VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

POKROČILÉ METODY PLÁNOVÁNÍ CESTY MOBILNÍHO ROBOTU
ADVANCED METHODS OF MOBILE ROBOT PATH PLANNING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. Lenka Maňáková

VEDOUCÍ PRÁCE
SUPERVISOR
RNDr. Jiří Dvořák, CSc.

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

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

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

Pokročilé metody plánování cesty mobilního robotu

Stručná charakteristika problematiky úkolu:

Úkolem systému pro plánování cesty robotu je najít cestu z počáteční do cílové pozice tak, aby se robot nedostal do kolize se známými překážkami a aby byla optimalizována nějaká kriteriální funkce. Pokročilé metody umožňují např. přeplánování cesty při změně scény (replanning algorithms), získání dobrého řešení ve vymezeném čase (anytime algorithms) nebo zefektivnění procesu hledání pomocí redukce počtu prozkoumávaných uzlů stavového prostoru.

Cíle diplomové práce:

1. Analyzovat přístupy k plánování cesty mobilního robotu.
2. Implementovat vybrané metody plánování cesty.
3. Provést a vyhodnotit ověřovací a srovnávací experimenty.

Seznam doporučené literatury:

ALGFOOR, Z. A. et al. A New Weighted Pathfinding Algorithms to Reduce the Search Time on Grid Maps. Expert Systems with Applications, vol. 71, 2017, pp. 319-331.

DAKULOVIĆ, M., PETROVIĆ, I. Two-way D* Algorithm for Path Planning and Replanning. Robotics and Autonomous Systems, vol. 59, issue 5, 2011, pp. 329-342.

COHEN, L. et al. Anytime Focal Search with Applications. In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18), 2018, pp. 1434-1441.

MAC, T. T. et al. Heuristic Approaches in Robot Path Planning: A Survey. Robotics and Autonomous Systems, vol. 86, 2016, pp. 13-28.

ZHANG, A. et al. Rectangle Expansion A* Pathfinding for Grid Maps. Chinese Journal of Aeronautics, vol. 29, issue 5, 2016, pp. 1385-1396.

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

V Brně, dne

L. S.

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

Abstrakt
Tato práce je zaměřená na pokročilé metody plánování cesty mobilního robotu. V teoretické části se zaměřuje na popis vybraných grafových metod, které o jsou specifické tím, že zefektivňují proces hledání nejkratší cesty a to například redukcí počtu prozkoumaných uzlů stavového prostoru. V rámci praktické části bylo vytvořeno simulační prostředí v jazyce Python a v tomto prostředí pak byly implementovány vybrané algoritmy.

Summary
This work is focused on advanced methods of mobile robot's path planning. The theoretical part describes selected graphical methods, which are useful for speeding up the process of finding the shortest paths, for example through reduction of explored nodes of the state space. In the practical part was created simulate environment in the Python language and in this environment, selected algorithms was implemented.

Klíčová slova
Plánování cesty, mobilní robot, A^* algoritmus, JPS algoritmus, Subgoal algoritmus, bounding box.

Keywords
Path planning, mobile robot, A^* algorithm, JPS algorithm, Subgoal algorithm, bounding box.

MAŇÁKOVÁ, L. *Pokročilé metody plánování cesty mobilního robotu.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2020. 58 s. Vedoucí RNDr. Jiří Dvořák, CSc.

Prohlašuji, že jsem diplomovou práci *Pokročilé metody plánování cesty mobilního robotu* vypracovala samostatně pod vedením RNDr. Jiřího Dvořáka, CSc. a užitím materiálů uvedených v seznamu literatury.

Bc. Lenka Maňáková

Na tomto místě bych ráda poděkovala svému vedoucímu diplomové práce RNDr. Jiřímu Dvořákovi, CSc. za cenné rady a připomínky a dále pak mým rodičům za podporu při studiu.

Bc. Lenka Maňáková

Obsah

Úvod

Plánování cesty je nedílnou součástí autonomního systému, který zodpovídá za pohyb z jednoho bodu do druhého. Tyto cesty jsou navrhovány různými technikami v závislosti na tom, zda systém prochází prostředím, které je známé, neznámé nebo částečně známé. Výzkum v této oblasti se zabývá nejvíce pozemní robotikou a manipulačními systémy. Nicméně plánování cesty nachází aplikaci nejen v robotice, ale i například ve strategiích her. Pro plánování cesty se využívají algoritmy umělé inteligence, přičemž tato práce je zaměřená na pokročilé metody. Oproti klasickým metodám, tyto metody umožňují například redukci prozkoumávaných uzlů, přeplánování cesty při nějaké změně scény nebo metody, které získávají dobré řešení ve vymezeném čase.

Cílem této diplomové práce je popsat dosavadní přístupy k plánování cesty mobilního robotu, popsat některé z pokročilých metod plánování cesty a vybrané z nich pak následně implementovat spolu s provedením porovnávacích experimentů.

První kapitola je věnována obecnému pohledu na plánování cesty, popis a zpracování prostředí a obecný popis pokročilých metod. Druhá část je věnována popisu vybraných algoritmů, které byly vybrány pro následnou implementaci. Patří mezi ně algoritmus A^*, JPS (*Jump point search*) a algoritmus Subgoal. Zároveň jsou v této kapitole popsány jejich různé modifikace, kdy jedna z nich je také implementována, a to A^* s omezováním prostoru prostřednictvím jistého *bounding boxu*. Ve třetí kapitole je popsáno navrhované simulační prostředí, ve kterém byly následně provedeny experimenty, kterým se věnuje kapitola čtvrtá.

1 Plánování cesty robota

Roboti jsou nedílnou součástí našeho života. Své využití často nalézají v automatizaci průmyslu, a to především například v logistice. Přesun z jednoho místa do druhého se může zdát pro člověka jednoduchý, ale pro robota je tento problém jeden z nejnáročnějších. Zkombinovat znalosti o problému, jako je výchozí a cílová pozice, výskyt překážek a jejich statičnost či dynamičnost, a obdržet z nich optimální řešení, vyžaduje implementaci inteligence do robotů. V autonomní robotice je tedy plánování cesty hlavním problémem.

1.1 Plánování cesty mobilního robota obecně

Plánování cest je základním kamenem mnoha důležitých aplikací v oborech zabývajících se například GPS, videohrami, robotikou, logistikou nebo simulací davu a může být implementováno ve statickém, dynamickém a real-time prostředí. Přestože řada výzkumů v posledních dvou desetiletích zlepšila přesnost a účinnost technik plánování cest, stále tento problém přitahuje velkou pozornost. [3]

Výběr správné inteligence, která ovládá mobilního robota, je náročný. Největší problém je samozřejmě samostatná navigace. Robot musí znát svou polohu vzhledem k cílové poloze a musí reagovat na nástrahy prostředí (překážky). Dle [1] lze jednoduše říci, že abychom vyřešili problém s navigací robota, musíme najít odpovědi na tři následující otázky: Kde se robot právě nachází? Kam chci aby robot směřoval? Jak se tam robot dostane? Odpovědí jsou pak tři základní funkce navigace a to je lokalizace, mapování a plánování pohybu.

Lokalizace pomáhá robotovi zjišťovat aktuální umístění v prostředí, ve kterém se pohybuje. Fyzicky mu k tomu napomáhají různé kamery, senzory a dálkoměry. Lokalizace se může vztahovat k lokálnímu prostředí (např. umístění ve středu nebo na kraji místnosti) nebo může být vyjádřena přímo jako souřadnice.

Mapování pomáhá robotovi získávat znalosti o možných směrech a umístěních. Mapa může být reprezentována např. grafem nebo maticí.

Plánování cesty je pak samotné gró použité inteligence hledající nejkratší cestu mezi startovací polohou a cílovou polohou. Tato diplomová práce je právě zaměřená na implementaci určitých algoritmů umělé inteligence, které nejkratší cestu hledají.

Níže uvedený obrázek 1.1 uvádí jeden z přístupů ke klasifikaci plánování cesty dle [1].

Obrázek 1.1: Klasifikace plánování cesty

Rozlišujeme dva druhy *prostředí*, ve kterých se robot pohybuje, a to na základě jeho neměnnosti.[2]

- *Statické prostředí* je neměnné, startovní a cílová pozice jsou pevně dány a překážky nemění své umístění v průběhu času. Jednoduše řečeno, pokud se jedná o ideální statické prostředí, tak je robot jediný objekt, který se hýbe.
- *Dynamické prostředí* během procesu vyhledávání cesty mezi startovní a cílovou pozicí může měnit umístění překážek a pozici cíle.

Plánování cesty v dynamickém prostředí je obvykle složitější než ve statickém prostředí, proto přístupy k plánování používané ve statickém prostředí nejsou vhodné pro dynamický problém. Algoritmy se totiž musí umět přizpůsobit jakékoli neočekávané změně.

K identifikaci počáteční a cílové polohy pro plánování cesty mobilního robota slouží jistá znalost mapy. Na základě této znalosti pak můžeme rozlišit plánování cesty *globální* a *lokální*. [1]

- *Globální plánování cesty* s sebou přináší prioritní znalost prostředí, jedná se o tzv. úmyslné plánování. Celé plánování je založené na znalosti mapy, robot zná prostředí, ve kterém se pohybuje. Nejdříve je vytvořena proveditelná cesta k cíli a až poté je k němu vyslán robot. Plánování se provádí offline.
- *Lokální plánování cesty* se provádí v prostředí, o kterém má robot omezenou znalost, jedná se o tzv. plánování cesty založené na reakci a akci (reaktivní plánování). V důsledku toho musí robot sestavovat odhadovanou mapu prostředí během procesu vyhledávání cesty, aby se vyhnul překážkám a získal tak vhodnou cestu k cílovému stavu. Plánování cesty tedy probíhá online.

Dle *optimálnosti* lze plánování cesty rozlišit na *exaktní* a *heuristické*. [1]

- *Exaktní plánování cesty* je metoda plánování cesty, která garantuje nalezení optimální cesty mezi startovní pozicí a cílovou pozicí, v případě že tato cesta existuje. V případě, že tato cesta neexistuje, tato metoda tuto skutečnost oznámí.
- *Heuristické plánování cesty* hledá oproti metodám exaktním kvalitní řešení za kratší dobu. V průběhu výpočtu používají jistý odhad nebo náhodné rozhodnutí (heuristiku). Podle typu použité heuristiky tyto metody mohou nalézt optimální řešení (pokud existuje) ale nemusí.

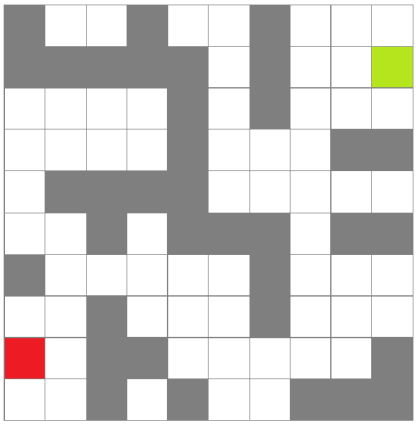
Další možnou klasifikací může být například to, kolik robotů se podílí na plánování cesty. Zde existují dva přístupy, a to *single-agent* (pouze jeden robot) a *multi-agent* (spolupracuje mezi sebou více robotů).

1.2 Prostředí a jeho předzpracování

Reprezentace prostředí může být dvojího druhu. Rozlišujeme *diskrétní prostředí* nebo *spojité prostředí*. [4]

1.2.1 Diskrétní prostředí

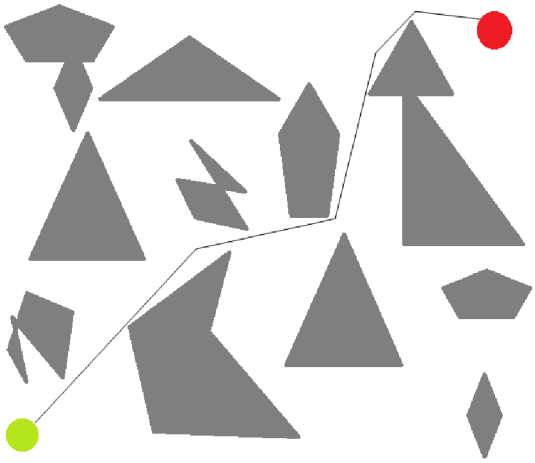
Diskrétní prostředí může být reprezentováno pomocí buněk či pomocí uzlů a hran například grafu viditelnosti. V případě buňkové reprezentace je jeho tvar závislý na tom, jak moc velký je kladen požadavek na přesnost překážek. Neboť čím menší je velikost buněk, tím je reprezentace překážek přesnější, ale zase o to je větší časová náročnost na práci prostředím. V opačném případě je pak problém ten, že buňka je označená za překážku i v případě, že překážku obsahuje pouze částečně. Diskrétní prostředí může mít tvar jako je například na obrázku 1.2. Směr pohybu cesty robota a tvar překážek je tedy omezen, nicméně pokud se zvýší počet buněk, tak se zajistí i větší přesnost, což ale s sebou přináší úskalí ve formě vyšší výpočetní náročnosti.



Obrázek 1.2: Diskrétní prostředí reprezentované pomocí buněk

1.2.2 Spojité prostředí

Spojité prostředí je podobné tomu reálnému prostředí. Pohyb robota v tomto prostředí není omezen, a proto nám dává možnost zadávat přesnější informace o topologii terénu. Oproti diskrétnímu prostředí se ale vyznačuje složitější náročností na prohledávání prostoru. Spojité prostředí je nutné diskretizovat, a to například pomocí rozkladu do buněk, grafů viditelnosti či Voronoiovými diagramy.



Obrázek 1.3: Spojité prostředí

1.2.3 Pracovní a konfigurační prostor

Jak uvádí například práce [4] a [5], rozlišujeme dva prostory pro robota, a to *pracovní prostor* a *konfigurační prostor*. V pracovním prostoru se robot pohybuje a vykonává v něm úkoly. Lze jej reprezentovat jako N-rozměrný euklidovský prostor $\mathbb{R}^N$ a označuje se jako W. V tomto prostoru se vyskytují překážky, které omezují pohyb robota a rozdělujeme je dle toho, jestli v průběhu času mění svou polohu či velikost (dynamické překážky) a nebo zůstávají neměnné (statické překážky). Aby bylo možné tento pracovní prostor popsat, byl zaveden pojem konfigurační prostor C, kde se robot chápe jako systém, který se nachází v určitém stavu. Konfigurační prostor je pak množina všech přípustných stavů (konfigurací). Hlavní myšlenkou je, že je robot reprezentován jako bod a úloha plánování cesty robotu je pak redukována na plánování cesty bodu. Ten se pohybuje v konfiguračním prostoru, který obsahuje dvě oblasti, a to volný konfigurační prostor a kolizní oblast. Aby bylo plánování cesty úspěšné, je nutné aby startovní i cílové stavy (konfigurace) patřily do volného konfiguračního prostoru. Obvykle se plánování cesty skládá z postupu, kdy se pracovní prostor převede na konfigurační prostor, ten je popsán grafovou strukturou a v něm se pak aplikuje algoritmus pro hledání cesty grafem. [3]

1.2.4 Stavový prostor

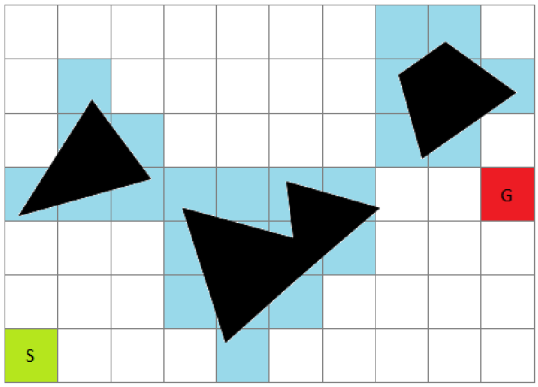
Plánování cesty lze formálně popsat jako procházení stavů $x \in X$, kde X je množina všech stavů, označovaná jako stavový prostor. Stavový prostor je definován jako dvojice $X = [S, P]$, kde S je konečná množina stavů, P je konečná množina operátorů reprezentujících přechody mezi stavy. Plánováním cesty v tomto stavovém prostoru pak rozumíme úlohu $U = (s_0, C)$, kde $s_0 \in S$ je počáteční stav a $C \in S$ je množina cílových stavů a hledá se řešení ve tvaru posloupnosti stavů. Tento stavový prostor bývá nejčastěji reprezentován orientovaným grafem. Uzly představují stavy a orientované hrany pak přechod mezi stavy. Plánování cesty pak spočívá v nalezení cesty v grafu mezí počátečním a cílovým uzlem. Stavová reprezentace úloh tvoří teoretický základ většiny metod umělé inteligence.

1.2.5 Metody zpracování pracovního prostoru

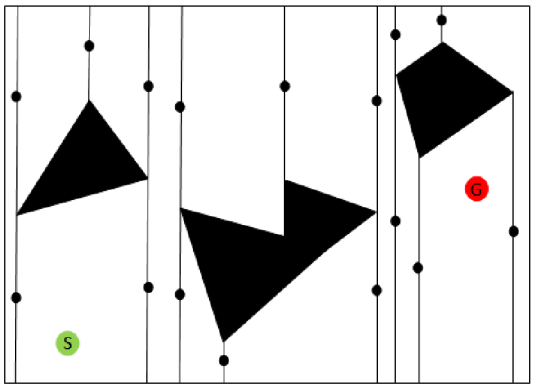
Metod zpracování pracovního prostoru je mnoho, nicméně k nejzákladnějším metodám dle [4], [5] a [6] patří:

- **Metody rozkladu do buněk**

  Princip spočívá v rozkladu daného prostředí do buněk, u kterých se určí, zda obsahují volný nebo obsazený prostor (překážku). Rozklad do buněk je dvojího typu, a to *aproximativní* a *exaktní*. Aproximativní metoda rozkládá prostor do buněk stejného tvaru. Grafovou strukturu tvoří uzly, které reprezentují volné buňky, a jsou spojené s hranami s jejich sousedy. Nevýhodou této metody je, že nemusí nalézt cestu, i když doopravdy existuje, a to z toho důvodu, že buňka, která je označena za obsazenou, sice překážku obsahuje, ale tato překážka buňku zcela nevyplňuje. Princip exaktního rozkladu spočívá v rozdělení prostoru do množiny nepřekrývajících se buněk. Tvar buněk je jednoduchý, většinou jde o trojúhelník nebo lichoběžník. Uzly grafové struktury představují body přechodu, které vznikají na hranici mezi sousedními buňkami. Tato metoda rozkladu vždy najde cestu, pokud cesta existuje.



