

**MATEMATICKO-FYZIKÁLNÍ  
FAKULTA**  
Univerzita Karlova

## **BAKALÁŘSKÁ PRÁCE**

Dominik Bernard

# **Algoritmy pro multi-agentní hledání cest s protivníkem**

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Marika Ivanová, Ph.D.

Studijní program: Informatika

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne .....

Podpis autora

Chtěl bych poděkovat především své vedoucí Mgr. Marice Ivanové, Ph.D. za její cenné rady a nápady, které mi pomohly tuto práci dokončit. Dále bych chtěl poděkovat mé přítelkyni, rodině a přátelům za podporu. Také bych chtěl poděkovat mému spolubydlícímu, který se staral o domácnost, když jsem pracoval na práci.

Název práce: Algoritmy pro multi-agentní hledání cest s protivníkem

Autor: Dominik Bernard

Department: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Marika Ivanová, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Multi-agentní hledání cest s protivníkem je grafový problém. V několika vrcholech grafu jsou rozmístěni pohybliví agenti. Agenti jsou rozděleni do dvou týmů: útočníci a obránci. Každý útočník má přiřazen jeden cílový vrchol v grafu. Cílem útočníků je obsadit přiřazené cílové vrcholy, zatímco cílem obránců je zabránit útočníkům v dosažení jejich cílů. Oba týmy střídavě provádějí jeden tah dle vybraného algoritmu. Při jednom tahu se smí všichni agenti přesunout po hraně grafu do sousedního volného vrcholu, nebo mohou zůstat v aktuálním vrcholu. Cílem této práce je implementovat jak několik existujících algoritmů pro tým obránců i útočníků, tak několik nových algoritmů a rozšíření pro tým obránců. Tyto algoritmy poté experimentálně porovnáme ve vytvořeném prostředí.

Klíčová slova: MAPF, AMAPF, protivník, autonomní agent, diskrétní simulace

Title: Algorithms for adversarial multi-agent path finding

Author: Dominik Bernard

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Marika Ivanová, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Adversarial multi-agent path finding is a graph problem. In several vertices of the graph, mobile agents are distributed, which are divided into two teams: attackers and defenders. Each attacker is assigned one target vertex in the graph. The goal of the attackers is to occupy the assigned target vertices, while the goal of the defenders is to prevent the attackers from reaching their objectives. Both teams alternately perform one move according to the selected algorithm. In one move, all agents are allowed to move along the edge of the graph to the neighboring free vertex or they can stay at the current vertex. The aim of this work is to implement both several existing algorithms for the team of defenders and attackers, as well as several new algorithms and extensions for the team of defenders. We will then experimentally compare these algorithms in the created environment.

Keywords: MAPF, AMAPF, adversary, autonomous agent, discrete simulation

# Obsah

Úvod	7
<b>1 Definice AMAPF</b>	<b>9</b>
1.1 Definice MAPF	9
1.2 Definice AMAPF problému	10
1.3 Koncové situace v AMAPF	11
1.4 Definice algoritmů obránců a útočníků	11
1.4.1 Algoritmy útočníků	12
1.4.2 Algoritmy obránců	12
1.5 Složitost MAPF a AMAPF problému	12
<b>2 Související práce</b>	<b>13</b>
2.1 Algoritmy pro MAPF	13
2.2 Kompletní a optimální algoritmy pro MAPF	14
2.2.1 Optimální MAPF algoritmy	14
2.3 MAPF pro sobecké agenty	15
2.4 Soupeřivá prostředí	16
<b>3 Omezení na podproblém</b>	<b>18</b>
3.1 Grafová omezení	18
3.2 Omezení na pozice agentů a cílů	18
3.3 Omezení na pozice obdélníkových území	18
<b>4 Analýza</b>	<b>21</b>
4.1 Algoritmus útočníků	21
4.1.1 LRA* pro útočníky	21
4.2 Algoritmy obránců	22
4.2.1 LRA* pro obránce	22
4.2.2 Bottleneck Simulation Allocation algoritmus	23
4.2.3 Vylepšení Bottleneck Simulation Allocation	26
4.2.4 Minimal Cut Bottleneck algoritmus	32
<b>5 Vývojová dokumentace</b>	<b>43</b>
5.1 Schéma projektu	43
5.1.1 Pluginy - algoritmy útočníků a obránců	44
5.1.2 PerformanceTester	45
5.2 Datové formáty	45
<b>6 Uživatelská dokumentace</b>	<b>48</b>
<b>7 Experimenty a diskuse</b>	<b>52</b>
7.1 Metoda	52
7.1.1 Porovnávané algoritmy	52
7.1.2 Mapy	52
7.1.3 Počty agentů	53
7.1.4 Timespan simulace	53

7.1.5	Ostatní parametry simulací . . . . .	53
7.1.6	Měřené hodnoty . . . . .	55
7.2	Výsledky . . . . .	55
7.3	Diskuse . . . . .	55
7.3.1	Úspěšnost algoritmů při 50 obrácích . . . . .	55
7.3.2	Úspěšnost algoritmů při 20 obrácích . . . . .	60
7.3.3	Úspěšnost algoritmů při 10 obrácích . . . . .	62
7.3.4	Doba trvání algoritmů s 50 obránci . . . . .	63
7.3.5	Doba trvání algoritmů s 20 obránci . . . . .	64
7.3.6	Doba trvání algoritmů s 10 obránci . . . . .	65
	<b>Závěr</b>	<b>66</b>
	<b>Literatura</b>	<b>68</b>
	<b>Seznam obrázků</b>	<b>70</b>
	<b>Seznam použitých zkratk</b>	<b>71</b>

# Úvod

Obrana oblasti před vpádem nepřátelských jednotek je problematika motivovaná mnoha problémy z reálného života. Jsou to například různé vojenské operace, při kterých se jednotky z jednoho týmu snaží zabránit v průniku soupeřovým jednotkám. Motivaci můžeme nalézt také v různých záchranných misích, nebo při použití robotů k plnění hlídacích úkolů (Agmon et al. [1]). Obrana oblasti je však také problém, který se řeší v mnoha počítačových hrách, jako například StarCraft, Warcraft nebo Age of Empires. V této problematice spolu soupeří dva týmy jednotek. Jeden tým se snaží proniknout na nepřátelské území a druhý tým se snaží zastoupit nepřítelům cestu a tím jim znemožnit jejich postup na cílové pozice.

Formálně se tento problém nazývá Multi-agentní hledání cest s protivníkem (anglicky adversarial multi-agent path finding, zkr. AMAPF). V tomto problému simulujeme pohyb jednotek po hrací ploše. Prostor, kde se simulace odehrává, je reprezentován grafem, na kterém se pohybují z vrcholu do vrcholu mobilní agenti. Agenti jsou rozděleni do dvou týmů — útočníci a obránci. Každému útočnickovi je přiřazen jeden cílový vrchol v grafu. Cílem útočnicků je obsadit přiřazené cílové vrcholy. Cílem obránců je pak zabránit útočnickům v dosažení jejich cílových vrcholů.

Oba týmy střídavě provádějí jeden tah. Provedení jednoho tahu útočnicků a obránců se nazývá časový krok. Oba týmy provádějí své tahy do předem stanoveného celkového času (počtu časových kroků). Jedná se tedy o diskrétní simulaci. Během jednoho tahu se smějí všichni agenti týmu přesunout podél hrany grafu na sousední neobsazené pole, nebo mohou zůstat stát na místě. Pohyb agentů týmu je řízen algoritmem týmu.

Hledání cesty v grafu je dlouhodobě známým a dobře prozkoumaným problémem z oblasti teorie grafů. Naopak, multi-agentní hledání cest s protivníkem představuje relativně nové odvětví, které zatím v odborných kruzích nebylo detailně prozkoumáno a poskytuje tak mnoho příležitostí pro další výzkum a inovace. Není známo mnoho algoritmů, které by se s aspektem soupeře v grafu vypořádaly. Zejména pro agenty obránce není vytvořeno mnoho strategií, jak efektivně postupovat proti soupeři. Obzvláště, pokud je v grafu útočných agentů mnohonásobně více než obránců. Proto se pokusíme tuto problematiku prozkoumat a navrhnout několik nových myšlenek pro pohyb obránců v grafu.

**Cílem této práce je:** implementovat základní algoritmy pro útočníky a obránce představené v článku *Area Protection in Adversarial Path-Finding Scenarios with Multiple Mobile Agents on Graphs* (Ivanová a Surynek [2]), a poté rozšířit a navrhnout nové algoritmy pro tým obránců. Dále provést experimentální porovnání algoritmů ve vytvořeném prostředí, které bude umožňovat i vizualizaci pohybu agentů. Porovnávat budeme jak rychlost výpočtu tak i úspěšnost algoritmů obránců na různých grafech a s různou inicializací.

V této práci si nejprve detailně představíme problém Multi-agentního hledání cest s protivníkem. Uvedeme si základní definice a značení, které budeme v průběhu celé naší práce používat.

V další kapitole prozkoumáme, na jakých základech AMAPF problém staví. Povíme si o problému multi-agentního hledání cest bez protivníka a představíme

některé práce, zabývající se tímto tématem.

V další kapitole si k základní definici problému přidáme několik omezení, abychom mohli lépe porovnávat a vizualizovat testované algoritmy. Vysvětlíme si, proč budeme používat konkrétní omezení.

Dále prozkoumáme základní algoritmy, které se již přímo tímto konkrétním AMAPF problémem zabývají. Vysvětlíme si algoritmus **bottleneck simulation allocation** (viz sekce 4.2.2), o kterém se píše ve výše zmíněném článku [2] na straně 6 a 7. Poté uvedeme, jaká vylepšení a jaké nové algoritmy implementujeme. Ukážeme si, jaké změny jsme na algoritmu bottleneck simulation allocation provedli (viz sekce 4.2.3), a proč jsme je udělali. Vysvětlíme algoritmus **Minimal cut bottlenecks** (viz sekce 4.2.4). Popíšeme, jaké techniky jsme použili.

Následně si představíme prostředí pro vizualizaci běhu algoritmů a pro testování jejich výkonu ve vývojové a uživatelské dokumentaci. Vysvětlíme si, jaké datové formáty jsme použili.

Nakonec si popíšeme, jakými experimenty jsme měřili výkonnost jednotlivých algoritmů. Experimenty provedeme a změříme. V diskusi si pak rozebereme, co naměřené hodnoty znamenají a porovnáme úspěšnost a trvání jednotlivých algoritmů. V závěru si shrneme získané poznatky a pokusíme se vytyčit, jak by bylo možné na naší práci navázat.

# 1 Definice AMAPF

AMAPF problém vychází z velmi známého problému hledání cesty v grafu. První z algoritmů na hledání cesty v grafu je Dijkstrův algoritmus. Algoritmus pracuje tak, že se postupně pro každý vrchol  $v$  učí délku nejkratší cesty z počátečního vrcholu. Na začátku je tato vzdálenost neznámá (nastavena na  $\infty$ ), ale později se tato vzdálenost aproximuje nalezenými cestami do  $v$ . Poté co vrchol uzavřeme, nejkratší cesta je už známa. Algoritmus byl poprvé představen vědcem E. Dijkstrou v roce 1959.

Dijkstrův algoritmus otevírá vrcholy na základě vzdálenosti od počátečního vrcholu. Pokud změním pořadí otevírání vrcholů tak, že ke vzdálenosti od počátečního vrcholu připočteme ještě heuristický odhad vzdálenosti k cílovému vrcholu, získáme algoritmus zvaný  $A^*$ . Tento algoritmus byl poprvé publikován v roce 1968 vědci P. E. Hartem, N. J. Nilssonem a B. Raphaellem.

Pokud na graf  $G$  přidáme více pohyblivých agentů, kteří spolu spolupracují a každý se snaží po bezkonfliktní cestě dostat do svého cílového vrcholu, bavíme se již o tzv. MAPF problému. MAPF problémem se zabývá mnoho studií a odborných článků.

Pro analýzu a experimentální vyhodnocení je nutné formulovat problém, pravidla pohybu agentů a koncové situace. Nejprve si tedy zavedme definici MAPF problému, kterou následně rozšíříme na definici AMAPF.

## 1.1 Definice MAPF

O „klasickém MAPFu“ píší v článku *Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks* autoři Stern et al. [3]. Zde je autory problém definován na stranách 1 a 2 takto:

**Definice 1** (Klasický MAPF). *Vstupem do klasického MAPF problému je tuple  $\langle G, \alpha_0, \delta \rangle$  kde  $G = (V, E)$  je neorientovaný graf,  $\alpha_0 : [1, \dots, k] \rightarrow V$  mapuje agenty na počáteční vrcholy a  $\delta : [1, \dots, k] \rightarrow V$  mapuje agenty na cílové vrcholy. Čas je diskretizován a v každém časovém kroku se agent nachází v jednom z vrcholů grafu a může provést jednu akci. Akce v klasickém MAPFu je funkce  $a : V \rightarrow V$  taková, že  $a(v) = v'$ . To znamená, že pokud je agent ve vrcholu  $v$  a provede akci  $a$ , pak bude se v příštím časovém kroku nacházet ve vrcholu  $v'$ . Každý agent má dva typy akcí *wait* a *move*. *Wait* akce znamená, že agent zůstane v aktuálním vrcholu i příští časový krok. *Move* akce znamená, že se agent pohne z aktuálního vrcholu  $v$  do sousedního vrcholu  $v'$  v grafu (tedy  $(v, v') \in G$ ).*

*Dále pro sekvenci akcí  $\pi = (a_1, \dots, a_n)$  a agenta  $i$  označme  $\pi_i[x]$  jeho pozici po provedení prvních  $x$  akcí v  $\pi$ , začínajících v agentově počátečním vrcholu  $\alpha_0(i)$ . Formálně:  $\pi_i[x] = a_x(a_{x-1}(\dots a_1(\alpha_0(i))))$ . Sekvenci akcí  $\pi$  nazveme **single-agent plánem** pro agenta  $i$  právě tehdy, když se agent provedením této sekvence akcí dostane z  $\alpha_0(i)$  do  $\delta(i)$ . To je právě tehdy, když  $\pi_i[|\pi|] = \delta(i)$ . **Řešení** je pak množina  $k$  single-agent plánů, jeden pro každého agenta.*

Kromě klasického pojetí MAPF problému existuje ještě i „neklasický MAPF“, který kromě pozice agenta v grafu zohledňuje ještě i například plochu agenta (jeho

velikost na mapě), nebo rychlost agenta. Tímto pojetím MAPFu se však v naší práci nebudeme zabývat.

## 1.2 Definice AMAPF problému

Adversarial multi-agent path finding (zkr. AMAPF) neboli česky multi-agentní hledání cest s protivníkem je problém který k MAPF problematice přidává element protivníka. Celý problém je definován následovně.

Vstupem do AMAPF problému se rozumí tuple čtyř hodnot  $\langle G, R, \alpha_0, \delta^A \rangle$  kde:

- $G = (V, E)$  je neorientovaný graf, na kterém se bude daná simulace provádět
- $R$  je množina všech agentů taková, že  $R = A \cup D, A \cap D = \emptyset$ , přičemž
  - $A$  je množina všech agentů prvního týmu (dále jen útočníci)
  - $D$  je množina všech agentů druhého týmu (dále jen obránci)
- $\alpha_0$  je počáteční přiřazení všech agentů na vrcholy v grafu  $G$ . Jedná se o prosté zobrazení:  $\alpha_0 : R \rightarrow V$ 
  - Obecně  $\alpha_t$  bude zobrazení mapující agenty  $R$  v čase  $t \in \mathbb{N}_0$  na vrcholy z  $V$ , tedy  $\alpha_t : R \rightarrow V$ .
  - Dále definujeme  $\alpha_t^D : D \rightarrow V$ . To bude restrikce funkce  $\alpha_t$  na podmnožinu  $D$  mapující agenty obránci na vrcholy z  $V$  v čase  $t$ .
  - Podobně definujeme i  $\alpha_t^A : A \rightarrow V$ , což je restrikce funkce  $\alpha_t$  na podmnožinu  $A$  mapující útočníky v čase  $t$  na vrcholy z  $V$ .
- A konečně  $\delta^A$  je zobrazení, které přiřazuje všem agentům útočnickům jeden cílový vrchol.
  - Jedná se tedy o prosté zobrazení  $\delta^A : A \rightarrow V$ .
  - Množina všech cílů útočnicků je tedy obraz množiny útočnicků podle  $\delta^A$ , to jest  $\delta^A(A) = \{\delta^A(a) : a \in A\}$ .

Simulace probíhá po jednotlivých časových krocích (diskrétní simulace). Předem stanovený maximální počet časových kroků označíme  $T$ . Oba týmy střídavě provádějí jeden tah v jednom časovém kroku  $t \in \{1, \dots, T\}$ . Množinu  $\{1, \dots, T\}$  nazýváme timespan simulace.

Během jednoho tahu mohou všichni agenti týmu provést jednu ze dvou akcí. První akce agentů se nazývá **move** akce. Během této akce se agent  $r$  přesune podél hrany grafu ze své aktuální pozice  $\alpha_t(r)$  na sousední neobsazené pole, tedy  $\alpha_t(r) \neq \alpha_{t+1}(r)$ . Druhá akce agentů se nazývá **wait** akce. Agent při provedení této akce zůstane stát na místě, tedy  $\alpha_t(r) = \alpha_{t+1}(r)$ .

Není povoleno, aby dva agenti přebývali v jednom časovém kroku ve stejném vrcholu, formálně  $\alpha_t(r_1) = \alpha_t(r_2) \Leftrightarrow r_1 = r_2$ .

Zároveň není povoleno, aby si v jednom tahu 2 agenti vyměnili místa. Po jedné hraně smí tedy během jednoho tahu cestovat pouze jeden agent. Formálně  $(\alpha_t(r_1) \wedge \alpha_t(r_2)) \Rightarrow (\alpha_{t+1}(r_1) \neq \alpha_t(r_2) \vee \alpha_{t+1}(r_2) \neq \alpha_t(r_1))$ .

Pohyb agentů týmu je řízen algoritmem týmu.

V základní definici AMAPF problému není specifikováno, v jakém pořadí budou jednotliví agenti provádět své pohyby. Proto v naší práci zavedeme, že nejprve provedou svůj tah všichni obránci, poté všichni útočníci.

Cílem útočnicků je obsadit přiřazené cílové vrcholy. Cílem obránců je pak zabránit útočnickům v dosažení jejich cílových vrcholů. Jakými cestami se budou obránci snažit dosáhnout svých cílových vrcholů, záleží na konkrétním algoritmu útočnicků. Jakým způsobem se budou obránci snažit zamezit útočnickům přístup na cílové vrcholy, záleží zas na konkrétním algoritmu obránců. Tyto algoritmy budeme porovnávat a budeme zjišťovat, kterému algoritmu se daří lépe na jakých instancích AMAPF problému.

### 1.3 Koncové situace v AMAPF

Po vykonání posledního tahu v posledním časovém kroku nastane vyhodnocení AMAPF simulace. Oběma týmům rozdělujeme jeden výherní bod, podle toho, jak se týmům dařilo. Mohou nastat tyto případy.

- **Výhra útočnicků:** Nastane v případě, že po vykonání posledního tahu jsou všichni útočníci na svých cílových pozicích. Tedy formálně pokud  $\forall a \in A : \alpha_T(a) = \delta^A(a)$ .
  - V tomto případě dostanou jeden výherní bod útočníci, obránci získají nula bodů.
- **Výhra obránců:** Nastane v případě, že po vykonání posledního tahu není ani jeden útočník na své cílové pozici. Tedy formálně pokud  $\forall a \in A : \alpha_T(a) \neq \delta^A(a)$ .
  - V tomto případě dostanou jeden výherní bod obránci, útočníci získají nula bodů.
- Pokud se po vykonání posledního tahu nacházejí pouze někteří útočníci na svých cílových pozicích, nastává třetí možnost. Vítězný bod se pak rozděluje v poměru, kolik útočnicků dosáhlo svých cílů.
  - Vyhodnocení, jak se útočnickům dařilo dosáhnout svých cílů provedeme tak, že spočítáme, kolik útočnicků je po provedení posledního tahu na cílových pozicích. Ohodnocení úspěšnosti útočnicků získáme jako  $\left| \{a : a \in A \wedge \alpha_T(a) = \delta^A(a)\} \right| / |A|$ , kde  $\rho$ . Tedy počet útočnicků, kteří dosáhli cíle dělený počtem všech útočnicků.
  - Podobně vyhodnotíme i algoritmus obránců. Na konci simulace spočítáme počet útočnicků na cílových pozicích. Ohodnocení úspěšnosti obránců získáme jako  $1 - \left( \left| \{a : a \in A \wedge \alpha_T(a) = \delta^A(a)\} \right| / |A| \right)$ . Jedná se tedy o jedna minus počet bodů, které získali útočníci.

### 1.4 Definice algoritmů obránců a útočnicků

Nyní si uvedeme definice, jakým způsobem mají fungovat jednotlivé algoritmy útočnicků a obránců.

### 1.4.1 Algoritmy útočníků

Algoritmy (neboli strategie) útočníků budou rozhodovat o pohybu agentů útočníků v jednotlivých krocích simulace. Cílem algoritmů bude maximalizovat ohodnocení, které na konci útočníci dostanou (maximalizovat počet útočníků na cílových pozicích).

Na začátku simulace proběhne inicializace algoritmu. Během inicializace dostane algoritmus útočníků tyto informace. Obdrží graf  $G$ , ve kterém bude simulace probíhat. Dále dostane informaci o umístění všech agentů (získá informace z funkce  $\alpha_0$ ). Nakonec též agent dostane informaci o cílových vrcholech v grafu a jejich přiřazení na útočící agenty (získá informace z funkce  $\delta^A$ ).

Během jednoho tahu bude mít pak algoritmus možnost provést s každým agentem útočníkem jednu akci ze seznamu akcí — `move`, nebo `wait`.

### 1.4.2 Algoritmy obránců

Algoritmy (neboli strategie) obránců budou rozhodovat o pohybu agentů obránců v jednotlivých krocích simulace. Cílem algoritmů bude maximalizovat ohodnocení, které na konci obránci dostanou (minimalizovat počet útočníků na cílových pozicích).

Na začátku simulace též proběhne inicializace algoritmu. Při inicializaci dostane algoritmus obránců tyto informace. Obdrží graf  $G$ , ve kterém bude simulace probíhat. Dále dostane informaci o umístění všech agentů (získá informace z funkce  $\alpha_0$ ). Algoritmus obránců však nedostane informaci o přiřazení konkrétních útočníků na konkrétní cílové vrcholy. Nezáká informace z funkce  $\delta^A$ , pouze získá množinu cílových vrcholů jako obraz množiny útočníků podle  $\delta^A$ , tedy  $\delta^A(A)$ .

Během jednoho tahu bude mít pak algoritmus možnost provést s každým agentem obráncem jednu akci ze seznamu akcí — `move`, nebo `wait`.

## 1.5 Složitost MAPF a AMAPF problému

Výpočetní složitosti MAPF problému se zabývá mnoho studií.

Pokud nám nezáleží na optimalitě řešení, můžeme využít kompletní **Korhauserův algoritmus**, který v článku *Non-Optimal Multi-Agent Pathfinding Is Solved (Since 1984)* popisují Röger a Helmert [4]. Tento algoritmus má též tu výhodu, že pracuje v polynomiálním čase.

Optimální řešení pro MAPF problém na neorientovaných grafech se ukázalo být NP-úplným problémem, což ve svém článku vysvětluje i Surynek [5].

O výpočetní složitosti AMAPF problému píše v článku *Adversarial Multi-Agent Path Finding is Intractable* autoři Ivanová a Surynek [6]. Zde se autoři zabývají tímto rozhodovacím problémem: „Pokud máme instanci AMAPF  $\Pi$ , existuje vyhrávající strategie pro tým útočníků?“ Autoři později v článku předkládají důkaz, že tento rozhodovací problém je EXPTIME-kompletní. Autoři dále zmiňují, že obdobný problém je i v jiných hrách dvou hráčů jako jsou šachy nebo go také EXPTIME-kompletní, což je způsobeno nárůstem komplexity při více hráčích.

## 2 Související práce

Nyní se budeme zabývat pracemi, které se naší problematice věnují. Nejprve si uvedeme několik článků, které se zabývají klasickým MAPF problémem. Později si uvedeme i práce, které se již přímo zabývají AMAPF problematikou.

Nejprve si připomeneme již zmíněný článek *Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks*, který napsali Stern et al. [3].

Autoři se v tomto článku kromě definic a vysvětlení MAPF problému zabývají i různými druhy konfliktů. Celkem pojmenovávají pět konfliktů, které mohou nastat: vrcholový konflikt, hranový konflikt, konflikt následování, kruhový konflikt a výměnný konflikt. Autoři vysvětlují, že v různých článcích se berou v úvahu různé druhy konfliktů.

Dále se autoři zabývají i různými optimalizovanými funkcemi. Říkají, že ve většině odborných studií se optimalizuje buďto tzv. Makespan, nebo Sum of costs. Ty jsou v článku definované takto:

- Makespan spočítáme jako celkový čas, tj. počet časových kroků, potřebný k dosažení cílů všemi agenty. Záleží nám tedy na nejdelším ze všech časových kroků agentů. Formálně pro MAPF řešení  $\pi = \{\pi_1, \dots, \pi_k\}$ , makespan je definován jako  $\max_{1 \leq i \leq k} |\pi_i|$ .
- Sum of costs se spočítá jako součet časových kroků, které každý agent potřeboval k dosažení svého cíle. Formálně je sum of costs definován jako  $\sum_{1 \leq i \leq k} |\pi_i|$ . Sum of costs se někdy také nazývá *flowtime*.

