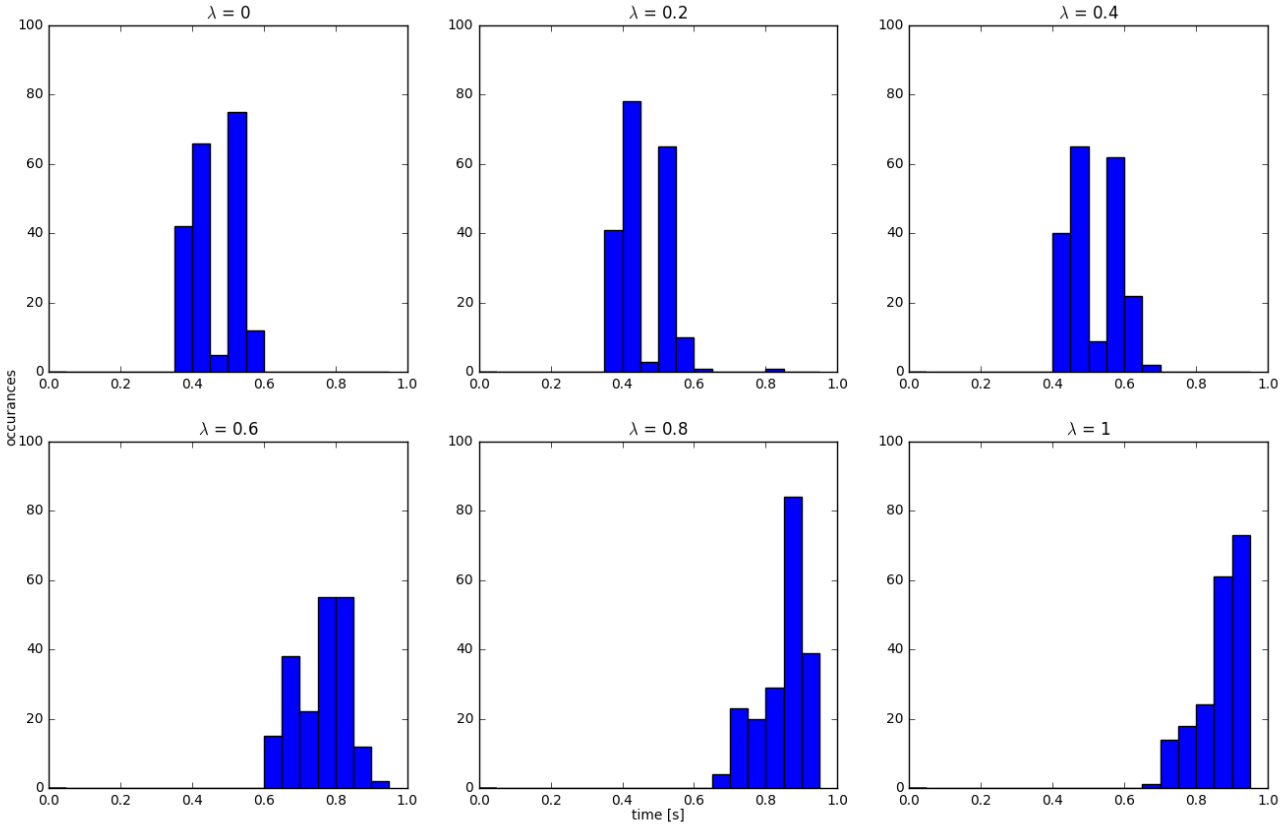


Figure 4.5: Distribution of runtimes for different values of λ.

insertions only in between these two nodes. This produced approximately 20-fold decrease in time per iteration.

The value $\lambda = 0$ corresponds to completely random, while the value $\lambda = 1$ corresponds to completely greedy, construction procedure. The greedier the construction, the stricter the candidate list restriction and when the construction is purely greedy, the candidate list at each step contains exactly one element (although the original version of GRASP deterministic in this case, our algorithm still maintains a degree of stochasticity due to the neighborhood generation procedure mentioned above). It is therefore reasonable to hypothesize that λ values close to 0 correspond to higher solution variance, while λ values close to 1 result in lower best average score, as the solution is becoming more random. The optimal value λ should therefore optimize tradeoff between maximizing the average solution values and minimizing their variance.

