VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

PLÁNOVÁNÍ CESTY PRO VÍCE ROBOTŮ

PATH PLANNING FOR MULTIPLE ROBOTS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE **Bc. Ondřej Sekáč**

AUTHOR

VEDOUCÍ PRÁCE **RNDr. Jiří Dvořák, CSc.**

SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ | FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav automatizace a informatiky
Student:	**Bc. Ondřej Sekáč**
Studijní program:	Strojní inženýrství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**RNDr. Jiří Dvořák, CSc.**
Akademický rok:	2019/20

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Plánování cesty pro více robotů

Stručná charakteristika problematiky úkolu:

Plánování cesty pro více robotů patří mezi důležité problémy robotiky a má řadu dílčích variant. Roboti mohou nebo nemusejí spolupracovat, případně mohou dokonce soutěžit. Každý z robotů může mít svůj vlastní cíl, nebo může být zadán společný cíl pro celou skupinu. Roboti se mohou pohybovat individuálně nebo ve skupině, která se může nebo nemusí při obcházení překážek rozdělovat, případně musí během svého pohybu zachovávat formaci předepsaného tvaru. Úkolem systému pro plánování cesty více robotů je zajistit dosažení daného cíle či cílů tak, aby se roboti nedostali do kolize s překážkami nebo mezi sebou a aby byla optimalizována nějaká kriteriální funkce.

Cíle diplomové práce:

1. Analyzovat přístupy k plánování cesty pro více mobilních robotů.
2. Implementovat vybrané metody plánování cesty.
3. Provést a vyhodnotit ověřovací a srovnávací experimenty.

Seznam doporučené literatury:

AYARI, A., BOUAMAMA, S. Dynamic Distributed PSO Joints Elites in Multiple Robot Path Planning Systems: Theoretical and Practical Review of New Ideas. Procedia Computer Science, vol. 112, 2017, pp. 1082-1091.

KRAFT, A. R. Abstraction Hierarchies for Multi-Agent Pathfinding. 2017, Electronic Theses and Dissertations. 1247. https://digitalcommons.du.edu/etd/1247

LIANG, J.-H., LEE, C.-H. Efficient Collision-Free Path-Planning of Multiple Mobile Robots System Using Efficient Artificial Bee Colony Algorithm. Advances in Engineering Software, vol. 79, 2015, pp. 47-56.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2019/20

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

Abstrakt
Tato diplomová práce se zabývá plánováním cesty pro více mobilních robotů. Teoretická část se věnuje popisu navigace robota – mapování a plánování cesty. Jsou zde popsány vybrané metody umělé inteligence používané při plánování cesty pro více robotů. V praktické části je implementováno simulační prostředí, ve kterém byly vybrané algoritmy srovnány pomocí experimentů.

Summary
This master thesis deals with path planning for multiple mobile robots. The theoretical part describes robot navigation – mapping and path planning. Selected methods of artificial intelligence used in multi-robot path planning are described. In practical part simulator is implemented, in which selected algorithms were compared using experiments.

Klíčová slova
Plánování cesty, mobilní robot, kooperativní plánování

Keywords
Path planning, mobile robot, cooperative planning

SEKÁČ, O. *Plánování cesty pro více robotů.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2020. 72 s. Vedoucí diplomové práce RNDr. Jiří Dvořák, CSc.

Prohlašuji, že jsem diplomovou práci „Plánování cesty pro více robotů“ vypracoval samostatně pod vedením vedoucího práce RNDr. Jiřího Dvořáka, CSc. s použitím materiálů uvedených v seznamu literatury.

Bc. Ondřej Sekáč

Rád bych poděkoval vedoucímu této práce RNDr. Jiřímu Dvořákovi, CSc. za odborné vedení, trpělivost a cenné rady. Dále děkuji rodině a přítelkyni za podporu při studiu.

Bc. Ondřej Sekáč

Obsah

Úvod **13**

1 Plánování cesty **15**
1.1 Formulace problému plánování cesty . . . 15
1.1.1 Stavový prostor . . . 15
1.1.2 Pracovní prostor . . . 16
1.1.3 Konfigurační prostor . . . 16
1.1.4 Plánování cesty . . . 16
1.2 Plánování cesty pro více robotů . . . 17
1.3 Zpracování prostředí . . . 19
1.3.1 Mapy cest (Roadmaps) . . . 19
1.3.2 Rozklad do buněk (Cell decomposition) . . . 21
1.3.3 Potenciálová pole (Potential fields) . . . 22
1.4 Metody plánování cesty . . . 22
1.4.1 Klasické metody . . . 23
1.4.2 Pokročilé metody . . . 25

2 Vybrané algoritmy **29**
2.1 A* Algoritmus . . . 29
2.2 Local Repair A* . . . 33
2.3 Multi Agent D* Lite . . . 33
2.3.1 D* Lite . . . 33
2.3.2 Modifikace pro více robotů . . . 38
2.4 Cooperative A* . . . 39
2.4.1 Rezervační tabulka . . . 39
2.5 Hierarchical Cooperative A* . . . 41
2.5.1 Reverse Resumable A* . . . 42
2.6 Windowed Hierarchical Cooperative A* . . . 44

3 Popis aplikace **45**
3.1 Použité technologie . . . 45
3.2 Uživatelské rozhraní . . . 46
3.3 Plug-in systém . . . 49
3.4 MovingAI benchmarks . . . 50

4 Vyhodnocení výsledků **53**

Závěr **63**

Literatura **65**

Seznam použitých zkratek a symbolů **71**

Seznam příloh **72**

Úvod

V posledních letech dochází díky pokroku v oblasti informačních technologií také k rozvoji robotiky. Plánování cesty pro více robotů patří mezi jedny z nejdůležitějších problémů, kterými se robotika zabývá. Úkolem systému plánování cesty pro více robotů je nalézt cestu pro každého robota do daného cíle, při které se roboti budou vyhýbat překážkám i sobě navzájem, a zároveň optimalizovat nějakou kriteriální funkci. Použití více robotů s sebou přináší jak výhody, tak nevýhody. Plánování cesty ovšem není omezeno pouze na průmyslovou automatizaci, nachází uplatnění v dopravě, zemědělství, armádě ale i v řadě komerčních aplikacích, např. videohrách.

Cílem této diplomové práce je analyzovat a popsat přístupy k plánování cesty pro více mobilních robotů, vybrané metody implementovat a tyto metody poté srovnat pomocí experimentů.

První kapitola se věnuje obecnému popisu navigace robota, konkrétně popisu a zpracování prostředí, formulaci problému plánování cesty pro více robotů a obecnému popisu metod plánování cesty. Druhá kapitola popisuje vybrané algoritmy, které byly vybrány pro implementaci. Jedná se o algoritmy Local Repair A*, Multi Agent D* Lite, Cooperative A*, Hierarchical Cooperative A* a Windowed Hierarchical Cooperative A*. Ve třetí kapitole je popsáno simulační prostředí implementované v programovacím jazyce C#. V tomto prostředí byly poté provedeny srovnávací experimenty, které jsou zhodnoceny ve čtvrté kapitole.

1 Plánování cesty

Nutnou podmínkou pro fungování autonomního robota je jeho navigace, která se skládá ze tří procesů [1, 2, 3]:

1. Lokalizace – schopnost robota určit svoji polohu a orientaci v prostředí. Odpovídá na otázku „Kde se nacházím?“
2. Mapování – uložení dat získaných ze senzorů robota při prozkoumávání prostředí do dané reprezentace. Dává odpověď na otázku „Jak vypadá okolní svět?“.
3. Plánování cesty – Proces nalezení posloupnosti akcí, které vedou k dosažení daného cíle. Jedná se o otázku „Jak se dostanu do cílové pozice?“.

Lokalizace a mapování jsou navzájem provázané – při lokalizaci robota v prostředí je nutné znát jeho reprezentaci a pro správné mapování je nutné znát současnou polohu, ze které byla daná data získána. Plánování cesty je s těmito procesy úzce spjato, jelikož hledáme cestu v závislosti na současné poloze a reprezentaci prostředí.

1.1 Formulace problému plánování cesty

1.1.1 Stavový prostor

Každou jedinečnou situaci, do které se robot v prostředí může dostat, nazveme *stav* x. Je důležité aby stav obsahoval právě ty informace, které potřebujeme k vyřešení problému. Množina stavů (označovaných x) se nazývá *stavový prostor* X. Aplikací *akce* u na daný stav x přejdeme do stavu x', což je dáno tzv. *přechodovou funkcí* f,

$$x' = f\left(x, u\right). \tag{1}$$

Množinu $U(x)$, která reprezentuje všechny možné akce proveditelné ve stavu x nazveme *akčním prostorem*. Můžeme také definovat množinu všech možných akcí ve všech stavech jako

$$U = \bigcup_{x \in X} U(x). \tag{2}$$

Dále definujeme množinu *cílových stavů* $x_G \subset X$. Cílem obecného problému plánování je nalézt konečnou posloupnost akcí, která převede počáteční stav x_0 na některý z cílových stavů z X_G.

Tento problém je možné interpretovat jako *stavový přechodový graf*, ve kterém vrcholy reprezentují stavový prostor X a orientovaná hrana grafu ze stavu x do stavu x' reprezentuje akci u splňující funkci $x' = f(x, u)$ [4].

1.1.2 Pracovní prostor

Prostředí, ve kterém se robot pohybuje nazveme *pracovní prostor* W [3]. Jedná se o n-rozměrný Euklidovský prostor ($\mathbb{R}^2$ nebo $\mathbb{R}^3$). V pracovním prostoru se mohou vyskytovat různé překážky, ty značíme $O \subset W$. Překážky mohou být buď statické (nemění svoji polohu) nebo dynamické.

1.1.3 Konfigurační prostor

Plánování cesty přímo v pracovním prostoru je z hlediska časové náročnosti velice neefektivní, jelikož stavový prostor je široký. Přináší také veliké problémy při zohlednění stupňů volnosti, různých tvarů a dalších mechanických omezení robota, které jsou pro různé aplikace odlišné. Pro zobecnění se používá tzv. *konfigurační prostor* C [3, 5].

V konfiguračním prostoru je robot reprezentován jako bod. *Konfigurací* q se myslí kompletní popis polohy a natočení robota v pracovním prostoru W. Konfigurační prostor C je tedy množinou všech konfigurací q. Překážky O vymezují *kolizní konfigurační prostor* C_{obs} tj. ty konfigurace, ve kterých by byl robot v kolizi s překážkou. Tzv. *volný konfigurační prostor* je potom množinou všech přípustných konfigurací $C_{free} = C \setminus C_{obs}$. Nalezení *přípustné cesty* je potom zobrazení

$$p : [0; L] \to C_{free}, \tag{3}$$

kde L je délka cesty p.

1.1.4 Plánování cesty

Problém plánování cesty lze tedy pomocí konfiguračního prostoru transformovat na problém hledání cesty ve stavovém přechodovém grafu [6]. Uzly grafu jsou přípustné konfigurace $c \in C_{free}$. Každá hrana (tj. akce, přechod mezi danými konfiguracemi) má danou cenu.

Je důležité brát v potaz účel daného robota, jelikož různé aplikace mohou mít různé požadavky. Ve většině případů se jedná o optimalizaci ujeté vzdálenosti (tj. hledání nejkratší cesty). Dále je nutné dbát na účinnost, přesnost a bezpečnost robota i ostatních členů prostředí. Hledáme tedy ideálně cestu, při které se vyhneme kolizi s překážkami a dostaneme se do cíle v co nejkratším čase a za použití co nejméně energie [3].

1.2 Plánování cesty pro více robotů

Výše byl definovaný problém plánování cesty robota, ze kterého vycházíme při definici problému plánování cesty pro více robotů. Obecně se jedná o situaci, kdy máme m robotů v k-rozměrném pracovním prostoru a každý robot má danou startovní a cílovou konfiguraci, tj. pozici a orientaci. Je požadováno nalezení cesty pro každého robota, při které se budou roboti vyhýbat překážkám i sobě navzájem [7].

Případy využití několika robotů současně jsou stále častější [8]. Jedná se jak o použití v přepravě, průmyslu, zemědělství, rybaření, těžbě např. dřeva, hledání ztracených osob, prohledávání neznámých planet nebo likvidace toxického odpadu, tak o vojenské využití – řízení bezpilotních letounů, pokládání nebo zneškodnění min, atd.

Využití několika robotů může mít oproti použití pouze jednoho robota několik potenciálních výhod [8, 9]:

- Prostorové rozložení – vykonání úkonů v rozlehlých pracovních prostorech, které přesahují možnosti jednoho robota. Např. odpálení rakety otočením dvou klíčů současně.
- Celkový výkon systému – systém několika robotů může lépe optimalizovat cenovou funkci jako např. čas potřebný k vykonání úkolu nebo celkovou energii spotřebovanou roboty.
- Sdílení informací – např. více robotů je lépe schopno se lokalizovat navzájem, pokud si vyměňují informace.
- Cena – použití několika jednoduchých (levnějších) robotů, které lze snadněji naprogramovat, může být levnější než použití jednoho komplexního (drahého) robota.
- Spolehlivost, flexibilita – při selhání jednoho robota jej může nahradit další.

Úlohy plánování cesty pro více robotů lze rozdělit do několika skupin podle různých kriterií [3, 7, 10, 11, 12, 13]:

1. Podle typů jednotlivých robotů
   (a) *Homogenní* – schopnosti robotů jsou identické.
   (b) *Heterogenní* – schopnosti robotů jsou různé. Každý robot má vlastní specializaci pro daný úkol. Obecně se jedná o náročnější plánování.

2. Podle vzájemného chování robotů

   (a) *Kooperativní* – každý robot zná plány všech ostatních robotů. Roboti pracují společně na společném cíli. Speciálním případem je skupina mobilních robotů, která musí zachovávat předem určenou formaci, např. sekání fotbalového hřiště nebo přenášení nějakého předmětu více roboty.

   (b) *Nekooperativní* – roboti neznají plány ostatních robotů a musí tak předvídat jejich pohyby.

   (c) *Antagonistické* – každý robot se snaží dosáhnou svého cíle a případně zamezit ostatním robotům v dosažení jejich cílů.

3. Podle povahy prostředí

   (a) *Statické* – obsahuje pouze překážky, které nemění svoji polohu.

   (b) *Dynamické* – obsahuje pohybující se překážky (např. lidé).

4. Podle znalosti prostředí

   (a) *Globální plánování cesty* – roboti mají úplnou znalost pracovního prostoru před plánováním cesty.

   (b) *Lokální plánování cesty* – roboti mají neúplnou nebo žádnou znalost okolního prostředí. Musejí tedy v reálném čase snímat pomocí senzorů polohu překážek, vytvářet mapu prostředí a hledat v ní cestu.

5. Podle času provádění plánování

   (a) *Offline* – nejdříve je provedeno plánování cesty pro všechny roboty, poté se roboti podle těchto plánů začnou pohybovat.

   (b) *Online*, příp. *real-time* – plánování cesty je spojeno s pohybem robotů. Nalezená cesta nemusí být optimální nebo vůbec nalezena, roboti ale netráví dlouhý čas plánováním a dokáží rychle reagovat i na změny prostředí.

6. Podle přístupu k řešení problému

   (a) *Centralizované* – bere v úvahu všechny roboty zároveň jako jeden propojený systém. Snaží se o optimalitu a úplnost, proto v praxi trpí velikou časovou náročností.

   (b) *Distribuované* – rozdělí plánování na menší nezávislé nebo slabě závislé problémy, které řeší každý robot zvlášť. Schopné rychle nalézt dobré řešení, avšak ztrácí na úplnosti.

1.3 Zpracování prostředí

Pro zobecnění problému plánování cesty robota do prohledávání v konfiguračním prostoru C je nutná vhodná reprezentace prostředí, ve kterém se robot pohybuje. Jak uvádí například [4, 14], existuje několik přístupů ke zpracování prostředí, kdy k nejznámějším například patří *mapy cest*, *rozklad do buněk* nebo *potenciálová pole*.

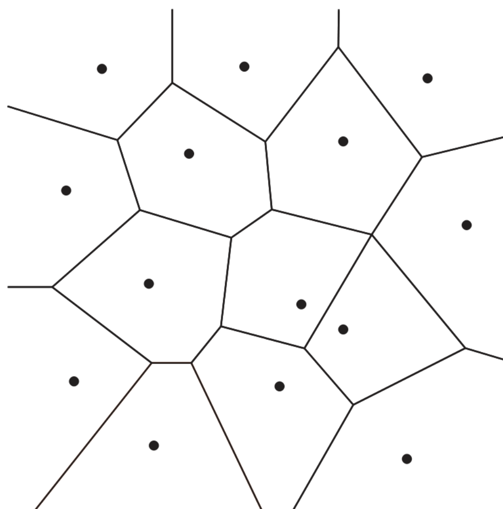
1.3.1 Mapy cest (Roadmaps)

Přístup ke zpracování prostředí pomocí map cest je založen na myšlence, že prostor C je zredukován či zmapován do sítě jednodimenzionálních křivek, které neprotínají překážky. Hledání řešení je poté omezeno na tuto síť a problém plánování cesty se stává problémem prohledávání grafu. Rozlišujeme dvě základní skupiny, a to *deterministické* a *pravděpodobnostní* metody.

1. **Deterministické metody**

   (a) **Voroného diagramy**

   Jedná se o metodu rozdělení prostoru podle určité množiny objektů (v našem případě například překážek). Voroného diagram je potom množina všech bodů, které jsou stejně vzdálené od dvou nebo více objektů [9, 15]. Tento diagram poté rozděluje prostor do oblastí, které obsahují právě jeden objekt. Cesta v tomto diagramu poté není nutně nejkratší, ale udává maximální možnou vzdálenost od překážek. Na obrázku 1 můžeme vidět příklad Voroného diagramu.

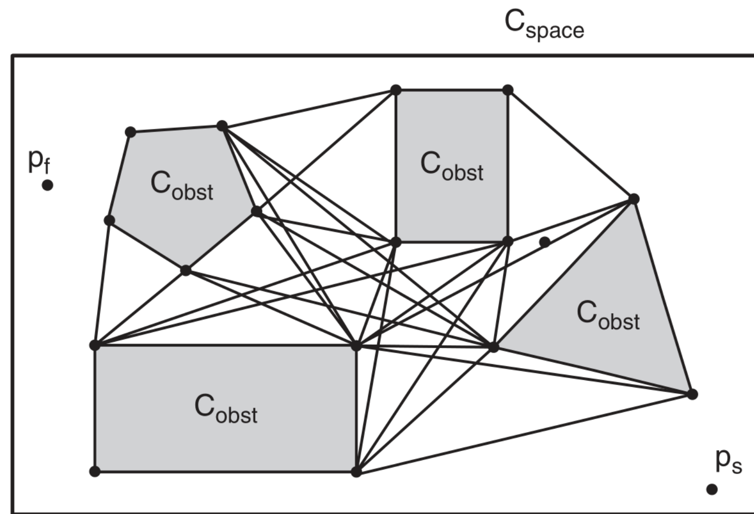


Obrázek 1: Voroného diagram 14 bodů v rovině [15]

   (b) **Graf viditelnosti**

   Vrcholy grafu viditelnosti jsou tvořeny vrcholy překážek a startovací a cílovou

pozicí. Hrany grafu jsou pak spojnice těch vrcholů, jejichž spojnice neprotíná žádnou překážku. Tvar překážek je omezen na polygon. Příklad grafu viditelnosti je zobrazen na obrázku 2.

