FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Hledání disjunktních cest v nerovinném prostředí
Student:	Petr Michalíček
Vedoucí:	doc. RNDr. Pavel Surynek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Práce se bude zabývat řešením problému nalezení disjunktních cest, kdy máme propojit vzájemně se nekřížícími cestami množinu dvojic míst v jistém prostředí. Přitom budeme chtít zohledňovat určitá kritéria, jako je celková délka cest, nebo jejich vzájemná vzdálenost. Speciálně se práce bude zaměřovat na případ hledání disjunktních cest v trojrozměrném prostředí s překážkami, který má význam pro plánování pohybu podvodních robotů připojených kabelem. Jako vhodné se jeví použít prohledávací techniky nebo překlad problému do jiného formalismu.

Úkoly pro uchazeče budou následující:
1. Prostudujte otázku hledání disjunktních cest teoreticky podle dostupné literatury.
2. Na základě provedené rešerše navrhněte nové metody nebo modifikujte metody existující, které by byly vhodné pro případ trojrozměrného prostředí s překážkami.
3. Navržené metody implementujte formou softwarového prototypu a porovnejte co do výkonnosti a kvality produkovaných řešení.

[1] David Silver: Cooperative Pathfinding. AIIDE 2005: 117-122

[2] Ken-ichi Kawarabayashi, Yusuke Kobayashi, Bruce A. Reed: The disjoint paths problem in quadratic time. J. Comb. Theory, Ser. B 102(2): 424-435 (2012)

[3] Laxmi Gewali, Dan Mazzella, Henry Selvaraj: Constrained Disjoint Paths in Geometric Networks. Int. J. Comput. Intell. Appl. 8(2): 141-154 (2009)

[4] Kai Fieger, Tomás Balyo, Christian Schulz, Dominik Schreiber: Finding Optimal Longest Paths by Dynamic Programming in Parallel. SOCS 2019: 61-69

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

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Hledání disjunktních cest v nerovinném prostředí

Petr Michalíček

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

13. května 2021

Poděkování

Chtěl bych poděkovat doc. RNDr. Pavlu Surynkovi, Ph.D. za skvělé vykonávání role vedoucího této bakalářské práce a za přínosné rady, díky kterým je tato práce nepochybně lepší.

Prohlášení

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

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

V Praze dne 13. května 2021 . . . . . .

Odkaz na tuto práci

Michalíček, Petr. *Hledání disjunktních cest v nerovinném prostředí.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Kooperativní hledání disjunktních cest je varianta problému multiagentního hledání cest, při kterém je zadán prostor a množina obsahující několik párů se startovní a cílovou pozicí z prostoru. Úkolem je nalézt vrcholově-disjunktní cesty ze startovní do cílové pozice pro každý z těchto párů. Při kooperativním hledání se agenti snaží spolupracovat, tedy každý má informace o pozicích ostatních a jejich společný cíl je, aby každý agent dosáhl cíle. V této práci je přestaveno několik nových algoritmů, vhodných pro řešení tohoto problému.

Populární a používané algoritmy CA*, HCA* a WHCA* byly upraveny tak, aby fungovaly pro hledání disjunktních cest na voxelové mřížce. Všechny 3 byly také zprovozněny na datové struktuře oktantový strom, a to přineslo v některých prostorech výrazné zrychlení. Dále byl vytvořen algoritmus Obstacle CA*, který dokáže sestavovat úspěšněji cesty v oblastech, kde se vyskytují úzké průchody. Nové metody MDCA* a MDWHCA* hledají cesty tak, aby mezi nimi byly co největší mezery a tím předcházelo kolizím.

Klíčová slova disjunktní cesty, kooperativní hledání cest, multiagentí hledání cest, voxelová mřížka, oktantový strom, A*, trojrozměrný prostor.

Abstract

Cooperative disjoint pathfinding is a variant of the multi agent pathfinfing problem. For a given space and a set of pairs of start and goal positions in space the objective is to find vertex-disjoint paths, which connecting the start and goal position from each given pair. In cooperative pathfinding, agents try to cooperate with each other. Every agent knows the positions of others. They all have a common objective, which is successfully complete paths for all agents. In this work new algorithms for disjoint pathfinding are presented.

Popular algorithms CA*, HCA* and WHCA* were modified to solving cooperative disjoint pathfinding problem with a voxel grid. They were also implemented in version, which works with octree. Using octree brings significant speed improvement in some spaces. The created algorithm Obstacle A* can find paths more successfull than other methods in regions with botllenecks. New methods MDCA* and MDWHCA* can be used for creating paths, which has bigger spaces among themselves.

Keywords disjoint paths, cooperative pathfinding, multi agent pathfinding, voxel grid, octree, A*, 3D space.

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

*

Úvod

Multiagentní hledání cest (MAPF) je problém, kde máme zadáno několik agentů různě rozmístěných po prostoru a úkolem je najít pro každého cestu do jeho cíle, aniž by došlo ke kolizím s ostatními agenty. Algoritmy, které toto řeší, se využívají v oborech, jako je robotika, plánování nebo vývoj počítačových her. Posledních 20 let je toto téma velmi populární, a tak je již velmi dobře prozkoumané, ale objevují se speciální varianty problému, které klasické multiagentní hledání cest neumí vyřešit.

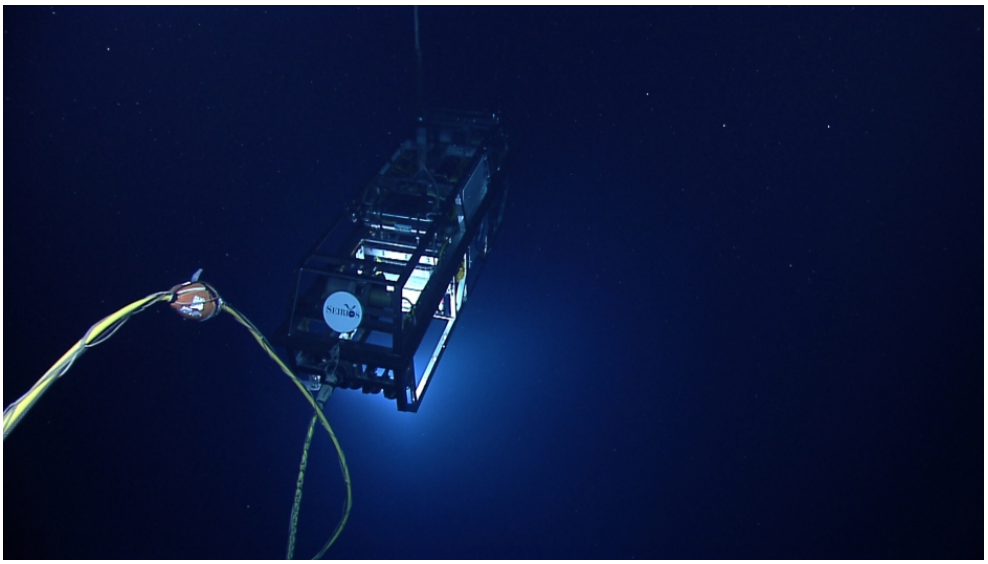
Jednou z těchto variant se zabývá i tato práce. Jedná se o případ, kdy jsou agenti ke svému startovnímu bodu připojeny kabelem. To znamená, že na pozici, přes kterou již nějaký agent prošel, nesmí žádný jiný vstoupit, protože by se do sebe kabely zamotaly. Tato varianta má význam pro plánování pohybu robotů v prostoru, kde se špatně šíří signál, a tak musí být připojeni kabelem. Příkladem takového prostředí je voda. Podvodních robotů existuje celá řada: miniponorky, vodní drony nebo roboti provádějící podvodní stavební práce.

Od ostatních prací na podobné téma se tato odlišuje tím, že se specializuje na trojrozměrné prostředí s překážkami, které připomíná podmořský prostor. Prostor bude modelován pomocí trojrozměrné voxelové mřížky. V ní je možné přiřadit každé pozici hodnotu, a tak rozlišit volnou pozici od překážky nebo od cesty.

Kromě samotného hledání cest se bude práce zabývat optimalizováním některých důležitých kritérií. Prvním z nich je celková délka cest, která by měla být co nejkratší, aby se roboti co nejrychleji dostali do cíle, a aby spotřebovali co nejmenší množství kabelů. Trochu netradiční je druhé kritérium vzájemná vzdálenost cest od sebe. Ta by naopak měla být co nejvyšší, aby kabely byly co nejdál od sebe. Tím se snižuje pravděpodobnost vzniku kolizí.

Text je rozdělen do 3 částí. První je teoretická a nazývá se teoretická východiska. Zde se nachází úvod do teorie grafů, seznámení s MAPF, poté jsou zde zmíněny ostatní práce blízké tomuto tématu, a nakonec jsou rozebrány současné algoritmy řešící tuto problematiku.

Další 2 části představují praktickou složku práce. V sekci vlastní řešení jsou představeny zcela nové nebo modifikované algoritmy vhodné pro hledání disjunktních cest v nerovinném prostředí s překážkami. Také je zde věnován prostor pro optimalizaci zkoumaných kritérií. V části experimenty jsou tyto metody a algoritmy mezi sebou porovnány a je vyhodnocena kvalita jimi produkovaných řešení.



Kapitola 1

Cíle práce

V teoretické části je nejprve potřeba zavést důležité pojmy z oblastí teorie grafů a hledání cest. Další cíl je popsat prostor, potom přesně definovat problém, kterým se tato práce zabývá a zavést značení, jaké bude používáno v textu. Poté je důležité prostudovat metody a algoritmy zabývající se hledáním disjunktních cest a vybrat takové, které budou užitečné při tvorbě vlastního řešení.

Hlavní cíl praktické části je navrhnout nové nebo modifikovat stávající metody, které budou vhodné pro hledání disjunktních cest v trojrozměrném prostředí s překážkami. Tyto nové algoritmy nebo jejich upravené verze by měly také optimalizovat tato kritéria: celková délka cest a vzájemná vzdálenost cest od sebe.

V poslední části je cílem ověřit experimentálně vlastnosti navržených algoritmů. Nové metody budou srovnány mezi sebou jak z hlediska výkonnosti, tak z hlediska kvality produkovaných řešení.

Kapitola 2

Teoretická východiska

2.1 Definice problému

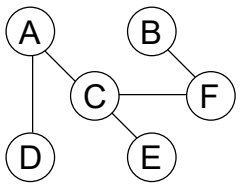
Úkolem této sekce je přesně popsat problém, kterým se tato práce zabývá a definovat související pojmy. Dále zde bude zavedeno značení, které se bude používat ve zbytku textu. Nejprve se text zabývá definicí grafů a souvisejících pojmů, poté bude popisován prostor, kde bude hledání cest probíhat. Dále bude vysvětlen samotný problém disjunktních cest a na konci budou popsány další kritéria, které při hledání budou zohledňovány, konkrétně: celková délka cest a vzájemná vzdálenost cest.

2.1.1 Graf a cesta v grafu

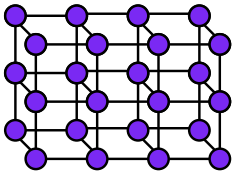
Hledání cest bylo původně definováno v grafovém prostředí. Existují 2 typy grafů: orientované a neorientované, které se liší definicí množiny hran. V této práci se pracuje pouze s neorientovanými grafy.

Definice 1. *Neorientovaný graf G je uspořádaná dvojice $G(E, V)$, kde:*

- *V je neprázdná množina, jejíž prvky nazýváme vrcholy grafu G.*
- *E je množina dvouprvkových podmnožin množiny V. Prvky množiny E se nazývají hrany grafu G [2].*



Obrázek 2.1: Neorientovaný graf



Obrázek 2.2: Graf mřížka(4,2,3)

Na obrázku 2.1 je příklad neorientovaného grafu, jehož zápis vypadá takto: $G(V = \{A, B, C, D, E, F\}, \quad E = \{\{A, C\}, \{A, D\}, \{B, F\}, \{C, E\}, \{C, F\}\})$.

Další důležitý pojem je *cesta*. Ten může mít 2 významy: Může tak být označován speciální typ grafu, a nebo to může označovat speciální podgraf.

Definice 2. *Graf P_n (kde $n \geq 0$) nazýváme cesta, pokud:*

$$V = \{1, 2, ..., n\}, \quad E = \{\{i - 1, i\}; i = 1, ..., n\}$$

Cesta v grafu G je pak posloupnost

$$(v_0, e_1, v_1, ..., e_t, v_T),$$

kde $v_0, v_1, ..., v_t$ jsou navzájem různé vrcholy grafu G, a pro každé $i = 1, 2, ..., t$ je $e_i = \{v_i - 1, v_i\} \in E(G)$ [2].

2.1.2 Mřížkový graf

Mřížka (nebo také *mřížkový graf*) je speciální typ grafu, který slouží jako základ používaného prostoru. Nejdříve je potřeba definovat operaci *kartézský součin grafů*, díky kterému pak bude definován pojem mřížkový graf:

Definice 3. *Kartézský součin 2 grafů H a K je graf $G = H \times K$, kde $V(G) = V(H) \times V(K)$. Vrcholy (h, k) a (h', v') z $V(G)$ spolu sousedí, pokud platí buď to:*

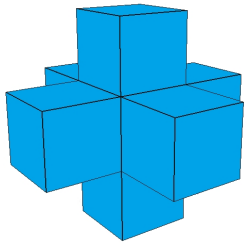
(1) *$k = k'$ a h, h' spolu sousedí v H, nebo*
(2) *$h = h'$ a k, k' spolu sousedí v K [3].*

Definice 4. *Graf mřížka (nebo mřížkový graf) $M_{x,y,...,z}$ je výsledek kartézského součinu $P_a \times P_b \times ... \times P_c$, kde $P_a, P_b, ..., P_c$ jsou cesty s $x, y, ..., z$ vrcholy.*

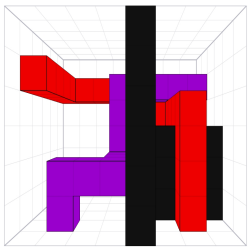
Příklad můžete vidět na obrázku 2.2, kde se jedná o mřížku s parametry 4, 3, 2. Mřížkový graf $M(x, y)$ s 2 parametry vypadá jako čtverec či obdélník a parametr x značí délku a y pak šířku. $M(x, y, z)$ má formu jako krychle nebo kvádr a parametry x, y, z zde představují délku, šířku a výšku. A jelikož se tato práce zabývá hledáním cest v nerovinném prostředí, tak budu využívat mřížky s 3 parametry jako základ prostoru, ve kterém budou algoritmy testovány.

Vrcholům v mřížce můžeme přidělit *souřadnice*, díky kterým je možná jejich jednoznačná identifikace.

Definice 5. *Nechť $M_{m,n,...,o}$ je graf mřížka, který vznikl kartézským součinem cest $P_a, P_b, ..., P_c$. Souřadnice $x, y, ..., z \in \mathbb{N}$ označují vrchol N, $N \in V(G)$, který vznikl kartézským součinem x. vrcholu z P_a, y. vrcholu z P_b, ..., z. vrcholu z P_c.*



Obrázek 2.3: Spojitost voxelů



Obrázek 2.4: Voxelová mřížka

2.1.3 Voxelová mřížka

Jako prostor pro hledání cest, byla v této práci zvolena *voxelová mřížka*. *Voxel* je prvek objemových dat reprezentující hodnotu. Pokud tyto prvky budeme ukládat do třírozměrné mřížky, vznikne tzv. „voxelová mřížka“. Jednotlivé voxely neznají svojí polohu, ale přistupuje se k nim pomocí souřadnic, které budou označovány jako (x, y, z) [4]. Zjednodušeně řečeno je voxel „to samé“ jako pixel, akorát v trojrozměrném prostoru.

Voxelovou mřížku lze také vytvořit z mřížkového grafu s 3 parametry, oba tyto objekty jsou si totiž velice podobné. To se provede tak, že každý vrchol z grafu $M(x, y, z)$ se převede na voxel se stejnými souřadnicemi, jaký měl daný vrchol a vložíme do něj hodnotu 0. Tím vznikne objekt, který má stejné vlastnosti jako graf $M(x, y, z)$. Každý voxel může sousedit s 3-6 vrcholy (viz obrázek 2.3) a jsou to takové vrcholy, se kterými by měl v mřížkovém grafu společnou hranu.

Pro získání všech *sousedů* voxelu budu používat funkci $neighbors(V, N)$, která bude nyní definována. Funkce přijímá na vstupu 2 argumenty: první z nich je voxelová mřížka V a druhým jsou souřadnice voxelu N. Funkce $neighbors(V, N)$ vrací souřadnice všech sousedů voxelu N ve voxelové mřížce V. Tedy s vrcholem se souřadnicemi x, y, z sousedí vrcholy:

$$[x+1, y, z], [x-1, y, z], [x, y+1, z], [x, y-1, z], [x, y, z+1], [x, y, z-1]$$

. Sousedících vrcholů může být méně. Pokud by nějaký ze z nich byl mimo mřížku, je z množiny odstraněn. Tento typ spojitosti se také nazývá „*face*“ a sousedé pak „*face neighbors*“ [5].

Ve voxelové mřížce tedy můžeme provádět většinu grafových operací a můžeme zde také hledat cesty. Ty zde vypadají trochu jinak než cesty v grafu, nejsou to posloupnosti vrcholů a hran, ale jsou to posloupnosti souřadnic voxelů, přes které cesty vedou.

Definice 6. *Cesta ve voxelové mřížce V s rozměry x,y,z je posloupnost:*

$$([x_1, y_1, z_1], [x_2, y_2, z_2], ..., [x_n, y_n, z_n]),$$

kde $(0 \leq x_i < x)$, $(0 \leq y_i < y)$ a $(0 \leq z_i < z)$ pro $i \in \{1, 2, ..., n\}$ a navíc platí, že $[x_i, y_i, z_i] \in neighbors(V, [x_{i+1}, y_{i+1}, z_{i+1}])$ pro $i \in \{1, 2, ..., n-1\}$.

Oproti vrcholům grafu jsou ve voxelech uchovávány hodnoty, čímž můžeme vyjádřit informace o prostoru. V této práci budou u voxelů rozlišovány tyto 3 druhy hodnot:

- Hodnota 0 znamená, že daný voxel je prázdný a tedy přes něj může vést cesta. Na obrázcích budou mít vždy bílou barvu.
- Hodnota 255 znamená, že na dané pozici je překážka a tím pádem tudy cesta vést nemůže. Na obrázcích budou vyznačeny černě.
- Hodnota v rozmezí $\{1, 2, ..., 254\}$ znamená, že tudy již nějaká cesta vede. Voxely na stejné cestě mají také stejnou hodnotu.

Příklad voxelové mřížky je na obrázku 2.4. Její rozměry jsou $(8, 6, 6)$ a jsou v ní vyznačené 2 cesty. Fialová cesta vede mezi voxely $[1, 1, 0]$ a $[7, 4, 4]$, zatímco červená mezi $[6, 1, 0]$ a $[0, 1, 4]$. V mřížce se vyskytuje několik překážek, které jsou vyznačené černě a zbylé vrcholy jsou volné.

2.1.4 Oktantový strom

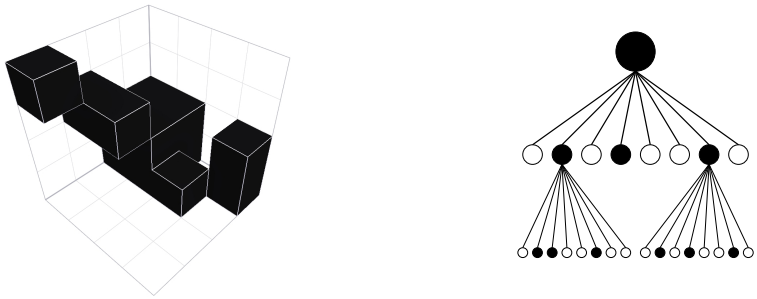
Kromě voxelové mřížky se bude využívat jako prostor pro hledání cest i *oktantový strom* (octree). Zjednodušeně je to trojrozměrná varianta kvadrantového stromu (quadtree), kde v kvadrantovém stromu má každý uzel 4 potomky a v oktantovém stromu 8 potomků. Informace o oktantovém stromu byly převzaty z [7].

Účel oktantového stromu je rekurzivní rozdělení 3D prostoru. Každý uzel má v sobě hodnotu, která popisuje region. V našem případě budeme rozlišovat 2 hodnoty - volná místa a překážky. V kořeni je uložen uzel, který je velký jako celý prostor. Ten se rekurzivně dělí na 8 potomků až do té doby, kdy každý uzel znázorňuje jeden voxel. Ty najdeme na poslední hladině stromu. Všechny *listy* nemusí být na poslední hladině, protože pokud hodnota uzlu kompletně popisuje daný region, už se dál rekurzivně nedělí a stává se listem.