We tried running the algorithm 200 times with 20 iterations for 5 different values of λ. The results on Fig. 4.5 and Fig. 4.6 confirm those expectations. The average time increases with λ, and the values of λ of 0, resp. 1, indeed result in higher, resp. lower, variance. Moreover, $\lambda = 0.2$ seems to offer the best tradeoff between average solution score and variance for this particular instance. This is in keeping with the results in [26].

Because the optimal level greediness is likely to depend on given problem instance, we implemented

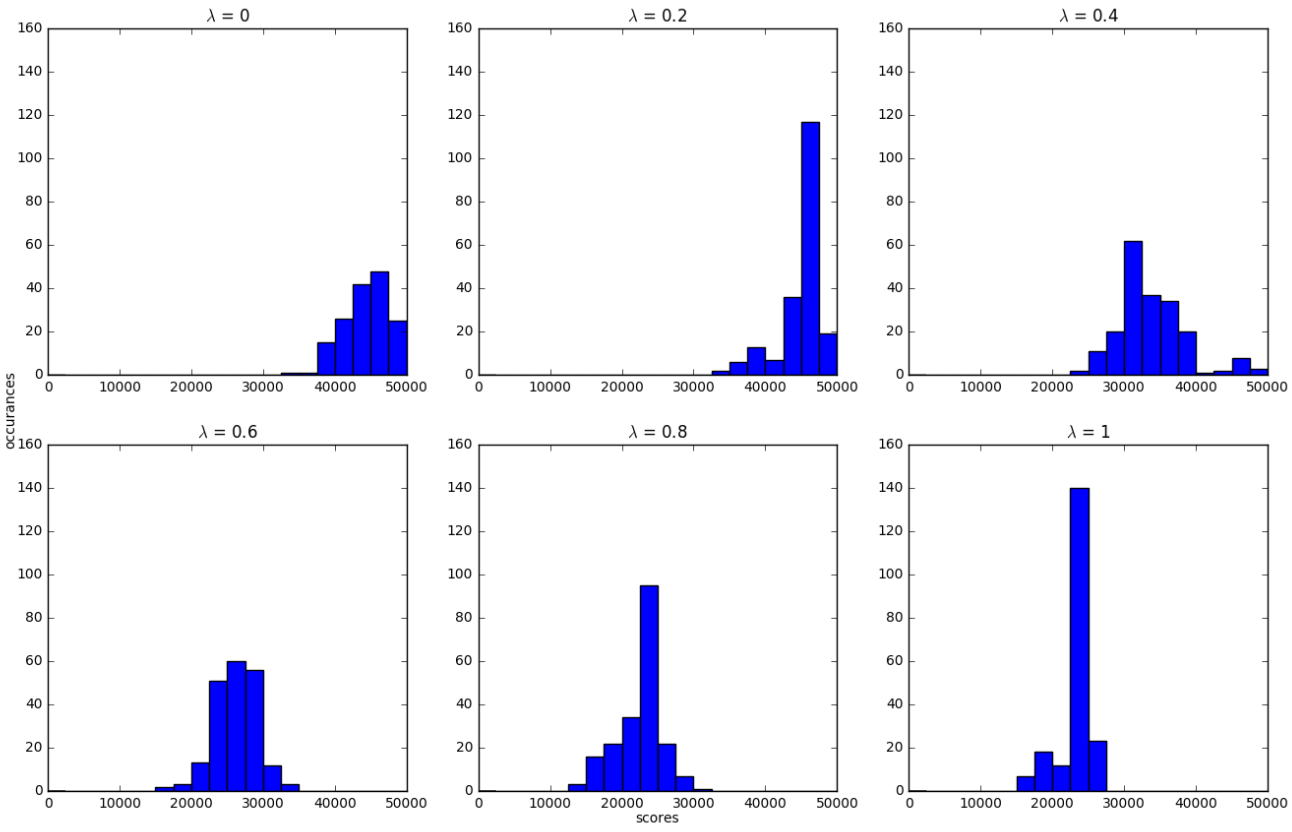


Figure 4.6: Distribution of scores for different values of λ.

λ	**Mean**	**Median**	**Variance**
0	46097.84	46532.77	20327896.70
0.2	45318.20	46778.22	9623616.19
0.4	33738.25	32591.80	23797530.67
0.6	26042.11	26473.71	8963460.47
0.8	22663.24	23149.05	9931249.23
1	22983.60	23414.85	4876972.44

Table 4.1: Score statistics for different values of λ.