### 2.1 Algoritmy pro MAPF

Algoritmy pro MAPF byly shrnuty a popsány už v roce 2005, v článku *Cooperative Pathfinding*, který napsal Silver [7]. Ve svém článku Silver popisuje algoritmy založené na myšlence prioritního plánování. Nejprve autor popisuje základní brute-force replanner algoritmus, který se nazývá **Local Repair A\*** (LRA\*). V tomto algoritmu nejprve všichni agenti naplánují cestu ze startovních pozic do cílových pozic. Při plánování používají A\* algoritmus a nijak nezohledňují ostatní agenty při svém plánování.

Poté se agenti začnou pohybovat po naplánované cestě v jednotlivých časových krocích, dokud by neměla nastat kolize. Pokud by agent  $a$  měl vstoupit na pole, na kterém se již nachází agent  $b$ , musí si agent  $a$  přeplánovat cestu k cíli (znovu pomocí A\*). Poté agent  $a$  vykoná krok z nové cesty. Tento algoritmus je však náchylný na zacyklení agentů, zvláště v úzkých místech grafu.

Abychom předešli těmto zacyklením, předkládá Silver ve svém článku nový algoritmus **Cooperative A\*** (CA\*). V tomto algoritmu rozvineme vstupní graf  $G = (V, E)$  do  $G' = (V', E')$  a to tak, že přidáme třetí dimenzi, kterou bude čas. Za každý vrchol  $v$  v  $G$  přidáme do  $G'$  vrcholy  $v_1, \dots, v_T$ , kde  $T$  je stanovený maximální počet časových kroků. Poté spojíme vrcholy pro wait akci. To uděláme následovně:  $\forall t : (v_t, v_{t+1}) \in E'$ . Dále přidáme hrany pro move akci. Pokud v původním grafu  $G$  vedla hrana z  $u$  do  $v$ , v novém grafu povede hrana z  $u_t$  do  $v_{t+1}$  a z  $v_t$  do  $u_{t+1}$  pro všechna  $t$ . Tento postup předpokládá, že není zakázán tzv. výměnný konflikt, při kterém si dva agenti během jednoho časového kroku vymění pozice (cestují po

jedné hraně grafu  $G$ ). Pokud bychom tento konflikt měli zakázaný, museli bychom graf  $G'$  ještě dále upravovat.

Poté, co agent nalezne cestu v grafu  $G'$ , uloží si cestu do rezervační tabulky. Další agenti, kteří budou plánovat svou cestu ze startovní do cílové pozice nesmějí porušit již rezervované pozice v tabulce. Tedy pokud si agent  $a$  naplánoval v čase  $t$  pobyt ve vrcholu  $v$ , agent  $b$  si již v tomto čase  $t$  pobyt ve vrcholu  $v$  naplánovat nesmí.

Dále Silver ve svém článku popisuje ještě dvě další vylepšení CA\* algoritmu. První je **Hierarchical Cooperative A\*** (HCA\*), ve kterém se autor zaměřuje na použití dobré heuristiky pro CA\* algoritmus.

Poslední z předvedených algoritmů od Silvera je pak **Windowed Hierarchical Cooperative A\*** (WHCA\*). Pracuje obdobně jako HCA\*, avšak místo naplánování celé cesty ze startu do cíle si agenti naplánují pouze cestu, která je dlouhá jako tzv. okno. Po uplynutí poloviny časových kroků v okně dochází k přeplánování cest a poté agenti pokračují v pohybu.

## 2.2 Kompletní a optimální algoritmy pro MAPF

V článku *Multi-Agent Path Finding – An Overview*, který napsal Stern [8] je popsáno, že ačkoli jsou výše popsané algoritmy rychlé a nabízejí často poměrně dobrá řešení pro MAPF problémy, nejsou ani kompletní, ani optimální.

- **Nekompletní** algoritmus nemusí nalézt žádné řešení MAPF problému, i když nějaké existuje.
- Nalezené řešení prioritně plánovací algoritmem může být **neoptimální** možné řešení vzhledem k optimalizované funkci (makespan, Sum of costs).

Dále autor uvádí jako příklad kompletního algoritmu **Kornhauserův algoritmus**, který jsme již zmínili v kapitole 1.5.

### 2.2.1 Optimální MAPF algoritmy

V článku od Sterna [8] nalezneme i přehled o optimálních algoritmech řešících MAPF problém. Základní optimální MAPF algoritmus je A\* algoritmus v tzv.  **$k$ -agent search space**.  $K$ -agent search space je graf  $G_k$ . Vytvoříme ho tak, že vrcholy budou množiny pozic (jedna pozice za každého agenta), na kterých se mohou vyskytovat agenti v jednotlivých časových krocích. Tedy, vrchol v  $G_k$  je vektor  $k$  vrcholů z původního grafu, tedy  $v_{G_k} = (v_1, \dots, v_k)$ . Hrana v  $G_k$  představuje společnou akci všech agentů, tj. množinu  $k$  akcí, jednu za každého agenta, které mohou agenti současně provést v určitém časovém kroku. Společné akce, které by vyústily v konflikt nebudou součástí hran v  $G_k$ .

Stern dále vysvětluje, že algoritmus pak funguje jako klasický A\* algoritmus. Pokud  $d(v_i, \delta(i))$  je vzdálenost nejkratší cesty z  $v_i$  do  $\delta(i)$ , pak heuristická funkce pro sum of costs je definovaná následovně:

$$h((v_1, \dots, v_k)) = \sum_{i \in \{1, \dots, k\}} d(v_i, \delta(i)). \quad (2.1)$$

A pro makespan následovně:

$$h((v_1, \dots, v_k)) = \max_{i \in \{1, \dots, k\}} d(v_i, \delta(i)). \quad (2.2)$$

Bohužel je velikost grafu  $G_k$  i větvící faktor exponenciální v  $k$  (počet agentů), a proto i na malých grafech nemůže být tento algoritmus při větších množstvích agentů použit.

Proto se používají různé úpravy a rozšíření tohoto algoritmu. Stern [8] v článku uvádí několik takových rozšíření. Jedno z nich je **Operator Decomposition**, kde když expandujeme počáteční vrchol  $(\alpha_0(1), \dots, \alpha_0(k))$ , uvažujeme pouze akce prvního agenta. Tento postup generuje množinu vrcholů s možnými umístěními agenta 1 v čase 1 a s umístěními ostatních agentů v čase 0. Když expandujeme tyto vrcholy, uvažujeme zase pouze akce druhého agenta.  $K$ -tý potomek počátečního vrcholu reprezentuje možné umístění všech agentů v čase 1. Takto pokračujeme v hledání, dokud nenalezneme vrchol reprezentující cílové vrcholy pro všechny agenty.

Autor v článku popisuje ještě několik dalších metod a algoritmů, rozšiřujících  $A^*$  algoritmus v  $k$ -agent search space. Jsou to algoritmy **Independence Detection** a **M\***. Autor dále popisuje optimální algoritmy založené i na zcela odlišných přístupech.

Jedním z jiných přístupů je **Conflict-Based Search** (CBS). CBS je dvojúrovňový algoritmus. V horní úrovni se řeší stromové prohledávání založené na konfliktech mezi agenty. Na spodní úrovni probíhá hledání nejkratší cesty pro pouze jediného agenta. Algoritmem se zabývají v článku *Conflict-Based Search For Optimal Multi-Agent Path Finding* autoři Sharon et al. [9]. Algoritmus je také vysvětlen v již zmíněném článku *Multi-Agent Path Finding – An Overview* od Sterna [8].

Další přístupy, které nejsou založené na rozšíření  $A^*$  algoritmu, zahrnují **The Increasing Cost Tree Search**, nebo **Constraint Programming**, které převádí problém na „constraint satisfaction problem“. Tyto algoritmy by však už byly nad rámec představy o problematice, kterou se zabýváme, a proto je nebudeme detailně popisovat.

## 2.3 MAPF pro sobecké agenty

Zajímavou variantou MAPF problematiky je takzvané multi-agentní hledání cest pro „sobecké“ agenty. Tito agenti spolu při hledání nejkratších cest nespolupracují. Každý agent se pouze snaží co nejrychleji dosáhnout svého cíle bez ohledu na okolní agenty v grafu. Jako příklad této varianty z reálného světa uveďme lidské řidiče na silnicích. Každý řidič se snaží dosáhnout svého cíle co nejrychleji, bez ohledu na to, jak tím ovlivní ostatní řidiče na silnici. Proto také občas vznikají kolony. Pokud se nadměrné množství řidičů rozhodne vydat stejnou trasou ke svým cílům, mnoho z nich je následně nuceno čekat, než se pohne řidič před nimi.

O tomto problému se píše v článku *Multi-Agent Path Finding for Self Interested Agents*, který napsali autoři Bnaya et al. [10]. Autoři se v tomto článku zabývají různými daňovými schémata, které mezi sebou porovnávají. V daňovém schématu přidávají autoři na určité vrcholy a hrany v některých časech „daň“. Daň se jako penále přičítá sobeckému agentovi, pokud se tento agent vyskytne

ve specifikovaném čase na zdaněné pozici. Cílem tohoto zdanění je implicitní koordinace mezi agenty, aby nedocházelo k přílišným konfliktům v grafu.

## 2.4 Soupeřivá prostředí

V klasické MAPF problematice máme pouze agenty, kteří spolu spolupracují a snaží se společně dosáhnout svých cílů. Dalším zajímavým odvětvím je MAPF pro sobecké agenty, při kterém se agenti o potřeby ostatních agentů v grafu nezajímají. Pokud však mezi agenty navíc přidáme soupeření, jedná se o zcela novou třídu problémů. Agenti se v tomto odvětví snaží předčít ostatní agenty, nebo dokonce ostatním agentům překazit jejich plány.

Do této třídy bychom mohli zařadit například problém **Multi-Robot Adversarial Patrolling**, o kterém píší v článku Agmon et al. [1]. V tomto problému se snažíme pomocí několika pohyblivých spolupracujících robotů hlídkovat v určité oblasti a nahlašovat objevení nepřátelských jednotek. Cílem robotů je maximalizovat zabezpečení oblasti a minimalizovat rizika, kterými jsou nepřátelské jednotky (vetřelci).

Dalším problémem, který se již velmi podobá našemu AMAPF problému, je **Adversarial Cooperative Path-finding**. Koncept byl poprvé představen v článku *Adversarial Cooperative Path-Finding: A First View*, který napsali Ivanová a Surynek [11]. Dále se o této problematice hovoří v článku *Adversarial Cooperative Path-finding: Complexity and Algorithms*, který napsali též Ivanová a Surynek [12].

V tomto problému máme několik týmů  $\theta = \{\vartheta_1, \vartheta_2, \dots, \vartheta_n\}$ , počet týmů je  $\leq$  počtu agentů. Každý agent má stejně jako v MAPFu přiřazený startovní vrchol. Oproti klasickému MAPFu má agent místo jednoho cílového vrcholu přiřazenou celou množinu cílových vrcholů. Cílem agenta je obsadit jeden z vrcholů jeho cílové množiny. Každý agent navíc patří do právě jednoho z týmů  $\theta$ . Cílem agentů jednoho týmu je dosáhnout svých cílů dříve, než to stihnou agenti ostatních týmů. Vítězným týmem se pak stane tým, který jako první obsadí všechny své cílové vrcholy, nebo má při ukončení simulace nejvíce agentů na cílových vrcholech. Při shodě mezi týmy bude vítězný tým ten, který má své agenty v posledním časovém kroku simulace blíže ke svým cílům (počítáme součet nejkratších cest).

Pro tento problém dále autoři výše zmíněných článků představují několik algoritmů. Prvním zmíněným algoritmem greedy metoda. Tato metoda pohybuje agenty na pozice s nejlepším ohodnocením, kde ohodnocení se počítá jako vzdálenost agenta od jeho cílové pozice. Dále autoři v článku používají algoritmus minimax s alfa-beta prořezáváním, Monte Carlo tree search a adversarial cooperative A\*.

Nakonec se zaměříme na samotný AMAPF problém. I tento problém byl již zkoumán v odborných kruzích. Článek zabývající se tímto problémem se nazývá *Area Protection in Adversarial Path-Finding Scenarios with Multiple Mobile Agents on Graphs — a theoretical and experimental study of target-allocation strategies for defense coordination* [2]. Autoři se zde zabývají složitostí AMAPF problému. Dále v článku nalezneme i návrhy algoritmů pro týmy útočníků a obránců.

Pro útočníky autoři používají jednoduchý Local Repair A\* algoritmus. Více se pak zaměřují na algoritmy pro obránce. Konkrétně implementují dvě varianty (random a greedy allocation) algoritmu Local Repair A\* pro obránce. Dále pak

přicházejí s novým algoritmem **Bottleneck Simulation Allocation**, který se snaží obránce posílat do úzkých míst v grafu a tím ztížit postup útočníků. Nakonec algoritmy pro obránce experimentálně porovnávají. V další kapitole se na Bottleneck Simulation Allocation algoritmus detailně podíváme. Také si popíšeme, jaké změny a vylepšení na algoritmu uděláme s cílem zlepšit jeho úspěšnost při bránění útočníkům v dosažení jejich cílů.

# 3 Omezení na podproblém

Abychom mohli jednotlivé algoritmy dobře porovnávat a vizualizovat, zavedeme si nějaká omezení na definici AMAPF problému. Budeme se tedy zabývat pouze podproblémem.

## 3.1 Grafová omezení

Ačkoli základní instance AMAPF problému uvažuje libovolný neorientovaný graf  $G$ , pro naše testování a vizualizaci se omezíme jen na speciální typy grafů. Konkrétně se budeme zabývat mřížkovými 2D grafy, kde jsou vrcholy uspořádané do 2D mřížky. Dovolíme si však udělat v grafu překážky (trhliny), a to tak, že některé vrcholy z mřížky vynecháme. Většina vrcholů bude tedy s ostatními vrcholy spojena čtyřmi hranami, až na vrcholy na okraji nebo vrcholy sousedící s nějakou překážkou.

## 3.2 Omezení na pozice agentů a cílů

V základní definici AMAPF problému uvažujeme libovolné přiřazení agentů útočníků i obránců na vrcholy v grafu. My si však i zde zavedeme následující omezení. Místo libovolných počátečních pozic si vytyčíme obdélníková území v grafu, která budeme považovat za počáteční území útočníků a obránců. Jelikož pracujeme v mřížkovém grafu, obdélník definujeme pomocí levého horního vrcholu a pravého dolního vrcholu. Do těchto obdélníkových území pak náhodně rozmístíme všechny příslušné agenty.

Také s cílovými pozicemi útočníků budeme postupovat obdobně. Místo libovolného rozmístění cílů na kterékoli vrcholy vytyčíme v grafu i v tomto případě obdélníkové území, kam poté náhodně rozmístíme všechny cíle agentů útočníků.

Takto budeme moci snadno hovořit o počátečním území obránců a útočníků a o cílovém území útočníků.

## 3.3 Omezení na pozice obdélníkových území

Ani omezení na obdélníková území nám však nestačí. Stále se totiž jedná o velmi širokou nabídku, jak rozmístit území obránců, útočníků i cílů útočníků. V této široké nabídce je však mnoho případů, při kterých i s triviálními strategiemi má jeden z týmů velikou výhodu (například pokud by se území útočníků rovnalo území cílů útočníků a území obránců by bylo někde daleko).

Proto si rozebereme jen několik zajímavých případů, kdy je soupeření mezi útočníky a obránci největší.

Jedním ze zajímavých způsobů pro porovnávání algoritmů agentů je, pokud na jednu stranu grafu umístíme počáteční území obránců na shodné místo jako počáteční území útočníků. Cílové území útočníků pak umístíme na opačnou stranu grafu. Tak bychom simulovali jakýsi závod v dosažení cílových pozic mezi útočníky a obránci. Útočníci i obránci mají v tomto rozestavení podobnou šanci na dosažení cílových vrcholů a žádný z týmů nemá z počátku výhodu. Přesto se v naší

práci tímto konkrétním typem rozestavení zabývat nebudeme. Je to z důvodu, že v tomto rozestavení nemají obránci většinou dostatek času na to, aby proti útočníkům použili nějakou sofistikovanou strategii. Obráncům pak nezbývá než se jen co nejrychleji přesunout na cílové vrcholy. My však chceme implementovat i jiné strategie, které se snaží bránit útočníkům už během jejich cesty k cílovým vrcholům.

Proto potřebujeme jiné rozestavení, ve kterém nám nebude vadit, pokud budou mít obránci počáteční poziční výhodu nad soupeřem. Důležité pro nás bude, že jednotlivé algoritmy obránců budou mít větší šanci ukázat, jak se jim daří bránit cílové pozice před útočníky. To budeme testovat i v případech, kdy budou stát obránci proti veliké přesile útočníků.

Počáteční rozestavení obdélníkových území uděláme tak, že si vstupní graf rozdělíme dvěma svislými čarami a dvěma horizontálními čarami na pomyslných devět obdélníků. Obránce vždy umístíme do středového obdélníku. Útočníky umístíme do libovolného krajního obdélníku a cílové pozice útočníků umístíme do obdélníku, který je středově souměrný podle středového obdélníku s vybraným obdélníkem útočníků (obrázek 3.1 ukazuje příklad možného rozložení počátečních území). Takto získají obránci možnost vstoupit útočníkům do cesty a znemožnit jim průchod na cílové pozice.

Počáteční území útočníků		
	Počáteční území obránců	
		Cílové území útočníků

Všimněme si, že počáteční území obránců je umístěné do středového obdélníku. Počáteční území útočníků je středově souměrné podle středového obdélníku s cílovým územím útočníků.

**Obrázek 3.1** Jedno z možných rozložení obránců a útočníků v grafu

# 4 Analýza

V této kapitole si vysvětlíme, jakými algoritmy se budeme v AMAPF problému zabývat. V naší práci se budeme podrobně zabývat algoritmy obránců. Nejprve si však vysvětlíme, jaký algoritmus bude používat útočící tým a poté se zaměříme na algoritmy pro obránce.

## 4.1 Algoritmus útočníků

Cílem naší práce není najít co nejlepší algoritmus pro útočníky, soustředíme se spíše na algoritmy obránců. Proto budeme pro útočníky používat jednoduchý LRA\* algoritmus. Tento algoritmus vychází z MAPF verze tohoto algoritmu, který je základně popsán už v sekci 2.1. Volba jednoduchého LRA\* algoritmu pro útočníky a sofistikovanějších algoritmů pro obránce není spravedlivá ve smyslu soutěžení mezi týmy. To nám ale nevádí, protože naším cílem je porovnat algoritmy obránců. V následující sekci si vysvětlíme, jak LRA\* upravíme pro AMAPF podmínky.

### 4.1.1 LRA\* pro útočníky

Algoritmus bude fungovat následovně. Nejprve si všichni agenti útočníci naplánují cestu ze startovních pozic do cílových pozic. Při plánování použijí A\* algoritmus. Ostatní agenti (útočníci i obránce) pro ně ve chvíli plánování budou představovat nemobilní překážky. Každý jednotlivý útočník si tedy naplánuje cestu bez vstupování do vrcholů, které jsou aktuálně obsazené jinými agenty.

Jako heuristiku pro A\* algoritmus používá každý útočník manhattanskou vzdálenost své a cílové pozice v grafu. To je přípustná i konzistentní heuristika, protože skutečná vzdálenost je určitě menší nebo rovná odhadované vzdálenosti. Navíc při udělení jednoho kroku se heuristický odhad k cíli zmenší maximálně o jedna.

Poté, co si agenti naplánují své cesty, začnou se po naplánované cestě v jednotlivých časových krocích pohybovat. Pohybují se ve své stanovené cestě, dokud by neměla nastat kolize s jiným agentem. Při posuzování rizika kolize nezáleží na tom, jestli by kolize nastala s agentem ze stejného, nebo opačného týmu. Riziko kolize nastane, pokud by měl agent  $a$  vstoupit na pole, na kterém se již nachází agent  $b$ . Aby kolize nenastala, agent  $a$  si v aktuálním vrcholu přeplánuje cestu k cíli (znovu pomocí A\*) a poté vykoná krok z nové cesty.

V momentě, kdy si agent útočník plánuje novou cestu nebo mění plán své stávající cesty však může nastat problém (v běžné simulaci tento problém často nastává). Může se stát, že nová cesta útočníka k jeho cílovému vrcholu aktuálně neexistuje (na trase mu překážejí ostatní agenti). V takovéto situaci si tento útočník uniformně náhodně vybere jednu z akcí, které aktuálně může provést a tu provede. Útočník si vždy může vybrat `wait` akci nebo si může vybrat jednu ze čtyř `move` akcí, pokud jsou proveditelné (záleží na útočnickově okolí v grafu).

K přeplánování cesty nejčastěji dochází v úzkých hrdlech grafu. To jsou místa (vrcholy) v grafu, které ve svém okolí mají mnoho překážek, ale zároveň spojují velké a důležité oblasti grafu (detailní popis úzkých hrdel se nachází v sekci 4.2.2, kde hrají úzká hrdla významnou roli).

Čím více agentů se v těchto úzkých místech najednou sejde, tím častěji zde dochází k přeplánování trasy. Proto jsme zavedli toto náhodné chování agentů, když nenaleznou žádnou cestu k cílovému vrcholu. Představa je taková, že pokud agent  $a$  narazil na překážku a nedokáže ji v aktuální situaci žádným způsobem obejít, je kolem něj nejspíše mnoho agentů se stejným problémem. Vykonáním náhodného kroku tak možná tento agent  $a$  uvolní cestu dalším agentům, kteří by jinak museli svou cestu také přeplánovat.

Takto algoritmus pracuje po celou dobu simulace.

## 4.2 Algoritmy obránců

Nyní si povíme o algoritmech pro obránce. Algoritmy pro obránce mají následující strukturu:

1. Inicializace algoritmu.
  - (a) Nalezení cílů pro obránce.
  - (b) Určení cest obránců k jednotlivým cílům.
2. Provádění tahů v rámci jednotlivých časových kroků v průběhu simulace s případným přeplánováním cesty pro obránce.

My se v následujících algoritmech budeme zabývat hlavně částí „Nalezení cílů pro obránce“. Hledání cest (pomocí A\* algoritmu) i provádění jednotlivých tahů již bude u všech algoritmů probíhat stejně.

Nejprve si vysvětlíme algoritmus LRA\* pro obránce. Poté se zaměříme na algoritmus z článku od Ivanové a Suryňka Bottleneck Simulation Allocation [2]. Dále se podíváme, jaké změny na algoritmu uděláme pro zvýšení jeho úspěšnosti. Nakonec si povíme o novém algoritmu Minimal Cut Bottlenecks. Tyto algoritmy budeme později porovnávat v experimentální části.

### 4.2.1 LRA\* pro obránce

Algoritmus LRA\* pro obránce funguje velmi podobně jako LRA\* pro útočníky (viz předchozí sekce 4.1.1).

Agenti obránce se při tomto základním algoritmu budou také snažit co nejrychleji dostat na cílové pozice útočníků. Žádný obránce však nemá přiřazený jeden konkrétní cílový vrchol, ke kterému by se chtěl tento obránce dostat. Místo toho je cílem obránce obsadit libovolný cílový vrchol, který patří nějakému útočníkovi. Jedná se tedy o takzvanou „greedy“ verzi LRA\* algoritmu.

Z důvodu, že se všichni obránce snaží obsadit libovolný nejbližší vrchol, používá každý obránce jako heuristiku pro A\* algoritmus nejmenší manhattanskou vzdálenost své a libovolné cílové pozice kteréhokoli útočníka v grafu. To je přípustná i konzistentní heuristika, protože skutečná vzdálenost k nejbližšímu cílovému vrcholu je určitě menší nebo rovná odhadované vzdálenosti k tomuto vrcholu. Při provedení jednoho kroku agenta se heuristický odhad k libovolnému cíli zmenší maximálně o jedna.

Poté, co si agenti naplánují své cesty, začnou se po těchto cestách v jednotlivých časových krocích pohybovat. Kolize agentů se řeší stejným způsobem, jako v algoritmu LRA\* pro útočníky (viz sekce 4.1.1).

## 4.2.2 Bottleneck Simulation Allocation algoritmus

Další z algoritmů, který si představíme, je algoritmus nazvaný Bottleneck Simulation Allocation. Tento algoritmus byl poprvé představen v článku *Area Protection in Adversarial Path-Finding Scenarios with Multiple Mobile Agents on Graphs* [2].

Základní myšlenka algoritmu je taková, že se obránci nesnaží obsazovat cílové vrcholy útočníků, ale snaží se obsadit významná úzká hrdla v grafu s cílem zabránit útočníkům v průchodu na cílové pozice.

Nejprve si vysvětlíme, co jsou to úzká hrdla v grafu a poté se podíváme, jak je hledat. Dále si vysvětlíme, jak si agenti obránci zvolí cíle své cesty a následně si ukážeme, jak se pohybují.

**Úzké hrdlo** v grafu si můžeme představit jako místo (vrcholy v grafu) mezi dvěma překážkami, které spojují významné oblasti grafu. Pokud bychom do úzkého hrdla rozmístili obránce, cesta z jedné strany úzkého hrdla na druhou už nebude možná, nebo se alespoň výrazným způsobem prodlouží. Obrázek 4.1 zobrazuje příklady úzkých hrdel.

Nyní se zkusíme zamyslet nad tím, jak úzká hrdla hledat. Některá úzká hrdla v grafu bychom mohli nalézt jednoduše, pokud bychom prošli graf zleva doprava a shora dolů (takto bychom mohli nalézt první úzké hrdlo v obrázku 4.1). Tato metoda však není vhodná, pokud se v obrázku vyskytují i úzká hrdla, která nejsou vodorovná nebo svislá (viz příklad druhého a třetího úzkého hrdla v obrázku 4.1).

