Zadání bakalářské práce

Název:	Multi-agentní hledání cest pro dynamické cíle v zobecnění hry Pac-man
Student:	Lukáš Kameník
Vedoucí:	doc. RNDr. Pavel Surynek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Cílem práce je navrhnout metodu pro plánování pohybu více Pac-manů v zobecnění hry Pac-man, kteří pronásledují vystrašené duchy. U Pac-manů je třeba se vyhýbat srážkám. Formálně se tedy jedná o variantu multi-agentního hledání cest s dynamickými cíli, kdy cíle mění polohu. Jako jeden z možných přístupů se nabízí přeplánování, tj. vytvoření nového plánu pokaždé, když cíle změní polohu, které lze různě zefektivnit využitím předchozích plánů. Úkoly pro řešitele jsou následující:

1. Prostuduje relevantní literaturu k multi-agentnímu hledání cest a k technikám automatického plánování pro dynamická prostředí.
2. Na základě prostudovaných technik navrhne novou metodu nebo modifikaci existující metody pro řízení Pac-manů při pronásledování duchů.
3. Navrženou metodu implementuje formou softwarového prototypu a ověří v syntetických scénářích hry Pac-man s více Pac-many. Výsledky zanalyzuje.

–

[1] Takayuki Osa, Naohiko Sugita, Mamoru Mitsuishi: Online Trajectory Planning in Dynamic Environments for Surgical Task Automation. Robotics: Science and Systems 2014
[2] David Silver: Cooperative Pathfinding. AIIDE 2005: 117-122
[3] Jur P. van den Berg, Dave Ferguson, James Kuffner: Anytime Path Planning and Replanning in Dynamic Environments. ICRA 2006: 2366-2371

Elektronicky schválil/a Ing. Karel Klouda, Ph.D. dne 10. prosince 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Multi-agentní hledání cest pro dynamické cíle v zobecnění hry Pac-man

Lukáš Kameník

Katedra aplikované matematiky
Vedoucí práce: doc. RNDr. Pavel Surynek, Ph.D.

9. května 2021

Poděkování

Rád bych poděkoval svému vedoucímu, doc. RNDr. Pavlu Surynkovi, Ph.D., za veškerou pomoc a vedení při tvorbě bakalářské práce. Také bych chtěl poděkovat svému otci za pomoc s jazykovou korekturou této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2021 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Lukáš Kameník. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Kameník, Lukáš. *Multi-agentní hledání cest pro dynamické cíle v zobecnění hry Pac-man*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato bakalářská práce řeší problém multi-agentního hledání cest s pohyblivými cíli v zobecnění hry Pac-man. Cílem práce je navrhnout metodu pro plánování pohybu více Pac-manů, kteří pronásledují vystrašené duchy. U Pac-manů je třeba se vyhýbat srážkám. Práce se zaměřuje na způsoby, jak problém pohyblivých cílů efektivně řešit.

Pro vyřešení problému multi-agentního hledání cest s pohyblivými cíli byly použity algoritmy od Davida Silvera, jmenovitě Cooperative A*, Hierarchical Cooperative A* a Windowed Hierarchical Cooperative A*, a algoritmus Generalized Adaptive A* pro efektivní řešení podobných problémů hledání cest jedním agentem. S využitím těchto algoritmů byly v této práci vytvořeny čtyři nové algoritmy, jmenovitě Cooperative Generalized Adaptive A* (CGAA*), Hierarchical Cooperative Generalized Adaptive A* (HCGAA*), Hierarchical+ Cooperative A* (H+CA*) a Hierarchical+ Cooperative Generalized Adaptive A* (H+CGAA*). Algoritmus Cooperative Generalized Adaptive A* dosahuje pravidelně nejlepších výsledků. Přínosem této práce je algoritmus, který může být efektivně použit i pro složitější problémy multi-agentního hledání cest.

Klíčová slova Pac-Man, MAPF, prohledávání s pohyblivými cíli, search-based MAPF algoritmy, modifikace Cooperative A*, Generalized Adaptive A*

Abstract

This bachelor thesis solves a multi-agent pathfinding problem with dynamic targets in a generalization of the Pac-Man game. This work aims to propose a method for planning the movement of multiple Pac-Men who chase frightened ghosts. Collisions should be avoided for Pac-Man. The work focuses on ways to effectively solve the problem of moving targets.

To solve the problem of multi-agent pathfinding with moving targets, algorithms from David Silver were used, namely Cooperative A*, Hierarchical Cooperative A* and Windowed Hierarchical Cooperative A*, and the Generalized Adaptive A* algorithm was used to effectively solve similar single-agent pathfinding problems. Using these algorithms, four new algorithms were created in this work, namely Cooperative Generalized Adaptive A* (CGAA*), Hierarchical Cooperative Generalized Adaptive A* (HCGAA*), Hierarchical+ Cooperative A * (H+CA*), and Hierarchical+ Cooperative Generalized Adaptive A* (H+CGAA*). The Cooperative Generalized Adaptive A* algorithm regularly achieves the best results. The benefit of this work is an algorithm that can be effectively used for more complex problems of multi-agent pathfinding.

Keywords Pac-Man, MAPF, moving target search, search-based MAPF algorithms, modification of Cooperative A*, Generalized Adaptive A*

Obsah

Úvod **1**

1 Cíl práce **3**

2 Teoretická východiska **5**
2.1 Pac-man 5
2.1.1 Duchové 6
2.1.2 Režimy pohybu duchů 6
2.1.3 Mapa 6
2.2 Multi-agentní hledání cest 7
2.2.1 Příklad problému 8
2.2.2 Distribuované vs. centralizované problémy 8
2.2.3 Nákladová funkce 9
2.2.4 Horní a spodní vrstva algoritmu 10
2.2.5 Algoritmy řešící MAPF 10
2.2.5.1 Optimální řešení 10
2.2.5.2 Suboptimální řešení 11
2.2.5.3 Řešení pomocí redukce 11
2.3 Heuristika 11
2.3.1 Manhattanská vzdálenost 11
2.4 A* 12
2.5 Generalized Adaptive A* 13
2.5.1 Odložené vyhodnocování 15
2.5.2 Pseudokód 15
2.6 Cooperative A* 19
2.6.1 Hierarchical Cooperative A* 20
2.6.2 Windowed Hierarchical Cooperative A* 21
2.7 Měření rychlosti algoritmu 23

3 Analýza **25**
3.1 Specifikace řešeného problému 25
3.1.1 Počáteční rozmístění Pac-manů 26
3.2 Porovnávání kvality algoritmů 26
3.3 Zvolené algoritmy 26

4 Návrh algoritmů **29**
4.1 Modifikace Generalized Adaptive A* 29
4.2 Modifikace CA* a HCA* 31
4.2.1 Okno rezervací 31
4.2.2 Priorita agentů v rezervační tabulce 32
4.2.3 Ukázka fungování algoritmů 33
4.2.4 Pseudokód 35
4.3 Algoritmy CGAA* a HCGAA* 36
4.3.1 Pseudokód 37
4.4 Algoritmy H+CA* a H+CGAA* 38
4.4.1 Pseudokód pro RRGAA* 39

5 Experimentální výsledky **43**
5.1 Implementační detaily softwarového prototypu 43
5.2 Experimenty, jejich výsledky a jejich analýza 43
5.2.1 Testování vlivu počtu Pac-manů na původní mapě . . . 45
5.2.2 Testování vlivu počtu Pac-manů na nové mapě 47
5.2.3 Testování vlivu velikosti okna 49
5.3 Analýza algoritmů 51

Závěr **53**

Bibliografie **55**

A Seznam použitých zkratek **59**

B Obsah přiloženého CD **61**

C Sofwarový prototyp **63**
C.1 Návod pro použití softwarového prototypu 63
C.2 Formát mapy 64
C.3 Ovládání softwarového prototypu 66
C.4 Soubor s výstupem z měření 66

Seznam obrázků

2.1 Originální mapa hry Pac-man [3] 7
2.2 Příklad MAPF problému 9

4.1 Příklad MAPF problému, kde musí být agentovi s číslem dva zvýšena priorita. 32
4.2 Problém, který je neřešitelný algoritmy použitými v této práci do prvního pohybu duchů 34

5.1 Nová mapa pro testování; rozměry mapy jsou 50×50 polí 44
5.2 Krabicové grafy znázorňující dobu řešení pro sto instancí problému se 30 Pac-many na původní mapě s velikostí okna 20 45
5.3 Grafy ukazující závislosti na počtu Pac-manů pro 100 instancí na původní mapě s velikostí okna 20; legenda v grafu (c) platí i pro grafy (a) a (b) 46
5.4 Grafy ukazující závislosti na počtu Pac-manů pro 50 instancí na velké mapě s velikostí okna 20; legenda v grafu (c) platí i pro grafy (a) a (b) 48
5.5 Grafy ukazující závislosti na velikosti okna pro 100 instancí na původní mapě s 15 Pac-many; legenda u grafu (c) platí i pro grafy (a) a (b) 49
5.6 Grafy ukazující závislosti na velikosti okna pro 100 instancí na původní mapě s 30 Pac-many; legenda u grafu (c) platí i pro grafy (a) a (b) 50

Seznam algoritmů

2.1 Manhattanská vzdálenost pro n-dimenzionální toroidní mapy . . 12
2.2 A* 14
2.3 Generalized Adaptive A* 19
2.4 Cooperative A* 20
2.5 Reverse Resumable A* 22

4.1 Modifikovaná procedura algoritmu GAA* 30
4.2 Modifikovaný CA* a HCA* 36
4.3 Pseudokód CGAA* a HCGAA* 37
4.4 Reverse Resumable Generalized Adaptive A* 41

Úvod

Pac-man je jedna z nejikoničtějších her, slyšela o ní i většina nehráčů. Práce se zaměřuje na variantu hry, ve které je více Pac-manů. Konkrétně na část, kdy Pac-mani chytají vystrašené duchy. Ovládání hry lze vidět jako variantu problému multi-agentního hledání cest s dynamickými cíli.

Multi-agentní hledání cest je již dlouho známou a řešenou problematikou, ve které je důležitá rychlost, za jakou je řešení nalezené. Existuje mnoho různých algoritmů, které řeší odlišné varianty problému různě rychle. Tyto algoritmy pak mohou být efektivně využity v praxi. Ať se jedná o využití u robotů ve skladu, robotů v továrně, autonomních vozidel nebo jednotek v počítačové hře.

Nejjednodušším řešením problému multi-agentního hledání cest s dynamickými cíli je přeplánování, tj. algoritmus pro nalezení cest spustit znovu vždy, když se cíle pohnou. Tato práce se zaměří na to, jak tento problém řešit efektivněji. Vzhledem k množství algoritmů, které problémy multi-agentního hledání cest řeší, se tato práce podrobně zaměřuje jen na ty algoritmy, které budou použity pro vytvoření nových algoritmů.

Nejprve budou v první kapitole zadefinovány a popsány potřebné pojmy a již existující algoritmy. V druhé kapitole budou zanalyzovány představené algoritmy a zkonkretizuji problém řešený v této práci. Poté ve třetí kapitole bude představena modifikace existujícího algoritmu nebo bude vytvořen nový algoritmus. Nakonec bude ve čtvrté kapitole tento algoritmus otestován a porovnán s nějakým již existujícím algoritmem, který přeplánuje cesty vždy, když se cíle změní.

Kapitola 1

Cíl práce

Cílem teoretické části práce je vysvětlit potřebné pojmy týkající se problému multi-agentního hledání cest s dynamickými cíli. Dále je cílem rešerše existujících algoritmů týkajících se řešeného problému. Dalším cílem teoretické části je podrobně popsat ty nalezené algoritmy, které budou dále využity v této práci. Pro tyto algoritmy dodat pseudokódy.

Cílem analýzy je popsání konkrétního problému multi-agentního hledání cest, který je řešen v této práci. Dále je cílem zdůvodnění volby konkrétních algoritmů.

Hlavním cílem praktické části je navržení nového algoritmu nebo modifikace existujícího na základě těch zanalyzovaných. Dalším cílem praktické části je implementace navrženého algoritmu a dalšího algoritmu, který byl popsán v teoretické části. V neposlední řadě je cílem experimentálně otestovat navržený algoritmus a porovnat ho s tím již existujícím. Nakonec výsledky zanalyzovat.

Kapitola 2

Teoretická východiska

Tato část bude zaměřena na zadefinovaní hry Pan-man a následně na popsání obecného problému hledání cest pro více agentů. Poté budou rozebrány různé přístupy k řešení tohoto problému. Pak bude vysvětlen termín heuristika a bude představena heuristika využívaná v této práci. Následně budou představeny algoritmy A* a jeho modifikovaná varianta Generalized Adaptive A*, které hledají cestu pro jednoho agenta. V neposlední řadě budou představeny algoritmy Cooperative A*, Hierarchical Cooperative A* a Windowed Hierarchical Cooperative A* od Davida Silvera, které řeší problematiku hledání cest více agentů. Nakonec bude v této kapitole popsáno, jak může být porovnána rychlost algoritmů.

2.1 Pac-man

Autorem hry Pac-man je „Namco“, kde devítičlenný tým vedl „Toru Iwatani“, který je považován za tvůrce Pac-mana, vydavatelé jsou „Namco“ a „Midway Games“. Hra jako taková byla vydávána opakovaně, poprvé byla uvedena jako arkádová hra v roce 1980. [1, 2]

Cílem hráče ve hře Pac-man je posbírat všechny kuličky, zatímco se vyhýbá duchům. Jakmile toho hráč dosáhne, tak postupuje do další úrovně, která je těžší. Pokud je Pac-man duchem chycen, tak ztrácí život a Pac-man i duchové se vrací do úvodních pozic, ale sebrané kuličky se neobnovují a o získané skóre se nepřichází. Ve chvíli, kdy hráč přijde o poslední život, hra končí. Duchové, kromě červeného, začínají ve společném domečku. Do tohoto místa se hráč nemůže dostat. Červený duch začíná hru na poli před domečkem a ostatní duchové odtamtud vycházejí postupně, a to v pořadí: růžový, modrý a oranžový. V [3] Jamey Pittman podrobně popisuje hru Pac-man. [3]

Pac-man má možnost se duchům bránit sebráním velké kuličky, které se v originální mapě nachází čtyři. Duchové se pak na chvíli vystraší, což způsobí, že se jejich barva změní na tmavě modrou, přestanou pronásledovat Pac-mana

a ten je může chytit. Chycení duchové se vracejí do domečku, odkud po chvíli zase vycházejí. [3]

2.1.1 Duchové

Ve hře jsou čtyři rozdílní duchové, kteří se snaží dosáhnout konkrétní cílové pozice. Ta je určitým způsobem odvozena od pozice Pac-mana. Každý duch tuto pozice vypočítává jiným způsobem. Cílová pozice červeného ducha je přímo pozice Pac-mana. Pro růžového ducha je to pozice o čtyři pole před Pac-manem. Cílová pozice modrého (tyrkysového) ducha je získána tak, že se určí vektor od červeného ducha k pozici dvě pole před Pac-manem a tento vektor se zdvojnásobí. Posledním duchem je oranžový. Jeho cílová pozice závisí na jeho vzdálenosti od Pac-mana. Pokud je ve vzdálenosti větší než osm polí, tak je jeho cílová pozice aktuální pozice Pac-mana, stejně jako u červeného ducha. Pokud je ve vzdálenosti menší než osm polí, je jeho cílové pole levý dolní roh. [3]

Pohyb duchů se řídí několika pravidly. Například duchové nikdy nejdou zpět na pole, odkud přišli. Na křižovatkách se pak rozhodnou jít na pole, které je bližší vzdušnou čarou k cílové pozici ducha. [3]

2.1.2 Režimy pohybu duchů

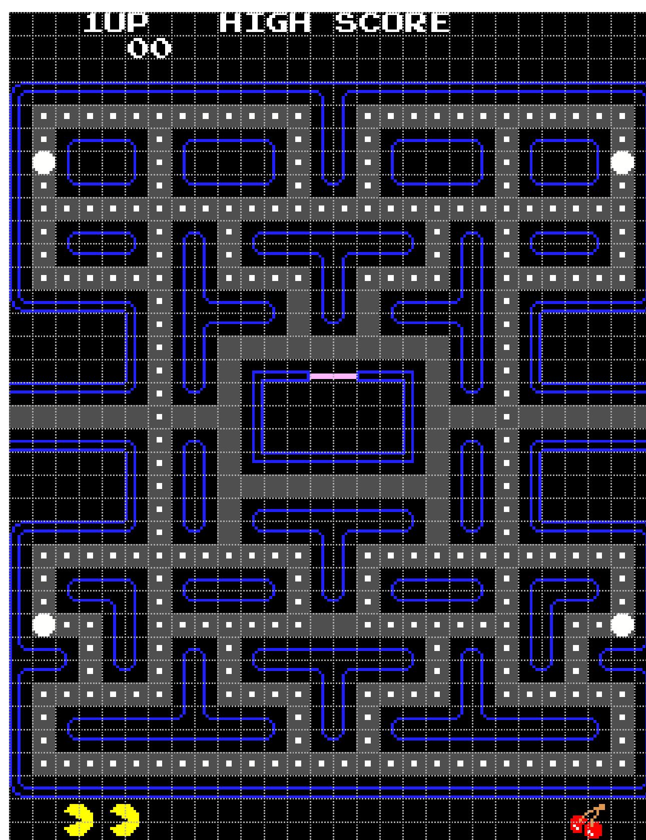
Duchové jsou vždy v jednom ze tří různých režimů: Chase (pronásledování), Scatter (rozptýlení) a Frightened (děs, vystrašení). Nejběžnější režim je Chase, v tomto režimu se duchové snaží dostat do své cílové pozice a nachází se v něm, když vychází z domečku. Periodicky se duchové přepínají na chvíli do režimu Scatter, v tu dobu se jejich cílová pozice změní na jeden roh mapy, který mají přiřazený. Okamžik změny mezi režimy Chase a Scatter je hráči signalizován změnou směru duchů – jedná se o jedinou výjimku, kdy se duch může otočit na místě. [3]

Do režimu Frightened se duchové přepnou ve chvíli, kdy Pac-man sebere velkou kuličku. Duchové v tomto režimu ignorují své cílové pozice a jejich pohyb je náhodný. Stále ale dodržují pravidlo, že se nesmí vrátit na pole, odkud přišli. Navíc jsou v tomto režimu pomalejší. [3]

2.1.3 Mapa

Hra Pac-man obsahuje jednu mapu, znázorněnou na obrázku 2.1. Na mapě je celkem 240 obyčejných kuliček a čtyři velké kuličky. Mapa má rozměry 28×36 polí, z toho je ale většina nepřístupná (zdi), přístupná jsou pro odlišení zbarvena šedě a je jich 300. První tři a poslední dva řádky neobsahují žádné přístupné pole, ale ve hře jsou využita pro vypsání informací pro hráče, například zbývající životy a skóre. [3]

Hlavní vlastností mapy je, že je tvořena pouze chodbami, které nikdy nekončí slepě. Veškeré chodby mají navíc šířku pouze jednoho pole. Další prvek



Obrázek 2.1: Originální mapa hry Pac-man [3]

je možnost průchodu krajem mapy na opačnou stranu mapy. Na mapě je jen jeden průchod, kde toho lze využít, a ten je na středu levého a pravého okraje mapy, kde hráč může projít zleva doprava a naopak. [3]

2.2 Multi-agentní hledání cest

Multi-agentní hledání cest (ang. Multi-Agent Path Finding, zk. MAPF) je problém, ve kterém je třeba nalézt cesty pro k agentů, a to tak, aby se nesrazili. Existuje mnoho způsobů, jak problém definovat [4, 5, 6, 7, 8, 9].