λ	**Mean**	**Median**	**Variance**
0	2.31e-02 s	2.13e-02 s	1.02e-05 s^2
0.2	2.30e-02 s	2.11e-02 s	1.57e-05 s^2
0.4	2.58e-02 s	2.45e-02 s	1.16e-05 s^2
0.6	3.82e-02 s	3.89e-02 s	1.83e-05 s^2
0.8	4.24e-02 s	4.32e-02 s	1.22e-05 s^2
1	4.36e-02 s	4.45e-02 s	1.05e-05 s^2

Table 4.2: Time / iteration statistics for different values of λ.

a mechanism which continually adjusts the values of λ based on average solution score found using the value, similar to the approach known as *reactive* GRASP [25]. The interval $[0, 1]$ is divided into n classes (we chose $n = 10$), each class associated with numbers q_i , p_i and a_i, initialized to $q_i = 1$, $p_i = \frac{1}{n}$ and $a_i = 0$ for all $i \in \hat{n}$. The value p_i represents the probability of choosing given class, q_i is helper qunatity used to compute p_i, and a_i stores the average score found using given class.

Every time a new solution is found, first, the average score for λ used to find the solution is updated, and then the values q_i and p_i are updated respectively as follows:

$$q_i = \begin{cases} \frac{a_i}{\text{best\_score}} & \text{if } a_i > 0, \\ 1 & \text{else,} \end{cases}$$
$$p_i = \frac{q_i}{\sum_{i=1}^{n} q_i},$$

where `best_score` is the highest score found among all iterations.

Overall, we created the following versions of our GRASP algorithm for performance comparison:

Random: λ is selected at random from the interval $[0, 1]$.

Fixed: λ is fixed at 0.2.

Biased: λ is selected at random from the interval $[0, 0.2]$.

Reactive: λ changes dynamically based on the scores of solutions found using corresponding values of λ.

4.4.2 Performance comparison

Fig. 4.7 shows the dependence of each algorithm's solution score on number of iterations (the number of generated route sample in the Tsiligirides' S-algorithm and the `iters` parameter in Pseudocode 8),

while Fig. 4.8 shows the relationship between score and algorithms' runtime. Each algorithm was run 200 times for each number of iterations in $[1, 40]$. The midpoint of each bar is the average solution found for given number of iterations / given time interval, while the bar length is equal to twice the standard deviation.

Tsiligirides' algorithm clearly outperforms every version of GRASP, both in terms of average found solution scores and runtime. Moreover, its solutions show relatively low variance. Among the GRASP variants, the reactive version seems to find only slightly better solutions than random version. Both biased and fixed perform similarly, and better than reactive and random. Although biased gives slightly higher average solution score, the variance of fixed decreases noticably faster with increasing iterations.

One of the possible causes behind the superior performance of the Tsiligirides' algorithm could be that the heuristic evaluation of candidate insertion which takes into account only distance of the candidate instead of its effect on the current solution. This hypothesis was, however, disproven in our experiments: no improvement in GRASP peformance was measured after the change of the evaluation.

Another factor playing a role in the performance difference is that GRASP implementation is in pure Python, whereas Tsiligirides algorithm relies heavily on vector operations in optimized NumPy library.

The most reasonable explanation follows from another important distinction between the two algorithms: GRASP considers inserting new candidates on any position along the intermediate solution, and therefore has to search a larger space for the optimal insertion. This hypothesis also explains observed higher variance of GRASP solutions.

Name	**Mean**	**Median**	**Variance**
tsili	62398.81	62584.48	24382079.95
random	36026.21	37869.43	110548051.17
reactive	38518.62	40811.82	105156604.74
biased	43930.06	46387.21	58669582.34
fixed	42488.18	46454.43	64570171.53

Table 4.3: Score statistics for different algorithm versions.

Name	**Mean**	**Median**	**Variance**
tsili	2.40e-02 s	2.24e-02 s	5.83e-05 s^2
random	2.40e-02 s	2.24e-02 s	5.83e-05 s^2
reactive	2.40e-02 s	2.24e-02 s	5.83e-05 s^2
biased	2.40e-02 s	2.24e-02 s	5.83e-05 s^2
fixed	2.40e-02 s	2.24e-02 s	5.83e-05 s^2

Table 4.4: Time / iteration statistics for different algorithm versions.