Proto zvolíme jinou techniku, která je též popsána ve výše zmíněném článku [2].

Pokud chceme nalézt úzké hrdlo v grafu, vyvinuli autoři článku následující strategii. Autoři předpokládají, že když se útočníci pohybují směrem k cílovým vrcholům, navštěvují častěji vrcholy, které jsou v úzkých hrdlech, než vrcholy, které se v úzkých hrdlech nenacházejí. Proto algoritmus Bottleneck Simulation Allocation při inicializaci nejprve simuluje pohyb útočníků ze startovních pozic na cílové pozice. Protože však obránci nevědí, který cílový vrchol patří kterému útočnickovi, odhadnou toto přiřazení náhodnou bijekcí mezi počátečními pozicemi útočníků  $\alpha_0(A)$  a cílovými pozicemi útočníků  $\delta^A(A)$ .

Simulace pohybu útočníků probíhá tak, že se jednoduše hledá nejkratší cesta mezi startovním a cílovým vrcholem. Při hledání se ignorují všichni agenti, kteří jsou rozmístění ve vrcholech grafu. Každému vrcholu na nejkratší cestě se zvedne počítadlo o jedna. Takto odsimulujeme pohyb všech útočníků v grafu.

Tímto postupem získáme pro různé vrcholy různou četnost návštěvnosti vrcholu. Všechny vrcholy si seřadíme podle návštěvnosti. Vybereme vrchol s největší četností návštěvnosti a označíme ho  $w$ . Pokud je vrcholů s největší četností více, vybereme ten vrchol, který je nejbližší k průměrné pozici obránců.

Poté, co získáme vrchol  $w$ , začneme prohledávat jeho okolí, abychom zjistili, jestli se v tomto místě nenachází úzké hrdlo. Prohledávání probíhá následujícím způsobem. Nejprve určíme okolí jako čtverec se středem v bodě  $w$  a s velikostí hrany čtverce  $d$ .

Jaká je správná délka  $d$  musíme experimentálně zjistit. V našich experimentech na našich grafech jsme zjistili, že se algoritmus chová nejlépe, pokud nastavíme  $d = 5$ . Pokud bychom však algoritmus testovali na jiných mapách, mohlo by se stát, že bude potřeba zvolit jiné  $d$ .

Když máme určené okolí, začneme toto okolí (vrchol po vrcholu) prohledávat. Pokud nalezneme překážku (chybějící vrchol na prohledávaných souřadnicích),

přidáme ji do grafu překážek  $G_p$ , který si vytváříme. Po prohledání celého okolí zkontrolujeme, kolik komponent souvislosti mají překážky v grafu překážek. Dvě překážky jsou v grafu  $G_p$  spojené hranou, pokud spolu v originálním grafu  $G$  vodorovně, svisle nebo úhlopříčně přímo sousedí. Takto nalezneme komponenty souvislosti v  $G_p$ .

Pokud jsme našli více než jednu komponentu souvislosti, pravděpodobně jsme našli úzké hrdlo. Vybereme jednu z komponent jako počáteční komponentu a nalezneme nejkratší propojení přes vrcholy v  $G$  do zbylé množiny komponent. Nejkratší propojení hledáme pomocí Dijkstrovho algoritmu, kde počáteční vrcholy jsou překážky z počáteční komponenty. Jako cílové vrcholy označíme všechny překážky z ostatních komponent. Propojení hledáme přes vrcholy v grafu  $G$ . Upravíme však sousedství tak, že dva vrcholy spolu sousedí, pokud vedle sebe leží vodorovně, svisle nebo i úhlopříčně. Vrcholy ležící na nejkratším propojení označíme jako nalezené úzké hrdlo.

Pokud však máme v množině komponent pouze jednu nebo žádnou komponentu souvislosti, úzké hrdlo se v tomto místě nenachází. Autoři algoritmu se v článku v takovém případě domnívají, že graf, na kterém simulaci provádíme, má příliš mnoho volnosti pro agenty, a proto není pro strategii Bottleneck Simulation Allocation vhodný. Obránci si v tomto případě vyberou jako své cíle libovolné cílové pozice útočníků a zbytek algoritmu probíhá stejně jako LRA\* pro obránce 4.2.1.

Nyní se však vraťme k případu, že jsme úzké hrdlo našli. V tomto případě zkontrolujeme, jestli máme dostatek volných obránců na to, abychom úzké hrdlo obsadili. V případě, že ano, přiřadíme některým volným obráncům vrcholy z úzkého hrdla jako jejich cílové vrcholy tak, aby počet vrcholů v úzkém hrdle byl rovný počtu obránců, kteří se pokusí úzké hrdlo obsadit. Poté spustíme další simulaci pohybu útočníků, avšak mezi překážky nyní přidáme i vrcholy z úzkého hrdla, které jsme se rozhodli blokovat. Takto opakujeme hledání a přiřazování obránců stále dokolečka, dokud máme dostatek obránců k blokování úzkých hrdel, nebo dokud nějaká úzká hrdla nalézáme. Pokud obránců už nemáme dostatek, přiřadíme zbývající obránce na libovolné cílové vrcholy.

Takto probíhá celá inicializace Bottleneck Simulation Allocation algoritmu.

Inicializaci jsme popsali v následujícím pseudokódu 1, který je převzatý z článku *Area Protection in Adversarial Path-Finding Scenarios with Multiple Mobile Agents on Graphs* na straně 7 [2].

Takto definovaný algoritmus na hledání úzkých hrdel je však náchylný k nalezení tzv. **falešných úzkých hrdel**. To jsou úzká hrdla v grafu, přes která se však nepohybují žádní agenti útočníci (příklad takového falešného úzkého hrdla můžeme vidět na obrázku 4.2). Abychom předešli obsazení těchto falešných hrdel, navrhuje autoři algoritmu ve výše zmíněném článku [2] toto řešení. Po nalezení úzkého hrdla provedeme znovu simulaci pohybu útočníků k cílovým vrcholům, avšak nyní zakážeme útočníkům vstupovat na vrcholy z nalezeného úzkého hrdla. Poté porovnáme cesty útočníků před zakázáním pohybu přes nalezené úzké hrdlo a po zakázání pohybu přes toto hrdlo. Pokud se cesty útočníků nezměnily, nalezené úzké hrdlo je falešné.

Jak postupovat v případě nalezení falešného úzkého hrdla autoři článku neříkají. V článku je pouze vysvětleno, že úzké hrdlo není potřeba obránci obsazovat. Proto jsme se v naší implementaci rozhodli, že nalezené falešné úzké hrdlo ignorujeme a vrchol  $w$ , od kterého jsme falešné úzké hrdlo hledali, vyřadíme z frekventované

---

**Algoritmus 1** Inicializace Bottleneck Simulation Allocation algoritmu. V rámci inicializace algoritmus hledá úzká hrdla a přiřazuje k nim obránce. Zbylé obránce algoritmus přiřadí na náhodné cílové vrcholy útočníků.

---

```

1: function BOTTLENECK SIMULATION PROCEDURE
2:   Data:  $G = (V; E), D, A$     ▷ Vstupní graf, množina obránců a útočníků
3:   Result: alokace cílů  $\delta^D$ 
4:    $T_{available} = \{\delta^A(a) : a \in A\};$ 
5:    $D_{available} = D;$ 
6:    $F = \emptyset$                                 ▷ zakázané vrcholy
7:    $\delta' = \text{Náhodný odhad } \delta^A;$     ▷ náhodná bijekce mezi počátečními pozicemi
   útočníků a cílovými vrcholy útočníků
8:   while  $D_{available} \neq \emptyset$  do
9:     for  $a \in A$  do
10:       $p_a = \text{nejkratšíCesta}(\alpha_0(a), \delta'(a), G, F);$     ▷ Nejkratší cesta hledaná
   pomocí A*, ostatní agenty ve vrcholech grafu ignorujeme
11:    end for
12:     $\gamma(v) = |\{p_a : a \in A \wedge v \in p_a\}|;$     ▷ Každému vrcholu spočítáme
   četnost  $\gamma$ , kolikrát se vyskytl na nějaké z nejkratších cest
13:     $w \in \text{argmax}_{v \in V} \gamma(v);$     ▷ nejnavštěvovanější vrchol
14:     $B = \text{prohledejOkolí}(w);$     ▷ Úzké hrdlo; metoda prohledejOkolí vrací  $\emptyset$ ,
   pokud úzké hrdlo nebylo nalezeno
15:    if  $B \neq \emptyset$  then
16:       $D' \subseteq D_{available}, |D'| = |B|$ 
17:       $\text{přiřadVrcholyObráncům}(B, D');$     ▷ Přiřadíme vybrané agenty na
   vrcholy z úzkého hrdla
18:       $D_{available} = D_{available} \setminus D';$ 
19:       $F = F \cup B$     ▷ úzké hrdlo přidáme k zakázaným vrcholům
20:    else
21:      break;
22:    end if
23:  end while
24:   $\text{přiřadVrcholyObráncům}(T_{available}, D_{available});$     ▷ Přiřadíme zbývající agenty
   na libovolné cílové vrcholy útočníků
25: end function

```

---

obsazovaných vrcholů. Jako nové  $w$  vybereme nový nejfrekventovaněji navštěvovaný vrchol. Pak pokračujeme v hledání nového úzkého hrdla od nového  $w$ .

Poté, co jsou všichni obránci přiřazeni na nějaký vrchol, končí inicializace Bottleneck Simulation Allocation a agenti se začínají pohybovat v jednotlivých časových krocích simulace.

Každý obránc se pohybuje po své naplánované cestě, dokud by neměla nastat kolize. V prvním kroku si každý obránc zkusí naplánovat cestu ke své cílové pozici, pomocí algoritmu  $A^*$ . Když se mu to povede, vydá se po naplánované cestě. Pokud by měl obránc zapříčinit následujícím krokem kolizi, tak se tento obránc pokusí přeplánovat svou cestu. Jestliže se obránci podaří naplánovat nová cesta, vydá se po ní.

Pokud se však obránci nepovede naplánovat cesta k jeho cílovému vrcholu (na začátku, nebo při přeplánování), zůstane stát na místě, tedy provede `wait` akci. Toto chování je u obránců nastaveno proto, že pokud neexistuje cesta obránc k jeho cíli, nejspíše se právě vyskytuje v místě s mnoha agenty. Když nevede cesta pro něj, možná snad alespoň úspěšně brání jiným agentům útočnickům v pohybu. Také se v simulaci často stává, že obránc stojí už v těsné blízkosti svého cílového vrcholu, ale na vrchol se aktuálně nemůže dostat, protože je vrchol obsazen jiným agentem. Kdyby obránc vykonal náhodný krok, jako v předchozích algoritmech, nejspíše by se od svého cílového vrcholu vzdálil. Nejlepší chování tedy je, pokud vyčká na místě.

### 4.2.3 Vylepšení Bottleneck Simulation Allocation

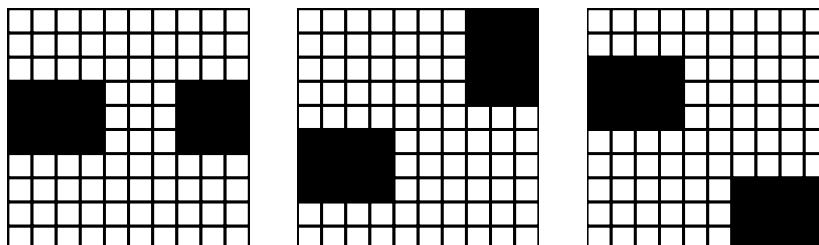
Nyní si vysvětlíme, jakými způsoby bychom mohli algoritmus Bottleneck Simulation Allocation vylepšit, aby s větší úspěšností zabraňoval útočnickům v dosažení jejich cílů. Také si povíme o zlepšení, které se snaží urychlit poměrně dlouhou inicializaci algoritmu.

Nový vylepšený algoritmus jsme nazvali **Bottleneck Nearest Defender Allocation**. Algoritmus se tak jmenuje po prvním vylepšení, kterým jsme se rozhodli původní algoritmus Bottleneck Simulation Allocation vylepšit. Souhrn celého algoritmu je popsán v pseudokódu níže 2.

#### Nearest Defender Allocation

První jednoduché vylepšení, které můžeme implementovat do stávajícího algoritmu, se jmenuje **Nearest Defender Allocation**. Toto vylepšení funguje následovně. Poté, co nalezneme nějaké úzké hrdlo a chtěli bychom k němu přiřadit nějaké obránc, nevybíráme obránc z množiny dostupných obránců náhodně, avšak vybereme ty obránc, kteří jsou aktuálnímu úzkému hrdlu nejbliže. Nejbližší obránc určíme pomocí  $A^*$  algoritmu. Počáteční vrcholy budou vrcholy z úzkého hrdla a cílové vrcholy budou všechny pozice obránců.

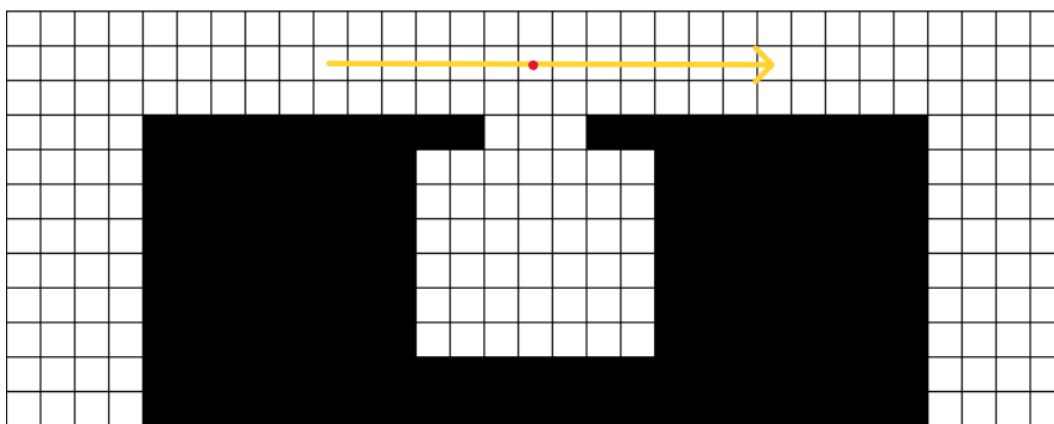
Tímto vylepšením docílíme zefektivnění algoritmu. Obránci se budou dostávat na vrcholy z úzkých hrdel v průměru za kratší čas, než kdybychom na úzká hrdla přiřazovali náhodné obránc.



Vrcholy v těchto obrázcích jsou zde znázorněné bílými čtverečky. Mezi dvěma vrcholy vede hrana, pokud spolu čtverečky hranově sousedí. Toto značení budeme dále používat i u ostatních obrázků.

Tyto obrázky zobrazují různé příklady úzkých hrdel. V prvním a druhém obrázku má úzké hrdlo velikost tři. Ve třetím obrázku je úzké hrdlo široké v nejužším místě pouze dva vrcholy.

**Obrázek 4.1** Příklady úzkých hrdel v grafu



V tomto obrázku je předpokládán největší pohyb agentů útočníků znázorněn žlutou šipkou. Pokud by byl největší výskyt útočníků na pozici (ve vrcholu) s červenou tečkou, bude tento vrchol označen jako  $w$  a z tohoto vrcholu bude hledáno úzké hrdlo. Nalezeno bude ale falešné hrdlo (úzké místo mezi překážkami pod šipkou).

**Obrázek 4.2** Příklad falešného úzkého hrdla v grafu

## Found Bottlenecks Caching

Další jednoduché zlepšení algoritmu, tentokrát na zrychlení inicializace, se jmenuje **Found Bottlenecks Caching**. Z našeho experimentálního měření vyplývá, že algoritmus Bottleneck Simulation Allocation je v základní verzi velice náchylný k nalézání falešných úzkých hrdel. Navíc algoritmus stejná falešná úzká hrdla nacházel opakovaně.

Že algoritmus najde jedno falešné úzké hrdlo vícekrát, můžeme vidět i z obrázku 4.2. V prvním hledání úzkého hrdla je nejfrekventovaněji navštěvovaný vrchol  $w$  označen červenou tečkou. Nalezené bude falešné úzké hrdlo pod šipkou a vrchol s červenou tečkou bude odstraněn z prvního místa frekventovaně navštěvovaných vrcholů. Často se však stane, že všechny vrcholy podél žluté šipky budou mít vysokou návštěvnost, a proto v několika dalších hledáních úzkého hrdla bude znovu nalezeno to stejné falešné úzké hrdlo. Navíc poté, co se nám konečně podaří nalézt nějaké skutečné úzké hrdlo, algoritmus znovu simuluje pohyb útočníků v grafu. Může se stát, že bude znovu nejvíce útočníků procházet kolem falešného hrdla po žluté šipce. Nalézání tohoto falešného úzkého hrdla se bude tedy znovu opakovat.

V tomto vylepšení si tak budeme pamatovat všechna dosud nalezená úzká hrdla (falešná i skutečná). Poté, co algoritmus nalezne nové úzké hrdlo, nejprve zkontrolujeme, jestli již úzké hrdlo není v množině nalezených úzkých hrdel. Pokud již v množině je, rovnou přejdeme na hledání nového úzkého hrdla a neztrácíme čas kontrolováním, jestli se jedná o falešné úzké hrdlo, či nikoliv.

Toto vylepšení se ukáže jako obzvlášť účinné na grafech, kde je rozmístěno mnoho překážek (například graf používaný v experimentální části s názvem „random-64-64-20“ 7.1) a algoritmus zde nalézal falešná úzká hrdla velmi často.

## Previous Bottleneck Inclusion

Jedno z jednodušších vylepšení, které jako další můžeme do algoritmu přidat, se jmenuje **Previous Bottleneck Inclusion**. V základní verzi algoritmu se úzké hrdlo hledá na čtvercové ploše se středem ve vrcholu  $w$ . Hledá se tak, že se snažíme nalézt alespoň dvě komponenty souvislosti v grafu překážek. V tomto vylepšení jsme si uvědomili, že do grafu překážek můžeme přidat i všechny vrcholy ležící v prohledávaném čtverci grafu  $G$ , které patří do nějakého nalezeného úzkého hrdla z předchozí iterace (tyto vrcholy jsou v algoritmu 1 postupně přidávány do množiny vrcholů  $F$ ). Pokud by se nám povedlo propojit okraj nějaké skupiny překážek s vrcholem z úzkého hrdla z předchozích iterací, může se jednat o další důležité úzké hrdlo (situaci znázorňuje obrázek 4.3).

## Randomized Shortest Paths

Další vylepšení se jmenuje **Randomized Shortest Paths**. V tomto vylepšení změním metodu, jakou generujeme odhady nejkratších cest útočníků v grafu. Pokud existuje více nejkratších cest z aktuální pozice útočníka na odhadnutou cílovou pozici útočníka, tak v algoritmu Bottleneck Simulation Allocation je deterministicky určené, která z nejkratších cest bude útočnickova cesta. To je důležité z důvodu zpětné kontroly, jestli se cesty změnila a nebo se jedná o falešné úzké hrdlo. Pokud by si útočník vybíral svou nejkratší cestu z množiny nejkratších

cest nedeterministicky, téměř jistě by se pokaždé při kontrole shodných cest zjistilo, že se cesty změnilly. Abychom tedy mohli kontrolu falešného úzkého hrdla provést, musí být výběr nejkratší cesty deterministický.

Při deterministickém výběru cesty však mají útočníci tendenci k dlouhému následování společné cesty. Kvůli tomuto jevu však nastává problém s nalezením úzkého hrdla, protože nejfrekventovaněji navštěvovaný vrchol  $w$  leží často někde zcela mimo hledané úzké hrdlo. Navíc se často stane, že nejfrekventovaněji navštěvovaných vrcholů je mnoho. Pak se z těchto vrcholů zvolí za  $w$  vrchol nejbližší středu obránců, což může být někde zcela mimo hledané úzké hrdlo.

V obrázku 4.4 můžeme vidět příklad odhadovaného pohybu útočníků, při kterém dojde ke zmíněnému problému. Odhadovaný deterministický pohyb útočníků (červené čtverečky) směrem k cílovým vrcholům (čtverečky se zelenými hranami) je znázorněn žlutou šipkou. Determinismus je zde zařízen tak, že pokud mají útočníci více nejkratších cest, které vedou k cílovému vrcholu, vyberou si vždy ten pohyb, který (pokud je to možné) minimalizuje  $x$ -ovou souřadnici. Jako druhé rozhodovací kritérium je minimalizování  $y$ -ové souřadnice.

Nejfrekventovaněji navštěvovaných vrcholů bude v případě tohoto obrázku mnoho, konkrétně jsou to všechny vrcholy podél nejdelší žluté šipky, tedy v řádku kde  $x = 1$ . Těchto vrcholů je dohromady 27. Za  $w$  bude zvolen ten vrchol, který je nejbližší průměrné pozici obránců (obránci pro zjednodušení v obrázku nejsou zakreslení). Pokud by ale průměrná pozice obránců ležela někde na levé nebo na pravé straně grafu, bude  $w$  vybráno špatně a obráncům se úzké hrdlo nalézt nepodaří. Obránci pak budou přiřazeni na libovolné cílové pozice útočníků a inicializace skončí. Přitom však vidíme, že úzké hrdlo v obrázku je, a dokonce by stačil i pouze jeden obránci k tomu, aby úzké hrdlo zatarasil. Tím by zcela znemožnil pohyb útočníků k cílovým vrcholům.

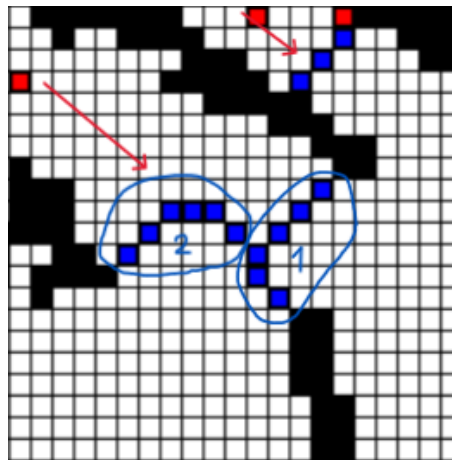
Z tohoto důvodu bychom potřebovali přidat do odhadování pohybů obránců nedeterminismus. Pokud bychom nejkratší cestu vybírali ze všech nejkratších cest náhodně, mnohem častěji se nám stane, že se nejvíce útočníků setká až ve skutečném úzkém hrdle, a poté se za úzkým hrdlem cesty útočníků znovu rozdělí. Tak bychom identifikovali pravá úzká hrdla mnohem častěji, než při deterministickém odhadu pohybu útočníků k cílovým vrcholům. Proto jsme se rozhodli nedeterminismus do odhadování cest útočníků skutečně přidat.

Přidat nedeterminismus do odhadování cest útočníků je jednoduchá záležitost. Cesty útočníků odhadujeme pomocí algoritmu  $A^*$ . Stačí tedy, když k heuristickému odhadu vzdálenosti útočníka k jeho cílovému vrcholu přičteme nějaký malý náhodný šum. Tím zaručíme, že bude útočník vrcholy na nejkratších cestách ze startu do cíle otevírat v náhodném pořadí. Díky tomu bude nejkratší cesta útočníka k jeho cílovému vrcholu vybrána náhodně z množiny nejkratších cest k cílovému vrcholu.

Potřebujeme však nyní změnit metodu, jakou kontrolujeme, jestli se jedná o falešné úzké hrdlo. Dosavadní metoda, při které kontrolujeme, jestli se odhadované cesty útočníků nezměnily, už nebude dále fungovat.

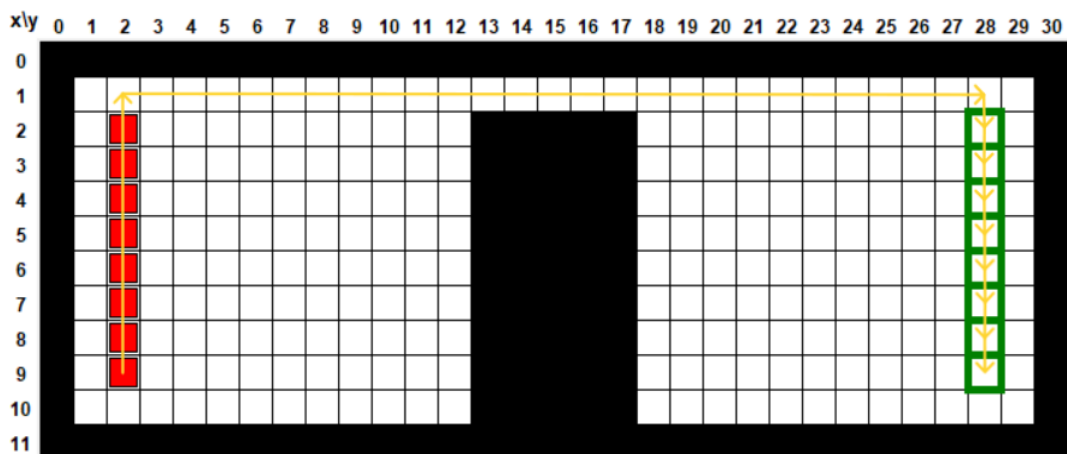
## Flow Through Bottleneck Detection

Jak budeme kontrolovat, jestli se jedná o falešná úzká hrdla či nikoliv, si povíme v dalším vylepšení. Toto vylepšení se jmenuje **Flow Through Bottleneck Detection**. Toto vylepšení přidáváme jednak z nutnosti nové metody, která funguje i pokud jsou odhadované cesty útočníků tvořené nedeterministicky. Za



V tomto obrázku můžeme vidět situaci z jednoho experimentu, při kterém se obráncům (modré čtverečky) postupným nalézáním úzkých hrdel povedlo zabránit útočníkům v pohybu přes široké úzké hrdlo. Obránci nejprve našli úzké hrdlo označené modrou jedničkou. V pozdější iteraci pak obránci našli úzké hrdlo označené modrou dvojkou. Červené šipky znázorňují, odkud přicházejí agenti útočníci (červené čtverečky).