MAPF problém je definován jako trojce $\langle G, s, t \rangle$, kde prostor, na kterém je hledání prováděno, je definován jako graf $G(V, E)$. Vrcholy grafu jsou všechny pozice, na které může agent vstoupit. Hrany grafu jsou možné přechody mezi těmito vrcholy. Startovní pozice/vrcholy agentů jsou zobrazení $s : [1, \ldots, k] \rightarrow V$. Cílové pozice/vrcholy agentů jsou $t : [1, \ldots, k] \rightarrow V$. [4, 5]

Čas je rozdělen do časových kroků. V jednom časovém kroku jsou agenti samostatně ve vrcholech grafu G a mohou provést jednu akci. Akce je relace $a : V \rightarrow V$ taková, že $a(v) = v'$. To znamená, že pokud je agent ve vrcholu v,

tak provedením akce a se pak přesune do vrcholu v' v dalším časovém kroku. Dovolené akce jsou čekání ve stejném vrcholu (wait) a pohyb z aktuálního vrcholu v do sousedních vrcholů v' (tj. $(v, v') \in E$). [4, 5]

Pro sekvence akcí $\pi = (a_1, \ldots, a_n)$ a agenta i je $\pi_i[x]$ definováno jako pozice, ve které se bude agent nacházet po provedení prvních x akcí z π, začínající v agentově startovním vrcholu $s(i)$. Formálně, $\pi_i[x] = a_x(a_{x-1}(\ldots a_1(s(i))))$. Sekvence akcí π je plánem jednoho agenta (někdy nazýváno cestou) jen tehdy, pokud je provedení plánu zahájeno ve startovním vrcholu $s(i)$ a po dokončení plánu se agent nachází v cílovém vrcholu $t(i)$, tj. $\pi_i[|\pi|] = t(i)$. Řešením je soubor k plánů jednotlivých agentů, které nejsou v konfliktu. [4, 5]

Vybrané definice z [7] v této práci používaných omezujících podmínek jsou níže. Nechť π_i a π_j jsou plány dvou různých agentů.

1. Konflikt mezi plány π_i a π_j nastane, pokud v těchto plánech budou agenti v jednom vrcholu ve stejný čas. Formálně, konflikt mezi π_i a π_j nastane, pokud v některém časovém kroku x se $\pi_i[x] = \pi_j[x]$.

2. Konflikt mezi π_i a π_j nastane, pokud mají agenti naplánované vzájemně vyměnit své pozice ve stejný čas. Formálně, konflikt mezi π_i a π_j nastane, pokud v některém časovém kroku x se $\pi_i[x+1] = \pi_j[x]$ a zároveň $\pi_i[x] = \pi_j[x+1]$.

2.2.1 Příklad problému

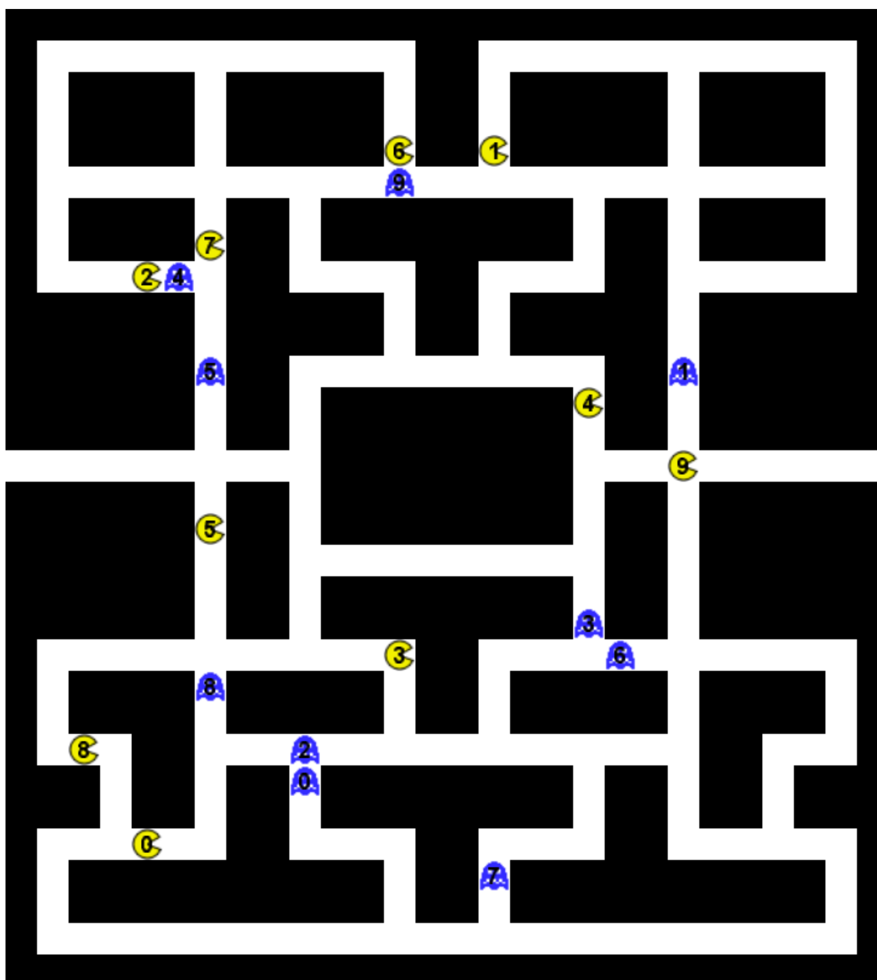
Na obrázku 2.2 je ukázka problému multi-agentního hledání cest s pohyblivými cíli. Problém může být chápán jako série podobných MAPF problémů se stacionárními cíli. Konkrétně se jedná o problém řešený v této práci, ten bude podrobně popsán v kapitole „Analýza“. Agenti jsou v tomto případě reprezentováni Pac-many a cílové vrcholy jsou duchové. Vrcholy V prohledávaného grafu $G(V, E)$ jsou reprezentovány bílými poli mapy. Jedná se o mapu hry Pac-man. Hrany E jsou mezi těmi vrcholy, kde spolu pole mapy sousedí (ne jen rohem).

2.2.2 Distribuované vs. centralizované problémy

MAPF problémy můžou být klasifikovány buď jako distribuované, nebo centralizované.

Distribuované problémy jsou takové, kde každý agent využívá k hledání cesty vlastní výpočetní sílu. Tuto cestu hledá sám, a to s využitím omezených znalostí o ostatních agentech. Spolupráce agentů probíhá prostřednictvím předávání zpráv. [9]

Na rozdíl od distribuovaných problémů je k řešení centralizovaných použita jedna výpočetní síla a algoritmus řešící MAPF problém má kompletní informace o všech agentech a celém prostředí [6, 7]. Vzhledem k podstatě pro-



Obrázek 2.2: Příklad MAPF problému

blému řešeném v této práci se bude práce zaměřovat pouze na algoritmy, které MAPF problém řeší centralizovaně.

2.2.3 Nákladová funkce

Kvalitu nalezeného řešení MAPF problému je možné posuzovat podle různých nákladových funkcí. Tato práce se bude zaměřovat na dvě nákladové funkce, a to sum-of-costs a makespan. Jedná se o nejběžněji používané přístupy. [7, 6]

Sum-of-costs je počet akcí potřebných k dosažení cílů všech agentů, kde agenti cíl už nikdy neopustí. Pro řešení $\pi = (\pi_1, \dots \pi_k)$ MAPF problému je sum-of-costs definován jako $\Sigma_{1 \leq i \leq k} |\pi_i|$. [7]

Makespan je počet časových kroků, které jsou potřeba k tomu, aby všichni agenti byli na svých cílových pozicích. Makespan je definován jako hodnota $\max_{1 \leq i \leq k} |\pi_i|$. [7]

Mezi další nákladové funkce, používané v literatuře, patří například Fuel. Tato nákladová funkce je podobná sum-of-costs s tím rozdílem, že akce wait se nezapočítává. Jedná se tedy o součet délek cest agentů bez akce wait. Tato nákladová funkce je rovná vzdálenosti, kterou agent urazí, a jedná se o ekvivalent spotřeby paliva. (Například u robota, kde spotřeba paliva při pohybu značně převyšuje spotřebu paliva v klidu.) [10]

2.2.4 Horní a spodní vrstva algoritmu

Dle [7, 6, 11] jsou některé algoritmy pro řešení MAPF problémů dvouúrovňové, lze je rozdělit na:

Horní vrstva: Anglicky „High Level solver“ je část algoritmu, která zajišťuje, aby jednotlivé plány agentů neměly konflikt. Tato část využívá spodní vrstvy pro nalezení plánů pro jednotlivé agenty.

Spodní vrstva: Anglicky „Low Level solver“ je část algoritmu, která hledá plán pro jednoho agenta s omezujícími podmínkami, které jsou této části zadané horní vrstvou algoritmu. Omezující podmínky jsou definovány konkrétním vrcholem prohledávaného grafu a časem, kdy je vrchol obsazen. Jako spodní vrstvu algoritmu lze použít například A*.

2.2.5 Algoritmy řešící MAPF

Algoritmy pro řešení MAPF problémů můžou být rozděleny do tří kategorií. První kategorii tvoří algoritmy založené na prohledávání, anglicky nazývány „search-based“, které hledají optimální řešení [7]. Druhou kategorií jsou algoritmy také založené na prohledávání, které hledají suboptimální řešení [6]. A třetí kategorií jsou takové algoritmy, které MAPF problém redukují na jiný problém, pro který existují velice efektivní řešící algoritmy [8].

2.2.5.1 Optimální řešení

Optimální řešení je takové, které má ze všech řešení nejmenší nákladovou funkci. Nejjednodušším přístupem je provést prohledávání na grafu, který kombinuje pozice všech agentů – k-agentní graf. Vrcholy v tomto k-agentním grafu jsou různé způsoby postavení agentů do vrcholů grafu, který je zadaný v MAPF problému, tak, že v každém vrcholu je nejvýše jeden agent. Počátečním, respektive koncovým vrcholem prohledávání k-agentního grafu je startovní, respektive cílová pozice všech agentů. V takovémto prostoru pak můžeme využít různých algoritmů pro prohledávání grafů, například A*.

Existují i další algoritmy pro nalezení optimálního řešení. Jedním z nich je CBS [7]. Jedná se o dvouúrovňový algoritmus. Horní vrstva je tvořena takzvaným Conflict Tree (CT). CT obsahuje v jednotlivých vrcholech stromu omezení pro agenty v konkrétním čase a místě. CT je prohledáván tak, aby

bylo nalezeno optimální řešení. Spodní vrstva může být libovolný algoritmus, který dokáže nalézt optimální plán pro jednoho agenta se seznamem omezujících podmínek získaných od horní vrstvy. [7]

2.2.5.2 Suboptimální řešení

Algoritmy pro hledání suboptimálních řešení se zaměřují na to dávat co nejlepší řešení, tj. řešení s hodnotou nákladové funkce blízké optimu. V některých případech tyto algoritmy můžou dát výrazně horší výsledky, než má optimální řešení, nebo řešení nenaleznou vůbec.

Příkladem takového algoritmu je Cooperative A* (CA*) [6]. Tento a další algoritmy od Davida Silvera budou podrobně představeny později.

2.2.5.3 Řešení pomocí redukce

Příkladem problému, na který je možné MAPF redukovat, je SAT problém. V takovém případě je MAPF problém zapsán jako formule, u které lze zjistit, zda je pravdivá nebo nepravdivá. Formule je utvořena pro konkrétní hodnotu nákladové funkce. Optimální řešení je pak možné nalézt tak, že je formule opakovaně vytvářena pro čím dál tím větší hodnoty nákladové funkce a je zkoušena, zda je pravdivá. Ve chvíli kdy je zjištěno, že formule je pravdivá, tak přečtením pravdivých proměnných je možné zjistit řešení MAPF problému. [8]

2.3 Heuristika

Heuristika dává přibližné řešení, které je založené na odhadu. Heuristika nezaručuje nalezení nejlepšího řešení, ale na druhou stranu nějaké řešení dává rychle. [12]

2.3.1 Manhattanská vzdálenost

V této práci se bude často využívat jako heuristika pro prohledávací algoritmy Manhattanská vzdálenost (někdy také Manhattanská metrika). Ta je definovaná jako:

$$h(\vec{x}, \vec{y}) = \sum_{i=1}^{n} |x_i - y_i|,$$

kde x_i a y_i je i-tá část vektoru $\vec{x}$, respektive $\vec{y}$, a n je dimenze prostoru, na kterém je prohledávání prováděno. [13]

Vzhledem k problému řešenému v této práci bude využívána modifikovaná verze, která funguje pro toroidní mapy. (Toroidní mapa je taková, ve které lze projít z levého nebo horního kraje do pravého, respektive dolního kraje mapy a naopak.) Výpočet této varianty Manhattanské vzdálenosti je ukázán v pseudokódu 2.1. Rozdíl je v části $|x_i - y_i|$, kde v případě, kdy hodnota $|x_i - y_i|$

je větší než polovina mapy, je tato hodnota nahrazena $mapSize_i - |x_i - y_i|$, kde $mapSize_i$ je velikost mapy v i-tém rozměru. [14]

```
1  Procedure Heurestics(x⃗, y⃗):
2  |  distance ← 0
3  |  for i ← 1 to n do
4  |  |  distance_i ← |x_i − y_i|
5  |  |  if distance_i > mapSize_i/2 then
6  |  |  |  distance_i ← mapSize_i − distance_i
7  |  |  end
8  |  |  distance ← distance + distance_i
9  |  end
10 return distance
```

Algoritmus 2.1: Manhattanská vzdálenost pro n-dimenzionální toroidní mapy

2.4 A*

A* (A star) je algoritmus pro hledání cest v grafech. V algoritmech pro řešení MAPF problémů je A* využíván jako spodní vrstva, v takovém případě je prohledávání provedeno na stejném grafu $G(V, E)$, jako je ten zadaný v MAPF problému. Případně může být použita místo algoritmu A* nějaká jeho modifikovaná varianta. Cílem algoritmu je nalézt nejkratší cestu ze startovního vrcholu *start* do cílového vrcholu *goal*. Algoritmus A* nalezne nejkratší/optimální cestu, pokud je ale takových cest více, vrácená cesta závisí na konkrétní implementaci. Algoritmus 2.2 obsahuje pseudokód pro A*. [15]

Algoritmus využívá pro nalezení optimální cesty f-skóre, které lze získat jako $f(x) = g(x) + h(x)$, kde x je libovolným vrcholem zadaného grafu, $g(x)$ nebo také g-skóre vrcholu x je přesná vzdálenost ze startovního vrcholu do vrcholu x a $h(x)$ nebo také h-skóre vrcholu x je délka cesty z vrcholu x do cílového vrcholu odhadnutá pomocí heuristiky. Aby A* nalezl nejkratší cestu, tak musí být heuristika přípustná, tzn. nikdy nesmí dát hodnotu větší než je skutečná vzdálenost mezi konkrétními vrcholy. Pokud navíc heuristika splňuje podmínku $h(x) \leq d(x, y) + h(y)$ pro každou hranu $(x, y) \in E$, kde $d(x, y)$ udává vzdálenost mezi dvěma vrcholy, tak potom je heuristika h monotónní (někdy označovaná jako konzistentní). V tomto případě algoritmus expanduje každý vrchol nejvýše jednou (expandovat znamená, že je v tomto průchodu cyklu vrchol vyndán z *openSet*, ten je označován jako *current* a jsou navštíveni jeho sousední vrcholy). Hodnoty f-skóre jsou využívané pro rozhodnutí, v jakém pořadí mají být vrcholy v *openSet* expandovány. [15]

Hledání cesty probíhá tak, že je udržovaný soubor otevřených vrcholů *openSet*, tj. soubor s vrcholy, které byly navštíveny. Z těchto vrcholů je

vyndán vždy ten vrchol s nejmenším f-skóre, v pseudokódu označován jako *current*. Tento vrchol je od tohoto momentu označován jako expandovaný. Jeho sousední vrcholy jsou označované jako *neighbor*. Pokud splňují podmínku $g(current) + d(current, neighbor) < g(neighbor)$ na řádce 15, tak mají aktualizováno g-skóre na řádce 16 i vrchol, ze kterého byly navštíveni na řádce 17. Pokud navíc nejsou v *openSet*, tak tam jsou přidány, řádky 18–20. Algoritmus skončí, pokud je z *openSet* vybrán cílový vrchol *goal* nebo pokud je *openSet* prázdný. V prvním případě je vrácena rekonstruovaná cesta, v druhém případě je vrácena informace, že cesta neexistuje. [15]

2.5 Generalized Adaptive A*

Generalized Adaptive A* (GAA*) je algoritmus pro hledání cest v grafech. Na rozdíl od algoritmu A* je GAA* vytvořen tak, aby efektivně řešil série podobných problémů, kde se mění pozice agenta, pozice cíle a vzdálenost mezi vrcholy. Algoritmus hledá cesty rychleji než A*, protože si mezi jednotlivými hledáními cest ukládá aktualizované h-skóre (aktualizovanou heuristiku). Generalized Adaptive A* je rozšířením algoritmu Adaptive A*, který byl prvně popsán v [16] a později vylepšen možností odloženého vyhodnocování [17]. [18]

Dle [16] je aktualizovaná heuristika získána pro každý expandovaný vrchol v jako:

$$h(v) \leftarrow g(goal) - g(v), \tag{2.1}$$

kde $h(v)$ je uložená heuristika vzdálenosti do cílového vrcholu z vrcholu v, $g(goal)$ je vzdálenost ze startovního vrcholu do cílového vrcholu a $g(v)$ je vzdálenost ze startovního vrcholu do vrcholu v.