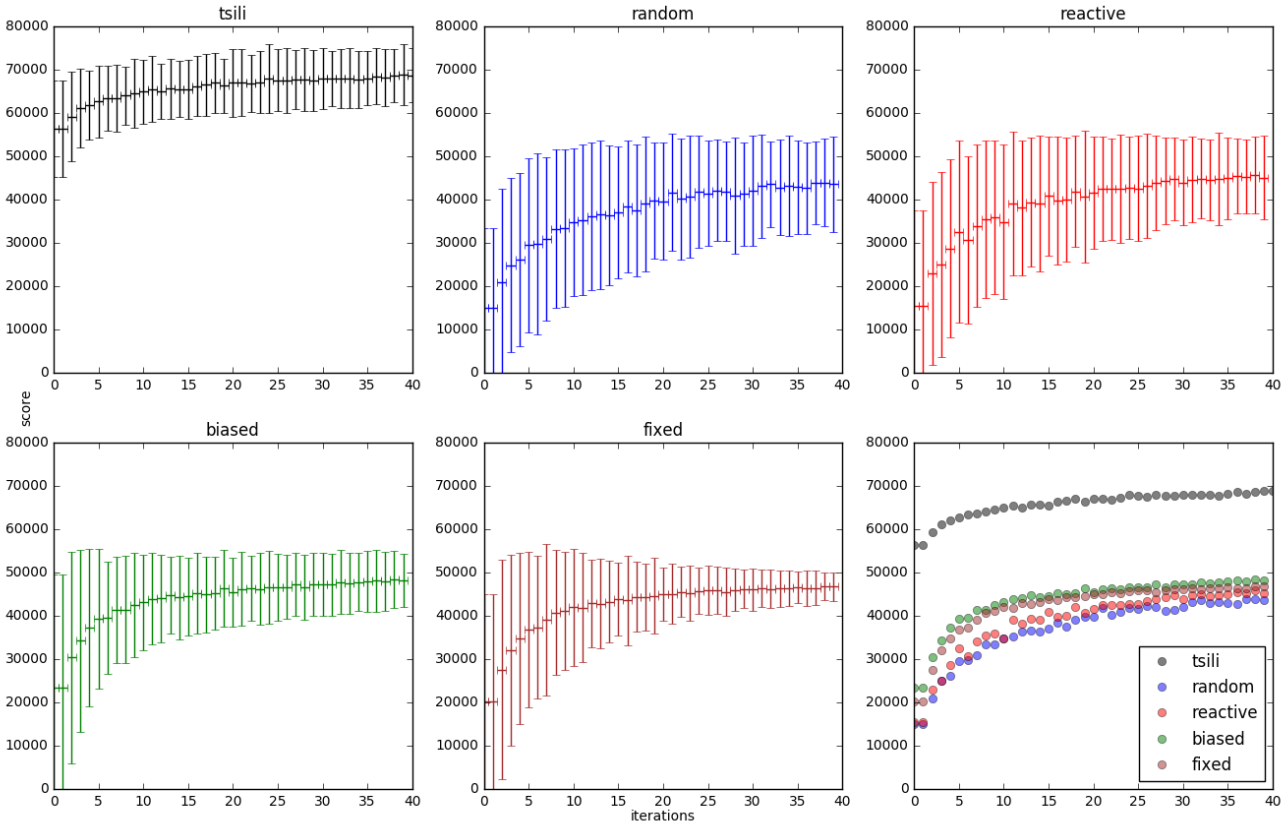


Figure 4.7: Performance comparison of our AOP algorithms - iterations against score.