**Obrázek 4.3** Příklad postupného zabraňování úzkého hrdla



V tomto příkladě můžeme vidět jeden z možných deterministických pohybů agentů útočníků (červené čtverečky) na cílové pozice (čtverečky se zelenými hranami). Pohyb je znázorněn žlutými šipkami.

**Obrázek 4.4** Příklad deterministického pohybu agentů

druhé však chceme nějakou robustnější metodu, která nám poví, jestli skutečně došlo k úspěšnému zabránění úzkého hrdla. Potřebujeme zkontrolovat, jestli se průchod útočníků přes úzké hrdlo skutečně snížil či nikoliv. Na této myšlence bude založené i toto vylepšení.

Poté, co nalezneme kandidátské úzké hrdlo, potřebujeme ověřit, jestli není falešné. Spustíme následující kontrolu. Znovu necháme vygenerovat nejkratší cesty útočníků k jejich odhadnutým cílovým vrcholům (odhadnutá bijekce útočníků na jejich cílové vrcholy zůstává stejná, jako při nalezení kontrolovaného úzkého hrdla). Nyní však nezkoumáme, jestli se některá z cest změnila, avšak podíváme se jen na vrchol  $w$ , ze kterého jsme našli ověřované kandidátské úzké hrdlo.

Pokud na vrchol  $w$  při této kontrole nevstoupil vůbec žádný útočník, označíme toto úzké hrdlo za úspěšně zablokované. To je jistě dobrá kontrola. Ukázalo se však, že tato podmínka je v některých situacích příliš striktní. Zvláště pak v případech různých „křižovatek“ v grafu, ze kterých vede více cest mezi překážkami. Pokud by se obráncům podařilo nalézt úzké hrdlo v jedné z cest vedoucích z křižovatky, jedná se nejspíše o důležité úzké hrdlo, které se obráncům nalézt. Stále však nejspíše nějakí agenti skrz křižovatku budou proudit jinými cestami.

Proto přidáváme ještě následující druhou podmínku. Pokud počet útočníků, kteří se pohybovali přes  $w$  klesl na méně než  $1/3$  původního počtu útočníků, označíme úzké hrdlo také za pravé. Pokud však ani jedna z podmínek není splněná a přes  $w$  ve vygenerovaných cestách dál proudí mnoho útočníků, označíme toto úzké hrdlo za falešné.

Zvolit  $1/3$  jako konstantu pro kontrolu pohybu útočníků přes  $w$  se ukázalo jako vhodné nastavení v našich experimentech. Na jiných grafech s jinými pozicemi útočníků však může být vhodné zvolit jinou konstantu.

Tímto způsobem efektivně odhalujeme falešná úzká hrdla i při odhadování nejkratších cest útočníků pomocí vylepšení Randomized Shortest Paths.

## Consecutive False Bottleneck Termination

Nyní již máme téměř všechna vylepšení hotová. Ještě přidáme jedno poslední vylepšení, které změní podmínku na ukončení inicializace útočníků. Jmenuje se **Consecutive False Bottleneck Termination**.

V původním algoritmu se nám občas stávalo, že algoritmus na hledání úzkých hrdel nenašel správný vrchol  $w$ , od kterého by úzké hrdlo hledal. Tak se stalo, že algoritmus našel pouze jednu, nebo žádnou komponentu v grafu překážek. Proto algoritmus ukončil inicializaci obránců na úzká hrdla a všechny zbylé obránce přiřadil k libovolným cílům útočníků, i když v grafu mnohokrát ještě důležitá úzká hrdla zbývala. Ačkoli jsme se v předchozích vylepšeních snažili proti tomuto nedostatku bojovat, přesto i ve vylepšeném algoritmu toto předčasné ukončení občas nastane.

Druhý problém, který původní algoritmus měl, bylo neustálé nacházení těch stejných falešných úzkých hrdel a zpomalování tak celého procesu inicializace. Proti tomuto problému jsme také přidali několik vylepšení, které mají omezit nalézání falešných úzkých hrdel a zabránit kontrolování jednoho úzkého hrdla vícekrát. Přesto se nám však i ve vylepšeném algoritmu stávalo, že algoritmus nalézal mnoho falešných úzkých hrdel po sobě, zvláště v grafech, které nejsou pro strategii zabraňování úzkých hrdel vhodné (jsou to grafy s velkým množstvím náhodně rozmístěných překážek, jak uvidíme později).

Z těchto dvou důvodů jsme se rozhodli přidat toto poslední vylepšení Consecutive False Bottleneck Termination. Při nenalezení úzkého hrdla (nalezené úzké hrdlo bude  $\emptyset$ ) neprovedeme ukončení přiřazování obránců na úzká hrdla. Místo toho přidáme  $\emptyset$  do nalezených falešných úzkých hrdel a pokračujeme v dalším hledání úzkého hrdla. Druhá část vylepšení je ta, že pokud nalezneme falešné úzké hrdlo, zvedneme počítadlo falešných úzkých hrdel o 1. Při nalezení pravého úzkého hrdla resetujeme počítadlo na 0. Pokud nalezneme více jak 10 úzkých hrdel za sebou (počítadlo vzroste na 11), ukončíme přiřazování obránců na úzká hrdla. Zbývající obránce stejně jako v původním algoritmu přiřadíme na libovolné cílové vrcholy útočníků.

Konstanta 10 se ukázala jako vhodná. Pokud nalezneme pár falešných úzkých hrdel po sobě, není to ještě důvod k ukončení hledání úzkých hrdel. To se nám stává i na běžných grafech. Pokud však více než 10-krát po sobě nalezneme falešné úzké hrdlo, nejspíše tato mapa není pro tento typ algoritmu vhodná, nebo jsme již zcela zahradili cestu všem útočníkům na jejich cílové vrcholy. Proto přiřadíme zbývající agenty na libovolné cílové vrcholy a inicializace skončí.

## Shrnutí Bottleneck Nearest Defender Allocation

Takto jsme dokončili přehled všech vylepšení, která jsme do algoritmu Bottleneck Simulation Allocation implementovali. Celý algoritmus si nyní shrneme v následujícím pseudokódu 2.

### 4.2.4 Minimal Cut Bottleneck algoritmus

Nyní se konečně dostáváme k poslednímu algoritmu, který je založen na nové myšlence. Algoritmus se jmenuje Minimal Cut Bottleneck, protože základní myšlenka algoritmu je ta, že se snažíme nalézt úzká hrdla v grafu pomocí nástroje z teorie grafů, kterým je minimální řez. Minimální řez nám pomůže nalézt cílové vrcholy pro agenty obránce tak, abychom zabránili agentům útočníkům v přístupu na cílové vrcholy.

Pro připomenutí si zopakujeme pár pojmů z teorie grafů, které budeme dále používat.

- **Síť** je orientovaný graf  $G = (V, E)$ , obsahující
  - **zdrojový** vrchol (source)  $s$
  - **stokový** vrchol (terminal)  $t$
  - Navíc každá hrana má svou **kapacitu**  $c$ , která je nezáporná.
- **Tok** v grafu je zobrazení  $f : E \rightarrow \mathbb{R}_0^+$ , které splňuje tyto podmínky:
  - $\forall e \in E : f(e) \leq c(e)$  — tok přes hranu je maximálně tak veliký, jako je kapacita hrany.
  - $\forall v \in V; v \neq s, t : f^0(v) = 0$  — tolik, kolik přiteče do vrcholu  $v$ , tolik z něj i odeče.
- **velikost toku**  $|f| = f^\Delta(t) = -f^\Delta(s)$
- **Maximální tok** v síti je tok maximální velikosti.

---

**Algoritmus 2** Inicializace Bottleneck Nearest Defender Allocation algoritmu. Jedná se o vylepšenou verzi Bottleneck Simulation Allocation algoritmu. V rámci inicializace hledá vylepšený algoritmus úzká hrdla a přiřazuje k nim nejbližší obránce. Po ukončení vyhledávání úzkých hrdel přiřadí algoritmus zbývající obránce na libovolné cílové vrcholy útočníků.

---

```

1: function BOTTLENECK NEAREST DEFENDER ALLOCATION
2:   Data:  $G = (V; E), D, A$     ▷ Vstupní graf, množina obránců a útočníků
3:   Result: alokace cílů  $\delta^D$ 
4:    $T_{available} = \{\delta^A(a) : a \in A\}$ 
5:    $D_{available} = D$ 
6:    $F = \emptyset$                                 ▷ zakázané vrcholy
7:    $Found = \{\emptyset\}$     ▷ nalezená úzká hrdla, iniciálně obsahuje prázdné hrdlo
8:    $Terminating = false$ 
9:    $\delta' = \text{Náhodný odhad } \delta^A$             ▷ náhodná bijekce útočníci  $\leftrightarrow$  cíle
10:  while  $D_{available} \neq \emptyset \wedge \neg Terminating$  do
11:    for  $a \in A$  do
12:       $p_a = \text{nejkratšíNedeterministickáCesta}(\alpha_0(a), \delta'(a), G, F)$     ▷ Nej-
      kratší cesta hledaná pomocí  $A^*$ , ostatní agenty ve vrcholech grafu ignorujeme
13:    end for
14:     $\gamma(v) = |\{p_a : a \in A \wedge v \in p_a\}|$     ▷ Každému vrcholu spočítáme
      četnost  $\gamma$ , kolikrát se vyskytl na nějaké z nejkratších cest
15:     $w \in \text{argmax}_{v \in V} \gamma(v)$                 ▷ nejnavštěvovanější vrchol
16:     $B = \text{prohledejOkolí}(w, F)$     ▷  $B$  je Úzké hrdlo; metoda prohledejOkolí
      vrací  $\emptyset$ , pokud úzké hrdlo nebylo nalezeno
17:     $Counter = 0$                                 ▷ počítadlo falešných úzkých hrdel
18:    while  $(B \in Found \vee \neg \text{ÚzkéHrdloZabráněno}(w, f, \delta'_A, B, F))$  do    ▷
      ÚzkéHrdloZabráněno vrací TRUE, pokud přes  $w$  přejde v novém hledání
      nejkratších cest útočníků méně než 1/3 z původního počtu, jinak FALSE
19:       $Counter = Counter + 1$ 
20:      if  $Counter > 10$  then
21:         $Terminating = true$ 
22:      break
23:      end if
24:       $Found = Found \cup \{B\}$ 
25:       $f(w) = 0$                                 ▷  $w$  již nebude nejnavštěvovanější vrchol
26:       $w \in \text{argmax}_{v \in V} f(v)$                 ▷ nové  $w$ 
27:       $B = \text{prohledejOkolí}(w, F)$                 ▷ nové úzké hrdlo
28:    end while
29:    if  $\neg Terminating$  then
30:       $D_{available} = \text{přiřadVrcholyNejbližšímObráncům}(B, D_{available})$     ▷
      Agentům z  $D_{available}$  nejbliže k  $B$  přiřadíme cílové vrcholy z  $B$ , vrací novou
      množinu  $D_{available}$  bez přiřazených agentů
31:       $F = F \cup B; Found = Found \cup \{B\}$ 
32:    end if
33:  end while
34:   $\text{přiřadVrcholyObráncům}(T_{available}, D_{available})$     ▷ Přiřadíme zbývající agenty
      na libovolné cílové vrcholy útočníků
35: end function

```

---

- **Řez sítě** (hranový) je množina hran  $R_E \subseteq E$  s následující vlastností. Po odstranění všech hran řezu  $R_E$  z grafu  $G$  vznikne graf  $G' = (V, E \setminus R_E)$ , ve kterém neexistuje orientovaná cesta mezi  $s$  a  $t$ . (Řezem jsou tedy například všechny hrany vycházející ze zdroje).
- **Kapacitu hranového řezu**  $R_E$  značíme  $c(R_E) = \sum_{e \in R_E} c(e)$ .
- **Minimální hranový řez** je řez minimální kapacity.
- **Max-flow min-cut theorem** nám dává do souvislosti maximální tok a minimální řez. Říká, že velikost maximálního toku v síti je rovná velikosti minimálního řezu v této síti. (Věta byla dokázána L. R. Fordem Jr. a D. R. Fulkersonem v roce 1956 [13].)
- **Minimální vrcholový řez** mezi vrcholy  $v, w \in G = (V, E)$  je minimální množina vrcholů  $R_V$  taková, že po odebrání těchto vrcholů z grafu  $G$  neexistuje cesta mezi vrcholy  $v$  a  $w$ .

Nyní si vysvětlíme, jak algoritmus funguje. Nejprve obarvíme graf dvěma barvami (oranžovou a modrou). Oranžové vrcholy budou ty vrcholy, na které se alespoň jeden z útočníků dokáže dostat rychleji než libovolný obránce. Modrou barvou pak budou obarvené ty vrcholy, na které se dokáže dostat alespoň jeden obránce rychleji než kterýkoli útočník.

Tohoto obarvení docílíme postupným takzvaným „rozléváním barvy“ z počátečních vrcholů obránců a útočníků. Budeme postupovat následovně. Na začátku spojíme všechny počáteční pozice obránců s novým vrcholem  $d$  a všechny pozice útočníků s novým vrcholem  $a$ . V prvním kroku algoritmu přiřadíme vrcholu  $d$  modrou barvu a vrcholu  $a$  oranžovou. Poté střídavě obarvujeme všechny dosud neobarvené vrcholy sousedící s nějakým modrým vrcholem modrou barvou a všechny dosud neobarvené vrcholy sousedící alespoň s jedním oranžovým vrcholem oranžovou barvou. To opakujeme dokud stále existují vrcholy, do kterých existuje cesta z  $d$  nebo z  $a$ . Algoritmus je popsán následujícími pseudokódy 3 a 4. Příklad jednoho konkrétního obarvení grafu je zobrazen v obrázku 4.5.

Poté, co získáme toto dvou-obarvení grafu, přichází na řadu další část algoritmu. Nejprve do obarveného grafu přidáme znovu dva pomocné vrcholy. Tentokrát to bude zdrojový vrchol  $s$  a stokový vrchol  $t$ . Vrchol  $s$  napojíme hranou na všechny vrcholy, do kterých se dokáží útočníci dostat rychleji než obránci (oranžové vrcholy). Vrchol  $t$  pak spojíme hranou se všemi cílovými vrcholy útočníků.

Zaměříme se nyní na to, proč jsme tímto způsobem připojili do grafu vrcholy  $s$  a  $t$ . Toto konkrétní napojení jsme udělali proto, že se budeme snažit nalézt cílové vrcholy pro obránce tak, aby zabránili v cestách útočníkům a zároveň aby se obránci na své cílové vrcholy dostali rychleji, než na ně dorazí útočníci. Proto zdrojový vrchol napojíme nejen na počáteční pozice útočníků, ale i na všechny pozice, na které se útočníci dokáží dostat rychleji než obránci.

Dále za použití Mengerovy věty 1 převedeme problém hledání minimálního vrcholového řezu na problém hledání maximálního toku v grafu.

**Věta 1** (Menger, 1927). *Nechť  $G = (V, E)$  je graf a  $A, B \subseteq V$ . Pak minimální počet vrcholů oddělujících v  $G$  množinu  $A$  od  $B$  je roven maximálnímu počtu disjunktních  $A-B$  cest v  $G$ .*

---

**Algoritmus 3** Algoritmus rozlévání barvy pro obarvení grafu dvěma barvami. Oranžové vrcholy budou ty vrcholy, na které se alespoň jeden z útočníků dokáže dostat rychleji než libovolný obránce. Modrou barvou pak budou obarvené ty vrcholy, na které se dokáže dostat alespoň jeden obránce rychleji než kterýkoli útočník.

---

```

1: function ROZLÉVEJBARVU
2:   Data:  $G = (V; E), D, A$    ▷ Vstupní graf, množina útočníků a obránců
3:    $G' = (V', E')$  kde  $V' = V \cup \{a, d\}; E' = E \cup \{(a, \alpha_0(r_{attacker})) : r_{attacker} \in A\} \cup \{(d, \alpha_0(r_{defender})) : r_{defender} \in D\}$ 
4:    $a[barva] = Oranzova, d[barva] = Modra$ 
5:    $H_M = \{d\}, H_O = \{a\}$            ▷ Modré a Oranžové Hraniční vrcholy
6:   while  $H_M \neq \emptyset \vee H_O \neq \emptyset$  do
7:      $H_M = \text{ObarviSousedyHranice}(G', H_M, Modra)$ 
8:      $H_O = \text{ObarviSousedyHranice}(G', H_O, Oranzova)$ 
9:   end while
10:   $G'.remove(s); G'.remove(t)$ 
11:  Return:  $G'$ 
12: end function

```

---



---

**Algoritmus 4** Pomocná funkce pro obarvení sousedů v rámci jednoho cyklu při rozlévání barvy.

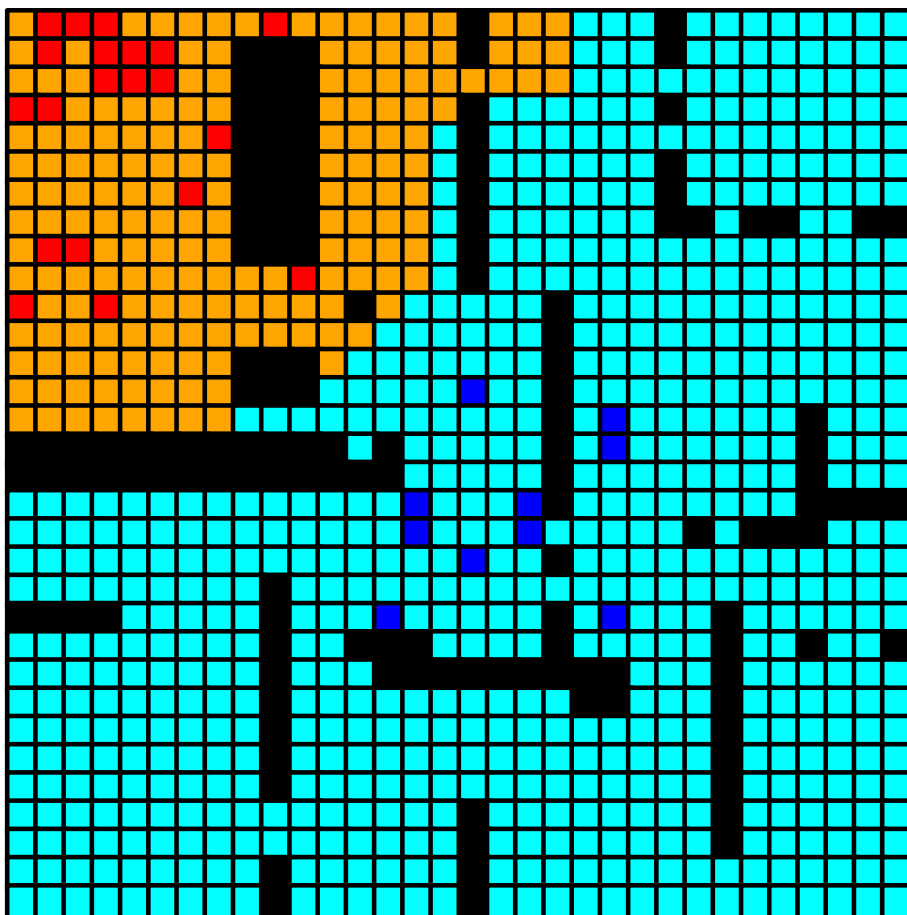
---

```

1: function OBARVISOUSEDYHRANICE
2:   Data:  $G = (V; E), H, Barva$    ▷ vstupní graf, vrcholy hranice, barva na obarvení
3:    $H_n = \emptyset$            ▷ množina pro nové hraniční vrcholy
4:   for  $h \in H$  do
5:     for  $s$  soused  $h$  do
6:       if  $s[barva] = \emptyset$  then
7:          $s[barva] = Barva$ 
8:          $H_n = H_n \cup \{s\}$ 
9:       end if
10:    end for
11:  end for
12:  Return:  $H_n$ 
13: end function

```

---



V tomto příkladě můžeme vidět počáteční pozice útočníků (červené čtverečky) a obránců (modré čtverečky). Dále je graf obarven dvěma barvami — oranžovou a světle modrou (cyan). Oranžovou barvou jsou vybarvená políčka, na které se útočníci dostanou rychleji než obránci. Světle modrou barvou jsou pak vyznačená políčka, na které se dostanou rychleji obránci než útočníci.

**Obrázek 4.5** Příklad obarvení grafu

Budeme postupovat tak, že si z grafu  $G$  vystavíme síť  $H$ , ve které budeme hledat maximální tok. Díky toku pak nalezneme minimální hranový řez. Pomocí minimálního hranového řezu v  $H$  poté nalezneme minimální vrcholový řez v  $G$ . Tento vrcholový řez následně použijeme jako cílové vrcholy pro tým obránců.

**Definice 2** (*Connectivity Algorithms*, Esfahanian [14], str. 9, algoritmus 9).

**Pomocnou síť  $H$**  postavíme z neorientovaného grafu  $G = (V_G, E_G)$  následovně. Za každý vrchol  $v \in V_G$  přidáme do sítě  $H$  dva nové vrcholy  $vA$  a  $vB$ .

Dále do sítě  $H$  přidáme orientované hrany.

- Za každý vrchol  $v \in V_G$  spojíme v síti  $H$  vrcholy  $vA$  a  $vB$  orientovanou hranou z  $vA$  do  $vB$ . Hrany  $(vA, vB)$  patří do **prvního typu** orientovaných hran v síti  $H$ .
- Každou hranu v grafu  $G$  nahradíme dvojicí orientovaných hran v  $H$ . Pro každou  $(u, v) \in E_G$  vytvoříme v  $H$  dvě hrany  $(vB, uA)$  a  $(uB, vA)$ . Tyto hrany patří do **druhého typu** hran v  $H$ .
- Žádné další typy hran v  $H$  nemáme.

Všem hranám nastavíme kapacitu  $c$  na jedna.

Dále vrchol  $sB$  označíme za **zdroj** a vrchol  $tA$  označíme za **stok**.

Nejprve si z grafu vystavíme síť  $H$ , dle definice 2. Při vytváření sítě  $H$  si zároveň budeme do slovníku ukládat mapování vrcholů v  $H$  na vrcholy v  $G$ . To nám později pomůže při zkonstruování minimálního vrcholového řezu.

Poté, co jsme si vytvořili síť  $H$ , nalezneme v síti minimální hranový řez. K tomu můžeme využít například **Fordův-Fulkersonův algoritmus**. Detaily tohoto algoritmu se zde zabývat nebudeme. Algoritmus byl popsán v článku od autorů L. R. Forda Jr. a D. R. Fulkersona v roce 1956 [13]. Díky Mengerově větě 1 víme, že velikost maximálního toku je rovna počtu vrcholů v minimálním vrcholovém řezu. Pomocí maximálního toku v  $H$  nalezneme minimální hranový řez v síti  $H$ . Nejprve ho upravíme tak, aby obsahoval pouze hrany prvního typu  $(vA, vB)$ . Následně hranový řez v síti  $H$  převedeme na vrcholový řez v grafu  $G$ .

**Tvrzení 2.** *Libovolný minimální řez v pomocné síti  $H$  můžeme upravit tak, aby obsahoval pouze hrany prvního typu  $(vA, vB)$ .*

*Důkaz.* Naším cílem je upravit minimální řez v síti  $H$ , aby obsahoval pouze hrany prvního typu  $(vA, vB)$ .

Nejprve si uvědomme, že každá hrana v minimálním řezu, která není prvním typem hrany  $(vA, vB)$ , je určitě druhým typem hrany  $(uB, vA)$ . Jiné typy hran totiž v síti  $H$  nemáme.

Postupně budeme tedy procházet hrany druhého typu  $(uB, vA)$  a budeme je nahrazovat nějakými hranami typu  $(vA, vB)$ .

Označme nyní jednu tuto hranu, kterou chceme v minimálním řezu prohodit, jako  $(xB, yA)$ . Tuto hranu můžeme nahradit buďto hranou  $(xA, xB)$ , která vede do počátečního vrcholu  $xB$  orientované hrany  $(xB, yA)$ . Nebo ji též můžeme nahradit hranou  $(yA, yB)$ , která vychází z koncového vrcholu  $yA$  orientované hrany  $(xB, yA)$ . Ať už provedeme jedno, či druhé prohození, stále se bude jednat o validní minimální hranový řez v síti  $H$ .

Hrany  $(xA, xB)$  i  $(yA, yB)$  jsou jistě hrany prvního typu  $(vA, vB)$ .

Navíc se stále jedná o řez, protože do počátečního vrcholu  $xB$  vede pouze jediná hrana  $(xA, xB)$  a z koncového vrcholu  $yA$  zase vede jediná hrana  $(yA, yB)$ . V síti  $H$  totiž neexistují hrany, které by vedly z nějakého vrcholu  $vA$  do jiného vrcholu  $uA$ , ani neexistují hrany, které by vedly z libovolného  $vB$  do libovolného  $uB$ .

Že je řez minimální, je také jasné. Pokud byl minimální před prohozením hrany, je i po prohození jedné hrany za jinou hranu se stejnou kapacitou stále minimální.