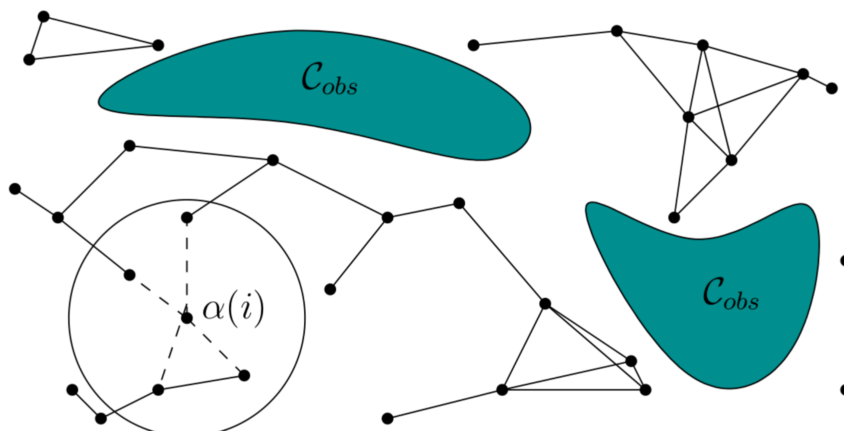


Obrázek 2: Graf viditelnosti [16]

2. **Pravděpodobnostní metody**

   (a) **Pravděpodobnostní mapy cest**

   Tato metoda náhodně generuje a propojuje velké množství bezkolizních konfigurací z volného konfiguračního prostoru C_{free}. Plánování cesty poté probíhá v takto vytvořeném grafu.



Obrázek 3: Příklad pravděpodobnostní mapy cest [4]

   (b) **Pravděpodobnostní stromy**

   Jedná se o metodu, kdy se postupně vytváří stromová struktura, která rychle prorůstá prostředím. Jedná se například o Rapidly-exploring Random Tree (RRT, rychle rostoucí náhodný strom), kde je konstrukce stromu náhodná – náhodná konfigurace z volného konfiguračního prostoru C_{free} je připojena

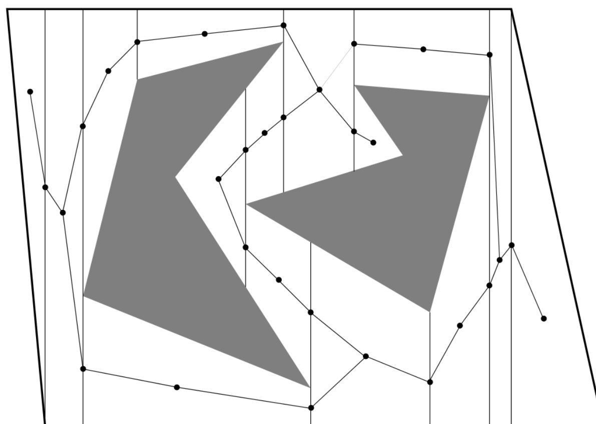
k nejbližší konfiguraci, která je součástí stromu. Příklad můžeme vidět na obrázku 4. Tato metoda je velice efektivní, s rostoucím počtem iterací roste pravděpodobnost dosáhnutí libovolného bodu v prostoru, která se limitně blíží 1 [4].

Obrázek 4: RRT po 45 iteracích (vlevo) a po 2345 iteracích (vpravo) [4]

1.3.2 Rozklad do buněk (Cell decomposition)

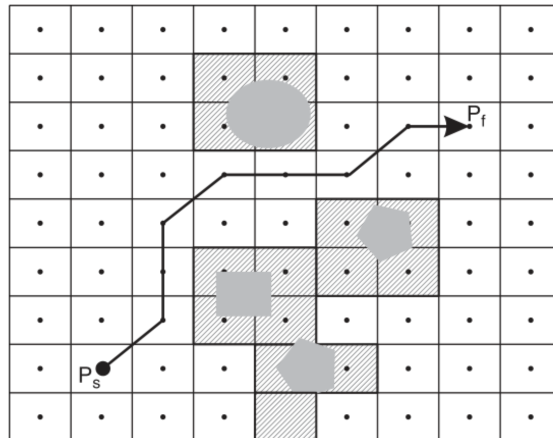
V metodě rozkladu do buněk je prostředí rozděleno na nepřekrývající se buňky. U každé buňky se určí, jestli obsahuje překážku či ne a na základě toho se vytvoří graf, kde vrcholy tvoří buňky neobsahující překážku a hrany pak spojnice mezi nimi. Rozlišujeme dva základní rozklady a to *aproximativní rozklad* a *exaktní rozklad*.

1. **Exaktní rozklad**
   Pracovní prostor je rozložen do vzájemně nepřekrývajících se buněk různého tvaru [4]. Mezi buňkami jsou umístěny body přechodu a nalezená cesta se pak skládá z počátečního stavu, ze kterého se jde skrze body přechodu do cílového stavu. Příklad exaktního rozdělení je na obrázku 5.



Obrázek 5: Příklad exaktního rozkladu [4]

2. **Aproximativní rozklad**
   Pracovní prostor je rozdělen do buněk stejného tvaru – většinou se jedná o čtverce, ale je možné použít také například šestiúhelníky nebo trojúhelníky [17]. Důležitou částí aproximace prostředí do této podoby je vhodná volba velikosti buněk. Čím menší velikost buněk použijeme, tím je rozklad přesnější, nicméně za cenu zvýšené výpočetní a paměťové náročnosti. Každá buňka je buď průchozí nebo neprůchozí, obsahuje-li alespoň část překážky. Příklad aproximativního rozkladu je zobrazen na obrázku 6 – vybarvené buňky obsahují překážky, jsou tedy označené za neprůchozí.



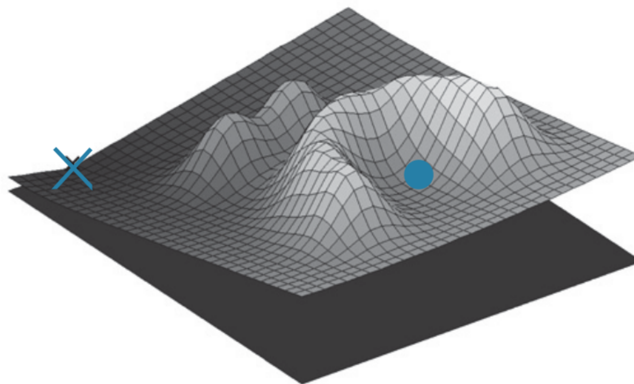
Obrázek 6: Příklad aproximativního rozkladu [16]

1.3.3 Potenciálová pole (Potential fields)

V tomto případě je robot reprezentován jako bod v konfiguračním prostoru a jako částice pod vlivem umělého potenciálového pole [16, 18, 19]. Cílový stav má přiřazen pozitivní potenciál, kdežto překážky mají odpudivý potenciál. Myšlenka je taková, že robot pohybující se v potenciálovém poli bude přitahován k cíli, zatímco bude odpuzován překážkami. Na rozdíl od mapy cest a rozkladu do buněk, cesta nalezena touto metodou následuje linii maximálního potenciálu spojité funkce potenciálového pole. Hlavní výhodou je rychlý výpočet potenciálové funkce. Tato metoda má však určité nevýhody, hlavní je, že se robot může zachytit v lokálním minimu, například když narazí na překážku ve tvaru písmene C. Další nevýhody jsou například oscilace v úzkých prostorech, případně úplné znemožnění cesty mezi blízkými objekty. Příklad potenciálového pole je na obrázku 7.

1.4 Metody plánování cesty

Po dokončení fází lokalizace a mapování (tj. zpracování prostředí) se pro plánování cesty mezi danou startovací a cílovou pozicí použijí konkrétní plánovací algoritmy. Výběr vhodného algoritmu je závislý na daném problému a jeho omezeních a požadavcích. Plánovacích



Obrázek 7: Potenciálové pole [15]

algoritmů existuje mnoho, každý se svými specifiky, uvedeme si zde proto nejvíce používané přístupy. Můžeme je rozdělit např. podle kriterií uvedených v podkapitole 1.2, případně je dělíme na tzv. metody klasické a pokročilé metody.

1.4.1 Klasické metody

Jak bylo řečeno v podkapitole 1.1, lze stavový prostor interpretovat jako graf, kde vrcholy představují všechny možné konfigurace robotů a hrany přechody mezi nimi. Cílem je najít posloupnost přechodů mezi těmito stavy takovou, aby se každý robot dostal z výchozí pozice do pozice cílové a aby se minimalizovala nějaká celková cenová funkce.

Obecně je problém plánování cesty pro více robotů NP-úplný. Metody, které tento problém řeší lze tedy rozdělit do dvou skupin:

1. *úplné* – vždy naleznou cestu (pokud existuje) nebo ověří, že neexistuje.
2. *neúplné* – nemusí nalézt žádnou cestu, i když existuje.

Vybrané neúplné metody budou představeny v kapitole 2, zde tedy představíme některé metody úplné, jak je popisují [20, 21, 22].

Grafové algoritmy

Řešení lze nalézt pomocí některého grafového algoritmu, jako je např. Dijkstrův [23] nebo A* (viz podkapitola 2.1). Uvažujeme-li prostředí reprezentované osmisměrnou mřížkou, má každý robot k dispozici 9 akcí (8 směrů + čekat). Pro n robotů je v každém kroku tedy možných až 9^n akcí. I když pro triviální případy je možné tento přístup využít, s rostoucím počtem robotů roste časová složitost i nároky na paměť exponenciálně.

Pattern databases

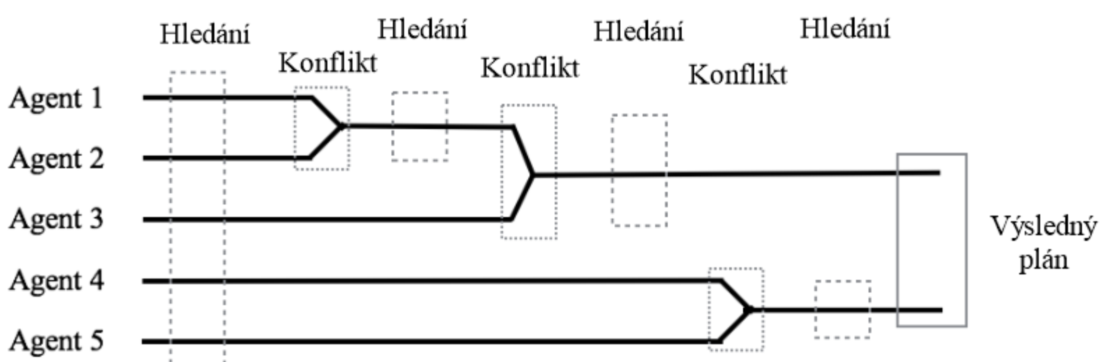
Pro některé problémy mohou být klasické heuristické funkce nedostatečné. Je možné využít techniky *Pattern database*, kdy se předem spočítají vzdálenosti do cíle v abstraktním stavovém prostoru (kde neuvažujeme některá omezení nebo proměnné). Tyto hodnoty jsou poté použity při hledání jako odhady vzdáleností v původním stavovém prostoru.

Operator Decomposition

Při expandování uzlů z fronty *open* dochází ke generování velkého počtu dalších uzlů, které nakonec nemusejí být ani použity. Pro redukci tohoto množství je možné použít metodu *Operator decomposition*. V této metodě jsou při expanzi uzlu brány v potaz pouze sousední stavy prvního robota, které se nazývají *přechodné*. Tyto stavy vkládáme normálně do fronty *open*, ale při jejich expanzi jsou brány v potaz pouze možné akce dalšího robota. Při expanzi uzlu posledního robota jsou generovány tzv. *standartní* uzly. Pouze tyto standartní uzly tvoří výslednou cestu.

Independence Detection

Často je počet robotů malý oproti velikosti pracovního prostoru, případně je jen málo míst kde by mohlo dojít ke kolizi mezi roboty. Není tedy nutné provádět hledání pro všechny roboty současně, ale můžeme vyhledat cestu pro každého zvlášť a poté zkontrolovat, jestli jsou nalezené cesty kolizní. Metoda *Independence Detection* rozdělí roboty do *nezávislých* skupin, tj. skupin, jejichž cesty nevedou ke kolizi. Nejprve je každý robot sám ve vlastní skupině. Pro každou skupinu je nalezena nejkratší cesta a simulován pohyb po této cestě (pro všechny skupiny současně). Pokud dojde ke konfliktu, je buď upravena cesta kolizních skupin nebo jsou tyto skupiny sloučeny. Tento proces se opakuje dokud nenalezneme bezkonfliktní cestu pro všechny skupiny. Celkový čas hledání je potom závislý na času hledání cesty největší skupiny.



Obrázek 8: Příklad hledání pomocí Independence Detection [22] (přeloženo)

Partial Expansion A*

Další metodou pro redukci velikosti prioritní fronty *open* je *Partial Expansion A** (PEA*). Při expanzi uzlu n jsou do prioritní fronty vloženy pouze ty uzly, pro které platí $f \leq f(n)$. Uzel n je opět vložen do fronty s hodnotou f rovnou nejnižší hodnotě f z expandovaných uzlů. Výhodou je tedy omezený počet uzlů ve frontě *open* a tedy i její nižší paměťová náročnost. Nevýhodou je ovšem opětovné expandování a generování uzlů, čímž může narůst časová náročnost algoritmu.

*Enhanced Partial Expansion A** staví na PEA*, využívá znalosti fungování použité heuristiky a snaží se minimalizovat čas potřebný k opětovnému expandování uzlů. Například při použití Manhattanské metriky dopředu víme, pro které uzly se hodnota f zvýší nebo sníží, v závislosti na poloze cílového uzlu. Tím je dosažena nejen paměťová, ale i časová úspora. Nevýhodou je ovšem nemožnost použití složitější heuristiky, jako např. Pattern database.

1.4.2 Pokročilé metody

Klasické metody mají několik hlavních nevýhod, především velkou časovou náročnost při vyšších dimenzích. Nabízí se využití některých meta-heuristik, které pomáhají při řešení optimalizačních problémů. I když tyto algoritmy nejsou úplné (tj. negarantují nalezení optimálního řešení), bývají často mnohem rychlejší než metody úplné. Kromě zde popsaných algoritmů patří mezi pokročilé metody například neuronové sítě, simulované žíhání, Tabu search nebo fuzzy logika. Konkrétní varianty těchto algoritmů pro řešení problému plánování cesty můžeme nalézt v [14].

Inteligence hejna

Inteligence hejna (SI – Swarm Intelligence) je systém kolektivního chování jednoduchých agentů, kteří spolu a se svým okolím lokálně interagují. Toto chování, i když není centrálně řízené, dává vzniknout složitému globálnímu chování hejna. Je to koncept založený na sociálním chování mravenců, ryb, ptáků, včel, světlušek atd. Podrobnější popis těchto algoritmů včetně srovnání jejich výhod a nevýhod se nachází v článku [24].

1. **Optimalizace hejnem částic**
   Optimalizace hejnem částic (PSO – Particle Swarm Optimization) poprvé popsali ve svém článku Kennedy a Eberhart v roce 1995 [25]. Jedná se o evoluční meta-heuristickou techniku inspirovanou sociálním chováním hejna ptáků při hledání potravy, uplatněním vzájemného sdílení informací. Toto chování je simulováno populací částic, kde každá částice představuje jedno konkrétní řešení. Tyto částice jsou náhodně rozmístěny ve stavovém prostoru, který postupně prohledávají a hledají optimální

řešení nebo alespoň řešení blízko optimálnímu. Každá částice má danou polohu P a vektor rychlosti v. V každé iteraci je vyhodnocena současná pozice částice pomocí tzv. *účelové funkce* a pokud je tato hodnota lepší než dosud nalezená, je uložena jako P_{best}. Nejlepší hodnota ze všech částic je uložena jako G_{best}. Dále je spočítána další pozice částice a její rychlost pomocí rovnic

$$v_{i+1} = v_i w + c_1 r(P_{best} - P_i) + c_2 r(G_{best} - P_i), \tag{4a}$$

$$P_{i+1} = P_i + v_{i+1}, \tag{4b}$$

kde r je náhodné číslo v intervalu $(0, 1)$, w je setrvačnost částice a c_1, c_2 jsou akcelerační koeficienty. Podmínka ukončení je většinou celkový počet iterací nebo určitý počet iterací, během nichž se hodnota účelové funkce nezmění o víc, než o předepsanou hodnotu.

Jednou z největších nevýhod je předčasná konvergence algoritmu, tj. možnost uvíznutí v lokálním optimu. Další nevýhodou je velká citlivost na dané parametry, které jsou pro každý problém různé. PSO a jeho varianty se nejčastěji uplatňují v problémech plánování cesty několika robotů, ve kterých skupina robotů prohledává neznámý prostor [10, 14, 26].

2. **Optimalizace mravenčí kolonií**

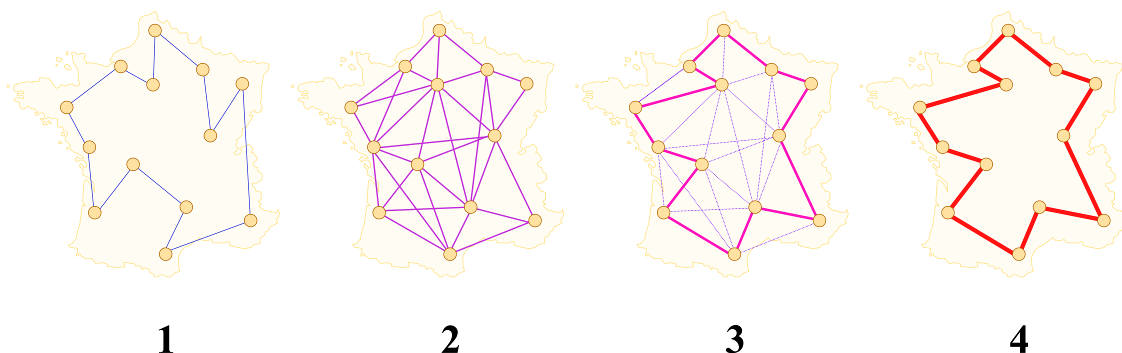
   Optimalizace mravenčí kolonií (ACO – Ant Colony Algorithm) je další ze skupiny algoritmů inteligence hejna. Jak název napovídá, jedná se o metodu inspirovanou chováním mravenců při hledání potravy. Mravenci jsou schopni nalézt nejkratší cestu k potravě a zpět do mraveniště a reagovat na změny v prostředí, aniž by využívali vizuální informace. Sdílení informací mezi mravenci probíhá pomocí feromonové (chemické) stopy. Při pohybu mravenců dochází k uvolňování feromonů na zem, čímž se označí cesta. Feromony se postupen času odpařují. Čím více mravenců se pohybuje po stejné trase, tím více je tato cesta atraktivnější pro další mravence. Jedná se o pozitivní zpětnou vazbu, kdy pravděpodobnost, že mravenec zvolí danou cestu je přímo úměrná počtu mravenců, kteří touto cestou prošli [27].

3. **Umělá včelí kolonie**

   Umělou včelí kolonii (ABC – Artificial Bee Colony) představil ve svém článku *An idea based on honey bee swarm for numerical optimization* v roce 2005 Dervis Karaboga [28]. Jedná se o metodu inspirovanou včelami hledajícími zdroje potravy v okolí úlu. Kolonie se skládá ze tří druhů včel: dělnice, hodnotící včely a průzkumnice. Dělnice jsou vyslány ke zdrojům nektaru, se kterým se poté vrací zpět do úlu. Poté prostřednictvím tance komunikují s ostatními včelami. Hodnotící včely podle tohoto tance vyberou zdroje potravy, ke kterým se vydají. Pokud je nějaký zdroj

vyčerpán, hodnotící včely se stávají průzkumnicemi. Průzkumnice provádí náhodné hledání v okolí úlu. Zdroje potravy reprezentují řešení a množství jejich nektaru odpovídá jeho kvalitě (fitness funkce) [24].

Výhodou ABC algoritmu je jeho jednoduchost, flexibilita a malé množství kontrolních parametrů, nevýhodou je nepřesnost nalezeného řešení a pomalá konvergence [29].