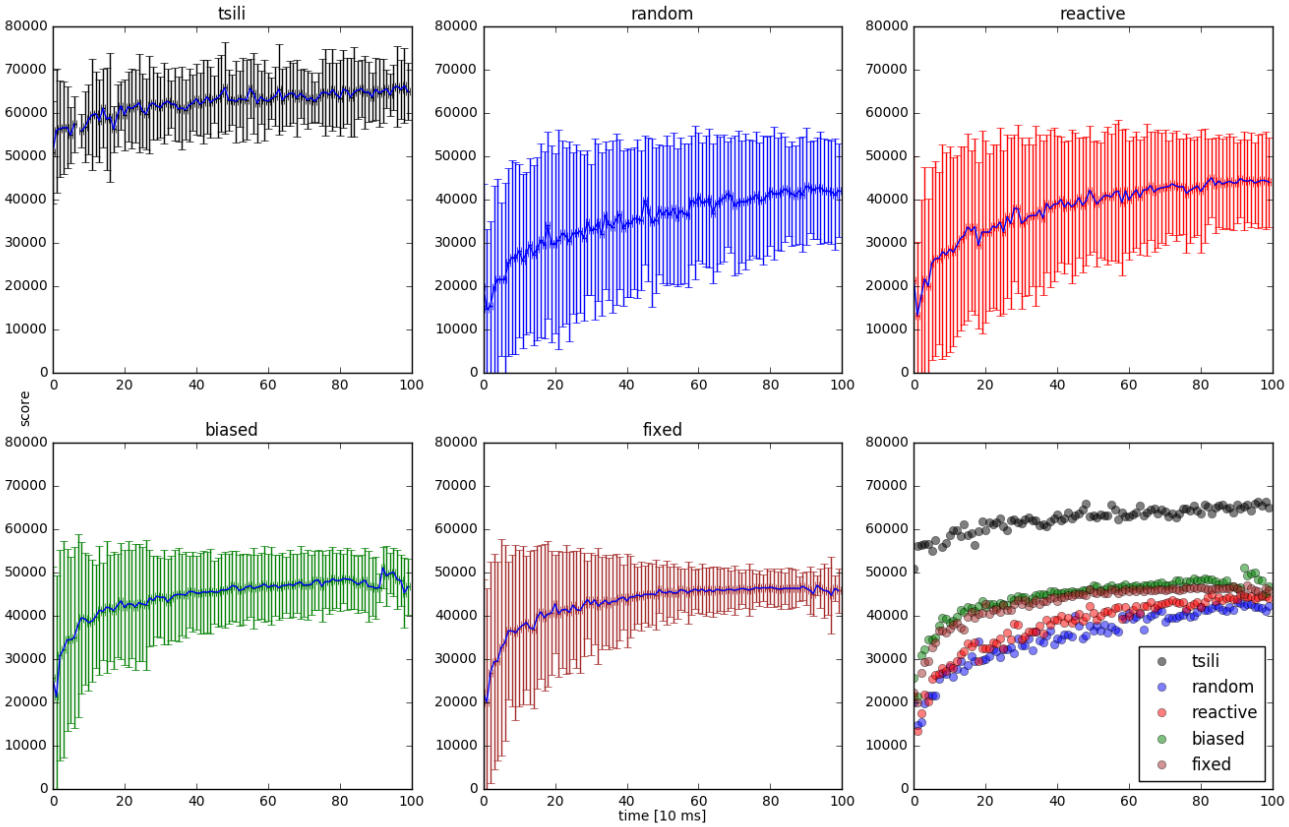


Figure 4.8: Performance comparison of our AOP algorithms - time against score. The time interval of 0 ms to 1000 ms into 100 buckets, means and standard deviations were computed for solutions in each bucket.

Algorithm 9 Proposed PAWS continual resolving algorihm.

function RESOLVE(ha, x, y, p_2, rem_dist)
  if rem_dist ≤ dist(a, base) or base reached **then**
    COMPUTE_VALUE_FROM_ROUTE(ha, x, y, rem_dist)
  end if
  $\sigma_2^0 \leftarrow 0$ ▷ Defender strategy - distribution over directed subgraph of depth max_depth
  $R_2^0 \leftarrow 0$ ▷ Defender regret table
  $\overline{\sigma}_2 \leftarrow 0$ ▷ Strategy accumulator to compute the average strategy
  $v_\sigma^0 \leftarrow 0$ ▷ Counterfactual values of nodes
  for t = 1 to T **do**
    $v_\sigma^t \leftarrow$ VALUES(ha, σ_2^{t-1}, v_σ^{t-1}, x, y, p_2, 0, rem_dist)
    $R_2^t, \sigma_2^t, \overline{\sigma}_2 \leftarrow$ REGRET_MATCHING(σ_2^{t-1}, v_σ^{t-1}, $\overline{\sigma}_2$)
  end for
  (a,c) ← $\max_{(a,c)\in A(ha)} \overline{\sigma}_2(a, c)$
  return RESOLVE(hac, x, y, $\sigma_2(b, c) * p_2$, rem_dist - dist(a,c))
end function