Nyní je potřeba definovat některé veličiny. Už při konstrukci oktantového stromu je potřeba znát *maximální hloubku*, dále v textu bude značena h. Tato hodnota určuje maximální počet hladin, které může strom mít nebo se dá vyjádřit, jako maximální počet rekurzivního dělení kořene stromu. Další důležitá veličina je velikost. Ta je uložena v každém uzlu a označuje délku hrany krychle tvořící prostor, který daný uzel reprezentuje. Velikost oktantového stromu je velikost uzlu uloženého v kořeni. Tato veličina bude značena jako l.

Struktura má tyto výhody. Oktantový strom má vždy podobu krychle, což je jednoduchý objekt a nejsou proto potřeba techniky pro více složité a sofistikované tvary. Stejnou výhodu má i voxelová mřížka, která má tvar kvádru. Navíc z voxelové mřížky lze sestavit oktantový strom jednoduše - stačí do mřížky doplnit prázdná místa či překážky tak, aby tvořila krychli

Obrázek 2.5: Voxelová mřížka převedená na oktantový strom



o rozměrech $2^n \times 2^n \times 2^n$. Výhodou oktantového stromu oproti voxelové mřížce je jeho *hierarchická struktura*. Díky ní mohou algoritmy v některých případech pracovat pouze s uzly ve vyšších hladinách stromu a vyhnou se tak práci s velkým množstvím dat v nižších hladinách.

I v oktantovém stromu bude používán stejný typ spojitosti uzlů, jako byl definován u voxelové mřížky v sekci 2.1.3. Každý vrchol sousedí s 6 uzly, s tzv. „face neighbors“ [5]. Nutno poznamenat, že sousedící vrcholy v oktantovém stromě mohou mít odlišnou velikost než původní vrchol.

2.1.5 Hledání disjunktních cest

Ještě před popisem problému samotného je potřeba vysvětlit, co jsou to *disjunktní cesty*. Ty mohou být buďto vrcholově nebo hranově disjunktní. V této práci se bude pracovat s první možností, takže když budou zmíněny disjunktní cesty, myslí se tím vrcholově disjunktní cesty.

Definice 7. *Mějme 2 cesty* $P_a(a_1, a_2, ..., a_m)$ *a* $P_b(b_1, b_2, ..., b_n)$*. Cesty* P_a *a* P_b *nazveme jako disjunktní, pokud pro všechna* $i \in \{1, 2, ..., m\}$ *a pro všechna* $j \in \{1, 2, ..., n\}$ *platí:*

$$a_i \neq b_j$$

Definice problému disjunktních cest byla převzata z [6]. Jelikož se v této práci bude pracovat s nerovinným prostředím, tak nebude definován jako prostor graf, ale bude použita voxelová mřížka, která se více hodí pro popis trojrozměrných prostorů. Místo vrcholů grafu jsou použity voxely, které lze jednoznačně identifikovat pomocí souřadnic. Zadání je tedy následující:

Je nám dán prostor $S_{x,y,z}$ ve formě voxelové mřížky s rozměry x, y, z, množina párů voxelů $tasks((start_1, goal_1), (start_2, goal_2), ..., (start_n, goal_n))$ ve V a množina překážek O, $O \subset S$. Řešením problému je n disjunktních cest $P_1, P_2, ..., P_n$ ve V takových, že P_i spojuje voxely $start_i$ a $goal_i$, pro všechna $i \in \{1, 2, ..., n\}$ a navíc pro všechna o, $o \in O$ platí, že $o \notin P_1 \cup P_2 \cup ... \cup P_n$.

Další způsob, jak by se dalo na problém pohlížet, je *multiagentní hledání cest* (MAPF) [8, 9]. Při MAPF pracujeme s *agenty*, kde každý agent dostane

jeden pár $(start, goal)$ a snaží se dostat ze startovní pozice do cílové. Všichni agenti se pohybují zároveň a mají nadefinovanou množinu akcí, ze které mohou každý tah vybírat. V tomto případě mohou čekat na místě, nebo se posunout na volný sousední voxel.

Kooperativní MAPF je varianta, kde agenti spolupracují a snaží se, aby všichni dosáhli svého cíle. Aby to bylo možné, každý zná informace o ostatních agentech (polohu, cíl, ...). V tomto případě hledáme disjunktní cesty, což odpovídá problému MAPF s kabelem. V této variantě pokud agent někam vstoupí, zanechá tam kabel (překážku) a na toto pole již není možno nikdy vstoupit. Tedy k problému hledání disjunktních cest lze přistupovat, jako k kooperativní variantě MAPF s kabely.

Množina obsahující zadání agentů bude dále v textu označována jako A a $|A|$ značí velikost této množiny, tedy počet agentů. Mezi pojmy A a $tasks$ prakticky není žádný rozdíl, označení A se bude využívat, pokud bude na problematiku nahlíženo jako na speciální případ MAPF a $tasks$ v případě, kdy na problém bude nahlíženo jako na klasické hledání disjunktních cest.

2.1.6 Další zkoumaná kritéria

I přesto, že hlavní úkol je nalezení všech disjunktních cest, bude se také hledět na optimalizaci některých dalších kritérií. Jedno z nich je *celková délka cest*. Délka cesty P je počet vrcholů (v tomto případě voxelů), přes které cesta prochází. Celková délka cest je pak součet délek všech cest a optimálně by měla být co nejmenší. Pokud například plánujeme cestu pro robota s kabelem, čím kratší je cesta, tím dříve se dostane do cíle a také bude potřebovat menší množství kabelů. Nemělo by se však stávat, že kvůli tomu, aby nějaká cesta nebyla dlouhá, jiná cesta vůbec nevznikne. Dále v textu se bude metrika celková délka cest $P_1, P_2, ..., P_n$ označovat jako $L(P_1, P_2, ..., P_n)$.

Další zohledňované kritérium je *vzájemná vzdálenost cest od sebe*. Opět bude uveden příklad ze situace, kdy hledáme cesty pro roboty s kabelem. Platí že, čím dál od sebe jednotlivé kabely jsou, tím je nižší šance kolize. Proto je potřebné, aby vzájemná vzdálenost od sebe byla co největší. Tato veličinu bude značena takto: $MD(P_1, P_2, ..., P_n)$. Její definice zní:

Definice 8. *Nechť $P_1, P_2, ..., P_n$ jsou disjunktní cesty. Vzájemná vzdálenost cest $MD(P_1, P_2, ..., P_n)$ je rovna:*

$$\sum_{k=1}^{n} \sum_{l=1}^{n} \left(\frac{\sum_{v \in V(P_k)} min(distance(v, y)), \quad y \in P_l}{length(P_k)} \right)$$

2.2 Související práce

Zde jsou zmíněny práce, které se zabývají podobným tématem nebo takové, jejichž výsledky výrazně přispěly k tvorbě algoritmů, které jsou představeny v sekci Vlastní řešení.

V práci od Hart, Nillson a Raphael (1968) byl představen algoritmus *A**, jeho popis naleznete v sekci 2.3.1. Tento algoritmus slouží jako základ pro všechny výstupy této práce. David Silver (2005) použil A* k vytvoření metod pro kooperativní hledání cest. Ve své práci představil 3 nové algoritmy: *CA**, *HCA** a *WHCA**, jejich popis je k nalezení v sekci 2.3.2. V experimentech tyto metody byly úspěšnější v tvorbě řešení než tehdejší používané metody. V této práci jsou tyto algoritmy upraveny tak, aby fungovali pro hledání disjunktních cest a na použitých datových strukturách.

Jedna z takových struktur je oktantový strom. Ten byl poprvé představen v textu od Meagher, Donald (1980) a slouží pro lepší zpracování trojrozměrného prostoru. A jelikož se zde zabývám právě hledání disjunktních cest v 3D prostoru, bude oktantový strom použit při hledání časově efektivního řešení.

Podobným tématem se zabývá bakalářská práce od bc. Martina Zukala (2019), kterou byla také dělána pod vedením doc. Pavla Surynka. Podobně se zde řeší hledání disjunktních cest. Odlišné je, že zatímco Zukal testuje své algoritmy ve 2D prostoru, tato práce se přímo zaměřuje na 3D prostředí s překážkami. Algoritmy jako Obstacle CA* přímo s překážkami pracují nebo se zde používají speciální datové struktury vhodné pro trojrozměrný prostor jako oktantový strom. Také M. Zukal převádí problém na boolovský problém splnitelnosti, a v této práci se problém nepřevádí na jiný formalismus.

Hledáním disjukntních cest se také zabývá text od Kawarabayashi, Kobayashi a Reed (2012). Ti sestavili algoritmus, který problém vyřeší v čase $O(n^2)$. Ten však funguje pouze pro fixní k počet agentů, ale cílem této práce je navrhnout metody fungující pro libovolný počet agentů.

Tento problém také řešili Gewali, Selveraj a Mazzella (2009). V jejich práci bylo zadání definováno tak, že na vstupu byla dána jedna startovní a cílová pozice a úkolem bylo sestavit 2 hranově disjunktní cesty vedoucí ze stejné startovní pozice do stejné cílové pozice. Bohužel je zadání natolik odlišné od našeho, že metody z této práce nelze plně použít.

2.3 Algoritmy pro hledání cest

V této kapitole jsou popsány nejznámější algoritmy, které se momentálně k hledání cest používají. I když se metody používají pro hledání cest klasických, dají se převést na algoritmy disjunktní. Na začátku jsou uvedeny jednoduché algoritmy, které mají nižší úspěšnost nalezení řešení, a nebo jsou časově neefektivní. Dále v této sekci pak rozebírám více účinné a efektivní metody z hlediska úspěšnosti nalezení cest a času, které se mohou zdát složitější. Většina algoritmů pro hledání cest je používána a popsána v 2D prostorech, ale budou fungovat i v prostorech s jiným počtem dimenzí.

U algoritmů bude zkoumána *úplnost* a *optimálnost*. Algoritmus je úplný, pokud vždy najde alespoň jedno z možných řešení v konečném čase, v tomto

případě pokud vždy najde všechny cesty. Algoritmus je optimální, pokud vždy najde nejlepší řešení, tedy pokud najde nejkratší cesty. Z definicí vyplývá, že pokud je algoritmus optimální, tak musí být i úplný [10].

2.3.1 Algoritmus A*

Jeden z nepopulárnějších algoritmů pro hledání cest je A* [13]. Ten kombinuje vlastnosti *Dijkstrova algoritmu* [14] a *hladového prohledávání* (viz pseudokód, řádek č. 13). A* prohledává prostor informovaně, tudíž používá *heuristickou funkci*. Ta vyjadřuje odhad vzdálenosti pozice od cíle [15]. V textu bude značena jako funkce h(pozice). V práci je jako heuristika použita *Manhattanská vzdálenost* [16]. Pro lepší představu je pod tímto textem umístěn pseudokód tohoto algoritmu.

Nutno podotknout, že tento algoritmus neslouží pro řešení MAPF, ale hledá vždy pouze jednu cestu. Je zde uveden proto, že na něm jsou vystavěny ostatní algoritmy, jak CA*, HCA* a WHCA*, které už MAPF řeší. A* může jednoduše být upraven pro řešení MAPF tak, že se jednotlivé cesty budou hledat pomocí A* postupně za sebou. Problém tohoto přístupu je, že se vůbec neřeší kolize mezi jednotlivými agenty, což může být problematické.

A* je úplný v konečných prostorech. Zda je optimální závisí na zvolené heuristice. Pokud je heuristika *přípustná* a zároveň *konzistentní*, pak je A* optimální algoritmus [15]. Aby heuristika h byla přípustná, musí pro všechny cesty z vrcholu n do GOAL platit: $h(n) < Min(cost(path(n, GOAL)))$. Pokud je pro všechny vrcholy c a n, kde c je soused vrcholu n, splněna podmínka: $h(n) \leq h(c) + cost(edge(n,\ c))$, pak je heuristika h konzistentní. Časová složitost algoritmu v nejhorším případě je stejná, jako složitost algoritmu *Breadth first search (BFS)*, tedy $O(|V| + |E|)$ [17].

2.3.2 Algoritmy pro kooperativní MAPF

Všechny tyto algoritmy byly představeny Davidem Silverem v publikaci [18].

CA*

V CA* se agenti snaží spolupracovat, aby všichni dosáhli cíle. Každý agent si naplánuje svojí cestu pomocí A* [13] a pak ji zaznamená do tzv. *tabulky rezervací*, aby se zabránilo kolizím. Tabulka rezervací je prostor, ve kterém se hledají cesty, rozšířený o časovou dimenzi a je zde zaznamenáno, kde se budou agenti vyskytovat v čase t. Ale v případě, kdy hledáme cesty disjunktní, nebude tabulka rezervací příliš prospěšná, protože místem kudy vede jiná cesta, již nelze nikdy projít, a tak časová dimenze zde bude zbytečná.

Pokud by se plánovala pouze 1 cesta, pak je CA* úplný a s přípustnou a konzistentní heuristikou i optimální, měl by stejné vlastnosti jako A*. To se pro plánování více cest změní. CA* je neúplný. Zda najde všechny cesty závisí na pořadí hledání cest, protože nějaký agent může zablokovat jiného.

Algoritmus 2.1: A*

Vstup: prostor $S_{x,y,z}$, START, GOAL $\in S$
Výstup: cesta mezi uzly START a GOAL

```
1  START.distance ← 0;
2  opened ← {START};
3  closed ← ∅;
4  parents ← ∅;
5  while opened ≠ ∅ do
6  |  current ← pop(opened);
7  |  if current = GOAL then
8  |  |  return reconstruct_path(parents, START, END);
9  |  end
10 |  for x in {neigbors(S, current)} and not in closed do
11 |  |  if x ∉ opened or x.distance > (current.distance + 1) then
12 |  |  |  x.distance ← (current.distance + 1);
13 |  |  |  x.h ← manhattan_distance(x, GOAL);
14 |  |  |  parents ←add {x, current};
15 |  |  |  opened ←add x;
16 |  |  end
17 |  end
18 |  closed ←add current;
19 end
20 return ∅
```

A může dokonce nastat taková situace, že řešení existuje a CA* jej nenajde nezávisle na volbě pořadí agentů. Jelikož není úplný, tak není ani optimální. Časová složitost může být pro tento algoritmus v nejhorším případě nekonečná. Dále se bude pracovat se speciálním případem, kdy algoritmus všechny cesty úspěšně najde, protože tak lze získat lepší odhad časové složitosti.

Tvrzení 1. *Časová složitost algoritmu je $O(k \times (|V| + |E|))$, kde k je počet agentů či počet hledaných cest (pokud algoritmus úspěšně najde všechny cesty).*

Důkaz. Jako základ toho důkazu poslouží časová složitost A*, která byla odvozena v sekci č. 2.3.1. Jestliže se sestavuje k cest, pak složitost sestavení všech cest je v nejhorším případě $k \times (|V| + |E|)$, kde $O(|V| + |E|)$ je složitost A*. Kromě samotného sestavování je do algoritmu CA* přidána práce s rezervační tabulkou. Do rezervační tabulky se přidává během běhu algoritmu k-krát cesta o maximálně $|V|$ vrcholech, tedy v nejhorším případě zabere práce s rezervační tabulkou $O(k \times |V|)$. Časová složitost algoritmu je $O(k \times (|V| + |E|))$.

□

HCA*

Hierarchický kooperativní A* (HCA*) vylepšuje heuristiku klasického CA* (viz pseudokód, řádek č. 6). Využívá tzv. *abstraktní prostor*, ve kterém nejsou zaznamenány ostatní cesty. Jako heuristická funkce se používá velikost cesty z N do G v tomto abstraktním prostoru, tedy jaká by byla vzdálenost do cíle, kdyby neexistovali ostatní agenti (viz řádek č. 15). Pokud by musela být cesta v abstraktním prostoru hledána pokaždé kdyby bylo potřeba znát heuristiku, tak by tento algoritmus byl velice časově neefektivní. Proto bylo potřeba najít způsob, jak by se nalezené cesty v abstraktním prostoru dali znovu využít. David Silver pro tento účel použil algoritmus *resumable A* (RRA*)* [18], který zvyšuje efektivitu prohledávání abstraktního prostoru a umožňuje opětovné použití výsledků (viz řádek č. 23).

I přesto, že heuristika HCA* je konzistentní i přípustná [18], tak algoritmus není optimální ani úplný. Důvody jsou stejné jako u CA*, agent může zablokovat cestu jinému a tak nemusí být nalezeno řešení. Časová složitost je podobná jako u CA*, přidá se pouze prohledávání abstraktního prostoru pomocí RRA*. Ten má v nejhorším případě složitost jako BFS, tedy $O(|V|+|E|)$. Tím pádem sestavení jedné cesty trvá $O(2 \times (|V|+|E|))$, kde 2 jako konstantu můžeme vynechat. Nyní bude věnován prostor časové složitosti.

Tvrzení 2. *Časová složitost HCA* je rovna $O(k \times (|V| + |E|))$, stejně jako u CA* (Pokud se podaří úspěšně najít všechny cesty).*

Důkaz. HCA* je velice podobný algoritmu CA*, jediné co se liší je heuristika. Kvůli ní se musí pro každou cestu inicializovat a případně i dál prohledávat abstraktní prostor. Na toto prohledávání se používá algoritmus RRA*, který v nejhorším případě expanduje všechny vrcholy, tedy bude mít pro sestavování jedné cesty stejnou složitost jako BFS, která je rovna $O(|V| + |E|)$. A jestliže se sestavuje k cest, pak práce s abstraktním prostorem zabere nanejvýš $O(k \times (|V| + |E|))$. Časová složitost HCA* je proto rovna $O(k \times (|V| + |E|))$. □

WHCA*

CA* a HCA* mají kromě jiných tyto 3 problémy: 1) Výsledek záleží na pořadí hledání cest. 2) Na začátku se vytvoří velký plán, ale většina plánu se nevyužije pokud dojde ke kolizi s jinou cestou a musí se vytvořit nový. 3) Algoritmus skončí jakmile dosáhneme cílové pozice. Všechny tyto problémy řeší windowed hiearchical A* (WHCA*), který limituje hledání pomocí tzv. *okna*. Vždy je vyhledána částečná cesta k cíli, tu pak agent sleduje a jakmile překročí určenou vzdálenost, je vyhledána další částečná cesta. Po každém cyklu se zavolá metoda *shuffle*, která promíchá seznam agentů, a díky tomu má sestavování každé cesty chvíli prioritu. Jako heuristika se používá vzdálenost bodů v abstraktním prostoru, kde je cesta vyhledána celá, aby agenti směřovali správným směrem.

Algoritmus 2.2: Hierarchical cooperative A* (HCA*)

Vstup: prostor $S_{x,y,z}$, množina tasks obsahující páry <START, GOAL $\in S$ >

Výstup: Množina paths obsahující cesty mezi všemi uzly START a GOAL z množiny tasks

```
1  Algorithm HCA*(G, tasks)
2      paths ← ∅;
3      reservation_table ← prostor S_{x,y,z} rozšířený o časovou dimenzi t;
4      for <START, GOAL> in tasks do
5          initRRA*(START, GOAL) [18];
6          h←Abstract_distance(node, GOAL);
7          path ← A*(S, START, GOAL, h, reservation_table);
8          for i in (0, length(path)) do
9              node ← path.at(i);
10             reservation_table.at([node_x, node_y, node_z, i]) ← RESERVED;
11         end
12         paths ←add path;
13     end
14     return paths;
15 Procedure Abstract_distance(node, GOAL)
16     if node ∈ closed then
17         return node.distance;
18     end
19     if ResumeRRA*(node) = True then
20         return node.distance);
21     end
22     return ∞
23 Procedure ResumeRRA*(node)
24     while opened ≠ ∅ do
25         current ← pop(opened);
26         closed ←add current;
27         if current = node then
28             return True;
29         end
30         for x in {neigbors(S, current)} and not in closed do
31             if x ∉ opened or x.distance > (current.distance + 1) then
32                 x.distance ← (current.distance + 1);
33                 opened ←add x;
34             end
35         end
36     end
37     return False;
```

WHCA* stejně jako HCA* a CA* je neúplný a neoptimální. I WHCA* má v nejhorším případě nekonečnou časovou složitost. Vyjádřit složitost v situaci, kdy algoritmus úspěšně najde řešení, je komplikovanější než u CA* a HCA*. Hlavní důvod je, že se agenti mohou pohybovat i poté, co dorazí do cíle. Proto i v této situaci může být časová složitost až $O(\infty)$.

Tvrzení 3. *V případě, že WHCA* najde úspěšně všechny cesty, je i tak jeho časová složitost* $O(\infty)$.