Obrázek 9: ACO použité pro řešení problému obchodního cestujícího [30]

Genetické algoritmy

John Holland poprvé předložil genetický algoritmus, založený na Darwinově teorii evoluce, již v roce 1960 [13, 14, 31, 32]. Jedná se o náhodný prohledávací algoritmus, kde řešení problému je zakódováno do chromozomů. Nejprve je vytvořena počáteční populace, která je složená z náhodných členů (chromozomů). Pro každý chromozom je poté při každé iteraci spočtena tzv. *fitness funkce*, která vyjadřuje kvalitu daného jedince. Podle této hodnoty jsou pomocí selekce vybráni jedinci určení k reprodukci. Poté dochází ke křížení – vzniku nových jedinců kombinací chromozomů dvou rodičů. Dále dochází k mutaci, kdy se mění genetická struktura některých jedinců. Tento cyklus se opakuje, dokud není splněna některá ukončovací podmínka, např. počet iterací nebo pokud se generace nevyvíjí dostatečně rychle.

Jednou z hlavních výhod genetického algoritmu je jeho jednoduchá paralelizace, nevýhodou je potom možnost uvíznout v lokálním minimu a jeho pomalá konvergence. Genetický algoritmus je také hojně využíván v kombinaci s ostatními přístupy, jako A*, ACO nebo potenciálovými poli.

2 Vybrané algoritmy

K implementaci byly v této diplomové práci zvoleny klasické metody, založené na grafových algoritmech. Plánování probíhá v konfiguračním prostoru získaném metodou rozkladu do buněk. Základním a nejpoužívanějším algoritmem pro hledání nejkratší cesty v grafu je A* algoritmus. Z něj vychází modifikace pro řešení problému plánování cesty pro více robotů popsaných v této kapitole.

2.1 A* Algoritmus

A algoritmus* ve svém článku *A formal basis for the heuristic determination of minimum cost paths* představili P. Hart, N. Nilsson a B. Raphael [33]. Jeho vstupem je graf, který má ohodnocené hrany (tzv. *ceny přechodu* z jednoho stavu do druhého), startovací a cílový stav. Výstupem je pak nejkratší cesta ze startovací do cílové pozice a nebo informace o tom, že taková cesta neexistuje.

Algoritmus využívá prioritní frontu (zvanou *OPEN*), ve které jsou jednotlivé stavy určené k expanzi seřazeny dle hodnoty *hodnotící funkce* f. Tato funkce má tvar

$$f(x) = h(x) + g(x), \tag{5}$$

kde $g(x)$ je funkce představující vzdálenost mezi počátečním stavem a aktuálním stavem a $h(x)$ je heuristická funkce, která představuje odhad vzdálenosti od aktuálního stavu do cílového stavu. Tato funkce musí být *přípustná*, což znamená že nesmí nadhodnocovat vzdálenost k cílové pozici. Další vlastností, kterou může mít tato heuristická funkce, je monotónnost. Takovou vlastnost má funkce, pokud splňuje podmínku

$$h(x) \leq d(x, y) + h(y), \tag{6}$$

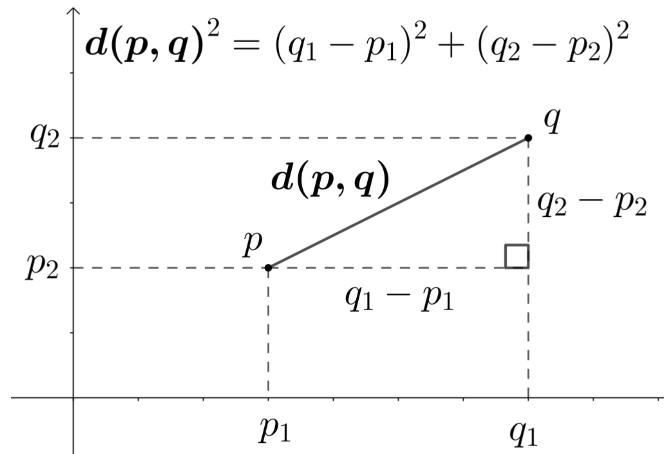
kde d je cena přechodu mezi stavem x a y. Pokud je tato vlastnost splněna, algoritmus navštíví každý uzel maximálně jednou.

Heuristická funkce může být libovolná, ale pro hledání nejkratší cesty je vhodné opírat se o teorii metrických prostorů. Zde představíme čtyři základní metriky a to *euklidovskou*, *manhattanskou*, *Čebyševovu* a metriku *octile* [34].

- *Euklidovská metrika* je dána vztahem

$$\rho_e(\vec{p},\vec{q}) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \qquad (7)$$

a představuje jednoduše délku vzdušné spojnice mezi stavem p a q, viz obr. 10.

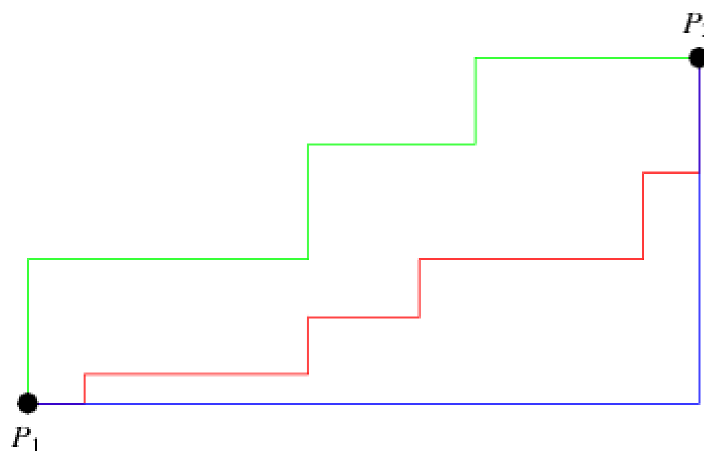


Obrázek 10: Euklidovská metrika [35]

- *Manhattanská metrika* je inspirovaná pravoúhlým systémem ulic na Manhattanu v New Yorku a je definovaná vztahem

$$\rho_m(\vec{p},\vec{q}) = \sum_{i=1}^{n}|q_i - p_i| \qquad (8)$$

Na obrázku 11 je zobrazen příklad manhattanské metriky, kde všechny cesty, ať už zelená, červená nebo modrá, mají stejnou délku.



Obrázek 11: Manhattanská metrika [36]

- *Čebyševova (maximální) metrika* počítá pouze s nejdelší složkou vzdálenosti, tj. je daná vztahem

$$\rho_c(\vec{p}, \vec{q}) = \max_{\forall i} |q_i - p_i| \,. \tag{9}$$

- *Octile metrika* kombinuje pohyb diagonální a přímý a mezi dvěma stavy (ve 2D mřížce) je definovaná jako

$$\rho_o(\vec{p}, \vec{q}) = \max(|p_1 - q_1|, |p_2 - q_2|) + (\sqrt{2} - 1) \cdot \min(|p_1 - q_1|, |p_2 - q_2|) \,. \tag{10}$$

Příklad Čebyševovy a octile metriky je zobrazen na obrázku 12.

1	1	1
1		1
1	1	1

a) Čebyševova metrika

$\sqrt{2}$	1	$\sqrt{2}$
1		1
$\sqrt{2}$	1	$\sqrt{2}$

b) Octile metrika

Obrázek 12: Hodnoty vzdáleností sousedních stavů na 2D mřížce

Nyní k samotnému popisu algoritmu. Jak již bylo řečeno na začátku této podkapitoly, algoritmus pracuje s prioritní frontou (*OPEN*) nenavštívených uzlů grafu. Uzly jsou seřazeny dle hodnoty funkce f a čím nižší je její hodnota, tím vyšší má uzel prioritu. V každé iteraci je uzel s nejnižší hodnotou funkce f odebrán a jsou spočítány hodnoty g a f jeho sousedních uzlů. V případě, že tyto uzly nejsou v prioritní frontě, tak jsou tam přidány, a pokud již ve frontě jsou, tak aktualizujeme hodnotu funkce f, pokud je nižší. Algoritmus pokračuje do té doby, dokud nenalezne koncový uzel nebo dokud nedojde k vyprázdnění fronty *OPEN*. Hodnota g koncového uzlu je pak hodnota nejkratší cesty ze startovacího uzlu do koncového. Tento postup je popsán v algoritmu 1.

Algoritmus 1 A* algoritmus

```
1:  function A*(start, goal)
2:      start.g ← 0
3:      start.f ← h(start, goal)
4:      Open ← {start}                          ▷ prioritní fronta (podle hodnot f)
5:      Closed ← ∅
6:      while Open ≠ ∅ do
7:          current ← Open.pop()                 ▷ prvek s nejlepší hodnotou f
8:          Closed.add(current)
9:          if current = goal then
10:             return ReconstructPath(current)
11:         end if
12:         for all neighbor ∈ Neighbors(current) do
13:             if neighbor ∈ Closed then
14:                 continue
15:             end if
16:             tempG ← current.g+Cost(current, neighbor)
17:             if tempG < neighbor.g then
18:                 neighbor.g ← tempG
19:                 neighbor.f ← tempG + h(neighbor, goal)
20:                 neighbor.parent ← current
21:                 if neighbor ∉ Open then
22:                     Open.add(neighbor)
23:                 end if
24:             end if
25:         end for
26:     end while
27:     return failure
28: end function

29: function ReconstructPath(current)
30:     path ← {current}
31:     while ∃current.parent do
32:         current ← current.parent
33:         path.prepend(current)
34:     end while
35:     return path
36: end function
```

2.2 Local Repair A*

Zatímco použití běžného A* algoritmu je při hledání cesty pro jednoho robota plně dostačující, při pohybu více robotů ve stejný čas může tento přístup lehce selhat a dojít ke srážce mezi roboty. Běžný postup, používaný například i ve videohrách, je využití takzvaného *Local Repair A** (LRA*) algoritmu [12, 33].

Local Repair A* nejprve vyhledá pro každého robota nejkratší cestu pomocí klasického A* algoritmu (viz podkapitola 2.1), nezávisle na ostatních. Bere v potaz pouze sousedy, kteří jsou v sousedních buňkách startovací pozice. Po vyhledání jednotlivých cest se po nich roboti začnou pohybovat. Při pohybu vždy robot kontroluje následující pozici, do které se má přesunout, a pokud by hrozila kolize, protože je na této pozici již jiný robot, označí tuto pozici jako překážku a přepočítá zbytek cesty do cíle s touto novou informací.

Je možné (a velice časté), že dojde k zacyklení jednoho nebo více robotů. Jedno z řešení tohoto problému je zvýšení tzv. *úrovně agitace* při každém přeplánování trasy. Při hledání je potom přidán náhodný šum úměrný úrovni agitace k hodnotě heuristické funkce h. Zvýšením náhodnosti chování robotů při plánování nové cesty se předpokládá, že uniknou z problematického místa, protože si vyberou jinou cestu.

Algoritmus Local Repair A* má několik nevýhod. Obzvláště v úzkých koridorech je velká pravděpodobnost ke vzniku zácpy, která může trvat dlouhou dobu, případně se ji nepodaří vyřešit vůbec. Roboti v zácpě neustále znovu plánují svoje cesty a tím dochází k opakovanému spouštění celého A* algoritmu. Toto vede k nepříznivému, „neinteligentnímu“ chování, případně k úplnému selhání hledání cesty.

Výhodou tohoto algoritmu je však jeho jednoduchost. V otevřených prostorech nebo problémech s málo roboty, kde nehrozí k častým kolizím, je jeho časová náročnost srovnatelná s klasickým A* algoritmem.

2.3 Multi Agent D* Lite

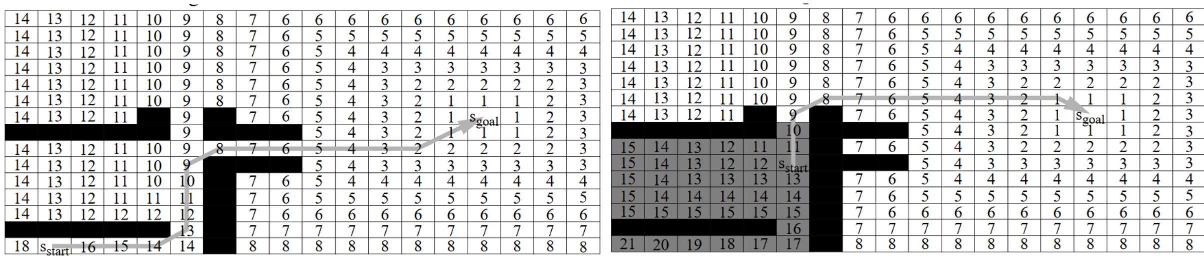
Jednou z hlavních nevýhod Local Repair A* je, že při hrozící kolizi dochází k přepočítání cesty pomocí A* algoritmu, který nedokáže využít informací z předchozích hledání. Nabízí se využití tzv. *replanning* algoritmů, které tuto vlastnost mají. Mezi tyto algoritmy patří například *Lifelong Planning A* (LPA*)* [37], *Dynamic A* (D*)* [38], *Focussed D** [39] nebo *D* Lite* [40].

2.3.1 D* Lite

D* Lite představují ve svém článku Sven Koenig a Maxim Likhachev [40]. Jedná se o rozšíření LPA* od stejných autorů, které používá stejnou strategii jako D*, jen je algoritmus

kratší a jednodušší na implementaci. Zároveň autoři tvrdí, že je tento algoritmus nejhůře stejně efektivní jako D*.

Na obrázku 13 můžeme vidět motivaci tohoto algoritmu. Robot se pohybuje v částečně neznámém prostředí, kde neznámé buňky považuje jako průchozí. Na obrázku 13a vidíme vzdálenosti do cíle pro každou buňku před zahájením pohybu robota. Jakmile se robot začne pohybovat a nalezne novou překážku, přepočítá hodnoty vzdáleností do cíle s touto novou informací. Nové hodnoty můžeme vidět na obrázku 13b, kde šedě jsou zvýrazněny hodnoty, které se od původních liší. Vidíme, že počet těchto buněk je malý, navíc jen minimum z nich je relevantních pro výpočet nejkratší cesty. Efektivní by tedy bylo, pokud bychom počítali nové hodnoty vzdáleností pouze pro ty buňky, které jsou důležité pro tento výpočet.



(a) Vzdálenosti do cíle před začátkem pohybu robota

(b) Vzdálenosti do cíle při nalezení nové překážky

Obrázek 13: Příklad situace vyžadující replanning [40]

D* Lite provádí prohledávání ve zpětném směru a tedy hodnota g každé buňky odpovídá nejkratší vzdálenosti do cílového stavu. Navíc počítá odhad této vzdálenosti o krok napřed, tzv. hodnotu rhs. Tato hodnota je dána vztahem

$$rhs(s) = \begin{cases} 0 & \text{pro } s = s_{start}, \\ \min_{s' \in Succ(s)} \left(g\left(s'\right) + c\left(s', s\right)\right) & \text{jinak}, \end{cases} \tag{11}$$

kde $c(s, s')$ je cena přechodu mezi stavy s a s' a $Succ(s)$ jsou následovníci stavu s. Stav s se nazývá *lokálně konzistentní*, pokud $rhs(s) = g(s)$. Pokud jsou všechny stavy lokálně konzistentní, lze nalézt nejkratší cestu ze stavu u do stavu v zpětným chodem ze stavu v tak, že vždy hledáme hranu (s, s') takovou, která minimalizuje $c(s, s') + g(s')$.

Jak bylo řečeno, není efektivní počítat tyto hodnoty pro všechny stavy, proto je použita heuristická funkce, která pomáhá vybírat ty stavy, které jsou důležité pro nalezení nejkratší cesty. Podobně jako v A* algoritmu je využita prioritní fronta. Tato fronta obsahuje lokálně nekonzistentní stavy, které je potřeba přepočítat. Klíčem v této frontě je dvousložkový vektor $k(s) = [k_1(s); k_2(s)]$, kde $k_1(s) = \min(g(s), rhs(s)) + h(s_{start}, s)$

a $k_2(s) = \min(g(s), rhs(s))$. Složka k_1 odpovídá f hodnotám v klasickém A* algoritmu a složka k_2 odpovídá hodnotám g. Klíče se porovnávají lexikálně, tj.

$$k(s) \leq k'(s) \Leftrightarrow k_1(s) < k'_1(s) \vee (k_1(s) = k'_1(s) \wedge k_2(s) \leq k'_2(s)).$$

Pseudokód algoritmu D* Lite je popsán v algoritmech 2 a 3. Hlavní funkce `Main` nejprve zavolá funkci `Initialize`, která nastaví hodnoty rhs a g pro všechny stavy na ∞. Pro cílový stav s_{goal} nastaví $rhs = 0$ a vloží tento stav do prioritní fronty U. Potom se zavolá funkce `ComputeShortestPath`, která provede hledání odpovídající A* algoritmu. Robot se poté začne pohybovat ze stavu s_{start} po nejkratší cestě směrem ke stavu s_{goal} (podél hrany (s, s'), která minimalizuje $c(s, s') + g(s')$). Pokud při pohybu zjistí, že se změnila cena některé hrany (u, v), zavolá metodu `UpdateVertex`, která přepočítá hodnotu rhs a klíč stavu u, případně jej odebere z fronty, pokud je nyní stav lokálně konzistentní, nebo jej do fronty zařadí, pokud je lokálně nekonzistentní. Poté se opět zavolá funkce `ComputeShortestPath`, která přepočítá nejkratší cestu ze stavu s_{start} do stavu s_{goal}.

Funkce `ComputeShortestPath` vybírá k expanzi lokálně nekonzistentní stav s nejmenším klíčem. Pokud je $g(s) > rhs(s)$, nastaví hodnotu $g(s) = rhs(s)$, jinak se nastaví hodnota $g(s) = \infty$. Tím je zaručeno, že je stav lokálně konzistentní. Poté se přepočítají hodnota g, hodnota rhs a příslušnost do prioritní fronty všech následníků stavu s, případně i stavu s samotného pomocí funkce `UpdateVertex`. Funkce `ComputeShortestPath` expanduje stavy, dokud je stav s_{start} lokálně konzistentní a klíč následujícího stavu určeného k expanzi není menší než klíč s_{start}. Pokud je $g_{start} = \infty$, potom neexistuje cesta ze stavu g_{start} do stavu g_{goal}.

Jelikož je při každé změně ceny nutné přepočítávat klíče prioritní fronty, docházelo by tím k neustálé změně pořadí v často rozsáhlé prioritní frontě, což je velmi výpočetně náročné. D* Lite tedy používá metodu převzatou z D*, díky které nemusí docházet ke změně pořadí v prioritní frontě. Při pohybu robota ze stavu s_{last} do stavu s_{start}, ve kterém je zjištěna změna cen přechodů, je hodnota složky k_1 klíčů prioritní fronty menší o maximálně $h(s_{last}, s_{start})$, hodnota složky k_2 nezávisí na h a je tedy nezměněna. Je tedy potřeba odečíst hodnotu $h(s_{last}, s_{start})$ od první složky všech klíčů stavů v prioritní frontě. Protože je tato hodnota pro všechny stejná, pořadí se touto operací nezmění. Místo toho při vkládání nového stavu do prioritní fronty přičteme tuto hodnotu k první složce klíče tohoto stavu. Hodnota $h(s_{last}, s_{start})$ se ukládá do proměnné k_m a je přičtena k první složce klíče ve funkci `CalculateKey`. Je potřeba ještě upravit funkci `ComputeShortestPath` tak, aby při odebrání stavu u s nejnižším klíčem k_{old}, přepočítala klíč tohoto stavu pomocí `CalculateKey`. Pokud je k_{old} < `CalculateKey`(u), je stav u opět vložen do prioritní fronty s tímto novým klíčem, jinak je tento stav expandován.