function VALUES(hab, σ_2, v_σ, x, y, p_2, d, rem_dist, subtree_visited)
  if rem_dist ≤ dist(a, base) or base reached **then**
    $v_\sigma(a, b) \leftarrow$ COMPUTE_VALUE_FROM_ROUTE(hab, x, y, rem_dist) **return** v_σ
  end if
  if d > max_depth **then**
    $v_\sigma(a, b) \leftarrow$ HEURISTIC(b, x, y, rem_dist)
    return v_σ
  end if
  for $(b, c) \in A(hab)$ **do**
    $v_\sigma \leftarrow$ VALUES(habc, σ_2, v_σ, x, y, p_2, d+1, rem_dist - dist(b,c))
  end for
end function

function REGRET_MATCHING(σ_2, v_σ, $\overline{\sigma}_2$)
  . . . perform regret matching using (2.3) . . .
end function

function COMPUTE_VALUE_FROM_ROUTE(hab, x, y, rem_dist)
  . . . compute value of route hab given attacked grid cell coordinates (x, y) and remaining distance rem_dist . . .
end function

Conclusion

In this paper, we laid the groundwork for our fundamentally new approach toward solving PAWS, giving an example of how state-of-the-art approaches to solving large imperfection games can be applied to the domain of green security games. We introduced the theoretical preliminaries of PAWS and DeepStack, outlined our approach of applying DeepStack to PAWS, researched the area of heuristics for the orienteering problem, implemented and compared several of the algorithms in terms of performance.

Future work might focus on implementing the CFR algorithm to compute Nash equilibria (optimal attacker and defender strategies given animal densities), compare it to the original algorithm used in PAWS, or modify it to introduce uncertainty into rewards.

Moreover, other approaches to approximating the evaluation function might be considered, such as graph convolutional networks [21], self-organizing maps, or mean field theory neural network technique [22].

Bibliography

[1] `https://github.com/priitj/grils-t`. Accessed: 2017-08-22.

[2] The planning of cycle trips in the province of East Flanders. *Omega*, 39(2):209–213, 04 2011.

[3] Bo An, Milind Tambe, O Ordonez, Eric Shieh, and Christopher Kiekintveld. Refinement of strong stackelberg equilibria in security games. In *In AAAI*, pages 587–593, 2011.

[4] Neil Burch, Michael Johanson, and Michael Bowling. Solving imperfect information games using decomposition. In *Proceedings of the Twenty-Eighth Conference on Artificial Intelligence (AAAI)*, pages 602–608, 2014.

[5] I-Ming Chao, Bruce L. Golden, and Edward A. Wasil. A fast and effective heuristic for the orienteering problem. *European Journal of Operational Research*, 88(3):475 – 489, 1996.

[6] Fei Fang, Thanh H Nguyen, Rob Pickles, Wai Y Lam, Gopalasamy R Clements, Bo An, Amandeep Singh, Milind Tambe, and Andrew Lemieux. Deploying PAWS: Field Optimization of the Protection Assistant for Wildlife Security. In *Proceedings of the Twenty-Eighth Innovative Applications of Artificial Intelligence Conference*, pages 3966–3973, 2016.

[7] Sam Ganzfried. *Computing Strong Game-Theoretic Strategies and Exploiting Suboptimal Opponents in Large Games*. PhD thesis, 2015.

[8] Bruce L. Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics (NRL)*, 34(3):307–318, 1987.

[9] Moshe Dror H. A. Eiselt, Gilbert Laporte. *Arc Routing: Theory, Solutions, and Applications*. Springer US, 2000.

[10] Sergiu Hart and Andreu Mas-Colell. A simple adaptive procedure leading to correlated equilibrium. *Econometrica*, 68(5):1127–1150, 2000.

[11] David Johnson and Lyle A. McGeoch. The traveling salesman problem: A case study in local optimization. 01 1997.

[12] Dmytro Korzhyk, Zhengyu Yin, Christopher Kiekintveld, Vincent Conitzer, and Milind Tambe. Stackelberg vs. nash in security games: An extended investigation of interchangeability, equivalence, and uniqueness. *Journal of Artificial Intelligence Research*, 41:297–327, 5 2011.

[13] Tomas Kroupa. http://staff.utia.cas.cz/kroupa/teaching.html. Accessed: 2017-11-17.

[14] Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965.

[15] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, October 1960.

[16] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker. pages 1–37, 2017.

[17] Matej Moravcik, Martin Schmid, Karel Ha, Milan Hladik, and Stephen J Gaukrodger. Refining Subgames in Large Imperfect Information Games. (iii):572–578, 2016.