Důkaz. U WHCA* se mění v každém cyklu priorita agentů, aby každý z nich měl chvíli maximální prioritu. O to se stará funkce *shuffle*, která znovu náhodně seřadí všechny agenty. Může ale nastat případ, kdy je možné všechny cesty dokončit pouze při jednom určitém seřazení agentů. Jelikož jsou ale řazeni náhodně, tak tato kombinace může nastat až v nekonečném čase, a proto je v nejhorším případě složitost $O(\infty)$, i když se podaří najít všechny cesty. □

2.3.3 Hledání cest na oktantovém stromu

Prostory jako voxelová mřížka jsou vhodné pro hledání cest a většina algoritmů na nich funguje bez větších úprav. Například u voxelové mřížky se dají sousední pozice získat jednoduše pomocí souřadnic a všechny hrany i vrcholy mají stejnou velikost. Naopak v oktantovém stromě se musí pro získání sousedů vrcholy prohledat část stromu. Navíc uzly zde mají různou velikost a nastává problém, že vyprodukovaná cesta vede přes uzly s různou velikostí, a tak musí být dále zpracována. V této sekci budou představeny metody, které umožňují hledání cest na oktantovém stromě.

Algoritmus pro hledání sousedů od Hanan Samet

Hledání sousedů je problém, kde je jako vstup dán uzel nebo pozice v prostoru a očekávaný výstup je seznam uzlů, na které se dá dostat z daného vrcholu, což jsou vrcholy, které mají s vstupním společnou hranu. Nejjednodušší metoda pro hledání sousedů v oktantovém stromě je pro všechny vrcholy zkontrolovat jejich pozice a velikost, a na základě toho rozhodnout, zda s původním sousedí. Tento přístup je velice časově náročný, protože musí pokaždé projít všechny uzly ve stromu.

Lepší algoritmus představil Hanan Samet ve své práci [19]. Jeho metoda hledá sousedu vrcholu v oktantovém stromě ve všech 27 možných směrech. V sekci 2.1.4 bylo určeno, že v této práci je definována spojitost pouze v 6 hlavních směrech, proto bude algoritmus upraven, aby hledal pouze tzv. „face neighbors“ [5]. Pro lepší představu o fungování metody je pod tímto textem, umístěn pseudokód algoritmu. Při jeho tvorbě byl použit popis nacházející se na webové stránce [20].

Metoda vždy hledá sousedy pouze v zadaném směru, pro každý vrchol musí být tedy zavolána jednou pro každý možný směr (viz pseudokód, řádek č. 3),

Algoritmus 2.3: Algoritmus od Hanan Samet pro hledání sousedů v oktantovém stromě

Vstup: Oktantový strom octree, uzel N, $N \in$ octree
Výstup: Množina neighbors obsahující všechny uzly, které sousedí s N

```
1  Algorithm Get_neighbors(octree, N)
2      neighbors ← ∅;
3      for Dir in Directions do
4          neig ← Get_greater_or_same_neighbors(octree, N, Dir);
5          neig_list ← Get_smaller_neighbors(octree, neig, Dir);
6          neighbors ← neighbors ∪ neig_list;
7      end
8      return neighbors;
9  Procedure Get_greater_or_same_neighbors(octree, N, Dir )
10     if N ∈ N.parent.children(opposite_dir(Dir) then
11         return N.parent.child(position(N) + Dir);
12     end
13     node ← Get_greater_or_same_neighbors(octree, N.parent, Dir);
14     if is_leaf(Node) then
15         return node;
16     end
17     else
18         return node.child(position(N));
19     end
20     return neigh;
21 Procedure Get_smaller_neighbors(octree, N, Dir )
22     candidates ← {N};
23     neig_list ← ∅;
24     while candidates ≠ ∅ do
25         current ← pop(opened);
26         if is_leaf(current) then
27             neig_list ←add current;
28         end
29         else
30             candidates ←add current.children(opposite_dir(Dir));
31         end
32     end
33     return candidates;
```

v tomto případě šestkrát. Samotný algoritmus je se rozdělen na 2 části. V té první se hledají pouze sousedící uzly, které mají stejnou nebo větší velikost než zdrojový uzel (řádek č. 9). Výstupem z této části bude maximálně 1 uzel (hledá se pouze v 1 směru). Nejdříve se testuje, zda není soused uzel se stejným rodičem. Pokud ne, tak je algoritmus rekurzivně zavolán pro rodiče uzlu, až se najde takový, který souseda obsahuje. Ten je poté nalezen a metoda ho vrátí.

V druhé části se hledají sousední uzly, které mají menší velikost než vrchol na vstupu (řádek č. 21). Jako vstup zde slouží uzel, který byl nalezen v první části metody. Ten se rekurzivně dělí na menší v opačném směru než se hledá, dokud se nenarazí na listy stromu, které není možno dál dělit. Ty pak metoda vrátí ve výstupu. V nejhorším případě má tento algoritmus složitost $O(n)$, ale tento případ nastává vzácně [19].

2.4 Výběr metod

Problém multiagentní hledání disjunktních cest je málo prozkoumaný a zatím bylo publikováno nízké množství metod, které tento problém řeší. Proto budou v praktické části navrhnuty nové algoritmy. Jako základ nových metod poslouží algoritmy pro kooperativní hledání cest CA*, HCA* a WHCA*, protože mají dobré výsledky pro hledání cest nedisjunktních (hlavně WHCA*). Nejdříve budou udělány pouze drobné úpravy tak, aby algoritmy fungovaly pro hledání cest disjunktních. Poté bude provedeno více změn, aby fungovaly pro speciální účely, jako je vyhýbání se překážkám nebo optimalizace vzájemné vzdálenosti cest.

Jako základní prostor pro hledání bude sloužit voxelová mřížka, která je jednoduchá na pochopení a také se s ní snadno pracuje. Jelikož se práce specializuje na trojrozměrné prostředí, budou některé metody implementovány, aby fungovaly na oktantovém stromě, což je struktura speciálně navržená pro 3D prostředí. Vlastnosti obou těchto prostorů budou srovnány v experimentální sekci.

Kapitola 3

Vlastní řešení

3.1 Algoritmy pro hledání disjunktních cest

3.1.1 CA*, HCA* a WHCA* pro disjunktní cesty

Algoritmy CA*, HCA* a WHCA* od Davida Silvera [18] vykazují dobré výsledky pro kooperativní MAPF, zvláště pak WHCA*. Proto byly modifikovány tak, aby fungovaly pro hledání cest disjunktních a zjistit tak, jaké budou mít výsledky pro řešení tohoto problému.

Algoritmus 3.1: CA* pro disjunktní cesty

Vstup: prostor $S_{x,y,z}$, množina tasks obsahující páry <START, GOAL $\in S$ >, heuristická funkce h
Výstup: Množina paths obsahující disjunktní cesty mezi všemi uzly START a GOAL z množiny tasks

```
1  paths ← ∅;
2  for node in tasks do
3  |  S.add_obstacles(node);
4  end
5  for <START, GOAL> in tasks do
6  |  S.remove_obstacles(START, GOAL);
7  |  path ← A*(S, START, GOAL, h);
8  |  S.add_obstacles(path);
9  |  paths ←add path;
10 end
11 return paths;
```

U všech tří byla odstraněna práce s rezervační tabulkou (viz algoritmus HCA* řádek č. 10), protože u disjunktních cest systém krátkodobých rezervací nefunguje. Jakmile agent projde nějakým místem, už žádný jiný tudy projít nesmí, vytvoří zde trvalou rezervaci. Proto není potřeba rezervační tabulky,

Algoritmus 3.2: WHCA* pro disjunktní cesty

Vstup: prostor $S_{x,y,z}$, množina tasks obsahující páry <START, GOAL $\in S$ >, velikost okna w

Výstup: Množina paths obsahující disjunktní cesty mezi všemi uzly START a GOAL z množiny tasks

```
1  Algorithm WHCA*(G, tasks, w)
2      paths ← ∅;
3      for node in tasks do
4          S.add_obstacles(node);
5      end
6      while tasks ≠ ∅ do
7          for <START, GOAL> in tasks do
8              h←Abstract_distance(node, GOAL);
9              S.remove_obstacles(START, GOAL);
10             partial_path ← WHA* (S, START, GOAL, h, w);
11             S.add_obstacles(partial_path);
12             if last(partial_path) = GOAL then
13                 paths ←add path;
14                 tasks ← tasks\{<START, GOAL>}
15             else
16                 START ← partial_path.at(w/2);
17             end
18             tasks ← shuffle(tasks);
19         end
20     end
21     return paths;
22 Algorithm WHA*(S, START, GOAL, h, w)
23     START.distance ← 0;
24     opened ← {START};
25     closed ← ∅;
26     parents ← ∅;
27     while opened ≠ ∅ do
28         current ← pop(opened);
29         if current = GOAL then
30             return reconstruct_path(parents, START, END);
31         end
32         if d ≥ w then
33             return reconstruct_path(parents, START, current);
34         end
35         for x in {neigbors(S, current)} and not in closed do
36             if x ∉ opened or x.distance > (current.distance + 1) then
37                 x.distance ← (current.distance + 1);
38                 parents ←add {x, current};
39                 opened ←add x;
40             end
41         end
42         closed ←add current;
43     end
44     return None
```

která pracuje s časovou dimenzí. Místo toho se budou zaznamenávat cesty přímo do prostoru. Po sestavení jsou všechny vrcholy (voxely), přes které cesta vede, přeměněny na překážky (viz Algoritmus 3 řádek č. 8), tudíž tudy již žádný jiný agent nemůže projít.

Při hledání disjunktních cest může nastat problém, že nějaká cesta bude vést přes startovní nebo cílovou pozici jiné cesty, která ještě nebyla sestavována. Pokud se takto stane, je hledání cest automaticky neúspěšné a je tedy potřeba tomu předejít. Proto na začátku algoritmu vždy budou všechny startovní a cílové pozice agentů označeny jako překážky. Jakmile pak začne hledání cesty, je překážka z její startovní a cílové pozice odstraněna. Po těchto změnách budou všechny 3 algoritmy fungovat pro hledání disjunktních cest. Jejich pseudokódy jsou zde uvedeny a v nich jsou barevně vyznačeny změny oproti originální verzi.

Algoritmus 3.3: HCA* pro disjunktní cesty

Vstup: prostor $S_{x,y,z}$, množina tasks obsahující páry <START, GOAL $\in S$ >

Výstup: Množina paths obsahující disjunktní cesty mezi všemi uzly START a GOAL z množiny tasks

```
1  paths ← ∅;
2  for node in tasks do
3  |  S.add_obstacles(node);
4  end
5  for <START, GOAL> in T do
6  |  initRRA*(START, GOAL) [18];
7  |  h←Abstract_distance(node, GOAL);
8  |  S.remove_obstacles(START, GOAL);
9  |  path ← A*(S, START, GOAL, h);
10 |  S.add_obstacles(path);
11 |  paths ←add path;
12 end
13 return paths;
```

Tyto pozměněné verze algoritmů pro disjunktní cesty mají podobné vlastnosti, jako jejich původní forma. Jsou neúplné a neoptimální, úspěšnost nalezení všech cest je ještě nižší, než u klasické verze. Časová složitost také je u všech tří nekonečná. Pokud bude zvažován případ, kdy budou úspěšně nalezeny všechny cesty, pak mají CA*, HCA* a WHCA* pro disjunktní cesty algoritmy složitost $O(k \times (|V| + |E|))$.

Tvrzení 4. *Časová složitost algoritmů CA* a HCA* pro disjunktní cesty je v nejhorším případě* $O(k \times (|V| + |E|))$*, pokud se úspěšně podaří najít všechny cesty.*

Tabulka 3.1: Srovnání vlastností upravených algoritmů CA*, HCA* a WHCA*

Algoritmus	Úplnost	Optimálnost	Časová složitost
CA*	ne	ne	$O(k \times (\|V\| + \|E\|))$
disjunktní CA*	ne	ne	$O(k \times (\|V\| + \|E\|))$
HCA*	ne	ne	$O(k \times (\|V\| + \|E\|))$
disjunktní HCA*	ne	ne	$O(k \times (\|V\| + \|E\|))$
WHCA*	ne	ne	$O(\infty)$
disjunktní WHCA*	ne	ne	$O(k \times (\|V\| + \|E\|))$

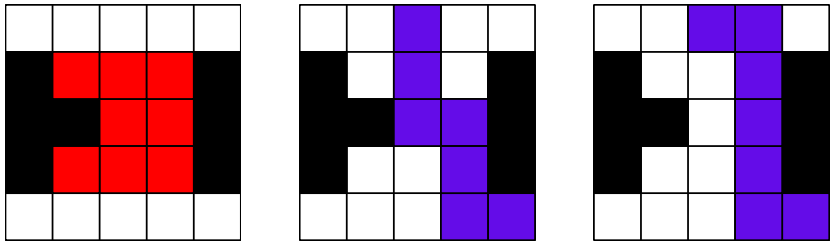
Důkaz. V algoritmech nebyly provedeny velké změny, jediná změna je odstranění práce s rezervační tabulkou a místo toho je prováděno přidávání překážek do prostoru. V nejhorším případě má přidání překážek pro všechny cesty složitost $O(|V|)$, protože každý vrchol může být součástí maximálně jedné cesty, a tak může být přidán do množiny překážek nanejvýš jednou. Tedy přidávání překážek zabere celkově čas $O(|V|)$, a to je lepší čas, než jaký by zabrala práce s rezervační tabulkou (viz 2.3.2). I když každý vrchol může být součástí jen jedné cesty, může být expandován vícekrát, maximálně k-krát. Proto i sestavení všech disjunktních cest zabere čas $O(k \times (|V| + |E|))$. Časová složitost CA*, HCA* pro disjunktní cesty je $O(k \times (|V| + |E|))$, takže je stejná jako časová složitost původních algoritmů. □

Tvrzení 5. *Časová složitost algoritmu WHCA* pro disjunktní cesty je v nejhorším případě* $O(k \times (|V| + |E|))$*, pokud se úspěšně podaří najít všechny cesty.*

Důkaz. Oproti klasickému WHCA*, který má časovou složitost $O(\infty)$, je hlavní rozdíl ve verzi pro disjunktní cesty to, že se agenti nepohybují po tom, co dorazí do cíle. Je to způsobeno tím, že i kdyby svojí cílovou pozici opustili, nepomohli by tím jinému agentovi, protože ten by tudy i tak nemohl projít. Kdyby ano, tak by se už nejednalo o disjunktní cesty. Proto každý agent expanduje maximálně k vrcholů. Složitost algoritmu WHCA* pro disjunktní cesty je $O(k \times (|V| + |E|))$, v případě, kdy se všechny cesty podařilo úspěšně nalézt. □

V tabulce č. 3.1 jsou porovnány vlastnosti algoritmů CA*, HCA* a WHCA* s jejich modifikacemi pro hledání disjunktních cest. V kolonce časová složitost je uvedena složitost v případě, že algoritmus najde všechny cesty, jinak může být nekonečná.

Z předběžných výsledků se ukazuje, že úspěšnost nalezení disjunktních cest u CA* a HCA* je podobná. Hledání pomocí HCA* zabere více času než u CA*, protože cesty ve skutečném prostoru se od těch v abstraktním prostoru výrazně odlišují a kvůli tomu musí být prohledána velká část abstraktního prostoru. Algoritmus WHCA* pro disjunktní cesty sestavuje cesty s nižší úspěšností než předchozí 2 algoritmy. Je to kvůli tomu, že v případě disjunktních cest



Vyznačení bottlenecku Ucpání průchodu Správné projití

Obrázek 3.1: Prostor s „bottleneckem“

se agenti nemohou vracet, a tak se často agenti dostanou do části prostoru, ve které uvíznou. Více bude zjištěno v experimentech.

3.1.2 Obstacle A*

V prostoru, kde se vyskytuje velké množství překážek lze nalézt tzv. „bottlenecky“. Tento výraz označuje úzké průchody, které kdyby se zaplnily překážkami tak bude prostor rozdělen na více souvislých komponent. Pokud hledáme disjunktní cesty, je potřeba aby vedly přes co nejmenší část „bottlenecku“, a tak tudy mohli projít i ostatní agenti. Problém je znázorněn na obrázku č. 3.1.

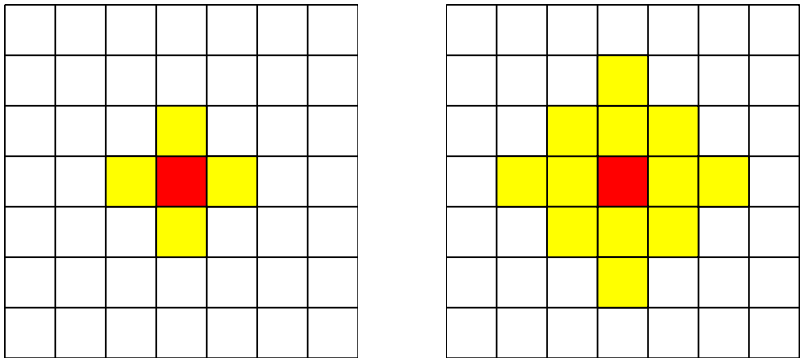
Algoritmus 3.4: Init obstacle nums

Vstup: prostor $S_{x,y,z}$, parametr p
Výstup: množina obstacle_nums obsahující 2 prvkové množiny, kde 1. prvek označuje vrchol V a druhý je hodnota, vyjadřující počet překážek v okolí vrcholu V.

```
1  obstacle_nums ← ∅;
2  for vrchol V in S do
3  |  obstacle_nums ←add {V, 0};
4  end
5  for obstacle O in S do
6  |  for V in neighbors_bigger_radius(O) do
7  |  |  obstacle_nums ←update {V, obstacle_nums[V] + p};
8  |  end
9  end
10 return obstacle_nums;
```

Popis algoritmu

Tento problém se snaží řešit nově navržený algoritmus *obstacle A**, a z něj odvozený *obstacle CA**. Algoritmus zkusí lokalizovat „bottlenecky“ a označit



neighbors neighbors_bigger_radius

Obrázek 3.2: Srovnání metod pro získávání sousedů

je tak, aby byly vrcholy z této oblasti penalizovány a heuristická funkce pro ně měla vyšší hodnotu. Díky tomu se agenti budou těmto oblastem vyhýbat a v případě nutnosti se je budou snažit projít co nejkratší cestou, a tak zde bude zachován prostor na projití i pro ostatní cesty.

Toho se docílí tak, že před samotným spuštěním algoritmu bude každému vrcholu v prostoru přiřazena hodnota, která reprezentuje počet překážek v okolí daného vrcholu. Toto číslo bude dále v textu označováno jako *obstacle_num*. Lze ho získat z množiny *obstacle_nums*, která bude obsahovat 2 prvkové množiny $\{V, obstacle\_num(V)\}$ pro každý vrchol V z prostoru. Ta je vytvořena na začátku běhu Obstacle A* v proceduře obstacle_nums_init, jejíž pseudokód je k nalezení zde.

Za povšimnutí stojí, že se v proceduře na řádku č. 6 nepoužívá pro nalezení sousedů fukce neighbors, ale neighbors_bigger_radius. Tato funkce nevrací pouze 6 „face“ sousedů, ale vrací jich až 32. Srovnání obou metod lze pozorovat na obrázku č. 3.2. Pokud by byla používána první metoda, mohlo by se stát, že nebudou penalizovány všechny vrcholy patřící do „bottlenecku“, protože ty které jsou ve středu této oblasti nejsou sousedé žádné překážky. Pokud se však použije funkce, která hledá sousedy ve větším prostoru, je větší šance, že se podaří najít a penalizovat všechny pozice v „bottlenecku“.

Heuristika obstacle A* pro vrchol V je definována následujícím předpisem:

$$h = manhattan\_distance(V, GOAL) + obstacle\_num(V)$$

Manhattanská vzdálenost může být nahrazena jinou heuristikou, je zde uvedena, protože je používána v této práci. Tato heuristika není přípustná ani konzistentní, ale tyto vlastnosti pro tento algoritmus nejsou potřeba. Naopak tato metoda nemá hledat nejkratší cestu, nýbrž cestu o trochu delší, která ale nezablokuje průchod pro ostatní.

Parametr p

V pseudokódu na řádku č. 7 se k hodnotě obstacle_num přičítá parametr p. Čím vyšší je jeho hodnota, tím víc jsou vrcholy poblíž překážek penalizovány a tím vyšší je celková délka cest.

Pro dobré fungování algoritmu musí mít parametr p nastaven na optimální hodnotu. Pokud bude příliš nízká, algoritmus bude fungovat úplně stejně jako CA*. Naopak, pokud bude moc vysoká, zvětší se celková délka cest příliš, a to povede k nižší úspěšnosti hledání cest, než by měl CA*. Pokud bude hodnota dobře nastavena, pak se celková délka cest zvedne pouze trochu a díky výhodě ze správného procházení skrz „bottlenecky“ úspěšnost sestavení cest vzroste.

