VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

NAVIGACE ROBOTU POMOCÍ GRAFOVÝCH ALGORITMŮ

ROBOT NAVIGATION BY MEANS OF GRAPH-BASED ALGORITHMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LUBOMÍR ČÍŽEK
AUTHOR

VEDOUCÍ PRÁCE — RNDr. JIŘÍ DVOŘÁK, CSc.
SUPERVISOR

BRNO 2011

ZADÁNÍ ZÁVĚREČNÉ PRÁCE

(na místo tohoto listu vložte originál a nebo kopii zadání Vaš práce)

ABSTRAKT

Tato práce se zabývá plánováním cesty robota pomocí grafových algoritmů. Její teoretická část se zabývá základními přístupy plánování cesty robota a věnuje bližší pohled na různé metody grafových algoritmů. V druhé části této diplomové práci bylo vytvořeno simulační prostředí navigace robota v jazyce C#. A v tomto prostředí byly implementovány vybrané metody grafových algoritmů.

Tato práce byla napsána v rámci výzkumného záměru MSM 0021630529: Inteligentní systémy v automatizaci.

ABSTRACT

This thesis deals with robot path planning by means of graph-based algorithms. The theoretical part contains basic approaches to robot path planning, and pay closer look at various methods of graph-based algorithms. In the second part of this thesis a simulation environment for robot navigation was created in C#. And in this environment chosen methods of graph-based algorithms have been implemented.

This thesis was written within the research project MSM 0021630529: Intelligent systems in automation.

KLÍČOVÁ SLOVA

Mobilní robot, plánování cesty, grafové algoritmy.

KEYWORDS

Mobile robot, path planning, graph-based algorithms.

BIBLIOGRAFICKÁ CITACE

ČÍŽEK, Lubomír. *Navigace robotu pomocí grafových algoritmu.* Brno, 2011. 63s. Diplomová práce na Fakultě strojního inženýrství Vysokého Učení Technického v Brně na Ústavu automatizace a informatiky. Vedoucí diplomové práce RNDr. Jiří Dvořák, Csc.

PROHLÁŠENÍ O PŮVODNOSTI PRÁCE

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a že jsem uvedl všechny využité prameny a literaturu.

_____

PODĚKOVÁNÍ

Tímto bych rád poděkoval vedoucímu mé diplomové práce, RNDr. Jiřímu Dvořákovi za jeho podněty a poznatky, které mi pomohly při vyhotovování této práce.

Obsah:

1 ÚVOD

Tato práce se zabývá plánováním cesty mobilního robota. V této oblasti bylo v posledních několika letech dosaženo velkého pokroku. Ať už jde o plánování cesty v plně známém prostředí, v komplexních částečně neznámých prostředích až po problematiku plánování cesty v kompletně neznámém prostředí a v reálném čase. V současné době dokáží autonomní roboty řešit složité praktické problémy ve složitých prostředích. Metody plánování cesty se uplatňují v mnoha aplikacích, jako je např. automatizace, počítačová animace, průmyslový design nebo chirurgie [2].

Navigace robota se zabývá řešením problému, jak se má robot přesunout z počáteční pozice do cílové. Robot musí být schopen reagovat na překážky, ať už statické či dynamické. Ve druhém případě se tak nedá plně spoléhat na informace, které robot měl před zahájením plánování cesty, a je třeba získat podrobnější informace o jeho okolí a robot musí mít schopnost je správně využít [1].
Plánování cesty robota se dá rozdělit na globální a lokální plánování. Globální hledání se snaží najít cestu bez kolizí s překážkami z počáteční pozice do cílové, a to před tím, než se robot začne pohybovat. Při tom je ovšem třeba vzít v úvahu několik faktorů, například zda je známá mapa daného prostředí, a s jakou přesností je robot schopen určit svoji počáteční a cílovou pozici. Potom může začít samotné plánování vhodné cesty, která by měla být nejen plně průchozí, ale také co nejkratší, ekonomická a případně splňovat ještě jiné dodatečné požadavky.

Naproti tomu se u lokálního plánování hledá cesta až během pohybu robota a jeho účelem je zabránění kolizím s překážkami, které nebyly uvedeny na mapě či během pohybu robota změnily svoji polohu. Robot analyzuje změny v prostředí a na jejich základě, pokud je to zapotřebí, upravuje cestu. Lokální plánování se používá pro pohyb robota po cestě získané za pomocí globálního plánování nebo pro plánování cesty v neznámém prostředí.

Dále můžeme plánování rozdělit na holonomní a neholonomní. V neholonomním plánování jsou na robota kladena omezení z hlediska pohybu (rychlost, setrvačnost...atd.) a času. Holonomní plánování je bez omezení [1].

Cílem této diplomové práce bylo analyzovat dosavadní přístupy plánování cesty robota se zaměřením na grafové algoritmy, vytvořit simulační prostředí pro plánování cesty robota, v něm implementovat vybrané algoritmy a provést pro ně ověřovací a srovnávací experimenty.

2 METODY NAVIGACE MOBILNÍHO ROBOTA

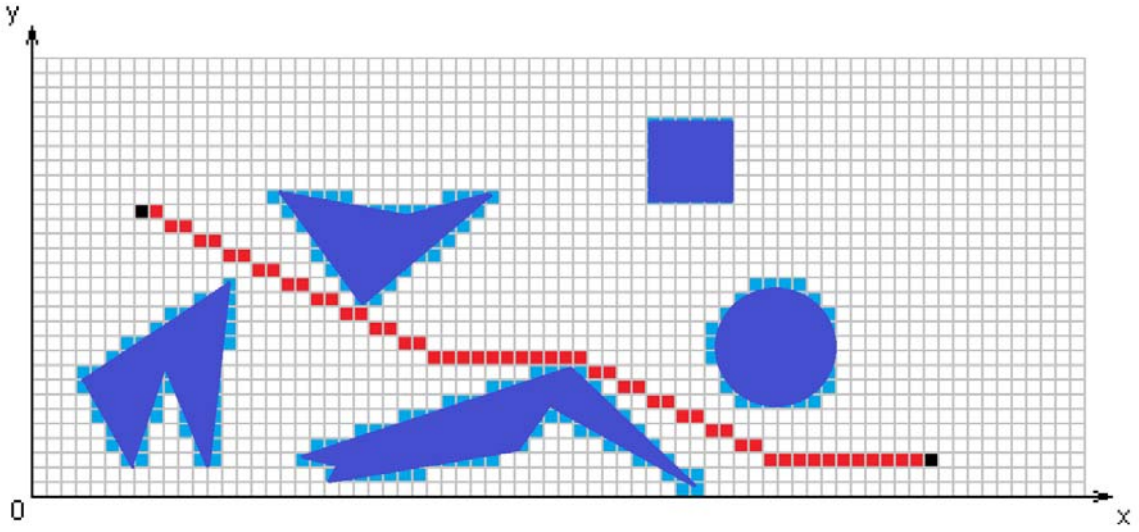
Existuje velké množství různých metod navigace mobilního robota a jejich různých modifikací, a proto budou v následující kapitole uvedeny pouze některé z nich. Všechny z těchto metod se skládají ze dvou částí. První je tzv. předzpracování, kdy se popisuje pracovní prostor pomocí grafu, mřížky nebo funkce. Druhá tzv. dotazovací část provádí hledání mezi počátečním a cílovým bodem. Informace uvedené v této kapitole byl čerpány z [1],[2],[3].

2.1 Metoda rozkladu do buněk

Metoda spočívá v rozkladu prostředí do buněk. Buňky mají svůj specifický tvar a velikost. Nejčastěji používané jsou čtvercové, trojúhelníkové a šestiúhelníkové buňky. Každá buňka obsahuje informaci zda její plocha obsahuje překážku nebo volný prostor. Na základě těchto informací se vytvoří graf, jehož vrcholy představují buňky bez překážek a jeho hrany jsou spojnice vrcholů představujících sousední buňky. Rozklad se dá provádět buďto aproximační nebo exaktní metodou rozkladu do buněk.

2.1.1 Aproximativní rozklad do buněk

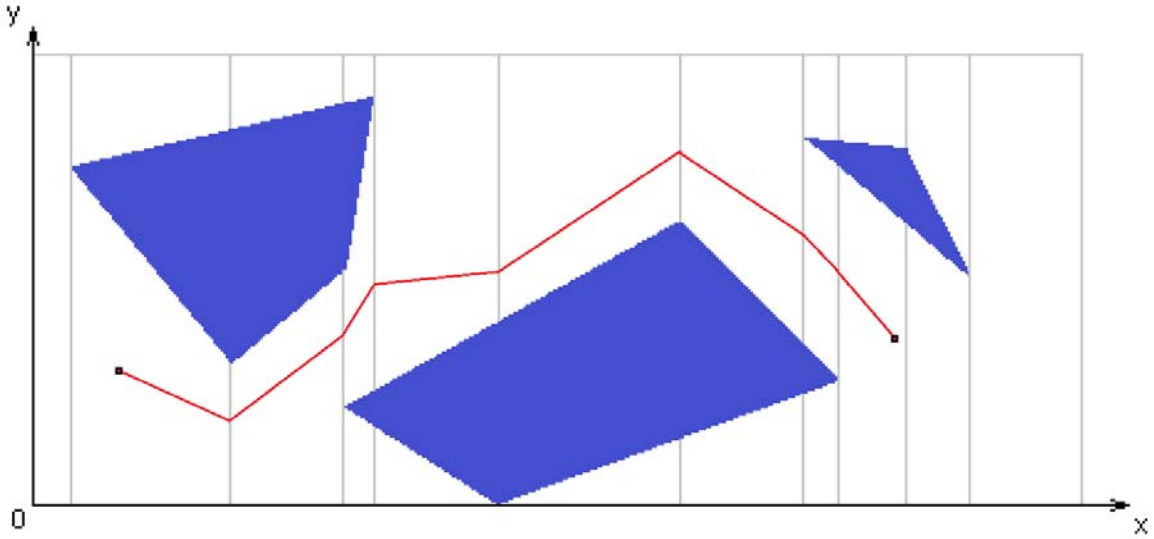
Metoda rozdělí prostředí do jednotlivých buněk stejného tvaru a tím ho převede na diskrétní tvar. Všechny buňky obsahující část překážky, bez ohledu na její velikost, se označí za neprůchozí. Z toho vyplývá, že s klesající velikostí buňky roste přesnost, ale také roste výpočetní a paměťová náročnost.



Obr. 1 Aproximativní rozklad prostředí do buněk.

2.1.2 Exaktní rozklad do buněk

Metoda rozdělí prostředí do množiny nepřekrývajících se buněk. Jejich tvar se volí záměrně jednoduchý, aby se daly jednoduše spočítat hranice mezi nimi kvůli potřebě určení bodu přechodu do další buňky Nejčastěji používanými tvary buněk jsou trojúhelníky nebo lichoběžníky. Ukázka rozkladu do lichoběžníkových buněk je na Obr. 2 . Trasa robota se skládá z počátečního a cílového bodu a bodů přechodu, které robot využívá k bezpečnému pohybu mezi překážkami v modelu prostředí.



Obr. 2 Exaktní rozklad do lichoběžníkových buněk.

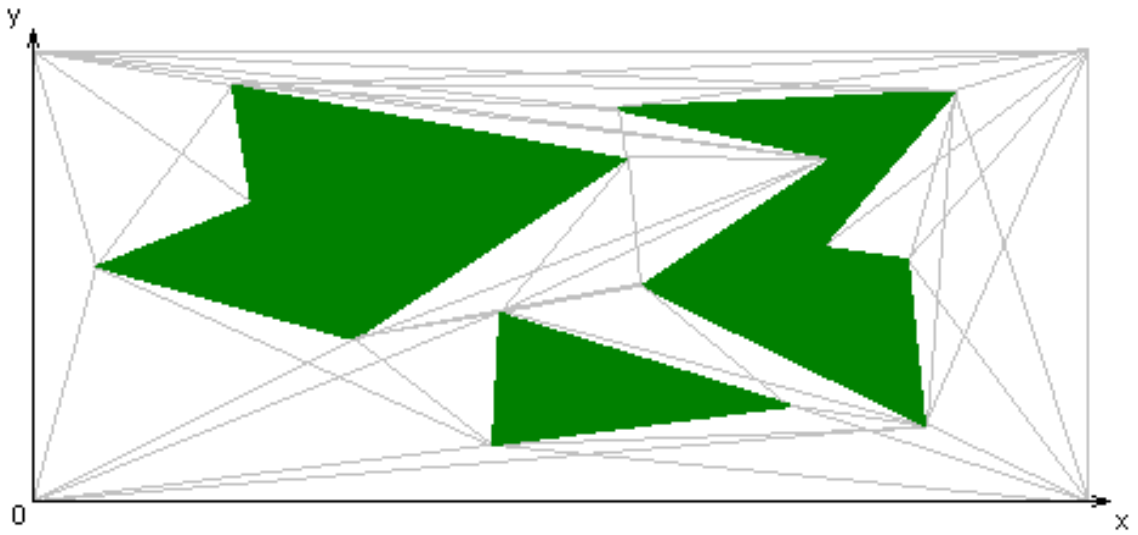
2.2 Metody mapy cest

Tyto metody vytvářejí mapu cest, která má podobu grafu reprezentujícího volný prostor. Přidáme startovní a cílový bod jako další vrcholy grafu a hrany grafu tvoří cesty, po kterých se může robot pohybovat. Pro hledání cesty se poté využije některý z algoritmů hledání cesty grafem.

2.2.1 Deterministické metody

Graf viditelnosti

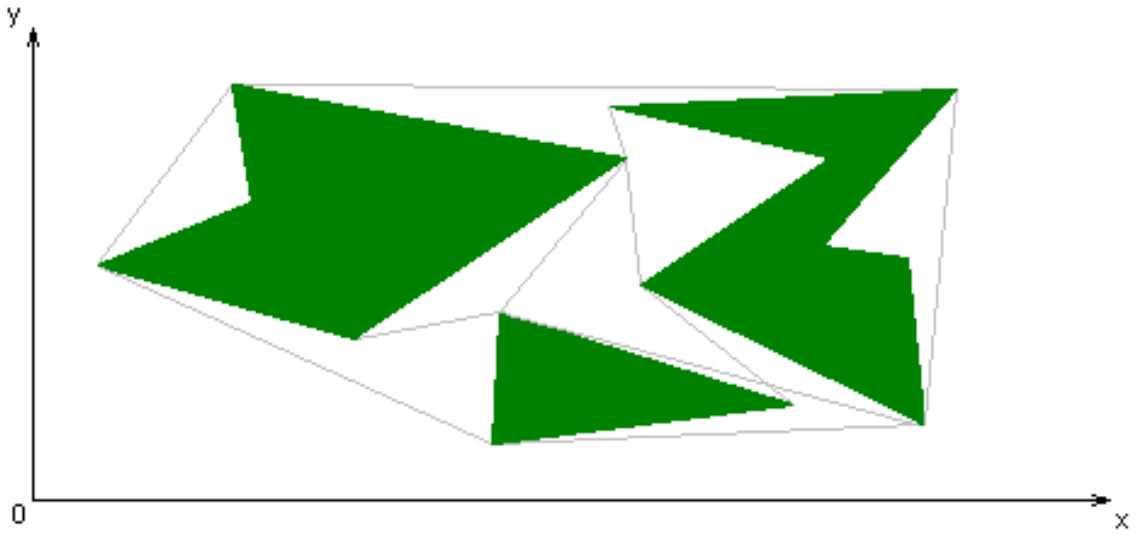
Graf viditelnosti představuje popis volného pracovního prostoru, kde vrcholy grafu představují startovní a cílový bod robota a také vrcholy všech překážek v pracovním prostoru robota. Hrany grafu jsou tvořeny spojnicemi vrcholů překážek, přičemž jsou vybrány pouze ty spojnice, které neprochází žádnou z překážek. Omezením grafu viditelnosti je, že je nemůžeme použít na nepolygonální překážky, a proto kružnice, elipsy aj. je nutné nejdříve převést na polygonální tvar.



Obr. 3 *Graf viditelnosti* [1].

Graf tečen

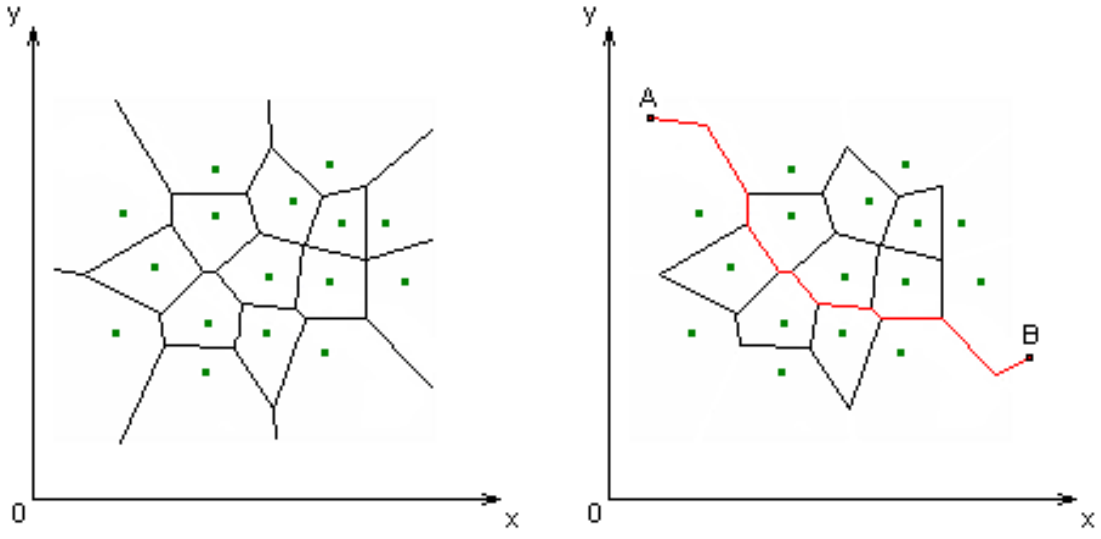
Graf tečen je redukovaný graf viditelnosti, ve kterém odstraníme hrany grafu, které nejsou tečnami jednotlivých vrcholů. A tímto snížením počtu hran zvýšíme rychlost prohledávání grafu.



Obr. 4 Graf tečen [1].

Voronoiův diagram

Voronoiův diagram je geometrická struktura, reprezentující informace o sousedství objektů v dané rovině tvořené body o stejné vzdálenosti od jedné nebo více překážek. Body společně tvoří hrany diagramu, po kterých se robot může pohybovat bez rizika kolize s překážkou. Diagram je tedy vytvořen body, které jsou ve stejné vzdálenosti od dvou či více překážek. Body o stejné vzdálenosti od tří nebo více překážek jsou považovány za vrcholy grafu, a body ve stejné vzdálenosti od dvou překážek jsou jeho hranami. Do grafu je třeba ještě zahrnout první a poslední úsek cesty, které vzniknou napojením startu a cíle na nejbližší hrany Voronoiova diagramu. Na Obr. 5 je znázorněno řešení pomocí Voronoiova diagramu v modelu prostředí s bodovými překážkami.



Obr. 5 Voronoiův diagram [2].

2.2.2 Pravděpodobnostní metody

Pravděpodobnostní mapy cest

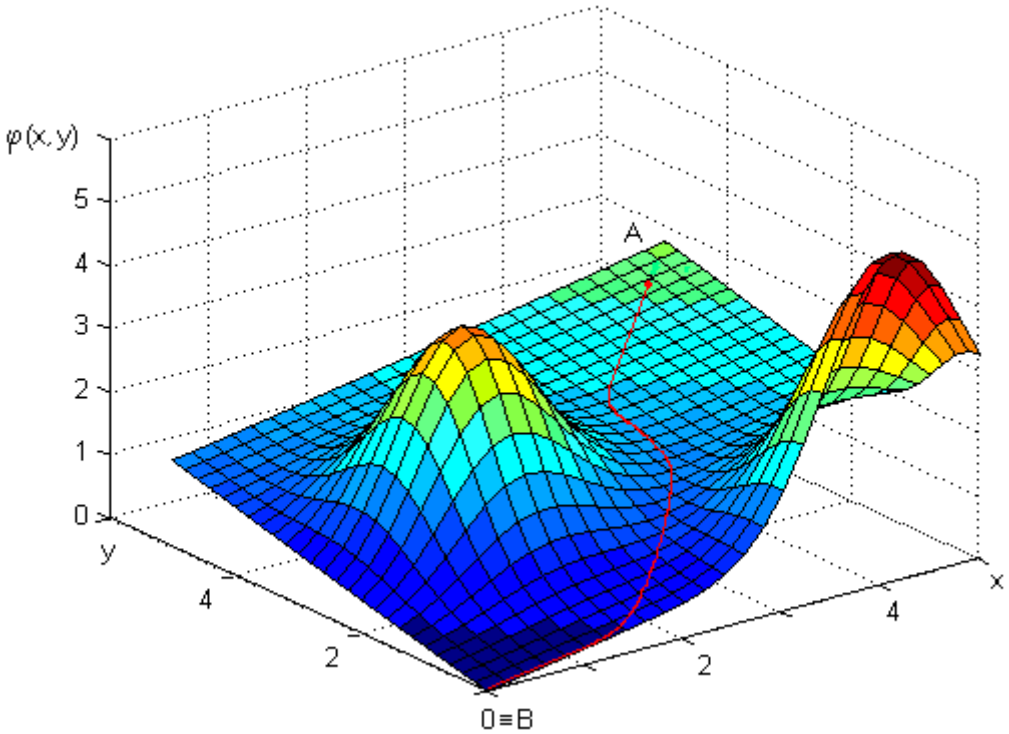
Tento algoritmus se skládá ze dvou fází. V první fázi, která je nazývaná učící, se vytvoří mapa cest. To se provádí náhodným výběrem pozice robota, u které se následně ověří, zda neleží v nějaké překážce. Pokud tam neleží, jsou vytvořeny nové vrcholy grafu. Poté se algoritmus pokouší propojit nově vytvořené konfigurace s ostatními pomocí přímých cest. Pokud není tato cesta v kolizi s překážkami, je přidána jako nová hrana. Tato fáze se několikrát opakuje, dokud graf dostatečně nepopisuje volný prostor, nebo do doby, kdy je vyčerpán přidělený čas či paměť. V druhé fázi, která se nazývá dotazovací, jsou nejprve k vrcholům přidány počáteční a cílový bod, které se opět propojí možnými cestami. A poté probíhá hledání nejkratší cesty v grafu. Pokud cesta není nalezena, může to byt způsobeno nedostatečným popsáním volného prostoru, a proto se opakuje první fáze. Tento algoritmus je vhodný především pro statická prostředí, kde je možné provádět opakovaně druhou fázi bez nutnosti opakovaní fáze první.