Algoritmus 2 D* Lite

$U.Insert(s, k)$ – vloží do prioritní fronty stav s s klíčem k
$U.Remove(s)$ – odstraní stav s z prioritní fronty
$U.TopKey()$ – vrátí nejmenší klíč z prioritní fronty ($[\infty; \infty]$ pokud je fronta prázdná)
$U.Pop()$ – odebere z fronty stav s nejmenším klíčem a vrátí jej

1: **function** CALCULATEKEY(s)
2: **return** $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))]$
3: **end function**

4: **function** INITIALIZE
5: $s_{last} \leftarrow s_{start}$
6: $U \leftarrow \emptyset$
7: $k_m \leftarrow 0$
8: $\forall s \in V : rhs(s) \leftarrow g(s) \leftarrow \infty$
9: $rhs(s_{goal}) \leftarrow 0$
10: $U.Insert(s_{goal},$ CALCULATEKEY$(s_{goal}))$
11: **end function**

12: **function** UPDATEVERTEX(u)
13: **if** $u \neq s_{goal}$ **then**
14: $rhs(u) \leftarrow \min_{s' \in Succ(u)}(c(u, s') + g(s'))$
15: **end if**
16: **if** $u \in U$ **then**
17: $U.Remove(u)$
18: **end if**
19: **if** $g(u) \neq rhs(u)$ **then**
20: $U.Insert(u,$ CALCULATEKEY$(u))$
21: **end if**
22: **end function**

Algoritmus 3 D* Lite (pokračování)

```
23: function ComputeShortestPath( )
24:     while U.TopKey() < CalculateKey(s_start) ∨ rhs(s_start) ≠ g(s_start) do
25:         k_old ← U.TopKey()
26:         u ← U.Pop()
27:         if k_old < CalculateKey(u)) then
28:             U.Insert(u, CalculateKey(u))
29:         else if g(u) > rhs(u) then
30:             g(u) ← rhs(u)
31:             for all s ∈ Pred(u) do
32:                 UpdateVertex(s)
33:             end for
34:         else
35:             g(u) ← ∞
36:             for all s ∈ Pred(u) ∪ u do
37:                 UpdateVertex(s)
38:             end for
39:         end if
40:     end while
41: end function

42: function Main( )
43:     Initialize( )
44:     ComputeShortestPath( )
45:     while s_start ≠ s_goal do
46:         if g(s_start) = ∞ then
47:             return failure
48:         end if
49:         s_start ← arg min_{s′∈Succ(s_start)}(c(s_start, s′) + g(s′))
50:         move to s_start
51:         scan graph V for changed edge cost
52:         if any edge costs changed then
53:             k_m ← k_m + h(s_last, s_start)
54:             s_last ← s_start
55:             for all edges (u, v) with changed edge costs do
56:                 update c(u, v)
57:                 UpdateVertex(u)
58:             end for
59:             ComputeShortestPath( )
60:         end if
61:     end while
62: end function
```

2.3.2 Modifikace pro více robotů

Kombinací metod plánování Local Repair A* a vyhledávání nejkratší cesty pomocí D* Lite dostáváme algoritmus použitelný pro plánování cesty více robotů. Pseudokód tohoto algoritmu, který nazveme *Multi Agent D* Lite (MA D* Lite)* můžeme vidět v algoritmu 4.

Algoritmus 4 Multi Agent D* Lite

```
 1: function Main( )
 2:     for all r ∈ Robots do
 3:         r.Initialize( )
 4:         r.ComputeShortestPath( )
 5:     end for
 6:     enRoute ← Robots where r.s_start ≠ r.s_goal
 7:     while count(enRoute) > 0 do
 8:         for all r ∈ enRoute do
 9:             occupied ← {s ∈ Succ(r.s_start) where s is s_start of any robot}
10:             Step(r, occupied)
11:             update enRoute
12:         end for
13:     end while
14: end function

15: function Step(r, occupied)
16:     if occupied changed since last call then
17:         r.k_m ← r.k_m + h(r.s_last, r.s_start)
18:         r.s_last ← r.s_start
19:         r.c(u, v) ← ∞ if u ∈ occupied ∨ v ∈ occupied otherwise 1
20:         for all u ∈ occupied do
21:             r.UpdateVertex(u)
22:         end for
23:         r.ComputeShortestPath( )
24:     end if
25:     if g(r.s_start) = ∞ then
26:         return failure
27:     end if
28:     r.s_start ← arg min_{s'∈Succ(r.s_start)}(c(r.s_start, s') + g(s'))
29: end function
```

Hlavní obslužná funkce `Main` zavolá pro každého robota funkce `Initialize` a `ComputeShortestPath`, které jsou identické s implementací popsanou v podkapitole 2.3.1. Každý robot si uchovává vlastní informace, tj. prioritní frontu, hodnoty g, rhs, k_m, hodnoty cen přechodů c a stavy s_{start} (aktuální pozice), s_{goal} (cílová pozice) a s_{last} (pozice poslední změny cen přechodů). Dále následuje hlavní smyčka algoritmu, kdy každý robot, který není v cílové pozici, ověří, zda-li se v jeho okolí vyskytují nějaký další robot. Pokud

ano, označí dané pozice jako překážky (*occupied*). Poté se zavolá funkce `Step`, která je analogií funkce `Main` z podkapitoly 2.3.1.

Funkce `Step` nejprve ověří, zda se změnily pozice označené jako *occupied*. Pokud ano, nastaví příslušné hodnoty pro k_m a s_{last}. Poté nastaví hodnoty cen přechodů pro stavy v *occupied* jako ∞, pro ostatní jako 1. Pro všechny stavy v *occupied* také přepočítá hodnoty g, rhs a příslušnost do prioritní fronty pomocí funkce `UpdateVertex`. Dále se přepočítá nejkratší cesta funkcí `ComputeShortestPath` s těmito novými informacemi. Pokud je $g(s_{start}) = \infty$, cesta neexistuje, jinak se nastaví současná pozice s_{start} na pozici $s' \in Succ(s)$ tak, aby se minimalizovala hodnota $c(s_{start}, s') + g(s')$.

2.4 Cooperative A*

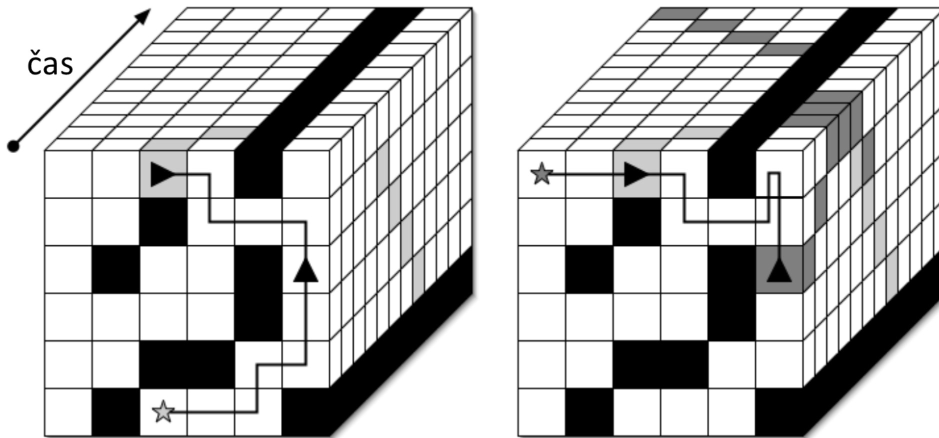
*Cooperative A** [12, 41] se snaží vyřešit problém LRA*, kdy roboti navzájem neznají plány ostatních. Je tedy potřebné, mít znalost nejen o statických překážkách, ale i o pohybu všech robotů. Jelikož není žádná možnost, jak do statické (2D) mapy zaznamenat dynamický pohyb robotů, je potřeba mapu rozšířit o další dimenzi – čas. Při pohybu mřížkou se tedy nemění pouze souřadnice polohy, např. $(x, y) \rightarrow (x, y + 1)$, ale i časová souřadnice $(x, y, t) \rightarrow (x, y + 1, t + 1)$. V některých situacích, například při zácpě před úzkým koridorem, je výhodné nedělat nic a počkat, než se prostor vyčistí. Je tedy vhodné zavést jednu další akci *čekat (wait)*, kdy přecházíme ze stavu (x, y, t) do stavu $(x, y, t + 1)$.

Hledání cesty tedy provádíme na této 3D mřížce, kde cena přechodu mezi stavy je rovna době, kterou zabere přechod mezi nimi, tedy jedna. Přechod je přípustný, pokud v cílové buňce není překážka nebo jiný robot. A* algoritmus nalezne cestu s nejnižší cenovou funkcí, tj. nejrychlejší cestu do cíle. Tato cena může být vyšší než samotná délka cesty, kvůli již zmíněné akci *čekat*.

2.4.1 Rezervační tabulka

Pro zohlednění pohybu robotů se využívá tzv. *rezervační tabulka*. Po té, co každý robot naplánuje svoji cestu, je každý stav této cesty zaznamenán do rezervační tabulky. Každý stav zanesený do této tabulky je při plánování dalšími roboty považován za překážku na dané pozici v daném čase.

Na obrázku 14 můžeme vidět, jak Cooperative A* funguje. První robot (vlevo), nalezne cestu do svého cíle a jednotlivé stavy zaznamená do rezervační tabulky (světle šedá). Druhý robot při svém plánování (vpravo) považuje tyto stavy za překážky, tím se vyhne kolizi s prvním robotem a úspěšně nalezne cestu do cíle. Tuto cestu poté také zaznamená do rezervační tabulky (tmavě šedá).

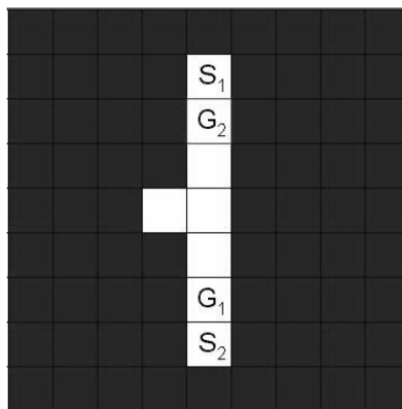


Obrázek 14: Rezervační tabulka [41] (přeloženo)

Rezervační tabulka bohužel nezohledňuje situaci, kdy roboti jedou přímo naproti sobě. Pokud například první robot zarezervuje stavy (x, y, t) a $(x + 1, y, t + 1)$, druhému robotovi nic nebrání v tom zarezervovat stavy $(x + 1, y, t)$ a $(x, y, t + 1)$. Řešením je buď zaznačit do rezervační tabulky jak cílový, tak výchozí stav akce, případně explicitně kontrolovat přímé kolize.

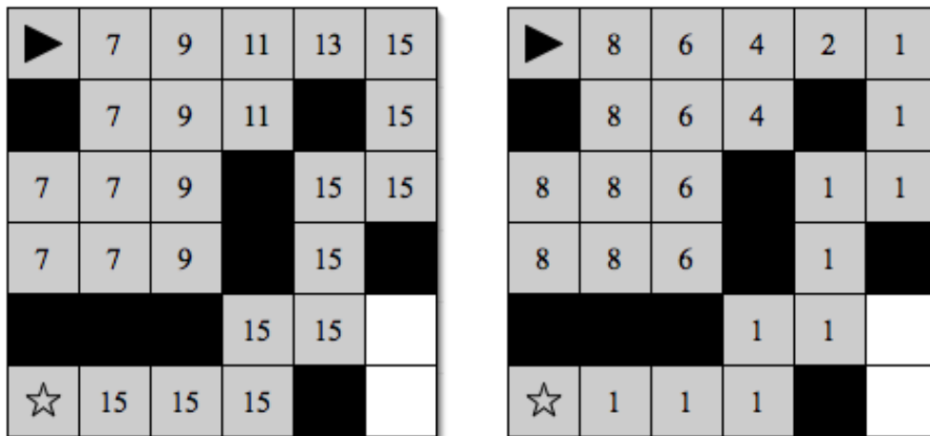
Jednou z nevýhod Cooperative A* algoritmu, je již zmíněná přidaná dimenze navíc, která značně zvyšuje výpočetní náročnost algoritmu. Rezervační tabulka, je naštěstí z podstaty řídká, jelikož počet rezervací v daném časovém okamžiku je přímo úměrný počtu robotů. Tabulku lze tedy efektivně implementovat jako hašovací tabulku, kde klíčem je daný stav (x, y, t).

Další z nevýhod je, že Cooperative A* (stejně jako žádný jiný distribuovaný hladový algoritmus) nedokáže vyřešit určité typy problémů, kdy řešení jednoho robota zamezí v možnosti dalšího robota nalézt přípustnou cestu. Jeden příklad tohoto problému můžeme vidět na obrázku 15. První robot při naplánování cesty z výchozí pozice S_1 do cíle G_1 zamezí v cestě druhému robotu, který se snaží naplánovat cestu z pozice S_2 do cíle G_2.



Obrázek 15: Příklad problému, který nelze vyřešit pomocí Cooperative A* [12]

Pro Cooperative A* lze použít libovolnou přípustnou heuristickou funkci. Při využití mřížkových map se nejčastěji používá Manhattanská metrika. Při použití na složitých mapách, kde dochází k plánování cyklických cest, je ovšem tato metrika nedostatečná.



Obrázek 16: Použití Manhattanské metriky v Cooperative A* (vlevo hodnoty f buněk, vpravo počet expanzí dané buňky) [41]

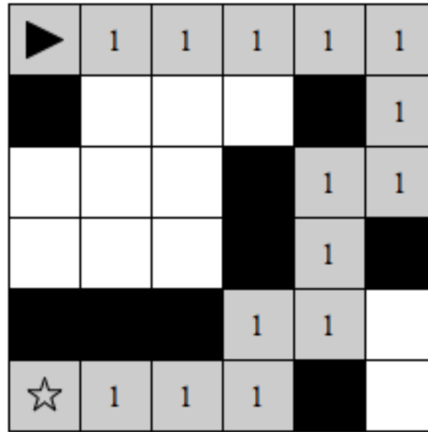
Na obrázku 16 vlevo vidíme hodnoty f buněk, které při plánování prohledá A* algoritmus. Vidíme, že algoritmus prohledal skoro všechny buňky, i ty co reálně nevedou k cíli. Při použití Cooperative A*, tedy plánování v prostoru rozšířeném o jednu dimenzi, vidíme na obrázku 16 vpravo, že tento algoritmus prohledává dané buňky několikrát, vždy v různých časech. Algoritmus se snaží vrátit zpět a prohledat buňky s lepší hodnotou f v dalších časech a tím se několikanásobně zhoršuje časová náročnost tohoto algoritmu.

2.5 Hierarchical Cooperative A*

Heuristickou funkci lze podle Silvera [12] zlepšit pomocí abstrakce stavového prostoru. Jako odhad vzdálenosti z dané buňky do cíle použijeme nejkratší možnou vzdálenost bez závislosti na čase a ostatních robotech. Jedná se tedy o délku cesty nalezenou A* algoritmem na 2D mřížce. Tuto vzdálenost nazveme *pravá vzdálenost*.

Na obrázku 17 vidíme, že při použití pravé vzdálenosti je počet prohledaných buněk minimální a robot postupuje přesně podél nejkratší cesty až do cíle. Pouze pokud při cestě narazí na další roboty, prohledá buňky navíc. Čas prohledávání je tedy přímo úměrný potřebné úrovni kooperace mezi roboty.

Pravou vzdálenost je potřebné spočítat pro každou buňku, kterou Cooperative A* prohledá. Pokud bychom pokaždé hledali cestu z dané buňky do cíle pomocí klasického A* algoritmu, bylo by celkové plánování cesty ještě pomalejší, než použití horší heuristické funkce. Jednou z možností je spočítat všechny abstraktní vzdálenosti dopředu, ale z důvodu, že se jedná o dynamický problém a tyto vzdálenosti bychom museli počítat pro



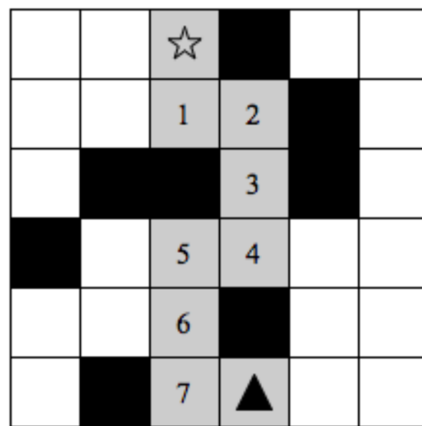
Obrázek 17: Použití pravé vzdálenosti (počet expanzí buněk) [41]

každého robota, není toto řešení adekvátní. Další možností je využití *Hierarchical A** [42], který počítá abstraktní vzdálenosti na požádání.

2.5.1 Reverse Resumable A*

Jednou z vlastností A* algoritmu je, že pokud využíváme monotónní heuristickou funkci, tak při prohledávání dané buňky (při vložení do seznamu *closed*) známe nejkratší cestu z výchozí pozice do této buňky g.

Této vlastnosti můžeme využít a hledat cestu „pozpátku“. Začneme-li v cílové pozici, tak po dokončení prohledávání (jakmile dojdeme do počáteční pozice) vidíme (jak ukazuje obrázek 18), že hodnoty g pro každou prohledanou buňku obsahují reálnou nejkratší vzdálenost do cíle.



Obrázek 18: Příklad zpětného prohledávání [41]

Po nalezení cesty pomocí zpětného prohledávání nemusíme mít expandované všechny potřebné buňky a tedy jejich g hodnoty. Naštěstí můžeme v prohledávání znovu pokračovat, dokud nenarazíme na danou buňku. Tento postup nazveme *Reverse Resumable A**, jehož pseudokód je popsán v algoritmu 5.

Cooperative A*, který jako heuristickou funkci používá pravou vzdálenost nazveme *Hierarchical Cooperative A**. Plánování cesty tedy probíhá tak, jak je popsáno v podkapitole 2.4. Pro odhad vzdálenosti voláme metodu `AbstractDist`, která ověří, jestli je daná buňka expandovaná. Pokud ano, vrátíme její g hodnotu. Pokud není, pokračujeme v hledání pomocí Reverse Resumable A* algoritmu, dokud neexpandujeme požadovanou buňku a nezískáme tak její g hodnotu.

Algoritmus 5 Reverse Resumable A*

```
 1: function InitialiseRRA*(O, G)
 2:     G.g ← 0
 3:     G.h ← h(G, O)
 4:     Open ← {G}                                  ▷ prioritní fronta (podle hodnot f)
 5:     Closed ← ∅
 6:     ResumeRRA*(O)
 7: end function
 8: function ResumeRRA*(N)
 9:     while Open ≠ ∅ do
10:         P ← Open.pop()                             ▷ prvek s nejlepší hodnotou f
11:         Closed.add(P)
12:         if P = N then
13:             return success
14:         end if
15:         for all Q ∈ Neighbors(P).reverse() do
16:             Q.g ← P.g + Cost(P,Q)
17:             Q.h ← h(Q, O)
18:             if Q ∉ Open and Q ∉ Closed then
19:                 Open.add(Q)
20:             end if
21:             if Q ∈ Open and f(Q) < f(Q in Open) then
22:                 Open.update(Q, f(Q))
23:             end if
24:         end for
25:     end while
26:     return failure
27: end function
28: function AbstractDist(N)
29:     if N ∈ Closed then
30:         return N.g
31:     end if
32:     if ResumeRRA*(N) = success then
33:         return N.g
34:     end if
35:     return ∞
36: end function
```

2.6 Windowed Hierarchical Cooperative A*

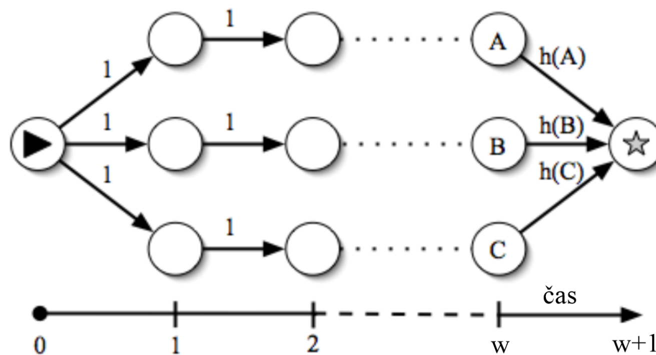
Jelikož je plánování v rozšířeném prostoru náročné na výpočetní čas i paměť, je vhodné toto prohledávání omezit. Silver [12] navrhuje určit horní hranici potřebné kooperace, kterou nazveme *okno* w. Kooperativní hledání je omezeno do hloubky dané touto hodnotou. Každý robot hledá neúplnou cestu do svého cíle, po které se potom začne pohybovat. V pravidelných intervalech se dané okno posune a hledá se nová (neúplná) cesta.

Jelikož by následování neúplné cesty nemuselo vézt správným směrem, případně by se robot mohl dostat do slepé uličky, je místo heuristické funkce použitá opět pravá vzdálenost (viz 2.5). Určování této vzdálenosti není omezeno hodnotou w. Jedná se tedy o abstrakci, kdy robot zvažuje přítomnost ostatních robotů pouze do vzdálenosti w, dále o nich neuvažuje.

Této vlastnosti je dosaženo pomocí takzvané *koncové hrany*, která vede z každého uzlu vzdáleného w přímo do uzlu cílového s cenou rovnou pravé vzdálenosti tohoto uzlu. Příklad můžeme vidět na obrázku 19. Navíc je možné pokračovat v plánování i po dosažení cílového stavu. Toto je například žádoucí, pokud je cílový stav v úzkém koridoru a daný robot by tak blokoval cestu ostatním. Obecně je tedy cena hrany mezi dvěma stavy P a Q dána pomocí rovnice

$$cost\,(P,Q) = \begin{cases} 0 \quad \text{pro } P = Q = G \text{ a } t < w, \\ \text{AbstractDist}\,(P,G) \quad \text{pro } t = w, \\ 1 \quad \text{jinak}, \end{cases} \tag{12}$$

kde G je cílový stav.



Obrázek 19: Příklad koncových hran (hran ze stavů A, B, C do cíle) [41] (přeloženo)

Výsledky hledání pravé vzdálenosti pomocí Reverse Resumable A* lze jednoduše použít i když se roboti pohybují. Aby tyto vzdálenosti zůstaly monotónní je potřeba pokračovat v hledání směrem k původní pozici robota, ne k té současné. Celková efektivita tohoto přístupu je závislá na velikosti odchylky robota od nejkratší cesty.

3 Popis aplikace

Následující kapitola se bude věnovat popisu aplikace *MultiRobotSimulator*, který je výstupem praktické části této diplomové práce. Jedná se o grafické prostředí umožnující vytvářet, importovat a editovat mapy prostředí, spouštět a vyhodnocovat výsledky daných algoritmů a pomocí plug-in systému jednoduše implementovat další algoritmy pro plánování cesty více robotů.

3.1 Použité technologie

Programovací jazyk C#

C# (vysl. *see sharp*) je moderní mnohoúčelový vysokoúrovňový objektově orientovaný programovací jazyk vyvíjený od roku 2000 firmou Microsoft zároveň s platformou *.NET Framework*. Nabízí podporu pro principy softwarového inženýrství, jakými jsou např. silné typování, hlídání hranic polí, detekce použití neinicializovaných proměnných nebo automatický garbage collector. Je vhodný pro vývoj softwarových komponent vhodných pro distribuované prostředí, jak pro velká zařízení se sofistikovanými operačními systémy, tak pro malá zařízení s omezenými funkcemi. Tento programovací jazyk byl schválen normalizační organizací ECMA [43]. Konkrétně se jedná o multiplatformní open-source platformu *.NET Core* verze 3.0. K vývoji bylo použito vývojové prostředí *Visual Studio 2019*, podporující např. IntelliSense, refaktorování kódu, statickou analýzu nebo editor grafických rozhraní [44].

Windows Presentation Foundation

Pro design grafického uživatelského rozhraní (GUI) byla zvolena technologie *Windows Presentation Foundation* (WPF) [44], která je součástí .NET Framework. Jedná se o moderního nástupce sady knihoven *Windows Forms*. Základem je renderovací engine založený na vektorové grafice využívající moderních zobrazovacích zařízení, podporující hardwarovou akceleraci a technologii *DirectX*. Pomocí datových vazeb umožňuje jednoduše oddělit vzhled od funkční logiky programu.

Stylet

Pro zjednodušení práce s architekturou Model-View-ViewModel, která je základem WPF, byla zvolena knihovna *Stylet* [45], která umožňuje jednoduchou údržbu a testovatelnost kódu. Navíc obsahuje poskytovatele závislostí, tzv. IoC kontejner, jehož bylo využito při implementaci plug-in systému (podkapitola 3.3).

QuickGraph

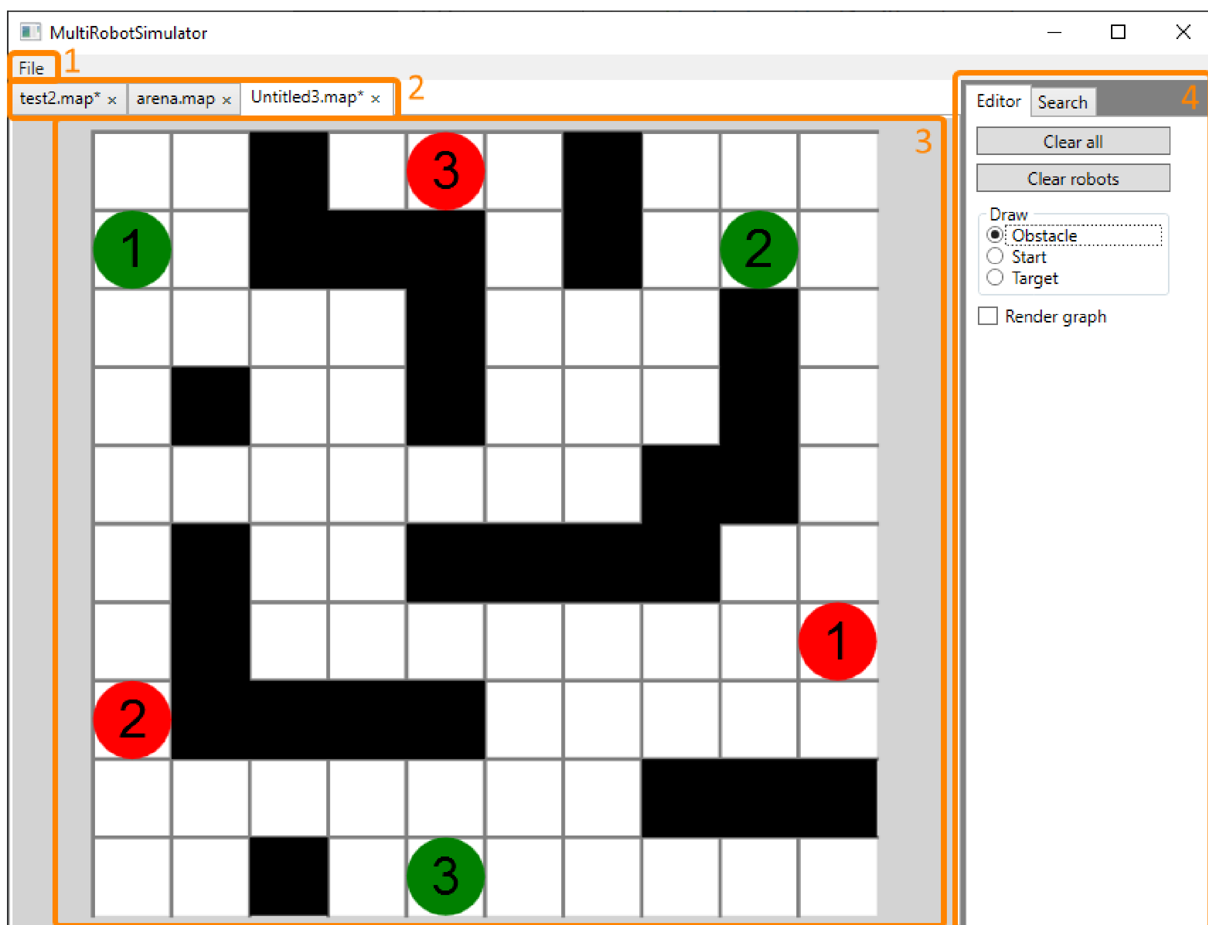
Pro reprezentaci prostředí jako grafu byla použita knihovna *QuickGraph* [46], která obsahuje generické datové struktury reprezentující neorientované a orientované grafy i některé algoritmy použitelné s těmito strukturami.

High Speed Priority Queue

Jelikož uvedené algoritmy využívají prioritní frontu, bylo použito knihovny *High Speed Priority Queue* [47], která je implementovaná speciálně pro plánování cest. Tato knihovna nabízí testovanou, rychlou, generickou prioritní frontu bez závislostí na jiných knihovnách.

3.2 Uživatelské rozhraní

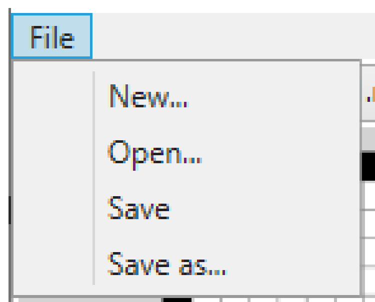
Cílem uživatelského rozhraní je umožnit uživatelům snadné a intuitivní používání programu. Na obrázku 20 můžeme vidět hlavní okno aplikace a její popsané části.



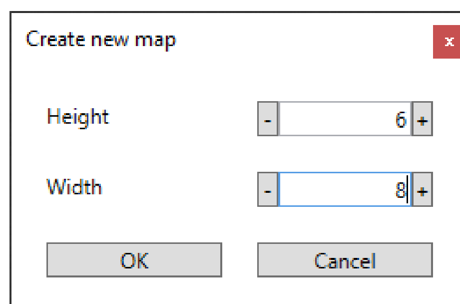
Obrázek 20: Hlavní okno aplikace

1 – Hlavní menu

Hlavní menu (viz obrázek 21) umožnuje uživateli vytvořit novou mapu, případně ji načíst ze souboru (formát popsán v podkapitole 3.4) nebo uložit. Při výběru možnosti vytvoření nové mapy se otevře dialog s nabídkou specifikovat požadovanou výšku a šířku (obrázek 22). Otevírání a ukládání map je zprostředkováno standardním dialogovým oknem operačního systému.



Obrázek 21: Hlavní menu



Obrázek 22: Dialog pro vytvoření nové mapy

2 – Záložky

Aplikace umožňuje mít otevřeno několik map zároveň a přepínat mezi nimi pomocí záložek. Všechny funkce aplikace probíhají na aktuálně zvolené mapě.

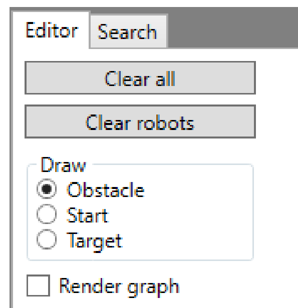
3 – Mapa prostředí

Prostředí je v aplikaci představováno sítí čtverců, kde bílé čtverce představují přípustné stavy a černé čtverce reprezentují překážky (nepřípustné stavy). Výchozí stavy robotů jsou zobrazeny jako zelené kruhy a cílové stavy jako kruhy červené. Stavy se stejnými čísly přísluší stejnému robotu. Stavy lze přidávat kliknutím levým tlačítkem myši v závislosti na vybraném módu na postranním panelu (4), pravým tlačítkem daný stav smažeme.

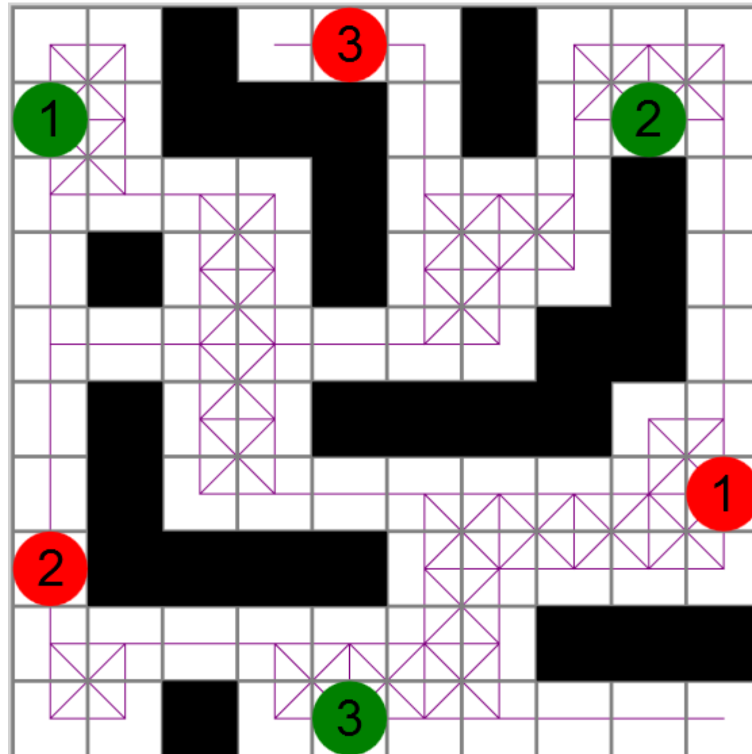
4 – Postranní panel

Postranní panel obsahuje dvě záložky:

- Editor (obrázek 23) – Obsahuje možnosti týkající se editoru mapy prostředí. Tlačítko *Clear all* smaže z mapy všechny překážky a výchozí a cílové stavy robotů. Tlačítko *Clear robots* smaže pouze výchozí a cílové stavy, překážky ponechá. Přepínačem *Draw* je možné vybrat mód kreslení do mapy. Možnost *Render graph* zobrazí v mapě hrany grafu, který reprezentuje toto prostředí. Toto zobrazení můžeme vidět na obrázku 24

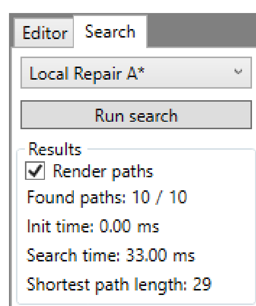


Obrázek 23: Postranní panel – záložka Editor



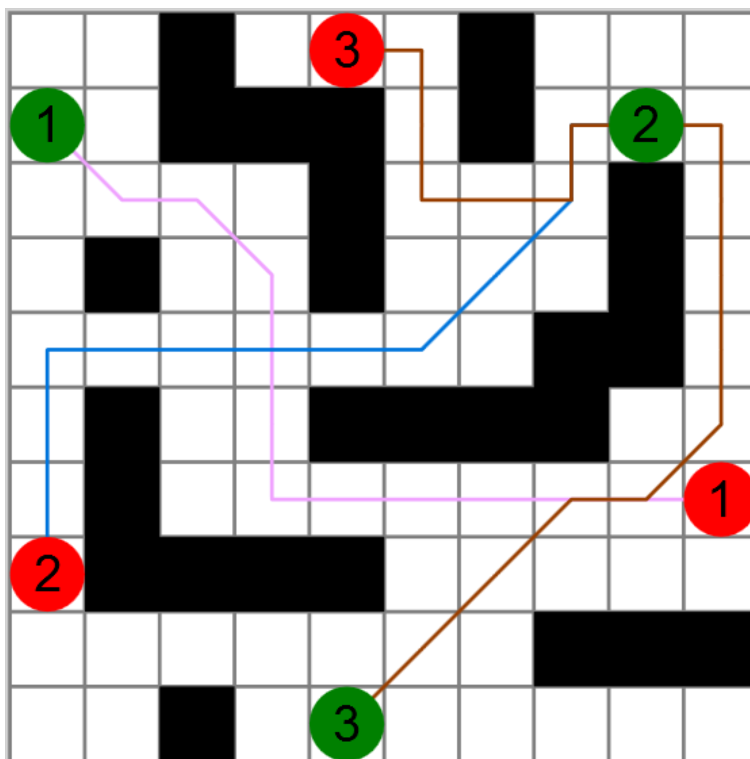
Obrázek 24: Zobrazení grafu v mapě (fialová)

- Search (obrázek 25) – V této záložce si může uživatel vybrat ve výběrovníku požadovaný algoritmus. Tlačítkem *Run search* se spustí hledání pomocí vybraného algoritmu. Po dokončení hledání se pod tímto tlačítkem zobrazí výsledky s těmito informacemi: počet úspěšně nalezených cest, čas inicializace algoritmu, čas potřebný k hledání a délka nejkratší cesty.



Obrázek 25: Postranní panel – záložka Search

Po dokončení hledání jsou v mapě zobrazeny nalezené cesty, odlišené barevně pro lepší přehlednost. Příklad můžeme vidět na obrázku 26.



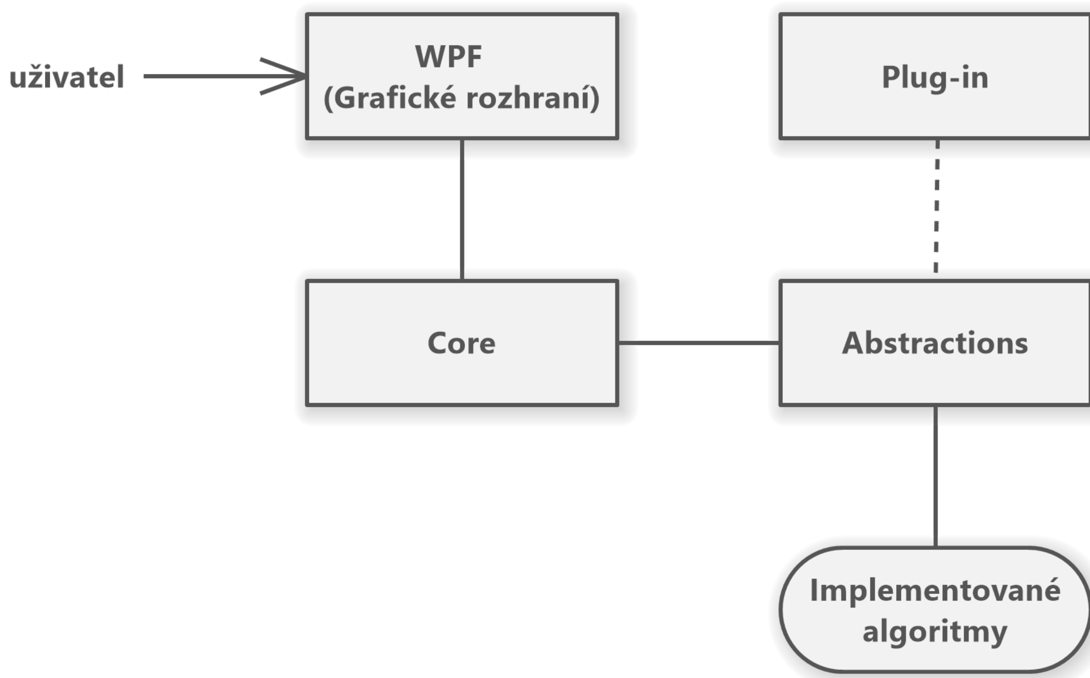
Obrázek 26: Zobrazení nalezených cest

3.3 Plug-in systém

Aplikace podporuje programování plánovacích algoritmů nezávisle na ostatních funkcích programu. Tohoto je dosaženo pomocí vrstvy abstrakce, jejíž schéma můžeme vidět na obrázku 27. Uživatel ovládá aplikaci pomocí grafického rozhraní, které je obsaženo v projektu `WPF`. Programová logika je obsažena v projektu `Core`, která pomocí veřejných rozhraní (angl. interface) a abstraktních tříd provádí plánování cest robotů.

Uživatel může naprogramovat vlastní algoritmus v jazyce C#, který aplikace při startu naimportuje a je možné jej použít stejně jako zabudované algoritmy. Pro implementaci algoritmu je nutné vytvořit třídu, která dědí z abstraktní třídy `AbstractAlgo`, případně rozhraní `IAlgo`. Vlastnosti a funkce této třídy jsou následující:

- `Graph` – graf reprezentující danou mapu prostředí
- `Name` – název algoritmu
- `Robots` – seznam robotů s danými vlastnostmi (start, cíl, nalezená cesta atd.)
- `Initialize()` – funkce volaná na začátku hledání



Obrázek 27: Schéma architektury

- `RobotFactory(start, target)` – funkce vytvářející instanci robota, umožňující její rozšíření uživatelem
- `RunSearch()` – funkce obsahující vlastní implementaci plánovacího algoritmu

Sestavenou knihovnu `.dll` musí uživatel před spuštěním aplikace umístit do složky `plugins`, ve které se nachází i textový soubor `readme.md` s bližšími informacemi jak postupovat při implementaci.

3.4 MovingAI benchmarks

Aplikace podporuje otevírání a ukládání map ve formátu, který byl zaveden Sturtevantem [48] v jeho článku *Benchmarks for Grid-Based Pathfinding*. Jedná se o sadu mřížkových map, která je k dispozici online (`https://movingai.com/benchmarks/`). Tato sada obsahuje mapy z komerčních počítačových her (např. Baldurs Gate II, Warcraft III a další), mapy z městských prostředí (Berlín, Londýn, ...), mapy bludišť a další náhodně vygenerované mapy. Tyto sady jsou běžně používané ve vědeckých výzkumech pro srovnávání výsledků jednotlivých algoritmů. Příklady těchto map můžeme vidět na obrázcích 28 a 29.

Obrázek 28: Mapa AR0300SR ze hry Baldurs Gate II [48]

Obrázek 29: Mapa London_2_512 [48]

4 Vyhodnocení výsledků

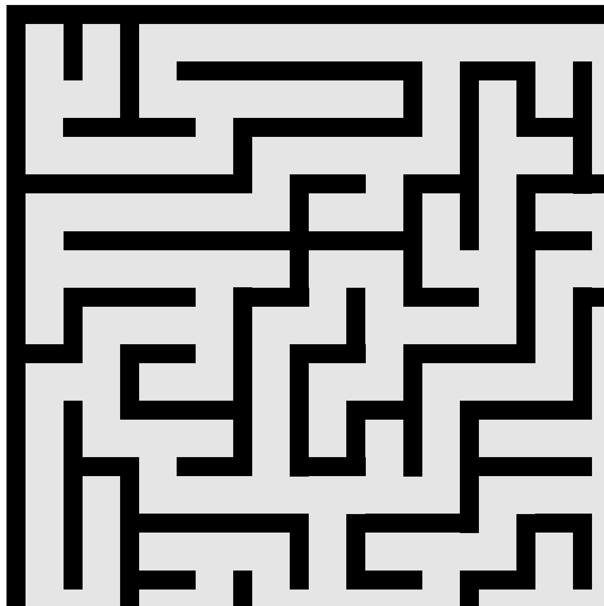
Ve vytvořeném simulačním prostředí byly provedeny 4 experimenty, každý na jiné mapě ze sady MovingAI [48] zmíněné v podkapitole 3.4. Na každé z map byly otestovány všechny uvedené algoritmy tak, že se postupně přidávalo po jednom robotu (s náhodnými souřadnicemi počátečního i koncového stavu) až do počtu 60.

Ceny přechodů mezi přilehlými buňkami byly nastaveny na 1, cena přechodu v diagonálním směru byly nastavena na $\sqrt{2}$. Jako heuristická funkce h byla zvolena metrika octile. V každém experimentu byl posouzen počet správně nalezených cest, celkový čas potřebný k jejich nalezení, průměrný počet expandovaných uzlů a průměrná odchylka od optimálního řešení. U algoritmů HCA* a WHCA*, které využívají prohledávání v abstraktním stavovém prostoru, byly zahrnuty i uzly expandované algoritmem RRA*. Pro algoritmus WHCA* byla zvolena hodnota okna $w = 16$. Maximální délka cesty byla omezena na 1000 uzlů (ochrana proti zacyklení). Hodnoty expandovaných uzlů a odchylek od optimální délky cesty jsou počítány pouze pro roboty, kteří úspěšně nalezli cestu.

Experimenty byly provedeny na počítači s procesorem Intel(R) Core(TM) i3-6100 @ 3.70GHz a operační pamětí 16 GB @ 2133 MHz.

Experiment 1

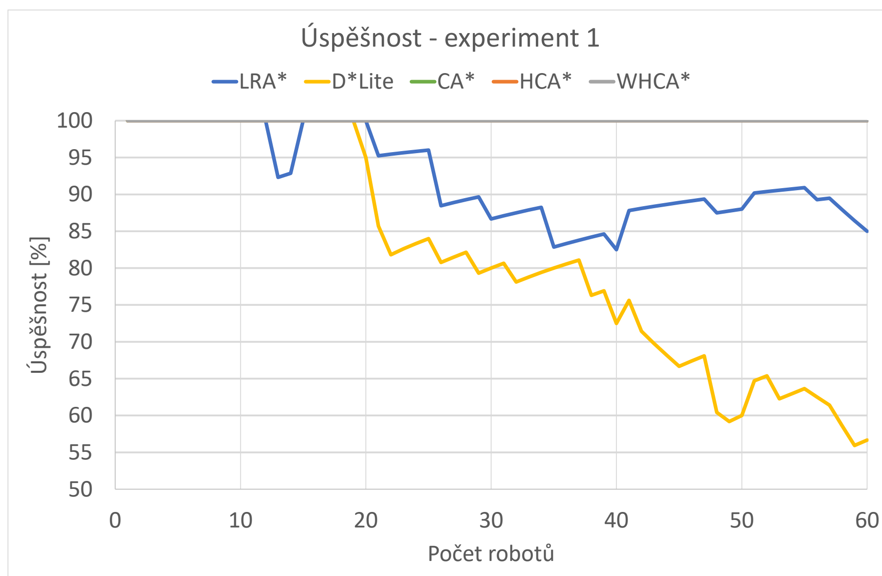
První experiment byl proveden na mapě `maze-32-32-2` (obrázek 30). Jedná se o menší mapu bludiště velikosti 32×32.



Obrázek 30: Mapa `maze-32-32-2` [48]

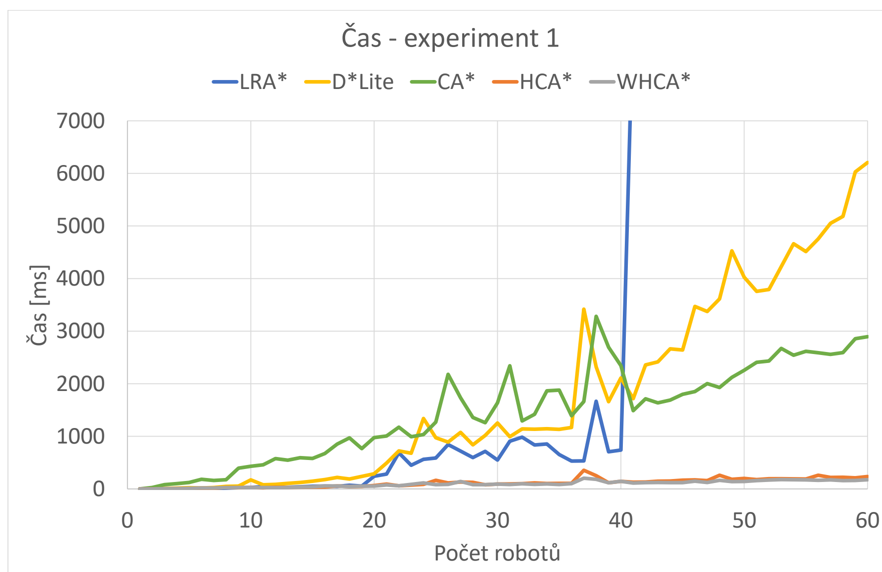
Na obrázku 31 vidíme úspěšnost nalezení cesty pro všechny roboty. S rostoucím počtem robotů se LRA* a MA D* Lite zhoršují, kde LRA* neklesne pod 80% úspěšnost,

zatímco MA D* Lite postupně klesá až k 56% při 59 robotech. K neúspěchu dochází z důvodu relativně úzkých koridorů, ve kterých se roboti bez kooperace nedokáží vyhnout. Kooperativní algoritmy jsou schopné nalézt všechny cesty ve všech případech.



Obrázek 31: Graf úspěšnosti nalezení cesty při prvním experimentu

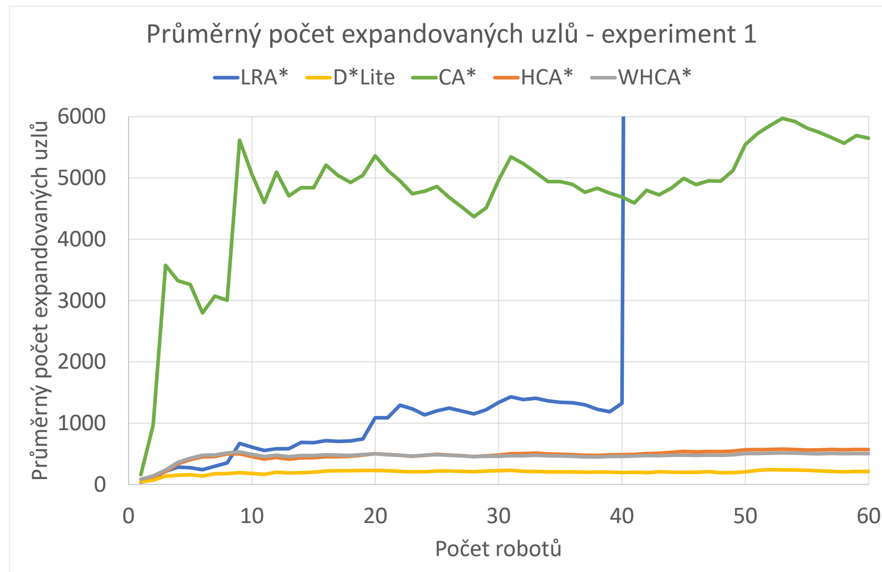
Na obrázku 32 můžeme pozorovat čas potřebný k plánování cest. U LRA* dochází při překročení hranice robotů k zacyklení, které musí být ošetřeno vnější podmínkou (zde zvolena maximální délka cesty). MA D* Lite a CA* mají také s větším počtem robotů problémy a jejich časy postupně rostou. HCA* se nedostane přes hranici 300 ms, WHCA dokonce nepřesáhne 200 ms.



Obrázek 32: Graf časové závislosti jednotlivých algoritmů při experimentu 1

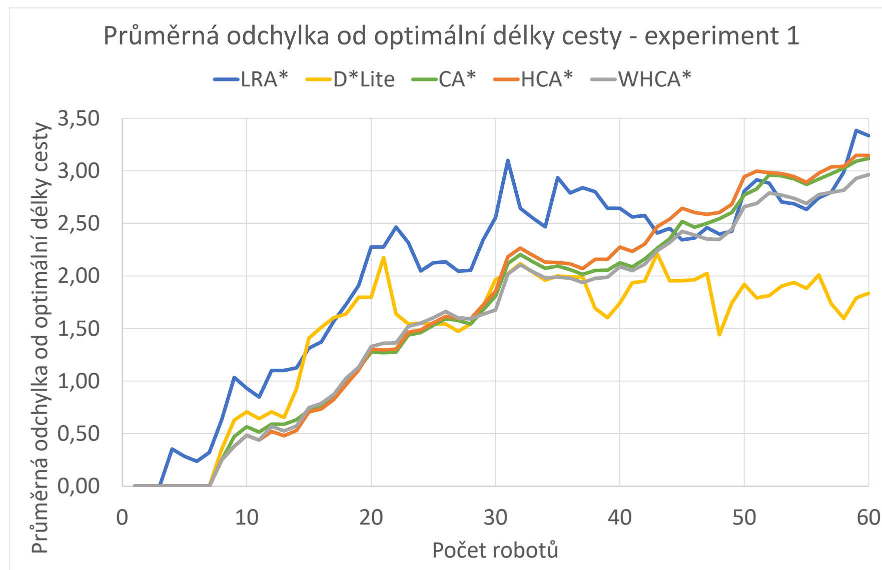
Obrázek 33 ukazuje průměrný počet expandovaných uzlů. Jak bylo zmíněno výše, od hranice 40 robotů dochází u LRA* k zacyklení, tedy obrovskému nárůstu prohledaných

stavů. Algoritmus CA* v porovnání s ostatními prohledává obrovské množství uzlů, jelikož dochází k prohledávání i těch uzlů, které nevedou k cíli, jak bylo zmíněno v podkapitole 2.4. Algoritmy HCA* a WHCA* prohledávají oba přibližně stejný počet uzlů, zatímco MA D* Lite jich expanduje méně než polovinu.



Obrázek 33: Průměrný počet expandovaných uzlů v experimentu 1

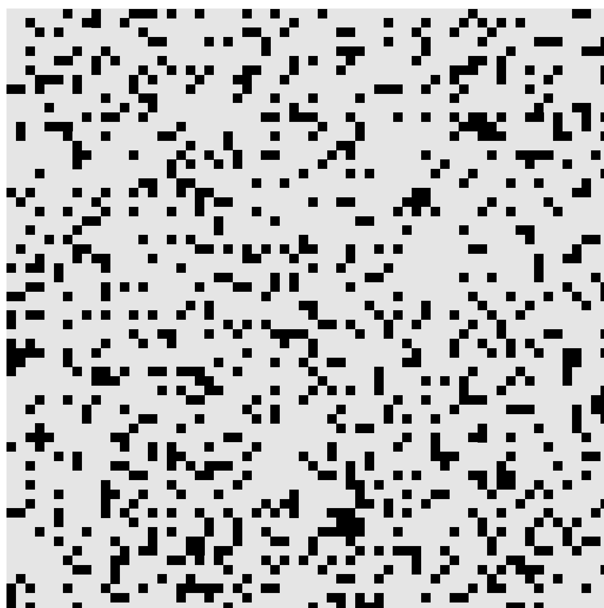
Na obrázku 34 vidíme, že s rostoucím počtem robotů roste i průměrná odchylka od optimální délky cesty každého robota. Toto je samozřejmě způsobeno nutností vyhýbat se kolizím. Vidíme, že u LRA* a MA D* Lite dochází rychleji k nárůstu této odchylky, protože roboti nekooperují a jsou nuceni častěji přeplánovat svoji trasu mimo tu optimální. Při větším počtu robotů tyto hodnoty přestávají růst tak dramaticky, toto je ale pravděpodobně způsobeno celkově nižším počtem nalezených cest.



Obrázek 34: Průměrná odchylka od optimální délky cesty při prvním experimentu

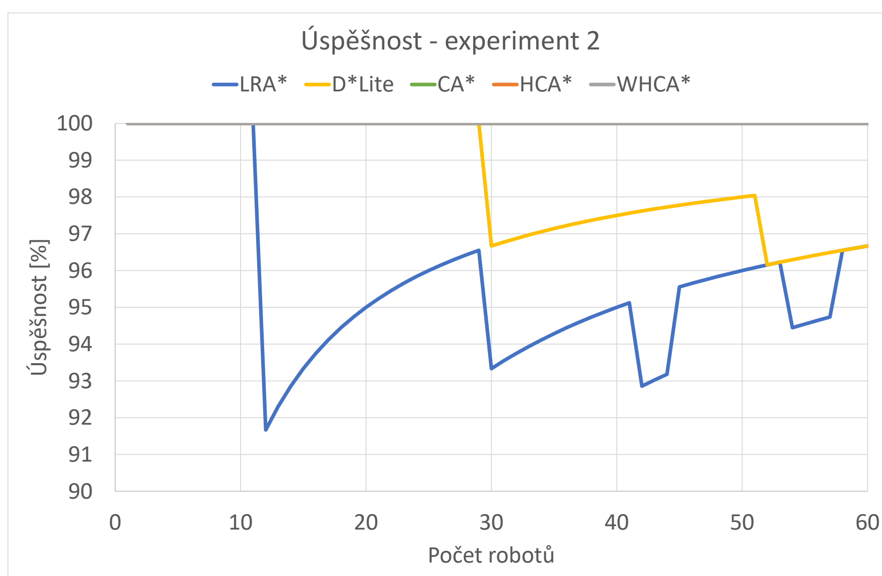
Experiment 2

V druhém experimentu se jedná o mapu `random-64-64-20` (obrázek 35) o velikosti 64×64, kde 20 % tvoří náhodně vygenerované překážky.



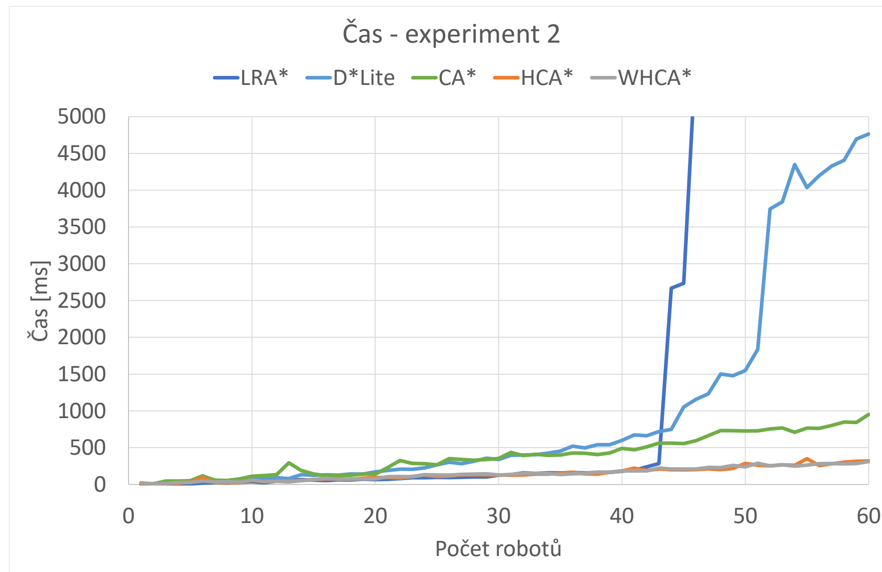
Obrázek 35: Mapa `random-64-64-20` [48]

V grafu úspěšnosti druhého experimentu (obrázek 36) vidíme, že algoritmy LRA* a MA D* Lite nejsou opět schopné nalézt cestu pro všechny roboty, oba však neklesnou pod hranici 95 %. Kooperativní algoritmy jsou schopné nalézt cestu v každém případě.



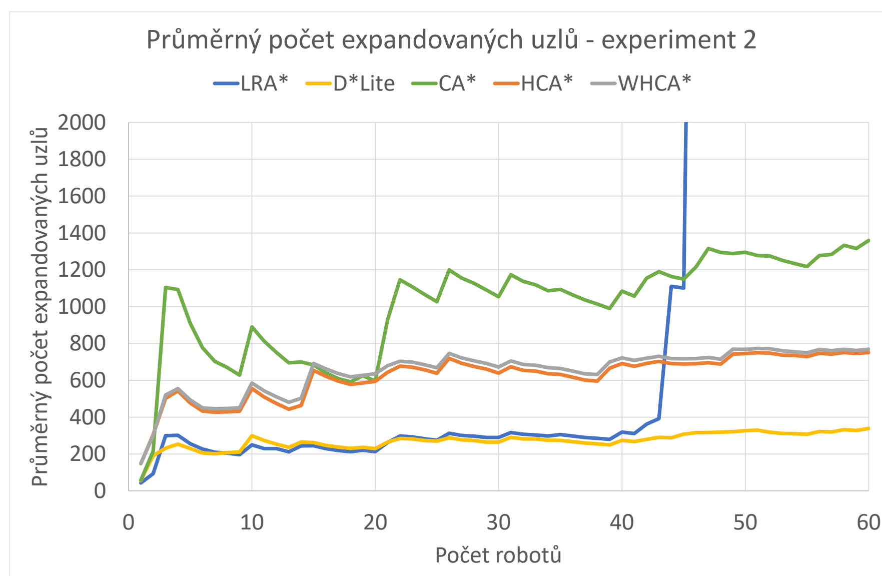
Obrázek 36: Graf úspěšnosti nalezení cesty při druhém experimentu

Na grafu časové závislosti (obrázek 37) vidíme, že opět při překročení hranice okolo 40 robotů dojde u algoritmů LRA* a MA D* Lite k zacyklení, tudíž i k obrovskému nárůstu času potřebného k hledání cesty. Algoritmus CA* potřebuje přibližně trojnásobný čas k nalezení cesty než algoritmy HCA* a WHCA*.



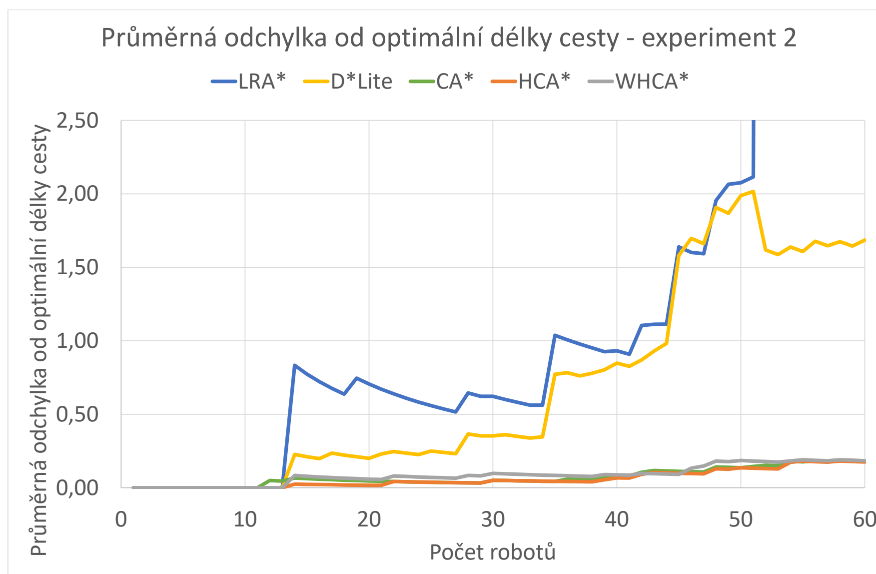
Obrázek 37: Graf časové závislosti jednotlivých algoritmů při experimentu 2

Obrázek 38 ukazuje průměrný počet expandovaných uzlů na jednoho robota. Vidíme, že algoritmy LRA* a MA D* Lite expandují přibližně stejný počet uzlů, ale při zacyklení dochází v algoritmu LRA* k opětovnému prohledávání, tudíž k velkému nárůstu expandovaných stavů. Algortimy HCA* a WHCA* expandují přibližně dvojnásobný počet uzlů než MA D* Lite a algoritmus CA* přibližně čtyřnásobek.



Obrázek 38: Průměrný počet expandovaných uzlů v experimentu 2

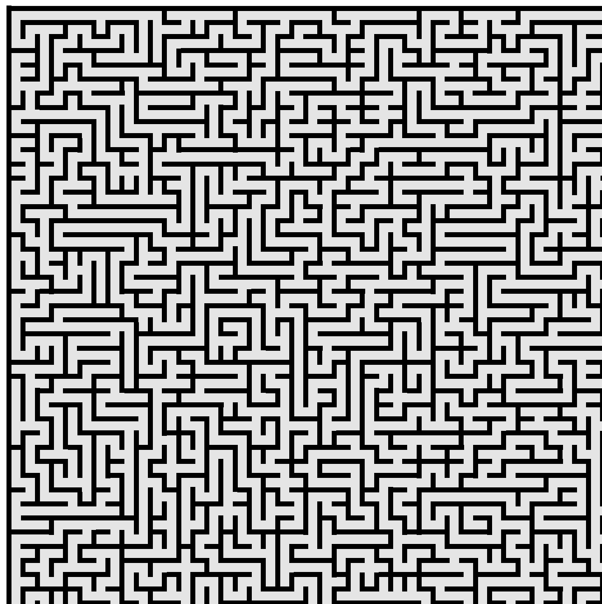
Zatímco kooperativní algoritmy dokáží nalézt cesty s velmi malými odchylkami od optimální délky, u algoritmů LRA* a MA D* Lite při vzrůstajícím počtu robotů narůstá průměrná délka cesty z důvodu vyhýbání se kolizím. Závislost průměrné odchylky od optimální délky trasy můžeme vidět na obrázku 39.



Obrázek 39: Průměrná odchylka od optimální délky cesty při druhém experimentu

Experiment 3

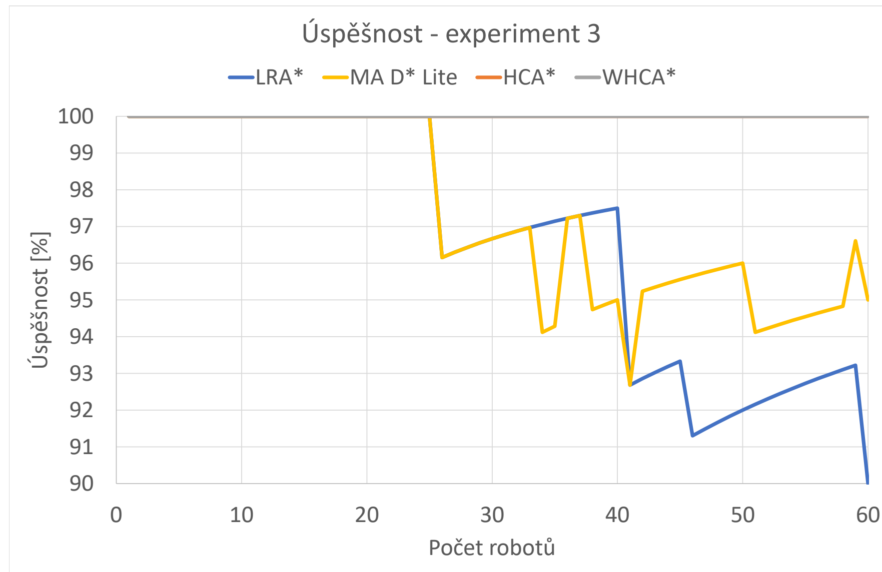
Jako třetí byla zvolena mapa `maze-128-128-2` (obrázek 40), která představuje složitější bludiště o rozměrech 128×128.



Obrázek 40: Mapa `maze-128-128-2` [48]

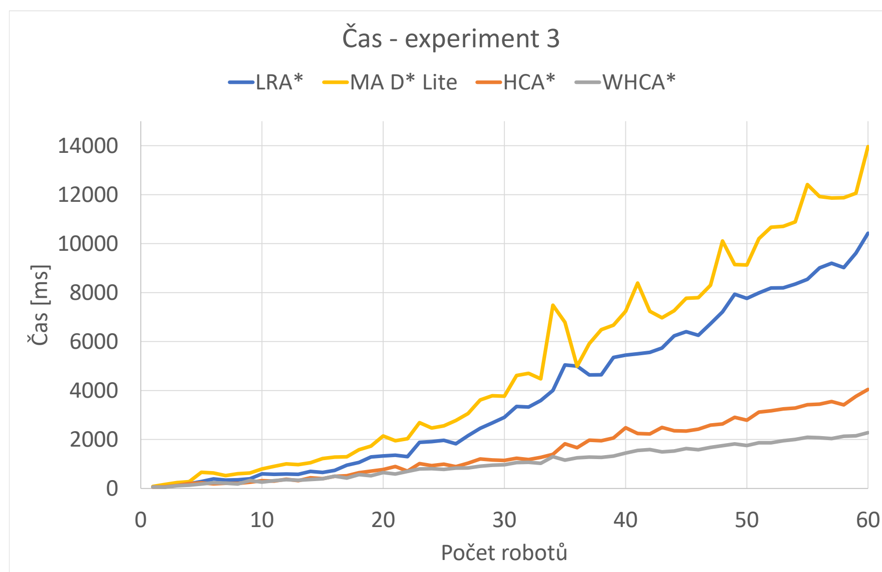
Vzhledem k povaze mapy byl algoritmus CA* naprosto nepoužitelný už při hledání cesty 1 robota. V experimentu byl tedy vynechán. Na grafu úspěšnosti na obrázku 41

vidíme, že algoritmy LRA* a MA D* Lite opět nenaleznou cestu pro úplně všechny roboty. Úspěšnost těchto algoritmů však neklesne pod 90 %.



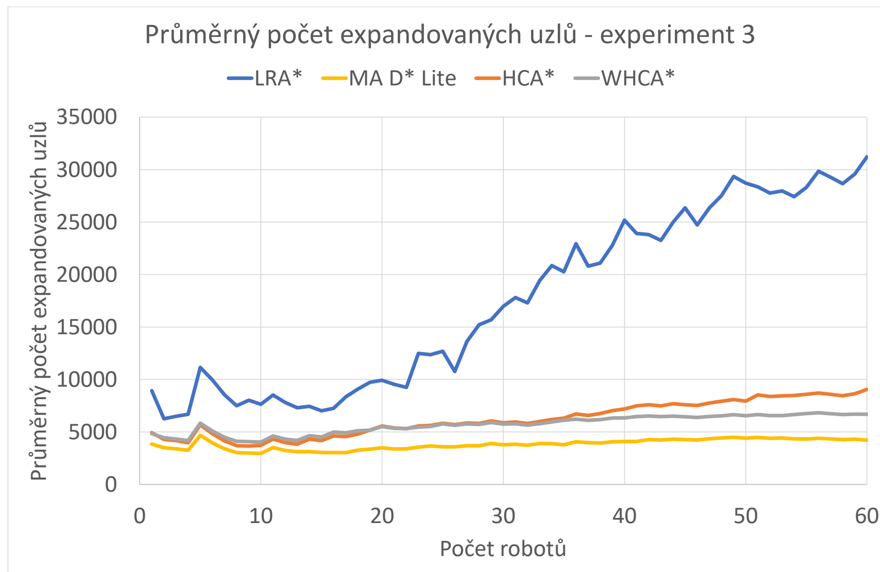
Obrázek 41: Graf úspěšnosti nalezení cesty při třetím experimentu

Obrázek 42 představuje graf závislosti času řešení na počtu robotů. Tentokrát nedochází k zacyklení, přesto platí, že algoritmy LRA* a MA D* Lite jsou výrazně pomalejší než HCA* a WHCA*.



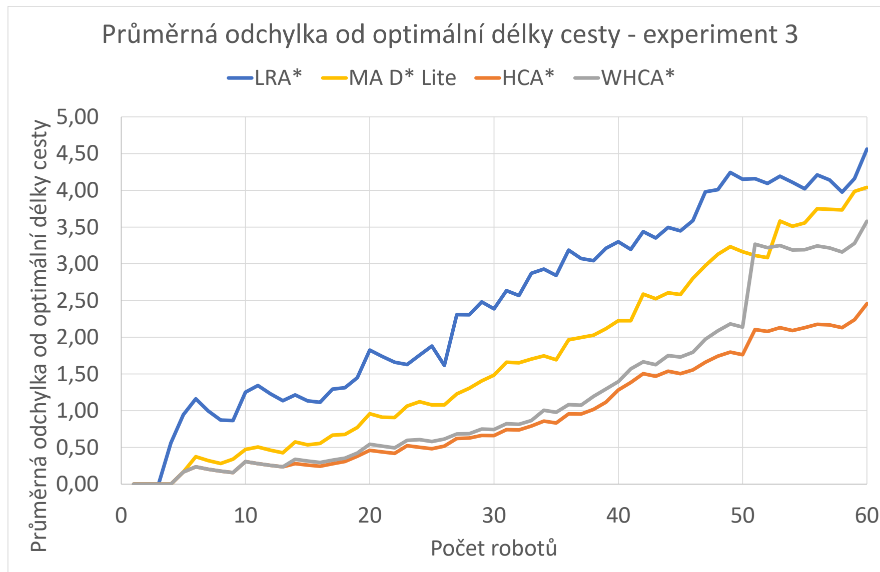
Obrázek 42: Graf časové závislosti jednotlivých algoritmů při experimentu 3

Na obrázku 43 vidíme, že zatímco algoritmus LRA* expanduje obrovský počet uzlů, MA D* Lite potřebuje expandovat uzlů nejméně a algoritmy WHCA* a HCA* přibližně 1,5× a 2× tolik.



Obrázek 43: Průměrný počet expandovaných uzlů v experimentu 3

Z grafu závislosti průměrné odchylky od optimální délky cesty na obrázku 44 vidíme, že algoritmus LRA* opět dosahuje největších odchylek od optimální trasy. Tentokrát ovšem ani kooperativní algoritmy HCA* a WHCA* nedokáží přesně následovat nejkratší cestu.



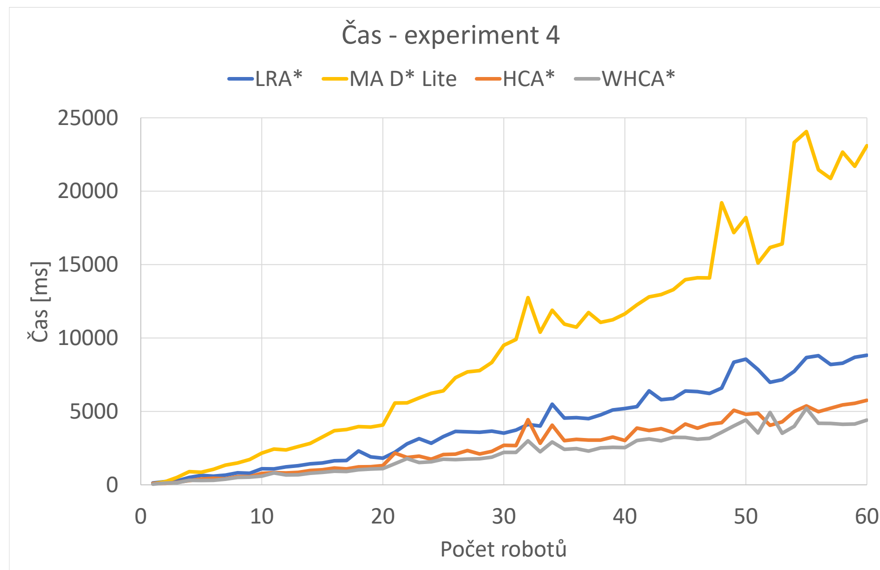
Obrázek 44: Průměrná odchylka od optimální délky cesty při třetím experimentu

Experiment 4

Čtvrtý experiment probíhal na mapě `Boston_0_256` (obrázek 45), diskretizované části mapy města Boston o rozměrech 256×256.

Obrázek 45: Mapa `Boston_0_256` [48]

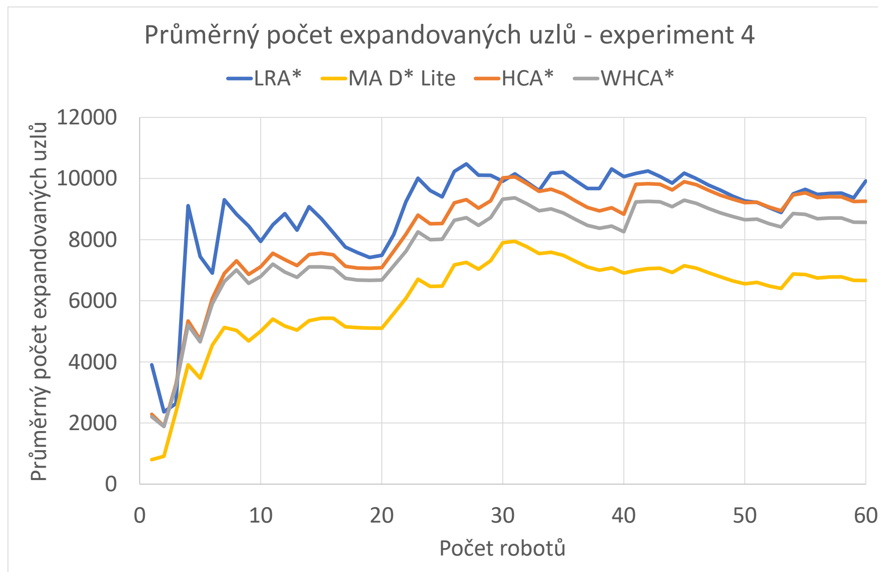
Jelikož je mapa poměrně rozsáhlá, je algoritmus CA* prakticky nepoužitelný. Již pro 4 roboty vzrostl čas potřebný pro plánování na 49 sekund. V experimentu byl z tohoto důvodu vynechán. Vzhledem k velikosti mapy a velkého počtu přípustných stavů nedocházelo v tomto experimentu k ucpávkám, které by znemožnili robotům nalézt cestu. Ve všech případech dokázali roboti nalézt 100 % požadovaných cest. Velký počet stavů se samozřejmě promítl na časové obtížnosti algoritmů, viz obrázek 46. Algoritmus LRA* potřeboval téměř dvojnásobek času než WHCA*, algoritmus MA D* Lite byl dokonce až 5krát pomalejší.



Obrázek 46: Graf časové závislosti jednotlivých algoritmů při experimentu 4

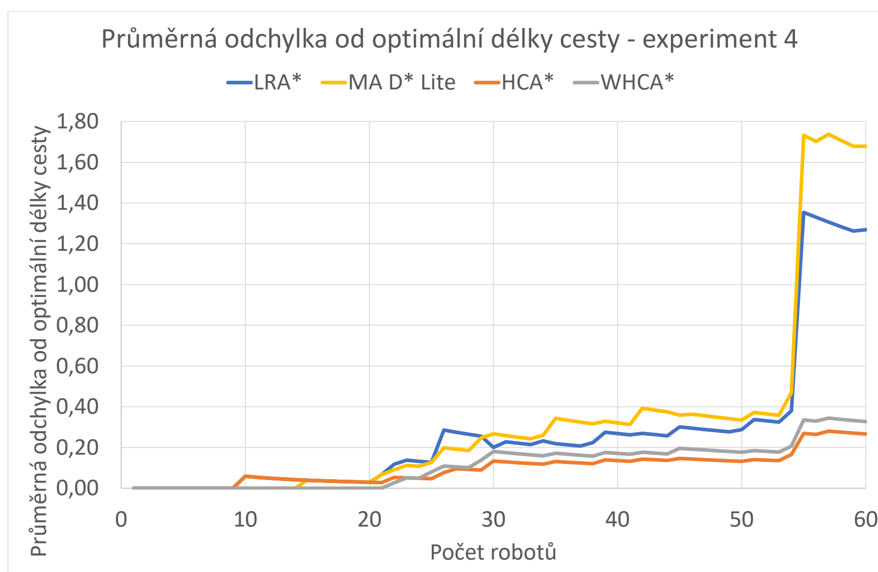
Na obrázku 47 vidíme, že algoritmus MA D* Lite je opět v počtu expandovaných uzlů nejúspornější. Jelikož na rozsáhlé mapě nedocházelo tak často ke kolizím, je algor-

timus LRA* v počtu expandovaných uzlů srovnatelný s kooperativními algoritmy HCA* a WHCA*.



Obrázek 47: Průměrný počet expandovaných uzlů v experimentu 4

Velikost mapy ovlivnila i průměrnou odchylku nalezených cest od optimálních délek. Jen zřídka se museli roboti vyhýbat ostatním, tudíž nedocházelo k častému odklonu od optimální cesty. Jen při počtu robotů větším než 54 docházelo k relativně větším odchylkám, kdy kooperativní algoritmy HCA* a WHCA* dosahovaly odchylek nižších, než algoritmy LRA* a MA D* Lite.



Obrázek 48: Průměrná odchylka od optimální délky cesty při čtvrtém experimentu

Závěr

Cílem této práce bylo popsat problematiku plánování cesty více robotů. První část práce se věnovala popisu navigace robota a jejích součástí – mapování, tj. zpracování a reprezentaci prostředí, a plánování cesty, tj. konkrétním metodám. Další část se věnovala popisu vybraných algoritmů používaných pro plánování cesty více robotů ve statickém prostředí. Praktická část práce spočívala v implementaci simulačního prostředí, ve kterém byly poté dané algoritmy otestovány ve čtyřech různých prostředích.

K implementaci bylo vybráno 5 algoritmů – Local Repair A*, vlastní implementace D* Lite pro více robotů nazvaná Multi Agent D* Lite, Cooperative A* a jeho varianty Hierarchical Cooperative A* a Windowed Hierarchical A*.

Navržené simulační prostředí aplikace umožňuje vytváření, ukládání a načítání map vlastních, případně map ze sady benchmarků MovingAI. Navíc aplikace podporuje jednoduchou implementaci dalších algoritmů pomocí plug-in systému, kdy není potřeba zasahovat do aplikační logiky aplikace. Tohoto lze využít např. ve výuce nebo dalším výzkumu na toto téma.

Algoritmy byly srovnány ve čtyřech experimentech, které se lišily jak velikostí, tak i topologií mapy. V každém experimentu byl postupně navyšován počet robotů od 1 do 60 a porovnávány následující informace: úspěšnost nalezení cesty všemi roboty, čas běhu algoritmu, průměrný počet expandovaných uzlů na jednoho robota a průměrná odchylka od optimální délky nalezené cesty.

Algoritmus Local Repair A* je jednoduchý na implementaci, avšak při vyšším počtu robotů dochází k častému přeplánování, tudíž k navýšení počtu expandovaných uzlů a času potřebného pro běh algoritmu. Navíc dochází k selhání nalezení cest pro všechny roboty. Algoritmus Multi Agent D* Lite byl navržen pro snížení počtu expandovaných uzlů při přeplánování. V tomto je algoritmus úspěšný, ve všech experimentech dosáhl nejmenšího počtu expandovaných uzlů ze všech algoritmů. Stejně jako u Local Repair A* však může dojít k selhání nalezení cesty pro všechny roboty. Navíc jeho základem je nalezení hrany grafu takové, která minimalizuje danou funkci, která je ovšem v této implementaci problematická, tudíž algoritmus nedosahuje takových časů jako ostatní algoritmy. V budoucnosti by se tento problém mohl vyřešit lepší implementací nalezení dané hrany, případně např. paralelizací výpočtu. Cooperative A* se ukázal jako nepraktický, protože expanduje velký počet zbytečných uzlů, což při složitých velkých mapách vede k obrovskému nárůstu výpočetního času. Hierarchical Cooperative A* řeší tento problém výpočtem odhadu vzdáleností v abstraktním prostoru pomocí Reverse Resumable A* algoritmu. Windowed Hierarchical Cooperative A* dále rozšiřuje abstrakci o zavedení tzv. okna, ve kterém uvažuje ostatní roboty a dále už ne. Oba tyto algoritmy obstály ve všech

experimentech, kdy časová náročnost i odchylky od optimální délky cesty byly nejmenší ze všech algoritmů, avšak za cenu většího počtu expandovaných uzlů.

Literatura

[1] MEYER, Jean-Arcady a David FILLIAT. Map-based navigation in mobile robots. In: *Cognitive Systems Research* [online]. 2003, s. 283-317 [cit. 2020-06-18]. DOI: 10.1016/S1389-0417(03)00007-X. ISSN 13890417. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S138904170300007X

[2] STACHNISS, Cyrill. *Robotic Mapping and Exploration* [online]. Berlin: Heidelberg: Springer Berlin Heidelberg, 2009 [cit. 2020-06-21]. DOI: 10.1007/978-3-642-01097-2. ISBN 978-3-642-01096-5. Dostupné z: https://www.researchgate.net/publication/220688303

[3] KOUBAA, Anis, Hachemi BENNACEUR, Imen CHAARI, et al. Introduction to Mobile Robot Path Planning. In: *Robot Path Planning and Cooperation* [online]. Cham: Springer International Publishing, 2018, 2018-04-06, s. 3-12 [cit. 2020-06-21]. Studies in Computational Intelligence. DOI: 10.1007/978-3-319-77042-0_1. ISBN 978-3-319-77040-6. Dostupné z: http://link.springer.com/10.1007/978-3-319-77042-0_1

[4] LAVALLE, Steven Michael. *Planning algorithms* [online]. New York: Cambridge University Press, 2006 [cit. 2020-06-21]. ISBN 9780521862059. Dostupné z: http://lavalle.pl/planning/

[5] PARKER, Lynne E. Multiple Mobile Robot Teams, Path Planning and Motion Coordination in. In: *Encyclopedia of Complexity and Systems Science* [online]. New York, NY: Springer New York, 2009, 2009, s. 5783-5800 [cit. 2020-06-18]. DOI: 10.1007/978-0-387-30440-3_344. ISBN 978-0-387-75888-6. Dostupné z: http://link.springer.com/10.1007/978-0-387-30440-3_344

[6] FERGUSON, Dave, Maxim LIKHACHEV a Anthony STENTZ. A Guide to Heuristic-based Path Planning. In: *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS)* [online]. 2005, s. 9-18 [cit. 2020-06-21]. Dostupné z: https://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/maxim/files/hsplanguide_icaps05ws.pdf

[7] YU, Wentao, Jun PENG, Xiaoyong ZHANG a Kuo-Chi LIN. A Cooperative Path Planning Algorithm for a Multiple Mobile Robot System in a Dynamic Environment. In: *International Journal of Advanced Robotic Systems* [online]. 2014 [cit. 2020-06-21]. DOI: 10.5772/58832. ISSN 1729-8814. Dostupné z: http://journals.sagepub.com/doi/10.5772/58832

[8] DUDEK, Gregory, Michael R.M. JENKIN, Evangelos MILIOS a David WILKES. A taxonomy for multi-agent robotics. In: *Autonomous Robots* [online]. 1996 [cit. 2020-06-21]. DOI: 10.1007/BF00240651. ISSN 0929-5593. Dostupné z: http://link.springer.com/10.1007/BF00240651

[9] YAN, Zhi, Nicolas JOUANDEAU a Arab Ali CHERIF. A Survey and Analysis of Multi-Robot Coordination. In: *International Journal of Advanced Robotic Systems* [online]. 2013 [cit. 2020-06-21]. DOI: 10.5772/57313. ISSN 1729-8814. Dostupné z: http://journals.sagepub.com/doi/10.5772/57313

[10] ASMA, Ayari a Bouamama SADOK. Dynamic Distributed PSO joints elites in Multiple Robot Path Planning Systems: theoretical and practical review of new ideas. In: *Procedia Computer Science* [online]. 2017, s. 1082-1091 [cit. 2020-06-21]. DOI: 10.1016/j.procs.2017.08.128. ISSN 18770509. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S1877050917314850

[11] SOLOVEY, Kiril, Oren SALZMAN a Dan HALPERIN. Finding a needle in an exponential haystack: Discrete RRT for exploration of implicit roadmaps in multi-robot motion planning. In: *The International Journal of Robotics Research* [online]. 2016, s. 501-513 [cit. 2020-06-21]. DOI: 10.1177/0278364915615688. ISSN 0278-3649. Dostupné z: http://journals.sagepub.com/doi/10.1177/0278364915615688

[12] SILVER, David. Cooperative Pathfinding. In: *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* [online]. 2005, s. 117-122 [cit. 2020-06-21]. Dostupné z: https://www.aaai.org/Papers/AIIDE/2005/AIIDE05-020.pdf

[13] ALAJLAN, Maram, Anis KOUBAA, Imen CHAARI, Hachemi BENNACEUR a Adel AMMAR. Global path planning for mobile robots in large-scale grid environments using genetic algorithms. In: *2013 International Conference on Individual and Collective Behaviors in Robotics (ICBR)* [online]. IEEE, 2013, 2013, s. 1-8 [cit. 2020-06-21]. DOI: 10.1109/ICBR.2013.6729271. ISBN 978-1-4799-2813-2. Dostupné z: http://ieeexplore.ieee.org/document/6729271/

[14] MASEHIAN, Ellips a Davoud SEDIGHIZADEH. Classic and Heuristic Approaches in Robot Motion Planning: A Chronological Review. In: *International Journal of Mechanical, Industrial Science and Engineering* [online]. World Academy of Science, Engineering and Technology, 2007 [cit. 2020-06-22]. DOI: 10.5281/zenodo.1074972. Dostupné z: https://www.researchgate.net/publication/249714449

[15] OPFER, Stephan, Hendrik SKUBCH a Kurt GEIHS. Cooperative Path Planning for Multi-Robot Systems in Dynamic Domains. In: *Mobile Robots - Control Architectures, Bio-Interfacing, Navigation, Multi Robot Motion Planning and Operator Training* [online]. InTech, 2011, 2011-12-02 [cit. 2020-06-22]. DOI: 10.5772/26440. ISBN 978-953-307-842-7. Dostupné z: http://www.intechopen.com/books/mobile-robots-control-architectures-bio-interfacing-navigation-multi-robot-motion-planning-and-operator-training/cooperative-path-planning-for-multi-robot-systems-in-dynamic-domains

[16] TSOURDOS, Antonios, Brian WHITE a Madhavan SHANMUGAVEL. *Cooperative Path Planning of Unmanned Aerial Vehicles* [online]. Washington, DC: American Institute of Aeronautics and Astronautics, 2010 [cit. 2020-06-22]. DOI: 10.2514/4.867798. ISBN 978-1-60086-779-8.

[17] ABD ALGFOOR, Zeyad, Mohd Shahrizal SUNAR a Hoshang KOLIVAND. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. In: *International Journal of Computer Games Technology* [online]. 2015, s. 1-11 [cit. 2020-06-22]. DOI: 10.1155/2015/736138. ISSN 1687-7047. Dostupné z: http://www.hindawi.com/journals/ijcgt/2015/736138/

[18] WALLAR, Alex a Erion PLAKU. Path planning for swarms in dynamic environments by combining probabilistic roadmaps and potential fields. In: *2014 IEEE Symposium on Swarm Intelligence* [online]. IEEE, 2014, 2014, s. 1-8 [cit. 2020-06-22]. DOI: 10.1109/SIS.2014.7011808. ISBN 978-1-4799-4458-3. Dostupné z: http://ieeexplore.ieee.org/document/7011808/

[19] MASEHIAN, Ellips a M. R. AMIN-NASERI. A voronoi diagram-visibility graph-potential field compound algorithm for robot path planning. In: *Journal of Robotic Systems* [online]. 2004, s. 275-300 [cit. 2020-06-22]. DOI: 10.1002/rob.20014. ISSN 0741-2223. Dostupné z: http://doi.wiley.com/10.1002/rob.20014

[20] GOLDENBERG, Meir, Ariel FELNER, Roni STERN, Guni SHARON a Jonathan SCHAEFFER. A* variants for optimal multi-agent pathfinding. In: *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence* [online]. 2012, s. 19-25 [cit. 2020-06-23]. Dostupné z: https://www.aaai.org/ocs/index.php/WS/AAAIW12/paper/viewPaper/5233

[21] STANDLEY, Trevor Scott. Finding optimal solutions to cooperative pathfinding problems. In: *Twenty-Fourth AAAI Conference on Artificial Intelligence* [online]. 2010, s. 173-178 [cit. 2020-06-23]. Dostupné z: https://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1926

[22] KRAFT, Aaron R. Abstraction Hierarchies for Multi-Agent Pathfinding. In: *Electronic Theses and Dissertations. 1247.* [online]. University of Denver, 2017, 73 p. [cit. 2020-06-23]. Dostupné z: https://digitalcommons.du.edu/etd/1247

[23] DIJKSTRA, E. W. A note on two problems in connexion with graphs. In: *Numerische Mathematik* [online]. 1959, s. 269-271 [cit. 2020-06-23]. DOI: 10.1007/BF01386390. ISSN 0029-599X. Dostupné z: http://link.springer.com/10.1007/BF01386390

[24] SINGHPAL, Narendra a Sanjeev SHARMA. Robot Path Planning using Swarm Intelligence: A Survey. In: *International Journal of Computer Applications* [online]. 2013, s. 5-12 [cit. 2020-06-22]. DOI: 10.5120/14498-2274. ISSN 09758887. Dostupné z: http://research.ijcaonline.org/volume83/number12/pxc3892274.pdf

[25] KENNEDY, J. a R. EBERHART. Particle swarm optimization. In: *Proceedings of ICNN'95 - International Conference on Neural Networks* [online]. IEEE, 1995, s. 1942-1948 [cit. 2020-06-22]. DOI: 10.1109/ICNN.1995.488968. ISBN 0-7803-2768-3. Dostupné z: http://ieeexplore.ieee.org/document/488968/

[26] NAKISA. A SURVEY: PARTICLE SWARM OPTIMIZATION BASED ALGORITHMS TO SOLVE PREMATURE CONVERGENCE PROBLEM. In: *Journal of Computer Science* [online]. 2014, s. 1758-1765 [cit. 2020-06-22]. DOI: 10.3844/jcssp.2014.1758.1765. ISSN 1549-3636. Dostupné z: http://thescipub.com/abstract/10.3844/jcssp.2014.1758.1765

[27] LIU, Shirong, Linbo MAO a Jinshou YU. Path Planning Based on Ant Colony Algorithm and Distributed Local Navigation for Multi-Robot Systems. In: *2006 International Conference on Mechatronics and Automation* [online]. IEEE, 2006, 2006, s. 1733-1738 [cit. 2020-06-23]. DOI: 10.1109/ICMA.2006.257476. ISBN 1-4244-0465-7. Dostupné z: http://ieeexplore.ieee.org/document/4026354/

[28] KARABOGA, Dervis. *An idea based on honey bee swarm for numerical optimization* [online]. 2005 [cit. 2020-06-23]. Dostupné z: https://www.semanticscholar.org/paper/AN-IDEA-BASED-ON-HONEY-BEE-SWARM-FOR-NUMERICAL-Karaboga/cf20e34a1402a115523910d2a4243929f6704db1

[29] LIANG, Jun-Hao a Ching-Hung LEE. Efficient collision-free path-planning of multiple mobile robots system using efficient artificial bee colony algorithm. In: *Advances in Engineering Software* [online]. 2015, s. 47-56 [cit. 2020-06-23]. DOI: 10.1016/j.advengsoft.2014.09.006. ISSN 09659978. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S0965997814001495

[30] DRÉO, Johann. Aco TSP. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2020, 2006 [cit. 2020-06-22]. Dostupné z: https://en.wikipedia.org/wiki/File:Aco_TSP.svg

[31] LAMINI, Chaymaa, Said BENHLIMA a Ali ELBEKRI. Genetic Algorithm Based Approach for Autonomous Mobile Robot Path Planning. In: *Procedia Computer Science* [online]. 2018, s. 180-189 [cit. 2020-06-23]. DOI: 10.1016/j.procs.2018.01.113. ISSN 18770509. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S187705091830125X

[32] GAO, Meijuan, Jin XU, Jingwen TIAN a Hao WU. Path Planning for Mobile Robot Based on Chaos Genetic Algorithm. In: *2008 Fourth International Conference on Natural Computation* [online]. IEEE, 2008, 2008, s. 409-413 [cit. 2020-06-23]. DOI: 10.1109/ICNC.2008.627. ISBN 978-0-7695-3304-9. Dostupné z: http://ieeexplore.ieee.org/document/4667315/

[33] HART, Peter, Nils NILSSON a Bertram RAPHAEL. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics* [online]. 1968, s. 100-107 [cit. 2020-06-21]. DOI: 10.1109/TSSC.1968.300136. ISSN 0536-1567. Dostupné z: http://ieeexplore.ieee.org/document/4082128/

[34] PATEL, Amit. Heuristics. *Amit-s A* Pages* [online]. 2020 [cit. 2020-06-23]. Dostupné z: https://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html

[35] Euclidean distance 2d. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2020, 2018 [cit. 2020-06-26]. Dostupné z: https://commons.wikimedia.org/wiki/File:Euclidean_distance_2d.svg

[36] BARILE, Margherita. Taxicab Metric. *MathWorld: A Wolfram Web Resource* [online]. [cit. 2020-06-26]. Dostupné z: https://mathworld.wolfram.com/TaxicabMetric.html

[37] KOENIG, Sven, Maxim LIKHACHEV a David FURCY. Lifelong Planning A. In: *Artificial Intelligence* [online]. 2004, s. 93-146 [cit. 2020-06-22]. DOI: 10.1016/j.artint.2003.12.001. ISSN 00043702. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S000437020300225X

[38] STENTZ, Anthony. Optimal and Efficient Path Planning for Partially Known Environments. In: *Intelligent Unmanned Ground Vehicles* [online]. Boston, MA: Springer US, 1997, 1997, s. 203-220 [cit. 2020-06-22]. DOI: 10.1007/978-1-4615-6325-9_11. ISBN 978-1-4613-7904-1. Dostupné z: http://link.springer.com/10.1007/978-1-4615-6325-9_11

[39] STENTZ, Anthony. The Focussed D* algorithm for real-time replanning. In: *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2* [online]. San Francisco: Morgan Kaufmann Publishers, 1995, s. 1662-1669 [cit. 2020-06-22]. Dostupné z: https://www.ijcai.org/Proceedings/95-2/Papers/082.pdf

[40] KOENIG, Sven a Maxim LIKHACHEV. D* Lite. In: *Eighteenth national conference on Artificial intelligence* [online]. American Association for Artificial Intelligence, 2002, s. 476-483 [cit. 2020-06-22]. Dostupné z: https://www.aaai.org/Papers/AAAI/2002/AAAI02-072.pdf

[41] SILVER, David. Cooperative Pathfinding. In: *AI Game Programming Wisdom 3* [online]. Charles River Media, 2006, s. 99-111 [cit. 2020-06-21]. ISBN 9781584504573. Dostupné z: https://www.davidsilver.uk/wp-content/uploads/2020/03/coop-path-AIWisdom-1.pdf

[42] HOLTE, Robert C., M. B. PEREZ, R. M. ZIMMER a A. J. MACDONALD. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In: *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 1* [online]. 1996, s. 530-535 [cit. 2020-06-25]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.1704&rep=rep1&type=pdf

[43] *ECMA-334: C# Language Specification* [online]. Geneva: ECMA International, 2017 [cit. 2020-06-21]. Dostupné z: https://www.ecma-international.org/publications/standards/Ecma-334.htm

[44] *Microsoft Docs: Technical documentation, API, and code examples* [online]. Microsoft, c2020 [cit. 2020-06-21]. Dostupné z: https://docs.microsoft.com/en-us/

[45] Stylet. *GitHub* [online]. c2020 [cit. 2020-06-21]. Dostupné z: https://github.com/canton7/Stylet

[46] QuickGraph. *GitHub* [online]. c2020 [cit. 2020-06-21]. Dostupné z: https://github.com/YaccConstructor/QuickGraph

[47] High Speed Priority Queue. *GitHub* [online]. c2020 [cit. 2020-06-21]. Dostupné z: https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp

[48] STURTEVANT, N. R. Benchmarks for Grid-Based Pathfinding. In: *IEEE Transactions on Computational Intelligence and AI in Games* [online]. 2012, s. 144-148 [cit. 2020-05-13]. DOI: 10.1109/TCIAIG.2012.2197681. ISSN 1943-068X. Dostupné z: http://ieeexplore.ieee.org/document/6194296/

Seznam použitých zkratek a symbolů

C	Konfigurační prostor
C_{obs}	Kolizní konfigurační prostor
C_{free}	Volný konfigurační prostor
RRT	Rychle rostoucí náhodný strom, Rapidly-exploring Random Tree
PEA*	Partial Expansion A*
SI	Inteligence hejna, Swarm Intelligence
PSO	Optimalizace hejnem částic, Particle Swarm Optimization
ACO	Optimalizace mravenčí kolonií, Ant Colony Algorithm
ABC	Umělá včelí kolonie, Artificial Bee Colony
LRA*	Local Repair A*
LPA*	Lifelong Planning A*
D*	Dynamic A*
MA D* Lite	Multi Agent D* Lite
CA*	Cooperative A*
HCA*	Hierarchical Cooperative A*
RRA*	Reverse Resumable A*
WHCA*	Windowed Hierarchical Cooperative A*
GUI	Grafické uživatelské rozhraní, Graphical User Interface
WPF	Windows Presentation Foundation

Seznam příloh

- `\src` – složka se zdrojovými kódy
- `\MultiRobotSimulator` – složka s aplikací
- `\MultiRobotSimulator\MultiRobotSimulator.WPF.exe` – spustitelný soubor aplikace