Jak najít optimální hodnotu parametru p pro obstacle A*? To je velmi obtížný a někdy i nemožný úkol. Tato hodnota závisí na mnoha věcech, z nich nejdůležitější jsou: původní heuristika, velikost prostoru, počet a rozmístění překážek, rozmístění startovních a cílových pozic agentů a také počet agentů. V této práci se dospělo k závěru, že nejvhodnější je určit hodnotu p experimentálně.

To znamená zkusit problém vyřešit na dané mapě pro různé hodnoty parametru a vybrat tu s nejlepšími výsledky. Při to je třeba uvědomit si, že minimální hodnota obstacle_num pro vrchol je 0 a maximální $32p$. Testované hodnoty parametru p by měli být v rozmezí od 0 po maximální rozdíl heuristické funkce pro 2 vrcholy. Pokud bude p záporné, budou cesty blízko překážek a zdržovat se v „bottlenecích“ co nejdéle, a to je nežádoucí.

Vlastnosti algoritmu a použití

Algoritmus obstacle A* je úplný a vždy najde řešení, pokud existuje. Není optimální, takže ne vždy najde nejkratší cestu, ale to není po této metodě vyžadováno. Naopak by sestavené cesty měli být o trochu delší, ale měli by se vyhýbat místům s velkou koncentrací překážek.

U algoritmu obstacle CA*, který vznikne tak, že pro hledání cest v algoritmu CA* místo klasického A* použijeme tento obstacle A*, se vlastnosti změní. Ten totiž není optimální, ale stejně jako CA* není ani úplný a může se stát, že nějaká cesta zablokuje průchod ostatním.

Obstacle CA* je vhodný na použití pro hledání cest v prostorech, ve kterých se vyskytují úzké prostory a oblasti s velkou koncentrací překážek. Díky změněné heuristice se agenti těmto místům snaží vyhnout nebo jimi projít co nejrychleji, což vede k zvýšení úspěšnosti nalezení cest. Pokud naopak hledáme cesty v prostorech s velkým množstvím volného místa, a nebo v oblastech s rovnoměrným rozmístěným překážek, je obstacle A* nevhodný na použití a může způsobit nižší úspěšnost sestavení cest nebo časovou neefektivitu.

3.1.3 Hledání disjunktních cest na oktantovém stromě

V této sekci jsou popsány algoritmy a metody, které slouží k zprovoznění hledání cest na oktantovém stromě.

Konstrukce oktantového stromu

Ještě přes spuštěním samotného algoritmu je potřeba sestavit oktantový strom. Na začátku je vytvořen prázdný oktantový strom, kde v kořeni je jeden uzel, který je zároveň list a má velikost jako celý prostor. Poté je do oktantového stromu postupně přidána každá překážka z prostoru. Jakmile jsou přidány všechny, je konstrukce dokončena. Startovní a cílové pozice agentů se také počítají jako překážky, aby nebyly obsazeny jinou cestou.

Překážky se přidávají rekurzivním algoritmem. Ten začíná v kořeni stromu. Podle pozice překážky se vybere potomek aktuálního uzlu, kam překážka patří a algoritmus je rekurzivně zavolám pro tohoto potomka. Rekurze skončí, pokud je dosaženo maximální hloubky stromu. Aktuální uzel se pak přemění z volného prostoru na prostor obsazený. Nakonec se zkontroluje, zda přidáním této překážky nevznikl souvislý blok 8 překážek a pokud ano, jsou rekurzivně sloučeny do jednoho uzlu, který je umístěn o 1 hladinu výš.

Tvrzení 6. *Konstrukce oktantového stromu v nejhorším případě zabere čas* $O(max\_depth \times |V|)$*, kde* $|V|$ *je počet vrcholů v prostoru.*

Důkaz. V nejhorším případě jsou všechny vrcholy v prostoru překážky, tedy počet překážek je roven $|V|$. Rekurzivní algoritmus pro přidávání překážek se zastaví v poslední hladině stromu, počet jeho volání je v nejhorším případě roven max_depth. Následuje slučování překážek, které proběhne maximálně jednou pro každou hladinu, a počet hladin je nanejvýš max_depth. Časová složitost sestavení oktantového stromu je $O(max\_depth \times |V|)$. □

Maximální hloubka stromu bude v této práci volena tak, aby v poslední hladině stromu byly uzly o velikosti 1, a tak reprezentovaly voxely. Aby toto platilo, musí být maximální hloubka stromu o velikosti l rovna $\lceil log_2 l \rceil$. Pro zjednodušení se bude střed kořenového uzlu vždy nacházet na souřadnicích $[\frac{l}{2}, \frac{l}{2}, \frac{l}{2},]$.

A* a CA* na oktantovém stromu

Aby mohl být zprovozněn CA*, musí být nejprve zprovozněn samotný A*. Ten bude na oktantovém stromě fungovat tak, jak je definován v sekci 2.3.1, ale je nutné definovat několik operací, které fungují dobře na voxelové mřížce, ale pro oktantový strom musí být upraveny.

Hledání sousedů bude prováděno pomocí algoritmu od Samet, který byl definován v sekci č. 2.3.3 (viz pseudokód A* pro oktantový strom, řádek č. 11). Jako heuristika bude používána také Manhattanská vzdálenost. Jelikož

Algoritmus 3.5: A* na oktantovém stromu

Vstup: Oktantový strom octree, uzly START, GOAL ∈ octree
Výstup: cesta mezi pozicemi START a GOAL

```
1  START.distance ← 0;
2  opened ← {START};
3  closed ← ∅;
4  parents ← ∅;
5  while opened ≠ ∅ do
6      current ← pop(opened);
7      if current = GOAL then
8          path = reconstruct_path(parents, START, END);
9          return transform_path(octree, path);
10     end
11     for x in {find_neighbors_samet(current)} and not in closed do
12         if x ∉ opened or x.distance > (current.distance + current.size)
            then
13             x.distance ← (current.distance + current.size);
14             x.h ← manhattan_distance(x.center, GOAL.center);
15             parents ←add {x, current};
16             opened ←add x;
17         end
18     end
19     closed ←add current;
20 end
21 return ∅
22 Procedure transform_path(octree, path )
23     pos ← pop(path);
24     transformed_path ← {pos};
25     for node in path do
26         if node.depth = max_depth then
27             transformed_path ←add node;
28             pos ← node;
29         end
30         else
31             next ← path[index(node) + 1];
32             border_node ← octree.find_border_node(pos, node, next);
33             sub_path ← octree.find_path_in_node(pos, border_node);
34             transformed_path ←add sub_path;
35             pos ← border_node;
36         end
37     end
38     return transformed_path;
```

uzly mohou mít různou velikost, používá se jako pozice pro tuto metriku jejich střed (viz řádek č. 14). Před spuštěním A* bude vždy startovní a cílová pozice agenta v oktantovém stromu rozvětvena na uzel v poslední hladině, tedy na voxel. Tím se zaručí, že cesta povede minimálně přes 2 uzly a zjednoduší se transformace cesty na konci algoritmu (viz pseudokód CA* pro oktantový strom, řádek č. 8).

Další problém je přiřazování vzdáleností k vrcholům. Ve voxelové mřížce se jednoduše počátečnímu vrcholu přiřadí vzdálenost 0 a všem ostatním je přidělena vzdálenost o 1 vyšší, než je vzdálenost jejich předchůdce. To u oktantového stromu nejde, protože zde má každý vrchol jinou velikost. Přesné by bylo přičíst velikost cesty vedoucí přes předchozí vrchol k současnému vrcholu. Tento výpočet by byl časově nákladný, a tak se v této práci bude používat odhad skutečné vzdálenosti. Pro tento účel bude sloužit velikost hrany uzlu, která zhruba vyjadřuje velikost průměrné cesty přes daný vrchol, a tak může sloužit jako odhad (viz řádek č. 13). Podle předběžných výsledků funguje CA* pro oktantový strom dobře jak s odhadem vzdálenosti, tak i s přičítáním 1 pro různě velké vrcholy.

S těmito změnami bude A* na oktantovém stromu fungovat - v množině *opened* budou uzly stromu a v každém cyklu se jeden vytáhne a najdou se jeho sousedi. Pro ty je vypočítána heuristika a na základě té a vzdálenosti jsou přidány do *opened* či aktualizovány. Algoritmus skončí, až agent dorazí do cílové pozice, která se vždy nachází v listovém uzlu na poslední hladině.

Takto nalezená cesta, ale povede přes uzly s různou velikostí. Lepší by však bylo, aby vedla pouze přes uzly na poslední hladině, tedy přes voxely. Proč? První důvod je lepší srovnání s cestou vyprodukovanou ostatními algoritmy. Další důvod je pak větší množství informací o tom, jakou trasou má agent jít, protože přes uzel ležící na vyšší hladině stromu může projít více způsoby. Zatímco pokud se bude cesta skládat pouze z voxelů, existuje jen jedna možnost projití. Také to pomůže při přidávání překážek na pozice, přes které cesta vede, které probíhá na konci každého cyklu CA* pro disjunktní cesty. Pokud bychom jako překážky přidávaly uzly na vyšších hladinách, vznikal by zbytečně nevyužitý prostor.

Proto je cesta vždy po sestavování transformována tak, aby vedla pouze přes uzly na poslední hladině oktantového stromu. Algoritmus pro transformování postupně projede všechny uzly, přes které cesta vede. Pokud je uzel na vyšší hladině, jsou nejprve nalezeny všechny voxely, které leží na hranici s dalším uzlem v cestě (viz řádek č. 22). Poté je z těchto hraničních pozicí vybrána ta, která je nejblíže k cíli. Následně je nalezena dílčí cesta vedoucí od předchozího uzlu k této pozici, které vede pouze přes voxely. Startovní a cílové pozice byly již před algoritmem převedeny na voxely, takže to nebude dělat problém.

CA* s těmito metodami bude fungovat i na oktantovém stromě a tento algoritmus byl i úspěšně implementován. Jeho průběh se výrazně neliší od verze fungující na voxelové mřížce. Více informací bude získáno v experimentech.

Algoritmus 3.6: CA* pro disjunktní cesty na oktantovém stromu

Vstup: prostor $S_{x,y,z}$, množina tasks obsahující páry <START, GOAL $\in S$ >, heuristická funkce h
Výstup: Množina paths obsahující disjunktní cesty mezi všemi uzly START a GOAL z množiny tasks

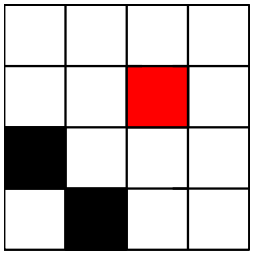
```
1  paths ← ∅;
2  octree ← construct_octree(S);
3  for node in tasks do
4  |  octree.add_obstacles(node);
5  end
6  for <START, GOAL> in tasks do
7  |  octree.remove_obstacles(START, GOAL);
8  |  octree.split_node_to_max_depth(START, GOAL);
9  |  path ← A*_octree(octree, START, GOAL, h);
10 |  octree.add_obstacles(path);
11 |  paths <--add-- path;
12 end
13 return paths;
```

HCA* na oktantovém stromu

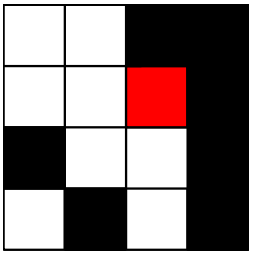
HCA* a CA* se liší pouze v heuristice. Heuristika HCA* je sofistikovanější a oproti odhadům vzdálenosti aktuálního vrcholu a cíle, se používá reálná velikost cesty mezi těmito 2 pozicemi v abstraktním prostoru, více informací naleznete v sekci č. 2.3.2. Při hledání na voxelové mřížce byla jako abstraktní prostor zvolena přesná kopie voxelové mřížky bez ostatních cest a agentů. Heuristika pak vyjadřovala informaci, jak dlouhá by byla cesta do cíle, kdyby neexistovaly ostatní agenti.

Je ale vhodné použít jako abstraktní prostor oktantový strom? Dospěl jsem k závěru, že ne. Prostor pro hledání cest si liší od abstraktního tím, že se v něm vyskytuje více překážek. U voxelové mřížky to nevadí, protože pro každý volný vrchol v prostoru se i tak v abstraktním prostoru vždy vyskytuje kopie se stejnou pozicí a velikostí. To u oktantového stromu neplatí. Tam se totiž po přidání překážky může změnit celá struktura stromu. Každý volný vrchol ze světa pro hledání cest bude obsažen i v abstraktním světě, ale tyto vrcholy mohou být na různých hladinách a tedy mít rozdílnou velikost.

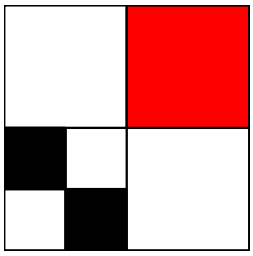
Tuto situaci ukazuje obrázek č. 3.3. Vidíme zde stejný svět znázorněn ve voxelové mřížce a v oktantovém stromě. Levé obrázky představují abstraktní prostory, tedy svět, kde neexistují ostatní agenti. V pravých obrázcích je pak model skutečného prostoru, kam byla již přidána jedna disjunktní cesta. Je vidět, že pokud by bylo potřeba znát heuristiku červeného vrcholu, tak ve voxelové mřížce je obraz v abstraktním prostoru stejný a budou tak do-



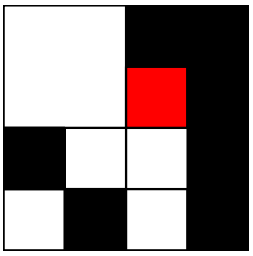
Voxelová mřížka - abstraktní prostor



Voxelová mřížka - reálný prostor



Oktantový strom - abstraktní prostor



Oktantový strom - reálný prostor

Obrázek 3.3: Rozdíly v abstraktním prostoru u použitých struktur

stupné přesné informace. Naopak obraz uzlu v oktantovém stromu se liší a má jinou velikost a hodnota heuristická funkce zde nebude tak přesná.

Kvůli této vlastnosti se heuristika HCA* stává nepřesnou. Není konzistentní, protože několik vrcholů v prostoru může být součástí 1 stejného vrcholu v abstraktním prostoru a heuristická funkce je pro ně tím pádem stejná, ale velikost cesty z každého z nich se může lišit. Je potřeba aby abstraktní prostor byl alespoň stejně detailní jako původní prostor.

HCA* byl na oktantovém stromu zprovozněn tak, že jako prostor pro hledání cest se používá oktantový strom, ale jako abstraktní prostor se používá voxelová mřížka, která je stejná jako prostor, ze kterého byl oktantový strom sestaven. Ta je vždy stejně nebo více detailní než oktantový strom.

U uzlu na vyšší hladině heuristika vrací velikost cesty z jeho středu do cíle. To může být nepřesné, ale u CA* používáme jako odhad vzdálenosti také vzdálenost středů uzlů, takže odhad této heuristiky by měl být stále přesnější než u CA*. U uzlů na poslední hladině vrací heuristika vzdálenost tohoto voxelu od cílového. Uzly na poslední hladině jsou v této práci vždy velké jako voxely, proto budou vždy obsaženy i ve voxelové mřížce se stejnou pozicí a velikostí.

WHCA* na oktantovém stromu

Podobně jako HCA* byl na oktantovém stromu zprovozněn i WHCA*. Jako abstraktní prostor je zde také použita voxelová mřížka. Problémové je využívání okna, protože je těžké určit, kdy je překročena jeho velikost kvůli různým

velikostem jednotlivých uzlů. Jako vzdálenost je použit pouze její odhad jako u CA*, tedy velikost uzlu je rovna vzdálenosti jeho předchůdce a k ní je přičtena velikost předchůdce. Každá částečná cesta bude mít o trochu jinou velikost, což ale fungování algoritmu příliš nevadí.

CA*, HCA* a WHCA* pro oktantový strom mají podobné vlastnosti jako jejich verze pro voxelovou mřížku. Jsou neoptimální a neúplné. Díky struktuře oktantového stromu by měli být rychlejší a potřebovat pro svůj běh méně paměti. Jejich úspěšnost se od originálu nebude příliš lišit, avšak drobné rozdíly mohou vznikat díky odhadování vzdálenosti velikostí uzlu. Více budou tyto metody srovnány v experimentech.

3.2 Pořadí sestavování cest

Jak již bylo zmíněno, u algoritmů CA*, HCA* velice záleží na pořadí sestavování jednotlivých cest. Předběžné pozorování ukazuje, že případ, kdy se cesty sestavují v nejhorším možném pořadí, má o 10-20% nižší úspěšnost nalezení všech cest, než když se sestavují v nejlepším možném pořadí. Určení priority agentů může být důležitější než samotný výběr algoritmu a je tedy vhodné se zamyslet, jak určit pořadí sestavování jednotlivých cest. V této sekci jsou uvedeny metody, pomocí kterých se přiřazovala priorita v této práci.

Relace priority

Ještě před uvedením daných metod je potřeba nejprve zadefinovat problém. Priorita agenta *prio* je číselná hodnota, která mu je přiřazena před sestavením cest. Tuto hodnotu přiřazuje zvolená metoda pro určování priority, popis těch používaných v této práci jsou k nalezení v sekci č .3.2. Na základě priorit jednotlivých agentů se určí výsledné pořadí hledání. Pokud má agent vyšší hodnotu priority než jiný, bude cesta pro něj vždy sestavována dříve. Aby tato platilo vždy, je potřeba v případě sestupného řazení nastavit hodnotu priority na $\frac{1}{prio}$.

Vztah, že agent má prioritu před jiným, je *binární relace* $\langle R_p, A, A\rangle$, kde A je množina obsahující všechny agenty. Mějme 2 agenty x a y. Agent x je v relaci R_p s agentem y, pokud má agent x prioritu před agentem y, tedy cesta pro agenta x bude sestavována před cestou pro agenta y. Tato skutečnost je označována xR_py. Informace o relacích a jejich vlastnostech byly čerpány z [21].

Tvrzení 7. *Relace R_p na množině A, která obsahuje všechny agenty, má tyto vlastnosti:*

- *není reflexivní ($\forall x \in A : xR_px$)*
- *není symetrická ($\forall x, y \in A : xR_py \implies yR_px$)*

- *je tranzitivní* $(\forall x, y, z \in A : xR_py \wedge yR_pz \implies xR_pz)$
- *je antisymetrická* $(\forall x, y \in A : xR_py \wedge yR_px \implies x = y)$
- *je asymetrická* $(\forall x, y \in A : xR_py \implies \neg(yR_px))$
- *je ireflexivní* $(\forall x \in A : \neg(xR_px))$

Definice vlastností byly také převzaty z [21].

Důkaz. Relace R_p je ireflexivní, protože nikdy nemůže být cesta nějakého agenta sestavována před cestou toho samého agenta. Tyto 2 cesty se budou vždy sestavovat najednou, a proto je tato relace ireflexivní. Jelikož je ireflexivní, tak logicky není reflexivní.

Pokud bude cesta pro agenta x sestavována před cestou pro agenta y, tedy platí xR_py, pak logicky nemůže být zároveň sestavována cesta pro agenta y před cestou pro agenta x, to je nemožné. Relace R tedy není symetrická, naopak je asymetrická. Každá asymetrická binární relace je i antisymetrická, a to platí i v tomto případě.

Priorita agentů je tranzitivní, protože pokud je cesta pro agenta x sestavována před cestou pro agenta y a zároveň cesta pro agenta y je sestavována před cestou pro agenta z, tak automaticky musí platit, že cesta pro agenta x je sestavována před cestou pro agenta z.

□

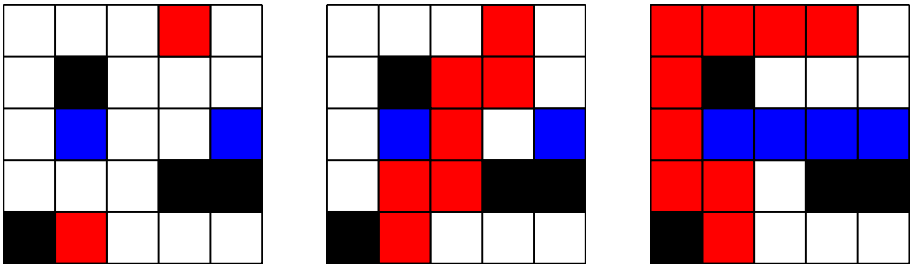
Relace R_p je zařazena podle jejích vlastností (ireflexivní, asymetrická, tranzitivní) mezi tzv. *ostrá uspořádání*. Množina agentů A, podle které se určuje pořadí hledání cest, je *úplně uspořádaná množina*.

Metody pro určení priority agentů

Většina v této práci použitých metod řadí jednotlivé agenty podle vzdálenosti startovní a cílové pozice. To má za následek to, že nejdříve budou sestaveny krátké cesty, a až později dojde k hledání delších cest (nebo naopak při sestupném seřazení). Podle předběžných výsledků funguje lépe vzestupné seřazení podle vzdálenosti, ale více bude ověřeno v experimentech.