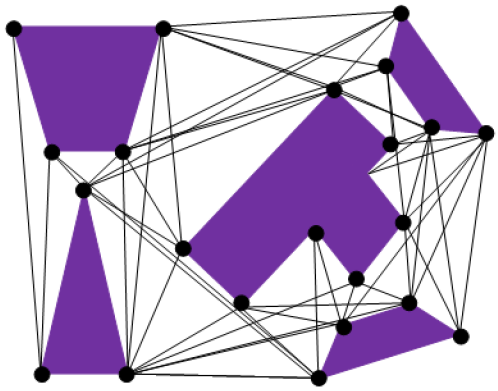
Obrázek 1.4: Aproximativní rozklad



Obrázek 1.5: Exaktní rozklad

- **Metody mapy cest**

  Metody mapy cest vytvářejí grafovou strukturu reprezentující volný pracovní prostor. Hrany tohoto grafu představují cesty kudy se robot může pohybovat. Rozlišujeme *deterministické* a *pravděpodobnostní* metody mapy cest. Příkladem deterministické metody je například graf viditelnosti, který tvoří grafovou strukturu ve které uzly představují startovní a cílové body a vrcholy překážek. Hrany pak představují spojnice uzlů, které neprotínají žádnou z překážek (navzájem na sebe vidí). Příkladem pravděpodobnostní metody je metoda pravděpodobnostních stromů. Myšlenkou je postupné vytváření prohledávacího stromu, který se snaží rychle a rovnoměrně prohledat konfigurační prostor. Počátek stromu je ve startovní konfiguraci, který se následně v každé iteraci rozrůstá do náhodné volné konfigurace o pevně zvolený krok.

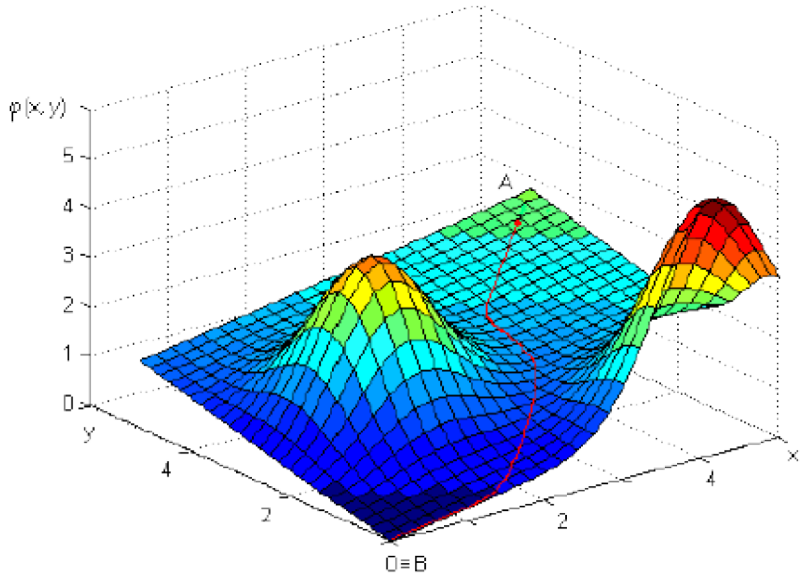


Obrázek 1.6: Graf viditelnosti

Obrázek 1.7: Pravděpodobnostní stromy [7]

- **Metody potenciálových polí**

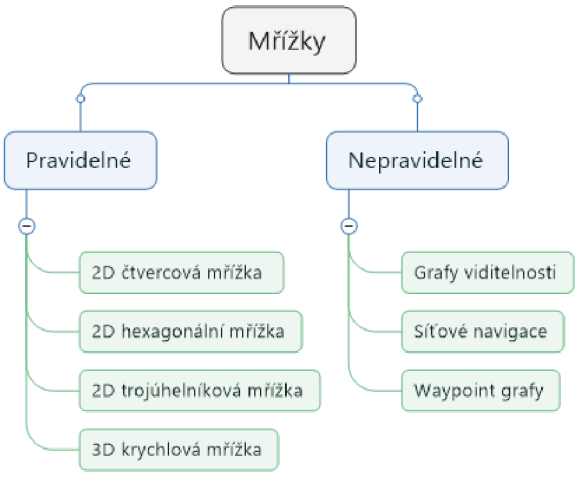
  Hlavní myšlenkou je pokrytí pracovního prostoru potenciálovým polem, které každému bodu prostoru přiřadí potenciál, jehož hodnota závisí na tom, jestli se jedná o startovací, cílovou nebo obsazenou pozici. Nevýhodou této metody je, že robot může uvíznout v lokálním minimu. Startovní pozice má potenciál vyšší než cílová, obsazené pozice jej mají vyšší jak volné pozice a robot se pohybuje ve směru opačného gradientu potenciálové funkce.



Obrázek 1.8: Potenciálové pole [5]

1.3 Algoritmy pro plánování cesty

Jak již bylo řečeno, plánování cesty spočívá v nalezení posloupnosti akcí, které transformují nějaký počáteční stav do požadovaného cílového stavu. Většina algoritmů hledání cesty je založena na atributech mřížky, jakožto reprezentace grafu. Mřížka se skládá z vrcholů, které jsou spojené hranami reprezentující graf. Dle [3] rozlišujeme 2 základní typy mřížek: pravidelné a nepravidelné, viz obr.1.9. Samotné algoritmy pak lze klasifikovat podle tohoto rozdělení, a to na základě vhodnosti použití, a právě práce [3] tuto klasifikaci algoritmů obsahuje.



Obrázek 1.9: Typy mřížek

Tato práce je zaměřena na pokročilé metody pro plánování cesty, a proto zde uvedu pouze tyto metody. Mezi pokročilé metody lze zařadit například:

- metody, které redukují počet prozkoumávaných uzlů,
- metody, které při změně scény dokáží reagovat a přeplánovat tak scénu, tzv. *replanning algoritmy*,

- metody, které získávají dobré řešení ve vymezeném čase, tzv. *anytime algoritmy.*

Prvním zmíněným metodám je věnována další kapitola a patří sem například algoritmus JPS [18], který pro redukci uzlů využívá jistý operátor pro určení tzv. bodů skoků. Další metodou je pak metoda využívající tzv. subgoal graf [20]. Zajímavým přístupem je využití tzv. bounding boxu, kterým lze omezit stavový prostor pouze na určitý prostor a teprve v případě neúspěchu jej zvětšovat. Podobným přístupem se zabývá článek [8], který tuto metodu kombinuje s algoritmem JPS.

Zbylým dvěma přístupům pokročilých metod budou věnovány následující podkapitoly.

1.3.1 Replanning algoritmy

Jak již bylo řečeno na začátku této práce, mobilní roboti se často používají v automatizaci průmyslu, a to například ve skladech, kdy mají za úkol přemístit jednu položku z bodu A do bodu B. V praxi se však často stává, že prostředí, ve kterém se robot pohybuje není statické (neměnné) ale dynamické. Na jeho naplánované cestě se může vyskytnou neznámá nebo pohybující se překážka. V takovém případě se původně vyřešená úloha v podobě nalezení cesty z bodu A do bodu B musí přeplánovat s ohledem na nově vzniklé překážky. Algoritmy, které se tímto zabývají se nazývají *replanning* algoritmy a patří sem například algoritmy D^* [12] a D^*lite [13], které jsou v současné době nejrozšířenější z těchto algoritmů vzhledem k jejich efektivnímu využití heuristiky a přírůstkových aktualizací. Protože tyto dva algoritmy jsou v zásadě velmi podobné, vysvětlím podrobněji algoritmus D^*lite, u kterého bylo dle [13] zjištěno, že je o něco účinější. Mějme tedy počáteční stav $s_s \in S$ a cílový stav $s_g \in S$, kde S je množina všech stavů v konečném stavovém prostoru. D^*lite udržuje nejmenší cenu přechodu z počátečního stavu do koncového stavu tím, že uchovává odhad ceny cesty $g(s)$ z libovolného stavu s do cílového stavu s_g. Dále pracuje s jednokrokovým odhadem $rhs(s)$ pro který platí:

- $rhs(s) = 0$, jestliže $s = s_g$
- jinak platí $rhs(s) = min_{s' \in Succ(s)}(c(s, s^{'}) + g(s^{'}))$,

kde $Succ(s) \in S$ označuje množinu následníků stavu s a $c(s, s^{'})$ označuje cenu přechodu z s do $s^{'}$. Algoritmus využívá heuristiku $h(s, s^{'})$ pro odhad ceny nejkratší cesty ze stavu s do stavu $s^{'}$ a je potřeba aby splňovala $h(s, s^{'}) \leq c^*(s, s^{'})$ a $h(s, s^{''}) \leq h(s, s^{'}) + c^*(s^{'}, s^{''})$ pro všechny stavy $s, s^{'}, s^{''} \in S$, kde $c^*(s, s^{'})$ je cena nejkratší cesty ze stavu s do stavu $s^{'}$. Dále využívá prioritní frontu, kde klíčovou hodnotou je:

$$key(s) = [k_1(s), k_2(s)] = [min(g(s), rhs(s)) + h(s_s, s), min(g(s), rhs(s))].$$

Platí, že $key(s) < key(s^{'})$, jestliže $k_1(s) < k_1(s^{'})$ nebo $k_1(s) = k_1(s^{'})$ a $k_2(s) = k_2(s^{'})$. Algoritmus expanduje z prioritní fronty stavy se zvyšující se prioritou aktualizováním jejich g hodnoty a rhs hodnoty jejich předchůdců, dokud v prioritní frontě není stav s klíčovou hodnotou menší než má počáteční stav. Zjišťování nejkratší cesty probíhá stejným způsobem jako zpětné vyhledávání algoritmu A^*. [11]

1.3.2 Anytime algoritmy

V praxi se často stává, že je potřeba vyřešit určitý problém v omezeném čase a právě *anytime algoritmy* jsou toho schopné. Obvykle tyto algoritmy začínají výpočtem počátečního, potenciálně vysoce suboptimálního řešení a pak se ve zbylém čase toto řešení snaží vylepšit. Algoritmy jsou většinou založené na algoritmu A^*, který je podrobně vysvětlen v kapitole 2.1. Tyto algoritmy využívají vlastnosti, že v mnoha případech zvětšování heuristických hodnot použitých algoritmem A^* obvykle sice vede na značné urychlení, ale za cenu ztráty optimality řešení. A^* má také další vlastnost a to tu, že pokud je použitá heuristika $h(s)$ konsistentní, to jest pokud platí

$$h(s) \leq c(s, s^{'}) + h(s^{'}),$$

pro každého následníka $s^{'}$ stavu s, jestliže s není cílový stav a jestliže je s cílový stav, pak $h(s) = 0$, tak pak pokud jsou heuristické hodnoty vynásobeny inflačním faktorem $\epsilon > 1$, pak cena vygenerovaného řešení je nejvýše ϵ-krát větší než cena optimálního řešení. Tyto skutečnosti vedly k vyvinutí algoritmu *weighted*A^*. [10]

Maxim Likhachev, Geoff Gordon a Sebastian Thrun ve svém článku *ARA*: Anytime A* with Provable Bounds on Sub-Optimality* [10] představili algoritmus *ARA**, který využívá několikanásobného prohledávání pomocí A^* a pojmu lokální nekonzistence. Stav nazveme lokálně konzistentní, když je snížena jeho hodnota g a to až do doby, kdy je znovu expandován [24]. Tento algoritmus začíná prohledáváním prostoru pomocí obyčejného A^* algoritmu s inflačním faktorem ϵ_0, s tím že každý stav je expandován pouze jednou. Jakmile je stav expandován během částečného prohledávání a je lokálně nekonsistentní, pak je přidán do seznamu, který obsahuje všechny takové stavy. Když je hlavní prohledávání u konce, tak jsou tyto stavy z tohoto seznamu přesunuty do nové prioritní fronty založené na novém inflačním faktoru ϵ, na základě které je provedeno nové prohledávání. To zvyšuje efektivitu každého prohledávání dvěma způsoby. Za prvé tím, že každý uzel je expandován pouze jednou a za druhé tím, že prozkoumáním nekonsistentních stavů z předchozího prohledávání se znovu využijí získané výsledky z předchozího prohledávání. Pokud je tedy inflační faktor mezi následnými prohledáváními snížen, je minimalizován zbytek potřebných výpočtů k nalezení řešení. [11]

2 Popis vybraných algoritmů

2.1 Algoritmus A^*

Algoritmus A* je algoritmus, který se řadí mezi informované metody prohledávání stavového prostoru a představili jej P. Hart, N. Nilsson a B. Raphael ve svém článku *A formal basis for the heuristic determination of minimum cost paths* [14]. A* tedy využívá pro prohledávání prostoru hodnotící funkci, na základě které pak vybere stav k expanzi, což vede k urychlení prohledávání. Hodnotící funkce je dána vztahem (2.1)

$$f(i) = g(i) + h(i) \tag{2.1}$$

V tomto vztahu funkce $g(i)$ představuje odhad vzdálenosti mezi počátečním stavem a aktuálním stavem i. Hodnota této funkce pro každý stav j, který je následníkem stavu i, se získá z výpočtu

$$g(j) = g(i) + c(i, j), \tag{2.2}$$

kde $c(i, j)$ je vzdálenost mezi stavem i a j. Funkce $h(i)$, neboli heuristická funkce, představuje odhad vzdálenosti mezi aktuálním stavem i a koncovým stavem s využitím jisté heuristiky.

Aby bylo možné použít určitou heuristickou funkci na problém hledání nejkratší cesty mezi dvěma stavy, tj. aby byl algoritmus optimální, je nutné, aby použitá heuristická funkce byla přípustná. Což znamená, že odhad vzdálenost mezi libovolným stavem a stavem koncovým musí být nezáporný a menší nebo roven skutečné vzdálenosti. Přípustná heuristická funkce nám tedy zaručuje nalezení optimálního řešení (pokud existuje). Pro hledání cesty je možné použít i nepřípustnou heuristickou funkci, ale za cenu, že nalezené řešení nemusí být optimální.

V problému hledání nejkratší cesty mezi dvěma stavy se nabízí využití teorie metrických prostorů, a to přesněji využití vybraných metrik, viz [15]. Mezi nejznámější metriky patří euklidovská metrika, která ve své podstatě představuje vzdálenost mezi dvěma stavy $A = (a_1, a_2)$ $B = (b_1, b_2)$ jako délku úsečky, která tyto dva body spojuje, viz. obrázek 2.1a. Matematicky lze euklidovskou metriku v $\mathbb{R}^2$ zapsat jako

$$\rho(A, B)_1 = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}. \tag{2.3}$$

Další metrikou je manhattanská, neboli součtová metrika, která je inspirovaná pravoúhlým systémem ulic v New Yorku na Manhattanu, viz obrázek 2.1b. Matematický zápis v $\mathbb{R}^2$ je

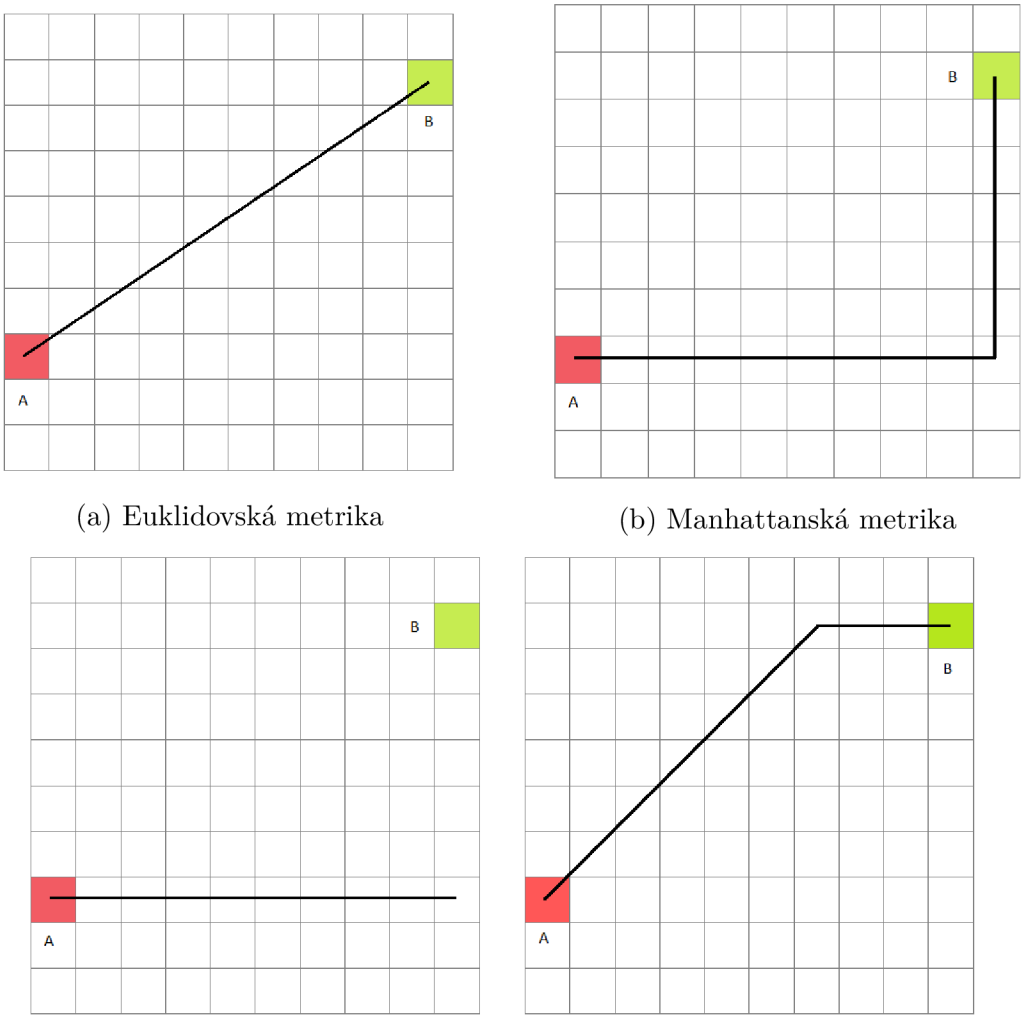
$$\rho(A, B)_2 = |a_1 - b_1| + |a_2 - b_2|. \tag{2.4}$$

Další známou metrikou je maximální (šachovnicová, Čebyševova) metrika, která představuje vzdálenost mezi dvěma body jako největší z jejich absolutních hodnot rozdílů podél jakékoliv souřadnicové osy, viz obrázek 2.1c. Odpovídající matematický zápis v $\mathbb{R}^2$ je

$$\rho(A, B)_{\infty_1} = max(|a_1 - b_1|, |a_2 - b_2|). \tag{2.5}$$

Podobnou metrikou, jako je metrika manhattanská, je metrika octile, která na rozdíl od manhattanské metriky pracuje s diagonálním směrem pohybu, viz obrázek 2.1d. Matematicky lze definovat tuto metriku jako

$$\rho(A, B)_{\infty_2} = max(|a_1 - b_1|, |a_2 - b_2|) + (\sqrt{2} - 1) \cdot min(|a_1 - b_1| + |a_2 - b_2|). \tag{2.6}$$



(a) Euklidovská metrika

(b) Manhattanská metrika

(c) Čebyševova metrika

(d) Octile metrika

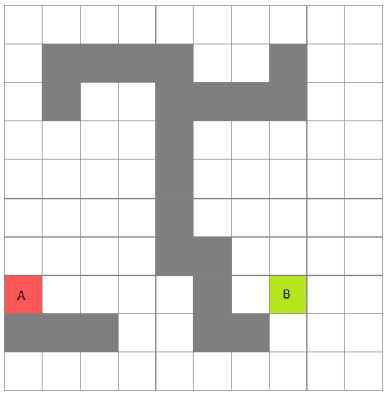
Obrázek 2.1: Grafická reprezentace metrik, kde červené políčko A představuje počáteční stav a zelené políčko B představuje koncový stav

Výběr metriky pro aplikaci v heuristické funkci pro hledání optimální cesty závisí na prohledávacím prostoru, tedy jestli použijeme mřížku, kde je povolen pohyb ve čtyřech nebo osmi směrech. Mahatannská metrika je použitelná pouze v mřížce, kde je pohyb umožněn pouze ve čtyřech směrech (horizontální a vertikální směr). Ve čtvercové mřížce, kde je umožněn pohyb v osmi směrech, je možné použít metriku octile či maximální. Euklidovskou metriku je možné využít v mřížkách s libovolným směrem pohybu.

Nyní k samotnému algoritmu. Je vytvořena prioritní fronta (seznam OPEN), jejíž prvky jsou čtveřice (stav i, hodnota $f(i)$, hodnota $g(i)$, předchůdce stavu i). Tato fronta je uspořádaná podle hodnoty funkce f. Do této prioritní fronty je přidán počáteční stav s nulovou hodnotou funkce f. Dokud není prioritní fronta vyprázdněna, tak se opakují následující kroky. Z prioritní fronty se vybere stav i s nejnižší hodnotou funkce f. Pokud se jedná o cílový stav, výpočet je u konce. V opačném případě se opakují následující kroky. Pro sousední stavy stavu i se vypočtou hodnoty f a g a tyto stavy se zapíšou do prioritní fronty. Pokud už tyto nově prozkoumané stavy v prioritní frontě jsou a mají menší hodnotu funkce g než tu nově vypočtenou, do fronty se nepřidávají. V opačném případě

jsou jejich hodnoty funkcí f a g přepsány na nové menší a jsou změněni jejich předchůdci. Stav i se z prioritní fronty odstraní a přesune se do seznamu CLOSE. Algoritmus končí ve chvíli, kdy je vybrán z prioritní fronty cílový stav nebo když se prioritní fronta vyprázdní. V tomto případě cesta neexistuje Pro názornost uvedu příklad.

Ve čtvercové mřížce na obrázku 2.2 se snažíme najít nejkratší cestu ze stavu A (červený čtvereček) do stavu B (zelený čtvereček), přičemž pohyb ze stavu i do stavu j je pouze ve vertikálním nebo horizontálním směru. Jako heuristická funkce je použita manhattanská metrika.

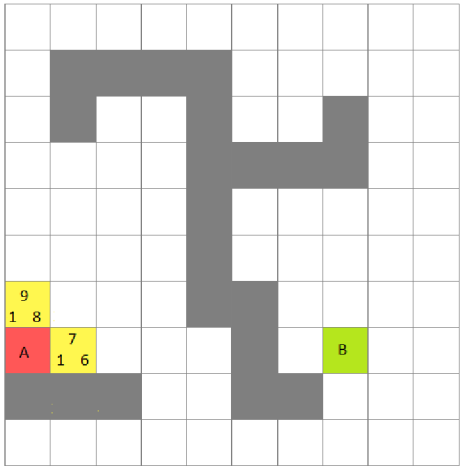


Obrázek 2.2: Výchozí stav problému

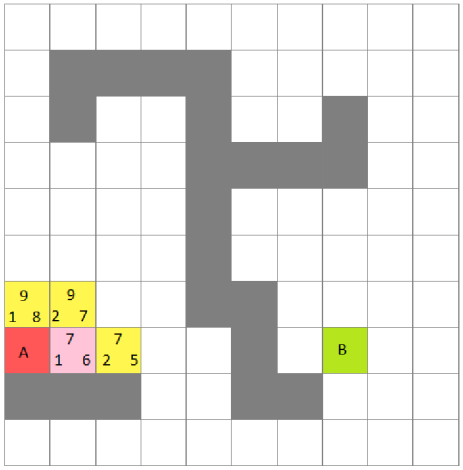
Pro upřesnění, hodnota nahoře uprostřed ve čtverečku značí hodnotu f, vlevo dole hodnotu g a vpravo dole hodnotu h, viz obr. 2.3.

Obrázek 2.3: Umístění hodnot uzlu v buňce reprezentující uzel grafu

Na začátku seznam OPEN obsahuje pouze počáteční stav A a tak je expandován zrovna on a jsou vypočteny hodnoty f a g pro jeho sousední stavy, viz obrázek 2.4



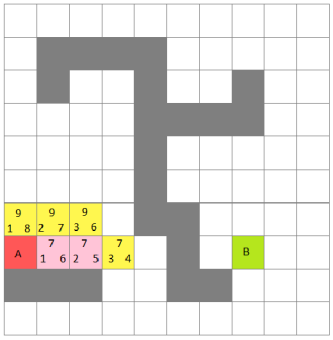
Obrázek 2.4: Krok 1



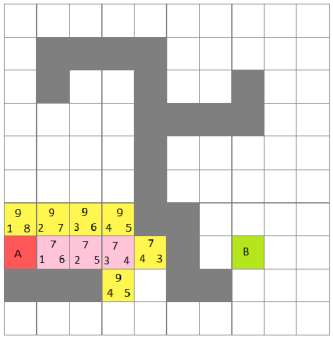
Obrázek 2.5: Krok 2

Nyní se všechny navštívené sousední stavy zapíšou do seznamu OPEN a stav A (počáteční stav) je odebrán a zapsán do seznamu CLOSE. Ze seznamu OPEN je vybrán takový

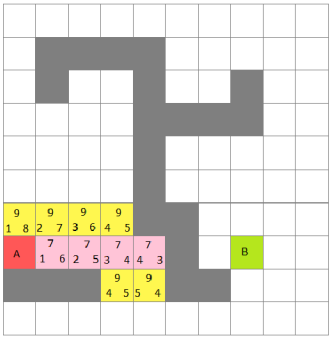
stav, který má nejmenší hodnotu funkce f a ten se expanduje (růžový čtvereček v obrázku 2.5). Opět jsou pro jeho sousedy vypočteny hodnoty funkcí f a g a tito sousedi jsou zapsáni do seznamu OPEN. Expandovaný stav se odebere ze seznamu OPEN a zapíše se do seznamu CLOSE. Nyní se ze seznamu OPEN vybere další stav, který se opět expanduje, obrázek 2.6a, odstraní se ze seznamu OPEN a zapíše se do seznamu CLOSE. Takto algoritmus pokračuje, dokud tedy není v seznamu OPEN koncový stav B nebo dokud není seznam OPEN vyprázdněn, což by znamenalo, že cesta neexistuje. Následující kroky pro náš výchozí stav problému jsou popsány na obrázku 2.6. Zajímavá situace nastává



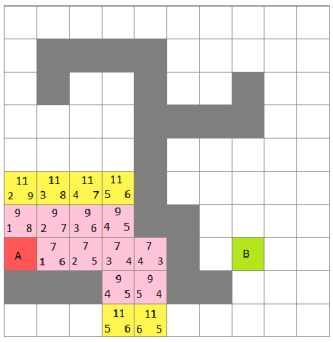
(a) Krok 3



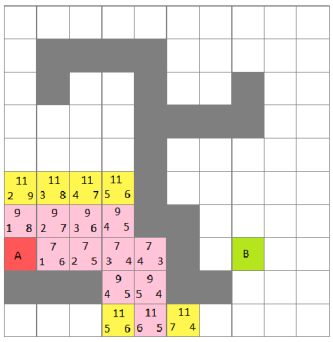
(b) Krok 4



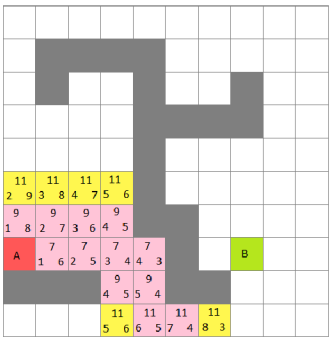
(c) Krok 5



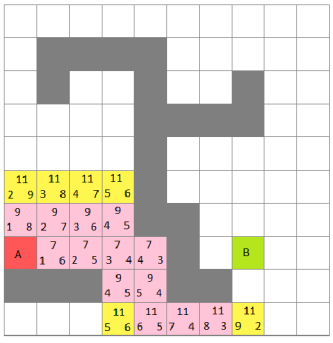
(d) Krok 6 až krok 11



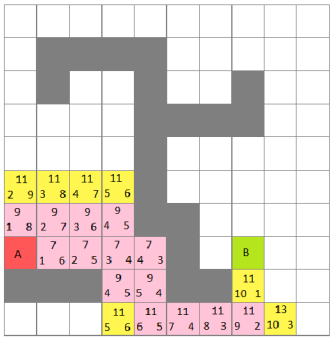
(e) Krok 12



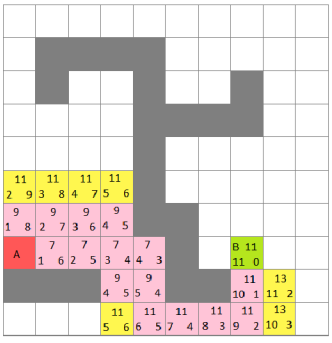
(f) Krok 13



(g) Krok 14



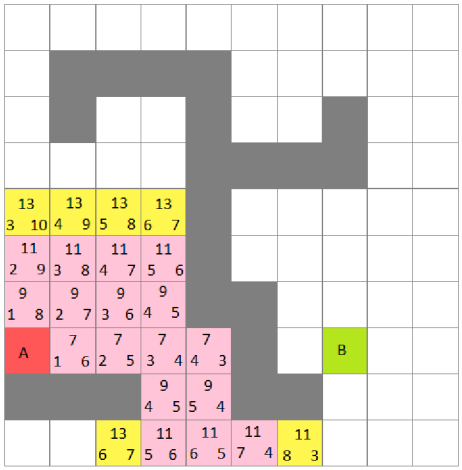
(h) Krok 15



(i) Krok 16

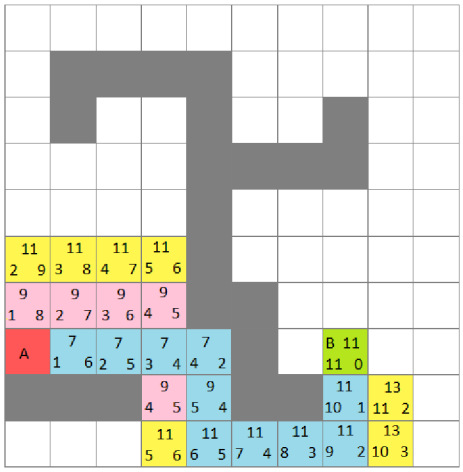
Obrázek 2.6: Jednotlivé kroky algoritmu A^*

v kroku 5, kdy v seznamu OPEN máme všechny stavy se stejnou hodnotou funkce f a to $f = 9$. V tomto případě je prostě vybrán libovolný z těchto stavů. Pro zjednodušení je na obrázku 2.6d zobrazen výsledek po provedení kroku 6 až kroku 11. V kroku 12, 13 a 14 je opět řešen problém s tím, že jsou v seznamu OPEN opět stavy se stejnou hodnotou funkce f. Já provedla ideální výběr stavů, které vedlo rychleji k nalezení nejkratší cesty.

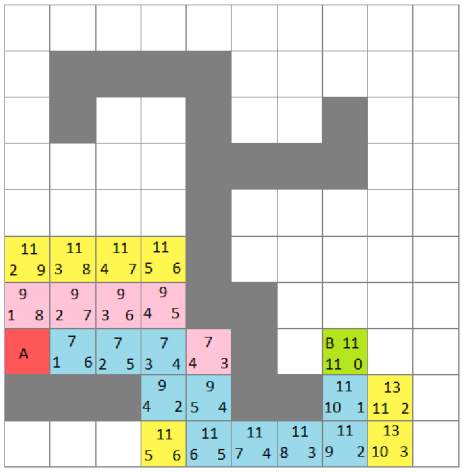


Obrázek 2.7: Prohledaný prostor v neideálním případě

V opačném, tj. neideálním stavu, by prohledaný prostor po těchto krocích vypadal jako na obrázku 2.7. Ovšem nutno dodat, že i v tomto případě by naše hledání bylo úspěšné. Výsledných nejkratších cest je více a dvě z nich jsou zobrazeny na obrázcích 2.8 a 2.9. Jedná se o modré čtverečky mezi stavy A a B.



Obrázek 2.8: Nejkratší cesta 1



Obrázek 2.9: Nejkratší cesta 2

Časová náročnost algoritmu A* závisí na použité heuristické funkci (viz výše). V nejhorším případě počet expandovaných uzlů roste exponenciálně. V optimálním případě je časová náročnost polynomiální, jestliže prohledávaný prostor má strukturu stromu, existuje pouze jeden cílový stav a heuristická funkce $h(i)$ splňuje následující podmínku

$$|h(i) - h^*(i)| = O(logh^*(i)),$$

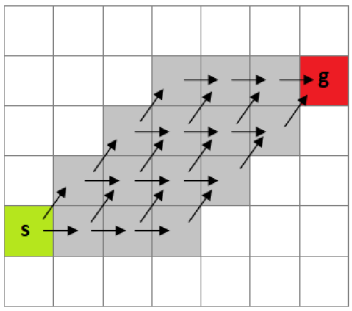
kde h^* je přesná vzdálenost uzlu i do koncového uzlu [16]. Níže je uveden pseudokód algoritmu A^*.

Algoritmus 1 A^* algoritmus

```
1: open ⟵ start
2: closed ⟵ ∅
3: g(start) = 0
4: while open¬∅ do
5:     u ← Extract − Min(open)
6:     if u == t break then
7:     end if
8:     for all edge e = (u, v) do
9:         distance ← g[u] + weight[e]
10:        if v ∉ open then
11:            g[v] ← distance
12:            h[v] ← Distance − To − Target(v, t)
13:            f[y] ← g[v] + h[v]
14:            open ← v
15:        else
16:            if distance < g[v] then
17:                g[v] ← distance
18:                h[v] ← Distance − To − Target(v, t)
19:                f[y] ← g[v] + h[v]
20:            end if
21:        end if
22:    end for
23:    closed ← u
24: end while
```

2.2 Algoritmus JPS

Algoritmus JPS, neboli Jump Point Search, je algoritmem, který zefektivňuje hledání cesty v mřížce. V mřížce, jakožto v prostoru, který nám simuluje náš prohledávací prostor, se setkáváme s vlastností symetrie. Znamená to, že některé cesty (nebo části cest) mají stejný počáteční a koncový bod, mají stejnou délku a vlastně se liší pouze v části stavů a v krajním případě se mohou lišit ve všech. Případ s vícero možnými nejkratšími cestami je zobrazen na obrázku 2.10.



Obrázek 2.10: Případ s vícero možnými nejkratšími cestami

Tato symetrie může zapříčinit to, že prohledávací algoritmus prochází zbytečně mnoho ekvivalentních cest. A právě algoritmus JPS tuto symetrii redukuje tím, že graf dle jistých pravidel prořezává, dokud nenajde nějaké „zajímavé“ stavy, které pak dále expanduje. Tuto metodu hledávání cest v mřížce, představili Daniel Harabor a Alban Grastier v roce 2011 ve svém článku *Online graph pruning for pathfinding on grid maps* [18] a dále pak rozšířili v článku *The JPS Pathfinding System* [19].

Algoritmus JPS vychází z algoritmu A*. Stále využíváme heuristickou funkci a seznamy OPEN a CLOSE. Liší se pouze v tom, že využívá jistý operátor pro expanzi uzlů, který se opírá o dva druhy pravidel, a to *pravidla pro prořezávání* a *pravidla pro body skoku*. Tato pravidla si představíme v následujících dvou podkapitolách.

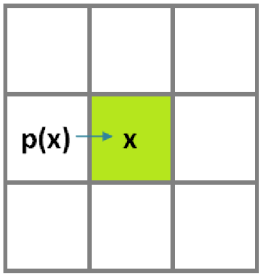
2.2.1 Prořezávací pravidla

Prořezáváním rozumíme ignorování jistých uzlů, jejichž expanze by k ničemu nevedla a bylo by zbytečné je přidávat do seznamu OPEN. Mějme uzel x, jeho předchůdce p a následníka y uzlu p, který je různý od x. Dále pak cestou $\pi = \langle n_0, n_1, ...n_k \rangle$ rozumíme cestu začínající v n_0 a končící v n_k. Obecně pro prořezání (ignorování) uzlu n, který je sousedem uzlu x, musí být podle [19] splněna jedna z následujících podmínek:

1. Existuje cesta $\pi' = \langle p, y, n \rangle$ nebo $\pi' = \langle p, n \rangle$, která je kratší než cesta $\pi = \langle p, x, n \rangle$.
2. Existuje cesta $\pi' = \langle p, y, n \rangle$ nebo $\pi' = \langle p, n \rangle$, která má stejnou délku jako $\pi = \langle p, x, n \rangle$, ale π' má diagonální pohyb dříve než π.

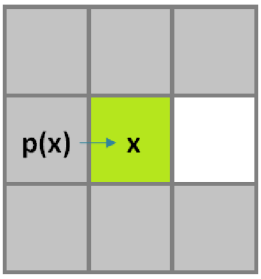
K prořezávání dochází ve dvou směrech, a to v přímém a diagonálním směru. Pro přehlednost si vysvětlíme jednotlivé směry zvlášť.

V přímém směru prořežeme (ignorujeme) sousední uzel n uzlu x v případě, že se do tohoto uzlu dostaneme z předchůdce p kratší nebo stejnou cestou, jako bychom se do tohoto uzlu dostali z uzlu p cestou vedoucí přes x. Na obrázku 2.11 je uveden příklad. V tomto případě mřížka neobsahuje žádnou překážku, tzn. přes všechny uzly lze přejít.



Obrázek 2.11: Výchozí stav při přímém prořezávání

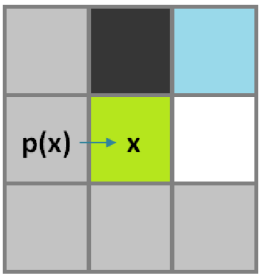
Předchůdcem uzlu x, je uzel $p(x)$. „Cena“ přechodu z uzlu do uzlu v přímém přechodu má například hodnotu 1 a v diagonálním $\sqrt{2}$. Na obrázku 2.12 je znázorněn princip prořezání. Všechny uzly, které jsou reprezentovány šedými políčky, jsou prořezány.



Obrázek 2.12: Princip přímého prořezávání

Prořezány jsou tedy všechny uzly kromě uzlu napravo od x. Takový sousední uzel uzlu x se dle [18] nazývá *přirozený soused* uzlu x.

Nyní situaci trochu pozměníme, a to tak, že nad uzel x umístíme překážku, viz obrázek 2.13.



Obrázek 2.13: Přímé prořezávání v prostoru s překážkou

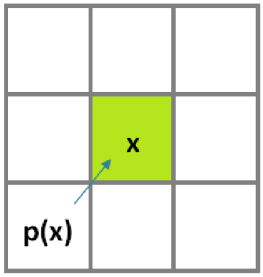
V tomto případě nemohl být prořezán uzel reprezentován modrou buňkou. Takový uzel se dle [18] nazývá *vynucený*, viz definice 1.

Definice 1. Nechť n je sousední uzel uzlu x. Uzel n nazveme *vynuceným* jestliže:

1. Uzel n není přirozeným sousedem uzlu x
2. Cesta $\pi = \langle p, x, n \rangle$ je kratší jak cesta $\pi' = \langle p, ..., n \rangle$, která neobsahuje x

Tato definice je velice důležitá, neboť propojuje pravidlo pro prořezávání grafu s pravidlem pro určování bodu skoku, které bude uvedeno v následující podkapitole.

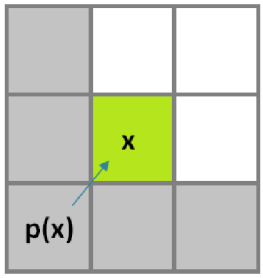
U diagonálního prořezávání je princip podobný. Prořežeme sousední uzel n uzlu x v případě, že se do tohoto uzlu dostaneme z předchůdce p striktně kratší cestou, jako



Obrázek 2.14: Výchozí stav při přímém prořezávání

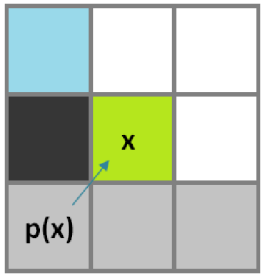
bychom se do tohoto uzlu dostali z uzlu p cestou vedoucí přes x. Na obrázku 2.14 je uveden výchozí stav případu.

Zde je předchůdcem uzlu x uzel $p(x)$ a „cena“ přechodu z uzlu do uzlu v přímém směru má opět hodnotu 1 a v diagonálním $\sqrt{2}$. Na obrázku 2.15 je znázorněn princip prořezání. Všechny uzly, které jsou reprezentovány šedými políčky, jsou prořezány.



Obrázek 2.15: Princip diagonálního prořezávání

Zbylé uzly (bílé buňky) patří mezi přirozené sousedy uzlu x. Jiná situace nastává opět v případě, kdy se v prostoru vyskytuje překážka. Tento případ je zobrazen na následujícím obrázku 2.16. Černá buňka představuje překážku, prořezány jsou všechny šedé uzly a uzel reprezentován modrou buňkou je dle definice 1 vynucený soused.



Obrázek 2.16: Princip diagonálního prořezávání

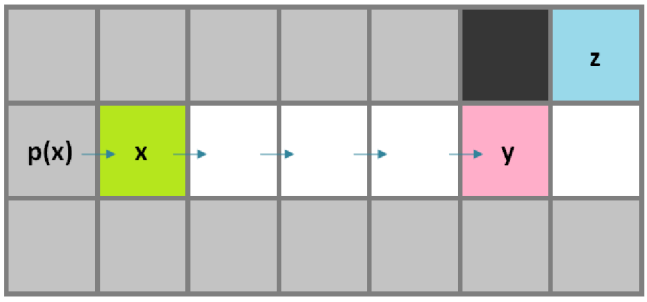
2.2.2 Pravidlo pro určování bodů skoku

Jak již bylo řečeno na začátku této kapitoly, JPS narozdíl od A* expanduje pouze vybrané uzly. Takové uzly nazýváme *body skoku* (Jump points). Body skoku proto, protože pokud takový bod (uzel) v našem prohledávacím prostoru existuje, „přeskočíme“ do něj a expandujeme jej. Níže uvedená definice z [18] říká, jak tyto uzly určujeme.

Definice 2. Uzel y je bodem skoku z uzlu x ve směru $\overrightarrow{d}$, pokud y minimalizuje hodnotu k tak, že $y = x + k\overrightarrow{d}$ a platí jedna z následujících podmínek:

1. Uzel y je cílový uzel.
2. Uzel y má nejméně jednoho vynuceného souseda.
3. $\overrightarrow{d}$ je diagonální pohyb a existuje uzel $z = y + k_i\overrightarrow{d_i}$, který leží k_i ($k_i \in \mathbf{N}$) kroků ve směru $\overrightarrow{d_i} \in \left\{\overrightarrow{d_1}, \overrightarrow{d_2}\right\}$ a takový, že z je bodem skoku z y podle podmínky 1 nebo podmínky 2 této definice, a kde směry $\overrightarrow{d_1}, \overrightarrow{d_2}$, jsou přímé směry, které se směrem d svírají úhel 45°.

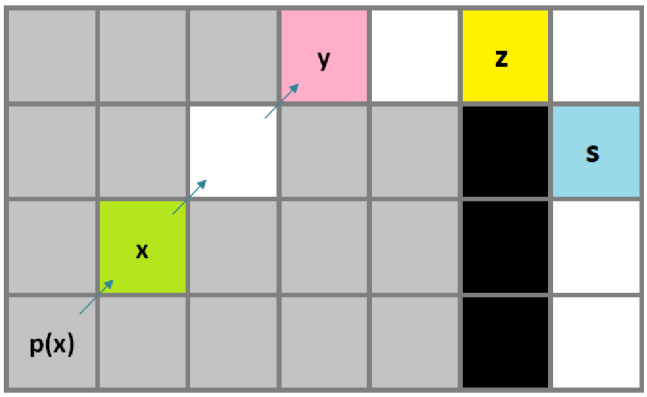
Podívejme se nyní na podmínku 2. Ta říká, že bodem skoku je takový uzel, který má nejméně jednoho vynuceného souseda. Vznik vynuceného souseda je vysvětlen v podkapitole 2.2.1. V případě přímého pohybu je vznik bodu skoku uveden na obrázku 2.17. Zde



Obrázek 2.17: Vznik bodu skoku (růžová buňka) při přímém pohybu

je expandován uzel x, jehož předchůdce je uzel $p(x)$. Šedé uzly nad a pod uzlem x jsou uzly, které odřezáváme (ignorujeme), dle podkapitoly 2.2.1. Vidíme, že zde není žádný „zajímavý“ uzel, a tak se posuneme o uzel dál. Podíváme se opět nad a pod uzel, opět nic „zajímavého“, a tak se pokračuje dál, dokud nenarazíme na uzel y. Nad ním se vyskytuje překážka, která způsobuje vznik vynuceného souseda z a tedy uzel y je našim bodem skoku. Uzel y je přidán do seznamu OPEN, je označen jako následník uzlu x a je mu přiřazena hodnota $g(y) = g(x) + c(x, y)$, stejně jako u algoritmu A^*. Takto jsme „skočili“ z uzlu x až na uzel y, bez toho, aby byl expandován jakýkoliv jiný uzel na cestě z x do y. U diagonálního pohybu je princip stejný.

Nyní se podívejme na podmínku 3 z definice 2. Takový případ je popsán na obrázku 2.18.



Obrázek 2.18: Vznik bodu skoku (růžová buňka) při diagonálním pohybu

Začínáme v uzlu x a posunujeme se postupně až k uzlu y. Uzel z je vzdálen od uzlu y o dva kroky (posuny) a má vynuceného souseda (modře vybarvená buňka s). Což

znamená, že uzel z je bodem skoku s předchůdcem y a y je bodem skoku s předchůdcem x. Oba dva uzly jsou přidány do seznamu OPEN. Všechny uzly šedé barvy jsou prořezány (ignorovány).

2.2.3 Popis algoritmu

Algoritmus JPS funguje stejně jako algoritmus A^* s tím rozdílem, že využívá jistý proces *Identify Successors* pro hledání množiny následníků (seznam OPEN) s rekurzivní funkci *jump*, viz [18].

U procesu *Identify Successors* pracujeme s množinou sousedů aktuálního uzlu x (*neighbours*), kteří nebyli při užití prořezávacích pravidel (podkapitola 2.2.1) odřezaní. Pomocí cyklu *for* (řádky 3 až 5 v pseudokódu, viz níže) se snažíme přidat do seznamu OPEN takový uzel (bod skoku), který je co nejdál od uzlu x a zároveň ve stejném směru jako je směr přechodu z uzlu x do libovolného sousedního uzlu n (např. pokud je uzel n napravo od x, hledáme bod skoku napravo od x atd.) Pokud se nám takový uzel podaří najít, přidáme jej do seznamu OPEN (množina *successors*), v opačném případě do seznamu OPEN nepřidáme nic. Proces pokračuje dokud množina sousedů (*neighbours*) není vyprázdněna. Níže je uveden pseudokód tohoto procesu.

Algoritmus 2 Identify Successors [18]

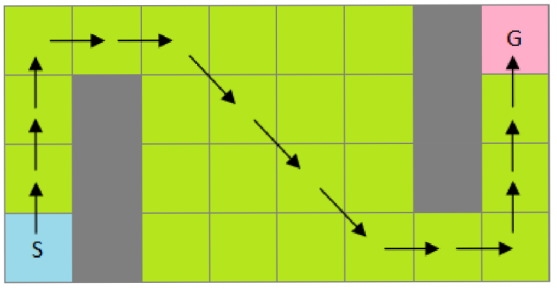
1: $successors(x) \longleftarrow \emptyset$
2: $neighbours(x) \longleftarrow prune(x, neighbours(x))$
3: **for all** $n \in neighbours(x)$ **do**
4: $\quad n \leftarrow jump(x, direction(x, n), s, g)$
5: $\quad$ add n to $successors(x)$
6: **end for**
7: **return** $successors(x)$

V korespondenci s tím, jak takové uzly klasifikovat (zda se jedná o naše body skoku), se využívá funkce *jump*. Ta vyžaduje kromě počátečního, koncového a aktuálního uzlu, také směr $\overrightarrow{d}$. Funkce totiž v tomto směru od x prochází uzel po uzlu a v korespondenci s definicí 2 zjišťuje, jestli se jedná o bod skoku.

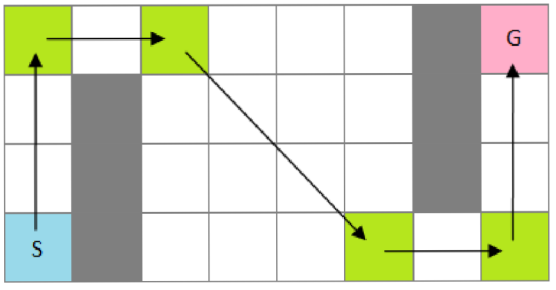
Algoritmus 3 Function *jump* [18]

```
1: n ← step(x, d)
2: if n is an obstacle or is outside the grid then
3:     return null
4: end if
5: if n = g then
6:     return n
7: end if
8: if ∃n' ∈ neighbours(n) s.t. n' is forced then
9:     return n
10: end if
11: if d is diagonal then
12:     for all i ∈ {1, 2} do
13:         if jump(n, d_i, s, g) is not null then
14:             return n
15:         end if
16:     end for
17: end if
18: return jump(n, d, s, g)
```

Pro vysvětlení, proč je algoritmus JPS lepší než algoritmus A^* je na obrázcích 2.19 a 2.20 zobrazeno srovnání algoritmů vzhledem k obsahu seznamu OPEN.



Obrázek 2.19: A^* algoritmus - zelené buňky značí buňky přidané do seznamu OPEN



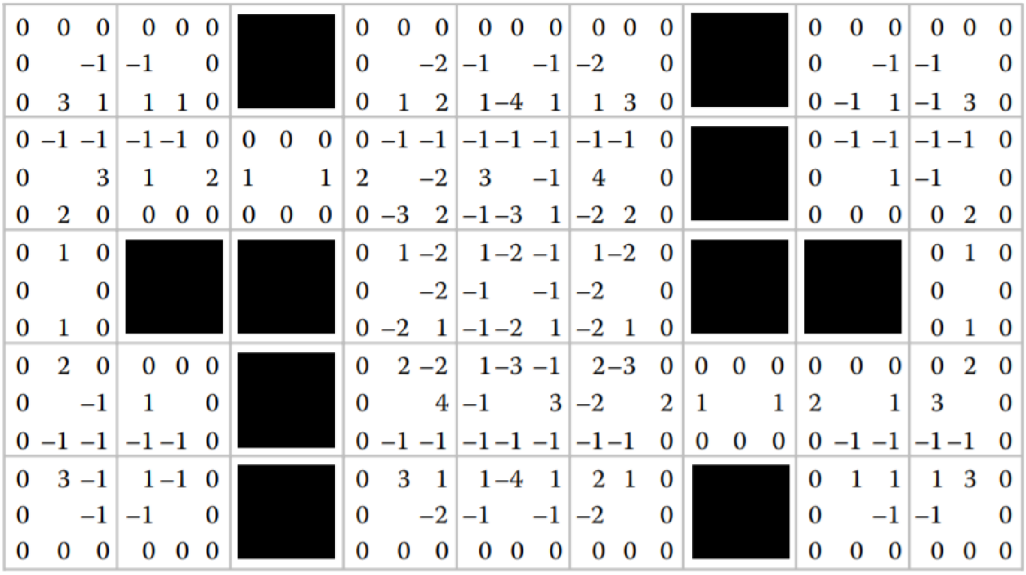
Obrázek 2.20: JPS algoritmus - zelené buňky značí buňky přidané do seznamu OPEN

2.2.4 Algoritmus JPS+

Daniel Harabor a Alban Grastier ve svém článku *The JPS Pathfinding System* [19] představili jisté vylepšení algoritmu JPS a to algoritmus JPS+. Skoky z jednoho uzlu mřížky do druhého se vyhneme mnoha zbytečným operacím v rámci seznamu OPEN. Nicméně identifikace těchto bodů skoku sebou přináší jistá úskalí a proto algoritmus JPS+ představuje vylepšení a to v prolomení symetrie, která nahradí každý sousední uzel, který leží ve stejném relativním směru, bodem skoku. Algoritmus rozděluje body skoků do čtyř skupin.

1. *Hlavní body skoku*, což jsou body, které mají vynuceného souseda.
2. *Přímé body skoku*, což jsou uzly ze kterých by pohyb v přímém směru vedl na nalezení hlavního bodu skoku.
3. *Diagonální body skoku*, což jsou uzly ze kterých by pohyb v diagonálním směru vedl na nalezení hlavního nebo přímého bodu skoku.
4. *Cílové body skoku*, což jsou cílové uzly.

Algoritmus provede předzpracování (tzv. preprocessing) stavového prostoru, jako je například zobrazeno na obrázku 2.21.



Obrázek 2.21: Předzpracování stavového prostoru [17]

Čísla v jednotlivých buňkách označují následující. Vezmeme si například buňku vlevo nahoře. Nuly a záporné číslo označují vzdálenost od konce mapy. Hodnota 3 značí, o kolik uzlů je vzdálen základní bod skoku. Hodnota 1 dole vpravo v rohu značí, že po provedení jednoho diagonálního tahu se dostaneme do základního bodu skoku. Taktové předzpracování prostoru může být dle [17] provedeno následujícím postupem:

1. Identifikace všech základních bodů skoku nastavením směrového příznaku v každém uzlu.
2. Označení vzdáleností všech přímých západních bodů skoku a západní stěny procházením mapy zleva doprava.

3. Označení vzdáleností všech přímých východních bodů skoku a východní zdi procházením mapy zprava do leva.

4. Označení vzdáleností všech severních přímých bodů skoku a severní zdi při procházení mapy shora dolů.

5. Označení vzdáleností všech jižních přímých bodů skoku a jižní stěny při procházení mapy zezdola nahoru.

6. Označení vzdáleností všech jihozápadních/jihovýchodních diagonálních bodů skoku a stěn procházením mapy zezdola nahoru.

7. Označení vzdáleností všech severozápadních/severovýchodních diagonálních bodů skoku a stěn procházením mapy shora dolů.

To co zvýhodňuje algoritmus JPS+ s jeho předzpracováním mapy je, že obsahuje mnoho rozhodnutí požadovaných pro vyhledávání, čímž je časová náročnost menší a proto je rychlejší než tradiční JPS. Například rekurzivní funkce *jump* (Algoritmus 3, na straně 31) je zcela odstraněna.

2.2.5 JPS s omezováním prostoru

Steve Rabin a Nathan R. Sturtevant ve svém článku *Combining Bounding Boxes and JPS to Prune Grid Pathfinding* [8] představili další možnost pro předzpracování stavového prostoru a to prostřednictvím využití jistého geometrického ohraničení (tzv. bounding boxu) pouze určité části stavového prostoru, na kterou bude omezeno prohledávání. Tyto bounding boxy jsou obecnou třídou přístupů k řešení problémů, které lze popsat v metrickém prostoru. V rámci předzpracování, cílový stav dosažitelný pomocí určité hrany (a sousedního stavu) v grafu je umístěn uvnitř příslušného bounding boxu. Při běhu algoritmu je pak pohyb ze stavu omezen na ty hrany, jejichž bounding box obsahuje cílový stav. Existují dva způsoby, jak můžeme předem vypočítat bounding box pro daný stav a sousední hranu. Prvním z nich je vybrat stav a dotazovat se všech ostatních stavů s příslušnými hranami, zda má být tento stav součástí bounding boxu daného stavu a hrany (tj. jestli je dosažitelný na optimální cestě začínající v tomto stavu s touto hranou). Druhou možností je provést výpočet nejkratší cesty z uzlu a přímo vypočítat bounding box tohoto uzlu z výsledného vyhledávání.

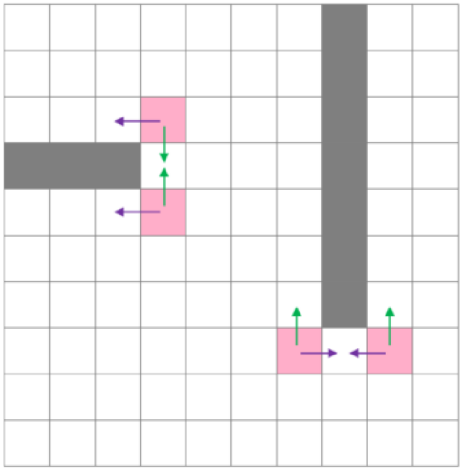
Bounding box lze kombinovat i například s algoritmem A^*. Tato práce implementuje vlastní návrh bounding boxu, jehož konstrukce je vysvětlená v poslední kapitole, kde byly prováděny porovnávací experimenty.

2.3 Algoritmus Subgoal

Tansel Uras, Sven Koenig a Carlos Hernández v roce 2013 představili ve svém článku *Subgoal graphs for optimal pathfinding in eight-neighbor grids* [20] algoritmus Subgoal. Tento algoritmus je založen na předzpracování prohledávacího prostředí reprezentovaného mřížkou s pohybem možným v osmi směrech. V rámci tohoto předzpracování algoritmus vytvoří jistý podgraf v mřížce, který se skládá z vrcholů (podcílů) umístěných v rozích překážek a hran spojujících ty vrcholy, které jsou navzájem přímo dosažitelné. Tyto hrany jsou dostačující pro nalezení nejkratších cest a jsou snadno sledovatelné. Algoritmus nejprve spojí start a cíl s grafem podcílů, poté hledá nejkratší cestu přes graf podcílů a sledováním této cesty hledá nejkratší cestu v mřížce. Autoři představili dva algoritmy, nejdříve *Jednoduchý Subgoal algoritmus* a poté *Dvouúrovňový Subgoal graf*. Pro popis těchto algoritmů je nejdříve nutné definovat následující pojmy.

Definice 3. *Podcílem* (neboli tzv. subgoalem) v mřížce rozumíme nezablokovanou buňku s, jestliže existují dva kolmé přímé směry c_1 a c_2, takové že pohyb ve směru $s + c_1 + c_2$ je zablokován a pohyby ve směrech $s + c_1$ a $s + c_2$ zablokované nejsou.

V definici 3 je jako výsledek součtu směrů $c_1 + c_2$ diagonální směr pohybu $d = c_1 + c_2$. Příklad podcíle je vyobrazen na obrázku 2.22. Šedé buňky představují překážky (zablokované buňky) a růžové buňky pak podcíle. Jednoduše řečeno se podcíle umísťují vždy do rohů překážek.



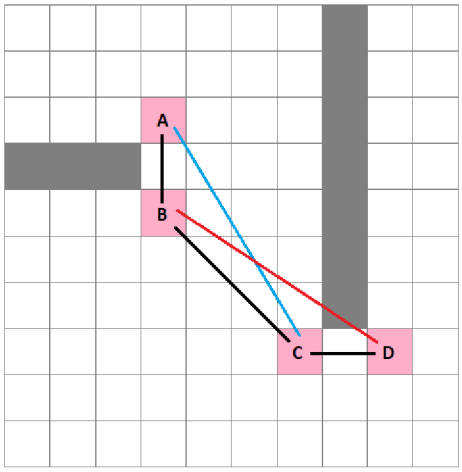
Obrázek 2.22: Umístění podcílů v mřížce, šipky značí směry c_1 a c_2, z definice 3

Pro následující definici je vhodné vysvětlit, jaký je rozdíl mezi *trajektorií* a *cestou*. Trajektorie mezi dvěmi buňkami s a s' je posloupnost pohybů v mřížce, která dostane robota ze stavu s do stavu s' při odstranění všech překážek. Cesta je trajektorie, která dostane robota ze stavu s do stavu s' v aktuální mřížce (tzn. za přítomnosti překážek).

Definice 4. Dvě buňky s a s' jsou *dosažitelné*, jestliže mezi němi existuje cesta délky $h(s, s')$. Dvě dosažitelné buňky s a s' jsou *bezpečně dosažitelné*, jestliže všechny nejkratší trajektorie mezi nimi jsou také cesty. Dvě bezpečně dosažitelné buňky s a s' jsou *přímo dosažitelné*, pokud žádná z nejkratších cest mezi nimi neobsahuje podcíl $s'' \notin \{s, s'\}$.

V definici 4 je uvažována délka, neboli vzdálenost dvou buněk $h(s, s')$ jako metrika *octile*, dle rovnice (2.6), která je definovaná v podkapitole 2.1 jako kombinace diagonálního

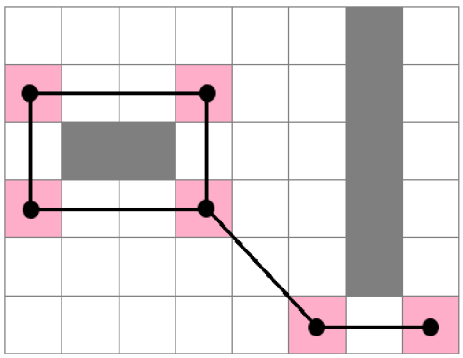
a přímého směru pohybu. Obrázek 2.23 znázorňuje subgoaly a jejich vzájemná spojení. A, B, C a D jsou subgoaly. $A - B$, $B - C$ a $C - D$ jsou *přímo dosažitelné*. $A - C$ jsou *bezpečně dosažitelné* ale nejsou *přímo dosažitelné*, protože nejkratší cesta mezi nimi vede přes podcíl B. $B - D$ jsou *dosažitelné* ale nejsou *bezpečně dosažitelné*, protože nejkratší trajektorie mezi nimi je blokovaná.



Obrázek 2.23: Subgoaly a jejich vzájemná spojení

Jakékoliv dvě *přímo dosažitelné* buňky s a s' jsou rovněž *bezpečně dosažitelné* a to z toho důvodu, že pokud by nebyly bezpečně dosažitelné, pak by existovala překážka, která by blokovala alespoň jednu z nejkratší trajektorii. Tato překážka by buď blokovala všechny nejkratší trajektorie mezi nimi (a nebyly by přímo dosažitelné), nebo by zavedla podcíl, který leží na jedné z nejkratších cest mezi nimi, což by bylo v rozporu s definicí přímé dosažitelnosti.

Definice 5. *Jednoduchý subgoal graf* $G_S = (V_S, E_S)$ je neorientovaný graf, kde V_S je množina podcílů a E_S je množina hran spojujících navzájem přímo dosažitelné podcíle. Délky hran jsou vzdálenosti podcílů ve smyslu octile metriky.



Obrázek 2.24: Jednoduchý subgoal graf

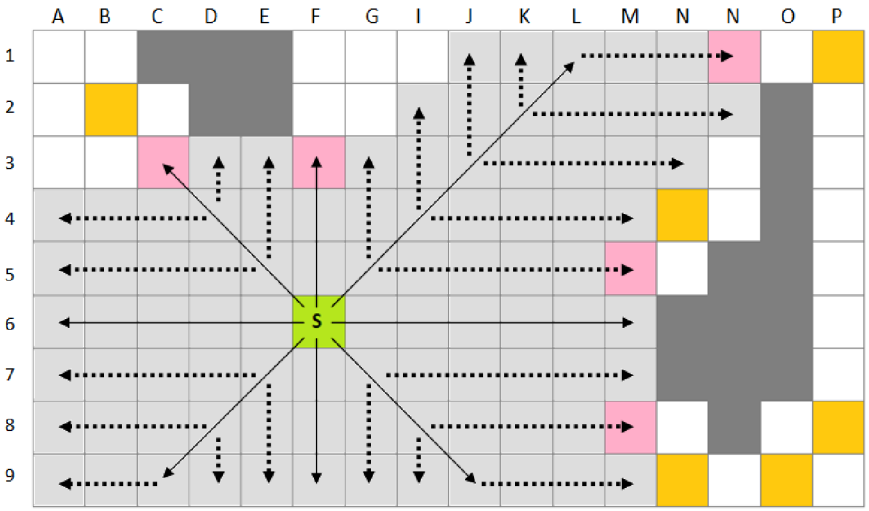
Níže je uveden Algoritmus 4, který vytváří jednoduchý subgoal graf, přičemž funkce *GetDirectHReachable(s)* vrátí množinu podcílů, které jsou přímo dosažitelné z buňky s. Získávání množiny podcílů je v rámci implementace složité a proto je této části věnována následující podkapitola.

Algoritmus 4 Konstrukce jednoduchého subgoal grafu [20]

```
1:  procedure ConstructSubgoalGraph( )
2:      V_S := E_S := ∅;
3:      for all unblocked cells s do
4:          for all perpendicular cardinal directions c_1 and c_2 do
5:              if s + c_1 + c_2 is blocked then
6:                  if s + c_1 and s + c_2 are unblocked then
7:                      V_s := V_s ∪ {s};
8:                  end if
9:              end if
10:         end for
11:     end for
12:     for all s ∈ V_S do
13:         S ← GetDirectHReachable(s);
14:         for all s' ∈ S do
15:             E_s := E_s ∪ {(s, s')}
16:         end for
17:     end for
18:     G_s := (V_S, E_S)
19: end procedure
```

2.3.1 Hledání přímo dosažitelných podcílů

Rychlá identifikace všech přímo dosažitelných podcílů z určitého stavu je důležitá jak pro sestavování jednoduchého subgoal grafu, tak i pro připojení startovní a cílové pozice do tohoto grafu. Algoritmus 5, který je uveden v závěru této podkapitoly, identifikuje všechny přímo dosažitelné stavy z určitého stavu s a vrací všechny podcíle mezi němi. Jak algoritmus funguje si ukážeme pomocí příkladu z obrázku 2.25.



Obrázek 2.25: Hledání přímo dosažitelných podcílů

V tomto obrázku, jsou růžově vyznačeny podcíle, které jsou přímo dosažitelné ze stavu s, oranžově podcíle, které nejsou přímo dosažitelné ze stavu s. Nepřerušované šipky hledají

přímo dosažitelné stavy ze stavu s, které mohou být nalezeny pohybem pouze v jednom směru a přerušované šipky hledají přímo dosažitelné podcíle ze stavu s, které mohou být nalezeny kombinací dvou směrů.

Algoritmus pracuje s tzv. *clearance* hodnotami. Tato hodnota buňky s ve směru d je získána pomocí funkce $Clearance(s, d)$ a jedná se o maximální počet pohybů, který může robot vykonat ze stavu s skrze směr d bez tohoto, aniž by narazil do podcíle či byl zastaven překážkou. Například, východní *clearance* hodnota pro stav s z příkladu na obrázku 2.25, je 6, protože buňka $M6$ je zablokovaná překážkou. Severní *clearance* hodnota je 2, protože $F3$ je podcíl. Tyto *clearance* hodnoty mohou být vypočteny za běhu nebo mohou být vypočteny předem. Algoritmus identifikace přímo dosažitelných podcílů pracuje ve dvou fázích:

1. **První fáze**
   V první fázi se indentifikují všechny přímo dosažitelné stavy ze stavu s, které mohou být dosaženy pouze v jednom směru. Toho je dosaženo sledováním *clearance* hodnot stavu s ve všech možných osmi směrech pohybu z tohoto stavu, aby se zjistilo jestli v těchto směrech jsou nějaké přímo dosažitelné podcíle. Například, severní *clearance* hodnota stavu s je 2 a algoritmus zkontroluje nejen pohyb o dva stavy nahoru, ale i o $2+1$, aby zjistil, jestli stav $F3$ je podcíl. Jak lze vidět na obrázku 2.25, výsledkem první fáze algoritmu je, že stavy $C3$ a $F3$ jsou přímo dosažitelné stavy ze stavu s.

2. **Druhá fáze**
   Ve druhé fázi se identifikují všechny přímo dosažitelné stavy ze stavu s, kterých lze dosáhnout kombinací pohybů v kardinálním směru c a diagonálním směru d. Je zde 8 možných kombinací kardinálních a diagonálních pohybů (na obrázku 2.25 jsou zobrazeny nepřerušovanou šipkou) a každá z těchto kombinací identifikuje odpovídající oblast. Algoritmus objevuje jednotlivé oblasti řádek (sloupec) po řádku (sloupci), jedná se o přerušované šipky na obrázku 2.25. Pro každý stav, který je přímo dosažitelný ze stavu s pohybem ve směru d se provede průzkum v kardinálním směru c. Začíná se řádkem (sloupcem) nejblíže ke stavu s a pokračuje se, dokud nejsou prozkoumány všechny řádky (sloupce).

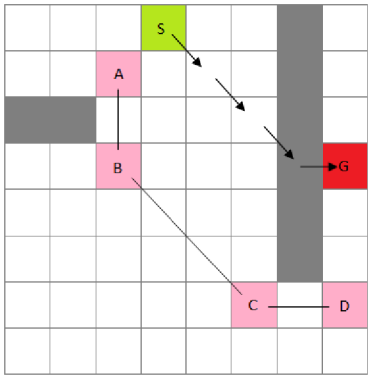
Nyní se aplikují tři pravidla, která říkají jak daleko každý z řádků (sloupců) prozkoumávat [23]. První z pravidel říká, že průzkum řádku (sloupce) končí ve chvíli, kdy je dosažen podcíl nebo přímo před tím, než narazí na překážku. Druhé pravidlo říká, že prozkoumaný řádek (sloupec) nemůže být delší než předchozí prozkoumaný řádek (sloupec). Třetí pravidlo rozšiřuje druhé pravidlo a říká, že prozkoumaný řádek (sloupec) nemůže být delší než předchozí a musí být o jeden stav menší, pokud předchozí řádek (sloupec) končil v podcíli. Například, pokud se podíváme na severovýchodní oblast od stavu s v obrázku 2.25, první řádek, který zkoumá, je řádek 5, u kterého algoritmus po 5 tazích narazí na podcíl $L5$. Protože průzkum tohoto řádku končí v podcíli, tak v následujícím řádku 4, může být provedeno pouze $5 - 1 = 4$ tahů a algoritmus se zastaví před podcílem $M4$, který není přímo dosažitelný ze stavu s.

Algoritmus 5 Hledání přímo dosažitelných podcílů [20]

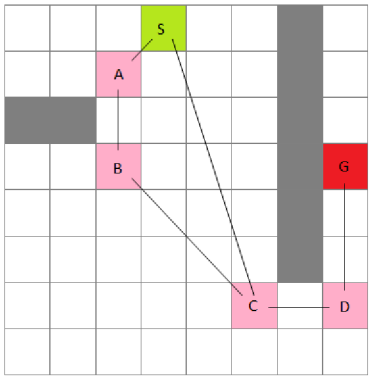
```
1: function Clearance(cell s, direction d)
2:     i := 0;
3:     while true do
4:         if movement not possible from s + id to s + id + d then
5:             return i;
6:         end if
7:         i := i + 1;
8:         if IsSubgoal(s + di) then
9:             return i;
10:        end if
11:    end while
12: end function
13: function GetDirectHReachable(cell s)
14:    S := ∅;
15:    for all directions d do
16:        if IsSubgoal(s + Clearance(s, d) × d) then
17:            S := S ∪ {s + Clearance(s, d) × d};
18:        end if
19:    end for
20:    for all diagonal directions d do
21:        for all cardinal directions c associated with d do
22:            max ⟵Clearance(s, c);
23:            diag ⟵Clearance(s, d);
24:            if IsSubgoal(s + max × c) then
25:                max := max − 1;
26:            end if
27:            if IsSubgoal(s + diag × d) then
28:                diag := diag − 1;
29:            end if
30:            for i = 1...diag do
31:                j :=Clearance(s + id, c);
32:                if j ≤ max and IsSubgoal(s + id + jc) then
33:                    S := S ∪ {s + id + jc};
34:                    j := j − 1;
35:                end if
36:                if j < max then
37:                    max := j;
38:                end if
39:            end for
40:        end for
41:    end for
42:    return S;
43: end function
```

2.3.2 Jednoduchý subgoal algoritmus

Jednoduchý subgoal algoritmus je založen na vytvoření jednoduchého subgoal grafu, viz def. 5, kdy umístí do rohů překážek podcíle a poté vytvoří mezi přímo dosažitelnými podcíli hrany. Pro nalezení nejkratší cesty mezi startovací a cílovou buňkou, je potřeba nejdříve start a cíl spojit s jejich příslušnými přímo dosažitelnými podcíli a poté nalézt nejkratší cestu (tzv. vysokoúrovňovou) mezi startem a cílem na tomto modifikovaném subgoal grafu. Poté algoritmus nalezne nejkratší cestu (tzv. nízkoúrovňovou) mezi startem a cílem nalezením nejkratších cest na mřížce mezi po sobě jdoucími podcíli na cestě a poté jejich spojením. Algoritmus 6 hledání nejkratší cesty popisuje. Pro lepší pochopení popíšeme algoritmus na příkladě. V prvním kroku algoritmu se ověří, jestli startovací pozice S a cílová pozice G nejsou přímo dosažitelné, viz obr. 2.26, kde růžové buňky A, B, C, D jsou podcíle, zelená buňka S je startovací pozice a červená buňka G je cílová pozice. Pokud jsou, tak algoritmus vrátí přímou cestu mezi nimi. V druhém kroku algoritmus

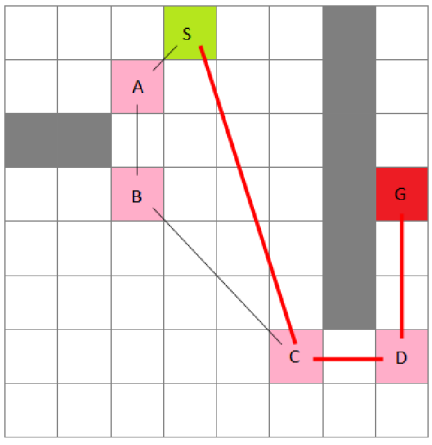


Obrázek 2.26: První krok



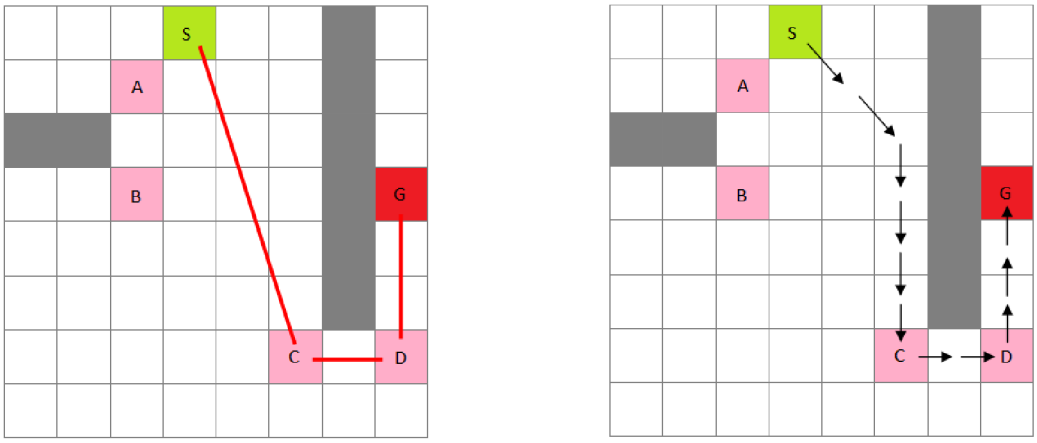
Obrázek 2.27: Druhý krok

spojí startovací a cílovou pozici s jejich odpovídajícími přímo dosažitelnými podcíli, viz obrázek 2.27. Tak vznikne jednoduchý subgoal graf. V třetím kroku algoritmus nalezne tzv. vysokoúrovňovou nejkratší cestu ze startovací pozice do cílové pozice prostřednictvím subgoal grafu, viz obrázek 2.28.



Obrázek 2.28: Třetí krok

V posledním kroku algoritmus nalezne nejkratší nízkoúrovňovou cestu následováním podcílů z vysokoúrovňové nejkratší cesty, viz obrázek 2.29.



Obrázek 2.29: Čtvrtý krok

Algoritmus 6 Prohledávání jednoduchého subgoal grafu [20]

```
1:  procedure ConnectToGraph(cell s)
2:      if s ∉ V_S then
3:          V_s := V_s ∪ {s};
4:          S ← GetDirectHReachable(s);
5:          for all s' ∈ S do
6:              E_s := E_s ∪ {(s, s')};
7:          end for
8:      end if
9:  end procedure
10: function FindAbstractPath(cells s, s')
11:     ConnectToGraph(s);
12:     ConnectToGraph(s');
13:     Π ←find a shortest path froms to s' over the modified graph;
14:     restore original graph;
15:     return Π;
16: end function
17: function FindPath(cells s, s')
18:     π ←TryDirectPath(s, s');
19:     if π ≠ nopath then
20:         returnπ;
21:     end if
22:     Π ← FindAbstractPath(s, s');
23:     if Π = nopath then
24:         return nopath;
25:     end if
26:     π := emptypath;
27:     for all segments(s_i, s_{i+1}) in Π, in increasing order of i do
28:         π := append(π, FindHReachablePath(s_i, s_{i+1}));
29:     end for
30:     return π;
31: end function
```

2.3.3 Dvouúrovňový subgoal algoritmus

Hledání nejkratší cesty pomocí jednoduchého subgoal grafu je rychlejší než vyhledávání v samotné mřížce. Mřížka je rozdělena na buňky, kde v jedné skupině jsou buňky, které se berou jako podcíle jednoduchého subgoal grafu a v druhé skupině jsou zbylé buňky, které se ignorují. Dvouúrovňový subgoal graf aplikuje tento nápad na jednoduchý subgoal graf. Podcíle jsou rozděleny do dvou skupin, a to na *lokální* a *globální* podcíle. Algoritmus pak ignoruje všechny lokální podcíle, které nejsou přímo dosažitelné ze startovní a cílové pozice, což opět vede ke zmenšení prohledávacího prostoru. Níže je uvedená definice dvouúrovňového subgoal grafu.

Definice 6. *Dvouúrovňový subgoal graf* $G_T = (V_T, E_T)$ pro daný jednoduchý subgoal graf $G_S = (V_S, E_S)$ je neorientovaný graf s následujícími vlastnostmi:

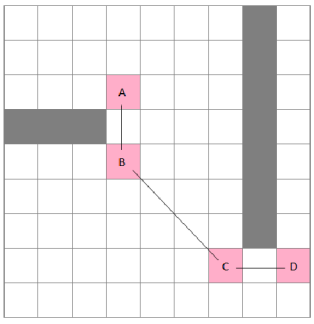
1. $V_T \subseteq V_S$ a $E_S \subseteq E_T \subseteq V_S \times V_S$.
2. Pro všechny $(s, s') \in E_T$, s a s' jsou dosažitelné.
3. Pro všechny $s, s' \in V_S$, vzdálenost mezi s a s' v dvouúrovňovém subgoal grafu $G'_T = (V_T \cup \{s, s'\}, E_T)$ je rovna vzdálenosti s a s' v jednoduchém subgoal grafu.

Konstrukce je následující. Začne se předpokladem, že všechny podcíle jsou globální a dvouúrovňový subgoal graf je nyní identický s jednoduchým subgoal grafem. Pro každý globální podcíl s dvouúrovňového subgoal grafu zkontrolujeme, zda je potřebný pro spojení jednoho nebo více párů jeho sousedních podcílů. Podcíl s není potřebný pro spojení podcílů s' a s'' jestliže platí alespoň jedna z následujících podmínek:

a) Existuje cesta mezi s' a s'' skrze pouze globální podcíle kromě s, která není delší jak $h(s', s) + h(s, s'')$.

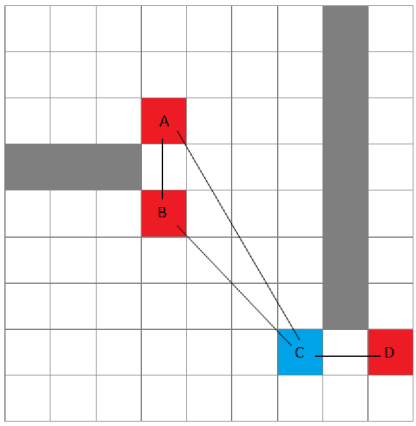
b) s' a s'' jsou dosažitelné.

Pokud tedy s není potřebný, tak je odstraněn z množiny V_T (takže se stane lokálním podcílem z množiny V_S/V_T) a jsou přidány hrany mezi všemi páry jeho sousedních podcílů, které splňují podmínku *b*) ale ne podmínku *a*). Startovní a cílová pozice je pak spojena právě s lokálními podcíli, a to s takovými, které jsou přímo dosažitelné. Konstrukci takového dvouúrovňového subgoal grafu si ukážeme na následujícím příkladě, kde je vyobrazen jednoduchý subgoal graf s podcíli A, B, C a D s příslušnými hranami, viz obr. 2.30.



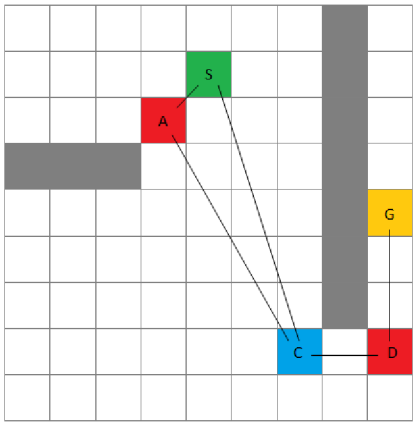
Obrázek 2.30: Jednoduchý subgoal graf

Tvar dvouúrovňového subgoal grafu vycházejícího z jednoduchého subgoal grafu závisí na pořadí v jakém jsou procházeny globální podcíle. V tomto případě podcíle prověřujeme například v pořadí A, B, C a D. A má pouze jeden sousední podcíl a jednoduše není potřeba pro spojení páru sousedních podcílů potřeba a je tedy následně odřezán. B má jeden pár sousedních podcílů A a C, což uspokojuje podmínku b), ale ne podmínku a). Následně je tedy B odřezán a je přidána hrana (A, C). C má 3 páry sousedících podcílů, ze kterých pár (A, D) nesplňuje ani podmínku a) ani podmínku b) a proto C není odřezáno. Nakonec podcíl D má pouze jeden sousední podcíl a jednoduše není potřebný pro propojení žádného páru sousedních podcílů. V důsledku toho dojde k odřezání podcíle D. Výsledný dvouúrovňový graf má pouze jeden globální podcíl C a přidanou hranu (A, C), viz obr. 2.31.



Obrázek 2.31: Dvouúrovňový subgoal graf, čevené buňky představují lokální podcíle a modrá buňka globální podcíl

Algoritmus v tomto případě postupuje následovně. V prvním kroku opět zkontroluje jestli startovací a cílová pozice nejsou přímo dosažitelné. Pokud ano, tak nám vrátí nejkratší cestu a algoritmus končí. V opačném případě se do vytvořeného dvouúrovňového subgoal grafu přidá startovní a cílový stav. Množina V_T je rozšířena o tyto dva stavy a množina E_T pak o ty hrany, které spojují start a cíl s přímo dosažitelnými podcíli a z těchto podcílů se následně do množiny V_T přidají ty podcíle, které tam ještě nejsou. Výsledný graf tedy obsahuje startovací pozici S, cílovou pozici G a podcíle A, C a D. Poté algoritmus postupuje stejně jako Jednoduchý subgoal algoritmus.



Obrázek 2.32: Spojení startu a cíle s dvouúrovňovým subgoal grafem

3 Implementace algoritmů

Pro implemetaci byly vybrány následující algoritmy:

- A^* algoritmus
- JPS algoritmus
- A^* algoritmus s kombinací bounding boxu ($A^* + BB$)
- JPS algoritmus s kombinací bounding boxu ($JPS + BB$)
- Jednoduchý subgoal algoritmus (SS)

Tato diplomová práce je zaměřená na pokročilé metody plánování cesty mobilního robotu a navazuje na diplomovou práci *Plánování cesty mobilního robotu* [6]. Podle této práce pak bylo voleno i uživatelské rozhraní.

3.1 Programovací prostředky užité pro návrh aplikace

Pro implementaci vybraných algoritmů bylo vytvořeno simulační prostředí mobilního robota prostřednictvím programovacího jazyka *Python* [25]. Ten je vyvíjen jako open source projekt a nabízí dynamickou kontrolu datových typů a podporuje různá programovací paradigmata, včetně například objektově orientovaného [21]. Základním prvkem objektově orientovaného programování (ve zkratce OOP) je *objekt*, který můžeme brát jako koncepčně ucelenou jednotku, který se skládá z několika částí:

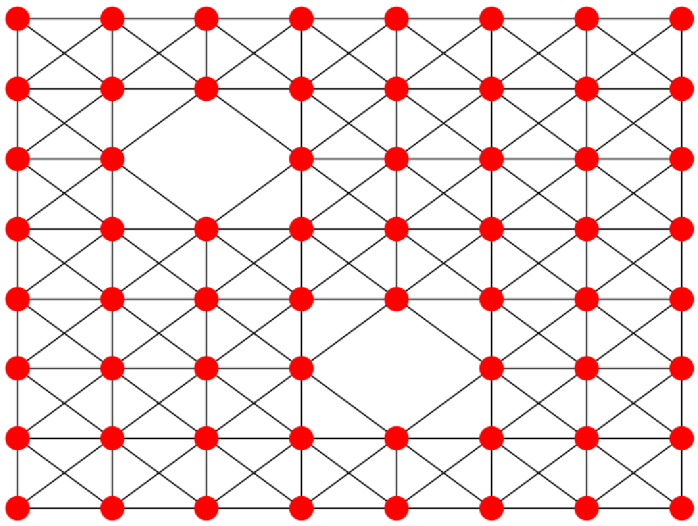
- vnitřní stav - soubor proměnných (atributy)
- vnitřní chování - soubor procedur (metody)
- protokol zpráv - veřejné rozhraní a způsob jeho mapování na vnitřní procedury objektu

Mezi další základní pojmy z OOP patří pojem *třída* a *instance třídy*. Třída charakterizuje vnitřní strukturu objektu a na základě ní je možné právě jednotlivé objekty tvořit. Instance třídy je pak tvořena podle dané třídy a jedná se datovou strukturu. Adresa, která od sebe odlišuje jednotlivé instance, je dána speciálním klíčovým slovem (např. self) [22]. Jazyk Python byl vybrán z důvodu jeho jednoduchosti a přehlednosti, neboť ve srovnání s jinými jazyky je jeho kód lépe čitelný.
Pro uživatelské rozhraní byla použitá multiplatformní sada modulů jazyka Python a to *Pygame* [27]. Ta obsahuje knihovny pro práci s grafikou, zvukem a vstupními zařízeními.
Jazyk Python nabízí knihovnu *NetworkX* [26], prostřednictvím které jsou v implementaci vytvářeny grafy. Tato knihovna například nabízí:

- datovou strukturu pro grafy, podgrafy a multigrafy
- balíček standardních grafových algoritmů
- uzly mohou představovat cokoliv, může se jednat o text, obrázek, atd.

- hrany mohou obsahovat vlastnosti typu ceny přechodu z jednoho uzlu do dalšího

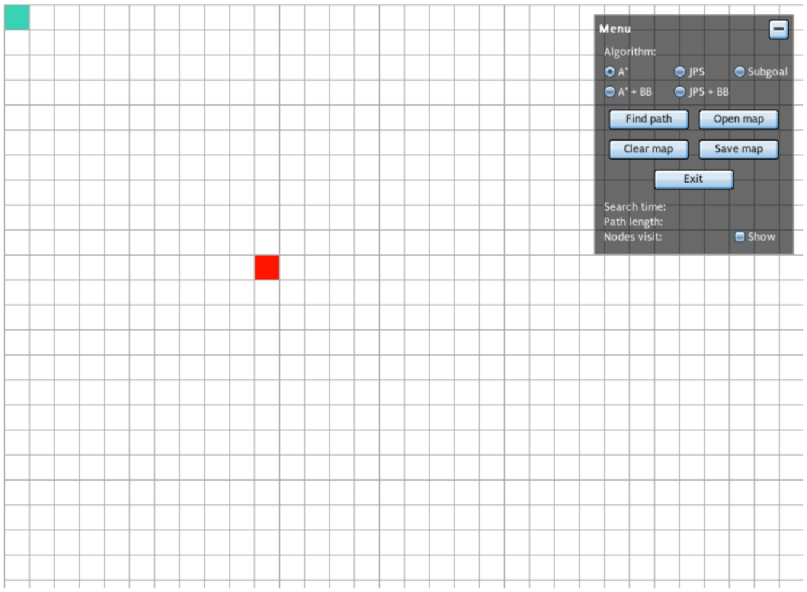
V této aplikaci uzly grafu představují pozice mobilního robota a hrany pak přechody mezi jednotlivými uzly grafu. Příklad takového grafu vytvořeného pomocí knihovny NetworkX je zobrazen na obrázku 3.1.



Obrázek 3.1: Graf vytvořený pomocí knihovny NetworkX

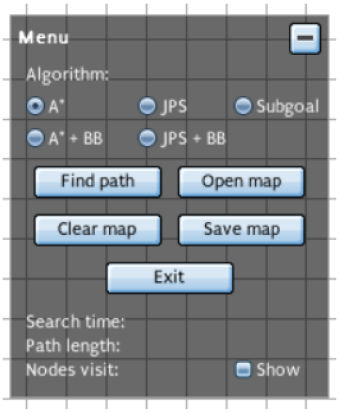
3.2 Návrh uživatelského rozhraní

Největší část uživatelského rozhraní tvoří mapa simulačního prostředí, která je pevně daná počtem uzlů odpovídajícího grafu. Další část uživatelského prostředí tvoří hlavní menu, které je tvořeno nabídkou akcí, které si volí uživatel.



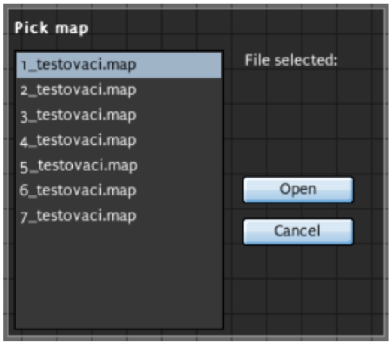
Obrázek 3.2: Uživatelské rozhraní aplikace

Na obrázku 3.2 je zobrazeno popisované uživatelské rozhraní, které je uživateli k dispozici po spuštění aplikace. Hlavní menu, viz obrázek 3.3, nabízí uživateli několik akcí.

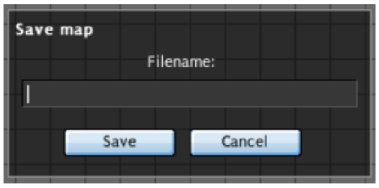


Obrázek 3.3: Hlavní menu aplikace

V prvé řadě je to výběr algoritmu, který má být použit pro nalezení nejkratší cesty. Dále nabízí načítání vytvořené mapy („Open map“), uložit novou mapu („Save map“) a odstranění aktuální mapy (překážek včetně případně nalezené nejkratší cesty ze startovací polohy do cílové polohy - „Clear map“), viz obrázek 3.4 a 3.5.

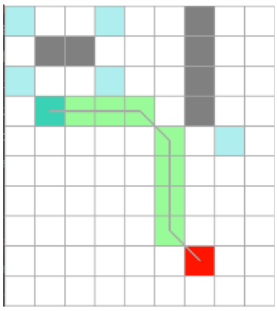


Obrázek 3.4: Načtení mapy ze souboru

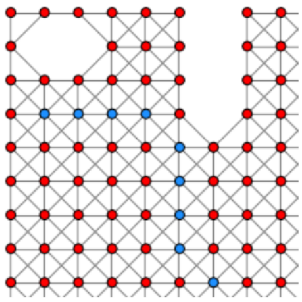


Obrázek 3.5: Uložení mapy do souboru

Pokud chce uživatel vytvořit mapu vlastní, lze tak učinit pomocí tlačítek na ovládací myši. Při spuštění aplikace je start a cíl nastaven na výchozích pozicích. Pro změnu pozice startu uživatel klikne na buňku startu (tyrkysová buňka) levým tlačítkem a umístí jej na novou pozici opět stiskem pravého tlačítka myši. Pozici cíle (červená buňka) lze měnit pravým tlačítkem myši. Překážky volí uživatel stiskem kolečka myši a nebo jejím stiskem a táhnutím. Odstranění překážek se provádí čistě stisknutím kolečka myši. Po vytvoření či nahrání mapy ze souboru uživatel klikne na tlačítko „Find path“ a aplikace zobrazí nalezenou nejkratší cestu, viz obrázek 3.6. Odpovídající nalezená cesta pomocí grafu v NetworkX je zobrazená na obrázku 3.7.



Obrázek 3.6: Nejkratší cesta - Pygame



Obrázek 3.7: Nejkratší cesta - NetworkX

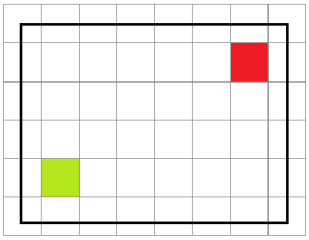
V rámci menu je zobrazena doba hledání nejkratší cesty, její délka a počet expandovaných uzlů, což jsou tři sledované hodnoty v rámci provedených experimentů. Tlačítko „Show“ zobrazuje expandované uzly. V rámci pracovní oblasti simulačního prostředí se pracuje s několika barev. Tyrkysová představuje pozici startu, červená pozici cíle, šedá reprezentuje překážky, zelená představuje nejkratší nalezenou cestu a světle modrá expandované uzly.

4 Výsledky experimentů

V této kapitole budou představeny provedené experimenty. Nejdříve byly provedeny srovnávací testy algoritmu A^* s jeho modifikací $A^* + BB$, kdy byl použit bounding box na zmenšení prohledávacího prostoru. Dále pak srovnání JPS s aplikací stejného bounding boxu a na závěr pak v experimentu 3 srovnání všech algoritmů. Cena přechodu z jednoho uzlu do druhého ve vertikálním a horizontálním směru byla nastavena na 1 a cena přechodu z uzlu do uzlu v diagonálním směru byla nastavena na $\sqrt{2}$. Jako metrika byla zvolena u všech algoritmů metrika octile. Výsledné sledované hodnoty byly počet expandovaných uzlů, délka nejkratší cesty a čas. Výsledné hodnoty času jsou průměrné hodnoty z dvaceti provedených vyhledávání.

4.1 Experiment 1

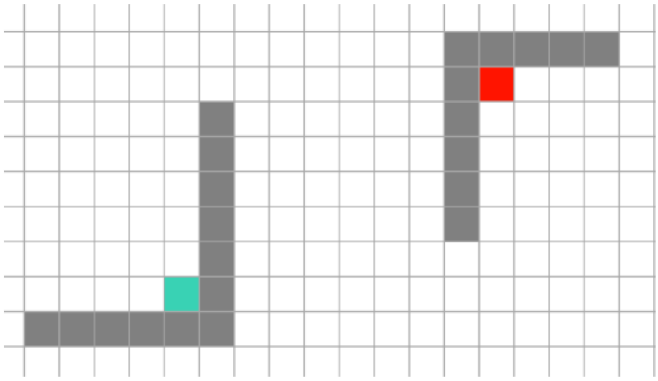
V tomto experimentu bylo provedeno porovnání algoritmu A^* a algoritmu A^* s použitím bounding boxu ($A^* + BB$), o kterém byla zmínka v podkapitole 2.2.5. Implementace byla provedena s vlastním návrhem bounding boxu, jehož konstrukce je znázorněna na obrázku 4.1.



Obrázek 4.1: Bounding box

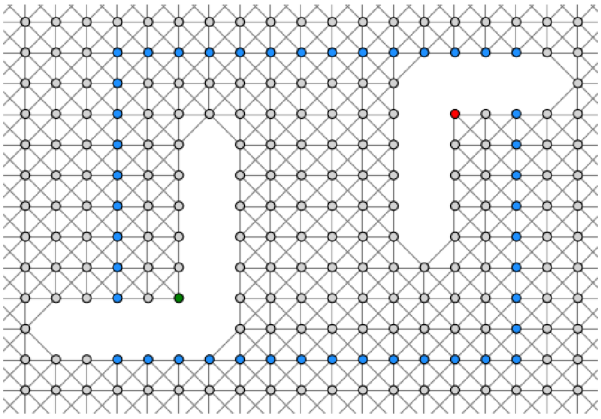
Princip je založen na jednoduché myšlence, kdy je vypočten ohraničující rámeček kolem startovacího a cílového uzlu. Tyto uzly jsou následně odstraněny a jsou tedy brány jako překážky. Algoritmus následně hledá nejkratší cestu v rámci tohoto rámečku a pokud cestu vně rámečku nenalezne, tak jej rozšíří. Počáteční velikost bounding boxu, stejně tak jako počet uzlů, o který je bounding box rozšířen, by měl být volen vhodně s ohledem na to, že například rozšiřování bounding boxu do stran o jeden uzel algoritmus zpomaluje.

Experiment byl proveden na dvou mapách. První z nich byla volena s nízkou náročností, viz obrázek 4.2.



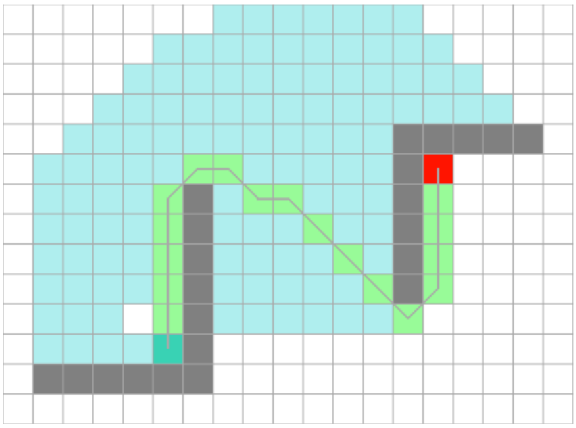
Obrázek 4.2: První mapa

Velikost bounding boxu byla zvolena o dva uzly větší jak velikost bounding boxu, který by obsahoval startovní a cílový uzel a následné rozšiřování bylo zvoleno o čtyři uzly. Počáteční bounding box je pro názornost zobrazen na obrázku 4.3, a to pomocí reprezentace odpovídajícího grafu v NetworkX. V něm zelený uzel odpovídá startovnímu

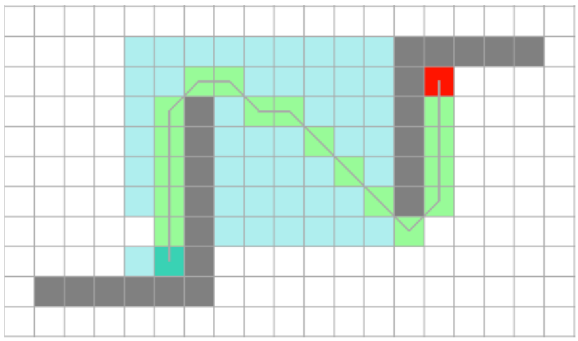


Obrázek 4.3: Bounding box

uzlu, červený cílovému uzlu a modré uzly pak znázorňují bounding box. Nalezené nejkratší cesty jsou na obrázcích 4.4 a 4.5, přičemž výsledky jsou uvedeny v tabulce 1.



Obrázek 4.4: A^* mapa 1



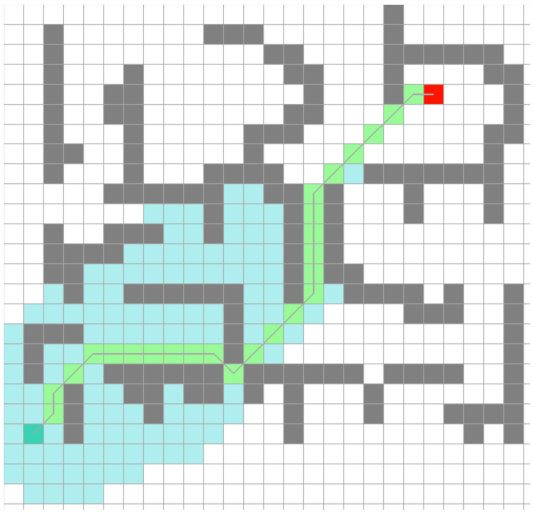
Obrázek 4.5: $A^* + BB$ mapa 1

Mapa 1			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
A*	20,9	161	18,83
A*+BB	20,9	79	9,46

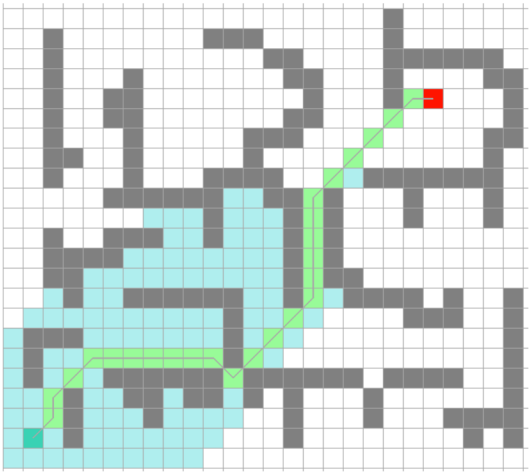
Tabulka 1: Vyhodnocení prvního experimentu na mapě 1

Z výsledků uvedených v tabulce 1 plyne, že algoritmus $A^* + BB$ má mnohem lepší výsledky. Použití bounding boxu na této mapě zredukovalo počet expandovaných uzlů o polovinu a časová náročnost se snížila o $9,37$ ms. Ačkoliv konstrukce bounding boxu není nikterak náročná, na tak jednoduché mapě dokáže algoritmus A^* vylepšit. Což však nemusí platit vždy.

Druhá mapa byla volena složitější. Nalezené nejkratší cesty algoritmů v simulačním prostředí jsou zobrazeny na obrázcích 4.6 a 4.7. Výsledky jsou uvedené v tabulce 2. Z výsledků plyne, že i zde je použitý bounding box vylepšením klasického A^* algoritmu.



Obrázek 4.6: A^* mapa 2



Obrázek 4.7: $A^* + BB$ mapa 2

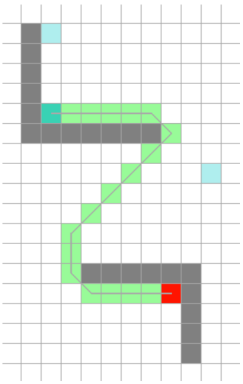
Mapa 2			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
A*	31,38	159	17,60
A*+BB	31,38	147	15,37

Tabulka 2: Vyhodnocení prvního experimentu na mapě 2

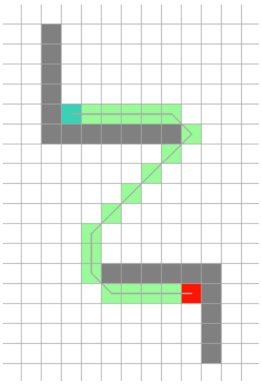
Nicméně jeho vylepšovací schopnost bude vždy závislá na použité mapě a vzdálenosti startovacího uzlu od cílového. Pokud startovací uzel s cílovým uzlem bude téměř v jedné rovině a na cestě mezi nimi budou překážky zasahující vysoko nad i pod jejich úroveň, bounding box bude muset být rozšířen několikrát a tím bude zhoršena jeho časová náročnost. Jeho konstrukci by bylo vhodné modifikovat.

4.2 Experiment 2

V druhém experimentu bylo provedeno srovnání algoritmu JPS s JPS+BB. Použitý bounding box byl stejný, se stejnými vlastnostmi, jako u experimentu 1. Jako první mapa byla vola stejně jednoduchá mapa, jako u předchozího experimentu. Na obrázku 4.8 a 4.9 jsou zobrazeny nalezené nejkratší cesty. v tabulce 3 obsahuje získané výsledky.



Obrázek 4.8: JPS mapa 1

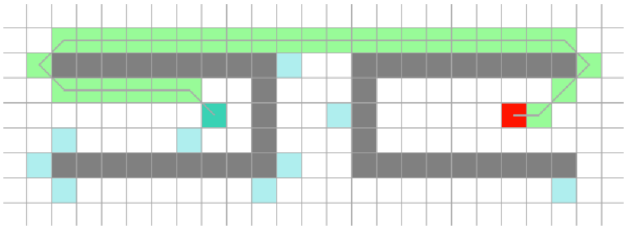


Obrázek 4.9: $JPS + BB$ mapa 1

Mapa 1			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
JPS	20,9	8	15,84
JPS+BB	20,9	6	8,38

Tabulka 3: Výsledky druhého experimentu na mapě 1

Z výsledků experimentu plyne, že stejně tak jako u algoritmu A^*, i zde bounding box pomáhá získat lepší výsledky. Počet expandovaných uzlů klesl o dva a čas se zrychlil o 7,46 ms. Nicméně přejděme k druhé mapě. Ta byla ve své podstatě volena také jednoduchá, a to s menší změnou, která krásně poukázala na nevýhodu navržené konstrukce bounding boxu. Na obrázku 4.10 je zobrazena nejkratší nalezená cesta jak pomocí algoritmu JPS tak i pomocí JPS+BB (vizuálně je shodná).

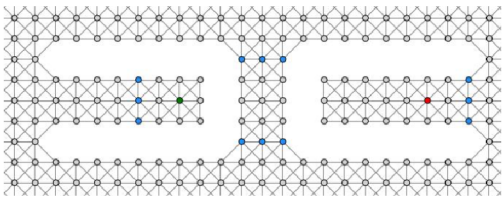


Obrázek 4.10: Nejkratší nalezená cesta pomocí JPS i JPS+BB

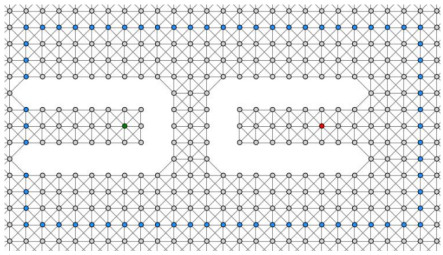
Mapa 2			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
JPS	34,49	19	16,68
JPS+BB	34,49	25	28,02

Tabulka 4: Výsledky druhého experimentu na mapě 2

Z výsledků uvedených v tabulce 4 plyne, že použitý bounding box prohledávací schopnosti algoritmu JPS zhoršil. Je to z toho důvodu, že hned v první iteraci se algoritmus zasekl v překážce, jak lze vidět na obrázku 4.11. V druhé itaraci se již bounding box rozšířil nad úroveň překážek, viz obrázek 4.12. Krok rozšíření byl zvolen o 4 uzly.



Obrázek 4.11: První iterace JPS+BB zobrazena v grafu v NetworkX



Obrázek 4.12: Druhá iterace JPS+BB zobrazena v grafu v NetworkX

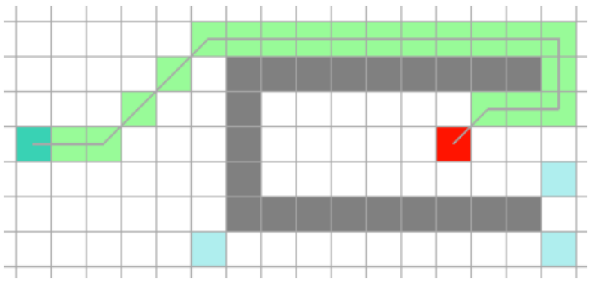
4.3 Experiment 3

V tomto experimentu byly porovnány všechny algoritmy dohromady. Bylo použito pět map z práce [6], které byly překreslené do simulačního prostředí.

Mapa 1			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
A*	20,49	224	25,92
A*+BB	20,49	195	21,73
JPS	20,49	7	18,37
JPS+BB	20,49	8	15,87
SS	21,66	6	4,51

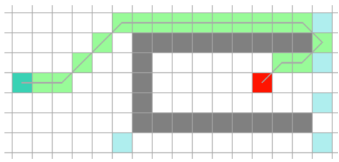
Tabulka 5: Výsledky třetího experimentu na mapě 1

První z testovaných map byla volena jednoduchá, pouze s jednou překážkou. Ze získaných výsledků, které jsou ukázány v tabulce 5, vyplývá, že algoritmus SS má jednoznačně nejmenší čas a počet expandovaných uzlů, nicméně nenalezl nejkratší cestu, viz obrázek 4.13. Algoritmus JPS+BB nalezl nejkratší cestu za nejrychlejší čas, ale expanduje

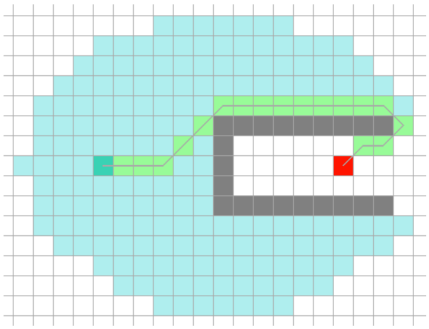


Obrázek 4.13: Mapa 1 - algoritmus SS

o jeden uzel navíc oproti algoritmu JPS. Na obrázku 4.14 je zobrazena nejkratší cesta pomocí algoritmu JPS a pro porovnání je na obrázku 4.15 zobrazena nejkratší nalezená cesta pomocí algoritmu A^*, který dopadl nejhůře. Lze zde vidět extrémní rozdíl v počtu expandovaných uzlů, které představují modré buňky.



Obrázek 4.14: Mapa 1 - JPS

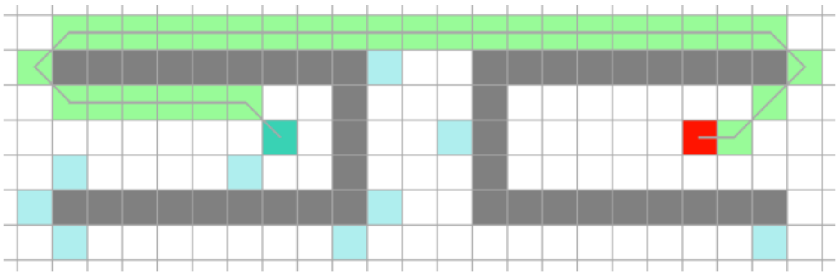


Obrázek 4.15: Mapa 1 - algoritmus A^*

Druhá testovaná mapa je podobná jako první, pouze je v ní zrcadlově přidána stejná překážka. Zde dle získaných výsledků (tabulka 6) dopadl nejlépe algoritmus JPS, jehož počet expandovaných uzlů i čas byl nejmenší vzhledem k nalezení nejkratší cesty. Nalezená nejkratší cesta je zobrazena na obrázku 4.16. Ačkoliv jsou výsledky algoritmu SS o mnoho lepší, co se počtu expandovaných uzlů a času týče, algoritmus opět nenalezl nejkratší cestu.

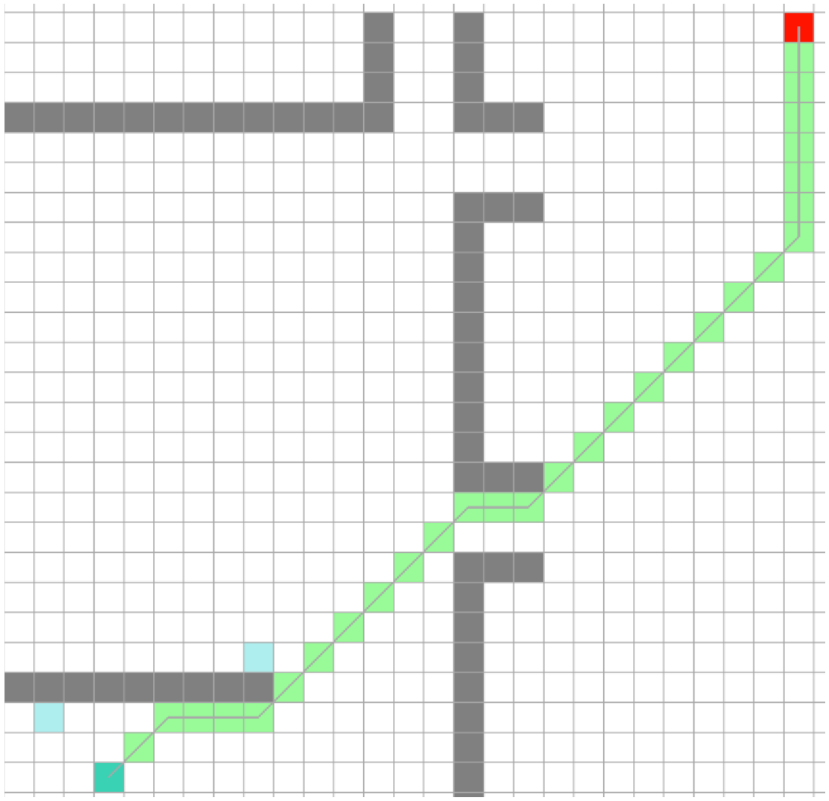
Mapa 2			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
A*	34,49	336	28,63
A*+BB	34,49	356	35,14
JPS	34,49	19	16,68
JPS+BB	34,49	25	28,02
SS	36,83	12	2,52

Tabulka 6: Výsledky třetího experimentu na mapě 2



Obrázek 4.16: Nejkratší nalezená cesta pomocí algoritmu JPS na mapě 2

Situace, kdy algoritmus SS měl jednoznačně nejkratší čas nalezení cesty se opakovala i u mapy 3 a 4, ale nenalezl nejkratší cestu. Nicméně nejkratší cesta na mapě 3 byla nejrychleji nalezená pomocí algoritmu JPS, viz obrázek 4.17 a tabulka 7, přičemž algoritmus JPS+BB má stejný počet expandovaných uzlů a časovou náročnost lehce přes 1 ms větší.

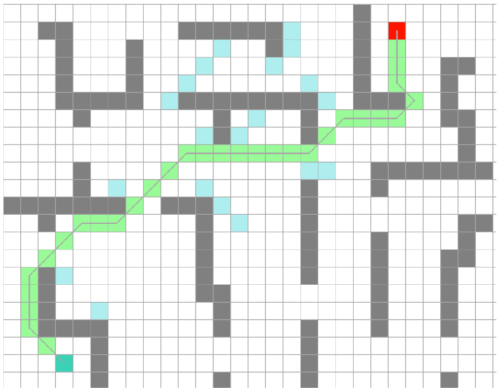


Obrázek 4.17: Nejkratší nalezená cesta pomocí algoritmu JPS na mapě 3

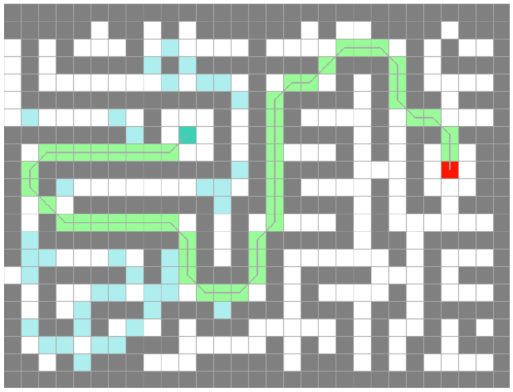
Mapa 3			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
A*	37,46	137	20,06
A*+BB	37,46	134	14,53
JPS	37,46	11	13,22
JPS+BB	37,46	11	14,57
SS	38,63	19	4,17

Tabulka 7: Výsledky třetího experimentu na mapě 3

Výsledky z hledání nejkratší cesty na mapě 4 a 5 ukazují ve prospěch algoritmu JPS (tabulka 8 a tabulka 9) a nalezení těchto nejkratších cest je zobrazeno na obrázku 4.18 a 4.19. Obecně z výsledku tohoto experimentu lze říci, že je nejvhodnější algoritmus JPS, neboť jeho výsledky jsou nejlepší jak po časové, tak i po množství expandovaných uzlů.



Obrázek 4.18: Mapa 4 (JPS)



Obrázek 4.19: Mapa 5 (JPS)

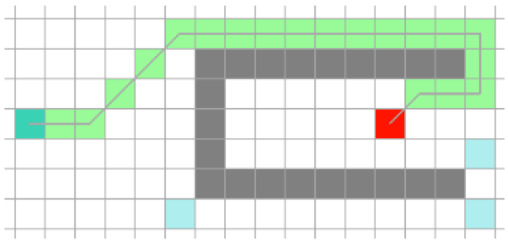
Mapa 4			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
A*	36,38	264	25,67
A*+BB	36,38	273	23,80
JPS	36,38	39	12,95
JPS+BB	36,38	42	18,70
SS	38,14	74	7,36

Tabulka 8: Výsledky třetího experimentu na mapě 4

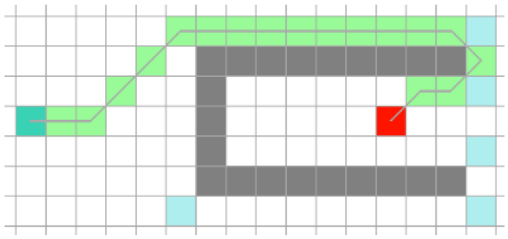
Mapa 5			
Algoritmus	Délka cesty	Expandované uzly	Čas [ms]
A*	54,8	173	13,29
A*+BB	54,8	181	18,16
JPS	54,8	67	11,68
JPS+BB	54,8	68	20,18
SS	62,41	80	12,71

Tabulka 9: Výsledky třetího experimentu na mapě 5

Z výsledků třetího experimentu vyplývá, že kromě mapy 5, časová náročnost algoritmus SS byla jednoznačně nejlepší, nicméně nikdy nenalezl nejkratší cestu. Tato skutečnost je dána tím, že algoritmus je omezen svou vlastností, že musí procházet skrze podcíle, které jsou umístěny v rozích překážek. Toto omezení by mohlo být vyřešeno optimalizací, která by tyto podcíle vynechala, pokud by existovala kratší cesta než skrze ně, viz obrázek 4.20 a 4.21.



Obrázek 4.20: Algoritmus SS bez optimalizace



Obrázek 4.21: Algoritmus SS s optimalizací

Závěr

Cílem této diplomové práce bylo provést analýzu přístupů k plánování cesty mobilního robotu, dále pak implementovat vybrané metody plánování cesty a provést ověřovací a srovnávací experimenty. První kapitola diplomové práce byla zaměřená na obecný popis problematiky plánování cesty mobilního robotu společně s obecným popisem pokročilých metod algoritmů umělé inteligence. V druhé kapitole byly popsány vybrané algoritmy pro hledání nejkratší cesty. Praktická část diplomové práce se zaměřila na implementaci algoritmů a vytvoření simulačního prostředí pro provedení srovnávacích experimentů.

Diplomová práce byla zaměřená na pokročilé metody a pro implementaci byly zvolené algoritmy, které zefektivňují proces hledání pomocí redukce počtu prozkoumaných uzlů stavového prostoru. Bylo implementováno několik algoritmů, a to algoritmus A^*, JPS, včetně jejich modifikace s bounding boxem a Jednoduchý subgoal algoritmus.

V rámci srovnání implementovaných algoritmů byly provedeny 3 experimenty. V těchto experimentech byla vyhodnocována délka nalezené cesty, počet expandovaných uzlů a čas potřebný pro nalezení nejkratší cesty. V rámci experimentů byl implementován vlastní návrh redukce prohledávacího prostoru a to v podobě bounding boxu. Ten byl v prvním experimentu aplikován na algoritmus A^* a z výsledků, kdy byly provedeny testy na dvou mapách lze říci, že algoritmus vylepšuje, a to z hlediska časové náročnosti i počtu expandovaných uzlů. V druhém experimentu byl tento bounding box aplikován na algoritmus JPS, kde ukázal taky jistou redukci, jak v rámci časové náročnosti, tak i v počtech expandovaných uzlů. Nicméně jeho prospěšnost je závislá na použité mapě. V budoucnu by tento návrh bounding boxu mohl být optimalizován z hlediska jeho konstrukce. Ve třetím experimentu byly srovnány všechny algoritmy dohromady, kdy nejlepší výsledky měl algoritmus JPS. Srovnávací testy v tomto experimentu byly provedeny na pěti mapách. Problém nastal s algoritmem SS (Jednoduchý subgoal algoritmus), který sice dle získaných výsledků má nejlepší časovou náročnost a u mapy 1 a 2 i nejmenší počet prozkoumaných uzlů, nicméně jeho implementace by se měla vylepšit. Algoritmus SS u všech map nenašel nejkratší cestu. Omezením tohoto algoritmu je, že prochází podcíli umístěných v rozích překážek, což by se dalo optimalizovat vynecháním těch podcílů, před kterými existuje diagonální směr pohybu vedoucí k nalezení nejkratší cesty. Srovnávací experimenty nicméně ukázali, že použití pokročilých metod vede na lepší výsledky vyhledání nejkratší cesty vzhledem k počtu expandovaných uzlů a časové náročnosti.

Literatura

[1] KOUBAA, Anis, et al. Introduction to Mobile Robot Path Planning. In: *Robot Path Planning and Cooperation*. Springer, Cham, 2018. p. 3-12.

[2] SKALKA, Marek. *Srovnání lokalizačních technik*. Praha, 2011. Diplomová práce. Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra softwarového inženýrství. Vedoucí práce Obdržálek, David.

[3] ABD ALGFOOR, Zeyad; SUNAR, Mohd Shahrizal; KOLIVAND, Hoshang. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015.

[4] ŠINDELÁŘ, J. *Plánování cesty neholonomního mobilního robotu*. Brno, 2010. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství. Vedoucí práce Ing. Petr Krček.

[5] SEDLÁK, V. *Plánování cesty mobilního robotu pomocí mravenčích algoritmů*. Brno, 2011. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství. Vedoucí práce RNDr. Jiří Dvořák, CSc.

[6] KLOBUŠNÍKOVÁ, Z. *Plánování cesty mobilního robotu*. Brno, Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2018, 62 s. Vedoucí diplomové práce RNDr. Jiří Dvořák, CSc.

[7] TSARDOULIAS, E. G., et al. A review of global path planning methods for occupancy grid maps regardless of obstacle density. *Journal of Intelligent and Robotic Systems,* 2016, 84. 1-4: 829-858.

[8] RABIN, Steve; STURTEVANT, Nathan R. Combining bounding boxes and jps to prune grid pathfinding. In: *Thirtieth AAAI Conference on Artificial Intelligence.* 2016.

[9] ZILBERSTEIN, Shlomo. Using anytime algorithms in intelligent systems. *AI magazine,* 1996, 17.3: 73-73.

[10] LIKHACHEV, Maxim; GORDON, Geoffrey J; THRUN, Sebastian. *ARA**: Anytime A^* with provable bounds on suboptimality. In: *Advances in neural information processing systems.* 2004. p. 767-774.

[11] LIKHACHEV, Maxim, et al. Anytime Dynamic A^*: An Anytime, Replanning Algorithm. In: *ICAPS.* 2005. p. 262-271.

[12] STENTZ, Anthony, et al. The focussed D^* algorithm for real-time replanning. In: *IJCAI.* 1995. p. 1652-1659.

[13] KOENIG, Sven; LIKHACHEV, Maxim. *D*lite*. *Aaai/iaai*, 2002, 15.

[14] HART, Peter E.; NILSSON, Nils J.; RAPHAEL, Bertram. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 1968, 4.2: 100-107.

[15] DOŠLÁ, Zuzana; DOŠLÝ, Ondřej. *Metrické prostory. Teorie a příklady.* Masarykova univerzita, 2016.

[16] A^* search algorithm. *Wikipedie* [online]. [cit. 2020-05-30]. Dostupné z: https://en.wikipedia.org/wiki/A*_search_algorithm

[17] RABIN, Steve; SILVA, Fernando. An Extreme A^* Speed Optimization for Static Uniform Cost Grids [J]. *Game AI Pro 2: Collected Wisdom of Game AI Professionals.* 2015, 131.

[18] HARABOR, Daniel Damir; GRASTIER Alban. Online graph pruning for pathfinding on grid maps. In: *Twenty-Fifth AAAI Conference on Artificial Intelligence.* 2011.

[19] HARABOR, Daniel Damir; GRASTIER, Alban. The JPS Pathfinding System. In: *SOCS.* 2012.

[20] URAS, Tansel; KOENIG, Sven; HERNÁNDEZ, Carlos. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Twenty-Third International Conference on Automated Planning and Scheduling.* 2013.

[21] *Wikipedia* [online]. [cit. 2020-06-15]. Dostupné z: https://cs.wikipedia.org/wiki/Python

[22] *Voho* [online]. [cit. 2020-06-15]. Dostupné z: http://voho.eu/wiki/oop/

[23] URAS, Tansel; KOENIG, Sven. Subgoal graphs for fast optimal pathfinding. *Game AI Pro*, 2015, 2: 145-159.

[24] ČÍŽEK, L. Navigace robotu pomocí grafových algoritmů. Brno, 2011. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství. Vedoucí práce RNDr. Jiří Dvořák, CSc.

[25] HAGBERG, Aric; SCHULT, Daniel; SWART, Pieter. *Exploring network structure, dynamics, and function using NetworkX.* Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[26] VAN ROSSUM, Guido; DRAKE, Fred L. *The python language reference manual.* Network Theory Ltd., 2011.

[27] SHINNERS, Pete. PyGame - Python Game Development. Dostupné z: http://www.pygame.org