Pokud heuristika byla monotónní, tak i aktualizovaná heuristika bude monotónní. Navíc s využitím takto získané heuristiky není nikdy navštíveno více vrcholů než by bylo s A*, který využívá pouze základní heuristiku (tj. heuristika vypočítaná konkrétním algoritmem pro výpočet heuristik, například Manhattanská vzdálenost). [17, 19]

Tvrzení 1 *Heuristiky pro jednotlivé vrcholy nikdy neklesají při aplikování přiřazení 2.1.* [19]

Důkaz: Předpokládejme, že je heuristika vrcholu v aktualizována podle přiřazení 2.1. Pak byl vrchol v expandován a hodnota jeho f-skóre splňuje $f(v) \leq g(goal)$ $(f(v) = g(v) + h(v))$. Nyní odečtení $g(v)$ od obou stran nerovnice utvoří $h(v) = f(v) - g(v) \leq g(goal) - g(v)$, a tedy aktualizace nemůže snížit heuristiku vrcholu v, protože mění heuristiku z $h(v)$ na $g(goal) - g(v)$ (podle přiřazení 2.1). ■ [19]

Generalized Adaptive A* řeší série podobných problémů, kde je důležité, aby uložené heuristiky zůstaly monotónní. Dle [18] následné změny mohou nastat mezi jednotlivými hledáními cest:

```
 1 Procedure Path(start, goal):
 2     foreach vertex v ∈ V do
 3         g(v) ← ∞
 4         cameFrom(v) ← none
 5     end
 6     g(start) ← 0
 7     openSet ← {start}
 8     while openSet is not empty do
 9         current ← the vertex from openSet with the lowest f-value =
            g-value + h-value
10         Delete the vertex from openSet with the lowest f-value
11         if current = goal then
12             return ReconstructPath(cameFrom, current)
13         end
14         foreach neighbor of current do
15             if g(current) + d(current, neighbor) < g(neighbor) then
16                 g(neighbor) ← g(current) + d(current, neighbor)
17                 cameFrom(neighbor) ← current
18                 if neighbor ∉ openSet then
19                     Add neighbor to openSet
20                 end
21             end
22         end
23     end
24 return no path exists
25 Procedure ReconstructPath(cameFrom, current):
26     path ← {}
27     while cameFrom(current) ≠ none do
28         Append current to path
29         current ← cameFrom(current)
30     end
31     Reverse path order
32 return path
```

Algoritmus 2.2: A*

- Změní se startovní vrchol. V tomto případě není nutné nijak upravovat uložené heuristiky.
- Změní se cílový vrchol. V tomto případě je nutné aktualizovat uložené heuristiky. Vezměme *goal* jako původní cílový vrchol a *newGoal* jako nový cílový vrchol. Pak je h-skóre každého vrcholu v aktualizované jako:

  $$h(v) \leftarrow \max(H(v, newGoal), h(v) - h(newGoal)), \quad (2.2)$$

  kde $H(v, newGoal)$ značí odhad vzdálenosti mezi vrcholy v a *newGoal* spočítané základní heuristikou. [17]
- Změní se vzdálenost mezi některými vrcholy. Pokud se vzdálenost zvýší (například se vrchol stal nepřístupným, což bude signalizováno, že vzdálenost je nekonečno), tak GAA* nemusí nic udělat, protože h-skóre zůstane monotónní [18]. Pokud se vzdálenost zmenší (například se vrchol zpřístupní, vzdálenost se zmenší z nekonečna), tak je potřeba provést opravu [18] (ta bude zmíněna v části s pseudokódem).

2.5.1 Odložené vyhodnocování

Generalized Adaptive A* využívá odloženého vyhodnocování „lazy evaluation" k aktualizaci h-skóre. To znamená, že algoritmus stráví čas aktualizací h-skóre jen, pokud je to nutné. GAA* si kvůli tomu pamatuje některé informace z původních prohledávání. Například g-skóre, na řádcích 84 a 65, a některé informace o dokončeném prohledávání, jako je g-skóre cílového vrcholu, řádky 70 a 73. [17]

2.5.2 Pseudokód

Algoritmus 2.3 obsahuje pseudokód pro GAA*. V [17] Generalized Adaptive A* neaktualizuje g-skóre a h-skóre okamžitě, ale využívá k tomu proměnné *counter*, $search(v)$ a $pathcost(x)$:

- Hodnotou proměnné *counter* je x během x-tého prohledávání.
- Hodnotou $search(v)$ je x, pokud byl vrchol v navštíven v x-tém prohledávání. Tyto hodnoty jsou inicializovány nulami, řádky 4–6.
- Hodnotou $pathcost(x)$ je délka nejkratší cesty ze startovního vrcholu do cílového vrcholu, nalezené v x-tém prohledávání.

Pro inicializaci se použije `Initialize`, kde se nastaví počáteční hodnoty proměnných. Procedura `Path` je volána pro provedení dalšího prohledávání a procedura `ReconstructPath` sestaví soubor vrcholů vedoucí ze startovního vrcholu do cílového, řádek 74. (Pseudokód je stejný jako u A*, algoritmus 2.2.)

Generalized Adaptive A* použije proceduru `ComputePath`, aby prohledal graf (podobně jako A*), řádek 68. (Minimum z prázdného *openSet* je nekonečno, řádek 78) Procedura `InitializeVertex` se spustí, když je g-skóre nebo h-skóre potřeba, řádky 53, 63, 64 a 82. [18]

Dle [17, 18] následující body popisují, jak a za jakých podmínek jsou g-skóre a h-skóre nastavovány:

- GAA* inicializuje g-skóre vrcholu v na nekonečno, buď pokud vrchol v nebyl ještě ani jednou navštíven ($search(v) = 0$), řádek 44, nebo nebyl navštíven v aktuálním běhu ($search(v) \neq counter$), ale někdy dříve již navštíven byl ($search(v) \neq 0$), řádek 42.

- GAA* inicializuje h-skóre vrcholu základní heuristikou, pokud vrchol ještě nikdy nebyl navštíven ($search(v) = 0$), řádek 45.

- GAA* aktualizuje h-skóre vrcholu v podle přiřazení 2.1 na řádce 38, pokud jsou všechny následující podmínky splněny:

  1. Vrchol v ještě nebyl v tomto prohledávání navštíven (inicializován) ($search(v) \neq counter$).
  2. Vrchol v byl alespoň jednou navštíven (inicializován) ($search(v) \neq 0$).
  3. Vrchol v byl expandován prohledáváním, které ho naposledy navštívilo (inicializovalo) ($g(v) + h(v) < pathcost(search(v))$ – jedná se o stejnou podmínku, jako je na řádce 78 použita k řízení prohledávání [hodnota $g(v)$ nebyla změněna od posledního navštívení]).

- GAA* opraví (aktualizuje) h-skóre vrcholu v podle přiřazení 2.2 pro nový cílový vrchol. Tato aktualizace sníží h-skóre všech vrcholů o h-skóre nového cílového vrcholu. Toto h-skóre nového cílového vrcholu je přidané do souboru průběžných součtů oprav, na řádce 55. Konkrétně je hodnota $deltah(x)$ součet všech provedených oprav od prvního do x-tého prohledávání. Pokud byl vrchol v alespoň jednou navštíven ($search(v) \neq 0$), ale ještě nebyl navštíven aktuálním prohledáváním ($search(v) \neq counter$), pak GAA* aktualizuje jeho h-skóre součtem všech oprav (aktualizací) mezi prohledáváním, kdy byl vrchol v expandován, a aktuálním prohledáváním. Tento součet oprav má stejnou hodnotu, jako je rozdíl mezi hodnotou *deltah* během aktuálního prohledávání ($deltah(counter)$) a hodnotou *deltah* během prohledávání, které vrchol naposledy navštívilo ($deltah(search(v))$), řádek 40. Nakonec se vybere maximum z takto získané hodnoty h-skóre a základní heuristiky mezi vrcholem v a cílovým vrcholem, řádek 41.

```
1  Procedure Initialize():
2  |  counter ← 1
3  |  deltah(1) ← 0
4  |  foreach vertex v ∈ V do
5  |  |  search(v) ← 0
6  |  end
7  |  firstSearch = true
8  |  start ← vertex where start is now
9  |  goal ← vertex where goal is now
10 return
11 Procedure FixHeuristics():
12 |  openSet ← ∅
13 |  foreach vertex-action pair (vertex, action) whose action cost
   |    decreased do
14 |  |  InitializeVertex(vertex)
15 |  |  InitializeVertex(action(vertex))
16 |  |  if h(vertex) > d(vertex, action(vertex)) + h(action(vertex))
   |  |    then
17 |  |  |  h(vertex) ← d(vertex, action(vertex)) + h(action(vertex))
18 |  |  |  if vertex ∈ openSet then  Delete vertex from openSet
19 |  |  |  Add vertex to openSet with h-value h(vertex)
20 |  |  end
21 |  end
22 |  while openSet is not empty do
23 |  |  current ← the vertex from openSet with the lowest h-value
24 |  |  Delete the vertex from openSet with the lowest h-value
25 |  |  foreach neighbor of current do
26 |  |  |  InitializeVertex(neighbor)
27 |  |  |  if h(current) > d(current, neighbor) + h(neighbor) then
28 |  |  |  |  h(current) ← d(current, neighbor) + h(neighbor)
29 |  |  |  |  if current ∈ openSet then  Delete current from
   |  |  |  |    openSet
30 |  |  |  |  Add current to openSet with h-value h(current)
31 |  |  |  end
32 |  |  end
33 |  end
34 return
```

```
35 Procedure InitializeVertex(v):
36 |  if search(v) ≠ counter AND search(v) ≠ 0 then
37 |  |  if g(v) + h(v) < pathcost(search(v)) then
38 |  |  |  h(v) ← pathcost(search(v)) − g(v)
39 |  |  end
40 |  |  h(v) ← h(v) − (deltah(counter) − deltah(search(v)))
41 |  |  h(v) ← max(h(v), H(v, goal))
42 |  |  g(v) ← ∞
43 |  else if search(v) = 0 then
44 |  |  g(v) ← ∞
45 |  |  h(v) ← H(v, goal)
46 |  end
47 |  search(v) ← counter
48 return
49 Procedure Path():
50 |  if firstSearch = false then
51 |  |  newGoal ← vertex where goal is now
52 |  |  if goal ≠ newGoal then
53 |  |  |  InitializeVertex(newGoal)
54 |  |  |  if g(newGoal) + h(newGoal) < pathCost(counter) then
   |  |  |    h(newGoal) ← pathCost(counter) − h(newGoal)
55 |  |  |  deltah(counter + 1) ← deltah(counter) + h(newGoal)
56 |  |  |  goal ← newGoal
57 |  |  else  deltah(counter + 1) ← deltah(counter)
58 |  |  start ← vertex where start is now
59 |  |  counter ← counter + 1
60 |  |  FixHeuristics()
61 |  end
62 |  firstSearch ← false
63 |  InitializeVertex(start)
64 |  InitializeVertex(goal)
65 |  g(start) ← 0
66 |  cameFrom(start) ← none
67 |  openSet ← {start}
68 |  ComputePath()
69 |  if openSet is empty then
70 |  |  pathCost(counter) ← ∞
71 |  |  return no path exists
72 |  else
73 |  |  pathCost(counter) ← g(goal)
74 |  |  return ReconstructPath(goal)
75 |  end
76 return
```

```
77  Procedure ComputePath():
78      while g(goal) > min_{x∈openSet}(g(x) + h(x)) do
79          current ← the vertex from openSet with the lowest f-value =
              g-value + h-value
80          Delete the vertex from openSet with the lowest f-value
81          foreach neighbor of current do
82              InitializeVertex(neighbor)
83              if g(neighbor) > g(current) + d(neighbor, current) then
84                  g(neighbor) ← g(current) + d(neighbor, current)
85                  cameFrom(neighbor) ← current
86                  if neighbor ∈ openSet then
87                      Delete neighbor from openSet
88                  end
89                  Add neighbor to openSet with f-value
                      g(neighbor) + h(neighbor)
90              end
91          end
92      end
93  return
94  Procedure ReconstructPath(goal):
95      path ← {}
96      current ← goal
97      while cameFrom(current) ≠ none do
98          Append current to path
99          current ← cameFrom(current)
100     end
101     Reverse path order
102 return path
```

Algoritmus 2.3: Generalized Adaptive A*

Pokud se mezi jednotlivými prohledáváními sníží vzdálenost mezi vrcholy (změní se cena provedení akce pro přechod mezi nimi), tak je nutné opravit uložené h-skóre (heuristiku) vrcholů tak, aby zůstalo monotónní. Toho je docíleno voláním procedury `FixHeuristics` předtím než začne samotné prohledávání. Procedura funguje na principu Dijkstrova algoritmu. (Jinak řečeno A*, ale bez využití heuristiky.) Navíc tato procedura skončí až ve chvíli, kdy aktualizuje h-skóre všech vrcholů, které by jinak porušovaly monotónnost. [18]

2.6 Cooperative A*

Cooperative A* (zkratka CA*) je jedním z algoritmů pro suboptimální řešení MAPF problémů. V této práci bude CA* považován za horní vrstvu algoritmu

řešícího MAPF problém. Úloha je řešena jako série hledání cest jedním agentem. Tato prohledávání jsou prováděna v prostoru rozšířeném o dimenzi času a berou v potaz cesty ostatních agentů. Algoritmus 2.4 obsahuje pseudokód pro CA*. [6]

Bezkoliznost naplánovaných cest je zaručena použitím rezervační tabulky, do které každý agent zaznamená jednotlivé pozice své cesty s ohledem na čas, ve kterém se tam nachází. Tyto zaznamenané pozice jsou pro ostatní agenty, kteří hledají cestu, považovány za neprůchodné. Tento přístup je velice náchylný na správné pořadí agentů při hledání cest (někdy nazývané jako priorita agentů). Vzhledem k řídkosti záznamů v rezervační tabulce je vhodné ji implementovat jako hašovací tabulku. [6]

```
1 Procedure Paths():
2     resevationTable ← ∅
3     for all agents do
4         Find path with Low-level solver using procedure Path
5         Add path to the reservationTable
6     end
7 return
```

Algoritmus 2.4: Cooperative A*

2.6.1 Hierarchical Cooperative A*

Hierarchical Cooperative A* (HCA*) je upravená varianta CA*, která jako heuristiku nepoužívá Manhattanskou vzdálenost, ale využívá hodnot vrácené algoritmem Reverse Resumable A* (RRA*). Ten pro jednotlivé vrcholy vrací vzdálenost do cíle, která ignoruje všechny ostatní agenty, ale počítá se všemi zdmi. [6]

RRA* je upravená verze algoritmu A*, úpravy jsou:

- RRA* hledá cestu z cílového vrcholu (cíl agenta, pro kterého HCA* právě hledá cestu) do startovního vrcholu (vrchol, ve kterém agent aktuálně stojí). To způsobí, že veškerá g-skóre již expandovaných vrcholů obsahují požadovanou heuristiku pro algoritmus HCA*. [6]
- V algoritmu RAA* lze obnovit prohledávání kdykoli, když je potřeba nová heuristika pro HCA*. [6]

Algoritmus 2.5 obsahuje pseudokód pro Reverse Resumable A*. Procedurou `Initialize(start, goal)` by měla být nalezena heuristika pro všechny vrcholy, které budou potřeba pro nalezení cesty, pokud cesta mezi startovním vrcholem a cílovým vrcholem není omezená žádnými záznamy v rezervační tabulce. Nalezení potřebných vrcholů je docíleno tak, že se prochází sousední

vrcholy v opačném pořadí, než jsou procházeny ve spodní vrstvě algoritmu pro řešení MAPF problémů, řádek 13. [6]

Procedura `AbstractDist(toFind)` je využívána spodní vrstvou algoritmu pro řešení MAPF problémů jako heuristika. Pokud vrchol $toFind$, pro který je vyžadována heuristika, již byl expandován, tak procedura vrátí přímo hodnotu $g(toFint)$. Pokud ale vrchol ještě expandován nebyl (byl pouze navštíven nebo nebyl ani to), tak se zavolá procedura `Resume(toFind)`, která bude pokračovat v prohledávání grafu. Prohledávání ale bude nadále cílit do startovního vrcholu agenta (h-skóre používané na řádcích 11 a 12 je stále mezi cílovým *goal* a startovním *start* vrcholem agenta), ale skončí ve chvíli, kdy expanduje hledaný vrchol $toFind$. To zajistí, že všechny hodnoty g-skóre stále obsahují požadované hodnoty pro HCA*. [6]

2.6.2 Windowed Hierarchical Cooperative A*

Windowed Hierarchical Cooperative A* (WHCA*) je upravená varianta algoritmu HCA*, která pro zlepšení kvality řešení a rychlosti, v níž je řešení nalezeno, zavádí několik funkcionalit:

1. WHCA* zavádí tzv. okno velikosti w. Prohledávání horní vrstvou algoritmu pro řešení MAPF problémů limituje délky cest uložené v rezervační tabulce na velikost okna w. To znamená, že spodní vrstva po w krocích ignoruje ostatní agenty při hledání cesty. Další vlastností okna je, že každý agent musí mít cestu alespoň délky w, tedy formálně plán jednoho agenta i musí splňovat, že $|\pi_i| \geq w$. To zajistí, že agenti se vyhýbají ostatním agentům i potom, co dorazí do cíle. [6]

2. Další změnou ve WHCA* je zavedení speciálních hran, které vedou ze všech vrcholů ve vzdálenosti w rovnou do cílového vrcholu. Navíc prostor, který je prohledáván, je limitován pouze na takovou část grafu, kde všechny vrcholy jsou ve vzdálenosti w nebo blíže. Z těch, které jsou ve vzdálenosti w, pak vedou tyto speciální hrany do cílového vrcholu. Speciální hrany mají délku stejnou jako je vzdálenost do cílového vrcholu odhadnutá heuristikou. Vzhledem k tomu, že je jako heuristika využívána vzdálenost ignorující ostatní agenty, tak je tento přístup stejný, jako kdyby byla dohledána celá cesta. [6]

3. Poslední změnou ve WHCA* je nutnost cesty hledat znovu. (Cesty omezené na velikost okna je vhodné následovat pouze po určitou dobu.) Cesty jsou pro agenty hledané vždy, když urazí polovinu velikosti okna. Pokud je navíc hledání cest rozloženo do různých časových kroků, tak se agenti střídají v tom, kdo dokáže zarezervovat cestu nejdále, a tedy je vyřešen problém s pořadím, ve kterém cesty hledají. Rozložení hledání cest lze využít také pro rozdělení výpočetní zátěže mezi více časových kroků. [6]

```
1  Procedure Initialize(start, goal):
2      foreach vertex v ∈ V do
3          g(v) ← ∞
4      end
5      g(goal) ← 0
6      openSet ← {goal}
7      Resume(start)
8  return
9  Procedure Resume(toFind):
10     while openSet is not empty do
11         current ← the vertex from openSet with the lowest f-value =
            g-value + h-value
12         Delete the vertex from openSet with the lowest f-value
13         foreach neighbor of current in reverse order do
14             if g(current) + d(current, neighbor) < g(neighbor) then
15                 g(neighbor) ← g(current) + d(current, neighbor)
16                 cameFrom(neighbor) ← current
17                 if neighbor ∉ openSet then
18                     Add neighbor to openSet
19                 end
20             end
21         end
22         if current = toFind then
23             return success
24         end
25     end
26 return failure
27 Procedure AbstractDist(toFind):
28     if toFind was already expanded then
29         return g(toFind)
30     end
31     if Resume(toFind) = success then
32         return g(toFind)
33     end
34 return ∞
```

Algoritmus 2.5: Reverse Resumable A*

2.7 Měření rychlosti algoritmu

Existují různé přístupy, jak měřit rychlost algoritmů řešících MAPF problémy. Hlavní metrikou, podle které lze algoritmy porovnávat, je čas, který algoritmus potřebuje k vyřešení zadaného problému. Existuje několik způsobů, jak algoritmy porovnat pomocí měření času, příkladem může být: [7, 18]

1. Porovnat, jak dlouho trvá nalezení řešení pro jeden problém nebo množinu problémů. [18]
2. Porovnat, kolik ze zadané množiny problémů, která je řešena vždy ve stejném pořadí, je během zadaného časového okna algoritmus schopen vyřešit. [7]

Pokud jsou používané algoritmy založené na prohledávání, lze jako další metriku použít množství expandovaných nebo navštívených vrcholů spodní vrstvou algoritmu. [7, 18]

Kapitola 3

Analýza

Jako první bude v této kapitole upřesněn problém řešený v této práci. Následně bude specifikován způsob porovnání algoritmů, které tento problém řeší, a bude představena nová nákladová funkce. Nakonec bude vysvětleno, proč byly vybrány konkrétně ty algoritmy, které byly popsané v předchozí kapitole.

3.1 Specifikace řešeného problému

V této práci je řešen problém vycházející ze hry Pac-man, a to takový, kde je více Pac-manů a kde nemůže několik Pac-manů stát na jednom poli současně a nemůžou skrz sebe procházet (jedná se o obě omezující podmínky uvedené v sekci definující MAPF problém). Vzhledem k tomu, že v klasické hře Pac-man se každý duch specifickým způsobem zaměřuje na jediného Pac-mana ovládaného hráčem, tak to pro variantu s více Pac-many znamená, že se bude každý duch muset zaměřovat pouze na jednoho z Pac-manů. Tento problém bude vyřešen tak, že každý Pac-man bude mít po celou dobu přiřazeného jednoho "svého"ducha.

Tato práce se zaměřuje pouze na část hry, kde jsou duchové vystrašení (jsou ve fázi Frightened) a Pac-mani je mohou chytat. To znamená, že na mapě jsou rozmístěni Pac-mani, kteří se snaží chytit své duchy. Předpokládá se, že každý Pac-man může chytit pouze svého ducha. Tedy skrz ostatní duchy projde bez toho, aby se něco stalo. Rychlost Pac-mana je dvojnásobná vůči rychlosti duchů. To znamená, že se Pac-mani vždy pohnou dvakrát, než se pohnou duchové. Po dokončení svého úkolu jednotlivý Pac-man čeká na to, až ostatní Pac-mani chytí své duchy.

Popsaný problém bude řešen jako multi-agentní hledání cest, kde grafem je mapa hry. Konkrétně pole mapy se stanou vrcholy grafu a hrany budou mezi vrcholy, jejichž pole spolu sousedí ortogonálně (nikoli jen rohem).

3.1.1 Počáteční rozmístění Pac-manů

Vzhledem k tomu, že v klasické hře Pac-man jsou duchové vystrašeni pouze po sebrání velké kuličky, které se nacházejí v různých částech mapy, tak v problému řešeném v této práci se bude počítat s tím, že Pac-mani a duchové v době sebrání velké kuličky mohli být kdekoli kromě jednoho Pac-mana, který byl na jednom z míst s velkou kuličkou. Tato skutečnost ale bude zanedbána, protože můžou být použity nové mapy, které kuličky rozmístěné nemají. Navíc tato skutečnost není pro řešený problém podstatná. Pozice Pac-manů a duchů budou tedy náhodné s výjimkou, že Pac-man nikdy nebude začínat na stejném místě, jako je jeho duch. To by totiž Pac-man byl duchem chycen před zahájením fáze Frightened, navíc by vzhledem k problému řešeném v této práci byl duch chycen okamžitě, což pro hledání cest není zajímavý výsledek.

3.2 Porovnávání kvality algoritmů

Pro porovnávání kvality řešení nalezeného algoritmy budou použity nákladové funkce dříve popsané v kapitole s teoretickými východisky. Nákladová funkce makespan znázorňuje, jak dlouho trvalo, než byl poslední duch chycen. Nákladová funkce sum-of-costs nemůže být využita v původním znění, protože více Pac-manů může potřebovat chytit ducha na stejném poli a tedy někteří Pac-mani nemohou na svém cílovém vrcholu zůstat.

Pro tuto práci je zavedena nová nákladová funkce, která je modifikací sum-of-costs. Rozdíl je pouze ten, že Pac-mani nemusí na cílovém vrcholu zůstávat. Jedná se tedy o součet délek cest, než je cílový vrchol poprvé dosažen. Tato nákladová funkce bude pro odlišení nazývána kumulativní přežití duchů, protože sečtení délky životů všech duchů je stejné jako součet délek cest agentů, než jsou duchové chyceni.

Kromě srovnávání pomocí kvality řešení můžeme algoritmy řešící MAPF problémy porovnávat také podle času, jak dlouho jim trvá nalezení řešení. Měření rychlosti algoritmů budou probíhat podobně, jako je popsáno v poslední sekci kapitoly s teoretickými východisky.

3.3 Zvolené algoritmy

V kapitole s teoretickými východisky byly popsány pouze algoritmy, které budou v této práci nějakým způsobem využity. Algoritmy od Davida Silvera, jmenovitě CA*, HCA* a WHCA*, řeší problém multi-agentního hledání cest. Tyto algoritmy hledají suboptimální řešení a byly vytvořeny s myšlenkou, že budou využívány i v počítačových hrách, protože tam není většinou třeba optimálních cest a spíše záleží na rychlosti algoritmu. Vzhledem k tomu, že problém řešený v této práci vychází z počítačové hry, tak tyto algoritmy jsou vhodné. Navíc se cíle pohybují a je často nutné cesty hledat znovu, takže

suboptimalita nalezených cest bude mít menší vliv na kvalitu řešení, protože i optimální cesty by byly často měněny. Další výhodou těchto algoritmů je, že mohou být použity jako pouhá horní vrstva, protože samotné hledání cest provádí v těchto algoritmech A*. A tedy může být nahrazen upraveným algoritmem Generalized Adaptive A*, který efektivně hledá cestu pro jednoho agenta v sérii podobných problémů.

Jako heuristika pro A* a Generalized Adaptive A* bude použit algoritmus 2.1 představený v subsekci „Manhattanská vzdálenost“ v kapitole s teoretickými východisky, který odhaduje vzdálenost mezi dvěma vrcholy v mapách, kde lze procházet kraji mapy.

Kapitola 4

Návrh algoritmů

V této kapitole bude jako první popsána modifikovaná varianta algoritmu Generalized Adaptive A*, která bude využita v některých nových algoritmech. Následně budou popsány společné úpravy pro algoritmy Cooperative A*, Hierarchical Cooperative A* a všechny nově představené algoritmy. Poté budou postupně po dvojicích představeny nové algoritmy. Konkrétně se jedná o algoritmy Cooperative Generalized Adaptive A* (CGAA*), Hierarchical Cooperative Generalized Adaptive A*, Hierarchical+ Cooperative A* (H+CA*) a Hierarchical+ Cooperative Generalized Adaptive A* (H+CGAA*).

4.1 Modifikace Generalized Adaptive A*

Tato část práce se zaměřuje na změny provedené na algoritmu GAA* s ohledem na problém řešený v této práci. Vzhledem k tomu, že algoritmus bude využit jako spodní vrstva v některých nových algoritmech založených na CA* a HCA*, kde je hledání prováděno na prostoru rozšířeném o dimenzi času, tak je nutné udělat některé úpravy. Mezi úpravy patří takové, které umožňují mít více cílových vrcholů. Tyto cílové vrcholy budou vrcholy na souřadnicích cílového pole v různých časových krocích. Další úpravou je dodatečná specifikace času pro vrchol *neighbor*.

Provedené úpravy se odkazují na řádky algoritmu GAA* (2.3) a jsou rozepsány v následujících bodech:

- V proceduře `ComputePath` je cyklus na řádku 81. Jeho řídící podmínka je změněna na „*neighbor of current in next time step*“. Úprava specifikuje v jakém časovém kroku je *neighbor*.
- Vrcholy *goal* a *newGoal* označují pouze souřadnice cílového pole. Řádky 53–54 a 64 budou tedy prováděny pro každý cílový vrchol zvlášť.
- Procedura `ComputePath` je upravena tak, aby řídicí podmínka cyklu nevyžadovala hodnotu týkající se cílového vrcholu. Samotná řídicí pod-

```
1  Procedure FixHeuristics(freedVertices):
2      openSet ← ∅
3      foreach freedV ∈ freedVertices do
4          InitializeVertex(freedV)
5          foreach neighbor of freedV in previous time step do
6              if neighbor's time < 0 then
7                  continue
8              end
9              InitializeVertex(neighbor)
10             if h(neighbor) > d(neighbor, freedV) + h(freedV) then
11                 h(neighbor) ← d(neighbor, freedV) + h(freedV)
12                 if neighbor ∈ openSet then  Delete neighbor from
                    openSet
13                 Add neighbor to openSet with h-value h(neighbor)
14             end
15         end
16     end
17     while openSet is not empty do
18         current ← the vertex from openSet with the lowest h-value
19         Delete the vertex from openSet with the lowest h-value
20         foreach neighbor of current in previous time step do
21             if neighbor's time < 0 then
22                 continue
23             end
24             InitializeVertex(neighbor)
25             if h(current) > d(current, neighbor) + h(neighbor) then
26                 h(current) ← d(current, neighbor) + h(neighbor)
27                 if current ∈ openSet then  Delete current from
                    openSet
28                 Add current to openSet with h-value h(current)
29             end
30         end
31     end
32 return
```

Algoritmus 4.1: Modifikovaná procedura algoritmu GAA*

mínka na řádku 78 je nahrazena „*openSet is not empty*“ a po vybrání vrcholu z *openSet* s nejnižším f-skóre na řádku 79 je vložena podmínka „**if** *current* = *goal* **then return**“, kde *goal* znamená libovolný cílový vrchol. Jedná se o podobný přístup jako v popsaném algoritmu A* (2.2).

Modifikovaná procedura skončí ve chvíli, kdy je expandován cílový vrchol (stejně jako v A*). Oproti tomu, původní varianta končí, když podmínka $g(goal) > \min_{x \in openSet}(g(x)+h(x))$ přestane platit. Z této podmínky lze poznat, že pokud v *openSet* je již cílový vrchol jako vrchol s nejnižším f-skóre (zde $g(x)+h(x)$), tak tento vrchol nebude expandován a procedura skončí. S touto změnou tedy není nikdy expandováno méně vrcholů, než by bylo v původní variantě procedury. To znamená, že podmínka v proceduře `InitializeVertex(v)` na řádce 37 může zůstat beze změny.

- Upravená verze pseudokódu pro proceduru `FixHeuristics` je v algoritmu 4.1. Této proceduře je nyní předáván parametr $freedVertices$. Tento parametr bude dodán horní vrstvou algoritmu řešící MAPF problémy a bude obsahovat všechny vrcholy prohledávaného grafu, které byly při posledním prohledávání rezervovány, ale teď jsou volné. Další úpravou této procedury je specifikace, které vrcholy *neighbor* jsou v čase o jedna nižší než čas vrcholu *current*, řádek 5 a 20. Poslední úpravou je přidání podmínky, která přeskočí konkrétní *neighbor*, pokud je v neexistujícím časovém kroku (časovém kroku menším než 0), řádky 6–8 a 21–23.

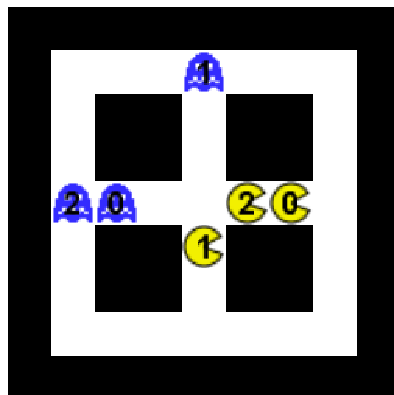
4.2 Modifikace CA* a HCA*

Horní vrstvy algoritmů řešící MAPF problémy budou v této práci mít určité společné funkcionality. Všechny testované algoritmy jsou založené na algoritmu Cooperative A* rozšířeném o některé funkcionality představené v algoritmu Windowed Hierarchical Cooperative A*. Veškerá rozšíření z WHCA* aplikovaná nebudou, protože je aplikovat přímo na CA* nelze nebo by bylo riziko výrazně horších výsledků. V poslední podsekci bude popsán pseudokód pro modifikované CA* a HCA*.

4.2.1 Okno rezervací

Jedná se o první funkcionalitu představenou u algoritmu WHCA*. Velikost okna je zadaná parametrem před zahájením prohledávání.

Důvodem použití okna jsou problémy, které původní CA* a HCA* má v případě, kdy agent dorazí do cíle a na této pozici zůstane. Pak totiž ostatní agenti s nižší prioritou hledání cest tuto pozici nemohou přejít, ale budou ji muset obejít, což vzhledem k problému řešeném v této práci značně prodlouží cestu, případně zcela znemožní cestu nalézt.



Obrázek 4.1: Příklad MAPF problému, kde musí být agentovi s číslem dva zvýšena priorita.

4.2.2 Priorita agentů v rezervační tabulce

Ve všech algoritmech vycházejících z CA* si agenti hledají cesty hladově (bez ohledu na ostatní agenty) a v konkrétním pořadí. To znamená, že agent může naplánováním své cesty znemožnit nalezení cesty jinému agentovi. Z tohoto důvodu je tento přístup náchylný na správné pořadí hledání cest. Existují i problémy, které tímto přístupem vůbec řešit nelze. Například na obrázku 4.2a je problém, kde pro vyřešení je v ideálním případě potřeba spolupráce obou agentů. Tento problém je v této práci řešitelný, protože se cíle v určitý okamžik pohnou, ale nalezené řešení nebude optimální.

Dalším problémem pro CA* je možnost, že rezervace agentů s vyšší prioritou znemožní agentovi s prioritou nižší provádění jakýchkoli akcí. Na obrázku 4.1 vidíme, že v případě, kdy jako první hledají cestu agenti nula a jedna, tak pro agenta s číslem dva neexistuje žádná sekvence akcí, která neporuší nějakou z dříve popsaných omezujících podmínek (pohyb na pozici jiného agenta nebo pohyb skrz jiného agenta). Pokud tento problém nastane, tak se zvýší priorita agentovi, který nemůže provést žádné akce. Pak se cesty hledají znovu s pozměněnými prioritami. Priorita je zvýšena různými způsoby podle toho, zda agent již dosáhl cíle (zda byl duch chycen). Pokud agent ještě svého cíle nedosáhl, tak pak se v pořadí hledání cest přesune na první pozici. Pokud ale agent již svého cíle dosáhl (duch je již chycen), tak se v pořadí hledání cest přesune před agenta, který cestu hledal před ním.

Zvyšování priority u agentů, kteří již cíle dosáhli, je pomalejší, protože není vhodné, aby zbytečně předběhli agenty, kteří ještě svého cíle nedosáhli. (Kdyby se těmto agentům zvýšila v těchto situacích priorita na maximum, tak by vytvořili neprostupné pole pro ostatní agenty.) Naopak u agentů, kteří svého cíle ještě nedosáhli, je vhodné, že jsou přesunuti na první pozici, protože stále potřebují dosáhnout cíle, a prohledávání se nezdržuje postupným testováním, zda už je priorita agenta dostatečná, aby nalezl cestu.