První pokus byl použít odhad vzdálenosti cílového vrcholu od startovního. Agenti se v tomto případě řadili podle manhattanské vzdálenosti těchto pozic. Kromě ní byla použita i eukleidovská vzdálenost [22], ale výsledky obou těchto metrik byly velice podobné. Kromě odhadů byla použita i skutečná vzdálenost. Pro každého agenta byla pomocí A* nalezena cesta, poté se změřila její velikost a na základě ní byla přiřazena priorita. Tato metoda je časově náročnější, ale určení pořadí je pak přesnější než u odhadu vzdálenosti.

Kromě použití vzdálenosti startu a cíle byla ještě použita metoda, která řadila agenty podle změny počtu souvislých komponent prostoru [2]. Nejprve se pomocí algoritmu BFS se spočítal počet souvislých komponent v prostoru.



Zadání problému | Priorita rudé cRm | Priorita modré mRc

Obrázek 3.4: Pořadí sestavování cest

Poté byla pomocí A* sestavena cesta agenta, vrcholy přes které vedla byly přidány do prostoru jako překážky, a následně byl počet komponent spočítán znovu. Priorita byla udělena na základě rozdílu těchto 2 čísel. Nápad za tímto seřazením je ten, že cesta která rozdělí graf na více souvislých komponent může způsobit situaci, kdy start a cíl nějakého agenta budou v různých souvislých komponentách a hledání tím pádem skončí automaticky neúspěchem. Pokud se tato rozdělující cesta bude sestavovat později, je možné takovému scénáři předejít.

Příklad je znázorněn na obrázku č. 3.4. Celý prostor je tvořen 1 souvislou komponentou. Po sestavení modré cesty (agent m) se počet komponent nezmění, ale po sestavení červené (agent c) bude prostor rozdělen na 3 komponenty. Pokud se bude sestavovat červená cesta jako první, nebude možné sestavit modrou, protože start a cíl budou v různých komponentách, jak je vidět na prostředním obrázku. Naopak pokud se sestaví nejdříve modrá, podaří se úspěšně sestavit obě 2 cesty. Tento případ by byl úspěšně vyřešen, pokud by agenti byli vzestupně seřazeni podle změny počtu komponent nebo i vzestupně podle vzdálenosti startovního a cílového vrcholu.

Jelikož rozdíl komponent bývá často u agentů stejný, vznikla metoda kombinující oba přístupy. Agenti jsou seřazeni nejprve podle rozdílu počtu souvislých komponent a pokud je toto číslo pro některé z nich stejné, jsou seřazeni sekundárně podle vzdálenosti startu a cíle.

Všechny zde zmíněny metody budou srovnány v experimentální sekci a bude použita ta s nejlepšími výsledky.

3.3 Optimalizace vzájemné vzdálenosti cest

Většina současných algoritmů pro hledání cest se snaží najít co nejkratší cestu. Pokud je ale cíl, aby jejich vzájemná vzdálenost od sebe byla co největší, tak nejkratší cesty nebývají řešením. V této sekci budou představeny algoritmy a metody, které se snaží najít cesty co nejdál od sebe.

3.3.1 MDCA*

Algoritmus A* používá heuristickou funkci, aby cesta směřovala k cílové pozici. Tato heuristika by mohla být upravena tak, aby cesta vedla k cíli a zároveň, aby vedla co nejdál od ostatních cest. Tento způsob je využit v navrženém algoritmu *mutual distances cooperative A** (*MDCA**).

Popis algoritmu

Základ algoritmu tvoří CA*, je změněná pouze heuristika. Ta má následující předpis:

$$h(node) = old\_heuristic(node) - p \times (path\_distances[node])$$

Path_distances je slovník, kde jako klíče slouží vrcholy (voxely) v prostoru a jako hodnoty jsou zde uloženy vzdálenosti daných vrcholů od nejbližší cesty. Vzniká v proceduře *path_distances_init*, která je volána před začátkem algoritmu A*. Jedná se o modifikované BFS, kde na začátku jsou do množiny opened vloženy všechny vrcholy, přes které vede cesta a je jim nastavena vzdálenost 0. Poté se expandují další vrcholy a jim přiřazena vzdálenost podle vrcholu, který je objevil. Poté co jsou expandovány všechny, vrátí algoritmus slovník se vzdálenostmi, který slouží v heuristice jako *path_distances*.

Vzdálenost vrcholu od nejbližší cesty je vynásobena *parametrem p*. To je konstanta, která určuje, jak moc daleko od sebe jednotlivé cesty mají být. Čím je tato hodnota vyšší, tím je vyšší vzájemná vzdálenost cest, ale tím je také vyšší celková délka cest.

Heuristika MDCA*

Pojem old_heuristic označuje původní heuristiku, která byla používána v CA*. V tomto případě je použita Manhattanská vzdálenost. Heuristika MDCA* od této původní heuristiky přebírá některé vlastnosti. Například pokud je $p > 0$ a old_heuristic je přípustná, pak i heuristika MDCA* h je přípustná.

Důkaz. V přípustné heuristice musí být pro všechny cesty z vrcholu n do GOAL splněno toto: $h(n) < Min(cost(path(n, GOAL)))$. Předpis pro heuristiku h je následující: $h(n) = h\_old(n) - p \times (path\_distances[n])$. Z předpokladů je známo, že h_old je přípustná, tedy: $h\_old(n) < Min(cost(path(n, GOAL)))$. Heuristika h vznikne tak, že od h_old odečteme $p \times path\_distance$. Z předpokladů víme, že $p > 0$ a z popisu algoritmu init_path_distances je jasné, že pro všechna n je $path\_distance(n) \geq 0$ a tím pádem součin těchto 2 čísel je nezáporný. Proto pro všechna n platí: $h(n) \leq h\_old(n)$, pokud je $p > 0$. Pokud je tedy přípustná heuristika h_old, je přípustná i heuristika h. □

Heuristika h ale nemusí být vždy konzistentní. Příklad je zobrazen na obrázku č. 3.5. Pro všechny vrcholy v konzistentní heuristice musí být splněno:

Algoritmus 3.7: Init path distances

Vstup: prostor $S_{x,y,z}$, množina paths_nodes obsahující všechny vrcholy, přes které vede cesta
Výstup: množina path_distances obsahující 2 prvkové množiny, kde 1. prvek označuje vrchol a druhý jeho vzdálenost od nejbližší cesty

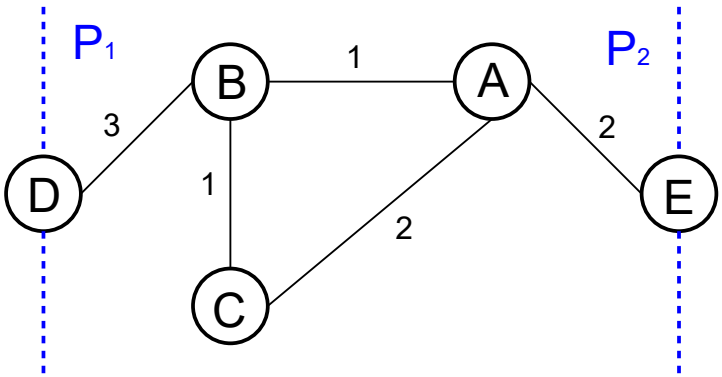
```
1  opened ← paths_nodes;
2  closed ← ∅;
3  path_distances ← ∅;
4  for node in opened do
5  |  path_distances ←add {node, 0};
6  end
7  while opened ≠ ∅ do
8  |  current ← pop(opened);
9  |  closed ←add current;
10 |  distance ← path_distances[current];
11 |  for x in {neigbors(S, current)} and not in closed do
12 |  |  if x ∉ opened then
13 |  |  |  path_distances ←add {x, distance};
14 |  |  |  opened ←add x;
15 |  |  end
16 |  end
17 end
18 return path_distances;
```

$h(n) \leq h(c) + cost(edge(n,\ c))$. Pokud si vypočteme heuristiku vrcholů A a B z příkladu, kde C je cílový vrchol a vrchol D je součást cesty $P1$ a vrchol E cesty $P2$, dostaneme $h(A) = 2 - p \times 2$ a $h(B) = 1 - p \times 3$. Měla by platit nerovnost $h(A) \leq h(B) + cost(edge(A, B))$, ale po dosazení dostaneme: $2 - p \times 2 \leq 1 - p \times 3 + 1$ a to je nesplněno pro $p \in (0, \infty)$. Heuristika MDCA* h tedy není vždy konzistentní.

Volba parametru p

Z předchozích odstavců může být vypozorováno, že vlastnosti heuristiky, a tedy i algoritmu, velmi závisí na volbě parametru p. Nyní budou odvozeny některé důležité hodnoty, a to hlavně *dolní a horní mezní hodnota parametru* p (dolní mez, horní mez).

Při hodnotě p, která je menší nebo rovna dolní mezi, by se měl algoritmus chovat tak, že celková délka cest nebude vyšší než u řešení vyprodukované pomocí původní heuristiky. I přesto může být vzájemná vzdálenost cest od



Obrázek 3.5: Nekonzistence heuristiky MDCA*

sebe vyšší než u původního řešení. Aby toto bylo splněno musí platit: Jsou zadány vrcholy A a B takové, že $p\_dist(A) > p\_dist(B)$ a $h\_old(A) > h\_old(B)$. Navíc $p\_dist(A) - p\_dist(B) = \arg\max_{x,y \in V} p\_dist(x) - p\_dist(y)$ a zároveň také $h\_old(A) - h\_old(B) = \arg\min x, y \in V |h\_old(x) - h\_old(y)|$. Pokud i přes to je garantováno, že $h(B) \leq h(A)$, pak parametr p má hodnotu menší nebo rovnou dolní mezi p.

Proč? Protože nahoře je ukázán nejextrémnější případ (rozdíl path_distance vrcholů je co největší a naopak rozdíl původních heuristik co nejmenší) a pokud ten vrchol s menší původní heuristikou má i tak menší výslednou heuristiku, bude tak platit i pro ostatní, které mají rozdíl path_distance menší. Jediné kdy path_distance ovlivní porovnání vrcholů je situace, kdy původní heuristiky uzlů jsou stejné.

Jaká je hodnota dolní meze parametru p? Lze ji určit pomocí této rovnice, kde se pracuje s vrcholy A a B z extrémního případu nahoře.

$$
\begin{aligned}
h(A) - p \times p\_dist(A) &\geq h(B) - p \times p\_dist(B) \\
h(A) - h(B) &\geq p \times (p\_dist(A) - p\_dist(B)) \\
\arg\min_{x,y \in V} |h\_old(x) - h\_old(y)| &\geq p \times \arg\max_{x,y \in V} p\_dist(x) - p\_dist(y) \\
\frac{\arg\min_{x,y \in V} |h\_old(x) - h\_old(y)|}{\arg\max_{x,y \in V} p\_dist(x) - p\_dist(y)} &\geq p
\end{aligned}
$$

Pokud je potřeba, aby řešení MDCA* měla co nejmenší celkovou délku cest a vzájemnou vzdálenost cest používala až jako sekundární kritérium, pak musíme zvolit p z intervalu

$$(0, \frac{min\ h\_\text{old diff}}{max\ p\_\text{dist diff}}).$$

Číslo max p_dist diff lze najít v množině path_distances v čase $O(|V|)$. Najít min h_old diff je v neznámých prostorech časově náročnější, zabere $O(|V|^2)$.

V našem případě, kdy používáme jako h_old Manhattanskou vzdálenost a jako prostor pravidelnou voxelovou mřížku, bude min h_old diff vždy rovno 1, a tak hledání není potřeba.

V opačném případě, pokud p bude větší nebo rovno horní mezní hodnotě p, bude algoritmus hledět pouze na to, aby vzdálenost vrcholu od nejbližší cesty byla co největší. Pouze pokud budou mít 2 uzly tuto vzdálenost stejnou, pak se bude rozhodovat podle původní heuristiky. Horní mezní hodnotu parametru p lze získat podobným způsobem, jak byla získána dolní mez, pomocí extrémního případu, kde vrcholy A a B budou mít maximální rozdíl v hodnotě h_old a minimální rozdíl path_distance, přičemž $h\_old(A) < h\_old(B)$ a $p\_dist(A) < p\_dist(B)$, potom musí platit, že $h(A) > h(B)$. Po sestavení a vyřešení podobné nerovnice jako zde, vyjde:

$$\frac{\arg\max_{x,y \in V} |h\_old(x) - h\_old(y)|}{\arg\min_{x,y \in V} p\_dist(x) - p\_dist(y)} \leq p.$$

Konstantu min p_dist diff lze získat z množiny path_distances v čase $O(|V|^2)$, nebo může být toto číslo nahrazeno 1, protože z popisu path_distances_init je známo, že min p_dist diff nemůže být menší než 1. Údaj max h_old diff jde nalézt v čase $O(|V|)$.

Pokud je potřeba, aby se při sestavování cesty pomocí MDCA* maximálně hledělo na vzájemnou vzdálenost cest od sebe a až sekundárně se rozhodovalo podle původní heuristiky, pak je nutné zvolit p z intervalu

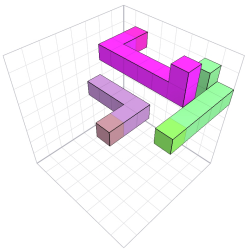
$$(\frac{max\ h\_\text{old diff}}{min\ p\_\text{dist diff}}, \infty).$$

Horní mezní hodnota p bude vždy větší nebo rovna dolní mezní hodnotě p, protože max h_old diff $\geq$ min h_old diff a min p_dist diff $\leq$ max p_dist diff. Pokud zvolíme p mezi horní a dolní mezní hodnotou, pak se bude při sestavování cesty hledět na obě kritéria: původní heuristiku a vzdálenost vrcholu od nejbližší cesty. Čím blíž je p horní mezi, tím větší prioritu má vzdálenost vrcholu od nejbližší cesty a naopak čím blíž je dolní mezi, tím větší prioritu má původní heuristika.

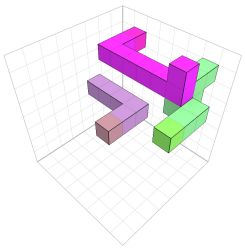
Pokud je parametr p menší než 0, pak heuristika MDCA* není ani konzistentní, ani přípustná. Celková délka cest bude stejná jako u řešení vyprodukovaném pomocí CA* s původní heuristikou, protože každé záporné p je menší než dolní mezní hodnota p. Navíc místo toho, aby byly cesty od sebe co nejdále, budou v řešení cesty navzájem co nejblíž a to rozporuje cílům této práce. Proto se p menší než 0 v této práci nepoužívá.

Na obrázku č. 3.6 je ukázáno, jak MDCA* funguje pro různé hodnoty parametru p. Prostor zde tvoří voxelová mřížka $8 \times 8 \times 8$, a úkolem bylo vytvořit 3 cesty:

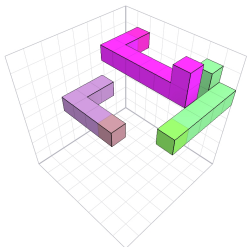
- z [7,6,6] do [5,1,7]



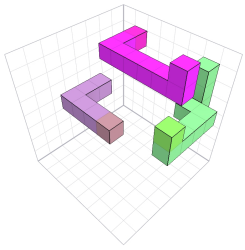
CA*, MD=119.45, L=22



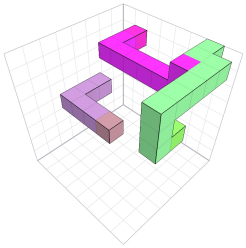
p=−3, MD=115.89, L=22



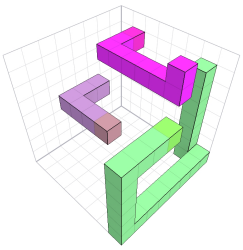
p=$\frac{1}{20}$, MD=130.37, L=22



p=3, MD=143.34, L=24



p=6, MD=162.25, L=26



p=20, MD=289.23, L=37

Obrázek 3.6: Srovnání řešení MDCA* pro různé hodnoty p

- z [2,3,4] do [4,6,4]
- z [3,0,5] do [7,1,6]

Dolní mezní hodnota p je $\frac{1}{20}$, horní pak 20. Pro srovnání je připojeno i CA* řešení. Je vidět, že s rostoucím p se zvyšuje vzájemná vzdálenost cest, ale s tím se zvyšuje i celková délka cest. Navíc po porovnání řešení od CA* a od MDCA*, $p = \frac{1}{20}$ lze vypozorovat, že jde zvýšit vzájemná vzdálenost cest, aniž by jsme zvýšili celkovou délku. Tím pádem pokud se zohledňuje vzájemná vzdálenost, je lepší než CA* využít MDCA* s takovým p, které je menší nebo rovno dolní mezní hodnotě p.

Vlastnosti algoritmu

MDCA* má podobné vlastnosti jako CA* pro disjunktní cesty. Je neúplný a neoptimální. Časová složitost může být nekonečná, pokud se nepodaří najít

žádné řešení. Pokud se bude počítat pouze s případy, kdy se řešení najde, pak je časová složitost algoritmu MDCA* $O(k \times (|V| + |E|))$.

Tvrzení 8. *Časová složitost MDCA* algoritmu je rovna* $O(k \times (|V| + |E|))$ *v případě, kdy jsou úspěšně nalezeny všechny cesty.*

Důkaz. Důkaz je založen na časové složitosti algoritmu CA* pro disjuntkní cesty, která je rovna $O(k \times (|V| + |E|))$ (viz tvrzení č. 4). Oproti CA* jsou v algoritmu MDCA* 2 změny. První z nich je rozdílná heuristika, která na časovou složitost nemá vliv, jelikož její výpočet je stejně náročný jako výpočet heuristiky CA*. Druhá je přidání procedury path_distances_init. Ta má ale stejnou složitost jako BFS, tedy $O(|V| + |E|)$, protože každý vrchol je expandován také maximálně jednou a na každou hranu lze narazit maximálně 2x. Procedura path_distances_init je jednou před sestavováním každé cesty, tedy k-krát. Proto časová složitost MDCA*, pokud najde všechny cesty, je $O(k \times (|V| + |E|))$. □

3.3.2 MDWHCA*

Kromě CA* byl pro optimalizaci vzájemné vzdálenosti cest navrhnut ještě jeden algoritmus. Je inspirován WHCA* a proto bude v tomto textu nazýván jako *MDWHCA**, tedy *Mutual distances windowed hierarchical A star*. U MDCA* první sestavované cesty vedou co nejkratší trasou k cíli, protože nemají ještě žádné informace o ostatních cest, což může vést ke snížení vzájemné vzdálenosti cest. U MDWHCA* se agenti střídají v sestavování částečných cest, takže tento problém zde bude redukován.

Algoritmus MDWHCA* probíhá stejně jako WHCA*, jenom je změněna heuristika stejně jako u MDCA*. Předpis pro heuristiku je tedy:

$$h(node) = old\_heuristic(node) - p \times (path\_distances[node]),$$

více informací o ní najdete v sekci č. 3.3.1 a zde byly zavedeny pojmy a metody, se kterými se v této sekci bude pracovat.

Oproti MDCA* je rozdílná původní heuristika algoritmu *h_old*, která je u MDWHCA* velikost cesty z aktuálního vrcholu do cíle v abstraktním prostoru. Tato heuristika je podobná odhadu vzdálenosti pomocí manhattanské metriky, minimální rozdíl je také 1, ale maximální rozdíl může být větší. Proto se horní a dolní mezní hodnota parametru p bude lišit. Obě čísla se dají spočítat podobným způsobem jako u MDCA*, proto zde tentokrát odvození těchto čísel nebude uvedeno.

Další změna v algoritmu je volání procedury path_distances_init. Ta je v MDCA* volána po dokončení každé cesty, aby byly informace o prostoru aktuální. Pokud by tato procedura byla v MDWHCA* volána po sestavení každé částečné cesty, byl by algoritmus časově neefektivní. Proto je metoda path_distances_init v MDWHCA* volána pouze na konci každého cyklu, tedy po tom co každý z agentů dokončí jednu částečnou cestu.

Tabulka 3.2: Heuristiky algoritmů pro hledání disjunktních cest

Algoritmus	Předpis heuristiky
CA*	manhattan_dist($N, GOAL$)
HCA*	abstract_dist(N)
WHCA*	abstract_dist(N)
obstacle CA*	manhattan_dist($N, GOAL$)+obstacle_num(V)
MDCA*	manhattan_dist($N, GOAL$)$-p\times$(path_distances[N])
MDWHCA*	abstract_dist(N)$-p\times$(path_distances[N])

Tabulka 3.3: Srovnání vlastností algoritmů pro hledání disjunktních cest

Algoritmus	Přípustná h	Konzistentní h	Účel algoritmu
CA*	ano	ano	Hledání nejkratších disjunktních cest
HCA*	ano	ano	Hledání nejkratších disjunktních cest
WHCA*	ano	ano	Hledání nejkratších disjunktních cest
obstacle CA*	ne	ne	Hledání cest v prostorech s bottlenecky
MDCA*	ne	ne	Optimalizace vzájemné vzdálenosti cest
MDWHCA*	ne	ne	Optimalizace vzájemné vzdálenosti cest

Pro vyšší hodnotu p se stává, že MDWHCA* často nenajde cestu. U MDCA* se nejprve prohledává celý prostor a až poté je sestavena cesta. Zatímco u MDWHCA* se cesta sestavuje postupně od začátku. Proto se ale stává, že proto, aby se agent vzdálil od ostatních cest zabloudí do míst, ze kterých se již nedostane do cíle a tak je hledání cest neúspěšné. Proto je lepší u MDWHCA* volit hodnoty parametru p blíže dolní mezní hodnotě.