Jsou jen dva speciální případy, kdy si nemůžeme vybrat, jestli hranu druhého typu nahradíme hranou vedoucí z koncového vrcholu, či do počátečního vrcholu. Tyto případy nastanou, když je jedním z vrcholů na orientované hraně v minimálním řezu zdroj nebo stok. Pokud narazíme na tento případ, vybereme si tu hranu, která nevede mezi  $(sA, sB)$  ani mezi  $(tA, tB)$ . To je proto, že nechceme, aby byl zdroj nebo stok součástí minimálního vrcholového řezu.

Pro pochopení je výměna hrany znázorněna v obrázku 4.6. □

Díky tvrzení 2 můžeme upravit minimální řez v síti  $H$  tak, aby obsahoval pouze hrany prvního typu  $(vA, vB)$ .

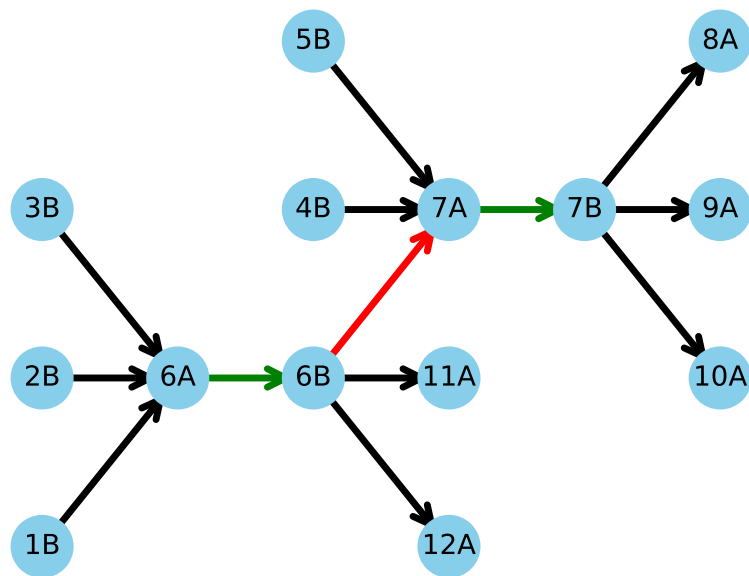
Nakonec nám zbývá už jen převést minimální hranový řez v síti  $H$  na minimální vrcholový řez v grafu  $G$ . To už však uděláme snadno, jelikož upravený minimální řez v  $H$  obsahuje pouze hrany typu  $(vA, vB)$ . Každá tato hrana přímo reprezentuje jeden vrchol v  $G$  (mapování jsme si na začátku uložili do slovníku). Proto stačí pomocí mapování převést hrany z minimálního řezu v  $H$  na vrcholy v grafu  $G$  a získali jsme minimální vrcholový řez, který jsme hledali.

Tento minimální řez představuje nejmenší množinu vrcholů, které je třeba obsadit obránci tak, aby byly znemožněny všechny cesty útočníků na cílové pozice útočníků. Vrcholy minimálního řezu se většinou nacházejí v úzkých hrdlech grafu, proto má algoritmus v názvu „bottleneck“. Když tedy máme minimální vrcholový řez nalezený, vrcholy z tohoto minimálního vrcholového řezu přiřadíme nejbližším obráncům za jejich cílové vrcholy.

Obrázek 4.7 zobrazuje pokračování příkladu z obrázku 4.5. V tomto obrázku můžeme vidět jak algoritmus pokračoval dál. Na oranžové a červené čtverečky byl připojen hranami nový vrchol  $s$  (zdroj). Na čtverečky se zeleným ohraničením (tj. cílové vrcholy útočníků) byl hranami připojen nový vrchol  $t$  (stok). Poté proběhlo hledání minimálního řezu (nejprve hranového a poté vrcholového). Výsledný vrcholový řez je též v obrázku zaznačen (fialové čtverečky).

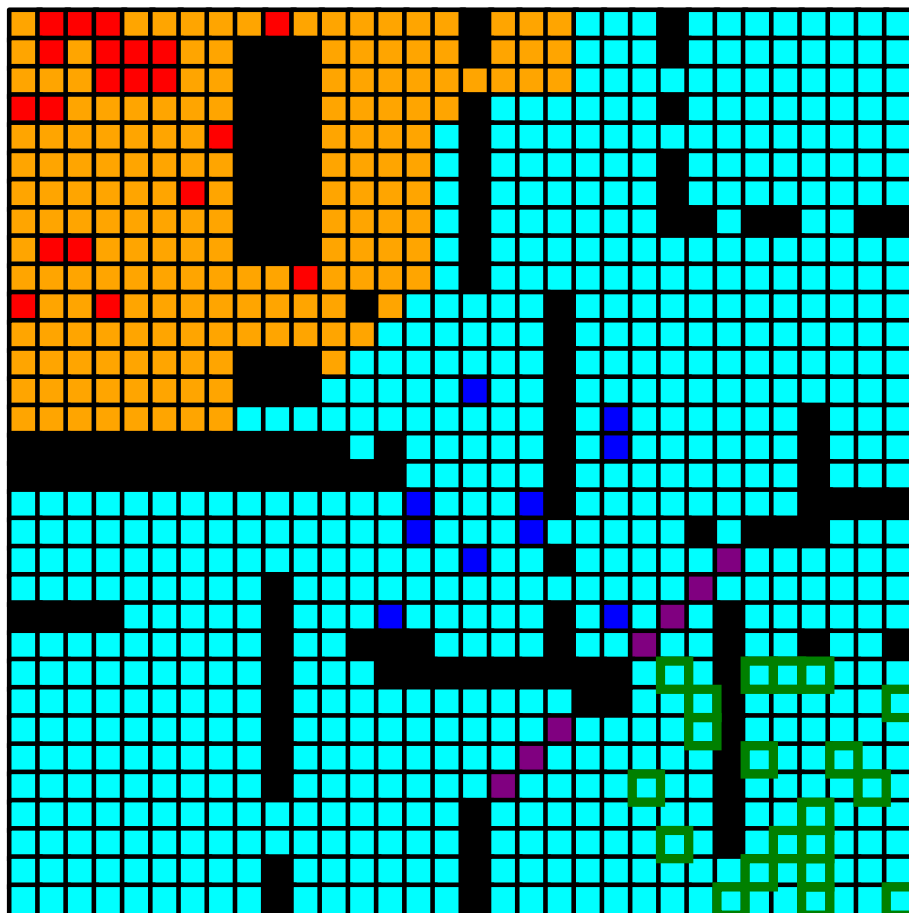
Všimněme si, že díky tomu, že jsme **zdroj** napojili na všechny vrcholy, na které se útočníci dostanou rychleji než obránci, bude minimální řez probíhat přes území, na které se obránci dostanou rychleji než útočníci. Pouze v případě, že by se nějaké cílové vrcholy útočníků nacházely v oblasti, kam se dříve dostanou útočníci, budou tyto vrcholy také součástí minimálního řezu. To je však požadované chování, protože i tyto vrcholy se snažíme zabránit před obsazením útočnickými, jen nemáme jistotu, že se na ně dostanou obránci včas.

Ostatním obráncům můžeme přiřadit jako jejich cílové vrcholy libovolné cílové vrcholy útočníků. Ukázalo se však, že je toto dodatečné přiřazení téměř vždy zbytečné. To je proto, že se útočníci přes minimální vrcholový řez většinou nestihnou dostat dřív, než je obsazený obránci. Navíc je toto přiřazení občas kontraproduktivní, protože obránci snažící se dostat na cílové vrcholy útočníků překážejí „důležitějším“ obráncům, kteří se snaží zabránit minimální řez. Poslední důvod zrušení přiřazení nadbytečných obránců na cílové vrcholy útočníků je ten,



V tomto obrázku můžeme vidět způsob, jakým se prohazují hrany typu  $(uB, vA)$  na hrany  $(vA, vB)$ . Nechtě červená hrana  $(6B, 7A)$  je hrana nalezená v minimálním řezu. Hrana je typu  $(uB, vA)$ . Prohodíme ji tedy s předchozí nebo následující hranou (v obrázku vyznačené zeleně). Obě jsou typu  $(vA, vB)$ . Jedná se stále o validní minimální řez, protože do  $6B$  vede jediná hrana a z  $7A$  také vede pouze jedna hrana.

**Obrázek 4.6** Prohození hrany v minimálním řezu



V tomto obrázku můžeme vidět pokračování příkladu z obrázku 4.5. Kromě pozic agentů a obarvení grafu zde můžeme vidět i cílové pozice útočníků (čtverečky se zeleným ohraničením). Dále v obrázku vidíme nalezený minimální vrcholový řez (fialové čtverečky). Všimněme si, že po obsazení fialových pozic obránci (modré čtverečky), zcela zaniknou všechny cesty mezi libovolným útočníkem a libovolným cílovým vrcholem útočníka. Tak se obráncům podaří splnit jejich úkol, kterým je zabránit útočníkům v dosažení jejich cílů.

**Obrázek 4.7** Nalezení minimálního vrcholového řezu

že pohyb těchto obránců prodlužuje dobu vykonávání algoritmu (doba trvání algoritmu je jeden z faktorů, který budeme u algoritmů pro obránce experimentálně porovnávat). Proto stačí ostatním obráncům přiřadit jako cílový vrchol jejich vlastní vrchol, na kterém právě stojí (tito obránce se během celé simulace nebudou hýbat).

Celý algoritmus inicializace je popsán následujícími pseudokódy 5 a 6.

---

**Algoritmus 5** Inicializace Minimal Cut Bottleneck algoritmu.

---

```

1: function MINIMALCUTBOTTLENECKINICIALIZATION
2:   Data:  $G = (V; E), D, A$     ▷ Vstupní graf, množina obránců a útočníků
3:   Result: alokace cílů  $\delta^D$ 
4:    $D_{available} = D$ 
5:    $G' = \text{RozlévejBarvu}(G, D, A)$     ▷ viz algoritmus 3
6:    $M = \text{SpočítejMinimálníŘez}(G')$ 
7:    $D_{available} = \text{přiřadVrcholyNejbližšímObráncům}(M, D_{available})$     ▷ Agentům
   z  $D_{available}$  nejbliže k  $M$  přiřadíme cílové vrcholy z  $M$ , vrací novou množinu
    $D_{available}$  bez přiřazených agentů
8:    $\text{PřiřadAgentyNaVlastníVrcholy}(D_{available})$ 
9: end function

```

---



---

**Algoritmus 6** Spočítání minimálního vrcholového řezu v grafu  $G$ .

---

```

1: function SPOČÍTEJMINIMÁLNÍŘEZ
2:   Data:  $G = (V; E), A$     ▷ obarvený graf, množina útočníků
3:    $V' = V \cup \{s, t\}$ 
4:    $E' = E \cup \{(s, v_{oranzovy}) : v_{oranzovy} \in V \wedge v_{oranzovy}.Barva == Oranzova\} \cup$ 
    $\{(t, v_{cilovy}) : v_{cilovy} \in \delta^A(A)\}$ 
5:    $G' = (V'; E')$ 
6:    $H, S = \text{VytvořPomocnouSítAMapování}(G')$     ▷  $H$  je pomocná síť,  $S$  je
   slovník s mapováním
7:    $M_{hranovy} = \text{SpočítejMinimálníHranovýŘez}(H)$ 
8:    $M_{hranovy} = \text{UpravjMinimálníHranovýŘez}(M_{hranovy})$     ▷ Aby obsahoval
   pouze hrany typu  $(vA, vB)$ 
9:    $M = \text{ZkonstruujMinimálníVrcholovýŘezZHranového}(M_{hranovy}, S, G')$ 
10:  Return:  $M$ 
11: end function

```

---

Takto jsme dokončili celou inicializaci algoritmu Minimal Cut Bottleneck. Dále probíhá klasická simulace časových kroků. Při jednom časovém kroku se pohybují všichni obránce po naplánované cestě ke svým cílovým vrcholům. Pokud nějaký obránce svůj cílový vrchol již obsadil, zůstává stát do konce simulace na místě (provádí wait akce). Pokud by měl obránce svým krokem způsobit kolizi (vstoupil by na vrchol obsazený jiným agentem), pokusí se obránce přeplánovat svou cestu. Pokud se mu to podaří, udělá krok po nové naplánované cestě. Pokud se mu to nepodaří, zůstane stát na místě.

Vždy když má agent naplánovat nebo přeplánovat svou cestu, udělá to pomocí algoritmu  $A^*$ . Jako heuristický odhad vzdálenosti používá agent manhattanskou

vzdálenost. Při plánování cesty jsou ostatní agenti v mapě považováni za statické překážky.

# 5 Vývojová dokumentace

V této kapitole si představíme, jak jsme celý projekt naprogramovali. Nejprve si rozebereme schéma celého projektu a dále se zaměříme i na detailnější prozkoumání podrobností.

## 5.1 Schéma projektu

Celý projekt jsme naprogramovali v Pythonu na operačním systému Windows 10. Projekt jsme však též zkusili spustit i na Linuxu (na distribuci Fedora) a zde program také bez problémů běží. Předpokládáme tedy, že bude program spustitelný i na jiných platformách díky přenositelnosti Pythonu a jeho knihoven.

Nyní již k samotné struktuře programu. Program je strukturován do souborů s jednotlivými třídami. V obrázku 5.1 je znázorněno schéma celého projektu, jak na sebe jednotlivé třídy navazují.

Vstupním bodem do celého programu je třída `MainModule`, která řídí běh celého programu. `Main` modul využívá ke svému fungování grafické uživatelské rozhraní (GUI modul). Dále je pak celé prostředí, kde se simulace odehrává, představováno `AMAPFEnvironment` třídou. Jednotlivým grafům, které může uživatel používat, se v našem projektu říká `mapy` (s koncovkou „.map“). O jejich načítání se stará třída `MapLoader`. Dále pak používáme pomocnou třídu `CSVFilesHandler`, která se stará o načítání a manipulaci s csv soubory, do kterých jsou ukládány všechny informace o simulaci. Dále pak ještě používáme pomocný soubor „constants“ s konstantami.

Kromě těchto tříd v projektu dále používáme interface `PluginInterface`, který slouží k tomu, aby od něj mohly dědit algoritmy útočníků a obránců. Od tohoto interface tedy dědí všechny algoritmy, které jsme implementovali jak pro pohyb útočníků, tak pro pohyb obránců.

Jednotlivé pluginy jsou do `Main` modulu nahrávány pomocí `PluginFinder` třídy, která je zodpovědná za nalezení všech tříd, které dědí od `PluginInterface` v zadané složce.

Důležitou součástí celého projektu je `PerformanceTester`, který se nespouští z GUI, ale pouze přes příkazovou řádku. Pomocí `PerformanceTesteru` můžeme spustit experimentální porovnávání všech algoritmů obránců na jednotlivých mapách.

Z `Main` modulu, který se uživatelem ovládá pomocí grafického uživatelského rozhraní (GUI), lze spouštět jednotlivé simulace. Po vytvoření instance `Main` modulu aplikace nejprve nalezne dostupné pluginy. K tomu používáme již zmíněný modul `PluginFinder`, který v zadané složce hledá potomky dědicí od interface `PluginInterface`.

Dále `Main` modul načte `mapy`. K načtení `map` se používá třída `MapLoader`, která umí nalézt všechny (.map) soubory v zadané složce. Také je tato třída zodpovědná za načítání `mapy` ze souboru do vnitřní struktury programu (do pythonovského slovníku). Slovník je později využit při tvorbě `AMAPF` prostředí.

Když jsou do `Main` modulu načtené všechny `mapy` i pluginy, vytvoří se pomocí GUI modulu grafické uživatelské rozhraní. K vykreslení ikonky a vypsání textu na

obrazovku používáme pythonní knihovnu TKinter. Poté se čeká na akci uživatele, který si může vybrat ze tří možností, které jsou mu nabízeny.

1. První možností pro uživatele je spustit simulaci na zadané mapě se zadanými počty útočníků a obránců na zadaných počátečních územích.
2. Pokud má uživatel nějaký soubor s metadaty ve správném formátu, ze kterého by chtěl simulaci spustit, může využít druhou nabízenou možnost.
3. Třetí možnost je, že si uživatel může nechat vizualizovat nějakou již proběhlou simulaci, pokud zadá správné soubory pro načtení.

Pokud si uživatel vybral první nebo druhou možnost, spouští se AMAPF simulace. Pokud uživatel zadal první možnost, načtou se z GUI zadaná data a vytvoří se AMAPF prostředí, na kterém bude simulace probíhat. Pokud uživatel zvolil druhou možnost, vytvoří se AMAPF prostředí z načtených dat ze souboru s metadaty.

Prostředí je vytvářeno pomocí třídy `AMAPFEnvironment`. Hlavní součástí prostředí je graf  $G$ , na kterém se agenti pohybují. K vytvoření a udržování grafu používáme pythonní knihovnu `NetworkX`. Další součástí prostředí jsou různé užitečné informace například o počtech a pozicích agentů, o cílech agentů nebo třeba informace o velikosti mapy.

Když máme vytvořené prostředí, vytvoříme i soubory, kam budeme ukládat informace o krocích agentů. Na práci s csv soubory používáme `CSVFilesHandler`.

Dále již probíhá samotná simulace. Nejprve se pozastaví uživatelské rozhraní. Následně jsou inicializovány vybrané algoritmy útočníků a obránců. Poté vždy střídavě provedou jeden tah obránci, a útočníci až do předem stanoveného počtu časových kroků. Tahy jsou průběžně ukládány do csv souboru.

Poté, co je simulace dokončena, nastává vizualizační část. Pokud uživatel na začátku zvolil třetí možnost, část simulace je přeskočena a rovnou se přechází na vizualizační část.

Během vizualizace jsou načtena metadata i informace o proběhlých krocích. Poté je celá simulace vizualizována v grafickém uživatelském rozhraní pomocí GUI modulu. Vizualizace je prováděna také pomocí pythonní knihovny `TKinter`.

Uživatel se z grafické vizualizace může též vrátit zpět do úvodního menu, kde si může zvolit novou možnost simulace nebo vizualizace.

### 5.1.1 Pluginy - algoritmy útočníků a obránců

Důležitou součástí celého projektu je jeho rozšiřitelnost o nové algoritmy útočníků a obránců. K tomu slouží interface `PluginInterface`. Pokud chceme rozšířit projekt o nový plugin, stačí nám ve správné složce vytvořit třídu implementující `PluginInterface`. Ke správnému fungování v pythonu používáme pro interface pythonní knihovnu `abc`.

Ať už přidáváme nový algoritmus pro útočníky nebo pro obránce, musíme implementovat následující dvě metody.

- Za prvé metodu `setup`, při které jako parametr dostane třída prostředí, ve kterém se bude simulace odehrávat. V této metodě si algoritmus může naplánovat cesty svých agentů, nebo se jinak připravit na následující simulaci.

- Za druhé metodu `make_step`, při které je algoritmu dovoleno provést jednu akci (`wait` nebo `move`) s každým agentem ze svého týmu.

Algoritmy jsou dále načítány pomocí `PluginFinder` třídy, která ověřuje, jestli se skutečně jedná o potomky `PluginInterface` třídy.

V našem projektu je struktura implementovaného algoritmu často následující. Algoritmus ve hlavní třídě implementuje zadaný interface. Tato třída pak pro každého přiřazeného agenta vytvoří instanci vlastní třídy „`SingleAgent`“. Tato instance si pamatuje svou aktuální i cílovou polohu, případně též naplánovanou cestu, pokud agent má nějaký plán. Dále je při jednotlivých algoritmech často implementována pomocná třída `A*`, díky které si agenti mohou naplánovat své cesty k cíli.

### 5.1.2 PerformanceTester

Performance tester slouží v projektu k experimentálnímu porovnávání implementovaných algoritmů pro obránce. Nespouští z GUI, ale pouze přes příkazovou řádku.

Performance tester spouští jednotlivé simulace opakovaně pro získání srovnání mezi jednotlivými algoritmy.

Tester nejprve nalezne všechny mapy, na kterých se budou algoritmy porovnávat. Pro každou mapu se následně spustí sada testů. Sada testů obsahuje všechny počáteční pozice útočníků dle omezení 3.3. Pro každou počáteční pozici spustíme simulace při různých počtech obránců. Pro každý počet obránců pak ještě simulaci několikrát opakujeme s jiným počátečním rozložením obránců, útočníků i cílů. Na tomto konkrétním počátečním rozložení pak spouštíme simulaci pro všechny algoritmy obránců, abychom je později mohli dobře porovnat.

## 5.2 Datové formáty

V této části si ještě řekneme pár slov k datovým strukturám a formátům, které používáme v naší práci.

Metadata ukládáme do csv tabulkového souboru. Metadata jsou uložena slovníkovým principem. V prvním sloupečku jsou uložena hesla, ve druhém sloupečku pak hodnoty přiřazené k heslům.

Csv soubor s metadaty musí obsahovat následující hesla a hodnoty:

- **rows**, s přiřazenou hodnotou počtu řádků v simulaci.
- **columns**, s přiřazenou hodnotou počtu sloupců v simulaci.
- **attackers count**, s přiřazenou hodnotou počtu útočníků v simulaci.
- **initial attackers id positions dict**, s přiřazeným slovníkem mapujícím id agenta útočníka na jeho pozici.
- **defenders count**, s přiřazenou hodnotou počtu obránců v simulaci.
- **initial defenders id positions dict**, s přiřazeným slovníkem mapujícím id agenta obránce na jeho pozici.

- **attackers destinations**, s přiřazeným slovníkem mapujícím id agenta útočníka na jeho cílovou pozici.
- **obstacles positions**, s přiřazeným listem s pozicemi překážek na mapě.

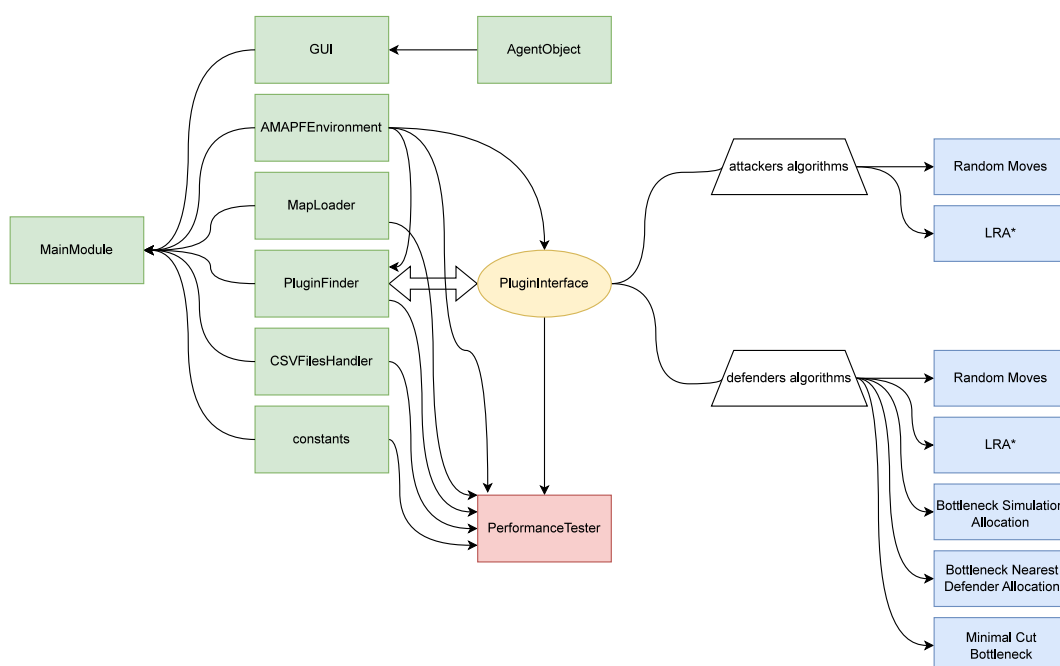
Slovníky a listy s pozicemi obsahují čárky, které slouží i jako oddělovače pro csv soubor, proto slovníky i listy před uložením vložíme do uvozovek (stringová reprezentace).

Kromě metadat ukládáme ještě informace o provedených krocích agentů do druhého souboru. Také používáme csv formát.

Soubor s provedenými kroky se generuje následovně. První řádek obsahuje názvy sloupečků. Konkrétní názvy jsou: move number, positions of attackers, positions of defenders, number of achieved destinations for attackers, number of achieved destinations for defenders.

Dále se při každém časovém kroku vygeneruje řádek s informacemi o aktuálních pozicích agentů. Do každého řádku se tedy ukládají za sebou tyto informace. Číslo aktuálního časového kroku, slovník mapující id útočníků na jejich aktuální pozice, slovník mapující id obránců na jejich aktuální pozice, číslo s počtem útočníků na cílových pozicích útočníků a číslo s počtem obránců na cílových pozicích útočníků.

Pomocí PerformanceTesteru máme možnost vygenerovat ještě poslední typ csv souboru. V tomto souboru budeme ukládat informace o doběhnutých simulacích. Tento soubor také obsahuje na prvním řádku hlavičkové informace o sloupečcích. Pro sloupečky používáme tyto názvy: map name, attackers count, defenders count, attackers position name, defenders algorithm, duration, repetition a attackers achieved destinations. Do každého řádku pak ukládáme informace o právě proběhlé simulaci. Konkrétně tedy pro každou simulaci vytvoříme zápis: jméno mapy, počet útočníků, počet obránců, jméno počáteční pozice útočníků, algoritmus použitý obránci, doba výpočtu algoritmu obránců, aktuální číslo opakování a počet útočníků, kterým se podařilo dosáhnout svých cílových pozic.



V tomto obrázku můžeme vidět schéma projektu. Vidíme zde, jak na sebe navazují jednotlivé komponenty a jak spolu jednotlivé třídy souvisí.

**Obrázek 5.1** Schéma projektu

## 6 Uživatelská dokumentace

Uživatel spustí program z příkazové řádky jednoduše pomocí příkazu „python src/main.py“. Poté se uživateli otevře uvítací obrazovka, kde si uživatel může nastavit jednotlivé parametry simulace nebo vizualizace.

Po spuštění programu se uživateli otevře uvítací obrazovka (viz obrázek 6.1). Zde má uživatel tři možnosti.

Jednak se může uživatel rozhodnout spustit zcela novou simulaci. V tom případě klikne na první tlačítko a zobrazí se druhá obrazovka 6.2, kde si uživatel nastaví jednotlivé parametry simulace.

Uživatel pak může vybrat buďto předpřipravenou mapu z kolonky maps, nebo uživatel nastaví vlastní parametry mapy, tj. rows, columns a obstacles (ve formátu: x,y „mezera“ x,y). Dále si uživatel volí počet útočníků a počet obránců. Také si vybírá algoritmy útočníků a obránců. Poté uživatel určuje startovní pozice útočníků i obránců a cílové pozice útočníků (jako obdélníková území – výseče mapy (formát: levý horní roh: x,y „mezera“ pravý dolní roh: x,y)). Poté může uživatel zapnout simulaci pomocí tlačítka „START SIMULATION“.

Uživatel se také může rozhodnout spustit simulaci z vytvořeného souboru s metadaty. V tom případě klikne na druhé tlačítko a zobrazí se další obrazovka 6.3, na které uživatel zadá cestu k souboru s metadaty i cestu k souboru, kam se budou ukládat informace o jednotlivých krocích. Dále uživatel vybere algoritmus útočníků i obránců a poté spustí simulaci pomocí „LOAD AND START SIMULATION“ tlačítka.

Poslední možností uživatele je přeskočení simulace a okamžité provedení vizualizace ze souborů. V tom případě klikne uživatel na třetí tlačítko a zobrazí se mu další obrazovka 6.4, na které uživatel zadá cestu k potřebným souborům. Po kliknutí na „LOAD AND VISUALIZE“ se simulace ihned vizualizuje.

Pokud je potřeba simulaci spočítat (v případě, že uživatel zapne simulaci pomocí „START SIMULATION“ nebo pomocí „LOAD AND START SIMULATION“), přepne se obrazovka do počítacího (computing) módu a v konzoli se vypisuje, jak probíhá simulace a kolikátý časový krok je už hotový.

Po skončení simulace nebo po kliknutí na tlačítko „LOAD AND VISUALIZE“ se obrazovka přepne do vizualizačního módu 6.5.

Zde se uživatel může proklikávat jednotlivými kroky simulace a dívat se na to, jak se agenti rozhodovali v jednotlivých krocích. Útočníci jsou červené čtverečky, obránci jsou modré čtverečky, číslo ve čtverečku udává ID agenta. Cílové pozice jsou políčka se zelenými hranami, číslo nad políčkem udává ID útočníka, který se snaží tohoto políčka dosáhnout.

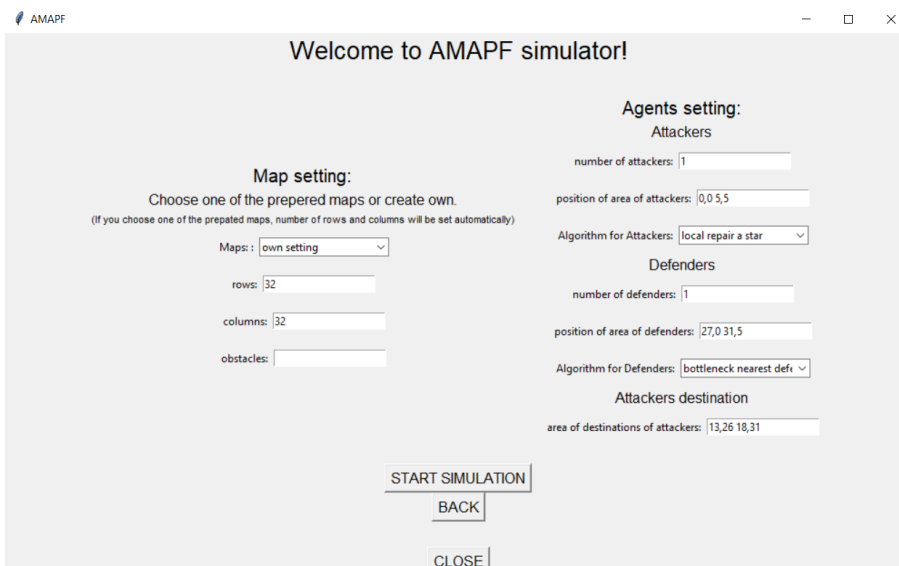
Pomocí tlačítka „next step“ se počítadlo zvýší o jedna a zobrazí se další krok simulace. Pomocí tlačítka „previous step“ se počítadlo sníží o jedna a zobrazí se minulý krok simulace. Pomocí tlačítka „run“ se počítadlo automaticky periodicky zvyšuje o jedna a zobrazují se stále další a další kroky simulace. Tlačítko se po kliknutí změní na „stop“. Po kliknutí na stop se automatická vizualizace zastaví. Pomocí tlačítka „to menu“ se uživatel přesune zpět na úvodní obrazovku.

Pokud by chtěl uživatel spustit testování pomocí PerformanceTesteru, udělá tak pomocí příkazu „python src/tester.py“.



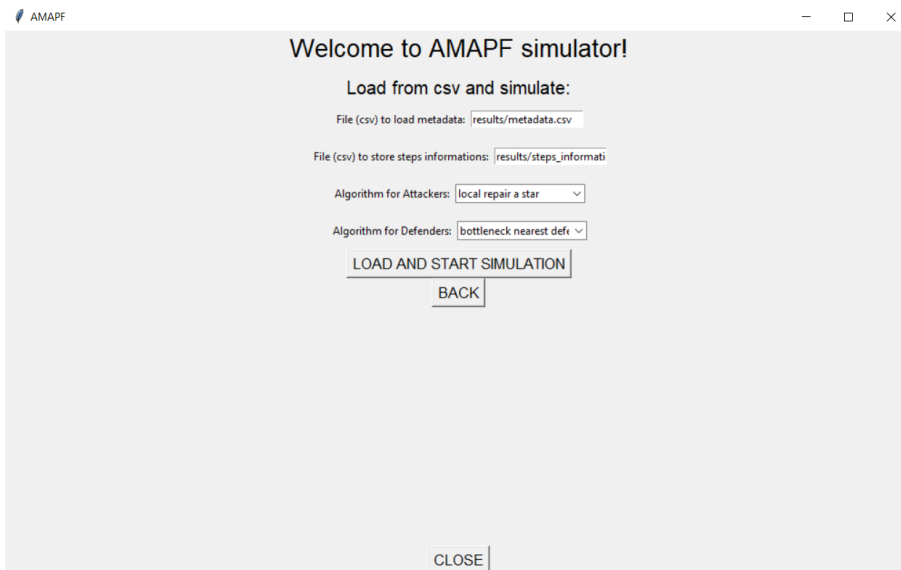
Takto vypadá uvítací obrazovka při spuštění Main modulu.

**Obrázek 6.1** Uvítací obrazovka



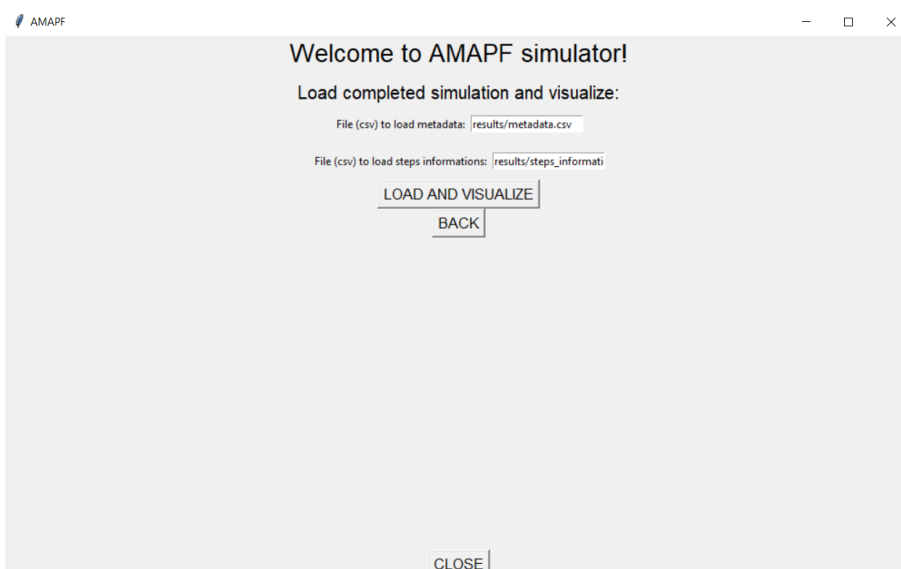
Takto vypadá uživatelské rozhraní pro start nové simulace z vlastního nastavení.

**Obrázek 6.2** Uživatelské rozhraní pro start nové simulace



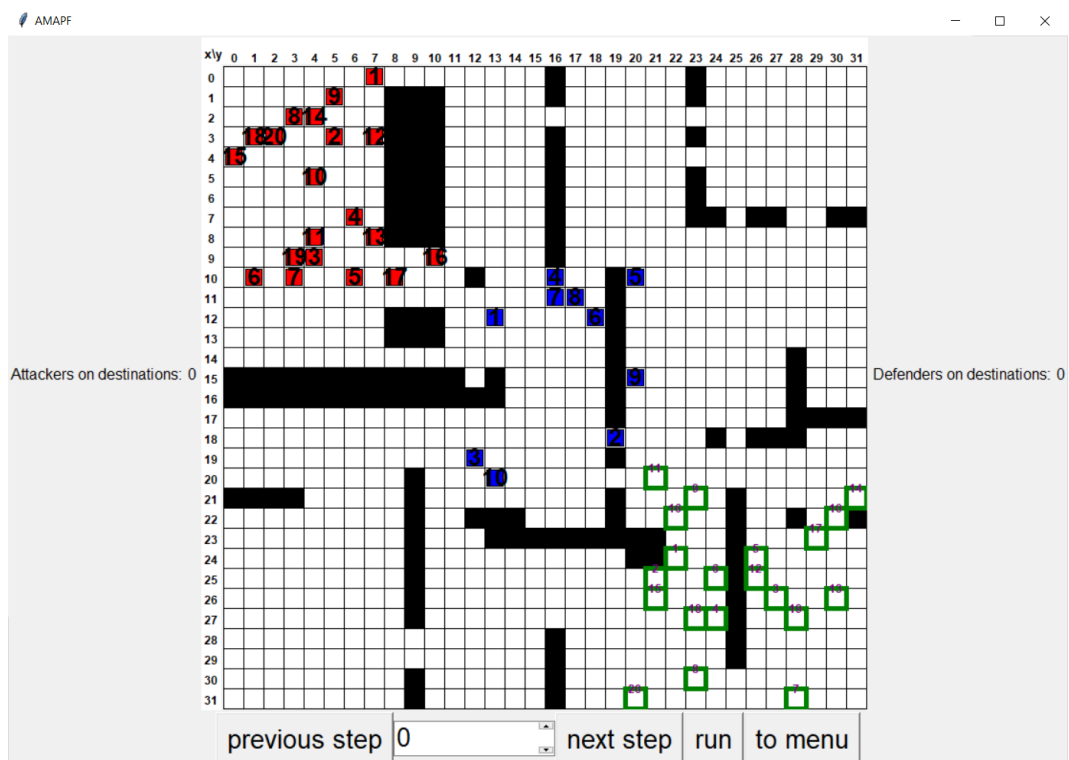
Takto vypadá uživatelské rozhraní pro start nové simulace ze souboru s metadaty.

**Obrázek 6.3** Uživatelské rozhraní pro start nové simulace z metadat souboru



Takto vypadá uživatelské rozhraní pro spuštění vizualizace ze souborů.

**Obrázek 6.4** Uživatelské rozhraní pro spuštění vizualizace



Vizualizační mód grafického uživatelského rozhraní.

Obrázek 6.5 Vizualizační mód

# 7 Experimenty a diskuse

Nyní se podíváme na to, jak jsme mezi sebou porovnávali jednotlivé algoritmy obránců. Nejprve si vysvětlíme, jak jsme nastavovali parametry simulací. Poté si představíme výsledky. V diskusi si vysvětlíme, co výsledky znamenají.

## 7.1 Metoda

Zde si detailně popíšeme všechny parametry simulací. Začneme výběrem algoritmů, které budeme porovnávat. Pokračovat budeme popisem map (grafů), které jsme pro porovnávání používali. Dále si řekneme, jak jsme nastavovali počty agentů. Dále si vysvětlíme, jaký jsme používali timespan. Poté si povíme i o ostatních parametrech, které jsme nastavovali.

### 7.1.1 Porovnávané algoritmy

V experimentální části budeme mezi sebou porovnávat jednotlivé algoritmy obránců. Konkrétně budeme porovnávat algoritmy obránců: Local Repair A\*, Bottleneck Simulation Allocation, Bottleneck Nearest Defender Allocation a Minimal Cut Bottleneck. Proti těmto algoritmům obránců použijeme vždy jako algoritmus útočníků Local Repair A\* algoritmus. Naším cílem není udělat jednotlivé simulace spravedlivé, nýbrž porovnat algoritmy obránců mezi sebou.

### 7.1.2 Mapy

Graf, na kterém se simulace odehrává, nazýváme v experimentální části mapa. Celkem pro porovnávání algoritmů obránců používáme sedm map. Konkrétně se jedná o mapy s těmito názvy:

- **Berlin** — mapa zobrazující centrum Berlína s mnoha podlouhlými překážkami.
- **darkforest** — mapa zobrazující temný hvozd s mnoha zákoutími a překážkami nepravidelných tvarů, které jsou rozmístěné po mapě. Mapa byla použita ve hře WarCraft 3.
- **largeempty** — velká prázdná mapa bez jediné překážky.
- **losttemple** — mapa zobrazující polorozbořené stěny prastarého chrámu uprostřed lesa. Nachází se zde jak pravidelné překážky, tak překážky nepravidelných tvarů. Tato mapa byla také použita ve hře WarCraft 3.
- **Milan** — mapa zobrazující centrum Milána s náměstím uprostřed mapy. Zajímavá je kruhová textura překážek a volný prostor uprostřed mapy.
- **NewYork** — mapa zobrazuje oblast v New Yorku. Obsahuje jak podlouhlé překážky, tak překážky pravidelných obdélníkových tvarů.
- **random-64-64-20** — překážky na této mapě netvoří žádné velké bloky, jsou náhodně rozmístěné. 20 % oblasti je pokryto překážkami.

Všechny tyto mapy byly převzaty z online repozitáře <https://www.movingai.com/benchmarks/>, který spravuje Sturtevant [15]. Některé byly ještě dále upravované (zmenšené). Mapy zobrazuje obrázek 7.1.

Mapy, které jsme použili, mají velikosti od  $60 \times 60$  čtverečků (mapa `largeempty`) do  $87 \times 87$  čtverečků (mapa `losttemple`). Každá mapa představuje jiné podmínky pro obránce a útočníky. Snažili jsme se vybírat takové mapy, které budou zastupovat různé obecné typy map a bude mezi nimi velká diverzita. To proto, abychom dobře otestovali algoritmy za různých podmínek, které mohou nastat.

### 7.1.3 Počty agentů

V základní definici AMAPF problému není specifikováno, kolik útočníků a kolik obránců bude umístěno do grafu. V našich experimentech zvolíme počet útočníků na 100. Počet útočníků, kterým se podařilo dosáhnout na konci simulace cíle, tak bude též představovat procentuální úspěšnost útočníků.

Počet obránců si v různých experimentech nastavíme různě, abychom mohli otestovat, jak úspěšné jsou algoritmy při velkém i malém počtu obránců. Pokaždé však nastavíme počet obránců menší než počet útočníků. To proto, že pokud by bylo obránců stejně jako útočníků, měli by obránci i při triviálních strategiích velkou šanci na vysoké ohodnocení úspěšnosti.

Konkrétní počty obránců, se kterými budeme experimenty provádět, budou: 50, 20 a 10 obránců. Poznamenejme, že při 10 obráncích již počet útočníků desetkrát převyšuje počet obránců.

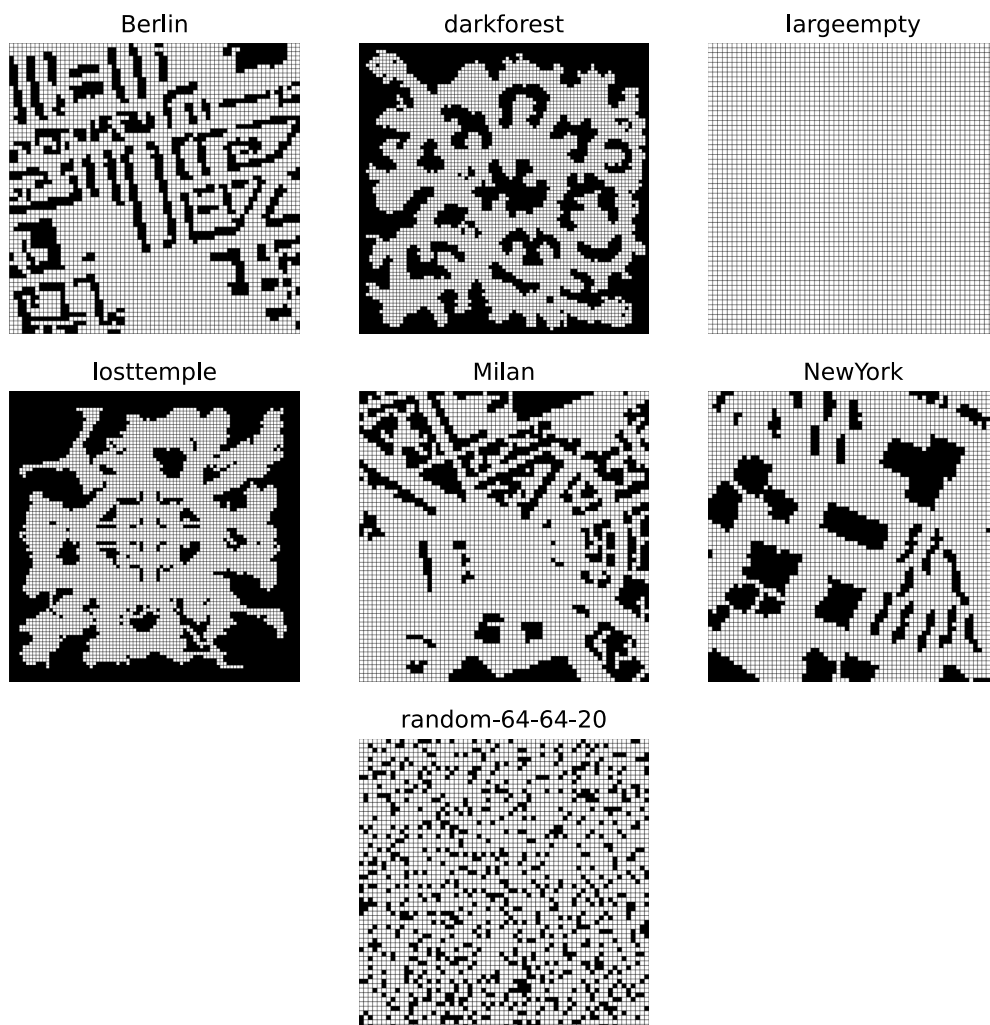
### 7.1.4 Timespan simulace

Timespan označuje počet časových kroků simulace, tedy počet tahů, které vykonají obránci i útočníci. Pro naše potřeby je vhodné stanovit timespan na 150, vzhledem k velikostem map, které používáme k experimentálnímu porovnávání algoritmů. Jedná se o počet, při kterém by měli mít všichni útočící agenti šanci dosáhnout svých cílů, pokud by jim v tom nebylo bráněno (ostatními útočícími agenty, nebo obránci). Zároveň se však jedná o počet, při kterém nemají útočníci příliš mnoho času nazbyt. To je důležité, abychom mohli dobře rozlišit, jestli se agentům obráncům podařilo alespoň částečně zabránit útočníkům v pohybu.

### 7.1.5 Ostatní parametry simulací

Mapu rozdělíme na devět čtverců dvěma svislými a dvěma vodorovnými čarami. Počáteční území obránců nastavíme (jak již bylo zmíněno v sekci o omezeních 3.3) jako prostřední čtverec mapy. Území útočníků bude v jednotlivých simulacích postupně umístěno do všech osmi krajových čtverců. Jednotlivé počáteční pozice útočníků jsou nazvány následovně (od levé horní pozice až po pravou dolní pozici): `toleft`, `topmiddle`, `topright`, `middleleft`, `middleright`, `bottomleft`, `bottommiddle` a `bottomright`. Území cílů útočníků pak bude ve čtverci, který je středově souměrný s aktuálním počátečním územím útočníků podle středového čtverce obránců.

Tímto způsobem můžeme pro každý algoritmus na každé mapě provést osm simulací. Důležité je podotknout, že konkrétní rozestavení agentů i cílů při každé konfiguraci zafixujeme do metadat souboru. To je proto, abychom mohli porovnávat různé algoritmy obránců při identických počátečních podmínkách.



Obrázek 7.1 Mapy používané pro porovnávání algoritmů

Abychom zprůměrovali jednotlivé běhy simulací, provedeme pro každou pozici počátečního území útočníků na jedné mapě pět opakování. Každé opakování bude mít jiné náhodné rozestavení všech agentů i cílů uvnitř čtverců. Na jednom tomto počátečním rozestavení pak necháme seběhnout simulace pro všechny algoritmy obránců.

Celkově tedy provádíme tyto simulace: simulujeme na sedmi mapách, při osmi počátečních pozicích útočníků, pro tři různé počty obránců (50, 20 a 10), s pěti opakováními, pro každý ze čtyř algoritmů obránců. To je dohromady  $7 * 8 * 3 * 5 * 4 = 3360$  simulací.

### 7.1.6 Měření hodnoty

Za prvé budeme na jednotlivých algoritmech obránců porovnávat úspěšnost v bránění útočníkům v obsazování jejich cílů. Úspěšnost v bránění zjistíme po doběhnutí simulace. Čím více je útočníků na cílových pozicích, tím hůře se obráncům dařilo. Naopak čím méně se jich dokázalo na cílové pozice dostat, tím je algoritmus úspěšnější.

Za druhé budeme také porovnávat dobu trvání algoritmu. Dobu trvání algoritmu spočítáme následovně. Před inicializací algoritmu obránců poprvé spustíme časovač, který pozastavíme až ve chvíli, kdy dobehne inicializace algoritmu obránců. Pak časovači zrušíme pozastavení, vždy když bude algoritmus obránců vyzván k tomu, aby udělal jeden tah s obránci. Po provedení jednoho tahu časovač znovu pozastavíme (během tahu útočníků časovač neběží). Na základě hodnoty z časovače určíme na konci simulace celkový čas běhu algoritmu obránců (v sekundách).

## 7.2 Výsledky

Výsledky jsme se rozhodli zprůměrovat ze všech pozic i opakování pro každou mapu. To proto, abychom mohli závěry simulací shrnout pouze do několika grafů a zjistit celkový trend v chování algoritmů.

Pro různé počty agentů si tedy vykreslíme tři grafy výsledků pro zobrazení průměrné úspěšnosti algoritmů 7.2, 7.3, 7.4.

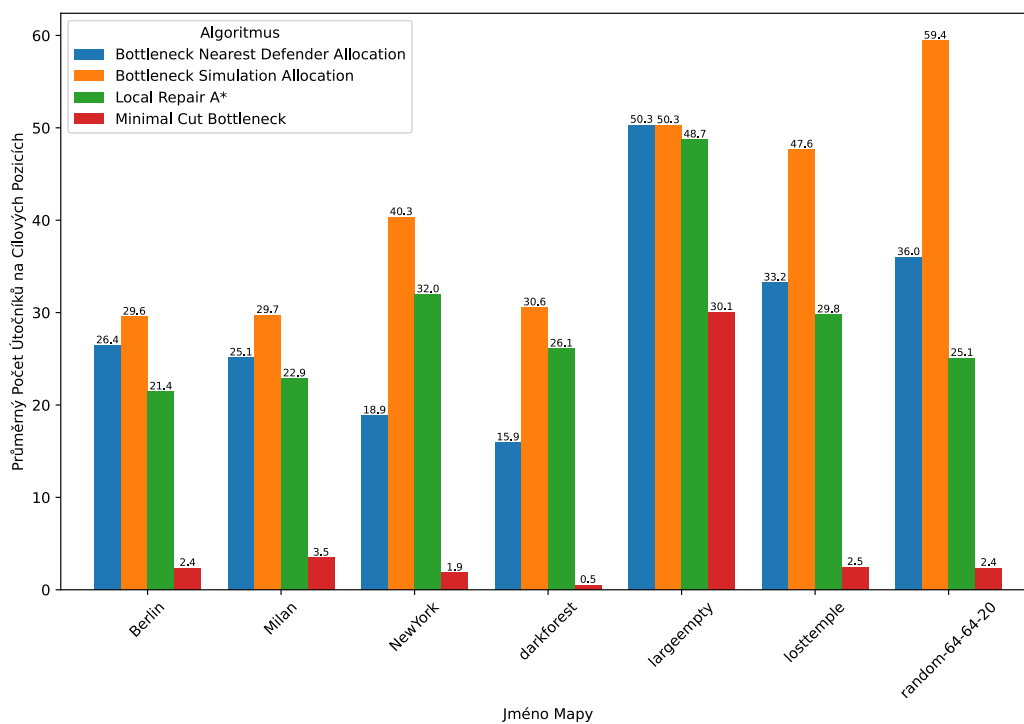
Poté si vykreslíme tři grafy výsledků pro zobrazení průměrné doby trvání algoritmů 7.5, 7.6, 7.7.

## 7.3 Diskuse

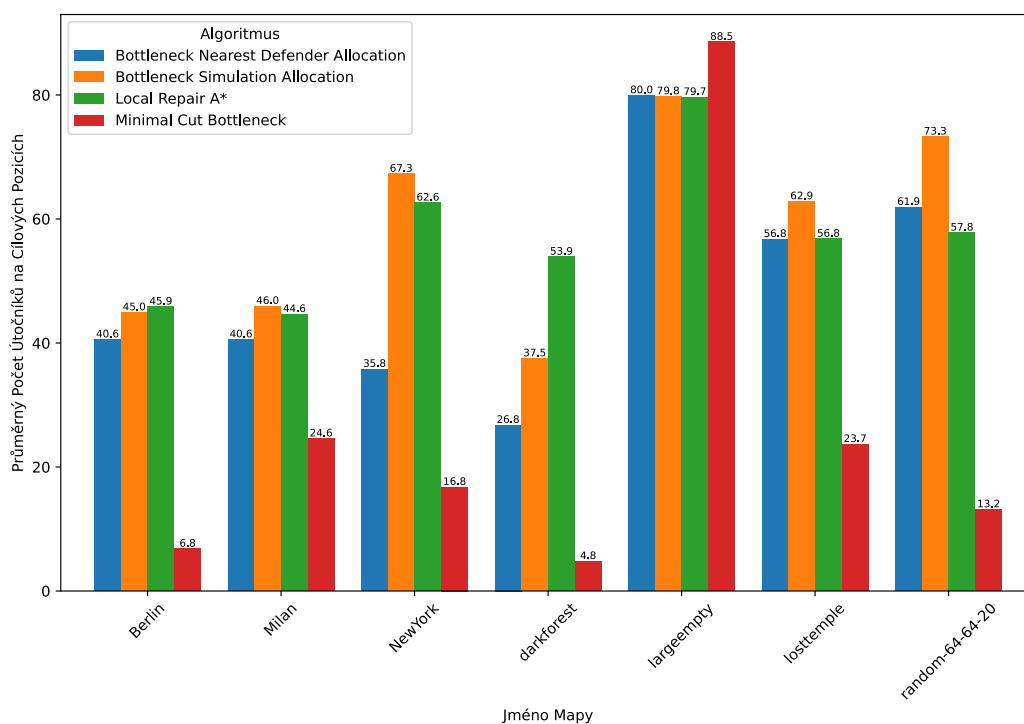
Při porovnání algoritmů obránců se ukázalo, že ve většině případů je nejefektivnějším algoritmem algoritmus **Minimal Cut Bottleneck**. Nyní si detailně rozebereme jednotlivé grafy.

### 7.3.1 Úspěšnost algoritmů při 50 obráncích

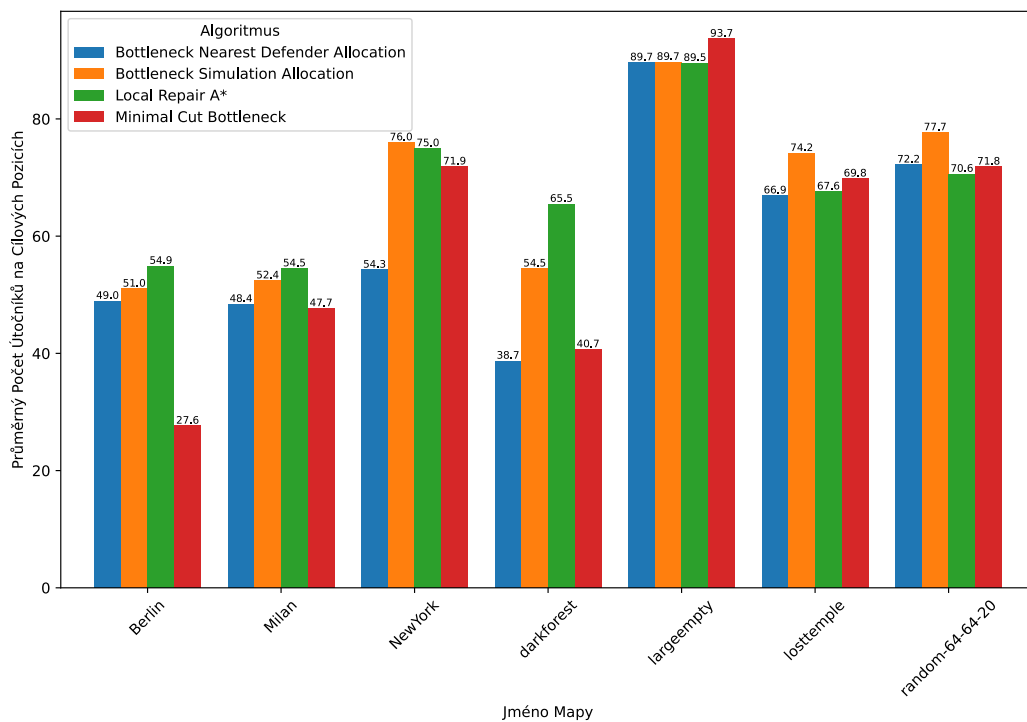
Nejprve se zaměříme na graf 7.2. Zde můžeme vidět, že nejlepší algoritmus na všech mapách byl algoritmus Minimal Cut Bottleneck. To je způsobeno tím, že se algoritmu podařilo nalézt a zastoupit minimální vrcholový řez. Kvůli tomu byli útočníci zcela odděleni od svých cílových vrcholů. Na mapě darkforest se dokonce



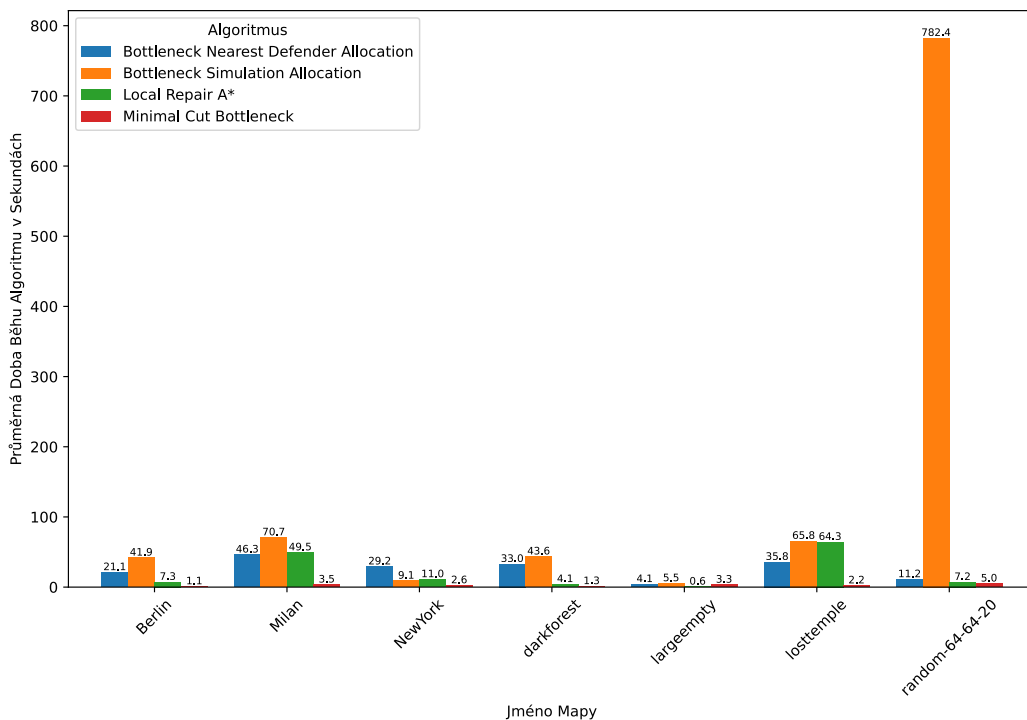
**Obrázek 7.2** Průměrný počet útočníků na cílových pozicích pro každý algoritmus a každou mapu s 50 obránci



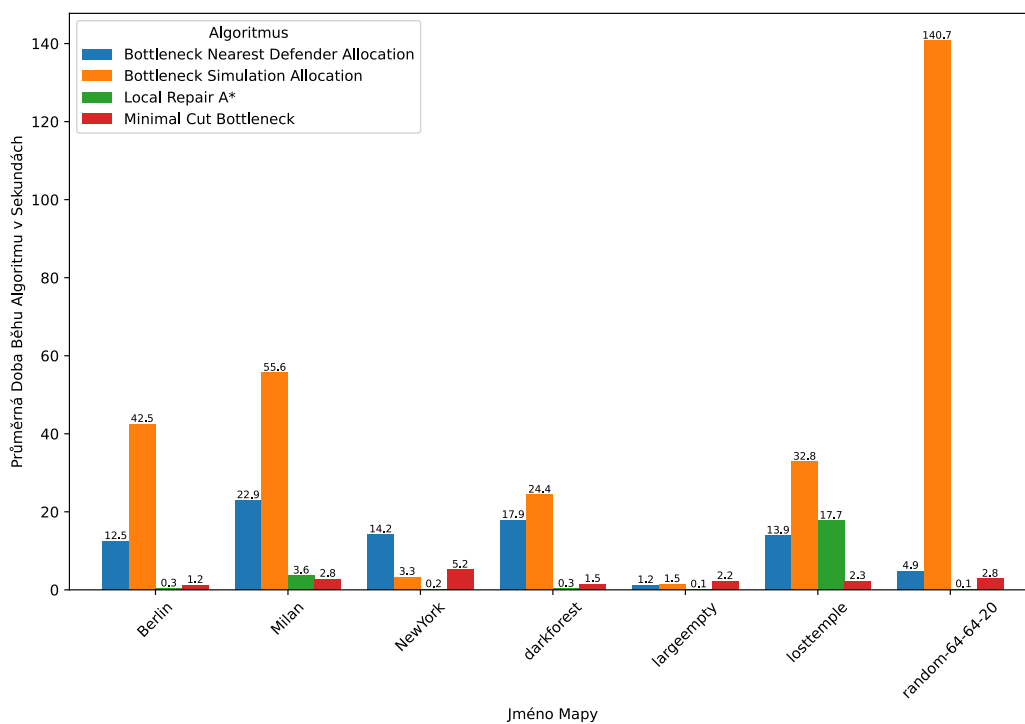
**Obrázek 7.3** Průměrný počet útočníků na cílových pozicích pro každý algoritmus a každou mapu s 20 obránci



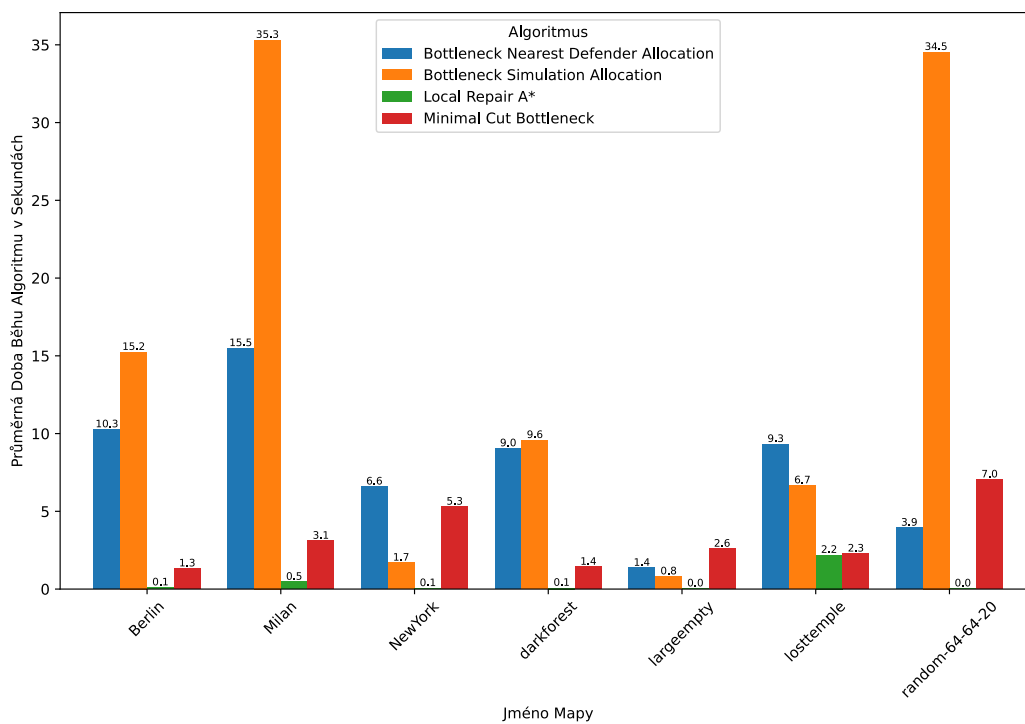
**Obrázek 7.4** Průměrný počet útočníků na cílových pozicích pro každý algoritmus a každou mapu s 10 obránci



**Obrázek 7.5** Průměrná doba trvání algoritmu s 50 obránci na mapě



Obrázek 7.6 Průměrná doba trvání algoritmu s 20 obránci na mapě



Obrázek 7.7 Průměrná doba trvání algoritmu s 10 obránci na mapě

obráncům dařilo tak dobře, že se na cílové pozice v průměru dostalo pouze 0.5 útočníka.

## Problém algoritmu Minimal Cut Bottleneck na mapě largeempty

Jediná mapa, na které měl algoritmus Minimal Cut Bottleneck trochu problém, je largeempty mapa — mapa bez překážek. Abychom si mohli vysvětlit tento propad v úspěšnosti, podíváme se na graf, který detailněji popisuje, jak probíhaly simulace na largeempty mapě (viz obrázek 7.8).

Na tomto obrázku můžeme vidět, že se agentům téměř vždy perfektně dařilo, pokud byli útočníci umístěni do rohových pozic. Ve chvíli, kdy však byli útočníci umístěni na počáteční území ve středech stran, měl algoritmus zásadní problém.

To je proto, že algoritmus sice našel minimální řez, avšak pro jeho obsazení bylo potřeba více než 50 obránců. V takové situaci obsadí útočníci řez pouze částečně a některá místa v minimálním řezu zůstanou neobsazená. To je však na této prázdné mapě veliký problém, protože těmito místy snadno proniknou na cílové pozice agenti útočníci. Proto byl v těchto situacích algoritmus tolik neúspěšný.

Řešením tohoto problému by bylo změnit chování algoritmu Minimal Cut Bottleneck následujícím způsobem. Pokud algoritmus naleznе minimální vrcholový řez větší velikosti, než kolik je aktuální počet dostupných obránců, změníme strategii a jako cílové vrcholy obránců zvolíme náhodné cílové pozice útočníků. Pak by se algoritmus v těchto případech na této mapě choval identicky jako Bottleneck Simulation Allocation nebo Bottleneck Nearest Defender Allocation. Dosáhl by tedy přibližně 50 % úspěšnosti, což by algoritmu pomohlo na pozicích topmiddle, middleleft a middleright.

Jak však uvidíme později, v jiných případech by změna strategie na výše popsanou naopak algoritmu uškodila.

## Úspěšnost ostatních algoritmů

Na druhém místě na grafech s 50 obránci se většinou umístil algoritmus Local Repair A\*, což se může zdát jako poměrně zajímavé překvapení. Je to však způsobené tím, že náš LRA\* algoritmus je „greedy“. To znamená, že obránci nemají přiřazený jeden konkrétní cílový vrchol, ale obsazují libovolné nejbližší cílové vrcholy útočníků. Díky tomu jsou nejprve obsazené nejbližší cílové vrcholy a až později vrcholy vzdálenější. V místech s mnoha překážkami vytvoří navíc obránci obsazující cíle často mimochodem nepřerušovaný zataras mezi dvěma překážkami. Tím se obráncům povede zamezit přístup útočníků do určité části mapy, nebo útočníkům přístup alespoň výrazně ztížit. Navíc se do této zatarasené oblasti již žádní obránci nebudou snažit dostat, protože obránci nemají přiřazený jeden konkrétní cílový vrchol. Proto tedy ostatní volní obránci obsadí další cílové vrcholy, což ještě posílí celkovou obranyschopnost. Tato výhoda LRA\* algoritmu je navíc zvýrazněna tím, že je obránců mnoho. Při menších počtech obránců již přestane být tato metoda tolik úspěšná.

Naproti tomu algoritmy Bottleneck Nearest Defender Allocation i Bottleneck Simulation Allocation přiřazují přebývajícím obráncům na libovolné konkrétní cílové vrcholy útočníků. Neobsazují tedy vrcholy postupně z kraje, a proto má LRA\* nad těmito algoritmy zmíněnou výhodu.

Pouze na mapách NewYork a darkforest se algoritmu Bottleneck Nearest Defender Allocation podařilo překonat algoritmus LRA\*.

Algoritmus Bottleneck Nearest Defender Allocation skončil na většině map třetí. Nejhůře dopadl algoritmus Bottleneck Simulation Allocation a to dokonce ve všech případech. Jen na mapě largeempty se algoritmus dělí o poslední místo s algoritmem Bottleneck Nearest Defender Allocation. Oba algoritmy totiž na prázdné mapě postupují zcela identickým způsobem. LRA\* má na této mapě jen o trochu lepší úspěšnost, protože algoritmus obsazuje vrcholy od nejbližšího. Tak se algoritmu LRA\* podařilo obsadit více krajních předních vrcholů, a tak se na cílové pozice dokázalo dostat průměrně o jeden a půl agenta méně.

Skutečnost, že Bottleneck Nearest Defender Allocation překonal Bottleneck Simulation Allocation, má jednoduché vysvětlení. Algoritmus Bottleneck Nearest Defender Allocation je rozšíření a vylepšení algoritmu Bottleneck Simulation Allocation, proto se bude umísťovat lépe.

Zajímavé je to, že se algoritmu Bottleneck Simulation Allocation dařilo tolik špatně. Algoritmu se takto nedařilo proto, že nedokázal nalézt dostatek důležitých úzkých hrdel. Po nalezení několika úzkých hrdel algoritmus našel prázdné hrdlo. Zbývající obránce přiřadil na náhodné cílové vrcholy útočníků. Útočníci pak jednoduše prošli kolem obránců ve volném prostoru na mapě a cílové pozice nebyly obránci příliš obsazené.

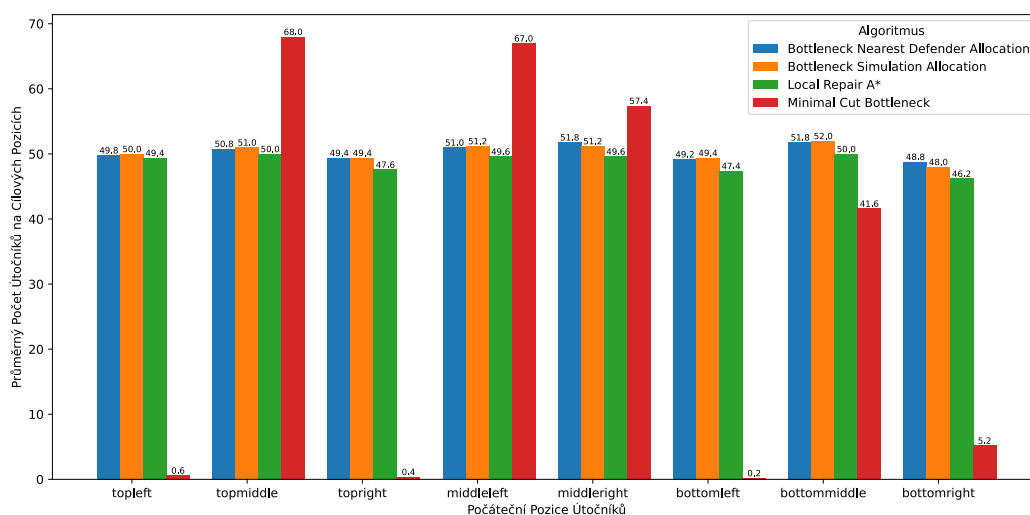
Obzvlášť na mapě random-64-64-20 se algoritmu velmi nedařilo. To je způsobeno tím, že úzkých hrdel na této mapě je příliš mnoho a algoritmus dokáže nalézt úzké hrdlo úplně v každé části mapy. Proto, když algoritmus začne úzká hrdla hledat, netvoří nalezená úzká hrdla žádný souvislý pás. V místě, kde algoritmus odhadl největší pohyb útočníků, se zatarasí předpokládané úzké hrdlo pomocí pár obránců. Zatarasení tohoto úzkého hrdla však nemá žádný veliký smysl, pokud by se nevytvořil souvislý pás zatarasů. Ten se však vytvořit nepodaří, protože příští nejfrekventovaněji navštěvovaný vrchol bude nejspíše nalezen na jiném místě v mapě, než přímo vedle právě nalezeného úzkého hrdla. Konkrétní příklad neúspěšného bránění na mapě random-64-64-20 můžeme vidět na obrázku 7.9.

Tento problém algoritmus Bottleneck Nearest Defender Allocation nemá díky svým vylepšením (Randomized Shortest Paths, Flow Through Bottleneck Detection a Consecutive False Bottleneck Termination). Tato vylepšení po čase hledání úzkých hrdel odhalí, že se nedaří nalézat vhodná úzká hrdla. Poté algoritmus přiřadí všechny obránce na náhodné cílové pozice útočníků a ukončí inicializaci. Tato strategie je na mapě random-64-64-20 poměrně úspěšná, protože se často stane, že se jednomu obránci podaří náhodou zabránit více cílových pozic najednou (cílové pozice jsou za sebou v nějakém úzkém zákoutí).

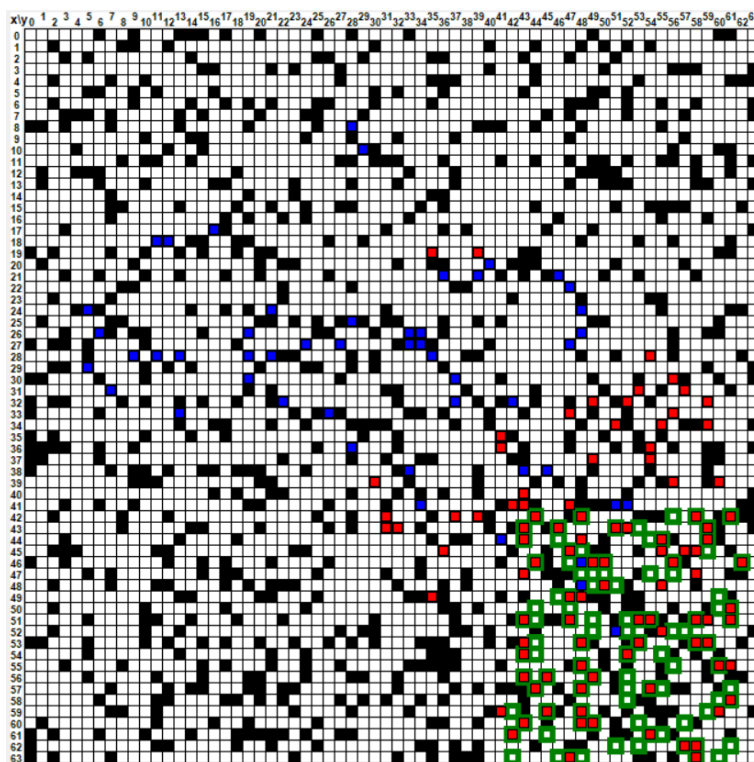
### 7.3.2 Úspěšnost algoritmů při 20 obráncích

Jak se dařilo algoritmům pro obránce při 20 obráncích zobrazuje graf 7.3.

Na všech mapách kromě largeempty byl znovu nejlepší algoritmus Minimal Cut Bottleneck. Jen na mapě largeempty se naopak tomuto algoritmu dařilo nejhůře. To je způsobeno ještě zesílením problému s minimálním řezem, který je daleko větší, než kolik je na mapě obránců. V tomto případě už obránci nemají ani nejmenší šanci zabránit celý minimální řez, a tak se útočníci pohodlně dostanou na své cílové pozice. Proč se útočníků dostane na cílové pozice jen 88.5 je způsobeno



**Obrázek 7.8** Průměrný počet útočníků na cílových pozicích pro každý algoritmus na mapě largeempty s 50 obránci



Obrázek zobrazuje typickou situaci neúspěšného bránění algoritmu Bottleneck Simulation Allocation na mapě random-64-64-20 při 50 obráncích. Obránci (modré čtverečky) sice nějaká úzká hrdla obsadili, avšak nepodařilo se jim vytvořit souvislý pás, takže kolem nich útočníci (červené čtverečky) jednoduše procházejí k cílovým vrcholům (čtverečky se zeleným ohraničením).

**Obrázek 7.9** Neúspěšné bránění algoritmem Bottleneck Simulation Allocation na mapě random-64-64-20

tím, že když se agenti obránci pokouší obsadit minimální vrcholový řez, často tento řez vede přes některé cílové vrcholy obránců. Tak se alespoň na těchto pozicích obráncům podaří zabránit pár útočnickům přístup k jejich vrcholu.

Na ostatních mapách je však stále taktika hledání minimálního vrcholového řezu jednoznačně nejlepší strategií. Často se však při simulacích v těchto mapách stane, že je vrcholový řez již větší, než jaký je počet obránců. Přesto algoritmus alespoň částečně brání tento minimální řez a je úspěšný. To je proto, že vrcholový řez je často větší jen o pár vrcholů. Obráncům se tak skoro podaří zabránit celý řez, v obraně zůstanou pouze malé skulinky (neobsazené vrcholy minimálního řezu). Mnoha útočnickům se těmito skulinkami nepodaří projít, protože útočníci si často v průchodu skulinou překážejí.

Z tohoto důvodu jsme také nezměnili algoritmus Minimal Cut Bottleneck na variantu popsanou dříve. Tato varianta algoritmu přiřazuje obránce na náhodné cílové vrcholy útočníků, pokud je minimální řez větší, než je počet obránců. To by nám při mnoha simulacích při 20 obráncích výrazně uškodilo.

Mezi ostatními třemi algoritmy nebyly na mapách již tak veliké rozdíly jako při 50 obráncích. Vidíme, že algoritmus LRA\* ztratil svou výhodu, kterou měl při 50 obráncích. Naopak na mapách Milan, NewYork, Berlin a darkforest je LRA\* překonán algoritmem Bottleneck Nearest Defender Allocation. Na mapách Berlin a darkforest je algoritmus dokonce poražen i algoritmem Bottleneck Simulation Allocation.

To je způsobeno tím, že tyto algoritmy našly úzká hrdla, díky kterým se pohyb útočníků na cílové vrcholy zpomalil, a tak se mnoha útočnickům nepodařilo dosáhnout svých cílů včas. Naopak LRA\* algoritmus již nezískal tak velkou výhodu z obsazování předních a nejbližších vrcholů, protože obránců není dostatek, aby byla tato strategie příliš efektivní.

Jen na mapě random-64-64-20 se algoritmu Bottleneck Simulation Allocation dařilo o poznání hůře než všem ostatním algoritmům. To je stále způsobeno tím, že algoritmus nenalézal vhodná úzká hrdla, která by na sebe navazovala.

### 7.3.3 Úspěšnost algoritmů při 10 obráncích

Úspěšnost algoritmů pro obránce pouze při 10 obráncích zobrazuje graf 7.4. Při tomto počátečním nastavení je již obránců nedostatek k zabránění celého minimálního řezu, a proto už zde algoritmus Minimal Cut Bottleneck tolik nedominuje.

Dobře se algoritmu Minimal Cut Bottleneck dařilo na mapách Berlin, Milan a darkforest. Tyto mapy po okrajích neobsahují velká prázdná prostranství. Proto i částečné obsazení minimálního vrcholového řezu často velmi ztížilo pohyb útočníků. Kvůli tomu se mnoho útočníků nestihlo dostat na svou cílovou pozici včas před ukončením simulace.

Stále vidíme, že na mapě largeempty se algoritmu Minimal Cut Bottleneck dařilo nejhůře. Stále se zde jedná o problém příliš velkého minimálního řezu, který obránci nemají žádnou šanci ubránit.

Velice úspěšným se při 10 obráncích ukázal algoritmus Bottleneck Nearest Defender Allocation. Umisťoval se na druhé či první pozici v úspěšnosti bránění útočnickům. Obzvláště na mapě NewYork se algoritmu podařilo výrazně překonat všechny tři zbývající algoritmy. To je způsobeno tím, že tento algoritmus obsazuje

úzká hrdla postupně, podle důležitosti úzkého hrdla.

Z toho čerpá algoritmus výhodu také nad Minimal Cut Bottleneck algoritmem. Když nalezne Minimal Cut Bottleneck minimální řez, vrcholy z tohoto minimálního řezu jsou obráncům přiřazovány v náhodném pořadí. Jednotlivé vrcholy v minimálním řezu však nejsou všechny stejně důležité. Některé vrcholy v minimálním řezu jsou na okraji mapy, jiné jsou přímo uprostřed, kudy se pohybuje nejvíce útočníků. Když má algoritmus Minimal Cut Bottleneck nedostatek obránců, nechá prázdné náhodné vrcholy v minimálním řezu. To však mohou být (a často také jsou) velice důležité vrcholy. Těmito mezerami v obraně pak útočníci snadno proniknou.

Algoritmus Bottleneck Nearest Defender Allocation přiřazuje obránce ihned, jak nalezne nějaké úzké hrdlo. Protože však algoritmus nalézá úzká hrdla tam, kde je největší předpokládaný pohyb útočníků, jsou obsazena nejprve důležitá úzká hrdla. Když na méně důležitá úzká hrdla už počet obránců nevystačí, není to až tak veliký problém.

Mezi úspěšností algoritmů Bottleneck Simulation Allocation a LRA\* už při 10 obráncích nebyl žádný velký rozdíl. Na mapách Berlin, Milan a darkforest se lépe dařilo algoritmu Bottleneck Simulation Allocation (algoritmu se alespoň občas podařilo zabránit důležitému úzkému hrdlu). Naopak na mapách NewYork, losttemple a random-64-64-20 se mírně lépe dařilo LRA\* algoritmu. To je proto, že na mapě random-64-64-20 se algoritmu Bottleneck Simulation Allocation obecně nedaří. Mapy NewYork a losttemple zas obsahují příliš mnoho volných prostranství, a tak se zde algoritmu Bottleneck Simulation Allocation nedařilo při deseti obráncích nalézt vhodná úzká hrdla, která by zkomplikovala útočníkům cestu.

### 7.3.4 Doba trvání algoritmů s 50 obránci

Nyní se zaměříme na dobu trvání algoritmů. Začneme s porovnáváním doby běhu algoritmů při 50 obráncích (znázorněno v grafu 7.5).

Zde si můžeme na první pohled všimnout, že algoritmus Bottleneck Simulation Allocation měl na mapě random-64-64-20 extrémně dlouhou dobu trvání. To jsme předpokládali již při vymyšlení vylepšeného algoritmu Bottleneck Nearest Defender Allocation (viz sekce 4.2.3).

Algoritmus je na této mapě tolik pomalý z několika důvodů. Random-64-64-20 je mapa, na které je náhodně rozmístěno mnoho překážek a tyto překážky netvoří žádné souvislé úseky. Proto, když se algoritmus rozhodne prohledat okolí kolem nalezeného  $w$ , nejspíše nalezne mnoho komponent souvislosti v grafu překážek. Algoritmus pak pokračuje tak, že si zvolí jednu komponentu souvislosti jako počáteční a zbylé komponenty označí jako cílové. Poté algoritmus hledá nejkratší cestu z počáteční komponenty do libovolné cílové komponenty.

Tímto způsobem ale většinou algoritmus nenalezne žádné vhodné úzké hrdlo. Často dokonce nalezne hrdlo, přes které zrovna nevede žádná odhadnutá cesta útočníků. Proto algoritmus označí nalezené hrdlo za falešné. Vrchol  $w$  pak vyřadí z frekventovaně navštěvovaných vrcholů a pokračuje v hledání nového úzkého hrdla. Protože si však algoritmus nepamatuje množinu nalezených úzkých hrdel, může jedno falešné úzké hrdlo nalézt mnohokrát za sebou.

Když se algoritmu konečně podaří nalézt úzké hrdlo, které není falešné, přidá si algoritmus toto úzké hrdlo do zakázaných vrcholů  $F$ . Poté algoritmus pokračuje

čuje novým hledáním úzkého hrdla. K nalezení nového úzkého hrdla však musí algoritmus znovu přeplánovat odhadované cesty útočníků. Tím pádem všechna  $w$ , která byla před tímto nalezeným úzkým hrdlem vyřazena z množiny frekventovaně navštěvovaných vrcholů, jsou vrácena zpět. Mnoho z těchto nevhodných  $w$  bude znovu velice frekventovaně navštěvovaných, a tak algoritmus bude falešná úzká hrdla nalézat znovu a znovu.

Algoritmus navíc nemůže skončit a přiřadit zbývající obránce na cílové vrcholy útočníků, protože úzká hrdla jsou stále nacházena, jen jsou téměř pořád označována za falešná.

Z těchto důvodů je inicializace tohoto algoritmu tolik pomalá.

I na jiných mapách můžeme vidět, že inicializace Bottleneck Simulation allocation algoritmu je často nejpomalejší, avšak nikde se už nejedná o tak extrémní případ.

Ostatní algoritmy se chovají na všech mapách poměrně rychle. Algoritmus Bottleneck Nearest Defender Allocation má občas také mírně delší inicializaci, ale není to nic zásadního. Na mapách Milan a losttemple má zvýšený čas i LRA\* algoritmus. Tento jev není způsoben dlouhou inicializací. Na těchto mapách docházelo k velmi častému hledání a přeplánování cest útočníků, kteří se ke svým cílovým vrcholům již nemohli dostat. To způsobilo nárůst času, který můžeme vidět v grafu.

Algoritmus Minimal Cut Bottleneck se chová rychle na všech mapách. To je způsobeno tím, že algoritmus nepřirazuje přebytečné obránce na žádné cílové vrcholy a místo toho je nechá stát celou simulaci na jednom místě. Pohyb je prováděn pouze s agenty, kteří putují do minimálního řezu. Poté, co tito obránci dosáhnou svých cílů, se již po zbytek simulace dále nepohybují.

### 7.3.5 Doba trvání algoritmů s 20 obránci

Jaká byla průměrná doba trvání algoritmů obránců při 20 obráncích zobrazuje graf 7.6.

Zde můžeme vidět, že rozdíly mezi dobami trvání jednotlivých algoritmů se snížily. Čas se zkrátil algoritmům Bottleneck Nearest Defender Allocation a Bottleneck Simulation Allocation. To je způsobeno jednak tím, že při méně agentech je u těchto algoritmů potřeba kratší čas na inicializaci. Dále je to způsobeno i během simulace, protože není potřeba tolika agentům plánovat cesty.

Většinou byl nejpomalejší algoritmus Bottleneck Simulation Allocation. Vidíme, že algoritmus má stále problémy s mapou random-64-64-20, které jsme vysvětlili již v předchozí diskusi. Jen na mapě NewYork je Bottleneck Nearest Defender Allocation algoritmus pomalejší, protože se algoritmu při hledání pravých úzkých hrdel lépe dařilo odfiltrovat falešná úzká hrdla. To algoritmus Bottleneck Simulation Allocation nedělá s takovou úspěšností, a proto byla zde inicializace Bottleneck Simulation Allocation algoritmu rychlejší.

Délka trvání algoritmu Minimal Cut Bottleneck zůstává stále na nízkých hodnotách, prakticky se nezměnila. To je za prvé proto, že inicializace algoritmu probíhá identicky při 50, 20 i 10 obráncích. Druhý důvod je, že při 50 obráncích zůstávalo často mnoho agentů stát na místě. Provádět akci `wait` není časově náročné. Pokud při 20 obráncích již nemáme agenty, kteří zůstávají na místě, nijak pozorovatelně se tím rychlost nezvyšuje.

Algoritmus LRA\* byl také rychlý. Pouze na mapě losttemple byl algoritmus pomalejší (mapa znázorněna na obrázku 7.1). To je způsobeno jednou speciální pozicí počátečního území útočníků a tou je bottomright. Pokud útočníci startují z pravého dolního území, cílové vrcholy se nacházejí v levém horním území. Tam si můžeme všimnout, že se nachází mnoho úzkých míst v mapě. Obráncům pak stačí obsadit pouze několik krajních cílových vrcholů útočníků a celá oblast cílů útočníků se zneprístupní. Přebývající obránci, kteří ještě neobsadili žádný vrchol, každé kolo neustále marně hledají cestu k cílovým vrcholům útočníků. Toto prohledávání celé mapy je časově poměrně náročné, a proto se průměrný čas algoritmu zvýšil.

### 7.3.6 Doba trvání algoritmů s 10 obránci

Nakonec si povíme, jaká byla průměrná doba trvání algoritmů obránců při 10 obráncích. Výsledky jsou zachycené v grafu 7.6.

Můžeme si všimnout, že doby trvání algoritmů i rozdíly mezi dobami se nadále snižují. Na některých mapách jako Milan nebo random-64-64-20 je stále zdaleka nejpomalejším algoritmem algoritmus Bottleneck Simulation Allocation. To je stále způsobeno pomalou inicializací, při které jsou opakovaně nalézána stejná falešná úzká hrdla.

Algoritmus Bottleneck Simulation Allocation je ve většině případů stále nejpomalejším algoritmem. Jen na mapách NewYork a losttemple je pomalejší algoritmus Bottleneck Nearest Defender Allocation. To je způsobeno tím, že Bottleneck Nearest Defender Allocation algoritmus častěji nalézal úzká hrdla, která agentům výrazněji ztížila cestu k cílovým vrcholům. To jednak přidalo algoritmu na úspěšnosti, avšak vedlejším efektem bylo, že si agenti obránci museli častěji přepřánovat své cesty a to algoritmus zdrželo.

Na ostatních mapách se doba trvání algoritmu Bottleneck Nearest Defender Allocation odvíjí nejvíce od délky inicializace.

Délka trvání algoritmu Minimal Cut Bottleneck zůstává stále na nízkých hodnotách, znovu se prakticky nezměnila. Inicializace algoritmu trvá stále stejně. Jediné zrychlení plyne z toho, že algoritmus při simulaci nepohybuje tolika obránci, protože jich má k dispozici pouze 10.

Algoritmus LRA\* je při 10 obráncích nejrychlejší, protože inicializace algoritmu je velmi jednoduchá. Navíc během simulace není téměř potřeba přepřánovat obráncům cesty. Jediná mapa, kde byl algoritmus nepatrně pomalejší, je losttemple. To je způsobeno stejným jevem, který jsme pozorovali už při 20 obráncích. Pokud útočníci startovali z pozice bottomright, podařilo se algoritmu LRA\* pomocí pár obránců zatarasit cílové vrcholy nalézající se v úzkém hrdle, za kterým se nalézaly všechny cílové pozice útočníků. Zbývající obránci pak museli každý tah přepřánovat cestu, což vedlo k prodloužení průměrného času algoritmu.

# Závěr

V této práci jsme se zabývali problémem multi-agentního hledání cest s protivníkem (AMAPF). Jedná se o grafový problém, při kterém proti sobě soupeří dva týmy agentů rozmístění v grafu — obránci a útočníci. Cílem útočníků je obsadit během diskrétní simulace přiřazené cílové vrcholy, zatímco cílem obránců je zabránit útočníkům v dosažení jejich cíle.

AMAPF problém jsme nejprve formálně definovali. Vycházeli jsme při tom z definice MAPF problému. Dále jsme prozkoumali vědecké práce, které se již tímto problémem zabývali a zařadili jsme AMAPF problém do širšího kontextu související problematiky.

Dále jsme si vysvětlili, jakým konkrétním podproblémem jsme se zabývali a stanovili jsme si některá omezení na graf a pozice agentů.

Poté jsme si detailně vysvětlili algoritmy zabývající se tímto problémem, které byly již dříve představeny v odborných člácích (Local Repair A\* a Bottleneck Simulation Allocation). Algoritmy jsme implementovali ve vytvořeném prostředí, které též umožňuje vizualizaci běhu simulace. Dále jsme navrhli a implementovali nové algoritmy a vylepšení pro tým obránců (Bottleneck Nearest Defender Allocation a Minimal Cut Bottleneck).

Algoritmy obránců jsme dále experimentálně porovnávali na sedmi různých mapách s různými počty obránců i s různými počátečními rozestaveními.

Výsledky měření jsme detailně rozebrali v kapitole 7. Při porovnávání algoritmů obránců jsme zjistili, že nejúspěšnějším algoritmem na většině map při různých počtech obránců, je algoritmus Minimal Cut Bottleneck. Když se algoritmu podaří nalézt minimální řez, který je menší, než jaký je celkový počet obránců, algoritmus si vedl velmi dobře.

Na prázdných mapách se algoritmu Minimal Cut Bottleneck dařilo špatně, protože minimální řez byl velmi často větší, než jaký byl počet obránců.

Při nízkém počtu obránců byl Minimal Cut Bottleneck občas překonán algoritmem Bottleneck Nearest Defender Allocation, protože algoritmus Bottleneck Nearest Defender Allocation obsazuje úzká hrdla postupně podle předpokládané důležitosti úzkého hrdla.

Algoritmus Bottleneck Simulation Allocation byl v průměru většinou překonán ostatními implementovanými algoritmy. Při nižších počtech agentů byl však algoritmus na některých mapách také poměrně úspěšný. Ukázalo se také, že na některých mapách má algoritmus velice dlouhou inicializaci, která je způsobena opakovaným nalézáním stejných falešných úzkých hrdel.

Algoritmus LRA\* byl překvapivě úspěšný při mnoha obráncích, avšak nedosahoval takové úspěšnosti jako Minimal Cut Bottleneck algoritmus. Pokud bylo obránců méně, úspěšnost LRA\* algoritmu klesala a byl často překonán kromě Minimal Cut Bottleneck algoritmem také algoritmem Bottleneck Nearest Defender Allocation.

## Další práce

Předmětem dalšího zkoumání by mohlo být zdokonalování Minimal Cut Algoritmu, aby dosahoval na prázdných mapách alespoň takové úspěšnosti, které

dosahovaly ostatní algoritmy. Dále by mohla být přidána další rozšíření do algoritmu Bottleneck Nearest Defender Allocation.

V následující práci bychom se mohli zaměřit nejen na inicializační část algoritmu ale též na zdokonalování algoritmů během simulace. Agenti by spolu mohli více spolupracovat. Například obránci by si navzájem mohli vyměňovat své cílové pozice, pokud si vzájemně překážejí na svých trasách. Toto zlepšení by mohlo být předstupněm pro takzvaný „online AMAPF“, který by mohl být předmětem dalšího zkoumání.

V naší práci jsme se totiž zabývali přiřazováním obránců na cílové pozice jen během inicializace. V průběhu simulace se už cílové pozice neměnily. V online AMAPF by obránci své cílové pozice měnili i za běhu simulace. To by mohlo obráncům pomoci v úspěšném bránění útočníkům i při veliké početní nevýhodě. I pokud by obránců nebylo dostatek na zabránění celého minimálního řezu, přesto by mohli být obránci úspěšní. Musela by však být vynalezena nová strategie na vhodné přesouvání obránců v průběhu simulace na různé cílové pozice.

Další průzkum by se též mohl zabývat větším přiblížením k reálným problémům. AMAPF problém je poměrně velká abstrakce problémů z reálného světa. Proto bychom v navazující práci mohli uvažovat agenty, kteří v jednom časovém kroku neobsazují pouze jeden vrchol v grafu ale mají nějakou plochu. Také by bylo zajímavé uvažovat agenty, kteří kromě své pozice počítají i s aktuální rychlostí agenta a směrem jeho pohybu.

Můžeme říci, že Multi-agentní hledání cest s protivníkem představuje zajímavé odvětví, které stále nabízí široké možnosti pro další zkoumání a inovace.

# Literatura

1. AGMON, N.; KAMINKA, G. A.; KRAUS, S. *Multi-Robot Adversarial Patrolling: Facing a Full-Knowledge Opponent* [online]. [cit. 2024-06-08]. Dostupné z: <https://arxiv.org/abs/1401.3903>.
2. IVANOVÁ, M.; SURYNEK, P. *Area Protection in Adversarial Path-Finding Scenarios with Multiple Mobile Agents on Graphs — a theoretical and experimental study of target-allocation strategies for defense coordination* [online]. [cit. 2024-06-03]. Dostupné z: <https://arxiv.org/abs/1708.07285>.
3. STERN, Roni; STURTEVANT, Nathan; FELNER, Ariel; KOENIG, Sven; MA, Hang; WALKER, Thayne; LI, Jiaoyang; ATZMON, Dor; COHEN, Liron; KUMAR, T.; BOYARSKI, Eli; BARTÁK, Roman. *Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks* [online]. [cit. 2024-06-08]. Dostupné z: [https://www.researchgate.net/publication/333915713\\_Multi-Agent\\_Pathfinding\\_Definitions\\_Variants\\_and\\_Benchmarks](https://www.researchgate.net/publication/333915713_Multi-Agent_Pathfinding_Definitions_Variants_and_Benchmarks).
4. RÖGER, G.; HELMERT, M. *Non-Optimal Multi-Agent Pathfinding Is Solved (Since 1984)* [online]. [cit. 2024-06-16]. Dostupné z: <https://doi.org/10.1609/socs.v3i1.18267>.
5. SURYNEK, P. *An Optimization Variant of Multi-Robot Path Planning is Intractable* [online]. [cit. 2024-06-16]. Dostupné z: <https://doi.org/10.1609/aaai.v24i1.7767>.
6. IVANOVÁ, Marika; SURYNEK, Pavel. Adversarial Multi-Agent Path Finding is Intractable. In: *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. 2021, s. 481–486. Dostupné z DOI: 10.1109/ICTAI52525.2021.00078.
7. SILVER, D. *Cooperative Pathfinding* [online]. [cit. 2024-06-08]. Dostupné z: <https://doi.org/10.1609/aiide.v1i1.18726>.
8. STERN, R. *Multi-Agent Path Finding – An Overview* [online]. [cit. 2024-06-08]. ISBN 978-3-030-33273-0. Dostupné z DOI: 10.1007/978-3-030-33274-7\_6.
9. SHARON, G.; STERN, R.; FELNER, A.; STURTEVANT, N. *Conflict-Based Search For Optimal Multi-Agent Path Finding* [online]. [cit. 2024-06-16]. Dostupné z: <https://doi.org/10.1609/aaai.v26i1.8140>.
10. BNAYA, Z.; FELNER, A.; STERN, R.; ZIVAN, R.; OKAMOTO, S. *Multi-Agent Path Finding for Self Interested Agents* [online]. [cit. 2024-06-16]. Dostupné z: <https://doi.org/10.1609/socs.v4i1.18292>.
11. IVANOVÁ, M.; SURYNEK, P. *Adversarial Cooperative Path-Finding: A First View* [online]. [cit. 2024-06-16]. Dostupné z: <https://cdn.aaai.org/ocs/ws/ws0993/7058-30727-1-PB.pdf>.
12. IVANOVÁ, Marika; SURYNEK, Pavel. Adversarial Cooperative Path-Finding: Complexity and Algorithms. In: *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*. 2014, s. 75–82. Dostupné z DOI: 10.1109/ICTAI.2014.22.

13. FORD, L. R. Jr.; FULKERSON, D. R. *Maximal Flow Through Network* [online]. [cit. 2024-06-14]. Dostupné z: [http://www.cs.yale.edu/homes/lans/readings/routing/ford-max\\_flow-1956.pdf](http://www.cs.yale.edu/homes/lans/readings/routing/ford-max_flow-1956.pdf).
14. ESFAHANIAN, A. *Connectivity Algorithms* [online]. [cit. 2024-06-14]. Dostupné z: [http://www.cse.msu.edu/~cse835/Papers/Graph\\_connectivity\\_revised.pdf](http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf).
15. STURTEVANT, Nathan R. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012, roč. 4, č. 2, s. 144–148. Dostupné z DOI: 10.1109/TCIAIG.2012.2197681.

# Seznam obrázků

3.1	Jedno z možných rozložení obránců a útočníků v grafu . . . . .	20
4.1	Příklady úzkých hrdel v grafu . . . . .	27
4.2	Příklad falešného úzkého hrdla v grafu . . . . .	27
4.3	Příklad postupného zabraňování úzkého hrdla . . . . .	30
4.4	Příklad deterministického pohybu agentů . . . . .	30
4.5	Příklad obarvení grafu . . . . .	36
4.6	Prohození hrany v minimálním řezu . . . . .	39
4.7	Nalezení minimálního vrcholového řezu . . . . .	40
5.1	Schéma projektu . . . . .	47
6.1	Uvítací obrazovka . . . . .	49
6.2	Uživatelské rozhraní pro start nové simulace . . . . .	49
6.3	Uživatelské rozhraní pro start nové simulace z metadat souboru .	50
6.4	Uživatelské rozhraní pro spuštění vizualizace . . . . .	50
6.5	Vizualizační mód . . . . .	51
7.1	Mapy používané pro porovnávání algoritmů . . . . .	54
7.2	Průměrný počet útočníků na cílových pozicích pro každý algoritmus a každou mapu s 50 obránci . . . . .	56
7.3	Průměrný počet útočníků na cílových pozicích pro každý algoritmus a každou mapu s 20 obránci . . . . .	56
7.4	Průměrný počet útočníků na cílových pozicích pro každý algoritmus a každou mapu s 10 obránci . . . . .	57
7.5	Průměrná doba trvání algoritmu s 50 obránci na mapě . . . . .	57
7.6	Průměrná doba trvání algoritmu s 20 obránci na mapě . . . . .	58
7.7	Průměrná doba trvání algoritmu s 10 obránci na mapě . . . . .	58
7.8	Průměrný počet útočníků na cílových pozicích pro každý algoritmus na mapě largeempty s 50 obránci . . . . .	61
7.9	Neúspěšné bránění algoritmem Bottleneck Simulation Allocation na mapě random-64-64-20 . . . . .	61

# Seznam použitých zkratek

MAPF — multi-agent path finding (without adversary), česky Multi-agentní hledání cest (bez protivníka), definice v sekci 1.1

AMAPF — Adversarial multi-agent path finding, česky Multi-agentní hledání cest s protivníkem, definice v sekci 1.2

LRA\* — Local Repair A\* algorithm, česky A\* algoritmus s lokálními opravami (viz sekce 2.1)

CA\* — Cooperative A\* algorithm, česky kooperativní A\* algoritmus (viz sekce 2.1)

HCA\* — Hierarchical Cooperative A\* algorithm, česky hierarchický kooperativní A\* algoritmus (viz sekce 2.1)

WHCA\* — Windowed Hierarchical Cooperative A\* algorithm, česky hierarchický kooperativní A\* algoritmus s posuvným oknem (viz sekce 2.1)

CBS — Conflict-Based Search, algoritmus pro hledání optimálního řešení MAPF problému (viz sekce 2.2.1)