[18] Todd W Neller and Marc Lanctot. An Introduction to Counterfactual Regret Minimization Regret in Games. pages 1–38, 2013.

[19] Thanh H Nguyen, Francesco M. Delle Fave, Debarun Kar, Aravind S Lakshminarayanan, Amulya Yadav, Milind Tambe, Noa Agmon, Andrew J Plumptre, Margaret Driciru, Fred Wanyama, and Aggrey Rwetsiba. Making the most of our regrets: Regret-based solutions to handle payoff uncertainty and elicitation in green security games. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9406, pages 170–191, 2015.

[20] Thanh H. Nguyen, Rong Yang, Amos Azaria, Sarit Kraus, and Milind Tambe. Analyzing the effectiveness of adversary modeling in security games. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, pages 718–724. AAAI Press, 2013.

[21] Alex Nowak, Soledad Villar, Afonso S. Bandeira, and Joan Bruna. A Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. pages 1–6, 2017.

[22] Carsten Peterson and James Anderson. Neural Networks and NP-complete Optimization Problems; A Performance Study on the Graph Bisection Problem. *Complex Systems*, 2(1):59–89, 1988.

[23] James Pita, Manish Jain, Janusz Marecki, Fernando Ordóñez, Christopher Portway, Milind Tambe, Craig Western, Praveen Paruchuri, and Sarit Kraus. Deployed armor protection: The application of a game theoretic model for security at the los angeles international airport. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems: Industrial Track*, AAMAS '08, pages 125–132, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[24] Martyn Plummer et al. Jags: A program for analysis of bayesian graphical models using gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, volume 124, page 125. Vienna, Austria, 2003.

[25] Marcelo Prais and Celso C Ribeiro. Reactive grasp: an application to a matrix decomposition problem in tdma traffic assignment. pages 1–21, 1998.

[26] Mauricio G.C. Resende and Celso C. Ribeiro. GRASP: Greedy randomized adaptive search procedures. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Second Edition*, pages 287–312, 2014.

[27] Wouter Souffriau, Pieter Vansteenwegen, Greet Vanden Berghe, and Dirk Van Oudheusden. A path relinking approach for the team orienteering problem. *Computers and Operations Research*, 37(11):1853 – 1859, 2010. Metaheuristics for Logistics and Vehicle Routing.

[28] Wouter Souffriau, Pieter Vansteenwegen, Greet Vanden Berghe, and Dirk Van Oudheusden. The Multiconstraint Team Orienteering Problem with Multiple Time Windows. *Transportation Science*, 47(1):53–63, 2011.

[29] M Fatih Tasgetiren. A Genetic Algorithm with an Adaptive Penalty Function for the Orienteering Problem. 4(2):1–26.

[30] T Tsiligirides. Heuristic Methods Applied to Orienteering. *The Journal of the Operational Research Society Palgrave Macmillan Journals on behalf of the Operational Research Society J. OpI Res. Soc*, 35(9):797–809, 1984.

[31] Pieter Vansteenwegen. Planning in tourism and public transportation: Attraction selection by means of a personalised electronic tourist guide and train transfer scheduling. 7, 01 2008.

[32] Pieter Vansteenwegen, Wouter Souffriau, and Dirk Van Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 209(1):1–10, 2011.

[33] Pieter Vansteenwegen, Wouter Souffriau, Greet Vanden Berghe, and Dirk Van Oudheusden. Iterated local search for the team orienteering problem with time windows. *Computers and Operations Research*, 36(12):3281–3290, 2009.

[34] C. Verbeeck, P. Vansteenwegen, and E. H. Aghezzaf. An extension of the arc orienteering problem and its application to cycle trip planning. *Transportation Research Part E: Logistics and Transportation Review*, 68:64–78, 2014.

[35] Kevin Waugh et al. *Abstraction in large extensive games*. PhD thesis, University of Alberta, 2009.

[36] Rong Yang, Benjamin Ford, Milind Tambe, and Andrew Lemieux. Adaptive resource allocation for wildlife protection against illegal poachers. *Aamas*, (Aamas):453–460, 2014.

[37] Rong Yang, Albert Xin Jiang, Milind Tambe, and Fernando Ordandez. Scaling-up security games with boundedly rational adversaries: A cutting-plane approach. *IJCAI International Joint Conference on Artificial Intelligence*, pages 404–410, 2013.

[38] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. *Advances in Neural Information Processing Systems*, 20:1729–1736, 2007.