3.4 Shrnutí nových metod

Na začátku byly modifikovány algoritmy pro kooperativní hledání cest CA*, HCA* a WHCA* tak, aby fungovaly pro případ hledání disjunktních cest. V těchto modifikovaných algoritmech bylo provedeno co nejméně změn, aby byly co nejvíce podobné jejich původním verzím. Jako prostor pro hledání cest byla zvolena voxelová mřížka. Všechny 3 algoritmy pro disjunktní cesty byly poté zprovozněny i na oktantovém stromě.

Dále byla změněna heuristika CA* tak, aby byly pozice, kolem kterých je velké množství překážek, penalizovány. Cesty by se pak měly těmto regionům vyhýbat a nebo se zde zdržovat co nejméně. Tento algoritmus byl nazván Obstacle A* a měl by úspěšnější v hledání řešení na mapách, kde se vyskytují úzké průchody.

Mezi cíli této práce byla i optimalizace vybraných kritérií. Jelikož o co nejmenší celkovou délku cest se snaží algoritmy, které byly již implementovány (CA*, HCA*, WHCA*), bylo potřeba ještě vymyslet metodu, která se bude

snažit o co největší vzájemnou vzdálenost cest od sebe. Pro tento účel byly navrhnuty algoritmy MDCA* a MDWHCA*, které na základě parametru p hledají cesty, které mezi sebou mají větší mezery.

Všechny tyto algoritmy jsou neúplné a neoptimální, kvůli interakcím mezi agenty. Agenti může mít od jiného zablokovanou cestu, a tak bude jeho trasa delší, a dokonce může být zablokována cest úplně, což vede k nevyřešení problému. Srovnání vlastností těchto nových algoritmů je provedeno v tabulkách Heuristiky algoritmů pro hledání disjunktních cest a Srovnání vlastností algoritmů pro hledání disjunktních cest. Další údaje o všech nových algoritmech budou získány experimentálně v sekci č. 4.

Kapitola 4

Experimenty

4.1 Popis experimentů

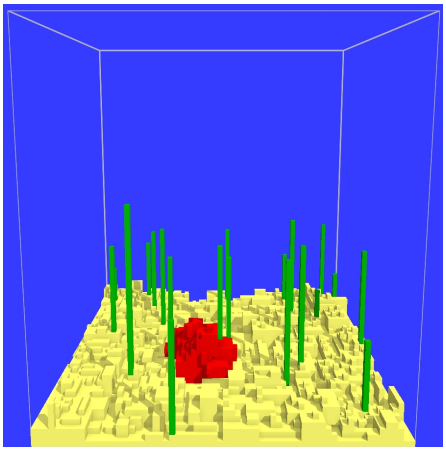
V této sekci jsou popsány experimenty teoreticky, praktická část je popsána v sekci č. 4.2. Naleznete zde informace o mapách pro hledání cest, které budou použity v experimentech. Dále je umístěn přesný popis zkoumaných kritérií a popis použitého hardware a software. A na konci kapitoly jsou ještě jednou vyjmenovány všechny použité algoritmy.

4.1.1 Použité mapy

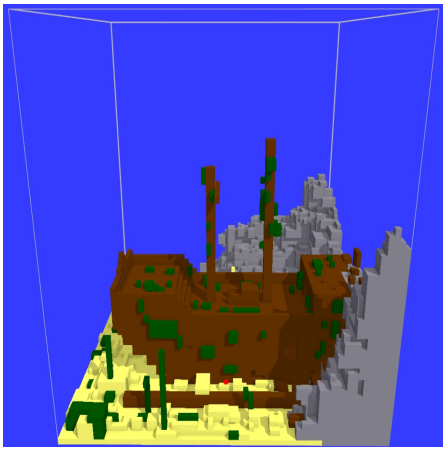
Pro experimenty byly vytvořeny 4 speciální mapy, na kterých se budou hledat disjunktní cesty. Všechny jsou ve formě voxelové mřížky a všechny mají stejné rozměry. Jejich původní velikost byla $64 \times 64 \times 64$, ale hledání cest probíhalo moc dlouho, takže byly zmenšeny na $32 \times 32 \times 32$. Díky těmto rozměrům ($2^5 \times 2^5 \times 2^5$) je lze snadno převést na oktantový strom o maximální hloubce 5. Jak již bylo nastíněno v úvodu, navržené algoritmy mohou využívat třeba podvodní roboti. Proto jsou na mapách znázorněny místa pod vodou a jiná, kde by mohly být tyto roboti uplatněni.

První mapa se nazývá sea_bottom a znázorňuje typické mořské dno. Ze všech čtyř je tu nejvíce volného prostoru a nejsou zde žádné úzké a problémové průchody. Tento model může vyjadřovat situace, kdy se roboti musí potopit na dno a vykonat zde nějakou činnost. Podobná je mapa shipwreck, kde je také vyobrazeno mořské dno a skály. Uprostřed mapy je umístěn vrak lodi, který může být roboty prozkoumáván. Je zde hodně volného prostoru, ale do vraku lodi vede pouze pár úzkých průchodů. Tato oblast je tedy kombinací míst s velkou koncentrací překážek a míst s koncentrací velmi malou.

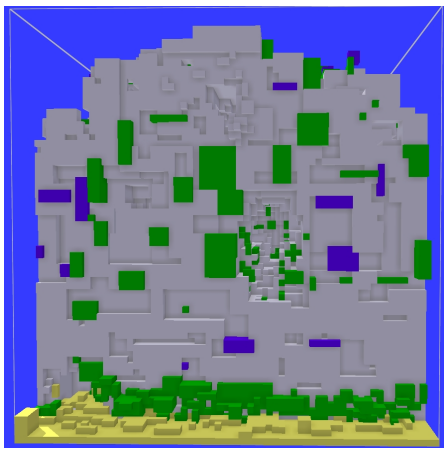
Třetí mapa v pořadí se jmenuje cave a je na ní vyobrazena podmořská jeskyně. Ta pokrývá prakticky celou mapu a vede do ní pouze omezené množství vchodů. Toto je naopak příklad prostředí s velkou koncentrací překážek a často se zde může stát, že se cesty navzájem zablokují. Sewers je poslední mapa a



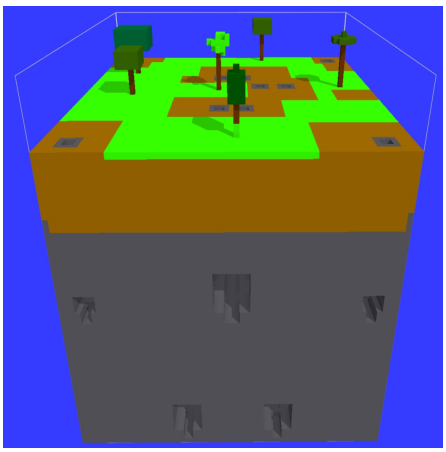
Bottom, 2 549 překážek



Shipwreck, 4 421 překážek



Cave, 23 859 překážek



Sewers, 25 179 překážek

Obrázek 4.1: Zobrazení map použitých v experimentech

modelovat systém kanalizace. Jedná se také o stísněný prostor, dokonce jsou tu ještě užší průchody než v jeskyni. Oblast kanalizace by se jako jediná dala popsat jako pravidelná. Všechny mapy jsou k nalezení na obrázku č. 4.1.

4.1.2 Zkoumaná kritéria

Nejdůležitějším zkoumaným kritériem je úspěšnost nalezení řešení. Pod tímto pojmem se bude uvádět výsledek výrazu počet úspěšně vyřešených modelů ÷ počet všech modelů a tato veličina bude značena jako MSR (model success rate). Jako model je označována uspořádaná dvojice {Space, Tasks}, kde Space je prostor ve které se budou hledat cesty a množina Tasks obsahuje zadání problému, konkrétně startovní a cílové pozice agentů. Model se považuje za úspěšně vyřešený, pokud se podařilo najít všechny cesty. U experimentů bývá

mapa u sady modelů stejná a mění se pouze zadání agentů.

Kromě úspěšnosti vyřešení modelu se ještě pracuje s úspěšností sestavení cesty. Tento údaj lze získat po vyřešení výrazu počet úspěšně sestavených cest ÷ celkový počet sestavovaných cest. Tato hodnota není tak informačně přínosná jako úspěšnost modelu, protože některé algoritmy se po první neúspěšně sestavované cestě mohou zastavit a nesnažit se o dokončení ostatních, ale pro algoritmy navržené v této práci se tak neděje. Veličinu budeme značit PSR (path success rate). Někdy se místo PSR použije hodnota s názvem CP (completed paths), která označuje počet úspěšně sestavených cest.

Vzájemná vzdálenost cest (MD) a celková délka cest (L) jsou další kritéria, které budeme zkoumat a již byly představeny v sekci č. 2.1.6. Jejich velikost bude ve většině případů moc velká a těžká pro představu, proto se místo nich bude používat průměrná velikost cesty a průměrná vzájemná vzdálenost cest. Zatímco průměrnou velikost cesty dostaneme, když vydělíme celkovou délku všech cest počtem všech sestavovaných cest, tak průměrná vzájemná vzdálenost cest je rovna součtu vzájemné vzdálenosti cest všech modelů vydělením počtem všech modelů.

Další důležitá veličina je čas (t). Proto bude měřen čas běhu každého algoritmu.

4.1.3 Systém

Co se týká hardware, experimenty byly prováděny na zařízení s 4-jádrovým procesorem Intel Core i7-5700HQ, která má základní frekvenci 2.70 GHz, s pamětí RAM o velikosti 16 GB a s operačním systémem Windows 10.

A nyní pár vět o použitém software. Všechny testované algoritmy byly implementované v programovacím jazyku *Python* verze 3.7. Pro matematické operace a pro zpracování voxelové mřížky byla použita knihovna *numpy* [23]. Celá voxelová mřížka je uložena do numpy ndarray a s touto strukturou se pracuje v implementovaných algoritmech.

Pro tvorbu map (ze sekce 4.1.1) a voxelových mřížek a také pro jejich vizualizaci byl použit freeware program *MagicalVoxel* [24] ve verzi 0.99.6.2. Obrázky v této práci, na kterých je zobrazena voxelová mřížka byly vytvořeny zde. V této technologii lze vytvořit 3D mapy a exportovat je do formátu .vox, a nebo naopak .vox soubory otevřít.

Pro převod tohoto formátu na numpy ndarray byla využita knihovna v jazyce Python *py-vox-io* [25]. Knihovna obsahuje i obrácenou funkci, která předvádí numpy ndarray na .vox soubory, které mohou zobrazeny v Magical Voxel. Pro vizualizaci struktury oktantových stromů byla použita knihovna *Open3D* [26].

4.1.4 Testované algoritmy

V experimentech budou otestovány všechny algoritmy, které byly navrženy v této práci. Nejdříve budou mezi sebou srovnány CA*, HCA* a WHCA* pro disjunktní cesty na voxelové mřížce. Další experiment bude o metodách, které určují pořadí sestavování cest. Následně budou testovány datové struktury voxelová mřížka a oktantový strom. Obstacle CA* bude poté otestován a srovnán s klasickým CA* pro disjunktní cesty. A v posledním experimentu se bude řešit optimalizace vzájemné vzdálenosti cest a tedy algoritmy MDCA* a MDWHCA*.

Pokud se bude v experimentech mluvit o nějakém algoritmu, například o CA*, HCA*, WHCA*, ..., myslí se tím verze algoritmu pro hledání disjunktních cest, pokud není napsáno jinak.

4.2 Výsledky experimentů

Experimenty lze nalézt v softwarovém prototypu označeny stejnými čísly, jako v tomto textu.

‘

4.2.1 Experiment č.1 - metody pro řazení agentů

První experiment se bude věnovat metodám pro sestavení pořadí hledání jednotlivých cest. Bylo vybráno 5 metod. První 2 metody řadí agenty podle odhadu vzdálenosti jejich začáteční a cílové pozice. Odhad podle manhattanské vzdálenosti bude označován *manhatt* a odhad podle eukleidovské vzdálenosti pak *euclid*. V algoritmu označeném jako *path_len* jsou agenti seskupeni podle skutečné velikosti cesty, nejprve se sestaví všechny cesty pomocí A* a změří se jejich délka a na základě toho jsou pak seřazeny.

Poslední metody nazvané *splitting* používají k uspořádání cest hodnotu počet souvislých komponent před sestavením cesty dělený počtem souvislých komponent po sestavení cesty. V *split + euclid* jsou pak cesty primárně řazeny podle rozdílu počtu souvislých komponent a sekundárně podle eukleidovské vzdálenosti. Více informací o těchto algoritmech jsou k nalezení v sekci č. 3.2. ↑ pak označuje vzestupné řazení a ↓ řazení sestupné.

Pro všechny metody na řazení agentů se pro sestavování cest používal algoritmus CA* pro disjunktní cesty a jako prostor byla zvolena voxelová mřížka. Všechny algoritmy navržené v této práci jsou si podobné, a proto ten způsob uspořádání, který funguje dobře pro CA*, bude fungovat dobře i pro ostatní algoritmy.

Tabulka 4.1: Experiment 1.1

Metoda	Seřazení	MSR	CP	$tSF[s]$	L	MD
euclid	↑	15/25	876	0.013	41.892	608 011
euclid	↓	8/25	870	0.010	43.685	630 141
manhatt	↑	15/25	882	0.014	41.941	616 452
manhatt	↓	6/25	863	0.009	43.565	620 528
path_len	↑	16/25	885	26.429	41.928	621 246
path_len	↓	5/25	863	23.520	47.043	616 607
spliting	↑	13/25	884	885.196	42.139	633 850
spliting	↓	5/25	870	937.940	43.057	619 001
split + euclid	↑	15/25	880	964.271	42.207	619 811
split + euclid	↓	8/25	876	934.043	45.367	632 191

Experiment 1.1

První část experimentu 1 bude značená jako experiment 1.1. V tom se na mapě cave náhodně vygeneruje 25 různých modelů, a v každém z nich se hledají cesty pro 36 agentů. Veličiny jsou popsané v sekci č. 4.1.2. Nová je veličina tSF zde neoznačuje délku běhu celého algoritmu pro hledání cest, ale pouze čas běhu metody pro přiřazení priority agentům. Experiment 1.1 oproti experimentu 1.2 vyhodnocuje 5 různých veličin a přináší tak podrobné informace o zkoumaných metodách. Výsledky se nachází v této tabulce.

V úspěšnosti bylo nejlepší vzestupné seřazení podle reálné velikosti cesty ze startu do cíle. Dobrou úspěšnost mělo také přiřazování priority na základě odhadu vzdálenosti mezi startem a cílem pomocí manhattanské metriky a eukleidovské metriky vzestupně a také sestupné uskupení podle rozdílu počtu souvislých komponent se sekundárním řazením podle odhadu vzdálenosti. Nejlepší funkce se liší od nejhorší o 22 úspěšně sestavených cest (sestavovalo se celkem 900 cest), což je markantní rozdíl a lze vidět, jak je řazení agentů důležité.

Také stojí za povšimnutí rozdíly v časech metod. Zatímco odhad vzdálenosti startu a cíle trval pro všechny méně než milisekundu, tak řazení podle skutečné vzdálenosti zabere desítky sekund a sčítání souvislých komponent pak trvá déle než 14 minut. Dále je v tabulce zobrazena určitá souvislost mezi průměrnou délkou cesty a úspěšností nalezení cesty, čím nižší délka cest, tím je sestavování cest úspěšnější. Mohlo by to být proto, že čím kratší cesty se sestaví, tím je menší šance, že nějaká cesta zablokuje jinou. Hodnoty D jsou se od sebe příliš neliší a není tu vidět souvislost s ostatními veličinami.

Vzestupné pořadí mělo vždy lepší úspěšnost než sestupné. Je to nejspíš kvůli tomu, že při sestupném uspořádání se velké cesty stavějí na začátku a ty často zablokují průchod pro malé. Zatímco u menších cest je menší pravděpodobnost, že zablokují průchod jiné cestě.

Tabulka 4.2: Experiment 1.2

Metoda	Seřazení	10	12	14	16	18	20	celkem
euclid	↑	10/100	9/119	9/138	6/153	7/177	3/182	**44/869**
euclid	↓	9/99	10/120	8/138	7/155	5/174	3/182	**42/868**
manhatt	↑	10/100	10/120	8/138	7/154	7/177	4/189	**46/878**
manhatt	↓	9/99	10/120	8/137	6/153	4/174	3/183	**40/866**
path_len	↑	10/100	9/119	8/137	8/155	8/178	4/189	**47/878**
path_len	↓	9/99	10/120	8/137	7/155	5/172	3/183	**42/866**
spliting	↑	10/100	9/119	8/137	7/156	7/176	3/181	**44/869**
spliting	↓	10/100	9/119	8/137	7/156	6/175	3/180	**43/867**
split+euclid	↑	10/100	9/119	9/138	6/153	7/177	3/183	**43/870**
split+euclid	↓	9/99	10/120	8/138	7/155	5/173	3/182	**42/867**

Experiment 1.2

Experiment 1.2 se více zaměřuje na úspěšnost hledání cest, protože je to hlavní kritérium. Odehrává se na mapě shipwreck a zkoumá se zde, jak se mění úspěšnost pro různý počet agentů. Výsledky jsou obsaženy v této tabulce, kde sloupce označují různý počet agentů a řádky odlišné metody pro určování priority. Hodnoty v tabulce jsou v tvaru x/y, x znamená počet úspěšně vyřešených modelů a y počet úspěšně sestavených cest. V experimentu bylo vytvořeno 10 modelů pro každou testovanou hodnotu počtu agentů.

Před analýzou výsledků experimentu č.2 je dobré si uvědomit, že na mapě shipwreck se oproti mapě cave, která byla používána v předchozím pokusu, do vraku vede pouze několik úzkých průchodů. Kvůli tomu se reálná velikost cesty může výrazně lišit od odhadu velikosti, a tak metody řadící podle odhadů nemusí být tak úspěšné.

Z tabulky lze pozorovat, že sestupná řazení opět neměly dobré výsledky a u všech algoritmů dopadly hůř než řazení vzestupné. Nejvyšší úspěšnost měla také, jak u experimentu 1.1 metoda path_len, podobně úspěšný bylo uspořádání podle odhadu manhattanské vzdálenosti, čímž se vyvrátil předpoklad, že výsledky algoritmů řadících podle odhadu mohou být horší. Naopak horší výsledky v tomto pokusu měly funkce řadící podle změny počtu souvislých komponent, ale i odhad podle eukleidovské vzdálenosti.

Výsledky experimentu 1

Metody řadící podle změny počtu souvislých komponent dopadly v experimentech nejhůř. Počítání těchto hodnot trvalo velmi mnoho času a úspěšnost nebyla lepší než u ostatních metod. Uspořádání podle skutečné délky cesty přineslo nejlepší úspěšnost ze všech algoritmů. I přesto nebude používáno, protože zlepšení úspěšnosti není tak výrazné a sestavování cest pořád trvá velké množství času, i když ne tak moc jako počítání souvislých komponent.

Nejlépe dopadly seřazení podle odhadu vzdálenosti. Ty mělo jen o trochu menší úspěšnost než u řazení podle skutečné velikosti cesty a oproti nim

Tabulka 4.3: Výsledky experimentu č. 2

w	MSR	$PSR[\%]$	$t[s]$	L	MD
8	2/20	96.2	512	64.04	2 107 590
16	10/20	98.5	528	56.39	1 944 061
24	12/20	98.6	577	53.35	1 835 645
32	12/20	98.8	666	51.62	1 775 919
40	14/20	99.0	815	50.68	1 747 748
48	15/20	99.2	825	50.35	1 750 636
HCA*	15/20	99.3	532	48.66	1 695 004

byly mnohem časově efektivnější. Použití eukleidovské i manhattanské metriky mělo velice podobné výsledky, ale nakonec bylo rozhodnuto pro použití manhattanské vzdálenosti, která mělo úspěšnost sestavení cesty o trochu lepší. Bylo zjištěno, že vzestupné seřazení funguje lépe než sestupné u odhadů vzdálenosti.