Pravděpodobnostní stromy

Základní myšlenkou tohoto algoritmu, který byl poprvé představen v roce 1998, je inkrementální vytváření prohledávacího stromu, který zkouší rychle a rovnoměrně prohledat pracovní prostor. Strom se vytváří tak, že se v každí iteraci rozroste směrem k náhodně vygenerované pozici robota o nový vrchol. Vrcholy stromu tedy představují pozice robota a hrany představují akce, pomocí kterých se robot do dané pozice dostane. Základní verze algoritmu se ovšem v praxi příliš nepoužívá, protože generovaný strom se rozrůstá všemi směry stejně, což vede k pomalé konvergenci k cíli.

2.2.3 Potenciální pole

Metoda potenciálového pole popisuje prostředí potenciálovou funkcí $E(x,y)$. Startovní pozice robota má vyšší potenciál než pozice cílová. Překážky mají vyšší potenciál než volný pracovní prostor. Robot hledající cestu se pohybuje ve směru opačného gradientu potenciálové funkce.



Obr. 6 Funkce potenciálu pole vytvořena v prostředí Matlab [3].

2.2.4 Mravenčí algoritmy

Mravenčí algoritmy jsou pravděpodobnostní algoritmy rojové inteligence, inspirované chováním skutečných mravenců například při vyhledávání potravy, stavbě mraveniště, rozdělování prací či přepravě. Hlavní myšlenkou mravenčích algoritmů je fakt, že principy samoorganizace, které umožňují vysoce koordinované chování reálných mravenců, mohou být použity pro koordinaci populací umělých jedinců, kteří budou vzájemně spolupracovat na řešení výpočetních problémů. Tato spolupráce probíhá pomocí nepřímé komunikaci mezi členy v kolonii pomocí změn ve feromonové stopě. Pro hledání cesty grafem se využívají tzv. umělí mravenci, kteří mají stejně jako jejich biologické předlohy pravděpodobností chovaní. Takže čím vyšší hodnotu feromonu detekují na cestě, tím větší je pravděpodobnost, že se touto cestou vydají. A čím častěji mravenci procházejí touto cestou, tím její feromonová vrstva narůstá a naopak pokud mravenci tuto cestu budou využívat méně, feromon se začne vypařovat a jeho hodnota klesá. Pro použití mravenčích algoritmů při plánovaní dráhy robota je třeba popsat pracovní prostor pomocí grafu a určit startovní a cílový bod. [2][3]

2.2.5 Genetické algoritmy

Genetické algoritmy vycházejí z podstaty evolučního vývoje, při kterém se prosazuji jedinci s žádoucími vlastnostmi, jejichž genom je dále šířen prostřednictvím potomků. Pro manipulaci s genomem se používá řada genetických operátorů, mezi které patři dědičnost, selekce, mutace či křížení. Jedním kriteriem, které může být zohledňováno při výběru rodičovského genomu, je hodnota „fitness“ funkce, která vyjadřuje kvalitu řešení reprezentovaného tímto jedincem. Při řešeni úloh za použiti genetických algoritmů reprezentuje genom konkrétní řešeni a jeho fitness funkce je kladná hodnota odpovídající hodnotě účelové funkce. Princip funkce algoritmu je tedy takový, že nejprve pomocí náhodného generovaní či jiných heuristických metod vytvoříme počáteční populaci určitého počtu genomů, kterou následně měníme aplikací genetických operátorů do té doby, kdy je splněna podmínka ukončení, což může např. být vyčerpání času, či zjištění, že za určitou dobu nedošlo ke zlepšení účelové funkce. Dříve se genom reprezentoval především za pomocí binární reprezentace, v dnešní době se však považuje za výhodnější použití nebinární reprezentace. Pro plánování cesty robota bylo navrženo velké množství metod pro diskrétní i spojité prostředí. Metody mohou využívat kromě klasických operátorů také operátory specifické, jako je např. operátor vyhlazení cesty [1][3].

2.2.6 Fuzzy logika

Navigace mobilních robotů pomocí fuzzy logiky je velmi používanou metodou díky schopnosti pracovat s nepřesnými vstupními informacemi. Často se tyto metody používají v kombinaci s neuronovými sítěmi a genetických algoritmy. Slovní proměnné definují vzdálenost robotu od nejbližší překážky a úhel mezi osou robota a touto překážkou. Výstupem je natočení a zrychlení robota. Šířku fuzzy množin těchto proměnných můžeme modifikovat za pomoci použití koeficientů.

Jiným možným přístupem je využití fuzzy inferenčního systému inspirovaného lidským chováním, jenž je založen na vyhledávaní volného prostoru, kde se nevyskytují překážky, s účelem dosáhnutí cílové pozice. Ovšem při výskytu některých konkávních překážek tento způsob selhává a je nutné použít nějakou jinou strategii, jako například sledování stěny s využitím podcílů.[3]

3 GRAFOVÉ ALGORITMY

Grafové algoritmy slouží k prohledávání orientovaných grafů. Přestože v dnešní době mají počítače vysoký výpočetní výkon, je prakticky nemožné při řešení problémů testovat všechny možnosti. Jedním způsobem, jak efektivně zvládnout hledání správného řešení, je rozdělení problému do různých stavů. Jeden z těchto stavů bude počáteční stav, stavy s požadovanými vlastnostmi budou cílové stavy, a mezi jednotlivými stavy se bude moci přecházet pomocí předem definovaných pravidel. Tím nám vznikne stavový prostor, který si můžeme představit jako orientovaný graf. Jeho vrcholy reprezentují jednotlivé stavy a hrany pravidla, jejichž splněním se dostaneme z jednoho stavu do druhého. Řešení pak spočívá v nalezení cesty grafem.

Metody prohledávání grafu se dělí na neinformovaní a informované, a ty se dají nadále dělit na heuristické a inkrementální (podrobněji viz [14], [15]).

Neinformované metody

Neinformované metody nemají k dispozici žádné údaje o stavovém prostoru, které by jim pomohly při hledání cesty k cíli. Proto musí procházet graf systematicky vrchol po vrcholu dokud nenaleznou řešení. Jednotlivé algoritmy se od sebe liší jen tím, jakým způsobem toho prohledávání provádějí. Mezi tyto metody patří prohledávaní do šířky, prohledávání do hloubky nebo obousměrné prohledávání.

Heuristické metody

Heuristické metody prohledávání, jsou schopny najít nejkratší cestu rychleji než metody neinformovaného hledání. Mezi patří různé metody jako například A*, Greedy, Hill-climbing, a jsou široce používány v oblasti umělé inteligence. Tyto metody používají heuristické znalosti ve formě aproximací vzdáleností k cíli a zaměřily se na hledání a nalezení nejkratší cesty při řešení problémů plánování cesty.

Inkrementální metody

Většina výzkumu prohledávacích metod je zaměřena na řešení „jednorázových“ problémů plánování cesty. Nicméně mnoho systémů umělé inteligence se musí přizpůsobovat neustále se měnícímu prostředí, nebo jeho modelům. V těchto případech nemusí v průběhu řešení problému původní řešení již platit, nebo již nemusí být optimální. V tomto případě je potřeba řešení přeplánovat na novou situaci. Mezi příklady praktického využití patří plánování letecké evakuace zraněných lidí v krizových situacích nebo plánování leteckého provozu. Stejně tak je třeba řešit řadu obdobných problémů plánování, například pokud někdo chce provést řadu „what-if“ analýz nebo zjištění nákladů na plánování subjektů, jejichž předpoklady nebo účinky se v průběhu času mění. Z těchto důvodů může hledání být často se opakující proces. V této situaci mnoho systémů umělé inteligence musí provádět přeplánování od nuly, to je vyřešit problém plánovaní cesty samostatně. Nicméně toto může být neefektivní, a to především ve velkých oblastech s častými změnami, čímž se výrazně omezuje schopnost systémů reagovat a často je činí nepřijatelnými. Tyto problémy se stávají ještě více obtížné pokud dojde ke změně v průběhu plánování. Naštěstí změny při problému plánovaní cesty jsou obvykle malé. Například letadla nadále nebudou moci přistát na letištích pro leteckou evakuaci. To naznačuje, že kompletní přepočet problému může být zbytečný, protože některé z předchozích výsledků hledání mohou být znovu použity. A toto je základem inkrementálních vyhledávacích metod. Všimněte si, že použití terminologie je bohužel poněkud problematické, protože pojem "inkrementální vyhledávání" se také týká on-line vyhledávání.[5]

3.1 Metoda A*

A* algoritmus pracuje se dvěma seznamy, Open a Closed pro řízení systematického hledání minimální ceny cesty od počátečního vrcholu k cílovému vrcholu grafu. Zpočátku Open seznam

obsahuje počáteční vrchol a Closed seznam je prázdný. V každém cyklu algoritmu je nejslibnější Open vrchol expandován a přesunut do Closed seznamu, a jeho následníci uzly jsou zařazeny do seznamu Open. Takže Closed seznam obsahuje ty vrcholy, které byly expandovány a to vytvářením jejich následníků a Open seznam obsahuje ty vrcholy, které byly získány expanzí ale samy ještě nebyly expandovány. Hledání končí vybráním cílového vrcholu pro expanzi. Výsledná cesta může být získána sledováním zpětných ukazatelů od cílového vrcholu do počátečního vrcholu. Pořadí, ve kterém jsou vrcholy expandovány, je určeno pomocí hodnotící funkce f(s) = g(s) + h(s), kde g(s) je cena v současné době nejlepší známé cesty od počátečního vrcholu do vrcholu s a h(s) je heuristický odhad hodnoty h*(s), což je cena nejlepší cesty od vrcholu s k cílovému vrcholu. Chování algoritmu A* závisí z velké části na heuristice h(s), která řídí hledání. Pokud h(s) je přípustné, což znamená že není vyšší než h*(s), a pokud jsou vrcholy expandovány v souladu s f(s), pak je první cílový vrchol, který je vybrán pro expanzi, zaručeně optimální. Heuristika je nazývána konzistentní pokud $h(s) \leq c(s,s') + h(s')$ pro všechna s a s', kde c(s, s') je cena hrany vedoucí z vrcholu s do vrcholu s' . Pokud h(s) je konzistentní a vrcholy jsou expandovány v pořadí daném f(s), jsou ceny g vrcholu zaručeně optimální, když vrchol je vybrán pro expanzi a není nikdy expandován více než jednou. Povšimněme si, že konzistence zde znamená přípustnost a nepřípustnost znamená nekonzistenci.

Pokud h(s) není konzistentní, je možné pro A* najít lepší cestu k vrcholu poté, co vrchol je expandován. V tomto případě lepší cena g vrcholu musí být předána jeho následníkům. A* tohoto obvykle dosáhne tím, že přesune vrcholy s lepší cenou g ze seznamu Closed do seznamu Open. Když je vrchol nakonec znovu expandován, zlepšené ceny g se předají jeho následníkům, které pak může být potřeba také obnovit. V důsledku toho mohou být stejné vrcholy expandovány vícekrát. [11]

The pseudocode below assumes the following:
(1) $g(s_{start}) = 0$ ang g-values of the rest of states are set to ∞.
(2) OPEN = $\{s_{start}\}$.

```
1 procedure ComputePath()
2 while (s_goal is not expanded)
3       remove s with smallest f(s) from OPEN;
4       for each successor s' of s
5       if g(s') > g(s) + c(s, s)'
6          g(s') = g(s) + c(s, s)';
7          insert/update s' in OPEN with f(s') = g(s') + h(s');
```

Obr. 7 A Search: ComputePath function [10].*

3.2 Metoda Lifelong Planning A*

Patří mezi inkrementální vyhledávací metody na bázi DynamicSWSFFP, což jí umožňuje znovu používat informace z předchozích hledání pro nalezení nejkratší cesty pro sérii podobných problémů plánování cesty a dosáhnout výsledku potenciálně rychleji, než je možné při řešení jednotlivých problémů plánování cesty od nuly.Metoda Lifelong Planning A* (LPA*) dostala svůj název v analogii k "celoživotnímu vzdělávání" proto, že opakovaně využívá informace z předchozích hledání (někdy se také používá termín neustálé plánování). LPA* opakovaně hledá nejkratší cesty z počátečního do cílového vrcholu daného grafu, zatímco odstraňujeme, přidáváme či měníme hrany, protože zadané předpoklady, ceny pohybu, nebo jejich důsledky se změnily. LPA * zobecňuje jak DynamicSWSF-FP tak A* a slibuje, že najde nejkratší cesty rychleji, než tyto dvě metody vyhledávání individuálně, protože v sobě spojuje jejich techniky. Tato metoda je snadno pochopitelná, snadno analyzovatelná a snadno optimalizovatelná. Její první hledání je stejné jako u metody A *, která mezi vrcholy se stejnou f-hodnotou preferuje stavy s menší g-hodnotou. Následná vyhledávání jsou ale potenciálně rychlejší, protože opakovaně používají části předchozích vyhledávacích stromů, které jsou totožné s novým vyhledávacím stromem, a používají efektivní metody pro identifikaci těchto částí. To může snížit vyhledávací čas, a to především pokud velké části vyhledávacího stromu jsou stejné, například je-li problém plánování cesty změněn jen nepatrně a změny jsou blízko k cíli. LPA* je také

schopen zpracovat změny grafu v průběhu hledání a může být rozšířen o nepřípustnou heuristiku, kritéria účinnějšího rozhodování mezi stavy a nedeterministické grafy.[5]

3.2.1 Základní princip

Metoda Lifelong Planning A* se aplikuje na problém plánování cesty ve známém konečném grafu, ceny jehož hran se zvyšují nebo snižují v závislosti na čase (tyto změny nákladů mohou také být použity pro modelování hran nebo vrcholů, které jsou přidány či odstraněny). S označuje konečnou množinu vrcholů grafu, succ(s) $\subseteq$ S označuje množinu následníků vrcholu s $\in$ S, pred (s) $\subseteq$ S označuje množinu předchůdců vrcholu s $\in$ S, $0 < c(s, s') \leq \infty$ označuje cenu pohybu od vrcholu s k vrcholu s' $\in$ succ(s). LPA* vždy určuje nejkratší cestu z daného počátečního vrcholu $s_{start} \in S$ do daného konečného vrcholu $s_{goal} \in S$ se znalostí jak topologie grafu tak aktuálních cen hran. Používáme funkci g*(s), která nám popisuje vzdálenost vrcholu s $\in$ S od počátečního vrcholu, tedy náklady na nejkratší cestu od s_{start} do s. [5] Vzdálenost od počátku by se měla řídit vztahem (1)

$$g*(s)=\begin{cases}0 & pro\ s = s_{start} \\ min_{s' \in pred(s)}(g*(s')+c(s',s)) & pro\ s \neq s_{start}\end{cases} \quad (1)$$

3.2.2 Detaily

V předchozí kapitole byl vysvětlen základní princip metody LPA *. Nyní budou vysvětleny podrobnosti. LPA* je inkrementální verze A*, které platí pro stejný konečný problém plánování cesty jako A*. Také sdílí s A* skutečnost, že používá nezápornou a konzistentní heuristiku h(s), která aproximuje vzdálenost od konečného vrcholu od vrcholu s pro zaměření svého vyhledávání. Konzistentní heuristika se řídí pravidlem trojúhelníkové nerovnosti $h(s_{goal}) = 0$ a $h(s) \leq c(s, s') + h(s')$ pro všechny vrcholy s $\in$ S a s' $\in$ succ s), když platí $s \neq s_{goal}$.

Pseudokód, uvedený na Obr. 8 , používá pro správu prioritní fronty následující funkce: U.TopKey() vrací nejmenší prioritu ze všech vrcholů, které se nachází v prioritní frontě U. Pokud je U prázdná, pak U.TopKey() vrací [∞, ∞]. U.Pop() odstraní vrchol s nejmenší prioritou z prioritní fronty U a tento vrchol navrátí. U.Insert(s, k) vkládá vrchol s do prioritní fronty U s prioritou k. A U.Remove(s) odstraní vrchol s z prioritní fronty U.

procedure CalculateKey(s)
{01} return [min(g(s); rhs(s)) + h(s; s_{goal}); min(g(s); rhs(s))];

procedure Initialize()
{02} U = Ø;
{03} for all s $\in$ S rhs(s) = g(s) = ∞;
{04} rhs(s_{start}) = 0;
{05} U.Insert(s_{start} ; CalculateKey(s_{start}));

procedure UpdateVertex(u)
{06} if (u ≠ s_{start}) rhs(u) = $min_{s' \in Pred(u)}$(g(s') + c(s'; u));
{07} if (u $\in$ U) U.Remove(u);
{08} if (g(u) ≠ rhs(u)) U.Insert(u; CalculateKey(u));

procedure ComputeShortestPath()
{09} while (U.TopKey() < CalculateKey(s_{goal}) OR rhs(s_{goal}) ≠g(s_{goal}))
{10} u = U.Pop();

```
{11}     if (g(u) > rhs(u))
{12}              g(u) = rhs(u);
{13}              for all s ∈ Succ(u) UpdateVertex(s);
{14}     else
{15}             g(u) = ∞;
{16}             for all s ∈ Succ(u) ∪ UpdateVertex(s);

procedure Main()
{17} Initialize();
{18} forever
{19}     ComputeShortestPath();
{20}     Wait for changes in edge costs;
{21}     for all directed edges (u; v) with changed edge costs
{22}              Update the edge cost c(u; v);
{23}              UpdateVertex(v);
```

Obr. 8 Pseudokód LPA[5].*

3.2.3 Proměnné

LPA* udržuje odhad g(s) hodnoty funkce g*(s), tj. vzdálenosti každého vrcholu s od počátku. Počáteční vyhledávání LPA* počítá g-hodnoty každého vrcholu v přesně stejném pořadí jako A*. LPA* pak přenáší g-hodnoty do dalších hledání. LPA také udržuje jiný druh odhadu vzdáleností od počátku. Rhs-hodnoty jsou výhledové hodnoty (založené na hodnotách g) počítané o jeden krok dopředu podle vztahu (2)

$$rhs(s)=\begin{cases}0 & pro\ s=s_{start}\\ min_{s'\in pred(s)}(g(s')+c(s',s)) & pro\ s\neq s_{start}\end{cases} \qquad (2)$$

Vrchol je nazýván lokálně konzistentní, právě když jeho g-hodnota se rovná její rhs-hodnotě. Tento koncept je podobný podmínce Bellmanovy rovnice pro nediskontované deterministické sekvenční rozhodovací problémy. Pokud jsou všechny vrcholy lokálně konzistentní, pak všechny jejich g-hodnoty splňují vztah (3).

$$g(s)=\begin{cases}0 & pro\ s=s_{start}\\ min_{s'\in pred(s)}(g(s')+c(s',s)) & pro\ s\neq s_{start}\end{cases} \qquad (3)$$

Porovnáním rovnice s (1) vyplývá, že všechny g-hodnoty vrcholů jsou rovny jejich vzdálenosti od počátku. Takže g-hodnoty všech vrcholů se rovnají jejich vzdálenostem od počátku pokud všechny vrcholy jsou lokálně konzistentní. Tento koncept je důležitý, protože pak můžeme zpětně najít nejkratší cestu z s_{start} do libovolného vrcholu u tím, že se budeme pohybovat od aktuálního vrcholu s, začínajícího na vrcholu u, na jakéhokoliv předchůdce s', který minimalizuje g(s') + c(s,s'), dokud nedosáhneme vrcholu s_{start}. Nicméně, LPA* neudělá každý vrchol lokálně konzistentní. Místo toho, používá heuristiku zaměřenou na vyhledávání a aktualizuje pouze g-hodnoty, které jsou relevantní pro vypočtení nejkratší cesty. A* udržuje OPEN a CLOSED seznamy. CLOSED seznam se stará o to, aby A* nereexpandoval vrcholy. LPA* neudržuje CLOSED seznam, protože používá kontrolu pomocí lokálních konzistencí, aby se zabránilo reexpandováni vrcholů. Seznam OPEN je prioritní fronta, která umožňuje A* vždy expandovat okrajový vrchol s nejmenší hodnotou funkce f. LPA* také udržuje

prioritní frontu pro tento účel. Její prioritní fronta obsahuje vždy pouze lokálně nekonzistentní vrcholy. Hodnoty funkce k(také nazývané key - klíč) vrcholů v prioritních frontě zhruba odpovídají hodnotám funkce f použité v A*, a LPA* vždy přepočítá g-hodnotu vrcholu ("expanduje vrchol") v prioritní frontě s nejmenší hodnotou funkce k. To je podobné A*, který vždy expanduje vrchol v prioritní frontě s nejmenší hodnotou funkce f. Expanzí vrcholu máme na mysli vykonávaní příkazů {10-16} (čísla v závorkách odkazují na čísla řádků na Obr. 8). Hodnotou funkce k(s) vrcholu s je vektor o dvou složkách:

$$k(s) = [k_1(s), k_2(s)] \quad (4)$$

kde $k_1(s) = \min(g(s), rhs(s)) + h(s)$ a $k_2(s) = \min(g(s), rhs(s))$. Priorita vrcholu v prioritní frontě je vždy stejná jako jeho hodnota funkce k. Klíče jsou srovnány podle lexikografického uspořádání. Například klíč k(s) je menší nebo roven klíči k'(s), právě tehdy, když $k_1(s) < k'_1(s)$ nebo ($k_1(s) = k'_1(s)$ a zároveň platí $k_2(s) \leq k'_2(s)$). První složka z klíče $k_1(s)$ odpovídá hodnotě $f(s) = g^*(s) + h(s)$, kterou používá A*, protože obě g-hodnoty a rhs-hodnota LPA* odpovídají g-hodnotě A* a hodnota h z LPA* odpovídají hodnotě h z A*. Druhá složka klíče $k_2(s)$ odpovídá g-hodnotě z A*. LPA* vždy expanduje vrchol v prioritní frontě s nejmenší hodnotou k_1, která odpovídá hodnotě f z A* s preferováním vrcholu s nejmenší hodnotou k_2, která odpovídá hodnotě g z A* vyhledávání. To je podobné A*, který vždy expanduje vrchol v prioritní frontě s nejmenší hodnotou f, s preferováním vrcholu s nejmenší hodnotou g. Výsledné chování LPA* a A* je také podobné. Klíče expandovaných vrcholů v LPA* neklesají v průběhu času. To je podobné k A*, kde hodnoty f expandovaných vrcholů jsou také neklesající v průběhu času, a pokud A* se rozhoduje mezi vrcholy se stejnou hodnotou f ve prospěch menších hodnot g, tak [f(s), g(s)] také neklesá v průběhu času (protože všichni potomci expandovaného vrcholu mají striktně větší g-hodnoty, než samotný expandovaný vrchol). [5]

3.2.4 Algoritmus

LPA* je znázorněn na Obr. 8 . Hlavní metoda Main () nejprve volá Initialize () k inicializaci problému plánování cesty {17}. Initialize () nastaví počáteční g-hodnoty všech vrcholů na nekonečno a stanoví jejich rhs-hodnoty podle rovnice Chyba: zdroj odkazu nenalezen{03-04}. Tím pádem je zpočátku s_{start} jediný lokálně nekonzistentní vrchol a je vložen do jinak prázdné prioritní fronty s klíčem vypočítaným podle rovnice (4) {05}. Tato inicializace zaručuje, že první zavolání ComputeShortestPath() provádí přesně A* hledání, to znamená, že expanduje přesně stejné vrcholy jako A* v přesně stejném pořadí. Všimněte si, že ve skutečném provedení, initialize () potřebuje pouze inicializovat vrchol pokud na něj narazí při vyhledávání, a proto není nutné inicializovat všechny vrcholy dopředu. To je důležité proto, že počet vrcholů může být velký, a jen minimum z nich by mohlo být prozkoumáno v průběhu vyhledávání. LPA* pak čeká na změny v ceně hran {20}. Pro zachování invariantů (1-3), když došlo ke změnám v cenách některých hran, se volá UpdateVertex () {23} který aktualizuje hodnoty rhs a klíče vrcholů potenciálně ovlivněných změnami v cenách hran, tak i jejich členství v prioritní frontě, v souladu s tím jak se stávají lokálně konzistentními nebo nekonzistentními, a nakonec přepočítá nejkratší cestu {19} zavoláním funkce ComputeShortestPath(), která opakovaně expanduje lokálně nekonzistentní vrcholy v pořadí jejich priorit {10}.

Lokálně nekonzistentní vrchol s je nazýván lokálně nad-konzistentní pokud platí $g(s) > rhs(s)$. Pokud ComputeShortestPath() expanduje lokálně nad-konzistentní vrchol {12-13}, pak se nastaví g-hodnota vrcholu na jeho rhs hodnotu {12}, čímž se stane vrchol lokálně konzistentní. Lokálně nekonzistentní vrchol s je nazývám lokálně pod-konzistentní pokud platí $g(s) < rhs(s)$. Pokud ComputeShortestPath() expanduje lokálně pod-konzistentní vrchol {15-16}, pak se prostě nastaví g-hodnota vrcholu na nekonečno {15}. Tím se stane vrchol buď lokálně konzistentní nebo nad-konzistentní. Pokud expandovaný vrchol byl místně nad-konzistentní, pak změna jeho g-hodnoty může mít vliv na lokální konzistenci všech jeho následníků {13}. Podobně, pokud expandovaný vrchol byl lokálně pod-konzistentní, pak i jeho nástupci mohou být ovlivněni {16}. Pro uchování

invariantů zavoláme funkci ComputeShortestPath(), která aktualizuje rhs-hodnoty těchto vrcholů, kontroluje jejich lokální konzistenci, a přidává nebo je odstraní z prioritní fronty {06-08}.

LPA* expanduje vrcholy až do doby, kdy je s_{goal} lokálně konzistentní a klíč vrcholu k expanzi není menší než klíč s_{goal}. To je podobné k A*, který expanduje vrcholy, dokud neexpanduje s_{goal}. V tento okamžik by se g-hodnota s_{goal} měla rovnat jeho vzdálenosti od počátku a f-hodnota vrcholu k expanzi není menší než f-hodnota s_{goal}. Pokud po skončení hledání $g(s_{goal}) = \infty$, tak žádná cesta z s_{start} do s_{goal} neexistuje. Jinak je možné zjistit nejkratší cestu od s_{start} do s_{goal} tím, že se vždy pohybujeme od aktuálního vrcholu s, počínaje v s_{goal}, na toho předchůdce s' který minimalizuje g(s') + c(s,s') dokud nedosáhneme vrcholu s_{start}. To je podobné tomu, co může udělat A*, pokud nepoužívá zpětné ukazatele (backpointers).[5]

3.2.5 Možné modifikace a optimalizace

Optimalizace algoritmu LPA* je podrobně popsána v [5]. V následující podkapitole si popíšeme několik jednoduchých způsobů, jak optimalizovat LPA *, které nezmění, které vrcholy nebo v jakém pořadí LPA* expanduje, nebo v jakém pořadí se rozšiřuje je.

- Někdy je vrchol odstraněn z prioritní fronty, jen aby byl ihned s jiným klíčem vložen zpět. Například, vrchol můžeme odstraněn na {07} a pak znovu vložen na {08}. V tomto případě je mnohem účinnější vrchol v prioritní frontě ponechat, aktualizovat jeho klíč a to pouze změnit jeho pozice.
- Když UpdateVertex () na řádku {13} počítá hodnoty rhs pro následníky lokálně nad-konzistentního vrcholu je zbytečné, aby hledal minimální hodnotu mezi všemi svými předchůdci. Je plně dostačující vypočítat hodnotu rhs jako minimum ze své staré hodnoty rhs a součet nové hodnota g, lokálně nad-konzistentního vrcholu a ceny hrany vedoucí z lokálně nad-konzistentního vrcholu na jeho následníka.
- Když UpdateVertex () na řádku {1} počítá hodnoty rhs pro následníky lokálně pod-konzistentního vrcholu, jediná hodnota g, která se změnila je hodnota g lokálně pod-konzistentního vrcholu. Vzhledem k tomu že se zvýšila, hodnota rhs jeho následníka může být touto změnou ovlivněna pouze pokud, jeho stará hodnota rhs byla vypočítána za pomoci lokálně pod-konzistentního vrcholu. Toto může být použito k rozhodování, zda hodnoty rhs následníku musí být přepočítány nebo ne.
- Druhá a třetí optimalizace se týkají výpočty hodnoty rhs následníků vrcholu, po tom co jeho hodnota g byla změněna. Podobné optimalizace mohou být použity pro výpočet hodnoty rhs vrcholu, po tom co byla změněna jedna z jeho vstupních hran.
- Nakonec můžeme zavést novou proměnnou p(s), která splňuje rhs(s) = g(p(s)) + c(p(s), s), pro všechny vrcholy s, čímž se zbavíme některých výpočtů. Například, nyní můžeme napsat "if ($s \neq s_{start}$ a p(s) = u) místo více těžkopádné podmínky "if ($s \neq s_{start}$ a rhs(s) = g(u) + c(u, s))."

Existuje množství různých modifikací a algoritmů vycházejících z LPA*. Mezi nejvíce rozšířené patří D* Lite, je popsán v této práci v oddílu 3.4.
Druhou, velmi mírnou modifikací je algoritmus GLPA *, který je podrobně popsán v [13]. Tento algoritmus kombinuje LPA* s nedeterministickým Minimaxem. V praxi se používá jako základ pro vývoj nových algoritmů pro rychlé heuristické přeplánovače. Algoritmus jeho dopředné verze je uveden na Obr. 9 .

procedure Initialize()
{01} $U = \emptyset$;
{02} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{03} $rhs(s_{start}) = 0$;
{04} UpdateState(s_{start});

procedure UpdateState(u)
{05} if ($u \neq s_{start}$) $rhs(u) = \min_{s \in S, a \in A(s): succ(s,a)=u}(g(s) + c(s, a, u))$;
{06} if ($u \in U$ and $g(u) \neq rhs(u)$) U.Update($u, K(u)$);
{07} else if ($u \in U$ and $g(u) = rhs(s)$) U.Remove(u);
{08} else if ($u \notin U$ and $g(u) \neq rhs(u)$ and NotYet(u)) U.Insert($u, K(u)$);

procedure ComputePlan()
{09} while (U.TopKey() $< K(s_{goal})$ or $rhs(s_{goal}) \neq g(s_{goal})$)
{10} u = U.Pop();
{11} if ($g(u) > rhs(u)$)
{12} $g(u) = rhs(u)$;
{13} for all $s \in Succ(u)$ UpdateState(s);
{14} else
{15} $g(u) = \infty$;
{16} for all $s \in Succ(u) \cup \{u\}$ UpdateState(s);

procedure Main()
{17} Initialize();
{18} forever
{19} ComputePlan();
{20} for all inconsistent states $s \notin U$ U.Insert($s, K(s)$);
{21} Wait for changes in action costs;
{22} for all actions with changed action costs $c(u, a, v)$
{23} Update the action cost $c(u, a, v)$;
{24} UpdateState(v);

Obr. 9 GLPA - dopředná verze [13].*

3.3 Metoda Dynamic A*

Algoritmus se jmenuje D*, protože to se podobá A*, až na to, že je dynamický v tom smyslu, že ceny hran se mohou změnit v průběhu pohybu po nalezené cestě. Za předpokladu, že pohyb robota je správně propojen s pře-plánovacím procesem, je zaručeno , že výsledná cesta bude optimální.

3.3.1 Basic D*

Definice

Prostředí může být formulováno jako soubor stavů, označujících lokaci robota a propojených hranami, z nichž každá má přiřazeny náklady pohybu. Robot začíná v určitém stavu a pohybuje se podél hran do jiných stavů, dokud nedosáhne cílového stavu, označeného jako G. Každý stav X s výjimkou G má zpětný ukazatel (backpointer) do dalšího stavu Y označeného b(X) = Y. D* používá zpětné ukazatele k reprezentaci cesty k cíli. Cena pohybu po hranách od stavu Y do stavu X je kladné číslo přidělené k hraně funkcí c(X, Y). Pokud stav Y nemá hranu vedoucí do stavu X, potom funkce c(X, Y) není definovaná. Stavy X a Y jsou sousedy ve zkoumaném prostředí, pokud máme definovanou funkci c(X, Y) nebo c(Y, X).

Stejně jako A*, také D* udržuje seznam stavů OPEN. Seznam OPEN se používá k šíření informací o změnách ceny hran a pro výpočet nákladů cesty do jednotlivých stavů. Každý stav X má přiřazenu značku t(X). Tyto značky se přiřazují podle následujícího pravidla: t(X) = NEW pokud X nikdy nebylo na seznamu OPEN, t(X) = OPEN jestliže X je v současné době na seznamu OPEN, a t(x) = CLOSED jestliže X je již není na seznamu OPEN. Pro každý stav X, D* udržuje odhad součtu ceny hran z X do G dán funkcí ceny cesty h(G, X). Za vhodných podmínek tento odhad je ekvivalentem k

optimální (minimální) ceně cesty ze stavu X do stavu G. Pro každý stav X na seznamu OPEN (tj. t(X) = OPEN), klíčová funkce k(G, X) je definována jako minimum z hodnoty h(G, X) před její modifikací a hodnoty h(G, X) v době, kdy byl stav X umístěn na seznam OPEN. Klíčová funkce klasifikuje stav X na seznamu OPEN na jeden ze dvou typů: stav RAISE, pokud k(G, X) < h(C, X) a LOWER stav, pokud k(G, X) = h(G, X). D * používá stav RAISE k označení, že cena cesty vzrostla a stav LOWER k označení k poklesu ceny cesty. Tento proces probíhá opakovaným odstraňováním stavů ze seznamu OPEN. Pokaždé, když stav je odstraněn ze seznamu, je expandován z důvodu předání informace o změně nákladů na své sousedy. Tito sousedé jsou zase přidáni do seznamu OPEN, kde se celý proces opakuje.

Stavy na seznamu OPEN jsou seřazeny podle hodnot jejich klíčových funkcí. Parametr $k_{mín}$ je definován jako minimální k(X) pro všechny stavy X takové, že t(X) = OPEN. Parametr $k_{mín}$ představuje důležitou limitní hodnotu. Cesty, jejichž cena je menší nebo rovná $k_{mín}$ jsou optimální, a ty dražší než $k_{mín}$ nemusí být optimální. Parametr k_{old} je definován tak, aby se rovnal $k_{mín}$ před posledním odstraněním stavu ze seznamu OPEN. Nebyly-li žádné stavy odstraněny, k_{old} není definován.

Pořadí stavů označené jako $\{X_P, X_N\}$ je definováno jako sekvence, za předpokladu že platí $b(X_{i+1}) = X_i$ pro všechna i, pro která platí $1 \leq i < N$, a $X_i \neq X_j$ pro všechna (i, j), která splňují $1 \leq i < j \leq N$. Takže tato sekvence definuje cestu zpětných ukazatelů z X_N do X_1. Sekvence $\{X_1, X_N\}$ je označena jako monotónní, pokud (t(X) = CLOSED a $h(G, X_i) < h(G, X_{i+1})$) nebo ($t(X_i)$= OPEN a $k(G, X_i) < h(G, X_{i+1})$) pro všechna i taková, že platí $1 \leq i < N$. D* buduje a udržuje monotónní posloupnost {G, X} reprezentující snižování cen cesty pro každý stav X, který je nebo byl na seznamu OPEN. Vzhledem k sekvenci stavů $\{X_1, X_N\}$ stav X_i je předek stavu X_j, pokud platí $1 \leq i < j \leq N$, a je následníkem stavu X_j, pokud platí $1 \leq j < i \leq N$.

Pro dvoustavovou funkci zahrnující cílový stav se používají následující zkratky: h(X) ≡ h(G, X) a (X) ≡ k(G, X). Podobně pro sekvenci se používá {X} = {G, X}. Zápis f(º) se používá pro označení funkce, která je nezávislá na její doméně.[6]

Algoritmus

Základní D* algoritmus se převážně skládá ze dvou funkcí: *PROCESS-STATE* a MODIFY-COST. PROCESS-STATE je použita pro výpočet optimální ceny cesty k cíli, a MODIFY-COST se používá ke změně ceny hrany nákladové funkce c(º) a vložení ovlivněných stavů do seznamu OPEN. Algoritmus pro PROCESS-STATE je na Obr. 10 a algoritmus pro MODIFY-COST je uveden na Obr. 11 . Mezi dále použité funkce patří MIN(a, b), která vrací minimum dvou skalárních hodnot a, b; Less(a, b), která vrací TRUE, pokud a je menší než b a FALSE v ostatních případech; COST(X), která vrací h(X) pro stavX; MIN-STATE, která vrací stav na seznamu OPEN s minimálním k(º) hodnotou (NULL pokud je seznam prázdný); MIN-VAL, která se vrací $k_{mín}$ v seznamu OPEN (NO-VAL, pokud je seznam prázdný); DELETE(X), kterou se odstraní stav X ze seznamu OPEN a nastaví t(X) = CLOSED; INSERT(X, h_{new}), která vypočítá k(X) = h_{new} pokud t(X) = NEW, k(X) = MIN(k(X), h_{new}) je-li t(X) = OPEN a k(X) = MIN (h(X), h_{new}), pokud t(X) = CLOSED. Dále nastaví h(X)= h_{new} a t(X) = OPEN a vloží nebo aktualizuje stav X v seznamu OPEN seřazeném podle k(º).

Funkce LESS a COST jsou používány raději než < a h(º), protože jejich sémantika je redefinována v podkapitole 3.3.2 pro práci s vektory místo skaláry.

Ve funkci PROCESS-STATE (viz Obr. 10) na řádcích LI až L3, stav X s nejnižší hodnotou k(º) je odstraněn z OPEN seznamu. Jestliže X je LOWER stav (tj. k(X) = h(X)), jeho cesta cena je optimální, protože h(X) je rovna starému k_{min} Na řádcích L8 až L13 prozkoumáme pro každé Y, které je sousedem stavu X, zda cena cesty mezi nimi nemůže být snížena. Kromě toho sousední stavy, které mají značku t(X) = NEW obdrží počáteční cenu cesty a změny ceny cesty jsou provedeny na každém sousedovi Y, který má zpětného ukazatele na X, bez ohledu na to, zda nová cena je větší nebo menší než stará. Protože tyto stavy jsou potomci X, jakékoli změny ceny cesty do stavu X ovlivňují také i ceny jejich cest. V případě potřeby jsou zpětné ukazatele stavu Y přesměrovány tak, aby byla získána monotónní posloupnost {Y}. Všichni sousedé, kteří obdrží novou cenu cesty, jsou umístěni do seznamu OPEN, a tím i oni promítnou změnu ceny cesty do svých sousedů. [6]

Function: PROCESS-STATE ()

L1 X = MIN-STATE( )
L2 if X = NULL then return NO-VAL
L3 k_{old} = k(X) ; DELETE(X)
L4 if k_{old} < h(X) then
L5 for each neighbor Y of X :
L6 if t(s) ≠ NEW and h(s) ≤ k_{old} and h(X) > h(s) + c(s, X) then
L7 b(X) = Y ; h(X) = h(s) + c(s, X)
L8 if k_{old} = h(X) then
L9 for each neighbor Y of X:
L10 if t(s) = NEW or
L11 (b(s) =X and h(s) ≠ h(X)+c(X,Y) or
L12 (b(s) ≠ X and h(s)>h(x)+c(X,Y))then
L13 b(s) = X; INSERT(s, h(X) + c(X, Y))
L14 else
L15 for each neighbor Y of X :
L16 if t(s) = NEW or
L17 (b(s) =X and h(s)≠ h(X)+c(X,Y) then
L18 b(s) = X ; INSERT(s, h(X) + c(X, Y))
L19 else
L20 if b(s)≠X and h(s) > h(X)+c(X, Y) and t(X) = CLOSED then
L21 INSERT(X, h(X))
L22 else
L23 if b(s) ≠ X and h(X)>h(s)+c(s,X) and
L24 t(s) = CLOSED and h(s) > k_{old} then
L25 INSERT(s, h(s))
L26 return MIN- VAL( )

Obr. 10 D - funkce Process-State() [6].*

Jestliže X je RAISE stav, jeho cena cesty nemusí být optimální. Před tím, než stav X předá změnu ceny jeho sousedům, na řádcích L4 až L7 přezkoumáme jeho optimální sousedy, abychom zjistili, zda nemůžeme snížit hodnotu h(X). Na řádcích L15 až L18 jsou změny cen předány NEW stavům a přímým následníkům stejným způsobem jako u LOWER stavů. Pokud stav X je schopen snížit cenu cesty stavu, který není přímým následníkem (řádky L20 a L21), X je umístěn zpět na seznam OPEN pro budoucí expanzi. Tato akce je nutná, aby se zabránilo vytvoření uzavřené smyčky ve zpětných ukazatelích. Pokud je možné snížit cenu cesty X pomocí suboptimálního souseda (řádky L23 až L25), je soused umístěn zpět do OPEN seznamu. Tím pádem je aktualizace "odložena" do doby až soused má optimální cenu cesty.

Ve funkci MODIFY-COSt(zobrazené na Obr. 11), se cena cesty hrany aktualizuje na změněnou hodnotu. Vzhledem k tomu že se cena cesty pro stav Y změní, stav X je umístěn na seznamu OPEN. Když je X expandován pomocí funkce PROCESS-STATE, tak se vypočítá nové h(s) = h(x) + c(X, Y) a Y se umístí do seznamu OPEN. Dodatečné expandování stavu předá cenu cesty následníkům Y.

Function: MODIFY-COSt(X, Y, cval)
LI *c(X, Y) 0* = cval
L2 if t(X) = CLOSED then INSERT(X, h(x))
L3 return MIN- VAL()

Obr. 11 D - funkce Modify-Cost(X, Y, cval) [6].*

Funkce MOVE-ROBOT ukazuje, jak se používá PROCESS-STATE a MODIFY-COST k

přesunu robota optimální cestou ze stavu S do stavu G. Jak nám ukazuje Obr. 12 na řádcích LI až L3 t(º) = NEW pro všechny stavy, h(G) je nastavena na nulu, a G je uveden na seznamu OPEN. Funkce PROCESS-STATE je potom opakovaně volána (na řádcích L5 a L6), dokud nezískáme první cestu vedoucí ke stavu, ve kterém se nachází robot(tj.. t(S) = CLOSED), nebo není zjištěno že žádná cesta neexistuje (tj. val = NO-VAL a t(S) = NEW). Robot pak následuje zpětné ukazatele v pořadí {R}, dokud se nedosáhne cíle, nebo se nezjistí nesrovnalosti (řádky L10 a L11) mezi detekovanou cenou cesty s(º) a známou cenou cesty c(º) (např. z důvodu zjištěné překážky). Všimněte si, že tyto rozdíly se mohou objevit kdekoli, nejen v posloupnosti {R}. MODIFY - COST je volán k nápravě c(º) a ovlivněné stavy vloží do seznamu OPEN. PROCESS-STATE je pak opakovaně volán na řádku L13 až do doby, kdy bude platit val ≥ h(R) z důvodů změny cen cesty a případného vypočítání nové posloupnosti {R} vedoucí k cíli. Robot nadále sleduje zpětné ukazatele v posloupnosti směrem k cíli. Funkce vrací GOAL- REACHED, když je cíl nalezen a NO – PATH, je-li nedostupný.

Function: MOVE-ROBOt(S, G)
L1 for each state X in the graph:
L2 t(X) = NEW
L3 INSERT(G, 0)
L4 val = 0
L5 while t(S) ≠ CLOSED and ≠ NO - VAL
L6 val = PROCESS-STATE()
L7 if t(S) = NEW then return NO- PATH
L8 R = S
L9 while R ≠ C ;
L10 for each(X, Y) such that s(X, Y) ≠ c(X, Y):
L11 **v**al = MODIFY - COST(X, Y, s(X, Y))
L12 while LESS(val, COST(R)) and val ≠ NO-VAL
L13 va1 = PROCESS-STATE( )
L14 R = b(R)
L15 return GOAL-REACHED

Obr. 12 D - funkce Move-Robot(S, G) [6].*

Je třeba poznamenat, že řádek L7 v MOVE-ROBOT jen detekuje, že neexistuje žádná posloupnost hran od robota k cíli (například je-li rozpojen graf). Nezjistí však případ, kdy všechny cesty k cíli jsou zablokovány překážkami. Abychom zajistit správnou detekci těchto případů, je možné hranám přiřadit velké kladné hodnoty pro překážky, a průchodným hranám mohou být přiřazeny malé kladné hodnoty. Hodnota pro překážky by měla být zvolena tak, aby byla větší než nejdelší možná cesta. Pokud po ukončení smyčky na řádku L5 je h(S) ≥ hodnotě překážky, tak problém hledání cesty nemá řešení. Stejně tak platí, že problém hledání cesty nemá řešení, když po ukončení smyčky na řádku L12 je h(R) ≥ hodnotě překážky.[6]

3.3.2 Focused D*

Tato metoda vychází z verze Basic D* s následujícími změnami a je to jedna z nejpopulárnějších metod, jelikož kombinuje efektivnost heuristických a inkrementálních metod.[8]

Definice

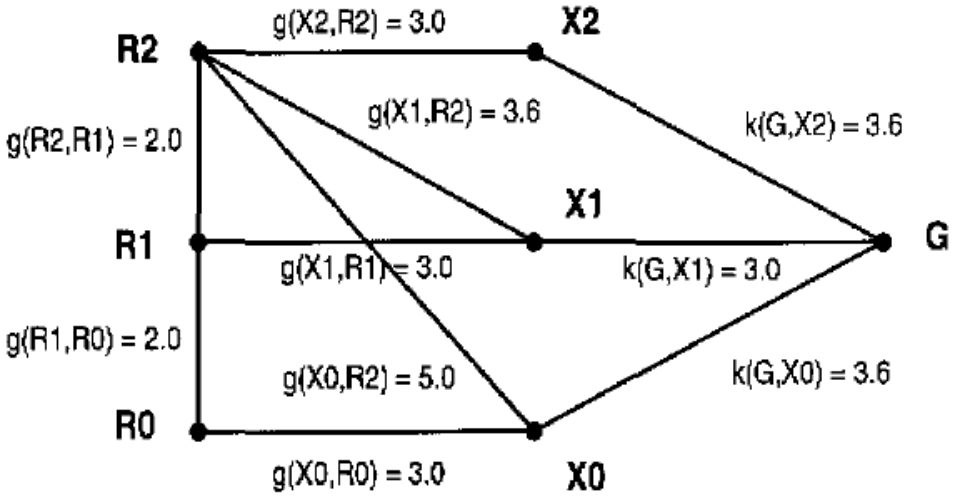
Stavy jsou seřazeny v seznamu OPEN podle hodnoty f(º), získané z $f_{b(}X_i, R_i)$, kde X je stav v seznamu OPEN a R_i je stav na kterém se nacházel robot v době, kdy X byla vložena do seznamu OPEN. Nechť {Ro, R_1,..,R_N} je sekvence stavů obsazených robotem v dobách, kdy byly stavy X přidány do OPEN seznamu pomocí funkce MODIFY-COST. Hodnota f_B(º) je dána vztahem

$f_{b(}X_i, R_i) = f(X_i, R_i) + d\ (R_1, R_0)$,

kde d(º) je funkce vzniklého zkreslení daná takto:

$d(R_i, R_0) = g(R_1, R_0) + g(R_2, R_1) + ... + g(R_i, R_{i-1}) = 0$ pro $i > 0$ a
$d(R_0, R_0) = 0$ pro $i = 0$

Stavy jsou v OPEN seznamu vzestupně seřazeny podle $f_b(º)$, přičemž při stejných hodnotách $f_b(º)$ jsou seřazeny podle rostoucích hodnot f(º) a při stejných hodnotách f(º) jsou seřazeny podle rostoucích hodnot k(º). A vazby k(º) jsou řazeny libovolně, což znamená, že vektor hodnot [$f_b(º)$,f(º) ,k(º)] je uložen společně s každým stavem na seznamu. [6]



Obr. 13 Ukázka výpočtu f(º) [6].

Správné uspořádání:

f(X1,R2) = k(G,Xl) + g(Xl,R2)=6.6
f(X2,R2) = k(G,X2) + g(X2,R2) = 6.6
f(X0,R2) = k(G,XO)+ g(XO,R2) = 8.6

Počáteční zkreslené uspořádání:

fB(X0,RO) = k(G,XO) + g(XO,R0) = 6.6
fB(X1,Rl) = k(G,Xl) + g(X1,Rl) + g(R1,RO) = 8.0
fB(X2,R2) = k(G,X2) + g(X2,R2) + g(R2,Rl) + g(R1,RO) = 10.6

Upravené zkreslené uspořádání:

fB(X1,R2) = k(G,Xl) + g(X1,R2) + g(R2,Rl) + g(R1,RO) = 10.6
fB(X2,R2) = k(G,X2) + g(X2,R2) + g(R2,Rl) + g(R1,RO) = 10.6
fB(X0,R2) = k(G,XO) + g(XO,R2) + g(R2,Rl) + g(R1,RO) = 12.6

Vezměme příklad na Obr. 13 . Robot začíná na pozici R_0 a přidá stav X_0 do seznamu OPEN. Pak se posune na R_1 a vloží X_1 do seznamu. Konečně se posune do R_2 a X_2 se vloží do seznamu. Na pozici R_2 pak expanduje tři stavy na seznamu OPEN. Pokud hodnoty f(º) tří stavů byly přepočítávány s robotem na pozici R_2, tak by správné seřazení stavů na OPEN seznamu mělo být $\{X_1, X_2, X_0\}$. Všimněte si, že vazby v hodnotách f(º) pro X_1 a X_2 , jsou řazeny ve prospěch menšího k(º). Jelikož stavy byly do seznamu vloženy na různých lokacích, seřazení podle hodnoty $f_b(º)$ je $\{X_0, X_1, X_2\}$. První stav odstraněný ze seznamu je X_0. Jeho hodnota $f_b(º)$ se změní z $f_b(X_0, R_0)$ na $f_b(X_0, R_2)$, a pak se vrátí zpět na seznam s upravenou hodnotou. Další odstraněný stav je X_1. Jeho hodnota $f_b(º)$ se změní z $f_b(X_1, R_1)$ na $f_b(X_1, R_2)$ a pak je vrácen zpět na seznam s upravenou hodnotou. Další odstraněný stav je opět X_1, protože má stejné hodnoty $f_b(º)$ a f(º) jako X_2, ale má nižší hodnotu k(º). Vzhledem k tomu, že má správnou hodnotu $f_b(º)$ (tj. vypočítanou na současné pozici robota, R_2) je tento stav expandován. Všimněte si, že toto je první stav, který by měl být expandován. Další odstraněný stav je X_2. Jelikož ten už má správnou hodnota $f_b(º)$, je expandován. A konečně stav X_0 je odstraněn. Jelikož i jeho hodnota $f_b(º)$ byla správně nastavena pro novou pozici robota, je expandován. A tím tyto tři stavy byly expandovány ve správném pořadí.

Pro graf reprezentovaný osmi-směrovým kartézským polem stavů, dobrou heuristikou, která splňuje monotónní omezení, je heuristika daná součinem vzdálenosti a minimální ceny hran. Pokud je

kartézské pole indexováno pomocí (i, j), nechť (x_1, x_2) jsou (i, j) souřadnice stavu X. Nechť je C_{min} minimální cena c(º) v poli potom, co všechny ceny přechodů jsou normalizovány vzhledem k délce nediagonální hrany. Nechť je $d_i = |x_i - y_i|$ a $d_j = |x_j - y_j|$. Pak $g(X, Y) = C_{min} (\sqrt{2}\, d_j + d_i - d_j)$ pokud platí $d_i \geq d_j$ a $g(X, Y) = C_{min} (\sqrt{2} d_i + d_j - d_i)$ je-li $d_i < d_j$.

Nechť R_{cur} je poslední stav robota, na němž byly zjištěny rozpory mezi snímačem dat a mapou, a nechť R_{prev} je předchozí takovýto stav. Oba jsou inicializovány k počátečnímu stavu robota. Funkce stavu robota r(X) vrací stav robota, kdy byl stav X naposledy vložen nebo upraven na seznamu OPEN. Parametr d_{curr} je vzniklé zkreslení od počáteční stavu robota do jeho současného stavu; je to zkratka pro d (R_{curr}, R_0) a je inicializován jako d_{curr} = d (R_0, R_0) = 0. Nechť je f_{min} minimální hodnotou f(º) na seznamu OPEN a k_{val} je odpovídající hodnotou k(º). Následující těsnopisný záznam se používá pro $f_{b(}$º), f(º) a g(º) : $f_{b(}X) \equiv f(X, r(X))$, $f(X) \equiv f(X, r(X))$ a $g(X) \equiv g(X, r(X))$.[6]

Rozšíření algoritmu

Většina rozšíření algoritmu je omezena na funkce pro porovnání ceny cesty a správu seznamu OPEN. Proto funkce COST, LESS, INSERT, MIN-STATE, a MIN-VAL jsou ovlivněny a budou pozměněny. COST(R) bude místo h(R) pro pozici robota R vracet vektor hodnot [f(R,R_{curr}), h(R)]. Funkce Less(a, b) bude místo porovnání dvou skalárů porovnávat vektory [a_1, a_2] a [b_1, b_2]. Funkce Less vrací True pokud $a_1 < b_1$ nebo ($a_1 = b_1$ a $a_2 < b_2$), jinak vrací False.

Předtím, než budou upraveny funkce INSERT, MIN-STATE, a MIN-VAL, se zavedou dvě nové funkce. PUT-STATE (X) nastaví t(X) = OPEN a vloží X do seznamu OPEN podle vektoru [$f_{b(}$X), f(X), g(X)] a funkce GET-STATE vrací stav ze seznamu OPEN s minimální hodnotou vektoru (NULL pokud je seznam prázdný).

Upravená funkce INSERT je uvedena na Obr. 14 . Na řádku L1 stav s robotem R, se kterým se pracuje v MOVE-ROBOT, je uložen jako nové kontaktní místo pro vyhledávání. Na řádcích L2 až L4 je prozkoumán současný stav robota, aby se zjistilo, zda se robot od posledního vložení stavu do seznamu OPEN pohyboval. Pokud ano, d (º) je aktualizován přidáním dolní meze ceny cesty od předchozího stavu robota na aktuální stav(d (R_{curr}, R_0) = d (R_{prev}, R_0) + g(R_{curr}, R_{prev})). Hodnoty pro h(X) a k(X) jsou zjištěny na řádcích L5 až L11. Zbývající dvě hodnoty ve vektoru jsou spočítány na řádku L12, a stav je vložen na L13.

Function: INSERt(X, hnew)
L1 $R_{curr} = R$
L2 if $R_{curr} \neq R_{prev}$ then
L3 $d_{curr} = d_{curr} + g(R_{curr}, R_{prev})$
L4 $R_{prev} = R_{curr}$
L5 if t(X) = NEW then $k(X) = h_{new}$
L6 else
L7 if t(X) = OPEN then
L8 $k(X) = MIN(k(X), h_{new})$
L9 DELETE(X)
L10 else $k(X) = MIN(h(X), h_{new})$
L11 $h(X) = h_{new}$; $r(X) = R_{curr}$
L12 $f(X) = k(X) + g(X)$;$f_{b(}X) = f(X) + d_{curr}$
L13 PUT-STATE(X)

Obr. 14 D - funkce INSERT(X, hnew) [6].*

Funkce MIN-STATE uvedená na Obr. 15 , vrací stav ze seznamu OPEN s minimální hodnotu f(º). Aby toto provedla, funkce načte stav ze seznamu OPEN s nejnižší hodnotu $f_{b(}$º). Pokud stav byl zařazen na seznam OPEN v době, když byl robot na předchozí lokaci (řádek L2), pak je znovu vložen do seznamu OPEN na řádku L3. Touto operací dosáhneme toho, že opravíme vychýlení stavu za

pomoci současného stavu robota, přičemž ponecháme stavu jeho h(º) a k(º) hodnoty beze změny. MIN-STATE pokračuje v načítání stavů ze seznamu OPEN, dokud nenajde ten, který má aktualizovanou odchylku (tj. byl umístěn na seznam OPEN s robotem na jeho současné pozici).

Function: MIN-STATE 0
L1 while X = GET-STATE() ≠ NULL
L2 if r(x) ≠ R_{curr} then
L3 h_{new}= h(X); h(X) = k(X); DELETE(X); lNSERT(X, h_{new})
L4 else return X
L5 return NULL

Obr. 15 D - funkce Min-State() [6].*

Upravená funkce MIN-VAL uvedená na Obr. 16 vrací hodnoty f(º) a k(º) stavu na seznamu OPEN, jenž má nejmenší hodnotu f(º).

Function: MIN-VAL()
L1 X = MIN-STATE()
L2 if X=NULL then return NO-VAL
L3 else return <f(X),k(X)>

Obr. 16 *D* - funkce Min-Val() [6].*

Funkce PROCESS-STATE a MODIFY-COST se syntakticky neliší od jejich popisu daného v oddíle 3.3.1. Ale protože obě funkce končí zavoláním funkce MIN-VAL, tak navrací vektor [f_{min},k_{val}] místo k_{min}. Tento vektor je přiřazen do proměnné val v průběhu MOVE-ROBOT, a je použit předefinovanou funkcí LESS ke zjištění, zda PROCESS-STATE byl zavolán v dostatečném počtu pro zaručení optimality. Vzhledem k tomu, že pro stav robota R, který je přepočítáván, platí R = R_{curr} , pak g(R, R) = 0 a f(R, R) = h(R). Z toho vyplývá, že optimalita je zaručena pro stav R, pokud platí, že f_{min} > h(R) nebo (f_{min} = h(R) a $k_{val} \geq$ h(R)).[6]

3.4 D* Lite

V kapitole 3.2 jsme si popsali LPA*,které opakovaně určuje nejkratší cesty mezi počátečním vrcholem a cílovým vrcholem zatímco se mění ceny hran grafu. Jeho první vyhledávání je totožné s vyhledáváním A*, ale následné vyhledávání znovu využívá informace z předchozích vyhledávání. D* Lite vychází z LPA* a opakovaně určuje nejkratší cestu mezi současným vrcholem, na kterém se nachází robot, a cílovým vrcholem v závislosti na tom, jak se mění ceny hran grafu, zatímco se robot pohybuje směrem k cílovému vrcholu. D* Lite nedělá žádné předpoklady o tom, jak se bude měnit cena hrany, ať už poroste nahoru nebo bude klesat, zda se bude odehrávat v blízkosti současného vrcholu robota nebo daleko od něj, nebo zda se změna odehrává proto, že se změnilo prostředí, ve kterém se robot pohybuje či pouze se změnily znalosti robota o něm. Problém cílem řízené navigace v neznámém terénu pak je zvláštním případem tohoto problému, kde graf je osmi-směrová mřížka, ceny jejíchž hran jsou zpočátku rovny jedné a změní se na nekonečno, když robot najde překážku. Nejdříve bude popsaná jednoduchá verze D* Lite a pak sofistikovanější verze.[8]

3.4.1 První verze

Již bylo řečeno, že mnohé vzdálenosti k cíli zůstávají beze změny jak se robot pohybuje do cílového vrcholu a také během tohoto procesu poznamenává překážky. Takže můžeme použít verzi LPA* pro problém cílem řízené navigace v neznámém terénu. Verze prezentovaná na Obr. 8 vyhledává od počátečního vrcholu do cílového vrcholu. Její g-hodnoty jsou odhady vzdálenosti od počátku. Potřebujeme tedy upravit směr hledání tak, aby g-hodnoty byly odhady vzdálenosti do cíle.

Takováto verze LPA* prohledává od cílového vrcholu k počátečnímu vrcholu a může být získána tak, že se obrátí všechny hrany z původního grafu a zamění se jeho počáteční a cílový vrchol. Heuristika h(s,s') nyní musí být kladné a zpětně konzistentní číslo, které splňuje vztahy $h(s_{start},s_{start})=0$ a $h(s_{start},s) \leq h(s_{start},s')+c(s',s)$ pro všechny vrcholy $s \in S$. Obecně řečeno, jelikož se robot pohybuje a tím se změní počáteční vrchol, heuristika potřebuje splnění tohoto předpokladu pro všechna $s_{start} \in S$. Pokud po skončení hledání $g(s_{start})= \infty$, pak neexistuje žádná konečná cesta od počátečního k cílovému vrcholu. Jinak můžeme sledovat nejkratší cestu od počátečního vrcholu do cílového vrcholu tak, že se vždy budeme pohybovat od současného vrcholu s, počínaje od počátečního vrcholu, do libovolného následníka s', který bude mít nejmenší c(s,s') + g(s') dokud nebude dosaženo cílového vrcholu. Chceme-li vyřešit problém cílem řízené navigace v neznámém terénu, funkce CalcKey(), Initialize(), UpdateVertex() a ComputeShortestPath() mohou zůstat beze změny. Nicméně funkce Main() potřebuje rozšířit tak, že bude pohybovat robotem a pak přepočítá klíče vrcholů v prioritní frontě. To je nutné z důvodu, že při pohybu robota se mění heuristika, protože je počítána s ohledem na současný vrchol robota. Toto však pouze změní klíče vrcholů v prioritní frontě, ale ne lokální konzistenci vrcholů a to, zda jsou v prioritní frontě. Na Obr. 17 je uvedena výsledná vyhledávací metoda, nazývaná D* Lite First Version.

```
procedure CalcKey(s)
{01'} return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s))];

procedure Initialize()
{02'} U = ∅;
{03'} for all s ∈ S rhs(s) = g(s) = ∞;
{04'} rhs(s_goal) = 0;
{05'} U.Insert(s_goal, CalcKey(s_goal));

procedure UpdateVertex(u)
{06'} if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'));
{07'} if (u ∈ U) U.Remove(u);
{08'} if (g(u) ≠ rhs(u)) U.Insert(u, CalcKey(u));

procedure ComputeShortestPath()
{09'} while (U.TopKey() < CalcKey(s_start) OR rhs(s_start) ≠ g(s_start))
{10'}    u = U.Pop();
{11'}    if (g(u) > rhs(u))
{12'}       g(u) = rhs(u);
{13'}       for all s ∈ Pred(u) UpdateVertex(s);
{14'}    else
{15'}       g(u) = ∞;
{16'}       for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);

procedure Main()
{17'} Initialize();
{18'} ComputeShortestPath();
{19'} while (s_start ≠ s_goal)
{20'}    /* if (g(s_start) = ∞) then there is no known path */
{21'}    s_start = arg min_{s'∈Succ(s_start)}(c(s_start, s') + g(s'));
{22'}    Move to s_start;
{23'}    Scan graph for changed edge costs;
{24'}    if any edge costs changed
{25'}       for all directed edges (u, v) with changed edge costs
{26'}          Update the edge cost c(u, v);
{27'}          UpdateVertex(u);
{28'}       for all s ∈ U
{29'}          U.Update(s, CalcKey(s));
{30'}       ComputeShortestPath();
```

Obr. 17 D Lite: first version [8].*

Hlavní funkce Main() první verze D* Lite nejdříve volá Initialize() pro inicializaci problému

prohledávání{17'}. Initialize() nastaví počáteční g-hodnoty všech vrcholů do nekonečna a stanoví jejich hodnoty rhs v souladu s ekvivalentem (2). Takže zpočátku cílový vrchol je jediný lokálně nekonzistentní vrchol a je vložen do jinak prázdné prioritní fronty s klíčem vypočítaným podle vzorce uvedeného dříve {05'}. Je třeba podotknout, že vlastní provedení funkce Initialize() je potřebné k inicializaci vrcholu až když se s ním setká při vyhledávání, a proto není nutné inicializovat všechny vrcholy dopředu. To je důležité, protože počet vrcholů může být velký, a jen málo z nich by může být dosaženo v průběhu vyhledávání. První verze D* Lite pak počítá nejkratší cestu z aktuálního vrcholu, ve kterém se nachází robot s_{start}, do cílového vrcholu {18'}. Pokud robot ještě nedosáhl cílového vrcholu {19'}, tak se posune o jeden krok podél nejkratší cesty a aktualizuje s_{start}, aby odrážel aktuální vrchol, na kterém se nachází robot {21'-22'}. (V pseudokódu, jsme zahrnuli komentář popisující, jak robot může odhalit, že neexistuje žádná cesta, ale nepředepisuje, co by měl dělat v tomto případě. Například v problému cílem řízené navigace v neznámém statickém terénu by se měl zastavit a oznámit, že neexistuje žádná cesta, protože překážky nezmizí. Pak vyhledává změny v cenách hran {23'} a při změně hran pro zachování invariantů (1-3) zavolá UpdateVertex() {27'}, který aktualizuje hodnoty rhs a klíčů potenciálně ovlivněných vrcholů, stejně jako jejich členství v prioritní frontě, pokud se stanou lokálně konzistentní nebo nekonzistentní. Nakonec aktualizuje klíče všech vrcholů v prioritní frontě {28'-29'}, přepočítá nejkratší cestu {30'} a pak celý postup opakuje. Platí následující věta [8]:

Věta 1: ComputeShortestPath() první verze D* Lite expanduje každý vrchol nejvýše dvakrát, a to nejvýše jednou, když je lokálně nekonzistentní a nejvýše jednou, když je lokálně nad-konzistentní, a tím se ukončí. Pak můžeme následovat nejkratší cestu od počátečního vrcholu do cílového vrcholu tak, že se vždy pohybujeme z aktuálního vrcholu s, počínaje počátečním vrcholem, do libovolného následníka s', který bude mít minimální c(s,s') + g(s'), dokud nedosáhneme cílového vrcholu.
Důkazy ke všem větám a ostatním vlastnostem uvedených v tomto oddíle jsou uvedeny v [7].

3.4.2 Druhá verze

První verze D* Lite má tu nevýhodu, že opakované přeřazování prioritní fronty {28'-29'} může být náročné, protože prioritní fronta často obsahuje velké množství vrcholů. Druhá verze D* Lite, uvedená na Obr. 18 , používá vyhledávací metody odvozené z D* (viz 3.3), abychom se vyhnuli neustálému přeřazování prioritní fronty. Rozdíly oproti první verzi D* Lite jsou vyznačeny tučně. Heuristika h(s,s') nyní musí být nezáporná a dopředně-zpětně konzistentní, to znamená že musí splňovat $h(s,s'') \leq h(s,s') + h(s',s'')$ pro všechny vrcholy $s,s',s'' \in S$. Rovněž musí být přípustná bez ohledu na to, který vrchol je cílovým vrcholem. To znamená, že musí splňovat $h(s,s') \leq c^*(s,s')$ pro všechny vrcholy $s,s' \in S$, kde $c^*(s,s')$ označuje cenu nejkratší cesty z vrcholu $s \in S$ k vrcholu $s' \in S$. Heuristiky s těmito vlastnostmi také splňují vlastnosti, které musí splnit heuristiky první verze D* Lite [7].

procedure CalcKey(s)
{01"} return [min(g(s); rhs(s)) + h($s_{star,}$ s) + $\mathbf{k_m}$; min(g(s), rhs(s))];
procedure Initialize()
{02"} U = Ø;
{03"} $\mathbf{k_m = 0;}$
{04"} for all $s \in S$ rhs(s) = g(s) = ∞;
{05"} rhs(s_{goal}) = 0;
{06"} U.Insert(sgoal; CalcKey(sgoal));

procedure UpdateVertex(u)
{07"} if ($u \neq s_{goal}$) rhs(u) = $\min_{s' \in Succ(u)}$(c(u; s') + g(s'));
{08"} if ($u \in U$) U.Remove(u);
{09"} if (g(u) ≠ rhs(u)) U.Insert(u, CalcKey(u));

procedure ComputeShortestPath()

```
{10"} while (U.TopKey() <CalcKey(s_start) OR rhs(s_star) ≠ g(s_start))
{11"}   k_old = U.TopKey();
{12"}   u = U.Pop();
{13"}   if (k_old< CalcKey(u))
{14"}           U.Insert(u, CalcKey(u));
{15"}   else  if (g(u) > rhs(u))
{16"}           g(u) = rhs(u);
{17"}           for all s ∈ Pred(u) UpdateVertex(s);
{18"}   else
{19"}           g(u) = ∞;
{20"}           for all s ∈ Pred(u) ∪ UpdateVertex(s);

procedure Main()
{21"} s_last = s_start;
{22"} Initialize();
{23"} ComputeShortestPath();
{24"} while ( s_star t≠ s_goal)
{25"}   /* if (g(s_start) = ∞) then there is no known path */
{26"}   s_start = arg mins'∈Succ( sstart )(c( s_start, s') + g(s'));
{27"}   Move to s_start;
{28"}   Scan graph for changed edge costs;
{29"}   if any edge costs changed
{30"}           km = km + h(s_last, s_start);
{31"}           s_last = s_start;
{32"}           for all directed edges (u, v) with changed edge costs
{33"}                   Update the edge cost c(u, v);
{34"}                   UpdateVertex(u);
{35"}           ComputeShortestPath();
```

Obr. 18 D Lite: second version [8].*

Druhá verze D* Lite používá klíče, jež jsou dolními hranicemi klíčů, které používá první verze D* Lite pro odpovídající vrcholy. Inicializuje je stejným způsobem jako první verze D* Lite. Poté co se robot posunul z vrcholu s k nějakému vrcholu s', kde zjistil změny v cenách hran, první složka klíče mohla být snížena nejvýše o h(s,s'). Druhá složka klíče nezávisí na heuristice a tak zůstává beze změny. Takže aby byla zachována nižší hranice, D* Lite musí odečíst h(s,s') od první složky klíče všech vrcholů v prioritní frontě. Nicméně protože h(s,s') je stejná pro všechny vrcholy v prioritní frontě, takže pořadí vrcholů v prioritní frontě se nezmění pokud hodnota h(s,s') není odečtena. Když potom jsou počítány nové klíče, jejich první komponenty jsou příliš malé v porovnání k ostatním klíčům v prioritní frontě. Takže je třeba přidat h(s,s') k jejich první komponentě. Pokud se robot znovu pohne a znovu zjistí změnu v ceně hrany, pak potřebujeme znovu přidat konstantu. Tohle se provádí pomocí proměnné k_m (tu nazýváme modifikátor klíče) {30"}. Proto vždy, když vypočítáme nové klíče, proměnná musí být přidána do jejich první komponenty, jak to bylo provedeno v {01"}. Tím pádem se po pohybu robota pořadí vrcholů v prioritní frontě nemění a prioritní fronta se nemusí přeřazovat. Na druhé straně, klíče jsou vždy dolními mezemi odpovídajících klíčů první verze D* Lite potom, co první komponenta klíče první verze D* Lite byla zvýšena o aktuální hodnotu k_m, tj. dolními mezemi hodnot vypočítaných pomocí CalcKey() {01"}. Využíváme této vlastnosti následující změnou ComputeShortestPath(). Po tom, co funkce ComputeShortestPath() odstranila vrchol u s nejmenším klíčem k_{old}=U.TopKey() z prioritní fronty {12"}, použije funkci CalcKey() pro výpočet klíče, který by měl mít. Pokud k_{old} < CalcKey(u) pak se odstraněný vrchol znovu vloží do fronty s klíčem vypočítaným pomocí CalcKey() {13"-14"}. Takže platí, že klíče všech vrcholů v prioritní frontě jsou dolní meze odpovídajících klíčů první verze D* Lite potom, co první složky klíčů první verze D* Lite byly zvýšeny o současnou hodnotu k_m. Pokud je $k_{old} \geq$ CalcKey(u), pak platí, že k_{old} = CalcKey(u),

jelikož k_{old} byla dolní hranice hodnoty vrácené CalcKey(). V tomto případě ComputeShortestPath() provádí stejné operace pro vrchol u jako ComputeShortestPath() první verze D* Lite {15"-20"}. Funkce ComputeShortestPath() provádí tyto operace pro vrcholy v přesně stejném pořadí jako ComputeShortestPath() z první verze D* Lite, což znamená, že druhá verze D* Lite sdílí mnoho vlastností s první verzí D* Lite, včetně její správnosti. Ve [8] byla dokázána následující větu:

Věta 2: ComputeShortestPath() druhé verze D* Lite expanduje vrchol nejvýše dvakrát, a to nejvýše jednou, když je lokálně pod-konzistentní, a nejvýše jednou, když je lokálně nad-konzistentní, a tím se ukončí. Pak můžeme následovat nejkratší cestu od počátečního vrcholu do cílového vrcholu tak, že se vždy pohybujeme z aktuálního vrcholu s, počínaje počátečním vrcholem, do libovolného následníka s', který bude mít minimální c(s,s') + g(s'), dokud nedosáhneme cílového vrcholu. [8]

3.4.3 Optimalizace

Optimalizace algoritmu D*Lite, a to jak první tak i druhé verze, je obdobná optimalizaci algoritmu LPA*, která je popsaná v této práci v kapitole 3.2.5 nebo více podrobně v [5]. Obě optimalizované verze algoritmu D*Lite, jsou zmíněny v [7] a [8], kde jsou porovnány s jinými metodami. Ukázka pseudokódu optimalizované druhé verze algoritmu D* Lite je uvedena na Obr. 19 .

procedure CalcKey(s)
{01'''} return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))]$;
procedure Initialize()
{02'''} $U = \emptyset$;
{03'''} $k_m = 0$;
{04'''} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{05'''} $rhs(s_{goal}) = 0$;
{06'''} U.Insert$(s_{goal}, [h(s_{start}, s_{goal}); 0])$;
procedure UpdateVertex(u)
{07'''} if $(g(u) \neq rhs(u)$ AND $u \in U)$ U.Update$(u, CalcKey(u))$;
{08'''} else if $(g(u) \neq rhs(u)$ AND $u \notin U)$ U.Insert$(u, CalcKey(u))$;
{09'''} else if $(g(u) = rhs(u)$ AND $u \in U)$ U.Remove(u);
procedure ComputeShortestPath()
{10'''} while (U.TopKey() $\dot{<}$ CalcKey(s_{start}) OR $rhs(s_{start}) > g(s_{start}))$
{11'''} u = U.Top();
{12'''} k_{old} = U.TopKey();
{13'''} k_{new} = CalcKey(u);
{14'''} if$(k_{old} \dot{<} k_{new})$
{15'''} U.Update(u, k_{new});
{16'''} else if $(g(u) > rhs(u))$
{17'''} $g(u) = rhs(u)$;
{18'''} U.Remove(u);
{19'''} for all $s \in$ Pred(u)
{20'''} $rhs(s) = \min(rhs(s), c(s, u) + g(u))$;
{21'''} UpdateVertex(s);
{22'''} else
{23'''} $g_{old} = g(u)$;
{24'''} $g(u) = \infty$;
{25'''} for all $s \in$ Pred$(u) \cup \{u\}$
{26'''} if $(rhs(s) = c(s, u) + g_{old})$
{27'''} if $(s \neq s_{goal})$ $rhs(s) = \min_{s' \in \text{Succ}(s)}(c(s, s') + g(s'))$;
{28'''} UpdateVertex(s);
procedure Main()
{29'''} $s_{last} = s_{start}$;
{30'''} Initialize();
{31'''} ComputeShortestPath();
{32'''} while $(s_{start} \neq s_{goal})$
{33'''} /* if $(rhs(s_{start}) = \infty)$ then there is no known path */
{34'''} $s_{start} = \arg\min_{s' \in \text{Succ}(s_{start})}(c(s_{start}, s') + g(s'))$;
{35'''} Move to s_{start};
{36'''} Scan graph for changed edge costs;
{37'''} if any edge costs changed
{38'''} $k_m = k_m + h(s_{last}, s_{start})$;
{39'''} $s_{last} = s_{start}$;
{40'''} for all directed edges (u, v) with changed edge costs
{41'''} $c_{old} = c(u, v)$;
{42'''} Update the edge cost $c(u, v)$;
{43'''} if $(c_{old} > c(u, v))$
{44'''} $rhs(u) = \min(rhs(u), c(u, v) + g(v))$;
{45'''} else if $(rhs(u) = c_{old} + g(v))$
{46'''} if $(u \neq s_{goal})$ $rhs(u) = \min_{s' \in \text{Succ}(u)}(c(u, s') + g(s'))$;
{47'''} UpdateVertex(u);
{48'''} ComputeShortestPath();

Obr. 19 D Lite: second version (optimized version) [8].*

3.5 Anytime Prohledávání

Algoritmy Anytime plánování velmi rychle najdou první, možná velmi sub-optimální řešení a pak neustále pracují na jeho zlepšení, dokud nevyčerpají plánovací čas. Myšlenka Anytime plánování byla navržena v AI komunitě již před nějakým časem, a hodně práce bylo uděláno na vývoji algoritmů pro Anytime plánování.

Nicméně výrazně méně práce bylo věnováno aplikaci algoritmů Anytime plánování na grafově založené vyhledávání. Jednoduchým a docela běžným způsob jak transformovat libovolný vyhledávací algoritmus na Anytime plánovací algoritmus, je iterační zvyšování oblasti prohledávaného stavového prostoru. Ze začátku se prohledává malá oblast stavového prostoru v okolí současného stavu pro řešení, které vypadá nejslibněji v závislosti na odhadech vzdálenosti k cíli pro stavy na okraji regionu a cenách na dosažení těchto stavů (tyto druhy vyhledávání jsou obyčejně nazývány jako real-time vyhledávání). Potom, co získáme počáteční řešení, iterativně zvyšujeme prohledávaný prostor dokud nevyčerpáme všechen čas, který máme k dispozici, nebo prohledávaný region se rozrostl tak, že pokrývá celý stavový prostor. Takové hledání obvykle vykazuje dobré výsledky v libovolné doméně. Jedná se zejména o vyhledávání v oblastech, kde nalezení jakékoliv kompletní cesty je obtížné v rámci stanoveného časového rozsahu a nalezení částečné cesty je přijatelným řešením. Bohužel tyto algoritmy typicky neposkytují žádné mezní hodnoty kvality jejich řešení a mohou dokonce navrhnout cesty, jež vedou k selhání v oblastech, které mají stavy s nevratnými podmínkami (např. jednosměrná cesta, která vede do slepé uličky).[9]

3.5.1 A* s váženou heuristikou

Normálně A* pracuje se vstupní heuristikou h(s), která musí být konzistentní. To znamená, že platí $h(s) \leq c(s, s') + h(s')$ pro všechny následníky s' vrcholu s, je-li $s \neq s_{goal}$, a $h(s) = 0$, pokud $s = s_{goal}$. Funkce c(s, s') označuje cenu hrany od vrcholu s k vrcholu s' a musí být kladná. Konzistence zaručuje, že heuristika je přípustná: h(s) není nikdy větší než skutečná cena dosažení cíle z vrcholu s. Navyšování heuristiky (tzn. použití $\varepsilon * h(s)$ pro $\varepsilon > 1$) často vyústí do snížení počtů stavů k expanzi a tedy rychlejšího vyhledávání. Nicméně navyšování heuristiky může také porušit přípustnost a z toho důvodu není zaručeno, že řešení bude optimální. Pseudokód A* s navyšováním heuristiky je uveden na Obr. 20 pro snadné srovnání s dalšími algoritmy jako ARA* (viz 3.5.2) a AD* (viz 3.5.3) .

A* má dvě funkce na převod hodnot ze stavů do reálných čísel: g(s) je cena ceny cesty od počátečního vrcholu do vrcholu s a $f(s) = g(s) + \varepsilon * h(s)$ je odhad celkové vzdálenosti od počátečního vrcholu do cílového pokud cesta prochází vrcholem s. A* také udržuje prioritní frontu OPEN, ve které jsou stavy k expanzi. Fronta OPEN je seřazena podle f(s), takže A* vždy expanduje stavy, které se zdají být na nejkratší cestě od počátečního vrcholu do cílového. A* inicializuje seznam OPEN s počátečním stavem s_{start} {02}. Pokaždé, kdy expanduje stav s {04 - 11}, odstraní s z fronty OPEN a pak aktualizuje hodnoty g všech sousedů vrcholu s. Pokud se sníží g(s'), tak vloží s' do fronty OPEN. A* se ukončí, jakmile expandujeme cílový stav.

Nastavením ε na 1 dosáhneme standardního A* s nenavýšenou heuristikou a získané řešení má zaručenou optimalitu. Pro $\varepsilon > 1$ může řešení být sub-optimální, ale sub-optimalita je ohraničena faktorem ε . Délka nalezeného řešení není větší než ε-násobek délky optimálního řešení. [12]

```
01 g(s_start) = 0; OPEN = Ø;
02 insert s_start into OPEN with f(s_start) = ε * h(s_start);
03 while( s_goal is not expanded)
04       remove s with the smallest f-value from OPEN;
05       for each successor s' of s
06       if s' was not visited before then
07                f(s') = g(s') = ∞;
08       if g(s') > g(s) + c(s, s')
09                g(s') = g(s) + c(s, s');
```

```
10          f(s') = g(s') + h(s');
11          insert s' into OPEN with f(s');
```

Obr. 20 A* s heuristikou váženou faktorem $\varepsilon \geq 1$ [12].

3.5.2 Anytime Repairing A* (ARA*)

Základní princip

Algoritmus ARA* funguje na principu několikanásobného prohledávání pomocí A*, kdy začíná s velkou hodnotou ε, kterou snižuje až do doby kdy ε = 1. Tím zaručíme, že po každém vyhledávání a získání řešení budeme mít optimální faktor ε. Ovšem spouštění A* kompletně od začátku pokaždé, když se nám sníží ε by bylo velmi nákladné. Nyní si vysvětlíme, jakým způsobem ARA* opakovaně využívá výsledků z předchozího vyhledávání.

Nejdříve si zavedeme pojem lokální nekonzistence. Stav se nazývá lokálně nekonzistentní, když je snížena jeho hodnota g (řádek 09, Obr. 20) a to až do doby, kdy je znovu expandován. Předpokládejme že stav s je nejlepším předchůdcem stavu s' : to znamená, že

$g(s') = \min_{s'' \in pred(s')} (g(s'') + c(s'', s')) = g(s) + c(s, s')$.

Potom pokud g(s) klesá, dostaneme $g(s') \min_{s'' \in pred(s')} > (s') (g(s'') + c(s'', s'))$. Jinými slovy, pokles g(s) zavádí lokální nekonzistenci mezi hodnotou g vrcholu s a hodnotou g jeho následníků. Na druhou stranu, kdykoliv je vrchol s expandován, tak se jeho nekonzistence opraví přepočítáním hodnoty g jeho nástupců,(řádek 08-09, Obr. 20). Tím se ale zase stanou lokálně nekonzistentními následníci vrcholu s. Tím se lokální nekonzistence předává následníkům vrcholu pomocí série expanzí. Eventuálně následníci přestanou býti závislí na vrcholu s, jejich g hodnoty přestanou být snižovány, a přestanou být vkládáni do seznamu OPEN. Vzhledem k této definici lokální nekonzistence je zřejmé, že seznam OPEN se skládá pouze ze všech lokálně nekonzistentních stavů: pokaždé, kdy je hodnota g snížena, stav je vložen do seznamu OPEN a pokaždé, když je stav expandován, tak je ze seznamu OPEN zase odstraněn až do doby, kdy jeho hodnota g bude zase snížena. Z toho důvodu může být seznam OPEN vnímán jako soubor stavů, z nichž musíme šířit lokální nekonzistenci.

U vyhledávání A* s konzistentní heuristikou je zaručeno, že nebudeme expandovat žádný stav více než jednou. Nicméně nastavením ε > 1 můžeme porušit konzistenci a z toho důvodu A* může expandovat některé stavy vícekrát. Ovšem když omezíme počet expanzí každého stavu pouze na jednou, pak suboptimalita ε pořád platí. Aplikaci tohoto omezení provedeme kontrolou kteréhokoliv stavu jehož hodnota g je snížena, a vložíme jej do fronty OPEN pouze tehdy, pokud dříve již nebyl expandován (řádek 10, Obr. 21). Seznam expandovaných stavů je udržována v CLOSED.

```
procedure fvalue(s)
01 return g(s) + ε * h(s);

procedure ImprovePath()
02 while(fvalue(s_goal) > min_{s∈OPEN}(fvalue(s)))
03      remove s with the smallest fvalue(s) from OPEN;
04      CLOSED = CLOSED ∪ {s};
05      for each successor s' of s
06              if s' was not visited before then
07                      g(s') = 1;
08      if g(s') > g(s) + c(s, s')
09              g(s') = g(s) + c(s, s');
10              if s' ∉ CLOSED
11                      insert s' into OPEN with fvalue(s');
12              else
13                      insert s' into INCONS;
```

procedure Main()
01’ $g(s_{goal}) = \infty$; $g(s_{start}) = 0$;
02’ *OPEN* = *CLOSED* = *INCONS* = Ø;
03’ insert s_{start} into *OPEN* with fvalue(s_{start});
04’ ImprovePath();
05’ $\varepsilon' = \min(\varepsilon, g(s_{goal}) / \min_{s \in OPEN \cup INCONS}(g(s)+h(s)))$;
06’ publish current ε'-suboptimal solution;
07’ while ε' > 1
08’ decrease ε;
09’ Move states from *INCONS* into *OPEN*;
10’ Update the priorities for all $s \in OPEN$ according to fvalue(s);
11’ *CLOSED* = Ø;
12’ ImprovePath();
13’ $\varepsilon' = \min(\varepsilon, g(sgoal) / \min_{s \in OPEN \cup INCONS}(g(s)+h(s)))$;
14’ publish current ε' -suboptimal solution;

*Obr. 21 Pseudokód ARA** [12].

S tímto omezením budeme expandovat každý stav maximálně jednou, ale seznam OPEN již nemusí obsahovat všechny lokálně nekonzistentní stavy. Ve skutečnosti bude obsahovat pouze lokálně nekonzistentní stavy, které ještě nebyly expandovány. Je však důležité mít přehled o všech lokálně nekonzistentních stavech, jelikož to budou výchozí body pro šíření nekonzistence pro budoucí vyhledávání. Toto se řeší tím, že se udržuje seznam INCONS, ve kterém jsou uvedeny všechny lokálně nekonzistentní stavy, které nejsou v OPEN (řádky 12-13, Obr. 21). Tak spojením INCONS a OPEN získáme seznam všech lokálně nekonzistentních stavů, který může být použit jako výchozí bod pro šíření nekonzistencí před každou novou vyhledávací iterací.

Jediným dalším rozdílem mezi funkcí ImprovePath a vyhledáváním A* je ukončovací podmínka. Vzhledem k tomu, že funkce ImprovePath používá výsledky z předchozích vyhledávání, s_{goal} se nemusí nikdy stát lokálně nekonzistentní, a proto nemusí být nikdy vložen do fronty OPEN. Tím ukončovací podmínka vyhledávání A* přestává platit. A* vyhledávání nicméně je také možné ukončit jakmile $f(s_{goal})$ je rovna minimální hodnotě f mezi všemi stavy na OPEN seznamu. To je podmínka, kterou používáme ve funkci ImprovePath (řádek 02, Obr. 21). Ta nám také zabraňuje expanzi stavu s_{goal} spolu s dalšími možnými stavy se stejnou hodnotou f. Všimněte si že ARA* už nadále neuchovává hodnoty f jako proměnné, protože mezi voláními funkce ImprovePath se změní ε, a bylo by neúměrně nákladné aktualizovat hodnoty f všech stavů. Místo toho funkce fvalue (s) počítá a navrací hodnoty f pouze pro stavy v seznamu OPEN a s_{goal}.

Nyní si přiblížíme funkci Main, která opakovaně vykonává prohledávání. Provede inicializace a pak opakovaně volá funkci ImprovePath s řadou klesajících ε. Před každým zavoláním funkce ImprovePath je vytvořen nový seznam OPEN tím, že se do něj přesune seznam INCONS. Vzhledem k tomu, že OPEN seznam musí být seřazen podle aktuální hodnoty f jeho stavů, je také znovu seřazen (řádky 09'-10 ', Obr. 21). Takže po každém zavolání funkce ImprovePath dostaneme řešení, které je sub-optimální nejvýše podle faktoru ε.

Jak je uvedeno v [10] faktor sub-optimality může také být počítán jako poměr mezi $g(s_{goal})$, což dává horní mez nákladů na optimální řešení, a minimální neváženou hodnotou f lokálně nekonzistentních stavů, což nám dává dolní mez ceny optimálního řešení. (Tohle platí pouze, pokud ε je větší nebo rovno jedné. Jinak $g(s_{goal})$ je již optimálním řešením.) Proto je skutečná hodnota sub-optimality ARA* vypočítána jako minimum mezi ε a poměrem (řádky 05' a 13', Obr. 21). Na první pohled může být použití této skutečné sub-optimality při rozhodování o tom, jak snížit ε mezi iteracemi vyhledávání lákavé (např. nastavením $\varepsilon = \varepsilon' - \Delta$), ale experimenty však naznačují že snižování ε v malých krocích je stále výhodnější. To z toho důvodu, že malý pokles ε má často za následek zlepšení řešení, a to navzdory skutečnosti, že skutečná hodnota sub-optimality z předchozího řešení byla již podstatně nižší než hodnota ε. Na druhé straně velký pokles ε může často vést během příštího vyhledávání k expanzi příliš velkého množství stavů. (Dalším užitečným doporučením z [10], které zatím nebylo dosud implementováno do ARA*, je prořezávání seznamu OPEN tak, aby nikdy

neobsahoval stavy, jejichž nevážená hodnota f je větší nebo rovna g(s_{goal}).) Během průběhu funkce ImprovePath šetříme výpočetní náklady tím, že znovu neexpandujeme stavy, které byly lokálně konzistentní a jejichž g-hodnoty byly správné již před zavoláním funkce ImprovePath. [11]

3.5.3 Anytime D*

Jak je uvedeno v předchozích oddílech, existují algoritmy pro efektivní zvládání dynamických prostředí (např. D* a D* Lite) a komplexních plánovacích problémů (ARA*). Nicméně existují případy, kdy čelíme komplexním plánovacím problémů v dynamickém prostředí.

Jako příklad uvažujme plánování pohybu robota ve frekventovaně používaných kancelářských prostorách. Plánovač pro takovýto úkol by v ideálním případě měl být schopen efektivně pře-plánovat pohyb robota, když nové informace naznačují, že bylo změněno prostředí. A také by mohlo být potřeba vytvářet sub-optimální řešení jelikož optimalita nemusí být vždy možná, zvlášť pokud na řešení problematiky máme omezený uvažovací čas.

Vzhledem k velké podobnosti mezi D* Lite a ARA* se zdálo vhodné podívat se na problematiku jejich sloučení do jednoho anytime inkrementálního algoritmu přeplánování, který by mohl splňovat požadavky uvedené v tomto příkladu.

A tak vznikl algoritmus Anytime Dynamic A* (AD*) [9]. Provádí řadu vyhledávání pomocí klesajícího inflačního faktoru pro získání řady řešení stejně jako ARA*. A když nastanou změny v prostředí, jež ovlivňují ceny hran v grafu, lokálně ovlivněné stavy jsou umístěny do fronty OPEN s prioritami rovnajícími se minimu z hodnoty jejich předchozího klíče a hodnoty jejich nového klíče, stejně jako u D* Lite. Stavy ve frontě jsou následně zpracovány do současného řešení, které je zaručeně ε-optimální.

Algoritmus

Algoritmus je uveden na Obr. 22 . Funkce Main nejprve nastaví inflační faktor ε na dostatečně vysokou hodnotu ε_0, takže první sub-optimální řešení je vygenerováno rychle (řádky 24 až 28). Pak, pokud nedojde ke změnám v cenách hran, funkce Main snižuje ε a zlepšuje kvalitu svých řešení, dokud není získáno optimální řešení, pro které platí ε = 1 (řádky 36 až 45). Tato fáze je přesně stejná jako u ARA*: pokaždé kdy ε je snížena, všechny nekonzistentní stavy jsou přesunuty z INCONS do OPEN a CLOSED je vyprázdněn.

Pokud jsou zjištěny změny v ceně hran, je zde velká šance, že stávající řešení již nebude ε-suboptimální. Pokud změny jsou významné, pak oprava stávajícího řešení pro znovuzískání ε-suboptimality může být výpočetně nákladná. V takovémto případě algoritmus zvýší ε, takže můžeme rychle najít nějaké méně optimální řešení (řádky 34 - 35). Vzhledem k tomu, že zvyšování ceny hran může způsobit, že některé stavy se stanou pod-konzistentní (tato možnost se v ARA* nevyskytuje) stavy musejí být vloženy do fronty OPEN s hodnotu klíče, která je minimem z jejich staré a nové ceny. Dále aby bylo zaručeno že pod-konzistentní stavy předají své nové ceny svým ovlivněným sousedům, jejich klíče musí použít neinflační heuristické hodnoty. Toto znamená, že klíče musí být vypočítány pro pod-konzistentní stavy jinak, než pro nad-konzistentní (řádky 01 – 04).

Začleněním těchto úvah je AD* schopen zpracovat jak změny v ceně hrany, tak změny v inflačním faktoru ε. Také může být mírně upraven tak, aby zvládal situace, kdy se počáteční stav s_{start} mění. Například kdy se po hledané cestě pohybuje agent. Pro aplikaci tohoto případu musíme nahradit řádek 29 s následujícími řádky:

29a. fork(MoveAgent());
29b. while ($s_{start} \neq s_{goal}$)

Funkce MoveAgent se provádí paralelně a sdílí proměnou s_{start} a kroky agenta podél aktuální cesty. Na každém kroku umožňuje funkci Main opravovat a zlepšovat výslednou cestu.

MoveAgent()

01. while ($s_{start} \neq s_{goal}$)
02. wait until a plan is available;
03. $s_{start} = \text{argmin}_{s \in Succ(sstart)}(c(s_{start}, s) + g(s))$;
04. move agent to s_{start};

Funkce $\text{argmin}_{s \in Succ(s)}$ f() vrací následníka s, který minimalizuje funkci f, takže v řádku 03 je stav s_{start} aktualizován na jednoho z jeho sousedů na stávající cestě k cíli. Vzhledem k tomu, že stavy ve frontě OPEN mají klíčové hodnoty přepočítávané pokaždé, když se změní ε, bude funkce automaticky zaměřena na aktualizovaný stav agenta s_{start}. Toto agentovi umožňuje pohodlně zlepšovat a aktualizovat výslednou cestu, zatímco se po ní pohybuje. [9][10]

key(s)
01. if(g(s) > rhs(s))
02. return [rhs(s) + ε * h(s_{start}, s); rhs(s)];
03. else
04. return [g(s) + h(s_{start}, s); g(s)];

UpdateState(s)
05. if s was not visited before
06. g(s) = ∞;
07. if($s \neq s_{goal}$) rhs(s) = $\min_{s' \in Succ(s)}$(c(s, s') + g(s'));
08. if(s ∈ OPEN) remove s from OPEN;
09. if(g(s) ≠ rhs(s))
10. if s ∉ CLOSED
11. insert s into OPEN with key(s);
12. else
13. insert s into INCONS;

ComputeorImprovePath()
14. while ($\min_{s \in OPEN}$(key(s)) < key(s_{start}) OR rhs(s_{start}) ≠ (s_{start}))
15. remove state s with the minimum key from OPEN;
16. if(g(s) > rhs(s))
17. g(s) = rhs(s);
18. CLOSED = CLOSED ∪ {s};
19. for all s' ∈ Pred(s) UpdateState(s');
20. else
21. g(s) = ∞;
22. for all s' ∈ Pred(s) ∪ {s} UpdateState(s');

Main()
23. g(s_{start}) = rhs(s_{start}) = ∞; g(s_{goal}) = ∞;
24. rhs(s_{goal}) = 0; ε= ε_0;
25. OPEN = CLOSED = INCONS = Ø;
26. insert s_{goal} into OPEN with key(s_{goal});
27. ComputeorImprovePath();
28. publish current ε-suboptimal solution;
29. forever
30. if changes in edge costs are detected
31. for all directed edges(u, v) with changed edge costs
32. Update the edge cost c(u, v);
33. UpdateState(u);
34. if significant edge cost changes were observed
35. increase ε or replan from scratch;

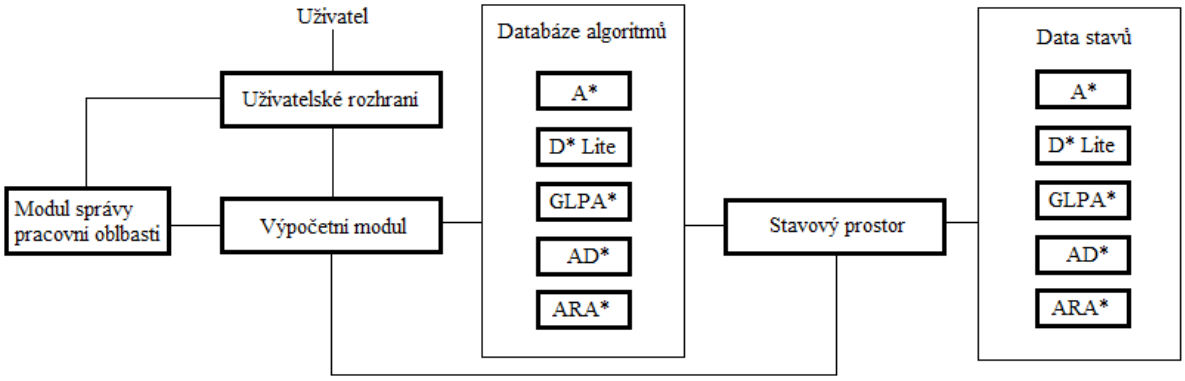
```
36.     else if ε > 1
37.             decrease ε ;
38.     Move states from INCONS into OPEN;
39.     Update the priorities for all s ∈ OPEN according to key(s);
40.     CLOSED = Ø;
41.      ComputeorImprovePath();
42.      publish current ε-suboptimal solution;
43.      if ε = 1
44.             wait for changes in edge costs;
```

Obr. 22 Pseudokód Anytime Dynamic A [9].*

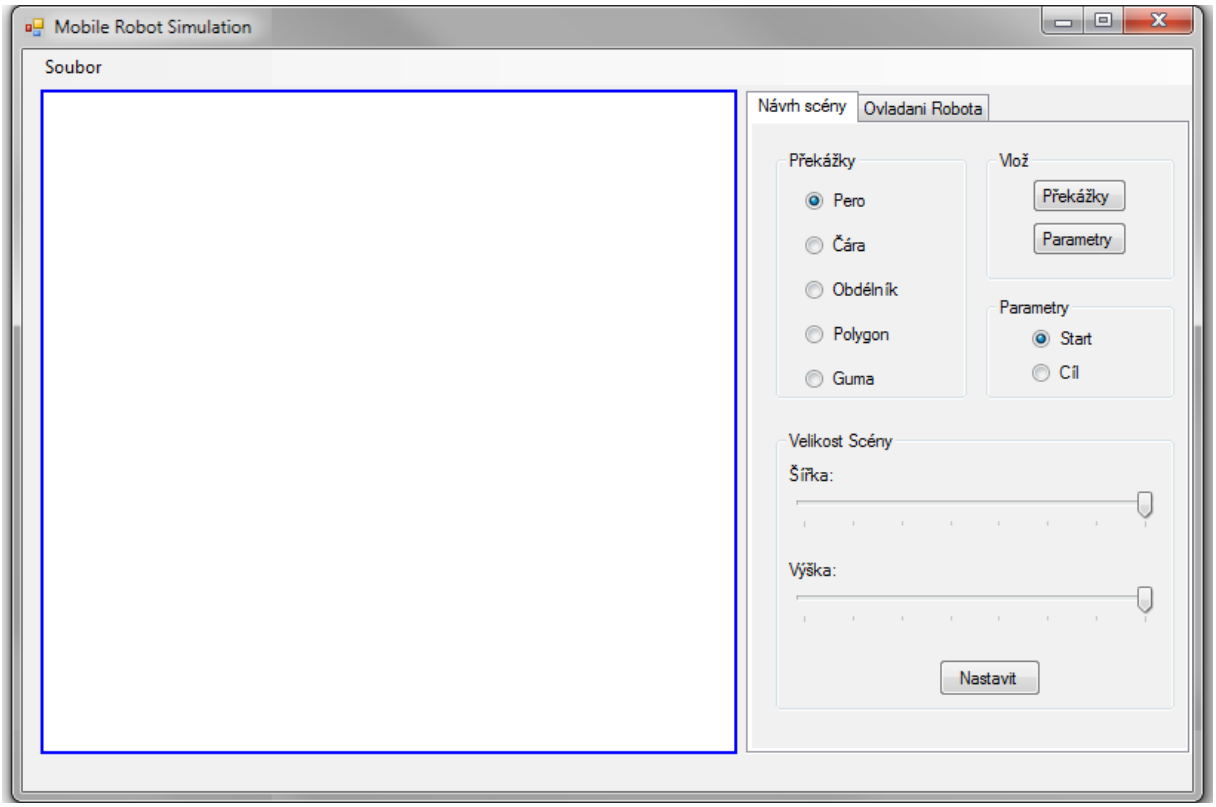
4 POPIS PROGRAMU

Simulační prostředí navigace robota bylo naprogramováno v prostředí Microsoft Visual C# 2008. Program byl vytvořen za použití objektově orientovaného programování a genericity. Záměrem bylo získání přehledného grafického prostředí s jednoduchým a intuitivním ovládáním.

Kód programu má rozsah cca 4300 řádků, a je součástí první přílohy. Vlastní struktura programu je uvedena na Obr. 23 .



Obr. 23 Struktura programu.



Obr. 24 Okno programu.

Simulační prostředí se skládá ze dvou hlavních částí:

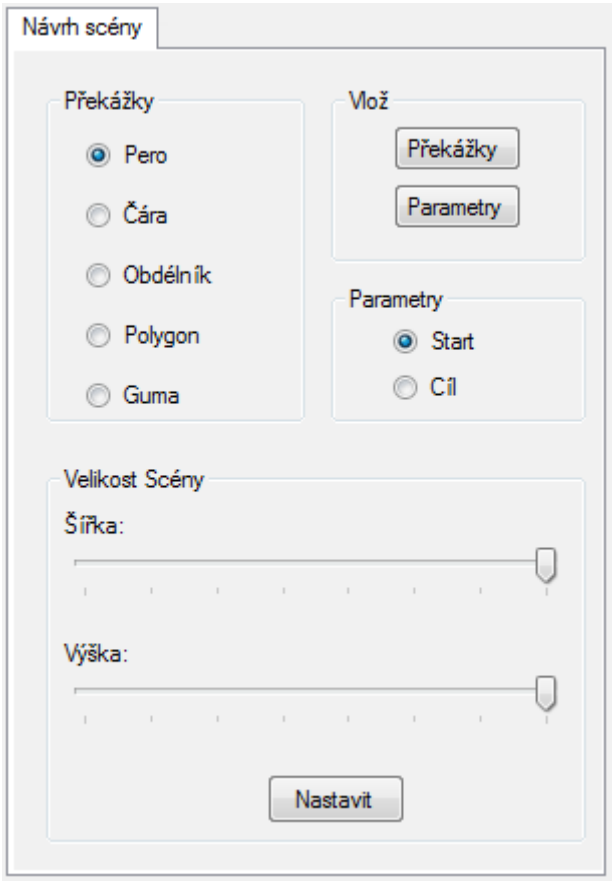
- Pracovní oblast (na levé straně), kterou program používá k vykreslení mapy prostředí, vkládání překážek, grafickému zobrazení prohledaných stavů a vyznačení nejkratší cesty robota. Velikost této plochy je volitelná z postranního panelu či menu. Pracovní plocha má volitelnou velikost až do 700x700 bodů.
- Na pravé straně jsou dva přepínatelné panely, které slouží k ovládání aplikace.

4.1 Práce s programem

Program se ovládá pomocí dvou panelů a jednoho menu.

Panel Návrh scény (viz Obr. 25) slouží k základní manipulaci s prostředím. Skládá se ze čtyř hlavních oddílů. Ve Vlož si můžeme vybrat, zda chceme vkládat vstupní parametry nebo manipulovat s překážkami. V sekci Parametry si zvolíme, zda chceme umístit startovní či cílovou pozici (samotné vložení probíhá kliknutím na zvolené místo v pracovní oblasti). Velikost scény slouží editaci rozměrů pracovní oblasti. A v sekci Překážky si můžeme zvolit typ překážky, který si přejeme vložit:

- Pero – Základní kreslicí nástroj, který může použít ke vložení překážek libovolných tvarů.
- Čára – Vloží rovnou čáru.
- Obdélník – Vloží obdélník.
- Polygon – Vloží libovolný mnohoúhelník. Nejdřív si levým tlačítkem zvolíme jeho vrcholy a stisknutím pravého ho pak vykreslíme.
- Guma – Slouží k mazání překážek nebo jejich částí.

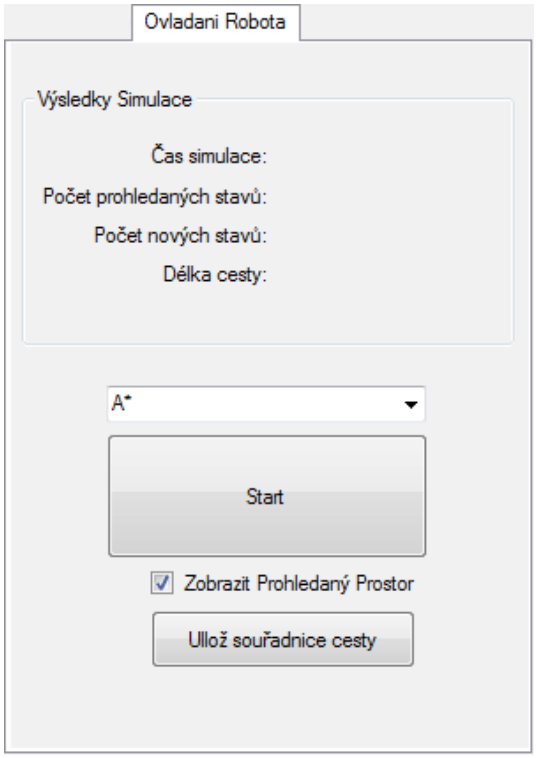


Obr. 25 Panel Návrh scény.

Panel Ovládání robota (viz Obr. 26) slouží k navigaci robota a získání základních údajů o prohledávání. Ve scrollovací nabídce si můžeme vybrat jeden z pěti algoritmů, který chceme použít k provedení plánování nejkratší cesty.

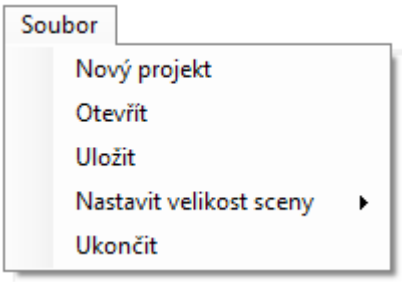
- A*
- Dynamic A* Lite Second Version – K naprogramování byla použita verze D* Lite Second Version. Její pseudokód je zobrazen na Obr. 19 .
- Anytime Repairing A* – Naprogramován podle pseudokódu uvedeného na Obr. 21 v oddílu 3.5.2.
- Anytime Dynamic A* – Naprogramován podle pseudokódu uvedeného na Obr. 22 v oddílu 3.5.3.
- Global Lifelong Planning A* - K naprogramování byla použita dopředná verze GLPA* uvedená na Obr. 9 .

Potom, co jsme si vybrali algoritmus a zvolili, zda chceme zobrazit prohledávanou část pracovní oblasti, můžeme tlačítkem Start zahájit samotné vyhledávání. Po jeho ukončení se nám v pracovní oblasti vykreslí nalezené řešení a v panelu Ovládání robota se nám vypíši základní informace o tomto prohledávání (čas, počet prohledaných stavů, délka nalezené cesty). Poté můžeme pomocí nabídky Návrh scény provést změny do prohledávaného prostředí, jako je přidání či odstranění překážek, a tlačítkem Start znovu spustit vyhledávání. Je však důležité nezapomenout, že jednotlivé algoritmy si mezi sebou nepředávají informace o předchozích hledáních, a proto jejich změna mezi hledáními není vhodná.
Po nalezení cesty robota, ji můžeme ve formě souřadnic expandovat do textového souboru pomocí tlačítka Uložit souřadnice cesty.



Obr. 26 Panel Ovládání robota.

Dalším důležitým ovládacím prvkem simulačního prostředí je menu Soubor (zobrazené na Obr. 27). To nám umožňuje vytvářet nové projekty, nahrávat předem vytvořené pracovní oblasti anebo je uložit, či zpracovávat grafické výsledky plánování cesty. Také nám umožňuje vybrat jednu ze sedmi přednastavených velikostí pracovní oblasti.



Obr. 27 Menu Soubor

Ve znázornění pracovní oblasti se používá několik základních barev k popsání pracovního prostoru:

- Modrá – reprezentuje překážky
- Azurová – reprezentuje nově prohledané stavy
- Zelená – reprezentuje přenesené stavy prohledané v předchozích vyhledáváních
- Červená – reprezentuje nejkratší nalezenou cestu
- Hnědá – reprezentuje počáteční bod
- Žlutá – reprezentuje cílový bod

5 VÝSLEDKY EXPERIMENTŮ

Byly provedeny experimenty na několika odlišných scénách s rozdílnou obtížností. Tyto experimenty probíhaly ve třech krocích:

1. Bylo provedeno hledání nejkratší cesty.
2. Scéna byla mírně pozměněna (simulace vlivu dynamického prostředí).
3. Bylo provedeno hledání nejkratší cesty na pozměněné scéně.

Výsledky algoritmů pak byly porovnány pro jednotlivé scény. Porovnání proběhlo podle času prohledávání, počtu nově prohledaných stavů a délky cesty.

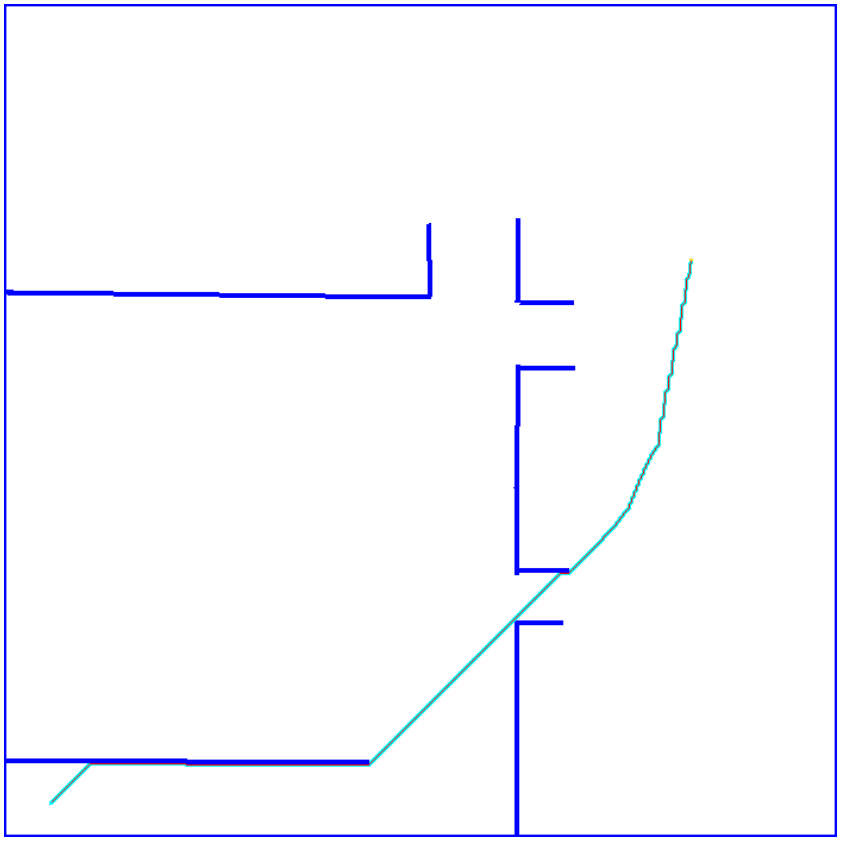
První scéna

Byla vytvořena scéna o velikosti 700x700 bodů s překážkami velmi nízké obtížnosti (viz Obr. 28), a po prvním prohledání byla upravena (viz Obr. 29).

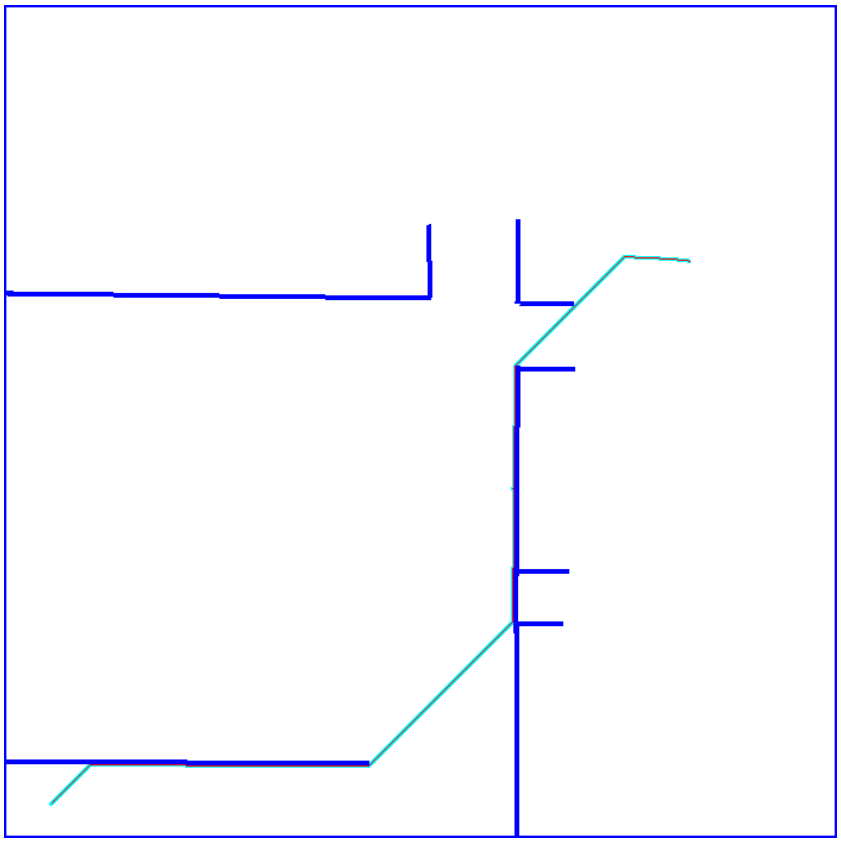
Použitý algoritmus						
	Čas řešení [min:s]		Počet prohledaných stavů		Délka cesty	
	1.	2.	1.	2.	1.	2.
A*	00:00,65	00:00,42	2446	2306	698	751
D* Lite	23:28,33	3:45:41,0	49119	100707	725	752
GLPA*	01:02,28	00:01,80	2669	1586	699	752
ARA*	07:03,19	07:09,97	2669	2768	701	754
AD*	00:00,75	00:00,51	2996	2230	699	752

Tab. 1 Srovnání algoritmů pro první scénu.

Z výsledků experimentů uvedených v Tab. 1 je vidět, že řešení tohoto triviálního problému nejrychleji našel algoritmus A*. Největší časovou náročnost mělo řešení získané algoritmus D* Lite. Toto bylo způsobeno tím, že zatímco všechny testované algoritmy jsou dopředné, D* Lite je algoritmem zpětným a tak narazil na překážky a komplikace, kterým se ostatní algoritmy vyhnuly. Co se týče délky výsledné cesty, jak při prvním plánování, tak při přeplánování, výsledky všech algoritmů byli totožné (s výjimkou D* Lite, protože rozdíly mezi algoritmy byli téměř zabedbatelné.



Obr. 28 Výsledek prvního vyhledávání algoritmem A v první scéně.*



Obr. 29 Výsledek druhého vyhledávání algoritmem A v první scéně.*

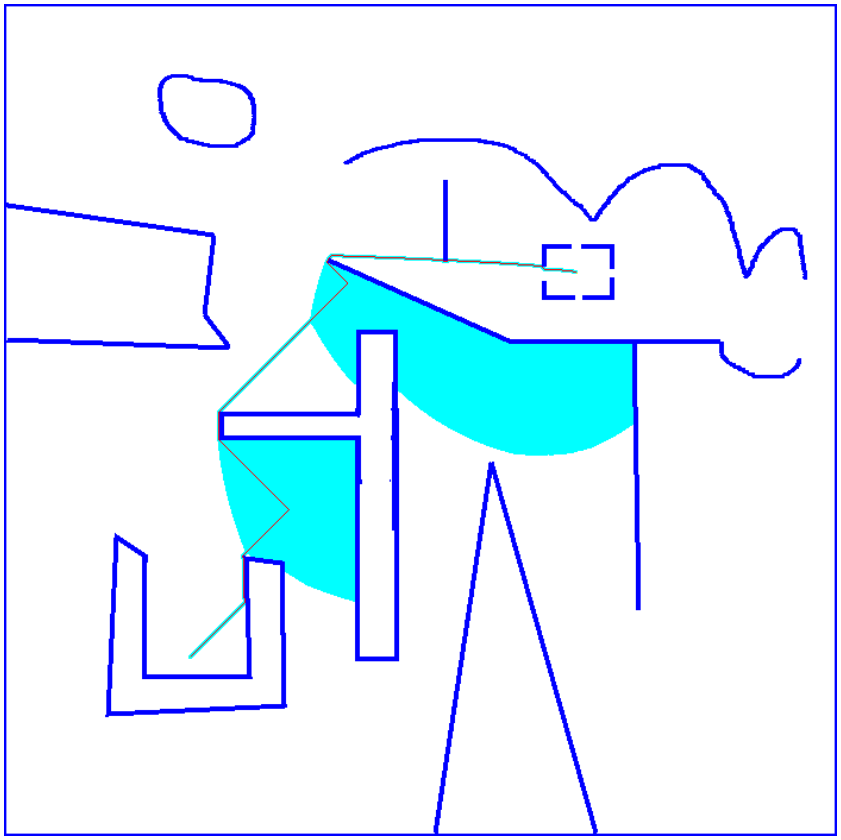
Druhá scéna

Byla vytvořena scéna o velikosti 700x700 bodů s překážkami střední obtížnosti (viz Obr. 30), a po prvním prohledání byla upravena (viz Obr. 31).

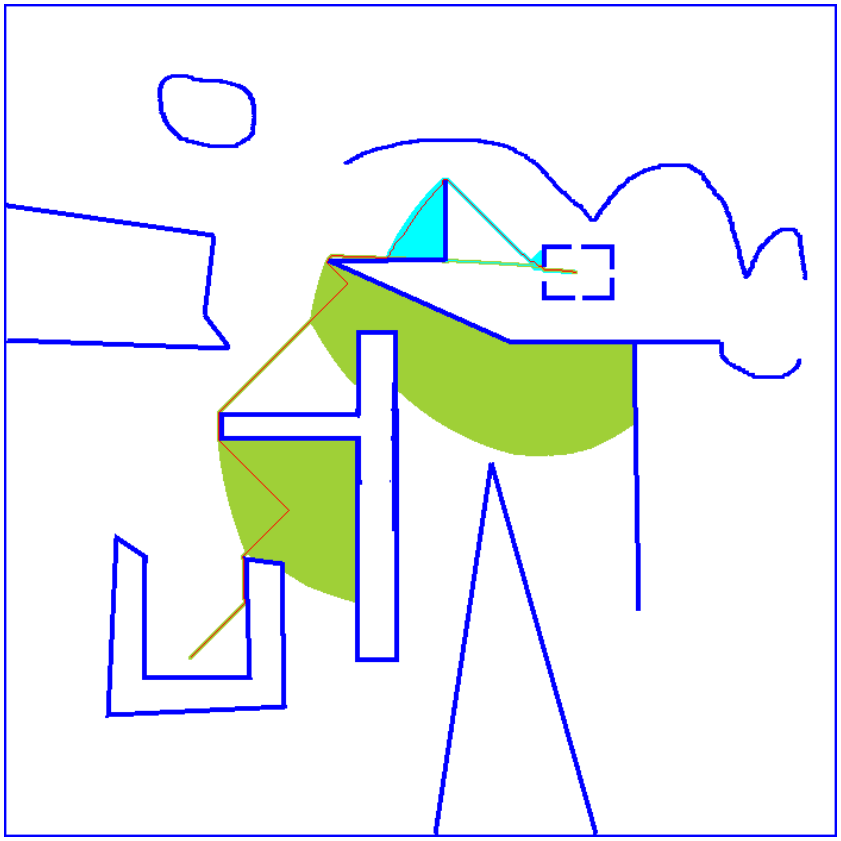
Použitý algoritmus						
	Čas řešení [min:s]		Počet prohledaných stavů		Délka cesty	
	1.	2.	1.	2.	1.	2.
A*	07:03,19	07:09,97	31690	33515	548	569
D* Lite	00:05,31	02:16,41	6437	15372	544	543
GLPA*	18:28,74	01:19,30	36040	2573	544	562
ARA*	17:16,98	18:12,33	42047	43393	546	583
AD*	20:57,64	22:06,75	42441	42586	544	582

Tab. 2 Srovnání algoritmů pro druhou scénu.

Z výsledků experimentu v Tab. 2 je vidět, že se nám zopakovala situace z první scény ovšem se zcela opačným výsledkem. Tentokrát D* Lite dosáhl v porovnání s ostatními algoritmy vynikajících výsledků, což zase bylo způsobeno tím že na rozdíl od ostatních testovaných algoritmů není řazený dopředně, ale zpětně.



Obr. 30 Výsledek prvního vyhledávání algoritmem GLPA ve druhé scéně.*



Obr. 31 Výsledek druhého vyhledávání algoritmem GLPA ve druhé scéně.*

Třetí scéna

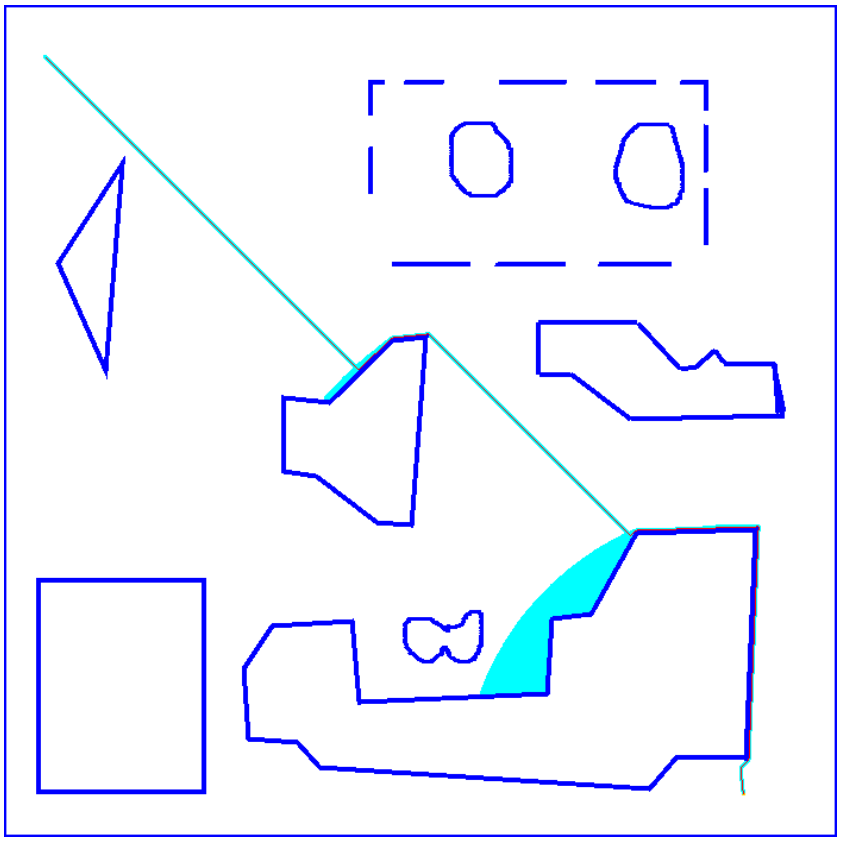
Byla vytvořena scéna o velikosti 700x700 bodů s překážkami nižší obtížnosti (viz Obr. 32), a po prvním prohledání byla upravena (viz Obr. 33).

Použitý algoritmus						
	Čas řešení [min:s]		Počet prohledaných stavů		Délka cesty	
	1.	2.	1.	2.	1.	2.
A*	00:12,89	00:03,28	10032	5674	824	1001
D* Lite	00:01,88	00:11,38	4054	4220	994	1040
GLPA*	01:11,29	01:55,91	12744	12084	825	991
ARA*	00:09,76	00:01,47	9162	4617	827	1002
AD*	00:13,61	00:01,65	9293	4573	825	1001

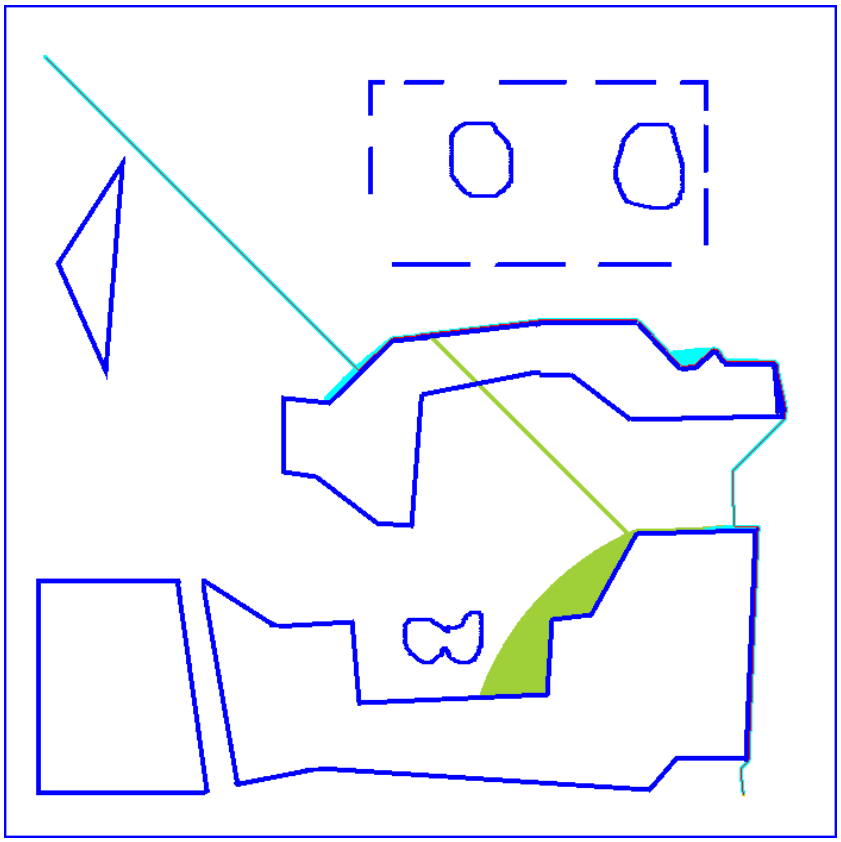
Tab. 3 Srovnání algoritmů pro třetí scénu.

Ve třetím experimentu, viz výsledky v Tab. 3, našel řešení nejrychleji zase D* Lite, protože kvůli svému řetězení našel úplně jinou trasu, než ostatní algoritmy, ale z toho důvodu jím nalezená cesta, v porovnání s ostatními algoritmy, byla také nejdelší. Nejkratší cestu našel GLPA*, ale také měl největší časové nároky. Rozdíly mezi ostatními algoritmy jsou zanedbatelné, protože jejich výsledky

byly téměř totožné.



Obr. 32 Výsledek prvního vyhledávání algoritmem AD ve třetí scéně.*



Obr. 33 Výsledek druhého vyhledávání algoritmem AD ve třetí scéně.*

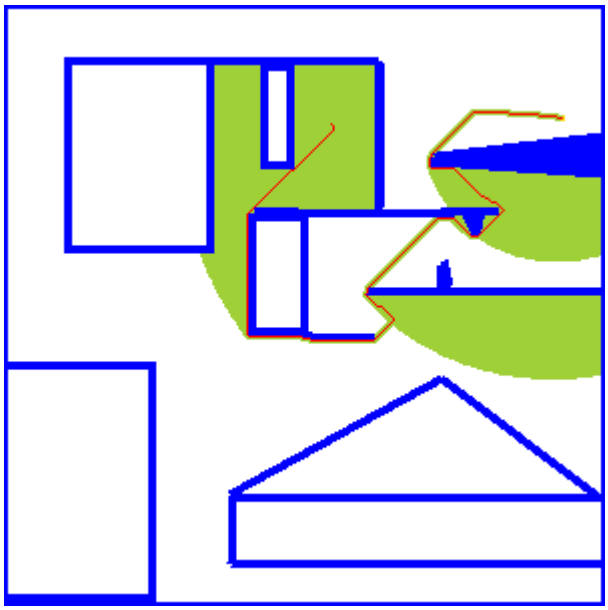
Čtvrtá scéna

Byla vytvořena scéna o velikosti 300x300 bodů s překážkami nízké obtížnosti (viz Obr. 34), a po prvním prohledání byla upravena (viz Obr. 35).

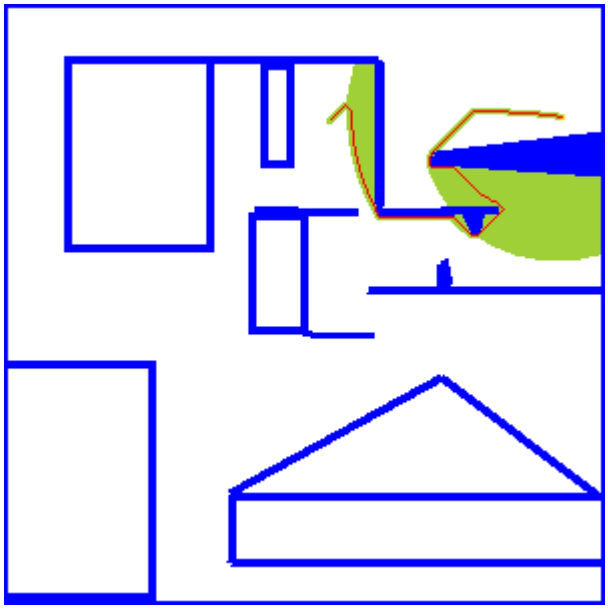
Použitý algoritmus						
	Čas řešení [min:s]		Počet prohledaných stavů		Délka cesty	
	1.	2.	1.	2.	1.	2.
A*	00:29,74	00:05,37	13794	6476	389	245
D* Lite	03:21,93	00:01,10	23430	160	372	221
GLPA*	01:32,72	00:18,94	14177	6716	372	218
ARA*	00:48,89	00:02,38	14117	4458	377	237
AD*	01:12,84	00:17,36	14142	4940	375	218

Tab. 4 Srovnání algoritmů pro čtvrtou scénu.

Z výsledků experimentů v Tab. 3 je vidět, že základní metoda A* získala nejrychlejší řešení, ale jím získaná cesta byla ze všech získaných řešení nejdelší. Nejkratší cestu našel algoritmus GLPA* za cenu větší časové náročnosti. Řešení, které kombinuje relativně krátkou cestu s malou časovou náročností, našel algoritmus ARA*.



Obr. 34 Výsledek prvního vyhledávání algoritmem ARA ve čtvrté scéně.*



Obr. 35 Výsledek druhého vyhledávání algoritmem ARA ve čtvrté scéně.*

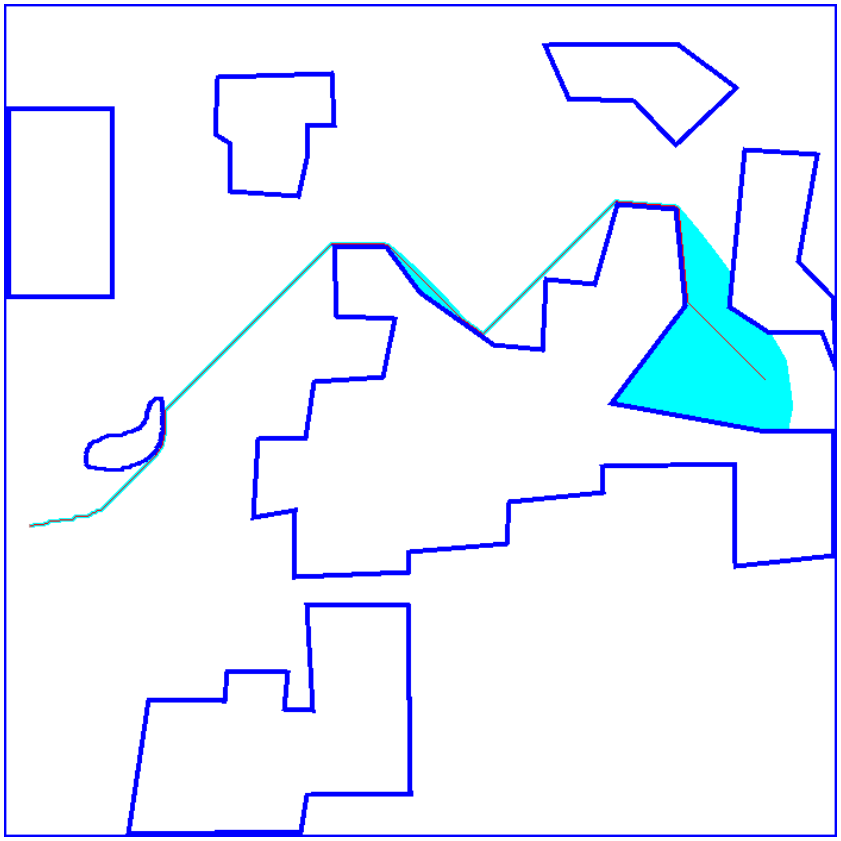
Pátá scéna

Byla vytvořena scéna o velikosti 700x700 bodů s překážkami (viz Obr. 36), a po prvním prohledání byla upravena (viz Obr. 37).

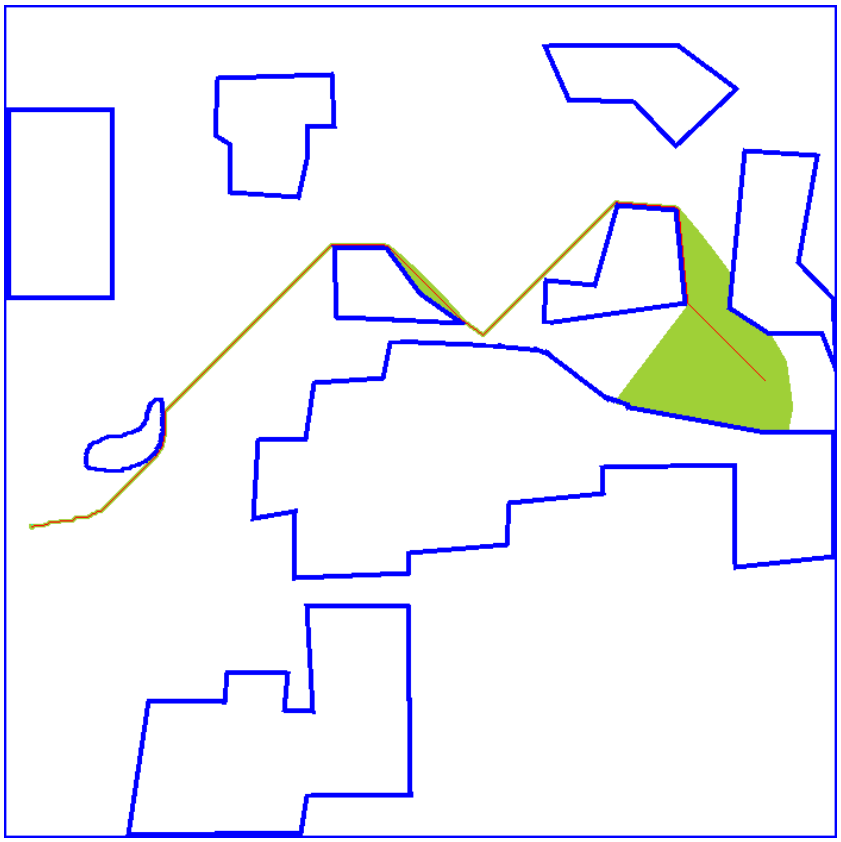
Použitý algoritmus						
	Čas řešení [min:s]		Počet prohledaných stavů		Délka cesty	
	1.	2.	1.	2.	1.	2.
A*	00:31,70	00:00,48	14340	2254	742	636
D* Lite	01:39,01	00:00,28	15795	2	720	720
GLPA*	02:47,93	01:00,64	14687	5318	746	736
ARA*	00:09,89	00:00,46	9248	2300	824	639
AD*	00:13,57	00:00,53	9075	2300	833	637

Tab. 5 Srovnání algoritmů pro pátou scénu.

V pátém experimentu, viz výsledky v Tab. 5, cestu nejrychleji našel algoritmus ARA*, ovšem její délka byla druhá největší. Nejdelší cestu našel druhý nejrychlejší algoritmus AD*. Nejkratší cestu nalezl algoritmus D* Lite. A jelikož D* Lite hledá novou cestu jen pokud po stávající cestě narazí na novou překážku, tak v upravené scéně ani nezačínal nové hledání a přenesl do ní výsledky z minulého hledání.



Obr. 36 Výsledek prvního vyhledávání algoritmem D Lite v páté scéně.*



Obr. 37 Výsledek druhého vyhledávání algoritmem D Lite v páté scéně.*

6 ZÁVĚR

V práci byla řešena problematika plánování cesty robota. První část diplomové práce je věnována problematice některých metod navigace mobilního robota. V práci se zabýváme aplikací grafových algoritmů pro navigaci částečně známým prostředím.

Existuje poměrně velké množství grafových algoritmů či jejich různých modifikací a v práci byly zmíněny ty nejdůležitější z nich.

V rámci práce bylo také vytvořeno simulační prostředí,v němž bylo implementováno pět vybraných algoritmů (A*, ARA*, AD*, GLPA*, D* Lite second version). Jeho účelem je snadný návrh a správa scén různých velikostí, intuitivní vkládání rozmanitých překážek a poté také testování implementovaných algoritmů. Byly zkoumány algoritmy pro statické scény, což jsou takové scény které se nemění v závislosti na čase, a v případě změny se musí nové řešení vypočítat celé od začátku. Dále byly studovány algoritmy pro dynamické scény, kdy se do nových hledání přenáší části starých řešení, ze kterých se potom vychází a šetří se čas a výpočetní kapacita. Nakonec byly analyzovány anytime algoritmy, které se používají v případech, kdy mají omezený výpočetní čas, a tak nejdříve hledají sub-optimální řešení, které následně vylepšují za předpokladu, že jim ještě zbývá nějaký čas. Byly také zkoumány kombinace uvedených algoritmů.

Byly také provedeny srovnávací experimenty na pěti různě složitých scénách, ve kterých jsme si ověřili některé z jejich teoretických vlastností. Z nich vyplynulo, že zatímco algoritmus ARA* je vhodnou volbou pro rychlé plánovaní cesty, jeho řešení nejsou zrovna ideální. Zatímco nejpomalejší z testovaných algoritmů, GLPA*, najde nejkratší optimální cestu, D* Lite (a AD*, který z něj vychází) slouží jako dobrý kompromis mezi rychlostí nalezení cesty a její optimalitou.

Co se týče implementace algoritmů, vycházel jsem z prací [8], [9], [12], [13], ale zatímco v nich byly ověřovací experimenty prováděny v prostředí s hrubou mřížkou a polygonálními překážkami. Já jsem použil extrémně jemnou mřížku, kde jedno pole mřížky mělo velikost jednoho pixelu, a tím dosáhl nejen zpřesnění plánování cesty, ale také překážek zdánlivě nepolygonálního charakteru. To je také důvodem vysoké časové náročnosti, která omezuje praktické použití. V budoucnu by se tento problém dal vyřešit zavedením hrubší mřížky a jejím zjemňováním.

Ovšem vybrání „nejlepšího“ algoritmu, jak nám vyplynulo ze srovnávacích experimentů, je téměř nemožné. Protože každý z nich má své silné a slabé stránky a plánování cesty v reálném prostředí záleží na mnoha různých parametrech. V praxi se většinou postupuje tak, že vybereme algoritmus, který splňuje nejvíce požadavků prostředí, ve kterém bude nasazen a pak jeho kód upravíme a optimalizujeme podle specifikací robota a jeho budoucí funkce.

SEZNAM POUŽITÉ LITERATURY

[1] GROSS, T. *Plánování cesty mobilního robota*. Brno, 2007. 71 s. Diplomová práce na Fakultě strojního inženýrství Vysokého Učení Technického v Brně na Ústavu automatizace a informatiky. Vedoucí diplomové práce RNDr. Jiří Dvořák, Csc.

[2] SEDLÁK, V. *Plánování cesty robota pomocí mravenčích systémů*. Brno, 2009. 43 s. Bakalářská práce na Fakultě strojního inženýrství Vysokého Učení Technického v Brně na Ústavu automatizace a informatiky. Vedoucí diplomové práce RNDr. Jiří Dvořák, Csc.

[3] JANOVEC, A. *Navigace mobilního robotu pomocí fuzzy logiky*. Brno, 2010. 77 s. Diplomová práce na Fakultě strojního inženýrství Vysokého Učení Technického v Brně na Ústavu automatizace a informatiky. Vedoucí diplomové práce RNDr. Jiří Dvořák, Csc.

[4] PASTEL, Amit. *Amit's Game Programming Information* [online]. 2011, 8th of April 20011 [cit. 2011-04-20]. Dostupné z: <http://www-cs-students.stanford.edu/~amitp/gameprog.html#paths>

[5] KOENIG, S. LIKHACHEV, M. FURCY, D. *Lifelong Planning A**. 2003. 54 s.

[6] STENZ, A.The D* Algorithm for Real-Time Planning of Optimal Traverses.1994. 34s.

[7] KOENIG, S. LIKHACHEV, M. *Improved Fast Replanning for Robot Navigation in Unknown Terrain**. College of Computing, Georgia Institute of Technology, Atlanta, Technical Report GIT-COGSCI-2002/3, 2001.

[8] KOENIG, S. LIKHACHEV, M. Fast Replanning for Robot Navigation in Unknown Terrain. In *IEEE TRANSACTIONS ON ROBOTICS*, VOL. 21, NO. 3, JUNE 2005.

[9] LIKHACHEV, M. FERGUSON, D. GORDON, G. STENZ, A. THRUN, S. *Anytime Dynamic A*: An Anytime, Replanning Algorithm*.American Association for Artificial Intelligence, 2005.

[10] LIKHACHEV, M. FERGUSON, D. GORDON, G. STENZ, A. THRUN, S. *Anytime Search in Dynamic Graphs.* Elsevier Science, October 2007.

[11] HANSEN, E. ZHOU, R. *Anytime Heuristic Search.* Journal of Artifcial Intelligence Research 28 , 2007, p. 267-297.

[12] LIKHACHEV, M. GORDON, G. THRUN, S. *ARA*: Anytime A* with Provable Bounds on Sub-Optimality*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[13] LIKHACHEV, M. KOENIG, S. *A Generalized Framework for Lifelong Planning A* Search.* American Association for Artificial Intelligence, 2005.

[14] RUSSEL, S. NORVIG, P. *Artificial Intelligence: A Modern Approach*. 2. vydání. New Jersey, USA : Prentice Hall, 2003. ISBN 0-13-790395-2. p. 59-136.

[15] MAŘÍK, V., a kol. *Umělá inteligence (1)*. Praha : Academia, 1993. ISBN 80-200-0496-3. p. 33-57.

SEZNAM PŘÍLOH

Příloha č. 1– Přiložené medium CD-R obsahující:

- Tento dokument v elektronické podobě
- Simulační prostředí
- Scény, na kterých byly prováděny experimenty a grafické výstupy těchto experimentů
- Zdrojový kód simulačního prostředí