Počet těchto změn priorit musí být kvůli možnosti neexistence cest limitován. Existuje totiž možnost, že problém nemá řešení, nebo že algoritmus neumí nalézt žádné řešení problému, viz výše zmíněný problém na obrázku 4.2a. Pro tento limit se nabízí faktoriál počtu agentů, jelikož s ním lze vyjádřit počet všech možných pořadí agentů. Prakticky ale není vhodné ho použít, protože například pro 10 agentů (což je jedním z nižších množství agentů využívané v experimentálním testování) je faktoriál více než 3 miliony a tudíž, kdyby bylo nutné otestovat všechna pořadí agentů, tak by to bylo výpočetně příliš náročné. V této práci bude tento limit nastaven na desetinásobek počtu agentů, pokud ale je agentů méně než šest, tak je využit faktoriál. Tento limit bude označován $maxChanges$. Jeho výpočet je v algoritmu 4.2 na řádcích 3–7.

V případě, že algoritmus nebude schopen nalézt řešení zadaného MAPF problému, budou všichni Pac-mani čekat. Následně je nutné cesty zkusit nalézt vždy, když se duchové pohnou.

Priorita je také změněna v případě, že se agent dostal svého cíle, to znamená, že Pac-man chytil svého ducha a již nemá co dělat. V takovém případě se priorita tohoto agenta sníží na minimum, cestu bude hledat jako poslední, aby se případně on vyhýbal ostatním agentům, nikoli naopak.

4.2.3 Ukázka fungování algoritmů

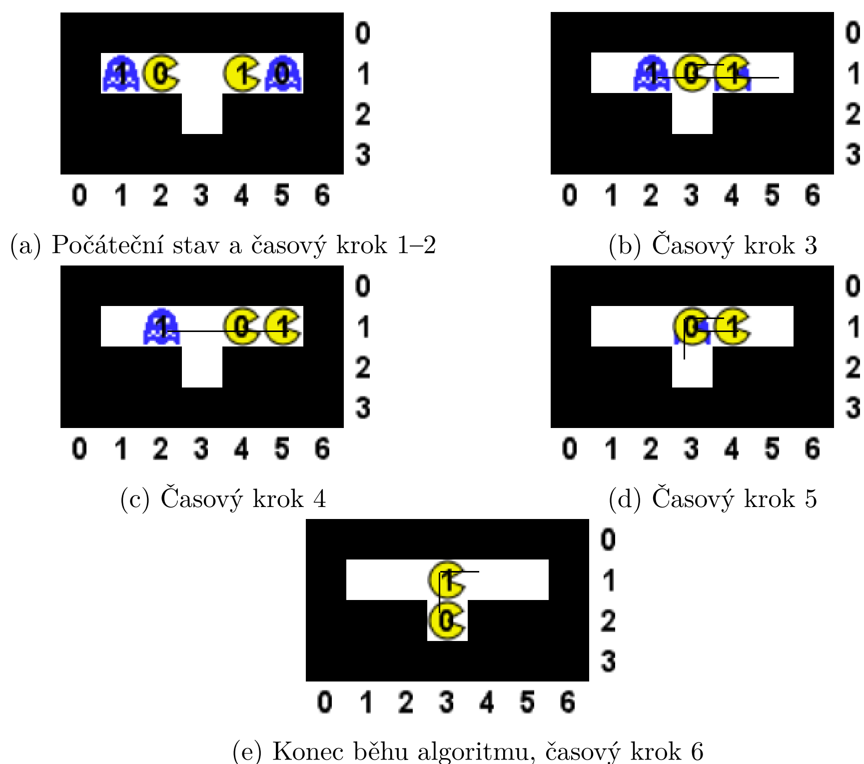
V této sekci bude ukázáno, jak CA* řeší problém na obrázku 4.2a. (Předpokládá se nastavení velikosti rezervační tabulky na čtyři a více – pokud by byla rezervační tabulka menší, řešení by vypadalo odlišně.)

Řešení je rozděleno po logických částech následovně:

1. Horní vrstva nalezne pomocí spodní vrstvy cestu pro Pac-mana nula. Ta vede přímo z pozice Pac-mana na pozici jeho ducha (tedy z (2, 1), přes (3, 1), (4, 1), a nakonec na (5, 1), kde čeká po zbytek velikosti rezervační tabulky). Následně se horní vrstva pokusí nalézt cestu i pro Pac-mana jedna. Jak již bylo dříve zmíněno, pro tohoto Pac-mana neexistuje žádná sekvence akcí stejné nebo větší délky, než je ta vyžadovaná rezervační tabulkou.

2. Vzhledem k tomu, že neexistuje cesta pro alespoň jednoho Pac-mana (v tomto případě Pac-mana jedna), tak se mu zvýší priorita a hledá se nyní cesta Pac-manovi jedna jako prvnímu. Tomu je stejně jako v bodě jedna pro Pac-mana nula nalezena cesta bez problému. V tuto chvíli, ale cestu nebude moci nalézt Pac-man nula, takže bude jeho priorita zvýšena.

3. Vhledem k tomu, že limit pokusů $maxChanges$ má pro dva agenty hodnotu dva (faktoriál dvou je dva, $2! = 2 \times 1 = 2$) a hledání cest již bylo vyzkoušeno dvakrát, tak se mezi tímto a následujícím časovým krokem



(a) Počáteční stav a časový krok 1–2 (b) Časový krok 3

(c) Časový krok 4 (d) Časový krok 5

(e) Konec běhu algoritmu, časový krok 6

Obrázek 4.2: Problém, který je neřešitelný algoritmy použitými v této práci do prvního pohybu duchů

Pac-mani nikam nepohnou (toto je vnímáno tak, že všichni Pac-mani provádí akci wait).

4. Nyní v dalším časovém kroku se Pac-mani mohou pohnout po druhé, ale duchové se ještě nikam nepřesunuli, tak Pac-mani znovu čekají. Mapa stále vypadá jako na obrázku 4.2a.

5. Teď se pohnou duchové. Duch nula se pohne z (5, 1) na (4, 1) a duch jedna z (1, 1) na (2, 1), protože to jsou pro ně jediné volné pozice. (Situace, kdy duchové mají pouze jediné pole, kam se mohou přesunout, v klasické mapě Pac-mana nikdy nenastane, ale zde budeme předpokládat, že by se v takovém případě přesunuli na tato pole.)

6. Algoritmus se znovu pokusí nalézt cesty. Vzhledem k tomu, že priorita byla zvýšena Pac-manovi nula a následně i jedna, tak nyní bude cesta hledána Pac-manovi nula. Jeho cesta bude začínat tam, kde aktuálně stojí a povede až na pole s jeho duchem (cesta vede přes pole (2, 1), (3, 1) a (4, 1) v tomto pořadí). Pro Pac-mana jedna již může být nalezena cesta, protože může Pac-manovi nula uhnout na pozici (5, 1). Jeho cesta bude vést z pozice (4, 1), kde se aktuálně nachází, na pole (5, 1), kde

bude čekat po dobu velikosti okna, a nakonec, když bude moci ignorovat rezervovaná pole Pac-mana nula, povede až na pole (2, 1), kde je jeho duch.

7. Oba Pac-mani mají cesty, takže je začnou následovat. Po pohybu Pac-manů mapa vypadá jako na obrázku 4.2b. Tenké černé čáry znázorňují cesty Pac-manů, začínají tedy na pozici Pac-mana a vedou až na pole s duchem.

8. Duchové se nepohnuli, nezměnila se priorita žádného Pac-mana, ani žádný duch nebyl chycen, takže není nutné hledat nové cesty. Pac-mani tedy následují své aktuální cesty. Mapa po dalším časovém kroku je znázorněna na obrázku 4.2c.

9. Vzhledem k tomu, že Pac-man nula chytil ducha, tak je jeho priorita snížena. To znamená, že jako prvnímu je cesta hledána Pac-manovi jedna. Nyní se pohnul zbylý duch na pole (3, 1). Pac-man jedna nalezne cestu přímo ze své aktuální pozice (5, 1) přes pole (4, 1) na (3, 1) s duchem. Pac-man nula musí stále hledat cestu na pole svého cíle (4, 1), přestože se v něm aktuálně nachází. Pac-man nula se musí vyhnout Pac-manovi jedna a jedinou možností je odejít ze svého cílového pole. To ale není problém, protože již ducha chytil, a nyní je pouze potřeba se jen vyhýbat. Jeho cesta bude začínat na poli (4, 1), kde se aktuálně nachází, poté bude muset jít na pole (3, 1) a pak využije pole (3, 2). Poté jeho cesta povede pozpátku zpět na pole (4, 1).

10. Poté co Pac-mani začnou následovat cesty, mapa se dostane do stavu znázorněném na obrázku 4.2d a pak do stavu na obrázku 4.2e. V tuto chvíli jsou všichni duchové chyceni a algoritmus končí vyřešením problému.

4.2.4 Pseudokód

Algoritmus 4.2 obsahuje pseudokód pro modifikovaný Cooperative A* a Hierarchical Cooperative A*. Procedura `Paths` je volaná vždy, když je vhodné vypočítat nové cesty, tj. před provedením prvního pohybu, po pohybu duchů nebo pokud agentovi klesla nebo vzrostla priorita.

Rozdíl mezi CA* a HCA* je v použité heuristice. V případě CA* je ve spodní vrstvě na řádku 12 používána Manhattanská vzdálenost pro toroidní mapy. U algoritmu HCA* je používána skutečná vzdálenost v grafu ignorující ostatní agenty. Ty jsou získány pomocí algoritmu RRA*. Ten je inicializován hned před hledáním cesty spodní vrstvou, tedy těsně před řádkem 12. Podrobnější fungování je popsáno v podsekci „Hierarchical Cooperative A*", která je v kapitole „Teoretická východiska".

Pokud se při hledání cest nepodaří nějakému agentovi žádnou cestu nalézt, tak je zaznamenáno, o kterého agenta se jedná, a hledání cest je zastaveno,

řádky 13–16. Následně je tomuto agentovi zvýšena priorita na řádce 20 a poté se zkouší znovu nalézt cesty pro všechny agenty.

```
1  Procedure Paths():
2      orederChanged ← 0
3      if number of agents > 5 then
4          maxChanges ← number of agents ∗10
5      else
6          maxChanges ← factorial of number of agents
7      end
8      while orederChanged < maxChanges do
9          resevationTable ← ∅
10         pacmanToBoost ← null
11         foreach agent a do
12             Find path for agent a with Low-level solver using
                procedure Path()
13             if No path was found for agent a then
14                 pacmanToBoost ← a
15                 break
16             end
17             Add agent a path to the reservationTable
18         end
19         if pacmanToBoost = null then return Success
20         Increase agent a priority
21         orederChanged ← orederChanged + 1
22     end
23 return Failure
```

Algoritmus 4.2: Modifikovaný CA* a HCA*

4.3 Algoritmy CGAA* a HCGAA*

Jedná se o první dva nové algoritmy, které vznikly spojením algoritmu Generalized Adaptive A* (GAA*) a algoritmů Cooperative A* (CA*) a Hierarchical Cooperative A* (HCA*). Celá jména jsou Cooperative Generalized Adaptive A* (CGAA*) a Hierarchical Cooperative Generalized Adaptive A* (HCGAA*). Oproti algoritmům CA* a HCA* používají nové algoritmy jako spodní vrstvu algoritmus GAA*. Potřebné modifikace GAA* byly popsány v sekci „Modifikace Generalized Adaptive A*“.

4.3.1 Pseudokód

Algoritmus 4.3 obsahuje pseudokód pro CGAA* a HCGAA*. Nové algoritmy mají vůči těm původním několik změn. Jednotlivé změny budou nyní rozepsány a budou ukázány v pseudokódu.

```
1  Procedure Paths():
2      orederChanged ← 0
3      if number of agents > 5 then
4          maxChanges ← number of agents ×10
5      else
6          maxChanges ← factorial of number of agents
7      end
8      while orederChanged < maxChanges do
9          resevationTable ← ∅
10         pacmanToBoost ← null
11         foreach agent a do
12             freedVertices ← every vertex which was reserved in
                 prevoius search but now is not reserved; oldPaths and
                 oldOrder is used
13             Find path with Low-level solver using
                 procedure Path(freedVertices)
14             if No path was found for agent a AND
                 pacmanToBoost = null then
15                 pacmanToBoost ← a
16             else
17                 Add agent a path to the reservationTable
18             end
19         end
20         oldPaths ← all current paths
21         oldOrder ← current order of agents
22         if pacmanToBoost = null then return Success
23         Increase agent a priority
24         orederChanged ← orederChanged + 1
25     end
26 return Failure
```

Algoritmus 4.3: Pseudokód CGAA* a HCGAA*

Hlavní změnou je nutnost mít pro každého agenta zvlášť uložené heuristiky a dodatečné informace k jejich úpravám. (Z implementačního hlediska může být pro každého agenta udržována samostatná instance třídy zařizující hledání cest.)

Na řádce 12 se přiřazuje do *freedVertices* soubor vrcholů, které byly pro spodní vrstvu při minulém prohledávání nepřístupné kvůli záznamu v re-

zervační tabulce, ale teď již přístupné jsou. Tento soubor vrcholů je využit algoritmem GAA* pro aktualizaci uložených heuristik. Podrobné využití bylo popsáno v sekci „Modifikace Generalized Adaptive A*“. Tyto vrcholy lze získat s využitím informací o předešlém prohledávání, tj. *oldPaths* a *oldOrder* uložené na řádcích 20 a 21. Z těchto informací lze zjistit, která pole byla při minulém hledání pro konkrétního agenta znepřístupněna rezervacemi. Tato pole lze otestovat, zda jsou i nadále zarezervována, a pokud nejsou, tak jsou přidána do souboru uvolněných vrcholů $freedVertices$.

V *oldPaths* a *oldOrder* jsou uložené pouze informace o posledním hledání. Kvůli tomu je nutné vždy opravit heuristiky pro všechny agenty a na rozdíl od CA* a HCA* nemůže být hledání cest ukončené ve chvíli zjištění neexistence cesty (původně byl po řádce 15 ukončen cyklus pomocí příkazu **break**).

Další změnou je úprava podmínky na řádce 14. Tato podmínka je rozšířena o AND *pacmanToBoost* = *null*. To způsobí, že pokud existuje více agentů bez cesty, tak bude priorita zvýšena tomu prvnímu. Tento přístup je výhodnější vzhledem k tomu, že změna priority agenta s vyšší prioritou vždy může způsobit změny cest pro agenty s prioritou nižší (to může znamenat, že agenti bez cest nějakou cestu naleznou).

Poslední změnou je, že řádek 17 je v části **else**, protože pokud cesta pro agenta nebyla nalezena, tak nemůže být přidána do rezervační tabulky.

4.4 Algoritmy H+CA* a H+CGAA*

Jedná se o poslední dva nové algoritmy. Algoritmy Hierarchical+ Cooperative A* (H+CA*) a Hierarchical+ Cooperative Generalized Adaptive A* (H+CGAA*) jsou rozšířením algoritmů HCA* a HCGAA*. Oba tyto nové algoritmy, stejně jako ty původní, využívají jako heuristiku vzdálenost ignorující ostatní agenty. Tyto vzdálenosti jsou pro původní algoritmy získávány algoritmem Reverse Resumable A* (RRA*), představeným v podsekci „Hierarchical Cooperative A*“ v kapitole „Teoretická východiska“. V nových algoritmech bude aplikován algoritmus GAA* na RRA*.

Důvodem pro aplikování GAA* na RRA* je předpoklad, že RRA* nebude muset expandovat takové množství vrcholů při dodatečném hledání heuristiky pro spodní vrstvu algoritmu řešící MAPF problémy.

Aplikováním GAA* na RRA* vznikne algoritmus Reverse Resumable Generalized Adaptive A* (RRGAA*). Při této aplikaci GAA* je méně komplikací než při modifikaci GAA* pro CA*, protože zde jsou ignorováni ostatní agenti, takže nejsou žádná pole zablokována a následně uvolněna, a tedy nemusí být z tohoto důvodu heuristika opravována. Také při této aplikaci je pouze jeden cílový vrchol, protože hledání není prováděno na prostoru rozšířeném o dimenzi času.

Pseudokódy horních vrstev algoritmů H+CA* a H+CGAA* jsou stejné jako u algoritmů, ze kterých vycházejí. Tedy pro H+CA* je pseudokód horní

vrstvy stejný jako pro HCA* a pro H+CGAA* je pseudokód horní vrstvy stejný jako pro HCGAA*.

Na rozdíl od původních algoritmů, kde je před hledáním cest spodní vrstvou inicializován algoritmus RRA*, je u nových algoritmů toto chování změněno. Tato změna je popsána v následující podsekci.

4.4.1 Pseudokód pro RRGAA*

Algoritmus 4.4 obsahuje pseudokód pro RRGAA*. `InitializeVertex` je stejná procedura jako v původním GAA* a tedy není znovu uváděna. Procedura `SearchAfterPacmanMoved` je volána hned před tím, než horní vrstva využije spodní vrstvu pro nalezení cesty. Tato procedura provede vše potřebné pro udržování správných heuristik (u původního GAA* je toto provedeno na začátku volání procedury `Path`) a nakonec na řádku 63 zavolá proceduru `Resume`, ta expanduje vrcholy mezi agentem a jeho cílem. Heuristika pro další vrcholy požadované spodní vrstvou je dopočítána stejně jako v původním RRA*, a to tak, že je v případě potřeby obnoveno prohledávání.

Před úplně prvním hledání cesty spodní vrstvou se RRGAA* inicializuje použitím procedury `Initialize`. Ta kromě potřebných inicializací pro proměnné z RRA* (řádky 5, 12–14) i GAA* (řádky 2, 3, 6, 8–11) nakonec zavolá proceduru `Resume` (stejně jako u původního RRA*). Toto je jediný případ, kdy se před hledáním cesty nepoužije procedura `SearchAfterPacmanMoved`.

Procedura `Resume` je, oproti její původní podobě v algoritmu RRA*, změněna na třech místech. Všechny změny se týkají přidání prvků z GAA*. Na řádku 21 je přidáno volání procedury `InitializeVertex(neighbor)`. Druhá změna je na řádcích 31–33, kde, pokud je vrchol $toFind$ zároveň cílovým vrcholem $goal$, je uložena hodnota $g(start)$ do $pathCost(counter)$. Poslední změnou jsou řádky 37–39, kde v případě, že nebyla nalezena žádná cesta, je uložena hodnota ∞ do $pathCost(counter)$.

```
1  Procedure Initialize(start, goal):
2      counter ← 1
3      deltah(1) ← 0
4      foreach vertex v ∈ V do
5          g(v) ← ∞
6          search(v) ← 0
7      end
8      start ← vertex where start is now
9      goal ← vertex where goal is now
10     InitializeVertex(start)
11     InitializeVertex(goal)
12     g(goal) ← 0
13     openSet ← {goal}
14     Resume(start)
15 return
16 Procedure Resume(toFind):
17     while openSet is not empty do
18         current ← the vertex from openSet with the lowest f-value =
            g-value + h-value
19         Delete the vertex from openSet with the lowest f-value
20         foreach neighbor of current in reverse order do
21             InitializeVertex(neighbor)
22             if g(current) + d(current, neighbor) < g(neighbor) then
23                 g(neighbor) ← g(current) + d(current, neighbor)
24                 cameFrom(neighbor) ← current
25                 if neighbor ∉ openSet then
26                     Add neighbor to openSet
27                 end
28             end
29         end
30         if current = toFind then
31             if toFind = goal then
32                 pathCost(counter) ← g(start)
33             end
34             return success
35         end
36     end
37     if toFind = goal then
38         pathCost(counter) ← ∞
39     end
40 return failure
```

```
41 Procedure AbstractDist(toFind):
42 |   if toFind was already expanded then
43 |   |   return g(toFind)
44 |   end
45 |   if Resume(toFind) = success then
46 |   |   return g(toFind)
47 |   end
48 return ∞
49 Procedure SearchAfterPacmanMoved():
50 |   newStart ← vertex where start is now
51 |   if start ≠ newStart then
52 |   |   InitializeVertex(newStart)
53 |   |   if g(newStart) + h(newStart) < pathCost(counter) then
   |   |    h(newStart) ← pathCost(counter) − h(newStart)
54 |   |   deltah(counter + 1) ← deltah(counter) + h(newStart)
55 |   |   start ← newStart
56 |   else  deltah(counter + 1) ← deltah(counter)
57 |   goal ← vertex where goal is now
58 |   counter ← counter + 1
59 |   InitializeVertex(start)
60 |   InitializeVertex(goal)
61 |   g(goal) ← 0
62 |   openSet ← {goal}
63 |   Resume(start)
64 return
```

Algoritmus 4.4: Reverse Resumable Generalized Adaptive A*

Kapitola 5

Experimentální výsledky

V této kapitole budou jako první popsány implementační detaily softwarového prototypu. Následně budou vysvětleny společné prvky experimentů provedených softwarovým prototypem. Nakonec budou představeny jednotlivé experimenty, jejich výsledky a tyto výsledky budou zanalyzovány.

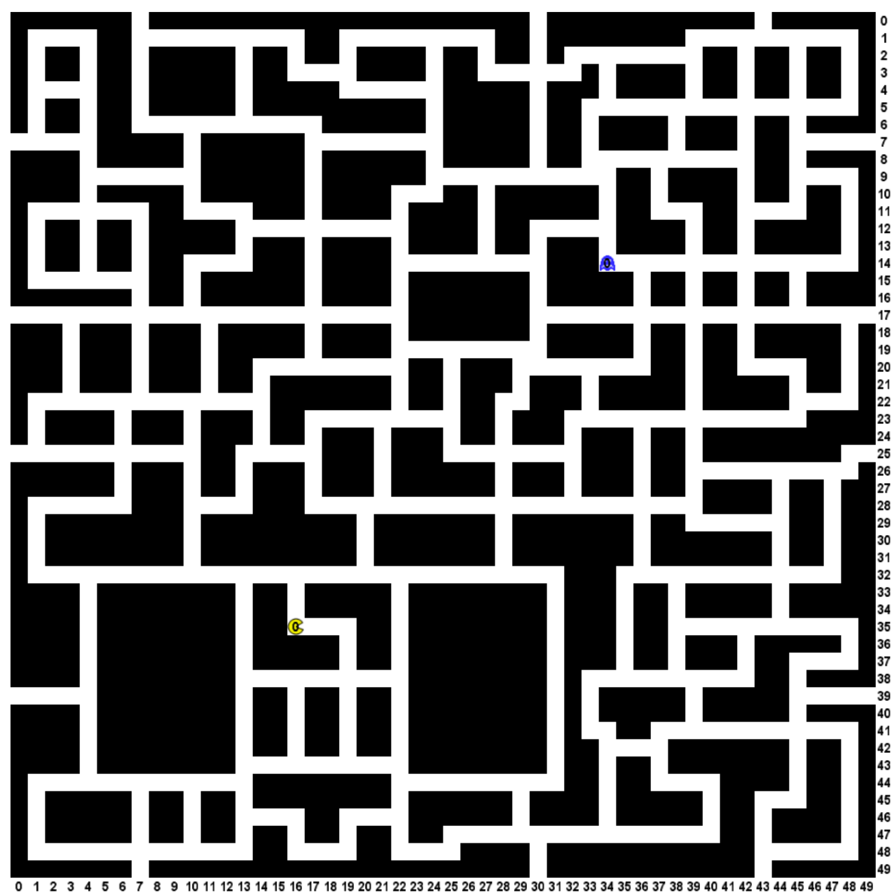
5.1 Implementační detaily softwarového prototypu

Softwarový prototyp je naprogramován v jazyce C# a pro vykreslování využívá open-source knihovnu MonoGame. Pro implementaci používá balíček OptimizedPriorityQueue. Samotný kód softwarového prototypu se nachází na přiloženém CD. Návod pro jeho použití je v příloze i na přiloženém CD.

5.2 Experimenty, jejich výsledky a jejich analýza

Pro otestování algoritmů je především využita mapa z původní hry, ta již byla předvedena na obrázku 2.2. Algoritmy jsou také testovány na nové mapě, která je na obrázku 5.1 a má rozměry 50×50 polí. Tato mapa má celkem 927 přístupných polí, to je více než trojnásobek vzhledem k původní mapě (ta má 300 přístupných polí, jak bylo dříve zmíněno v kapitole „Teoretická východiska“ v části věnující se mapě hry Pac-man). Tato mapa zachovává klíčové vlastnosti původní mapy, jmenovitě:

- mapa je tvořena pouze chodbami;
- neobsahuje žádné slepé chodby (neexistuje pole s pouze jedním přístupným sousedním polem);
- lze procházet hranami mapy.

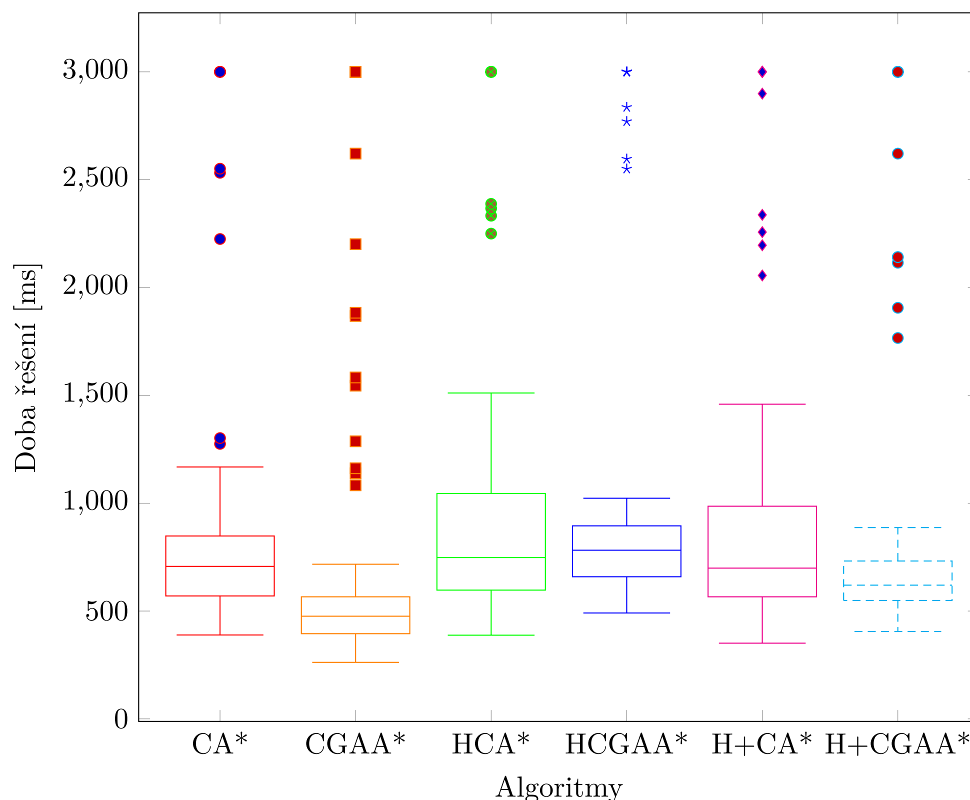


Obrázek 5.1: Nová mapa pro testování; rozměry mapy jsou 50×50 polí

Pro všechny testy je limit pro vyřešení jednoho problému nastaven na tři sekundy. Pokud do tohoto limitu nejsou všichni duchové chyceni, tak je řešení předčasně ukončeno. Tento limit se využije ve dvou různých případech:

1. problém se nestihne vyřešit kvůli jeho rozsáhlosti (velikost mapy, počet Pac-manů);
2. mapa je příliš zaplněná Pac-many a vzhledem k suboptimalitě všech testovaných algoritmů se objeví problémy s nalezením řešení problému.

Příkladem problémů druhého představeného typu mohou být situace, kde se Pac-manovi, který již chytil svého ducha, musí kvůli nemožnosti nalézt sekvenci akcí délky alespoň velikosti okna zvýšit priorita natolik, že předběhne Pac-many, kteří ještě svého ducha nechytili. Pro předběhnuté Pac-many je Pac-man, který je předběhl, překážkou. Tato skutečnost může v určitých případech vést k situaci, kde je Pac-man zablokován ostatními Pac-many, protože ti před ním zapisují své cesty do rezervační tabulky. V takovém případě musí



Obrázek 5.2: Krabicové grafy znázorňující dobu řešení pro sto instancí problému se 30 Pac-many na původní mapě s velikostí okna 20

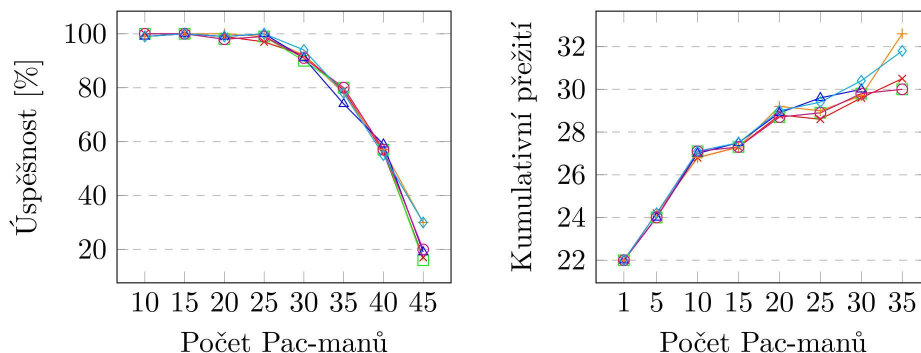
tento Pac-man čekat, dokud jeho duch nedorazí do části mapy, ve které je uvězněn.

Dalším příkladem problému druhého typu jsou situace, kde i přes změny priorit nelze nalézt cesty pro všechny Pac-many, a to vede k jejich hromadnému čekání. Toto čekání se opakuje do té doby, než se duchové nepřesunou do takových pozicí, kde problém zmizí. Po každém pohybu duchů se cesty zkouší hledat znovu, a to vede k dalšímu měnění priorit. Toto způsobuje značnou výpočetní zátěž.

V následujících podsekcích budou postupně popsány jednotlivé experimenty a jejich výsledky, také budou tyto výsledky zanalyzovány. Všechny výsledky měření, ze kterých byly jednotlivé grafy vytvořeny, jsou na přiloženém CD. Struktura výsledků měření je popsaná v příloze.

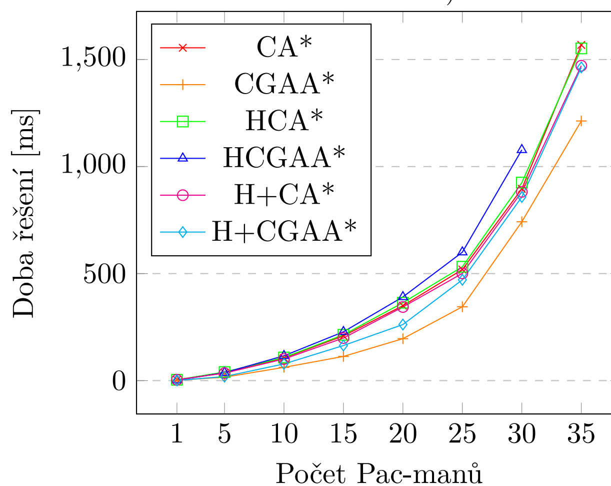
5.2.1 Testování vlivu počtu Pac-manů na původní mapě

Algoritmy byly testovány na původní mapě pro různé množství Pac-manů (1, 5, 10, ..., 45), pro každé množství Pac-manů bylo provedeno 100 testů



(a) Graf zobrazující procenta řešení, která doběhla před časovým limitem; pro neuvedená množství Pac-manů (1, 5) je úspěšnost 100 %

(b) Graf zobrazující průměrné kumulativní přežití duchů na jednoho ducha, to je počítáno z prostředních 50 % řešení (řešení mezi horním a dolním kvartilem)



(c) Graf zobrazující průměrnou dobu běhu algoritmu, průměr je počítán z prostředních 50 % řešení (řešení mezi horním a dolním kvartilem)

Obrázek 5.3: Grafy ukazující závislosti na počtu Pac-manů pro 100 instancí na původní mapě s velikostí okna 20; legenda v grafu (c) platí i pro grafy (a) a (b)

s náhodným rozmístěním Pac-manů a duchů. Velikost okna byla nastavena na 20. Grafy ukazující výsledky měření jsou na obrázcích 5.2 a 5.3.

Obrázek 5.2 obsahuje krabicový graf. Tento typ grafu byl zvolen pro svoji názornost zobrazení rozložení hustoty naměřených hodnot. Uvedený graf ukazuje dobu řešení pro jednotlivé algoritmy pro 30 Pac-manů. Na tomto grafu je vidět, že většina řešení na původní mapě doběhne v čase značně menším než je časový limit. Lze tedy předpokládat, že důvody pro předčasné ukončování jsou způsobené druhým uvedeným důvodem překročení limitu (problém suboptimality).

Graf na obrázku 5.3a ukazuje, kolik procent instancí problému se úspěšně dokončilo před násilným zastavením běhu algoritmu kvůli překročení limitu doby. V grafu je vidět, že úspěšnost klesá pro všechny algoritmy velice podobně. To podporuje předpoklad, že řešení je ukončené kvůli druhému představenému důvodu.

Graf na obrázku 5.3b ukazuje, jak se mění počet časových kroků, než Pacman chytil ducha. Tyto hodnoty jsou počítány z prostředních 50 % měření, kvůli měřením, která byla ukončena předčasně a nemůžou být tudíž použity při výpočtu průměru. Na tomto grafu je vidět, jak větší množství Pac-manů zvyšuje potřebné časové kroky k chycení ducha. Zároveň je z grafu patrné, že žádný z nových algoritmů nemá nijak výrazně horší výsledky kumulativního přežití duchů. To znamená, že nové algoritmy představené v této práci nenalézají horší řešení než původní algoritmy Davida Silvera.

Graf na obrázku 5.3c ukazuje průměrnou dobu běhu algoritmu. Průměr je počítán z prostředních 50 % měření. Na tomto grafu je vidět, že algoritmus CGAA* dosahuje pravidelně nejlepších výsledků v rychlosti řešení MAPF problémů.

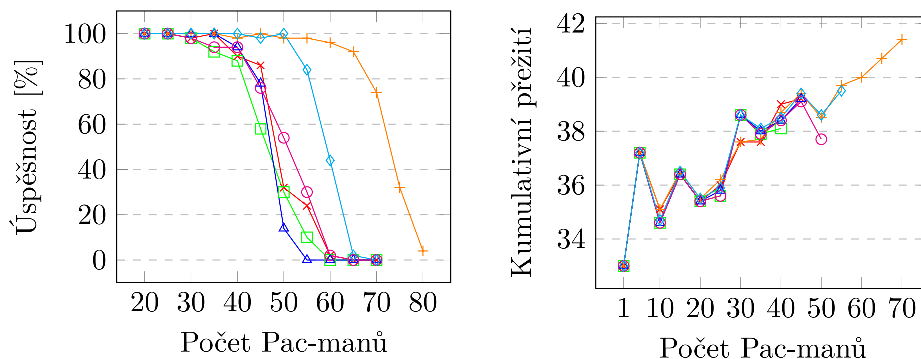
5.2.2 Testování vlivu počtu Pac-manů na nové mapě

Algoritmy byly testovány na nové mapě pro různé množství Pac-manů (1, 5, 10, . . ., 80), pro každé množství Pac-manů bylo provedeno 50 testů s náhodným rozmístěním Pac-manů a duchů. Velikost okna byla nastavena na 20. Grafy ukazující výsledky měření jsou na obrázku 5.4.

Graf na obrázku 5.4a ukazuje, kolik procent instancí problému úspěšně doběhlo před ukončením běhu algoritmu kvůli překročení limitu doby. Na rozdíl od měření na původní mapě zde klesá úspěšnost strměji. Algoritmům, které problémy řeší rychleji, začne klesat znatelně později. Z tohoto vyplývá, že se jedná o první důvod k ukončení běhu algoritmu (příliš rozsáhlé problémy).

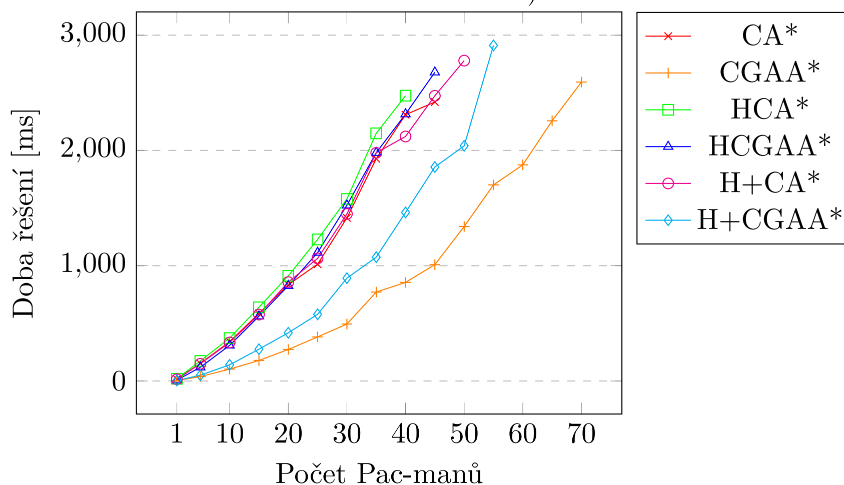
Graf na obrázku 5.4b ukazuje, jak se mění počet časových kroků, než Pacman chytil ducha. Tyto hodnoty jsou počítány z prostředních 50 % měření. Výsledky potvrzují analýzu provedenou u grafu 5.3b v minulé sekci.

Graf na obrázku 5.4c ukazuje průměrnou dobu běhu algoritmu. Průměr je počítán z prostředních 50 % měření. Na tomto grafu je vidět, že algoritmus CGAA* dosahuje pravidelně nejlepších výsledků v rychlosti řešení MAPF



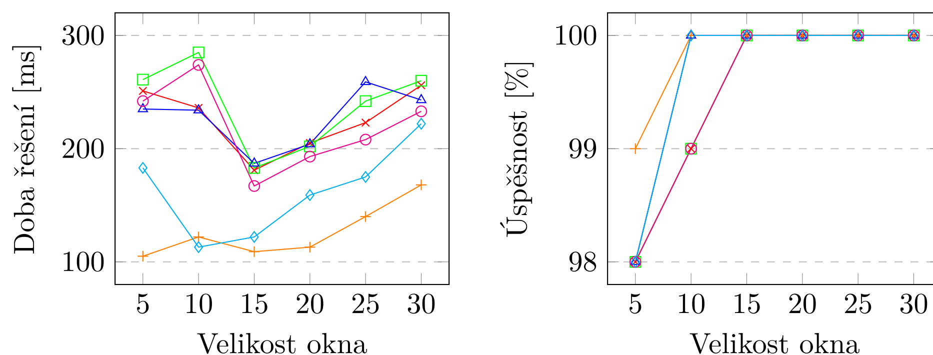
(a) Graf zobrazující procenta řešení, která doběhla před časovým limitem; pro neuvedená množství Pac-manů (1, 5, 10, 15) je úspěšnost 100 %

(b) Graf zobrazující průměrné kumulativní přežití duchů na jednoho ducha, to je počítáno z prostředních 50 % řešení (řešení mezi horním a dolním kvartilem)



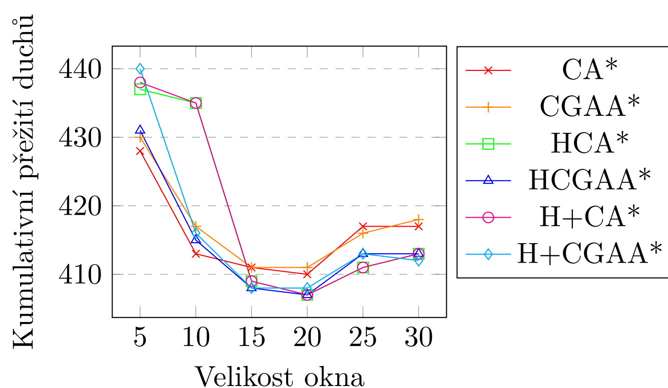
(c) Graf zobrazující průměrnou dobu běhu algoritmu, průměr je počítán z prostředních 50 % řešení (řešení mezi horním a dolním kvartilem)

Obrázek 5.4: Grafy ukazující závislosti na počtu Pac-manů pro 50 instancí na velké mapě s velikostí okna 20; legenda v grafu (c) platí i pro grafy (a) a (b)



(a) Graf zobrazující průměrnou dobu běhu algoritmu, průměr je počítán z prostředních 50 % řešení (řešení mezi horním a dolním kvartilem)

(b) Graf zobrazující procenta řešení, která doběhla před časovým limitem



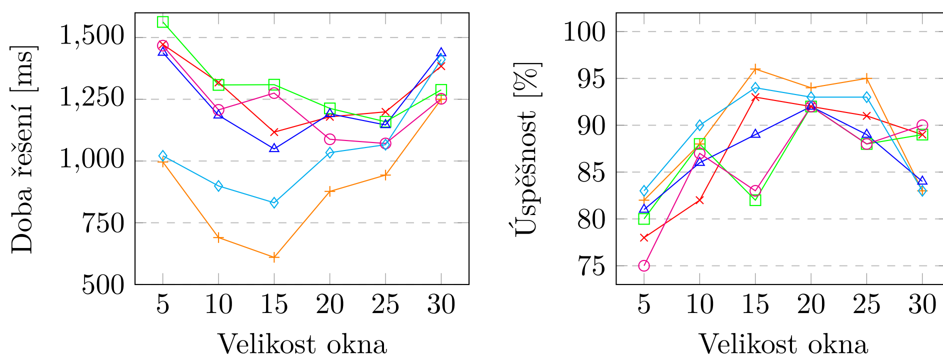
(c) Graf zobrazující kumulativní přežití duchů, to je počítáno z prostředních 50 % řešení (řešení mezi horním a dolním kvartilem)

Obrázek 5.5: Grafy ukazující závislosti na velikosti okna pro 100 instancí na původní mapě s 15 Pac-many; legenda u grafu (c) platí i pro grafy (a) a (b)

problémů. Zároveň je zde patrné, že algoritmus H+CGAA* dosahuje znatelně lepších výsledků než zbylé algoritmy. Tento graf také podporuje myšlenku, že úspěšnost v grafu na obrázku 5.4a klesá z důvodu příliš rozsáhlých problémů, protože průměrná doba řešení těsně před tím, než ji nelze spočítat kvůli příliš velkému množství předčasně ukončených řešení, je velice blízká časovému limitu tří sekund.

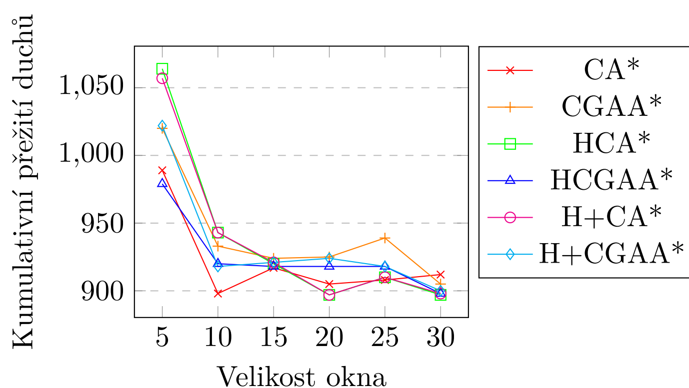
5.2.3 Testování vlivu velikosti okna

Algoritmy byly testovány na původní mapě pro různé velikosti okna (5, 10, 15, ..., 30), pro každou velikost okna bylo provedeno 100 testů s náhodným



(a) Graf zobrazující průměrnou dobu běhu algoritmu, průměr je počítán z prostředních 50 % řešení (řešení mezi horním a dolním kvartilem)

(b) Graf zobrazující procenta řešení, která doběhla před časovým limitem



(c) Graf zobrazující kumulativní přežití duchů, to je počítáno z prostředních 50 % řešení (řešení mezi horním a dolním kvartilem)

Obrázek 5.6: Grafy ukazující závislosti na velikosti okna pro 100 instancí na původní mapě s 30 Pac-many; legenda u grafu (c) platí i pro grafy (a) a (b)

rozmístěním Pac-manů a duchů. Vliv velikosti okna byl testován pro 15 a 30 Pac-manů. Grafy ukazující výsledky měření jsou na obrázcích 5.5 a 5.6.

Graf na obrázku 5.5a pro 15 Pac-manů a graf na obrázku 5.6a pro 30 Pac-manů ukazují závislost rychlosti nalezení řešení pro MAPF problém na velikosti okna. Tyto hodnoty jsou počítány z prostředních 50 % měření. Na těchto grafech lze vidět zpočátku klesání doby řešení, způsobené zkvalitněním plánů díky možnosti počítání s ostatními Pac-many dříve. Později doba řešení naopak roste, protože Pac-mani zbytečně plánují vyhýbat se ostatním Pac-manům, kteří jsou natolik daleko, že se jejich plány pravděpodobně změní kvůli pohybu duchů. Počítání s ostatními Pac-many je výpočetně náročné, proto také nalezení řešení pro větší okna trvá déle.

Graf na obrázku 5.5b pro 15 Pac-manů a graf na obrázku 5.6b pro 30 Pac-manů ukazují, kolik procent instancí problému úspěšně doběhlo před ukončením běhu algoritmu kvůli překročení limitu doby. Na těchto grafech je vidět, že zpočátku s rostoucí velikostí okna roste úspěšnost. To je především způsobené tím, že méně často nastává situace, kde je běh algoritmu ukončen předčasně, protože dojde k problému spojenému s druhým představeným důvodem k předčasnému ukončení běhu algoritmu. Vliv může mít i fakt, že doba řešení je vyšší a některé složitější instance nestihnou doběhnout. Na obrázku 5.6b pro 30 Pac-manů začíná od určité chvíle úspěšnost klesat. Toto může být způsobeno buď tím, že některé instance nestihnout doběhnout, protože velká velikost okna zpomalí běh algoritmu, nebo tím, že častěji nastávají problémy spojené s druhým představeným důvodem k předčasnému ukončení běhu algoritmu.

Graf na obrázku 5.5c pro 15 Pac-manů a graf na obrázku 5.6c pro 30 Pac-manů ukazují závislost kumulativního přežití duchů na velikosti okna. Hodnoty jsou počítány z prostředních 50 % měření. Na těchto grafech je vidět, že zpočátku s rostoucí velikostí okna klesá kumulativní přežití duchů. To je způsobené tím, že jsou nalezeny lepší plány pro Pac-many. Na grafu 5.5c pro 15 Pac-manů vzroste kumulativní přežití duchů pro velikosti okna 25 a 30. To může být způsobeno tím, že Pac-mani plánují cesty zbytečně dlouhé a v určitých případech si mohou zvolit zpočátku horší cestu. To kvůli tomu, aby se později vyhnuli duchovi, kterému by se vůbec vyhnout nemuseli, protože se jejich cesty stejně změní kvůli pohybu duchů.

5.3 Analýza algoritmů

V této části budou analyzovány algoritmy a jejich výsledky měření.

I přes to, že spodní vrstvy všech algoritmů vrací optimální cesty, tak je zadaný MAPF problém vyřešen odlišně. To je způsobeno tím, že optimálních cest může být více (cest, které mají stejnou délku a jsou nejkratší). Pokud jsou v určitou chvíli spodními vrstvami zvoleny různé optimální cesty Pac-manovi a_x pro MAPF problémy zatím ještě ve stejném stavu (Pac-mani i duchové jsou na stejných polích), tak to pravděpodobně bude mít za následek změnu průběhu řešení MAPF problému. Tato změna bude způsobena buď tím, že se Pac-man a_x přesune na jinou následující pozici a sám tedy změní řešení, anebo tím, že pro všechny Pac-many, kteří hledají cestu po Pac-manovi a_x, jsou v rezervační tabulce jiné záznamy, a tedy si možná budou muset zvolit jinou cestu. Tato skutečnost vede k tomu, že řešení MAPF problémů mohou být značně odlišná.

Pro algoritmy využívající Generalized Adaptive A* jako spodní vrstvu (jmenovitě CGAA*, HCGAA* a H+CGAA*) je vhodnější menší velikost okna než pro ostatní algoritmy. To je způsobené tím, že si tyto algoritmy ukládají aktualizované heuristiky a při větším okně je větší množství heuristik, které

je potřeba aktualizovat/opravit použitím procedury `FixHeuristics` dříve popsané v algoritmu 4.1. Samotné provedení této aktualizace trvá déle pro více polí, ale také heuristika po aktualizaci má méně užitečné hodnoty (hodnoty, které jsou dále od skutečné vzdálenosti k cíli).

Algoritmu HCA* trvá nalézt řešení déle než algoritmu CA*. To je způsobené tím, že HCA* používá pro získání heuristik algoritmus RRA*. Tento přístup není vhodný v mapách, které jsou tvořené pouze chodbami, protože výhoda přesných heuristik ani nevyváží časovou náročnost algoritmu RRA*.

Skutečnost, že HCA* není v testovaných mapách vhodný, je pravděpodobně jedním z důvodů, proč algoritmus CGAA* dosahuje lepších výsledků než algoritmy využívající RRA* pro získávání heuristik.

Algoritmus HCGAA*, oproti algoritmu CGAA*, nemá znatelně lepší výsledky než původní algoritmy. To je pravděpodobně také způsobené tím, že je při opravování heuristik potřeba inicializovat velké množství vrcholů (procedura `InitializeVertex` algoritmu 35). Při každé této inicializaci je potřeba získat základní heuristiku, to znamená případné další obnovení běhu algoritmu RRA*.

Závěr

Cíli teoretické části práce bylo vysvětlit potřebné termíny a udělat podrobnou rešerši existujících algoritmů týkajících se řešeného problému. Dalšími cíli práce bylo analyzovat problém a zdůvodnit volby konkrétních algoritmů. Hlavním cílem praktické části bylo navržení a implementace nového algoritmu nebo modifikace existujícího. V neposlední řadě bylo cílem experimentálně otestovat navržený algoritmus. Posledním cílem bylo výsledky zanalyzovat.

V teoretické části práce byly vysvětleny potřebné pojmy týkající se problému multi-agentního hledání cest s dynamickými cíli (například definice MAPF problému, definice heuristiky nebo popsání algoritmu A*), byla provedena potřebná rešerše různých přístupů k řešení problému a byly popsány algoritmy od Davida Silvera, jmenovitě CA*, HCA* a WHCA*, a algoritmus Generalized Adaptive A* (GAA*). Algoritmy od Davida Silvera byly různými způsoby modifikovány algoritmem Generalized Adaptive A* (GAA*). To vedlo k navržení a vytvoření čtyř nových algoritmů, jmenovitě Cooperative Generalized Adaptive A* (CGAA*), Hierarchical Cooperative Generalized Adaptive A* (HCGAA*), Hierarchical+ Cooperative A* (H+CA*) a Hierarchical+ Cooperative Generalized Adaptive A* (H+CGAA*).

Nové algoritmy byly následně experimentálně testovány a porovnány s výsledky algoritmů od Davida Silvera. Z analýzy těchto měření bylo zjištěno znatelné zrychlení vůči algoritmům od Davida Silvera pouze u algoritmu Cooperative Generalized Adaptive A* (CGAA*).

Bibliografie

1. SZCZEPANIAK, John. *The untold history of Japanese game developers.* SMG Szczepaniak, 2014. ISBN 0992926025.
2. NAMCO. *Pac-Man* [Computer software]. 1980.
3. PITTMAN, Jamey. The Pac-Man Dossier. In: *Gamasutra* [online]. 2009 [cit. 2021-03-23]. Dostupné z: `https://www.gamasutra.com/view/feature/132330/the_pacman_dossier.php`.
4. KORNHAUSER, D.; MILLER, G.; SPIRAKIS, P. Coordinating Pebble Motion On Graphs, The Diameter Of Permutation Groups, And Applications. In: *25th Annual Symposium onFoundations of Computer Science, 1984.* 1984, s. 241–250. Dostupné z DOI: `10.1109/SFCS.1984.715921`.
5. RYAN, Malcolm R. K. Graph Decomposition for Efficient Multi-Robot Path Planning. In: *IJCAI*. 2007, s. 2003–2008.
6. SILVER, David. Cooperative Pathfinding. *Aiide*. 2005, roč. 1, s. 117–122.
7. SHARON, Guni; STERN, Roni; FELNER, Ariel; STURTEVANT, Nathan R. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*. 2015, roč. 219, s. 40–66. ISSN 0004-3702. Dostupné z DOI: `https://doi.org/10.1016/j.artint.2014.11.006`.
8. SURYNEK, Pavel; FELNER, Ariel; STERN, Roni; BOYARSKI, Eli. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In: *Proceedings of the Twenty-second European Conference on Artificial Intelligence*. 2016, s. 810–818.
9. GILBOA, Arnon; MEISELS, Amnon; FELNER, Ariel. Distributed navigation in an unknown physical environment. In: *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. 2006, s. 553–560.

10. FELNER, Ariel; STERN, Roni; BEN-YAIR, Asaph; KRAUS, Sarit; NETANYAHU, Nathan. PHA*: Finding the shortest path with A* in an unknown physical environment. *Journal of Artificial Intelligence Research*. 2004, roč. 21, s. 631–670.
11. SHARON, Guni; STERN, Roni; GOLDENBERG, Meir; FELNER, Ariel. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*. 2013, roč. 195, s. 470–495.
12. PEARL, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company, 1984. Addison-Wesley series in artificial intelligence. ISBN 9780201055948.
13. BLACK, Paul E. Manhattan distance. In: *Dictionary of Algorithms and Data Structures* [online]. 2019 [cit. 2021-03-23]. Dostupné z: `https://xlinux.nist.gov/dads/HTML/manhattanDistance.html`.
14. WOLFE, Alan. Calculating the Distance Between Points in "Wrap Around" (Toroidal) Space. In: *The blog at the bottom of the sea* [online]. 2017 [cit. 2021-03-23]. Dostupné z: `https://blog.demofox.org/2017/10/01/calculating-the-distance-between-points-in-wrap-around-toroidal-space/`.
15. HART, Peter E.; NILSSON, Nils J.; RAPHAEL, Bertram. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*. 1968, roč. 4, č. 2, s. 100–107. Dostupné z DOI: `10.1109/TSSC.1968.300136`.
16. KOENIG, Sven; LIKHACHEV, Maxim. Adaptive A. In: *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. The Netherlands: Association for Computing Machinery, 2005, s. 1311–1312. AAMAS '05. ISBN 1595930930. Dostupné z DOI: `10.1145/1082473.1082748`.
17. KOENIG, Sven; LIKHACHEV, Maxim; SUN, Xiaoxun. Speeding up Moving-Target Search. In: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*. Honolulu, Hawaii: Association for Computing Machinery, 2007. AAMAS '07. ISBN 9788190426275. Dostupné z DOI: `10.1145/1329125.1329353`.
18. SUN, Xiaoxun; KOENIG, Sven; YEOH, William. Generalized Adaptive A*. In: *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*. Estoril, Portugal: International Foundation for Autonomous Agents a Multiagent Systems, 2008, s. 469–476. AAMAS '08. ISBN 9780981738109.

19. KOENIG, Sven; LIKHACHEV, Maxim. A New Principle for Incremental Heuristic Search: Theoretical Results. In: *Proceedings of the Sixteenth International Conference on International Conference on Automated Planning and Scheduling*. Cumbria, UK: AAAI Press, 2006, s. 402–405. ICAPS'06. ISBN 9781577352709.

Příloha A

Seznam použitých zkratek

MAPF Multi-Agent Path Finding

GAA* Generalized Adaptive A*

CA* Cooperative A*

HCA* Hierarchical Cooperative A*

RRA* Reverse Resumable A*

WHCA* Windowed Hierarchical Cooperative A*

CGAA* Cooperative Generalized Adaptive A*

HCGAA* Hierarchical Cooperative Generalized Adaptive A*

H+CA* Hierarchical+ Cooperative A*

H+CGAA* Hierarchical+ Cooperative Generalized Adaptive A*

Příloha B

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
results ........................... adresář s výsledky měření
exe ............ adresář se spustitelnou formou implementace
src
  impl ......................... zdrojové kódy implementace
  thesis ............. zdrojová forma práce ve formátu LaTeX
text ...................................... text práce
  thesis.pdf ..................... text práce ve formátu PDF
```

Příloha C

Sofwarový prototyp

Softwarový prototyp lze spustit pouze na Windows a pro jeho zapnutí stačí spustit „Bakalarska prace - Pac-man.exe“. Pro spuštění může být potřeba doinstalovat .NET framework 4. Pozor, při složitých měřeních systém Windows zobrazí hlášku, že program neodpovídá.

C.1 Návod pro použití softwarového prototypu

Nastavení softwarového prototypu je v souboru „setting.cfg“, který musí být umístěn ve stejné složce jako spouštěný program. Pokud nějaký parametr není definován nebo pokud hodnota parametru není ve správném formátu, je využita výchozí hodnota. Pokud hodnotou může být desetinné číslo, tak musí být použita desetinná tečka. Přípustné parametry a jejich hodnoty jsou:

- ResolutionWidth: <integer> – celé číslo udávající šířku okna na obrazovce v pixelech; (lze měnit i po zapnutí softwarového prototypu); výchozí hodnota je: 1366
- ResolutionHeight: <integer> – celé číslo udávající výšku okna na obrazovce v pixelech; (lze měnit i po zapnutí softwarového prototypu); výchozí hodnota je: 768
- Fullscreen: <boolean> – „true“ nebo „false“; umožňuje spustit softwarový prototyp v režimu celé obrazovky; výchozí hodnota je: false
- Scale: <decimal> – desetinné číslo udávající měřítko mapy; (lze měnit i po zapnutí softwarového prototypu); výchozí hodnota je: 1.0
- Seed: <integer> – celé číslo udávající počáteční seed pro generátor náhodných čísel, který je využit pro počáteční postavení Pac-manů, počáteční postavení duchů a pohyb duchů; pokud je parametr „RepeatSameMeasurement“ nastaven na „false“, tak jsou hodnoty seed pro vytváření

dalších problémů vygenerovány z prvního; výchozí hodnota je náhodné číslo získané od operačního systému

- RepeatSameMeasurement: <boolean> – „true“ nebo „false“; parametr označuje, zda má být opakován problém se zadanou hodnotou „seed“, nebo zda mají být postupně vytvořeny různé problémy na základě rozdílných hodnot seed vygenerovaných ze zadané hodnoty „seed“; výchozí hodnota je: false

- MapName: <file name with filename extension> – jméno souboru včetně jeho přípony, ze kterého je načtena mapa a případně i pozice Pac-manů a duchů (pokud je parametr „NumberOfPac-men“ nastaven tak, aby se využilo nastavení v tomto souboru); výchozí hodnota je: map.txt

- NumberOfPac-men: <integer> – počet Pac-manů (a duchů), kteří budou náhodně rozmístěni na mapě; pokud je parametr nastaven na číslo menší než 1, tak budou použity pozice Pac-manů a duchů ze souboru s mapou; výchozí hodnota je: -1

- SearchDepthLimit: <integer> – celé číslo udávající velikost okna (velikost rezervační tabulky); výchozí hodnota je: 20

- GhostsWaitingTime: <integer> – celé číslo udávající, kolik kol bude duch navíc čekat po svém tahu; (pokud je parametr nastaven na 1, tak bude Pac-man hrát dvakrát častěji než duchové); výchozí hodnota je: 1

- PerformedMeasurements: <integer> – celé číslo udávající, kolik problémů bude řešeno každým algoritmem při spuštění měření; výchozí hodnota je: 1

- ResultFileName: <file name with filename extension> – jméno souboru včetně jeho přípony, kam budou uloženy výsledky měření; výchozí hodnota je: Results.txt

- TerminateMeasurementTimeLimitInSec: <decimal> – desetinné číslo, které udává časový limit v sekundách, který když je překročen, tak je řešení problému ukončeno (měření pokračuje dalším problémem v pořadí; navíc se tento parametr použije pouze při měření); výchozí hodnota je: 3.0

C.2 Formát mapy

Soubor s mapou, která bude načtena, je zadán v parametru „MapName“. Musí se jednat o čistě textový soubor typu txt. V souboru s mapou musí být na začátku zadána mapa tvořená pouze znaky „X“ (velké písmeno x) značící nepřístupné pole (zeď) a „ “ (mezera) značící přístupné pole (chodba). Každý

řádek v souboru (standardně v editoru ukončen znaky konce řádku „\n“, „\r“ nebo „\r\n“) bude tvořit jeden řádek mapy. Mapa musí tvořit obdelník. Po mapě může soubor skončit nebo může následovat počáteční rozmístění Pac-manů a duchů. To je využito, pokud v parametru „NumberOfPac-men“ je nastaveno číslo menší než 1. Pozice každého Pac-mana je na samostatném řádku zadána ve formátu „start <integer>, <integer>“, kde první celé číslo indikuje pozici Pac-mana vodorovně (souřadnice X) a druhé celé číslo indikuje pozici Pac-mana svisle (souřadnice Y). Na dalším řádku následuje pozice ducha pro předchozího Pac-mana. Tato pozice je ve formátu „end <integer>, <integer>“. Celá čísla udávají jeho pozici stejně jako u Pac-mana. Další dvojice Pac-manů a jejich duchů jsou zadávány obdobně.

Příklad obsahu souboru s mapou:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
X            XX            X
X XXXX XXXXX XX XXXXX XXXX X
X XXXX XXXXX XX XXXXX XXXX X
X XXXX XXXXX XX XXXXX XXXX X
X                          X
X XXXX XX XXXXXXXX XX XXXX X
X XXXX XX XXXXXXXX XX XXXX X
X      XX    XX    XX      X
XXXXXX XXXXX XX XXXXX XXXXXX
XXXXXX XXXXX XX XXXXX XXXXXX
XXXXXX XX          XX XXXXXX
XXXXXX XX XXXXXXXX XX XXXXXX
XXXXXX XX XXXXXXXX XX XXXXXX
          XXXXXXXX          
XXXXXX XX XXXXXXXX XX XXXXXX
XXXXXX XX XXXXXXXX XX XXXXXX
XXXXXX XX          XX XXXXXX
XXXXXX XX XXXXXXXX XX XXXXXX
XXXXXX XX XXXXXXXX XX XXXXXX
X            XX            X
X XXXX XXXXX XX XXXXX XXXX X
X XXXX XXXXX XX XXXXX XXXX X
X   XX                XX   X
XXX XX XX XXXXXXXX XX XX XXX
XXX XX XX XXXXXXXX XX XX XXX
X      XX    XX    XX      X
X XXXXXXXXXX XX XXXXXXXXXX X
X XXXXXXXXXX XX XXXXXXXXXX X
X                          X
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
start 13, 23
end 14, 11
start 14, 23
end 13, 11
```

C.3 Ovládání softwarového prototypu

Po spuštění softwarového prototypu se objeví mapa s Pac-many a duchy. Napravo od mapy jsou vypsány některé statistiky a nastavení. Automaticky je jako výchozí algoritmus pro řešení inicializován algoritmus CA*. Stisknutím mezerníku se řešení problému posune o jeden časový krok dopředu. Stisknutím klávesy Enter se aktuální problém vyřeší. Klávesy A, S, D, F, G a H znovu načtou mapu a rozmístí Pac-many a duchy (podle hodnoty „seed“ nebo ze souboru – záleží na nastavení) a inicializují algoritmus CA*, CGAA*, HCA*, HCGAA*, H+CA* nebo H+CGAA* podle toho, která klávesa byla stisknuta. Klávesy Q, W, E, R, T a Y navíc oproti klávesám A, S, D, F, G a H spustí měření (je řešeno tolik problémů, kolik je nastaveno v parametru „PerformedMeasurements“; problémy jsou stejné/jiné podle parametru „RepeatSameMeasurement“ a výsledek je uložen do souboru s názvem podle parametru „ResultFileName“ (výstupní soubor je novým měřením přepsán); také je využit parametr „TerminateMeasurementTimeLimitInSec“). Stisknutím klávesy P se postupně provedou měření pro všechny algoritmy a výsledky se zapíší do souboru podle parametru „ResultFileName“. Klávesy + a − změní měřítko mapy (+ mapu zvětší, − mapu zmenší). Velikost okna softwarového prototypu lze změnit chycením kraje okna kurzorem a tažením.

C.4 Soubor s výstupem z měření

Na začátku souboru jsou vypsána nastavení, která byla aplikovaná při měření. Následují jednotlivá čísla seed, která byla využita pro jednotlivé MAPF problémy. Dále jsou postupně vypsány výsledky měření týkající se počtu časových kroků potřebných pro vyřešení problému „Turns“, kumulativního přežití duchů „Cumulative survival of ghosts“, expandovaných vrcholů „Fully expanded“, navštívených vrcholů „Generated“ a doby běhu algoritmu „Time“. Každá část výpisu týkající se konkrétního měřeného údaje má v souboru přímý výpis naměřených hodnot, součty naměřených hodnot, průměry naměřených hodnot a průměry počítané z prostředních 50 % naměřených hodnot. Všechny vypisované výsledky jsou rozděleny pro jednotlivé algoritmy.

Poté je vypsán počet předčasně ukončených měření pro jednotlivé algoritmy. V neposlední řadě jsou v souboru vypsány hodnoty specificky formátované pro jednodušší využití v grafech v této práci. Poslední vypsanou hodnotou měření je medián.

Příklad výstupu z měření spuštěného klávesou P:

```
Map name: map1.txt
Number of Pac-Man: 35
Search Depth Limit / Window Size: 20
Performed Measurements: 5
Repeat Same Measurement: False
Terminate Measurement Time Limit In Sec: 3
Seeds:
19546518; 1146105176; 231036127; 226938734; 1380450483;

Turns:
076; 078; 090; 055; 072;  - CA
035; 108; 090; 086; 049;  - CGAA
020; 096; 088; 094; 064;  - HCA
```

```
045; 043; 066; 066; 086;  - HCGAA
020; 096; 088; 094; 064;  - HpCA
045; 053; 056; 068; 080;  - HpCGAA
Sum Turns:
0371 - CA
0368 - CGAA
0362 - HCA
0306 - HCGAA
0362 - HpCA
0302 - HpCGAA
Avg Turns:
074 - CA
073 - CGAA
072 - HCA
061 - HCGAA
072 - HpCA
060 - HpCGAA
Avg Turn (Q1-Q3):
74 - CA
94 - CGAA
92 - HCA
58 - HCGAA
92 - HpCA
59 - HpCGAA

Cumulative survival of ghosts:
1114; 1104; 1157; 1093; 1075;  - CA
950; 1184; 1198; 1166; 1032;  - CGAA
626; 1178; 1245; 1181; 1072;  - HCA
1097; 979; 1206; 1110; 1144;  - HCGAA
626; 1178; 1245; 1181; 1072;  - HpCA
1097; 1081; 1109; 1116; 1202;  - HpCGAA
Sum cumulative survival of ghosts:
5543 - CA
5530 - CGAA
5302 - HCA
5536 - HCGAA
5302 - HpCA
5605 - HpCGAA
Avg cumulative survival of ghosts:
1108 - CA
1106 - CGAA
1060 - HCA
1107 - HCGAA
1060 - HpCA
1121 - HpCGAA
Avg cumulative survival of ghosts (Q1-Q3):
1118 - CA
1182 - CGAA
```

```
1201 - HCA
1098 - HCGAA
1201 - HpCA
1102 - HpCGAA

Fully expanded:
285270; 360352; 244575; 636161; 273062;  - CA
1740571; 267904; 218943; 198302; 1503476;  - CGAA
2065584; 444862; 252092; 321976; 277159;  - HCA
2124534; 2003022; 584973; 534955; 664212;  - HCGAA
1461211; 322746; 183804; 244767; 208206;  - HpCA
1495357; 1354290; 268578; 333875; 439836;  - HpCGAA
Sum Fully expanded:
1799420 - CA
3929196 - CGAA
3361673 - HCA
5911696 - HCGAA
2420734 - HpCA
3891936 - HpCGAA
Avg Fully expanded:
359884 - CA
785839 - CGAA
672334 - HCA
1182339 - HCGAA
484146 - HpCA
778387 - HpCGAA
Avg Fully expanded (Q1-Q3):
413696 - CA
228383 - CGAA
339643 - HCA
1040983 - HCGAA
250439 - HpCA
652247 - HpCGAA

Generated:
441487; 553549; 398833; 1165359; 402633;  - CA
2954411; 432441; 388155; 325978; 2725934;  - CGAA
2736234; 626887; 374242; 467574; 404893;  - HCA
2916537; 2759369; 706911; 640162; 813481;  - HCGAA
2120125; 505066; 304549; 389506; 335511;  - HpCA
2280402; 2225661; 355658; 457696; 601296;  - HpCGAA
Sum Generated:
2961861 - CA
6826919 - CGAA
4609830 - HCA
7836460 - HCGAA
3654757 - HpCA
5920713 - HpCGAA
Avg Generated:
```

```
592372 - CA
1365383 - CGAA
921966 - HCA
1567292 - HCGAA
730951 - HpCA
1184142 - HpCGAA
Avg Generated (Q1-Q3):
705913 - CA
382191 - CGAA
489567 - HCA
1368814 - HCGAA
399707 - HpCA
1013005 - HpCGAA

Time:
01.22; 01.43; 01.17; 03.00; 00.94;  - CA
03.00; 00.76; 00.72; 00.61; 03.00;  - CGAA
03.00; 01.49; 01.15; 01.18; 01.08;  - HCA
03.00; 03.00; 01.01; 00.91; 01.12;  - HCGAA
03.00; 01.36; 01.03; 01.09; 01.01;  - HpCA
03.00; 03.00; 00.72; 00.84; 01.06;  - HpCGAA
Sum Time:
00:00:07.7614055 - CA
00:00:08.0968669 - CGAA
00:00:07.8963328 - HCA
00:00:09.0339920 - HCGAA
00:00:07.4903974 - HpCA
00:00:08.6281906 - HpCGAA
Avg Time:
00:00:01.5522811 - CA
00:00:01.6193733 - CGAA
00:00:01.5792665 - HCA
00:00:01.8067984 - HCGAA
00:00:01.4980794 - HpCA
00:00:01.7256381 - HpCGAA
Avg Time (Q1-Q3) [ms]:
1869.48633333333 - CA
698.955633333333 - CGAA
1271.49283333333 - HCA
1637.85543333333 - HCGAA
1161.78486666667 - HpCA
1521.28456666667 - HpCGAA

Measurement Terminated:
1 - CA
2 - CGAA
1 - HCA
2 - HCGAA
1 - HpCA
```

```
2 - HpCGAA

Formated for graphs: (index, time)
(0,937)(1,1174)(2,1216)(3,1434)(4,3000) - CA
(0,611)(1,722)(2,764)(3,3000)(4,3000) - CGAA
(0,1082)(1,1146)(2,1177)(3,1492)(4,3000) - HCA
(0,907)(1,1006)(2,1120)(3,3000)(4,3000) - HCGAA
(0,1005)(1,1029)(2,1093)(3,1363)(4,3000) - HpCA
(0,720)(1,844)(2,1064)(3,3000)(4,3000) - HpCGAA

Formated for graphs: time double backslash
937\\1174\\1216\\1434\\3000\\ - CA
611\\722\\764\\3000\\3000\\ - CGAA
1082\\1146\\1177\\1492\\3000\\ - HCA
907\\1006\\1120\\3000\\3000\\ - HCGAA
1005\\1029\\1093\\1363\\3000\\ - HpCA
720\\844\\1064\\3000\\3000\\ - HpCGAA

Median:
1215 - CA
764 - CGAA
1177 - HCA
1120 - HCGAA
1093 - HpCA
1064 - HpCGAA
```