Nadále v experimentech se bude pro uspořádání agentů používat **odhad vzdálenosti startovní a cílové pozice pomocí manhattanské metriky ve vzestupném pořadí**.

4.2.2 Experiment č.2 - velikost okna WHCA*

Ve druhém experimentu se testuje funkčnost algoritmu WHCA* na hledání disjunktních cest pro různé hodnoty velikosti okna w (více informací o algoritmu naleznete v sekci č. 37). Jako mapa pro tento pokus byla zvolena sea_bottom, pro kterou bylo vygenerováno 30 modelů, v každém z nich se hledají cesty pro 50 agentů. Jako prostor pro hledání cest byla zvolena voxelová mřížka, stejné velikosti okna které fungují na tomto prostoru budou fungovat pro oktantový strom. Jako poslední algoritmus do experimentu byl přidán HCA*, protože WHCA* s příliš vysokou hodnotu w má stejné vlastnosti jako HCA*. Obrazně řečeno je HCA* stejný jako WHCA* s $w = \infty$. Veličiny jsou podobné jako u experimentu 1, rozdílný je čas t, který zde označuje čas běhu algoritmu.

Výsledky experimentu jsou zaznamenány v tabulce č. 4.3. Lze vidět, že s rostoucím w se zvyšuje úspěšnost hledání cest, avšak HCA* skončil nejúspěšnější. To se však v tomto pokusu nebude řešit, protože srovnáním CA*, HCA* a WHCA* se zabývá experiment č.3. Rozdíl MSR mezi nejvíce a nejméně úspěšným modelem se liší o 65 %, proto na správné volbě w velmi záleží. Také lze znovu pozorovat trend, že čím je nižší průměrná délka cesty, tím je vyšší úspěšnost sestavení cesty, pravděpodobný důvod byl popsán v závěru 1. experimentu.

Naopak instance algoritmy s nižším w byly časově efektivnější. Také vzájemná vzdálenost cest je u nich větší, ale to je pravděpodobně způsobeno

Tabulka 4.4: Experiment č. 3, výsledky algoritmu CA*

mapa	$\|A\|$	MSR	PSR[%]	$t[s]$	L	MD
Sea_bottom	72	25/30	99.72	54	48.20	3 556 423
Cave	24	29/30	99.72	20	41.60	275 978
Cave	30	20/30	98.78	30	42.48	443 566
Cave	36	18/30	98.33	39	42.97	643 350
Shipwreck	18	20/30	97.04	82	45.46	177 346
Sewers	20	23/30	98.33	16	44.96	213 302
Průměr	/	22.50/30	98.65	40.17	44.28	884 994

Tabulka 4.5: Experiment č. 3, výsledky algoritmu HCA*

mapa	$\|A\|$	MSR	PSR[%]	$t[s]$	L	MD
Sea_bottom	72	25/30	99.72	1 154	48.20	3 556 436
Cave	24	29/30	99.72	85	41.46	275 623
Cave	30	21/30	98.89	113	42.46	445 436
Cave	36	19/30	98.52	155	42.90	646 001
Shipwreck	18	19/30	96.67	348	45.53	176 847
Sewers	20	21/30	98.00	63	45.02	212 414
Průměr	/	22.33/25	98.59	319.78	44.26	885 459

Tabulka 4.6: Experiment č. 3, výsledky algoritmu WHCA*, w=32

mapa	$\|A\|$	MSR	PSR[%]	$t[s]$	L	MD
Sea_bottom	72	21/30	99.49	1 365	51.36	3 775 037
Cave	24	28/30	99.58	81	43.67	289 597
Cave	30	22/30	99.00	112	45.07	472 747
Cave	36	16/30	98.43	148	45.77	687 385
Shipwreck	18	19/30	96.11	850	48.20	184 828
Sewers	20	13/30	95.67	106	51.87	234 056
Průměr	/	19.83/30	98.05	443.67	47.66	940 608

Tabulka 4.7: Experiment č. 3, výsledky algoritmu WHCA*, w=48

mapa	$\|A\|$	MSR	PSR[%]	$t[s]$	L	MD
Sea_bottom	72	22/30	99.58	2127	49.99	3 688 323
Cave	24	29/30	99.72	81	42.64	283 423
Cave	30	21/30	98.89	128	43.71	458 547
Cave	36	19/30	98.52	146	44.17	664 665
Shipwreck	18	21/30	97.04	1902	47.04	184 524
Sewers	20	20/30	97.50	99	46.53	217 269
Průměr	/	22.0/30	98.49	747.16	45.68	916 125

tím že cesty jsou delší a vedou tak přes nové vrcholy, které jsou vzdálené od ostatních cest. Závěrečné rozhodnutí na základě výsledku experimentu č.2 je, že budeme používat v testech WHCA* s velikostmi okna 32 a 48. WHCA* s $w = 48$ hledá cesty s nejvyšší úspěšností, ale jeho běh zabere mnoho času. WHCA* s $w = 32$ má pak trochu nižší úspěšnost nalezení cesty, ale za to je časově efektivnější.

4.2.3 Experiment č.3 - srovnání CA*, HCA* a WHCA*

Popis experimentu č.3

Experimenty č. 1 a 2 byly méně důležité a sloužily jako příprava pro tento hlavní experiment č.3. Zde dojde ke srovnání hlavních 3 algoritmů, které byly navrženy v praktické části práce. Jsou to CA*, HCA* a WHCA* pro disjunktní cesty. Byly testovány 2 instance WHCA*, s hodnotou $w = 32$ a $w = 48$ V tomto pokusu bude prostor pro všechny metody voxelová mřížka, testování oktantového stromu bude probíhat v experimentu č.4.

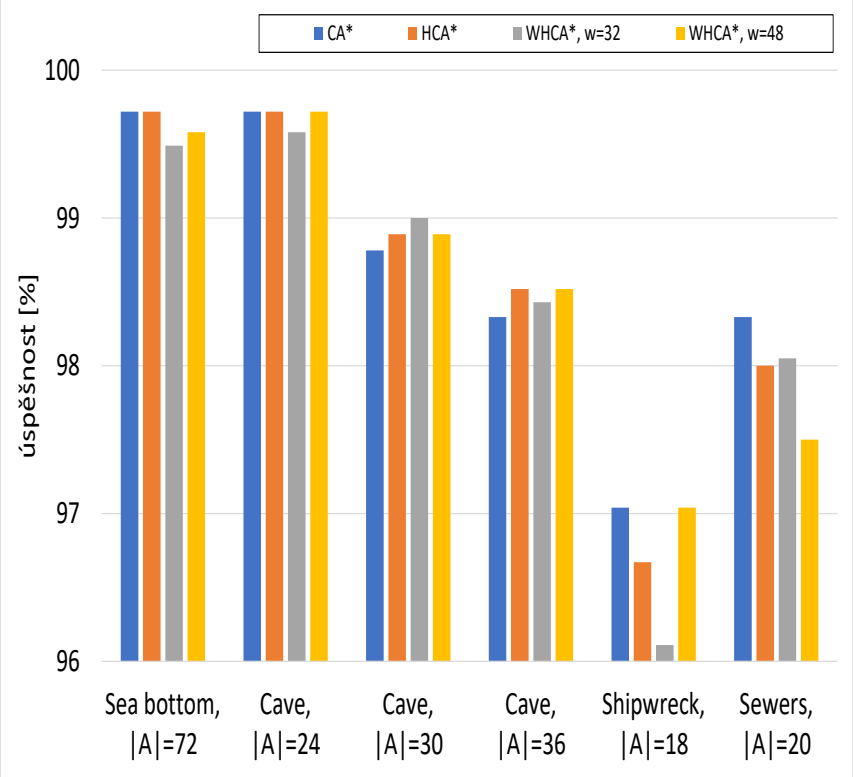
Při tomto pokusu byly použity všechny 4 mapy. Na mapě sea_bottom bylo hledáno 72 cest, na mapě shipwreck pouze 18 a na mapě sewers 20. Každému prostoru byl přiřazen počet agentů u něhož se očekává, že úspěšnost sestavení modelů bude vyšší než 0%, a zároveň nižší než 100%. Mapa cave byla do experimentu zařazena 3x, a to s 24, 30 a 36 agenty, aby bylo vidět, jak se liší vlastnosti metody při různých počtech agentů. Pro každou instanci mapy bylo vygenerováno 30 modelů, celkem byl tedy každý algoritmus testován na 180 modelech.

Výsledky experimentů najdete v tabulkách č. 7, 8, 9, 10, každý algoritmus má samostatnou tabulku. Je zde použita jedna nová veličina $|A|$, která značí velikost množiny s agenty, tedy počet agentů.

Výsledky experimentu č.3

Nejprve se bude věnovat pozornost úspěšnosti hledání cest. Nejvyšší MSR a PSR měl algoritmus CA* pro dijunktní cesty, jen o trochu nižší měly algoritmy HCA* a WHCA* s $w = 48$ pro disjunktní cesty. Naopak algoritmus WHCA* s $w = 32$ pro disjunktní cesty sestavoval cesty s velmi nízkou úspěšností. Metodám CA* a HCA*, kde agenti cesty sestavují postupně se dařilo na otevřených mapách, konkrétně sea_bottom, cave s nízkým $|A|$ a sewers, i když ta se mezi otevřené mapy neřadí. Naopak přístup, kde se sestavují cesty najednou po kouscích, který má algoritmus WHCA* byl úspěšný na mapách shipwreck a cave s vysokými hodnotami $|A|$, což jsou prostory s vysokou koncentrací překážek.

Nejkratší dobu běhu měl algoritmus CA* a metody využívající jako heuristiku vzdálenost v abstraktním prostoru byly výrazně časově náročnější. Největší rozdíl lze pozorovat v případech, kdy se cesta v abstraktním prostoru (svět bez ostatních agentů) výrazně liší od té skutečné, tedy na mapě



Obrázek 4.2: Graf zobrazují úspěšnost modelů v experimentu č.3

Sea_bottom, kde je obrovské množství agentů a dochází tak ke kolizím. Také na mapě shipwreck můžeme vidět velké rozdíly v t. Zde je to způsobeno tím, že do vraku vede pouze omezené množství průchodů, a po obsazení průchodu jiným agentem se cesta výrazně prodlouží. RRA* pak expanduje mnohem více vrcholů. Navíc se zde vrcholy expandují v hladinách kolem cíle a ve 3D prostoru trvá delší čas, než narazíme na hledaný uzel než v 2D prostoru. Navíc u problému hledání disjunktních cest se reálná cesta liší více od té v abstraktním prostoru než u hledání cest klasických. Kvůli těmto problému je heuristika využívají velikost cest v abstraktním prostoru časově neefektivní.

I zde je vidět, že čím nižší je průměrná velikost cest, tím je vyšší úspěšnost nalezení cesty. Za povšimnutí stojí vyšší L u cest sestavovaných algoritmem WHCA*. Je to kvůli tomu, že sestavená částečná cesta nemusí být nakonec ta nejkratší a kvůli tomu je pak celková délka cest vyšší než u algoritmů,

které sestavují cesty až po tom, co dojdou k cíli. Také MD je u WHCA* vyšší než u CA* a WHCA*, což je také způsobeno vyšší průměrnou velikostí cesty, protože cesty pak obsahují vrcholy, které jsou dal od ostatních cest, a tak D nakonec vzroste.

Závěr experimentu č.3

CA* byl ze všech nově navržených algoritmů pro hledání disjunktních cest nejlepší jak v úspěšnosti sestavení cesty, tak v časové efektivitě. Nejlepší výsledky měl na mapách sea_bottom a sewers. HCA* pro měl podobnou úspěšnost jako CA*, na mapě cave byly dokonce jeho výsledky lepší. Kvůli jeho heuristice však může být velmi časově náročný v prostorech, kde se velikost cesty v abstraktním prostoru bez agentů výrazně liší od cesty v prostoru skutečném.

Díky odlišnému přístupu k problému dokáže být algoritmus WHCA* užitečný. V některých případech je lepší totiž sestavovat cesty po kouskách najednou než postupně. I když v celkové úspěšnosti byl WHCA* nejhorší a jeho běh trval nejdéle ze tří algoritmů, tak na mapách shipwreck a cave s vysokým $|A|$, což jsou mapy s úzkými prostory, dokázal hledat cesty s nejvyšší úspěšností. Instance WHCA* s $w = 48$ měla mnohem lepší úspěšnost než varianta s $w = 32$, a to tak výrazně, že už instance s $w = 32$ dále v experimentech nebude používána. I když je časově efektivnější, propad v úspěšnosti je až moc velký a stále je pomalejší než CA* a HCA*.

4.2.4 Experiment č.4 - oktantový strom a voxelová mřížka

Popis experimentu č.4

V experimentu č. 4 se zjišťují rozdíly mezi 2 datovými strukturami, které se používají pro hledání cest v této práci. Jde o voxelovou mřížku a oktantový strom. Nejsou očekávány velké rozdíly ve veličinách úspěšnost sestavení cesty a průměrná délky cesty, protože algoritmy na obou strukturách fungují skoro stejně a odlišnosti mohou vznikat pouze kvůli nepřesnému určení vzdálenosti u oktantového stromu (viz sekce č. 3.1.3).

Naopak se očekávají rozdíly v času běhu mezi algoritmy na různých prostorech. Ten budou tentokrát vyjadřovat 2 veličiny: t a t_{oc}. Zatímco t měří dobu běhu celého algoritmu, hodnota t_{oc} je rovna veličině t od níž je odečten čas, který byl potřeba na konstrukci oktantového stromu. Je zde zařazena proto, že konstrukce oktantového stromu může být provedena kdykoli před hledáním cest. Navíc pokud se řeší více problémů ve stejném prostoru, je možné využít již zkonstruovaný oktantový strom. Ne vždy je potřeba sestavovat nový oktantový strom a tento případ vyjadřuje právě veličina t_{oc}.

Veličina MD nebyla v tomto experimentu měřena a větší prostor jí bude věnován v experimentu č.6. Není zde zařazena ani úspěšnost modelů, protože se neočekávají velké rozdíly mezi algoritmy fungujícími na voxelové mřížce a

Tabulka 4.8: Experiment 4.1, výsledky algoritmu CA* pro oktantový strom

mapa	$\|A\|$	CP	$t[s]$	$t_{oc}[s]$	L
Sea_bottom	72	720 / +0	125.2 / +111.7	123.7 / +109.9	47.4 / +0.5
Cave	40	385 / -1	68.1 / +54.5	58.4 / +44.8	42.6 / +0.9
Shipwreck	18	163 / +2	24.9 / -22.2	22.7 / -24.4	48.5 / +1.1
Shipwreck	28	247 / +0	41.5 / -28.5	39.4 / -30.6	46.2 / +0.8
Sewers	20	200 / +1	27.2 / +23	18.8 / +14.6	44.7 / +1.0

Tabulka 4.9: Experiment 4.1, výsledky algoritmu HCA* pro oktantový strom

mapa	$\|A\|$	CP	$t[s]$	$t_{oc}[s]$	L
Sea_bottom	72	720 / +2	683.2 / +261.1	681.6 / +259.5	47.6 / +0.7
Cave	40	393 / -1	119.9 / +56.1	109.8 / +46	42.9 / +1.3
Shipwreck	18	162 / +4	187.6 / +8.1	185.2 / +5.7	48.6 / +1.3
Shipwreck	28	247 / +2	281.6 / +23.9	279.3 / +21.6	46.0 / +0.5
Sewers	20	197 / -2	55.9 / +37.6	45.8 / +35.5	44.6 / +0.9

Tabulka 4.10: Experiment 4.1, výsledky algoritmu WHCA* s $w = 48$ pro oktantový strom

mapa	$\|A\|$	CP	$t[s]$	$t_{oc}[s]$	L
Sea_bottom	72	720 / +2	1478.5 / +904.1	1476.9 / + 902.5	47.5 / +0
Cave	40	392 / -2	342.3 / +280.3	333.9 / +271.9	43.0 / +0
Shipwreck	18	162 / +1	272.7 / -1726	270.7 / -1728	48.6 / -1
Shipwreck	28	247 / +2	405.7 / -1541	403.6 / -1543	46.1 / +0.9
Sewers	20	196 / -2	128.5 / +107.1	119.7 / +98.3	44.7 / -0.2

algoritmy fungující na oktantovém stromu a úspěšnost hledání cest je schopna tuto veličinu nahradit.

Experiment 4.1

V experimentu 4.1 došlo ke srovnání algoritmů pro oktantový strom a pro voxelovou mřížku. Pro pokus byly použity všechny 4 navržené mapy, s podobným počtem agentů jako v experimentu č.3, tentokrát byly zařazeny 2 instance mapy shipwreck s 18 a 28 agenty. Pro každou verzi mapy bylo vygenerováno 10 různých modelů.

Výsledky jsou zapsány v tabulkách č. 4.8, 4.9 a 4.10. Hodnoty jsou ve tvaru x/diff, kde x je výsledek pro daný algoritmus pracující na oktantovém stromě a hodnota diff vyjadřuje rozdíl mezi výsledkem algoritmu pracujícím na oktantovém stromu a výsledkem toho samého algoritmu pracujícím s voxelovou mřížkou.

Z výsledků lze vidět, že úspěšnost algoritmů pracující s oktantovým stromem se příliš neliší od verze pracující s voxelovou mřížkou, jak bylo předpokládáno. U CA* a u HCA* došlo na všech mapách k malému zvýšení L u algoritmu hledající cesty na oktantovém stromu. U WHCA* byla průměrná délka

cesty podobná u obou verzí. Tyto rozdíly jsou způsobeny nepřesným určování vzdálenosti při hledání cest na oktantovém stromu.

Největší odlišnosti byly u obou prostorů v délce běhu algoritmů. Konstrukce oktantového stromu trvala déle na mapách s více překážkami, tedy na sewers a shipwreck. Naopak na mapách sea_bottom a cave není mezi t a t_{oc} velký rozdíl. Nejdelší konstrukce trvala kolem 10 sekund, a to nehraje příliš velkou roli, nejspíš na větších mapách by se čas s konstrukcí stromu a čas bez konstrukce lišily více.

Na mapách sewers, cave a sea_bottom byla verze pracující s voxelovou mřížkou rychlejší, a to u všech 3 algoritmů. Naopak u mapy shipwreck měly u CA* a WHCA* menší čas běhu verze pracující s oktantovým stromem. Rozdíly byly v některých případech obrovské, u WHCA* na mapě shipwreck trvalo hledání cest na voxelové mřížce přibližně 7x déle než hledání na oktantovém stromu.

Metodám pro sestavování cest využívající oktantový strom se na mapě shipwreck daří proto, že cílové pozice agentů se generují pouze ve vraku lodi, což je vzhledem k velikosti mapy poměrně malý prostor. Na mapě pak kvůli tomu zůstane velké množství volného prostoru u sebe, a tak je možno pracovat při sestavování cest s uzly větších velikostí, a to šetří čas. Na ostatních mapách se startovní a cílové pozice mohou generovat ve velké oblasti. Kvůli tomu zde nejsou místa s velkou koncentrací volného prostoru, a tak nelze využít plný potenciál oktantového stromu.

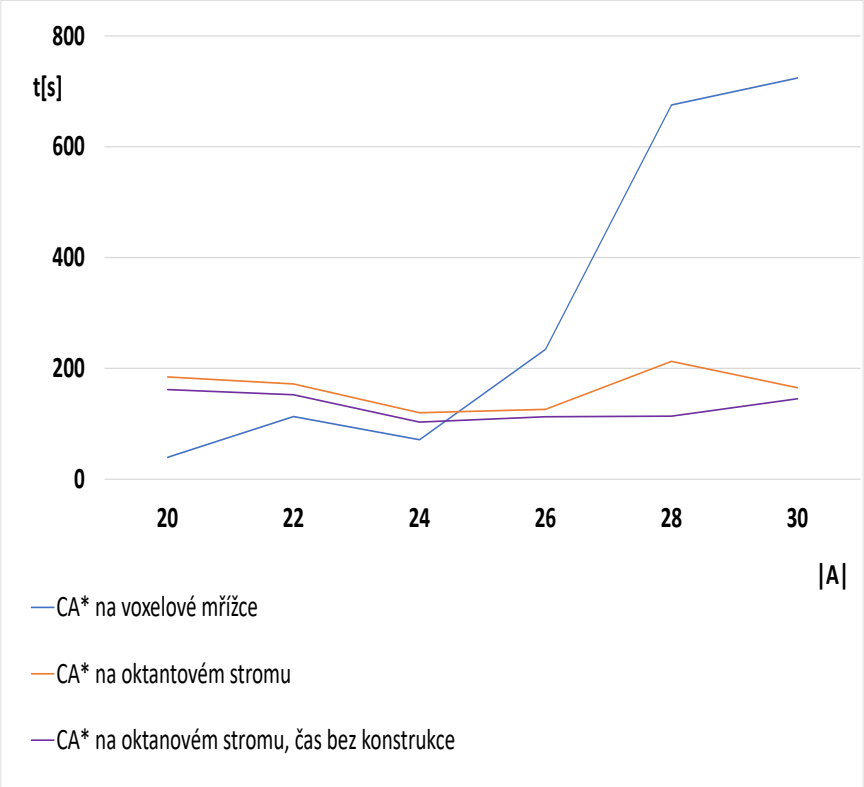
V experimentu 4.1 nedopadly metody používající oktantový strom příliš úspěšně a metody využívající voxelovou mřížkou předčily pouze v ojedinělých případech. Proto byl proveden ještě experiment 4.2, který se odehrával v prostoru, kde by měly být výhody oktantového stromu lépe využitelné.

Experiment 4.2

Experiment 4.2 probíhal pouze na mapě sea_bottom, ale v odlišné verzi, která má rozměry $64 \times 64 \times 64$. Startovní pozice byly generovány pouze v jednom z horních rohů a cílové pozice pak pouze v rohu opačném. Díky tomu vzniknou na mapě oblasti s větším množstvím volného prostoru, které se v oktantovém stromu spojí do uzlů s větší velikostí, a tak by měl být ušetřen čas.

Tentokrát byl testován pouze algoritmus CA*. V experimentu byly postupně hledány cesty pro 20, 22, 24, 26, 28 a 30 agentů. Pro každou z těchto hodnot bylo vygenerováno 10 modelů. Tentokrát byl měřen pouze čas sestavení cest, protože ostatní veličiny si byly v experimentu 4.1 u obou prostorů velmi podobné.

Výsledky experimentu nejsou tentokrát zapsány v tabulce, ale zobrazeny na grafu č. 4.3. Lze vidět, že pro nízké hodnoty $|A|$ je verze algoritmu pracující s voxelovou mřížkou časově efektivnější než verze využívající oktantový strom. Pro vyšší hodnoty $|A|$ se to ale obrátí. Zatímco čas běhu CA* s voxe-



Obrázek 4.3: Výsledky experimentu 4.2

lovou mřížkou rychle roste se zvyšujícím počtem agentů, tak čas běhu CA* s oktantovým stromem zůstává pro různý počet agentů podobný.

Důvod je ten, že pro vyšší počet agentů klesá úspěšnost nalezení cesty. Aby se algoritmus CA* ujistil, že neexistuje cesta do cíle, musí expandovat všechny vrcholy dostupné ze startu. To je na mapě o velikosti $64 \times 64 \times 64$ velmi časově náročné. Expandování všech vrcholů v oktantovém stromu je mnohem efektivnější, protože se mohou expandovat uzly větší velikosti a ty se na mapě, která obsahuje oblasti s vysokou koncentrací volného prostoru, hojně vyskytují.

Při sestavování disjunktních cest není oktantový strom tak silný nástroj, jako u sestavování cest klasických. Je to kvůli tomu, že po sestavení každé cesty jsou do prostoru vloženy její vrcholy jako překážky. Vkládání překážek jde u voxelové mřížky snadno, protože tím nejsou ovlivněny žádné jiné vrcholy.

Tabulka 4.11: Výsledky experimentu 5.1

p	MSR[%]	CP	L
0.2	70 / 45	354 / 457	46.74 / 44.12
0.4	75 / 50	354 / 456	46.81 / 43.95
0.6	85 / 75	355 / 469	46.91 / 44.75
0.8	80 / 70	353 / 465	46.91 / 44.80
1.0	70 / 55	351 / 465	47.01 / 45.46
1.2	75 / 65	351 / 467	47.18 / 45.88
1.4	75 / 70	352 / 468	47.37 / 46.31
1.6	75 / 70	352 / 467	47.42 / 46.53
1.8	80 / 65	353 / 465	47.61 / 46.66
2.0	65 / 55	350 / 464	48.08 / 47.28

Ale naopak v oktantovém stromu může vložení každé překážky změnit podobu zbytku prostoru, a je to tedy časově náročnější. Experimenty 4.1 a 4.2 však ukázaly, že i přes tento problém může použití oktantového stromu výrazně zrychlit běh algoritmu.

4.2.5 Experiment č.5 - obstacle CA*

V 5. experimentu se budou zkoumat vlastnosti algoritmu obstacle CA*. Očekává se od něj, že na mapách, kde se vyskytují úzké průchody a vyšší koncentrací překážek, bude mít lepší úspěšnost sestavování cest. To z našich map splňují shipwreck a sewers. Při tomtu pokusu se měří pouze 3 veličiny, u kterých se očekávají změny při rozdílných hodnotách p. Jsou to úspěšnost vyřešení modelu, úspěšnost sestavení cesty a celková délka cest.

Tento experiment je rozdělen na 2 části. V té první bude zkoumáno, jak se liší vlastnosti algoritmu na základě volby parametru p a podle výsledků se určí hodnota p, která bude používána v druhé části. V té se pak porovná obstacle CA* s metodami pro hledání disjunktních cest, které byly vytvořeny v této práci.

Experiment 5.1

Cílem tohoto experimentu je najít hodnotu parametru p, se kterou algoritmus obstacle A* bude mít vysokou úspěšnost hledání cest na mapách používaných v experimentu 5.1, tedy na shipwreck a sewers. Při tomto pokusu byly hledány cesty pro 18 agentů na mapě shipwreck a pro 24 agentů na sewers. Pro každý prostor bylo vygenerováno 20 různých modelů. Bylo vyzkoušeno 10 různých hodnot p od 0.2 do 2.

Výsledky jsou zobrazené v tabulce č. 4.11. Hodnoty v ní jsou ve tvaru x/y, kde x je výsledek v dané kategorii na mapě shipwreck s $|A| = 18$ a y je potom výsledek na mapě sewers, $|A| = 24$. Z tabulky lze vidět, že s rostoucím p roste

Tabulka 4.12: Výsledky experimentu 5.2 na mapě shipwreck, $|A| = 18$

Algoritmus	MSR	CP	$t[s]$	L
Obstacle CA*	24/30	531	257.92	46.66
CA*	19/30	519	93.00	46.30
HCA*	21/30	526	372.39	46.57
WHCA, $w = 48$	19/30	524	2469.06	48.12

Tabulka 4.13: Výsledky experimentu 5.2 na mapě sewers, $|A| = 24$

Algoritmus	MSR	CP	$t[s]$	L
Obstacle CA*	20/30	705	247.35	45.00
CA*	14/30	685	25.30	43.88
HCA*	17/30	694	77.36	44.06
WHCA, $w = 48$	17/30	689	101.63	45.52

také průměrná délka cesty. Z předchozích experimentů vyplynulo, že čím je L menší, tím větší bývá úspěšnost hledání cest.

Pro menší hodnoty p má obstacle CA* podobné výsledky jako klasický CA*, protože CA* je to samé jako obstacle CA* s $p = 0$. Jak se postupně L zvyšuje, tak se řešení více odlišují od řešení CA*. To způsobí zvýšení úspěšnosti nalezení cest, protože cesty vytvořené pomocí obstacle CA* se vyhýbají místům s vysokou koncentrací překážek nebo případně přes ně procházejí co nejrychleji, a proto je menší šance na zablokování. Pro vysoké hodnoty p ale celková délka cest vzroste natolik, že vyšší šance na zablokování cest kvůli vysokému L přebije výhodu obstacle CA*, tedy správné procházení skrz „bottlenecky“ a celková úspěšnost nalezení cesty kvůli tomu klesá a může klesnout i pod úspěšnost klasického CA*.

Jak pro mapu shipwreck, tak pro mapu sewers měl algoritmus nejlepší výsledky s $p = 0.6$ a tato hodnota bude použita experimentu č. 5.2.

Experiment 5.2

V tomto experimentu budou srovnány odlišnosti mezi algoritmem obstacle A* a ostatními dosud používanými algoritmy, tedy CA*, HCA* a WHCA*. Budou se zkoumat všechny veličiny, které se měřily v předchozím testu č. 5.1. Navíc bude opět přidána veličina t, protože je potřeba zjistit, o kolik déle bude trvat sestavování cest pomocí obstacle CA* než hledání pomocí klasického CA*.

Počet agentů je stejný jako u předchozího experimentu, tedy na mapě shipwreck 18 a na sewers pak 24. Pro každou mapu bylo vygenerováno 30 modelů, které algoritmy řešily. Výsledky jsou zobrazeny v tabulkách č. 4.12 a 4.13.

Výsledky experimentů ukazují, že obstacle CA* dokázal hledat disjunktní

Tabulka 4.14: Výsledky experimentu č.6

algoritmus	p	MSR	CP	$t[s]$	L	MD
CA*	/	8/10	718	13.9	46.43	3 295 563
HCA*	/	8/10	718	358.2	46.43	3 295 771
WHCA*, w=48	/	9/10	719	440.7	48.13	3 440 955
MDCA*	1/92	10/10	720	852.3	46.39	3 247 621
MDCA*	1	10/10	720	860.7	46.43	3 253 537
MDCA*	3	10/10	720	958.2	48.64	3 402 663
MDCA*	10	8/10	718	1185.5	52.65	3 725 928
MDCA*	92	9/10	719	1511.8	61.53	4 462 357
MDWHCA*, w=48	1/92	9/10	719	821.5	48.06	3 433 567
MDWHCA*, w=48	1	9/10	719	1118.0	49.62	3 564 264
MDWHCA*, w=48	3	5/10	712	4440.9	68.51	4 893 769
MDWHCA*, w=48	10	0/10	650	3302.9	119.23	7 302 205
MDWHCA*, w=48	92	0/10	453	1538.12	101.166	2 931 589

cesty s vyšší úspěšností než všechny ostatní algoritmy na obou zvolených mapách. Nebyl tak časově efektivní jako CA*, ale sestavování cest pomocí něj trvalo vždy méně času než pomocí WHCA* a na mapě shipwreck byl dokonce obstacle CA* rychlejší než HCA*. Z experimentu vyplývá, že díky použití obstacle CA* na mapách s „bottlenecky“ je schopno dosáhnout vyšší úspěšnosti sestavování cest než ostatní algoritmy pro hledání disjunktních cest navržené v této práci.

4.2.6 Experiment č.6 - vzájemná vzdálenost cest

Experimentu č.6 se zaměřuje na veličinu vzájemná vzdálenost cest od sebe, která je přesně definována v sekči č. 2.1.6. V této práci byly navrženy 2 nové algoritmy, které se specializují na optimalizaci této veličiny. Jsou to MDCA* založený na CA* a MDWHCA* založený na WHCA*. Očekává se, že obě tyto metody zvýší vzájemnou vzdálenost cest od sebe, ale tím se může zvýšit velikost cest, a tak snížit úspěšnost hledání. Při experimentu budou zkoumány ty samé veličiny, jako v experimentu č.3.

Výsledky experimentu 6 jsou k nalezení v tabulce č. 4.14. Dolní mezní hodnota p pro tento prostor je 1/92 a horní je rovna 92. MDA* s p pod dolní hodnotou by měl najít stejně dlouhou cestu jako A*, ale v pokusu se L dokonce zmenšilo. To je způsobeno tím, že v problému disjunktních cest může každá sestavená cesta ovlivnit ostatní, a tak i když první cestu je u MDCA* a CA* stejně dlouhá, tak další se mohou v délce lišit.

S rostoucím p se postupně MD zvyšuje, čímž se zvyšuje i průměrná délka cesty. U MDCA* to nemá velké dopady na úspěšnost nalezení cesty, ale u algoritmu MDWHCA* u vysokých hodnot p lze pozorovat markantní propad úspěšnosti. To je proto, že u MDCA* se agent pohybuje až potom, co je cesta dokončena, ale u MDWHCA* se agent pohybuje postupně po částečně sestavených cestách. Navíc se kvůli vysokému p snaží ostatním cestám vyhýbat,

kvůli tomu tvoří čím dál delší a delší cestu, která nakonec zablokuje jak ostatní agenty, tak i agenta, který tuto cestu vytváří.

Pokud je potřeba co nejvíce zvýšit vzájemnou vzdálenost cest, vhodný na použití se jeví algoritmus MDCA*. S rostoucím parametrem p u tohoto algoritmu roste i D, a neobjevuje se výrazné snížení úspěšnosti. MDWHCA* neměl v experimentu příliš dobré výsledky. Pro vyšší hodnoty p enormně klesá úspěšnost sestavení cesty a jeho běh trvá déle než běh MDCA*. Pro nižší hodnoty p však může pořád vyprodukovat dobré výsledky.

Závěr

V teoretické části byl přesně definován problém a související pojmy. Při rešerši bylo objeveno několik prací s podobným tématem, ale jejich řešení byly shledány nevhodné pro náš problém hledání disjunktních cest v 3D prostředí s překážkami. Proto bylo rozhodnuto, že v praktické části budou navrženy nové metody na základě algoritmů, které mají dobré výsledky pro hledání nedisjunktních cest.

Tyto metody byly pouze mírně upraveny, aby fungovaly pro řešení našeho problému a úspěšně implementovány. Navíc byly ještě některé z metod zprovozněny na datových strukturách, které se specializují na ztvárnění trojrozměrného prostoru, konkrétně se jedná o voxelovou mřížku a oktantový storm. Dále bylo navrhnuto několik nových variant těchto algoritmů, které se hodí pro speciální případy, jako optimalizace některých kritérií či speciální varianty prostoru.

V experimentech pak byly ověřeny vlastnosti všech navržených algoritmů. Hlavní rozdíly mezi nimi se objevovaly ve veličinách úspěšnost nalezení cesty, čas běhu algoritmu a v délce cest. Neobjevila se zde žádná metoda, která by neměla žádné uplatnění a každý algoritmus měl dobré výsledky v některé ze situací. Dále byly ověřeny výhody obou použitých datových struktur a bylo popsáno, v jakých případech je vhodné jakou použít.

Zde jsou vypsány návrhy, jak by se v práci dalo pokračovat a rozšířit ji a na které již nebyl prostor:

- Zkusit využít algoritmy THETA* a Lazy THETA*, které se specializují na hledání cest v mřížkové struktuře a mohly by tedy dobře fungovat pro voxelovou mřížku a možná i pro oktantový strom.

- Zprovoznit algoritmy tak, aby se mohl agent z každé pozice pohybovat do všech 27 směrů. Nyní je to možné pouze do 6 základních, tedy tzv. „face“ spojitost.

- Najít pokročilejší metody na určování priority jednotlivých agentů, protože zde tomu byl věnován pouze malý prostor.

Literatura

[1] MID CAYMAN RISE: Seirios photograped from Little Hercules while still secure to the shippickup cable that can be seen trending vertically to its top. In: NOAA PHOTO LIBRARY: National Oceanic and Athmospheric Administration [online]. 2019 [cit. 2021-03-28]. Dostupné z: https://photolib.noaa.gov/Collections/Voyage/Ocean-Exploration/Modern-Expeditions/OER/Mid-Cayman-Rise/emodule/1418/eitem/94417.

[2] MATOUŠEK, J. *Kapitoly z diskrétní matematiky* . Vyd. 2., opr., (V nakl. Karolinum 1.). Praha: Karolinum, 2000. ISBN 80-246-0084-6.

[3] AVGUSTINOVICH, S., a FON-DER-FLAASS D. Cartesian Products of Graphs and Metric Spaces. *European Journal of Combinatorics* [online]. 2000, 21(7), 847-851. ISSN 0195-6698. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0195669800904013.

[4] ČERNÝ, M. *Generování terénu s využitím voxelové struktury* [online]. Brno, 2016. Dostupné z: https://is.muni.cz/th/di4s7/. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Jiří Chmelík.

[5] CHEN, Y., SHE, J., LI, X., ZHANG, S., TAN, J. Accurate and Efficient Calculation of Three-Dimensional Cost Distance. *ISPRS International Journal of Geo-Information*. 2020, 9. 353. 10.3390/ijgi9060353.

[6] KAWARABAYASHI, K., KOBAYASHI, Y., REED, B. The disjoint paths problem in quadratic time, *Journal of Combinatorial Theory*, 2012, Series B, Volume 102, Issue 2, Pages 424-435, ISSN 0095-8956. Dostupné z: https://doi.org/10.1016/j.jctb.2011.07.004.

[7] MEAGHER, D. Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. 1980.

[8] MALCOLM, R. Graph Decomposition for Efficient Multi-Robot Path Planning. *IJCAI International Joint Conference on Artificial Intelligence*. 2007, 2003-2008.

[9] KORNHAUSER, D., MILLER, G. L., SPIKARIS, P. Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications. *FOCS* 1984, 241-250

[10] The University of British Columbia Computer Science: UBC Computer Science 322: Introduction to Artificial Intelligence [online]. Alan K. Mackworth, 2012. Dostupné z: `https://www.cs.ubc.ca/~mack/CS322/lectures/2-Search2.pdf`

[11] ZUKAL, M. *Multi-agentní hledání cest pro připojené roboty*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

[12] GEWALI, L., MAZZELLA, D., SELVARAJ, H. Constrained Disjoint Paths in Geometric Networks. *International Journal of Computational Intelligence and Applications*. 2009, 8, 141-154. 10.1142/S1469026809002552. 56. Dostupné z: `https://doi.org/10.1016/j.jctb.2011.07.004`

[13] HART, P., NILSSON, N., RAPHAEL, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107. Dostupné z: `https://doi.org/10.1109/tssc.1968.300136`

[14] DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959, 1(1), 269–271.

[15] INDIANA UNIVERSITY SOUTH BEND: C463 / B551 Artificial Intelligence [online]. danav@cs.iusb.edu, 2008. Dostupné z: `https://www.cs.iusb.edu/~danav/teach/c463/y5_informed_search.html`

[16] CRAW, S. Manhattan Distance. In: Sammut C., Webb G.I. (eds) *Encyclopedia of Machine Learning and Data Mining*. Springer, Boston, MA. 2017. Dostupné z: `https://doi.org/10.1007/978-1-4899-7687-1_511`

[17] Advanced Data Structures: BFS and DFS Time Complexity. In: Youtube [online]. Google, 2021, 2020. Dostupné z: `https://www.youtube.com/watch?v=ZpOy0-QBVPM`

[18] SILVER, D. Cooperative Pathfinding.. Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference, 2005, AIIDE 2005. 117-122.

[19] SAMET H. Neighbor finding in images represented by octrees, *Computer Vision, Graphics, and Image Processing*, Volume 46, Issue 3, 1989, Pages 367-386, ISSN 0734-189X. Dostupné z: https://www.sciencedirect.com/science/article/pii/0734189X89900388

[20] Advanced Octrees 4: finding neighbor nodes. THE INFINITE LOOP [online]. David, 2013, 2 December 2017. Dostupné z: https://geidav.wordpress.com/2017/12/02/advanced-octrees-4-finding-neighbor-nodes/

[21] DOMBEK, D., SCHOLTZOVÁ J. Binární relace [přednáška]. Kurz *Základy diskrétní matematiky, BI-ZDM*. In: [online]. 2018. Dostupné z: https://courses.fit.cvut.cz/BI-ZDM/lectures/ZDM-P03-relace-handout.pdf.

[22] O'NEILL B. Chapter 2 - Frame Fields, Editor(s): Barrett O'Neill, *Elementary Differential Geometry* (Second Edition), Academic Press, 2006, Pages 43-99, ISBN 9780120887354. Dostupné z: https://doi.org/10.1016/B978-0-12-088735-4.50006-7.

[23] HARRIS, C.R., MILLMAN, K.J., van der WALT, S.J. et al. Array programming with NumPy. *Nature* 585, 357–362 (2020). Dostupné z: https://doi.org/10.1038/s41586-020-2649-2.

[24] EPHTRACY. MagicaVoxel, verze 0.99.6 [software]. 1. července 2020 [cit. 2021-5-4]. Dostupné z: https://github.com/ephtracy/ephtracy.github.io/releases/tag/0.99.6.

[25] GROMGULL. py-vox-io, verze 0.1 [software]. 10. prosince 2017 [cit. 2021-5-4]. Dostupné z: https://pypi.org/project/py-vox-io/.

[26] ZHOU Q. and PARK J. and KOLTUN V. Open3D, Modern Library for 3D Data Processing. ArXiv:1801.09847. 2018. Dostupné z: http://www.open3d.org/.

Příloha **A**

Seznam použitých zkratek

BFS Breadth first search

CA* Cooperative A*

CP Complete paths

HCA* Hierarchical cooperative A*

MAPF Multi agent pathfinding

max p_dist diff $\arg\max_{x,y \in V} p\_dist(x) - p\_dist(y)$

MDCA* Mutual distances cooperative A*

min h_old diff $\arg\min_{x,y \in V} |h\_old(x) - h\_old(y)|$

MSR Model sucess rate

p_dist path_distance

PSR Path sucess rate

WHCA* Windowed hierarchical cooperative A*

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `installation.pdf` ..... návod na zprovoznění implementace
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS