FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Dronový vzdušný displej
Student:	Bc. Ondřej Marek
Vedoucí:	doc. RNDr. Pavel Surynek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Cílem práce je vytvoření algoritmických podkladů pro tzv. dronový displej, kde jednotlivé pixely jsou tvořeny barevně světélkujícími drony létajícími v prostoru. Pixely tedy mohou měnit nejen barvu, ale i polohu v prostoru. Hlavní otázkou je plánování pohybu a barevného projevu pixelových dronů s ohledem na uživatelem zvolená kritéria a požadovaný vizuální efekt. Úkoly pro uchazeče jsou následující:

1. Prozkoumat existující algoritmy pro multi-agentní hledání cest (MAPF), které umožňují integrovat různá kritéria na kvalitu výsledného plánu.

2. Adaptovat existující techniky MAPF pro použití v rámci dronového displeje a navrhnout vhodnou synchronizaci s barevným výstupem podle uživatelských požadavků (zobrazení tvaru a jeho změn)

3. Algoritmický návrh implementovat formou softwarového prototypu a otestovat syntetickými simulacemi, případně na skutečných dronech pro relevantní vizualizační scénáře, například vizualizaci barevných nápisů či vystoupení ve stylu „drone art“.

–

[1] Wataru Yamada, Kazuhiro Yamada, Hiroyuki Manabe, Daizo Ikeda: iSphere: Self-Luminous Spherical Drone Display. UIST 2017: 635-643
[2] Pavel Surynek: A novel approach to path planning for multiple robots in bi-connected graphs. ICRA 2009: 3613-3619
[3] Wolfgang Hönig, James A. Preiss, T. K. Satish Kumar, Gaurav S. Sukhatme, Nora Ayanian: Trajectory Planning for Quadrotor Swarms. IEEE Trans. Robotics 34(4): 856-869 (2018)
[4] James A. Preiss, Wolfgang Hönig, Gaurav S. Sukhatme, Nora Ayanian: Crazyswarm: A large nano-quadcopter swarm. ICRA 2017: 3299-3304

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 26. ledna 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Dronový vzdušný displej

Bc. Ondřej Marek

Katedra teoretické informatiky
Vedoucí práce: doc. RNDr. Pavel Surynek, Ph.D.

19. září 2021

Poděkování

Děkuji svému vedoucímu doc. RNDr. Pavlu Surynkovi, Ph.D. za pomoc při psaní této práce. Své kamarádce Pavle Ďuranové, děkuji za vytvoření krásné ukázky do mého vizualizačního programu. Mnohokrát děkuji za morální podporu své ženě Elišce Markové.

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 19. září 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Marek, Ondřej. *Dronový vzdušný displej*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021. Dostupný také z WWW: ⟨`https://gitlab.fit.cvut.cz/marekon9/drone-aerial-display`⟩.

Abstrakt

Tato práce se zabývá vytvořením algoritmických podkladů pro tzv. dronový displej, kde jednotlivé pixely jsou tvořeny světélkujícími drony. Tyto drony mohou měnit jak barvu, tak polohu v prostoru. Cílem této práce je plánování pohybu dronů s ohledem na uživatelsky zvolená kritéria a požadovaný vizuální efekt.

K řešení problému byl přizpůsoben známý algoritmus CBS (Conflict Based Search) problému multi-agentního hledání cest. Část algoritmu, která řeší nalezení konfliktů, využívá toho, že prostor displeje je rozdělen na stejně velké krychle. Díky tomu algoritmus může zkontrolovat, zda různé drony svým objemem nevstoupily do stejné části displeje ve stejný časový okamžik. K hledání cest se využívá velmi upravený algoritmus A*. Hustotu propojení pixelů předepisuje zobecněné 2^k sousedství do prostoru. Drony se pohybují mezi pixely buď přímo nebo po trajektori definované pomocí interpolačních křivek v prostoru. Algoritmus také bere v úvahu maximální rychlost a akceleraci dronů i jejich momentum.

Výsledkem této práce je nový algoritmus zvaný CSPT_CBS (Continuous Spacetime Conflict Based Search) a simulační program, ve kterém je možné navolit umístění a barvy pixelů a následně shlédnout vizualizaci pohybu dronů.

Klíčová slova dronový vzdušný displej, AED, 3D, multi-agentní hledání cest, MAPF, 2^k sousedství, zobecněné 2^k sousedství v prostoru, CBS, CSPT_CBS, hledání pomocí konfliktů ve spojitém čase a prostoru, A*, interpolační křivky

Abstract

This thesis deals with the creation of an algorithm for so called drone display where individual pixels are created by luminous drones. The drones can change their color as well as position in space. The goal of this thesis is a planning of the drones movement taking into account user defined criteria and required visual effect.

A well known algorithm called CBS (Conflict Based Search) that solves a problem of multi-agent path finding was adjusted for this cause. The conflict search part of the algorithm utilizes the fact that the display is split into cubes. Thus the algorithm can check if various drones occupy the same cube area at the same time. An adapted A* algorithm is used for the path finding part of the CBS. Generalized 2^k neighborhood in space describes a connectivity of neighboring pixels. The drones move between pixels utilizing either straight paths or trajectories defined by spacial interpolating curves. The algorithm also takes into account drones maximum speed, their acceleration and their momentum.

A new algorithm called CSPT_CBS (Continuous Spacetime Conflict Based Search) and a simulating program are conceived as the results of this thesis. In the simulating program one can choose placement and colors of individual pixels and then see a visualisation of the drones movement.

Keywords drone aerial display, AED, 3D, multi-agent path finding, MAPF, 2^k neighborhood, generalized 2^k neighborhood in space, CBS, CSPT_CBS, Continuous Spacetime Conflict Based Search, A*, interpolating curves

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Displeje jsou v dnešní době všude kolem nás. Jsou zaintegrované v různých přístrojích jako mobilní telefony, přenosné počítače, monitory, chytré hodinky, digitální čtečky knih, bilboardy a další. Dokonce i tuto práci si s vysokou pravěpodobností čtete na jednom z těchto přístrojů. Všechny tyto obvyklé přístroje mají ale společnou jednu vlastnost. Tou vlastností mám na mysli, že jejich obrazovka prezentuje informace na ploše. Oproti tomu tato práce si dává za cíl vyzkoumat, jak rozšířit obrazovou prezentaci do prostoru. Jedním z možných řešení, kterým se tato práce zabývá, je implementace pomocí většího množství světélkujících dronů. Hlavním cílem této práce je přijít s postupem, který zaručí plynulé přeskupení dronů bez srážek mezi jednotlivými sekvencemi umístění pixelů.

V první části literární rešerše zkoumáme, jaké podobné projekty již existují a jaké využívají postupy. V druhé části literární rešerše rozebíráme již existující algoritmy a teoretické konstrukce, které se týkají dané problematiky.

Na základě rešerše formulujeme matematickou verzi problému dronového displeje. Pro tento problém vzápětí definujeme algoritmus, který daný problém řeší. Následuje experimentální část, ve které vyhodnotíme výkon našeho algoritmu.

Na samotném konci této práce se zabýváme tím, jaká je struktura a jak funguje vizualizačního programu, který jsme vytvořili jako součást této práce. V této části představíme ukázku „drone art", jednoho z možných využití dronového displeje.

Kapitola 1

Cíl práce

Cílem práce je vytvoření softwarového prototypu, který demonstruje proveditelnost dronového displeje.

Dílčí cíle jsou následující:

1. Vytvořit matematickou definici popisující problém dronového displeje.
2. K tomuto problému vymyslet a popsat algoritmus s využitím adaptace existujících algoritmů multi-agentního hledání cest.
3. Implementovat nový algoritmus v jazyce C++ a integrovat jej v rámci vizualizačního programu, který zobrazí pohyb dronů v prostoru.
4. Změřit výkonnost algoritmu.

Kapitola 2

Podobné koncepty

Na úvod práce představíme v této kapitole několik obdobných konceptů.

2.1 iSphere - Světelný sférický dronový displej

Tato práce [1] se zabývá možností vytvoření sférického displeje za pomoci dronu. Základní architektura (viz obr. 2.1) se skládá ze tří částí:

1. Dron, který dodává schopnost létání displeji.
2. POV displej, který vytváří sférický obraz.
3. Ochranný obal, který vytváří bariéru, aby displej nebo vrtule nemohly narazit do věcí nebo do lidí.

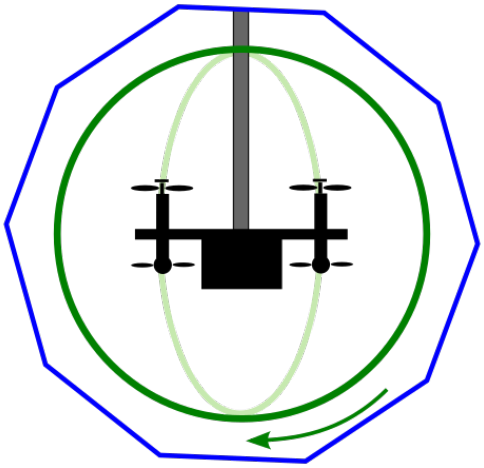
POV displej se skládá z několika rotujících LED pásek a k vytvoření obrazu využívá nedokonalosti vnímání lidského oka. Tato nedokonalost spočívá v tom, že oční čočka zachytává na krátký okamžik příchozí světlo. Pokud LED pásky rotují dostatečně rychle, vytváří to iluzi spojitého obrazu.

Autoři této práce diskutují o možnostech využití tohoto displeje a vidí potenciál využití při záchranných operacích, pro zobrazování informací většímu množství lidí, anebo pro videohovory. Svoji teoretickou práci zakončují vytvořením funkčního prototypu.

2.2 BitDrones interaktivní levitující drony

Další práce [2] se zabývá vytvořením zcela nového způsobu interakce s počítačem, kde místo standartní myši nebo klávesnice autoři používají sadu létajících dronů s cílem ponořit uživatele do prostorového rozhraní.

Tento přístup zvalidovali vytvořením prototypu, ve kterém jde využít až 12 dronů v prostoru $4 \times 4 \times 3$ metrů. Drony se ovládají buď jednoručně (například



Obrázek 2.1: Hrubý náčrt iSphere architektury. Dron veprostřed je spojen s LED páskami (zeleně), které se točí ve směru šipky. Celá konstrukce je vsazena do ochranného obalu vyznačeného modrou barvou.

dotekem, táhnutím, házením a dalšími způsoby), obouručně (například táhnutím, odtažením dvou dronů od sebe) nebo pomocí gest (například následování uživatele dronem). Tento způsob ovládání lze využít například při modelování v 3D kanvasu.

2.3 Crazyswarm

Autoři projektu Crazyswarm [3] vyvinuli architekturu pro provoz většího množství mini-kvadrakoptér ve vnitřních prostorách. Podle jejich přístupu takto dokáží operovat až s 49 drony Crazyflie 2.0 [4].

Celá architektura se skládá z několika částí:

1. Základní stanice - počítač se serverem.
2. Sledovací systém VICON - předává své výstupy do základní stanice.
3. Rádiový vysílač - umožňuje broadcast příkazů od základní stanice směrem k dronům.
4. Kvadrakoptéry - využívají svůj výpočetní výkon k navigaci v prostoru.

Systém funguje díky těmto krokům. Nejprve základní stanice vypočítá trajektorie všem kvadrakoptérám. Trasa se plánuje pro úseky spojené po částech polynomiálními křivkami s alespoň 4. spojitou derivací.

Kvadrakoptéry jsou pečlivě umístěny na předem daná místa na zemi a před vzletem se základní stanice sesynchronizuje se všemi kvadrakoptérami. Následně vydá hromadný příkaz ke vzletu. Během letu základní stanice přeposílá jednotlivým kvadrakoptérám jejich pozice. Kvadrakoptéry těchto informací využívají a samy si vypočítávají korekce trasy.

Pro účely tohoto projektu byl navržen nový komunikační protokol, který je tolerantní k výpadkům paketů (využívá se toho, že celý plán je uložen v paměti všech kvadrakoptér) a je idempotentní vůči duplikacím paketů.

Byly provedeny experimenty, které prověřili funkčnost a robustnost tohoto systému v prostoru $6 \times 6 \times 3$ metrů s 24 sledovacími kamerami VICON.

Kapitola 3

Teoretická východiska

V této kapitole představíme potřebná teoretická východiska týkající se multi-agentního hledání cest, která nám pomohou definovat problém displeje v další kapitole.

3.1 Multi-agentní hledání cest

Problém multi-agentního hledání [5] spočívá v tom, že máme sadu agentů, kteří se pohybují v daném prostředí reprezentovaném grafem, vyhýbají se překážkám a mají startovní a finální umístění. Cílem je najít bezkonfliktní (během konkrétního okamžiku nejsou přítomni ve stejném vrcholu) cesty agentů tak, aby každý agent dorazil do cíle. Někdy se také vyžaduje, aby agenti necestovali po totožné hraně ve stejný časový okamžik. Obrázek 3.1 ilustruje řešení MAPF instance pro 3 agenty.

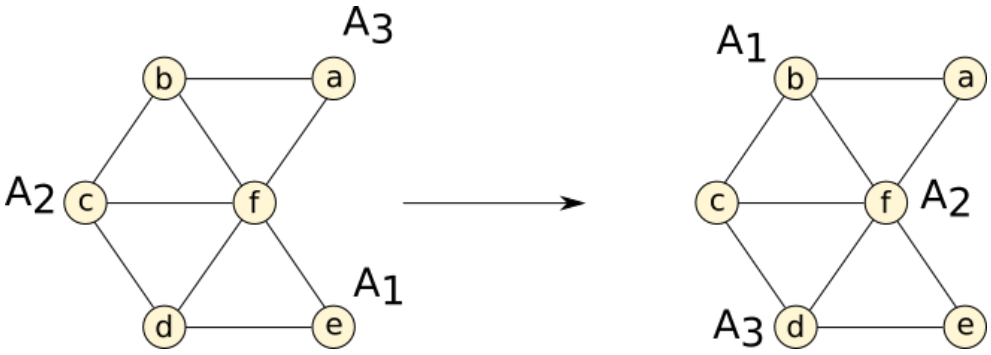
Definice (Problém multi-agentního hledání cest)**.** *Nechť máme graf* $G = (V, E)$ *reprezentující dané prostředí a nechť existují agenti* $A = (a_1, a_2, ..., a_n)$*. Nechť je dáno prosté počáteční umístění agentů* $\varphi_s : A \to V$ *a prosté finální umístění* $\varphi_f : A \to V$*. Problémem multi-agentního hledání cest je najít číslo* $m \in \mathbb{N}$ *a umístění* $(\varphi_s = \varphi_0, \varphi_1, \varphi_2, ..., \varphi_m = \varphi_f)$ *za splnění těchto podmínek:*

$$\forall i \in \hat{m}, \forall a, b \in A : \varphi_i(a) = \varphi_i(b) \Rightarrow a = b \tag{3.1}$$

$$\forall i \in (0, 1, 2, ..., m-1), \forall a \in A : \varphi_i(a) = \varphi_{i+1}(a) \vee \{\varphi_i(a), \varphi_{i+1}(a)\} \in E \tag{3.2}$$

Podmínka 3.1 značí, že v každém umístění nedojde k tomu, aby dva různí agenti obsadili stejný vrchol, tedy nedojde ke konfliktu. Druhá podmínka 3.2 znamená, že agenti mezi umístěními buď čekají na místě, nebo se pohybují po hranách grafu.

Je dokázáno, že verze MAPF problému s optimalizačním kritériem na co nejmenší m (tedy co nejkratší cesty) je NP-úplná. Důkaz tohoto tvrzení je ovšem obsáhlejší, proto se na něj pouze odkáži [6].



Obrázek 3.1: Ilustrační příklad multi-agentního hledání cest. Například tyto cesty jsou validním řešením MAPF instance:
Agent 1: (e, f, b, b)
Agent 2: (c, c, f, f)
Agent 3: (a, b, c, d)

3.1.1 Bibox

V reálném světě dochází k tomu, že chceme vyřešit MAPF instance na ploše nebo v prostoru. Grafy reprezentující dané prostředí jsou obvykle 2-souvislé. Toho využívá algoritmus Bibox [5], který umí spočítat řešení pro 2-souvislé grafy. Díky vlastnostem těchto grafů a relaxaci optimalizačního kritéria (nevyžadujeme nejkratší možné řešení) dokáže tento algoritmus najít řešení multi-agentního hledání cest v čase $\mathcal{O}(|V(G)|^3)$.

Pro popis ideje tohoto algoritmu zadefinujme nejprve pojmy týkající se 2-souvislosti grafů.

Definice (2-souvislost grafů)**.** *Buď dán graf $G = (V, E), |V(G)| \geq 3$. Graf G je 2-souvislý, pokud platí $\forall(u, v, w) \in V(G)^3, u \neq v \wedge u \neq w \wedge v \neq w$ pak existuje cesta mezi vrcholy u, w i v grafu $G \setminus v$.*

Definice (Ucho)**.** *Nechť $G = (V, E)$ je 2-souvislý graf. Ucho je cesta $U = (u_1, u_2, ..., u_k)$. Přidáním ucha říkáme operaci sjednocení grafů G a U tak, že vrcholy u_1 a u_k se namapují na existující vrcholy $x, y \in V(G) : x \neq y$.*

Klíčová je i následující známá věta o dekompozici 2-souvislých grafů.

Věta (Dekompozice 2-souvislých grafů)**.** *Každý 2-souvislý graf lze sestrojit z počátečního cyklu pomocí operace přidávání uší.*

Nyní přejdeme k hlavní myšlence algoritmu. Díky větě o dekompozici grafů 3.1.1 dokážeme rozdělit graf G na iniciální cyklus C_0 a ucha $C_1, C_2, ..., C_k$. Označme $C(C_i)$ jako indukovaný podgraf G obsahující všechny vrcholy z iniciálního cyklu a uch $C_0, C_1, ..., C_i$ (viz obrázek 3.2a). Dále předpokládejme, že robotů je přesně $n - 2$ (o dva méně než je počet vrcholů v grafu) a finální umístění volných míst se nachází v cyklu C_1.

Algoritmus Bibox funguje ve dvou fázích. V té první algoritmus iteruje přes podgrafy $C(C_k)$ až $C(C_1)$ - tedy velikost problému se s každou iterací postupně zmenšuje. V rámci každé i-té iterace algoritmus řeší jedno ucho. Zvolme orientaci ucha a označme jeho vrcholy $(u_{i,0}, u_{i,1}, ..., u_{i,s-1})$, kde s je počet vrcholů daného ucha. Nejprve budeme řešit agenta $a_{i,s-1}$ (v i-té iteraci algoritmu Bibox patří na pozici ucha $u_{i,s}$). Protože je graf 2-souvislý a skládá se z cyklů (daných uchy a jejich nejkratší spojnicí), můžeme rotovat těmito cykly tak, že dostaneme robota $a_{i,s}$ z libovolného místa do vrcholu $u_{i,0}$ (speciální případ je, když tento robot se již nachází v daném cyklu; i to ale umí algoritmus vyřešit). Poté orotujeme cyklus s uchem o jedno místo (robot se pak nachází na pozici $u_{i,1}$). Tento případ je ilustrován na obrázku 3.2b. Takto to algoritmus zopakuje pro agenty $a_{i,s-2}$ až $a_{i,0}$. Po s opakováních máme správné řešení pro ucho v dané iteraci.

Ve druhé fázi algoritmus řeší počáteční cyklus, ve kterém je potřeba zvolit jiný přístup. Díky tomu, že máme v počátečním cyklu C_0 2 volné vrcholy, lze pomocí interakcí s dalšími cykly prohodit 2 libovolné roboty. Z toho plyne, že je možné uspořádat roboty v tomto cyklu do správné finální permutace, kterou algoritmus nakonec dorotuje do správných pozic.

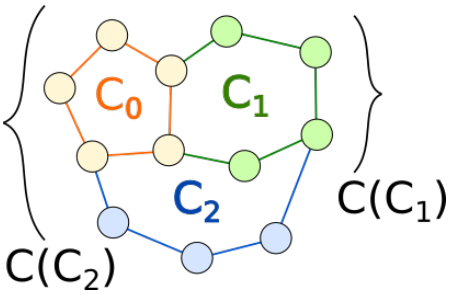
Toto byl hrubý nástin algoritmu Bibox, který hledí hlavně na rychlost výpočtu, který najde v polynomiálním čase. Nemá ovšem žádnou garanci kvality řešení. Výše popsaný algoritmus má ve své čiré formě zásadní omezení na počet agentů a jejich umístění. Toto omezení se dá ovšem překonat jednoduchými úpravami, které jsou popsány v původním článku.

3.1.2 M* algoritmus

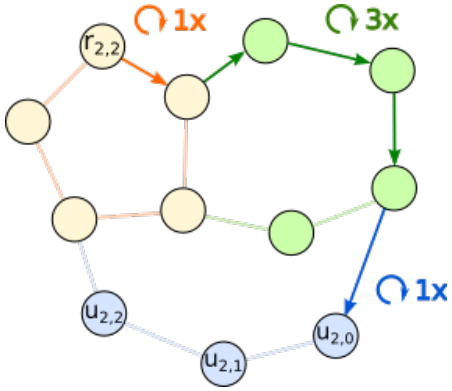
M* algoritmus [7] je optimální algoritmus založený na původním algoritmu A* [8] pro jednoho agenta. Dokonce je možné využít algoritmus A* pro výpočet MAPF instancí. Ten však prohledává celý konfigurační prostor instance, který roste exponenciálně s počtem robotů. Proto podle měření autorů článku i pro malý počet agentů algoritmus A* přestane poskytovat výsledky v rozumném čase. Oproti tomu M* má to vylepšení, že neprohledává celý konfigurační prostor, ale ořezává jej podle toho, kteří agenti spolu mohou interagovat. Agenty, jejichž cesty se nemohou potkat, řeší algoritmus M* samostatně.

Nyní představíme důležité pojmy pro tento algoritmus. Pro každého agenta $a_i \in \hat{n}$ nechť máme orientovaný graf $G^i = (V^i, E^i)$ představující prostředí pro konkrétného agenta s počátečním vrcholem ${v_I}^i$ a s finálním vrcholem ${v_F}^i$. Nechť $\varphi^i(v)$ je funkce, která agentu a_i přiřadí nejkratší cestu z libovolného vrcholu $v \in V^i$ do cíle ${v_F}^i$.

Prostředí pro všechny agenty definujme jako graf vytvořený kartézským součinem grafů: $G = \prod_{i \in \hat{n}} G^i$. Takový graf umožní transformovat MAPF instanci na hledání cesty v grafu, na který můžeme použít algoritmus založený na A*. Ještě definujme počáteční vrchol jako v_I a cílový vrchol jako v_F. Heuristickou funkci na odhad zbývající ceny $h : v \to r, v \in G(V), r \in \mathbb{R}$ zadefinujme



(a) Ilustrace dekompozice grafu. Na obrázku je vidět iniciální cyklus C_0, který je identický s grafem $C(C_0)$ a uchy C_1 a C_2. Složené závorky ilustrují grafy $C(C_1)$ a $C(C_2)$.



(b) Ilustrace umístění robota $r_{2,2}$ na vrchol $u_{2,1}$ pomocí rotací cyklů. V ukázce se po směru hodinových ručiček otočí cyklus C_0, poté cyklus tvořený uchem C_1 a jeho nejkratší spojnicí 3x a poté jedním otočením cyklu C_2 se robot dostane na pozici $u_{2,0}$. Po vyřešení robotů $r_{2,1}$ a $r_{2,0}$ se robot přesune na správnou pozici $u_{2,2}$.

Obrázek 3.2: Algoritmus Bibox

jako součet cen pro jednotlivé agenty: $h(v) : \sum_{i \in \hat{n}} h(v_i), v_i \in G(V^i)$. Notace $\Psi(v)$ označuje set agentů, kteří jsou ve vrcholu v v konfliktu.

Algoritmus M* se od A* liší pouze v drobnostech. Tou první je, že každý vrchol $v_k \in V(G)$ obsahuje konfliktní set C_k. Hlavní rozdíl oproti A* je v tom, že M* neexpanduje všechny sousedy libovolného vrcholu v_k, ale pouze jejich podmnožinu danou tímto předpisem: $\hat{V}_k = \{v_l : \forall i \in \hat{n} : (i \in C_k \wedge e^i{}_{k,l} \in E^i) \vee (i \notin C_k) \wedge v^i{}_l = \varphi^i(v^i{}_l)\}$. Množina je daná buď funkcí φ^i pro optimální trajektorii (pokud není agent v konfliktu), nebo sousedy vrcholu v_k. Pro každý vrchol si ukládáme set předchozích vrcholů. Aktualizace konfliktního setu se pak dělá přes zpětnou propagaci konfliktů přes zpětné sety. Při zpětné propagaci může dojít k tomu, že některé vrcholy bude potřeba kvůli změně konfliktního setu znovu otevřít.

K ujasnění předchozího textu ještě předvedeme pseudokód algoritmu M* (viz algoritmus 1).

Algoritmus je kompletní, optimální vzhledem k celkové ceně a vypočítá výsledky v exponenciálním čase vzhledem k počtu agentů.

3.1.3 Varianty M*

Stejný článek [7] dále popisuje možnosti, jak dosáhnout zrychlení algoritmu M*. Jednoduchý způsob, jakým zrychlit algoritmus, je použití již existující techniky pro algoritmus A* zvané *nafouknutá heuristika* (anglicky inflated

Algorithm 1 Algoritmus M*

```
1: function BACKPROPAGATE(vrchol v_k, set C_l, fronta open)
2:     if C_k ⊄ C_l then
3:         C_k ← C_k ∪ C_l
4:         if v_k ∉ open then
5:             open.add(v_k)
6:         end if
7:         for all v_m ∈ v_k.backset do
8:             BACKPROPAGATE(v_m, C_k, open)
9:         end for
10:    end if
11: end function
12:
13: function M*(graf G)
14:    for all v_k ∈ V(G) do
15:        v_k.cost ← inf
16:        C_k ← ∅
17:    end for
18:    v_I.cost ← 0
19:    v_I.previous ← ∅
20:    open ← {v_I}                    ▷ Prioritní fronta podle v.cost + h(v)
21:    while true do
22:        v_k ← open.pop()
23:        if v_k = v_F then
24:            return BACKTRACK(v_k)
25:        end if
26:        if Ψ(v_k) ≠ ∅ then          ▷ Neprocházíme stavy s konflikty
27:            continue
28:        end if
29:        for all v_l ∈ V̂_k do
30:            v_l.backset.append(v_k)
31:            C_l ← C_l ∪ Ψ(v_k)
32:            open.add(v_l)
33:            BACKPROPAGATE(v_k, C_l, open)
34:            if v_k.cost + f(e_k,l) < v_l.cost then   ▷ Našli jsme lepší řešení
35:                v_l.cost = v_k.cost + f(e_k,l)
36:                v_l.previous = v_k
37:            end if
38:        end for
39:    end while
40: end function
```

heuristic). Tato technika spočívá v tom, že hodnotu heuristiky $h(v)$ pro nějaký vrchol $v \in V(G)$ přenásobíme konstantou $\varepsilon > 1$. To způsobí, že cena nalezeného řešení nebude horší než ε-násobek ceny optimálního řešení.

Další zlepšení algoritmu autoři nalezli v modifikaci způsobu, jakým dochází ke zpracování konfliktních setů. Například když konfliktní set C_k obsahuje agenty $\{1, 2, 3, 4, 5\}$, nemusí to nutně znamenat, že všichni agenti spolu interagují. To se může stát, když v konfliktu jsou například dvojice agentů $(1, 2)$, $(2, 3), (4, 5)$. Varianta zvaná *rekurzivní M** (rM*) tento problém vyřeší rozdělením na největší disjunktní sety, které pak řeší separátně. V našem příkladě by se konfliktní set C_k změnil na $C_k = \{\{1, 2, 3\}, \{4, 5\}\}$.

Obě tyto varianty dokáží dle experimentálního vyhodnocení autorů zrychlit algoritmus a umožnit praktický výpočet pro větší množství agentů ve stejném čase.

3.1.4 Conflict Based Search

CBS [9] (hledání pomocí konfliktů) je optimální algoritmus multi-agentního hledání cest, který řeší problém tak, že řeší cesty jednotlivých agentů individuálně. To je rychlé, neboť cesta individuálního agenta se dá najít v lineárním čase vzhledem k velikosti grafu. Při individuálním plánování ovšem může nastat až exponenciálně mnoho konfliktů. Algoritmus se skládá ze dvou částí: vysokoúrovňová a nízkoúrovňová. Popíšeme postupně obě dvě.

Nejprve zadefinujeme několik pojmů. Termín *cesta* se použije pouze v kontextu jednoho agenta, pro všechny agenty se použije termín *řešení*. *Omezení* je trojice (a_i, v, t), která určuje agentu a_i, že v čase t nesmí vstoupit do vrcholu v. *Konzistentní cesta* je taková cesta, která neporušuje žádné omezení. Obdobně se definuje *konzistentní řešení*, což je takové řešení, kde každý agent dodržuje všechna omezení. *Konflikt* je čtveřice (a_i, a_j, v, t) , kde a_i a a_j jsou 2 různí agenti $(a_i \neq a_j)$ a v je vrchol, který oba agenti okupují v čase t.

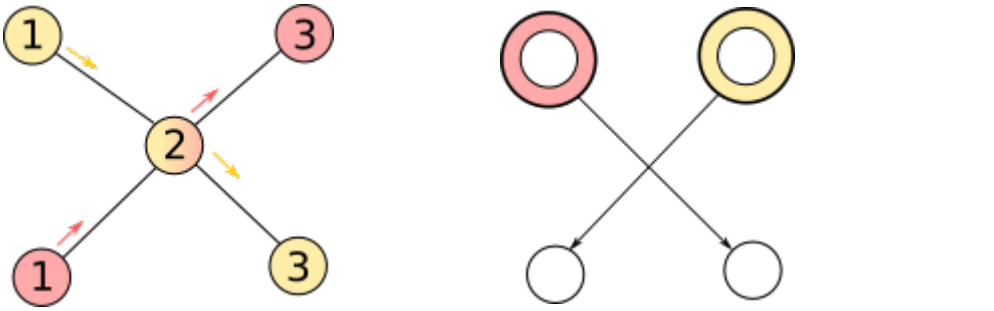
Hlavní myšlenka výsokoúrovňové části algoritmu tkví v tom, že algoritmus v každém kroku zkontroluje nalezené konzistentní řešení a při nalezení konfliktu jej přidá jako omezení agentům a_i a a_j. Pro účel algoritmu rozepíši, z jakých částí se skládá řešení $\mathcal{S}$. První částí jsou cesty jednotlivých agentů, značí se $\mathcal{N}.paths$. Další částí, která se značí $\mathcal{N}.cost$, je myšlena suma jednotlivých cen. Poslední část je $\mathcal{N}.constrains$. To je konfliktní tabulka, která každému agentu přiřazuje omezení. Nyní předvedeme celou vysokoúrovňovou část algoritmu.

Algorithm 2 CBS - Vysokoúrovňové hledání

```
1:  function CBS(MAPF instance)
2:      S.constrains ← ∅
3:      S.cost = 0
4:      for all agent do
5:          S.paths[agent] ← low_level_search(agent, ∅)      ▷ Inicializace cest.
6:          S.cost += S.paths[agent].cost
7:      end for
8:      OPEN ← {}                                              ▷ Prioritní fronta
9:      OPEN.push(S)
10:     while OPEN není prázdná do
11:         P ← OPEN.pop()                                     ▷ Řešení s nejnižší cenou
12:         C ← find_first_conflict(P)                         ▷ První nalezený konflikt
13:         if C = ∅ then
14:             return P
15:         end if
16:         D ← duplicate(P)                                   ▷ Zduplikuj stávající řešení
17:                                                            ▷ Vyřeš prvního agenta
18:         P.constrains += (C.a_1, C.v, C.t)
19:         P.cost −= P.paths[C.a_1].cost
20:         P.paths[C.a_1] ← low_lewel_search(C.a_1, P.constrains)
21:         P.cost += P.paths[C.a_1].cost
22:         Q.push(P)
                                                               ▷ Vyřeš druhého agenta
23:         D.constrains += (C.a_2, C.v, C.t)
24:         D.cost −= D.paths[C.a_2].cost
25:         D.paths[C.a_2] ← low_lewel_search(C.a_2, P.constrains)
26:         D.cost += P.paths[C.a_2].cost
27:         Q.push(D)
28:     end while
29: end function
```

Zajímavou myšlenkou je, co se stane, když je v konfliktu více agentů než 2. To se dá řešit dvěma způsoby. První způsob je, že se berou v potaz pouze první dva agenti. To bude fungovat, protože další konflikty algoritmus nalezne v dalších krocích. Tento způsob jsme využili i v ukázce algoritmu. Druhý způsob je, že algoritmus vezme v potaz všechny agenty a vygeneruje pro každého z nich řešení, ve kterém ostatním přidá omezení. Toho by šlo dosáhnout lehkou úpravou předvedeného algoritmu.

Nyní probereme nízkoúrovňové hledání. Tím může být jakýkoli algoritmus hledání cest, který nalezne cestu pro jednotlivého agenta při dodržení jednotlivých omezení. Autoři CBS použili ve svých experimentech známý algoritmus A* [8].



(a) Konflikt červeného a žlutého agenta v čase 2 při běhu CBS.

(b) Konflikt červeného a žlutého agenta při běhu CCBS.

Obrázek 3.3: Konflikty agentů v různých variantách CBS.

3.1.5 CBS se spojitým časem

Důležitým rozšířením CBS je CCBS [10] algoritmus, který umožňuje vyhledat konflikty agentů ve spojitém čase. Rozdíly popíši v následujících odstavcích.

Detekce konfliktů ve standartním CBS probíhá tak, že algoritmus zkontroluje, zda více agentů neokupuje stejný vrchol ve stejný časový okamžik. Při spojitém času ovšem může docházat k tomu, že agenti mohou dorazit do vrcholu grafu v jiný časový okamžik. Mimo jiné CCBS narozdíl od CBS uvažuje geometrický tvar agentů a jejich rychlost, a to může vést ke konfliktu, i pokud 2 různí agenti se pohybují po jiných vrcholech nebo hranách. Proto je nutné změnit definici konfliktu:

CCBS konflikt je definován jako čtveřice (a_i, t_i, a_j, t_j), která označuje, že pokud agent i vykoná akci a_i (akcí označujeme buď pohyb po hraně grafu nebo čekání ve vrcholu) v čase t_i a pokud agent j vykoná akci a_j v čase t_j, dojde ke konfliktu.

Autoři CCBS ve svých experimentech zjistili, že detekce konfliktů zabírá velké množství času při běhu algoritmu. Pro zkrácení tohoto času navrhují heuristiku, která se místo přesné detekce konfliktů omezí na to, zda je možné, aby se agenti srazili při provedení akcí v určitých časech. Tato heuristika přidává více falešně pozitivních konfliktů za cenu jejich rychlejší detekce.

Řešení konfliktů je podobné CBS. Stejně jako v CBS, algoritmus najde první konflikt a poté aplikuje omezení na zúčastněné agenty. Omezení v CCBS má podobu trojice $(i, a_i, [t_1, t_2))$, která značí, že i-tý agent nemůže vykonat akci a_i v časovém intervalu $[t_1, t_2)$.

Nízkoúrovňový algoritmus musí počítat se spojitým časem. Z tohoto důvodu autoři doporučují algoritmus SIPP [11], který umí najít cestu jednotlivému agentovi s dynamickými překážkami. Hlavní myšlenka SIPP je detekce intervalů bez kolizí pro každý vrchol, čímž dojde k rozdělení na časové úseky, a to umožní použití diskrétního algoritmu. SIPP vrátí sekvenci akcí, která umožní agentovi dojít do cíle.

3.1.6 Suboptimální CBS

Standartní verze CBS algoritmu vrací optimální výsledky, a to znamená, že tento algoritmus je ve svém základu NP-úplný. Někdy příliš nezáleží na kvalitě výsledku, anebo záleží pouze do určité míry. Článek o suboptimálních variantách CBS [12] pojednává o několika způsobech, jakými zrychlit vykonávání CBS algoritmu, a o tom, jaké důsledky mají na kvalitu výsledku.

3.1.6.1 Hladová varianta

První suboptimální varianta se nazývá **GCBS**, neboli hladová. Tato varianta v základu nezaručuje jakoukoli garanci kvality výsledku. Lze ji ale poupravit, aby hledala pouze ta řešení, jejichž cena je menší nebo rovna danému k.

GCBS přidává heuristiky, které se snaží minimalizovat počet konfliktů. Tyto heuristiky jsou důležité zejména ve vysokoúrovňové části CBS, ale i pro nízkoúrovňovou část algoritmu.

Autoři diskutují o několika vysokoúrovňových heuristikách a to:

- h_1: **počet konfliktů** – heuristika, která preferuje řešení s nejmenším množstvím konfliktů.
- h_2: **počet konfliktních agentů** – preferuje řešení s co nejmenším množstvím agentů, který mají alespoň jeden konflikt.
- h_3: **počet párů konfliktních agentů** – počítá každou dvojici agentů, které mají alespoň 1 konflikt. Preferuje řešení s nejmenším množstvím těchto dvojic.
- h_4: **vrcholové pokrytí** – definuje konfliktní graf, kde vrcholy představují konfliktní agenty (mohou být identifikovány pomocí h_2) a hrany jsou konflikty mezi agenty (mohou být identifikovány pomocí h_3). Heuristika h_4 poté pracuje s vrcholovým pokrytím tohoto grafu.
- h_5: **alternující** – použití více heuristik, v každém kroce se vybere jiná, a to pomocí metody round robin.

Podle provedených experimentů bylo zjištěno, že h_5 je nejrychlejší. Je ovšem také nejtěžší na implementaci, protože vyžaduje naimplementování ostatních a údržbu struktur, které ostatní heuristiky používají. Heuristika h_4 vrací lepší výsledky než ostatní, bohužel na úkor větší složitosti algoritmu, a je pomalejší. Heuristika h_2 funguje rychle na některých instancích, pomalu na jiných. První heuristika h_1 funguje v průměru rychleji než heuristika h_3. Heuristika h_3 ovšem funguje robustněji než h_1 a h_2. Autoři článku doporučují heuristiku h_3 jako dobrý kompromis mezi rychlostí algoritmu a jednoduchostí implementace.

Dalším způsobem, jakým zrychlit výpočet algoritmu, je zaměřit se na nízkoúrovňovou část algoritmu. Jedním z teoreticky možných řešení je využití

rychlejšího algoritmu, který může najít horší cestu, ovšem rychleji. Autoři zmiňují algoritmus WA* [13]. Při měření ovšem došli k závěru, že WA* nachází delší cesty, u kterých je větší pravděpodobnost, že dojde ke kolizi, a proto je ve skutečnosti výpočetní doba delší.

Ukázalo se, že lepším způsobem úpravy nízkoúrovňového algoritmu je takový, který se při plánování pohybu agenta aktivně vyhýbá jiným agentům i přesto, že výsledná cesta může být mnohem delší. Mimo konfliktní vrcholy ovšem algoritmus plánuje nejkratší možnou cestu. Příkladem může být, že když algoritmus najde cestu přes vrcholy (S_1, A, B, S_2) pro agenta a_1, při dalším běhu pro agenta a_2 algoritmus preferuje cesty, které se v čase $t = 1$ se vyhýbají vrcholu A a v čase $t = 2$ vyhýbají vrcholu B. Při výpočtu množství konfliktů algoritmus používá stejnou heuristickou funkci jako v předchozím případě.

3.1.6.2 Ohraničená varianta

BCBS je první varianta CBS algoritmu, která garantuje, že nalezené řešení nebude horší než daný w-násobek optimální ceny. Důležitým konceptem BCBS je tzv. *focal search*. Tento koncept upravuje způsob, jakým funguje hledací algoritmus, a dá se aplikovat na obě úrovně CBS algoritmu. *Focal search* požaduje existenci funkcí f_1 a f_2 a váhový faktor w. Funkce f_1 představuje cenovou funkci cest. Kromě standartní prioritní fronty zvané *OPEN* existuje druhá prioritní fronta navaná *FOCAL*, která používá heuristickou funkci f_2. *FOCAL* fronta obsahuje pouze část otevřených stavů. Buď $f_{1_{min}}$ stav s minimální cenou podle funkce f_1. Potom *FOCAL* fronta obsahuje všechny stavy n, pro které platí $f_1(n) \leq w * f_{1_{min}}$, tedy že jejich cena je maximálně w-násobkem minimální ceny.

Algoritmus BCBS ve vysokoúrovňovém hledání aplikuje *focal search* s těmito funkcemi: f_1 je původní cenová funkce algoritmu CBS, f_2 je konfliktní heuristická funkce z algoritmu GCBS. Obdobně na nízké úrovni f_1 je cenová funkce A* algoritmu a f_2 je opět stejná heuristická funkce jako na vysoké úrovni.

$BCBS(w_h, w_l)$ označuje algoritmus, který používá *focal search* s váhou w_h na vyšší úrovni a *focal search* s váhou w_l na nižší úrovni. Následující věta ukazuje, jaký vliv má volba w_h a w_l na kvalitu celkového řešení.

Věta. *Buď C^* cena optimálního řešení MAPF instance. Pro daná $w_h, w_l \geq 1$ vrátí algoritmus $BCBS(w_h, w_l)$ řešení s maximální cenou $w_h \cdot w_l \cdot C^*$.*

Pro požadovanou kvalitu s cenou maximálně w-násobku optimálního lze zvolit libovolná čísla w_l, w_h, pro která platí $w_h * w_l = w$. Je ale těžké vybrat správnou kombinaci těchto čísel a bylo změřeno, že pro dané hodnoty rychlost algoritmu závisí na konkrétních instancích. Kvůli tomu nelze obecně určit, jak rozdistribuovat hodnoty w_h a w_l. Proto autoři přišli s dalším vylepšeným algoritmem, ve kterém je potřeba pouze parametr w.

3.1.6.3 Vylepšená varianta

ECBS je další varianta CBS algoritmu, která má garanci w-násobku optimální ceny. Na nízké úrovni se opět využívá *focal search*, a to s váhou rovné w. Standartně nízká úroveň vrací pro každého agenta a_i pouze cestu a cenu řešení. V případě ECBS vrací i nejmenší cenu stavu z fronty *OPEN* značenou $a_i.lower\_bound$. Určitě platí $a_i.lower\_bound \leq a_i.cost \leq w * a_i.lower\_bound$.

Každé řešení s v konfliktním stromu CBS má definováno $LB(s) \coloneqq \sum_{i=1}^{n} a_i.lower\_bound$. Dolní odhad na cenu optimálního řešení C^* je definován $LB \coloneqq \min_{s \in OPEN} LB(s)$. Nyní lze definovat obsah fronty *FOCAL* jako všechny stavy, jejichž cena je maximálně w-násobek dolního obsahu. Formálně $FOCAL \coloneqq \{\forall s \in OPEN : s.cost \leq w * LB\}$. Nyní lze zadefinovat $ECBS(w)$ jako běh algoritmu ECBS s váhou $w \geq 1$ a nějakou implicitní konfliktní heuristikou z předchozích odstavců.

Věta. *Buď C^* cena optimálního řešení MAPF instance. Potom pro danou váhu $w \geq 1$ vrátí algoritmus $ECBS(w)$ řešení s maximální cenou $w * C^*$.*

Důkaz. Protože LB je dolní odhad na optimální cenu, musí platit $LB \leq C^*$. A protože algoritmus prochází pouze ty stavy, jejichž cena je maximálně w-násobek dolního odhadu, musí pro cenu nalezeného řešení C_{res} platit tento vztah: $C_{res} \leq w \cdot LB \leq w \cdot C^*$. □

Algoritmus ECBS vrátí řešení s požadovanou kvalitou. Autoři algoritmu experimentálně ověřili, že v praxi algoritmus ECBS běží rychleji v porovnání s BCBS.

3.2 Rozlišené a nerozlišené plánování

Výše představený problém multi-agentního hledání cest 3.1 je nazván *rozlišený*. Důvod je, že každý agent má známou výchozí i cílovou polohu. *Nerozlišený* problém, představený v článku [14], se liší tím, že cílové pozice nemají určené konkrétní agenty.

Definice (Nerozlišený problém multi-agentního hledání cest)**.** *Nechť máme graf $G = (V, E)$ reprezentující dané prostředí a nechť existují agenti $A = (a_1, a_2, ..., a_n)$. Nechť je dána prostá funkce reprezentující počáteční umístění agentů $\varphi_s : A \to V$ a cílové pozice $F = (f_1, f_2, ..., f_n) | \forall i, j \in \hat{n} : f_i, f_j \in V \wedge f_i = f_j \Rightarrow i = j$. Problémem nerozlišeného multi-agentního hledání cest je najít číslo $m \in \mathbb{N}$ a umístění $(\varphi_s = \varphi_0, \varphi_1, \varphi_2, ..., \varphi_m)$ za splnění těchto podmínek:*

$$\forall i \in \hat{m}, \forall a, b \in A : \varphi_i(a) = \varphi_i(b) \Rightarrow a = b \tag{3.3}$$

$$\forall i \in (0, 1, 2, ..., m-1), \forall a \in A : \varphi_i(a) = \varphi_{i+1}(a) \vee \{\varphi_i(a), \varphi_{i+1}(a)\} \in E \tag{3.4}$$

$$\varphi_m(A) = F \tag{3.5}$$

Podmínky 3.3 3.4 jsou stejné jako v nerozlišeném případě. Navíc podmínka 3.5 zaručí správné finální umístění agentů.

3.2.1 Síťový algoritmus

Pro zajímavost uvedu rozhodující verzi diskrétního algoritmu [15], která pro dané K rozhodne, zda existuje řešení nerozlišeného problému multi-agentního hledání cest. Předpokládejme nyní hrany s uniformní cenovou funkcí $c : E \Rightarrow R, \forall e \in E(G) : c(e) = 1$. Článek, který pojednává o plánování trajektorií [14] zmiňuje, že vhodný graf prostředí lze vygenerovat například pomocí algoritmu SPARS [16].

Síťový algoritmus využívá dynamicky generovanou síť, která pro daný graf $G = (V, E)$ obsahuje celkem $T \cdot |V(G)|$ vrcholů.

Definice (Časově rozšířená síť)**.** *Nechť je zadána instance multi-agentního vyhledávání cest 3.1 a dáno číslo $T \in \mathbb{N}$. Časově rozšířená síť S obsahuje $(T+1)*V(G)$ vrcholů. Pro každý časový okamžik t okopírujeme všechny vrcholy $v_i : i \in |V(G)|$ z původního grafu G a označíme $v'_{it} : i \in (1, ..., |V(G)|), t \in \hat{T}$. Hranu přidáme do sítě S právě tehdy, pokud byla v původním grafu a v novém grafu vede mezi dvěma časovými okamžiky. Formálně:*

$$\forall i, j \in (1, 2, ..., |V(G)|), t \in (0, 1, 2, ..., T-1) : \\ (v'_{it}, v'_{j\,t+1}) \in V(S) \Leftrightarrow (v_i, v_j) \in V(G) \vee i = j$$

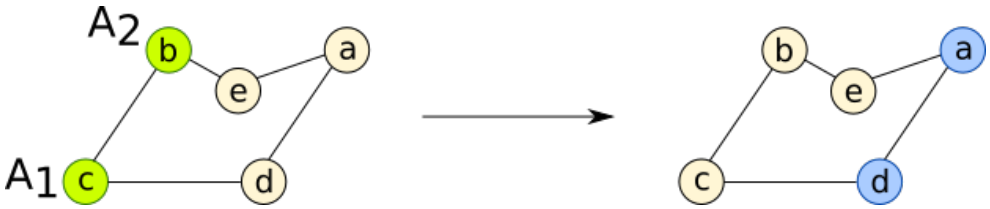
Cenová funkce je uniformní, formálně platí: $\forall e \in E(S) : c(e) = 1$. Zadefinujme nyní superzdroj S^+ jako sadu vrcholů o celkové velikosti $|A|$, kde každý vrchol $z \in S^+$ představující agenta $a_i, i \in (1, 2, ..., |A|)$ je umístěn na vrchol $v'_{j1}, j \in (1, 2, ..., |V(G)|)$ právě tehdy, je-li jeho počátační pozice na vrcholu v_j. Dále zadefinujme superstok S^- jako set vrcholů, kde každý vrchol $s \in S^-$ odpovídá finálnímu umístění agenta. Tedy každý vrchol v'_{ij} :
$\forall i \in (1, 2, ..., |A|), j \in (0, 1, 2, ..., |T|)$ je stokovým vrcholem právě tehdy, je-li v_i finálním umístěním.

Edmonsonův-Karpův algoritmus [17] dokáže pomocí časově rozšířeného grafu určit, zda lze najít cestu v dané instanci v maximálním čase T.

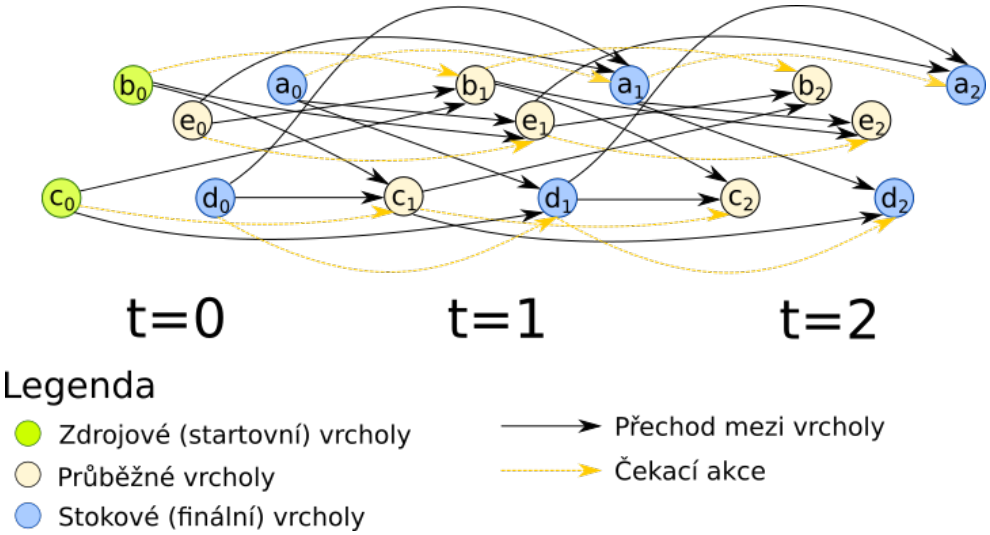
3.2.2 Lineární součtové přiřazení

Alternativně se dá nerozlišený problém multi-agentního plánování cest řešit aproximativně tím, že se nejprve přiřadí finální umístění agentů [14] a poté se spustí klasický algoritmus vyhledávání cest v rozlišené verzi tohoto problému.

Finální umístění agentů se dá najít vyřešením problému lineárního součtového přiřazení (LSAP), který představíme v následující definici [18].



(a) Instance nerozlišeného multi-agentního hledání cest. Ze startovních vrcholů vyznačených zelenou barvou se agenti 1 a 2 musí dostat do cílových vrcholů vyznačených modrou barvou.



(b) Časově rozšířená síť stejné instance jako 3.4a s časovým parametrem $T = 2$. Všechny hrany i vrcholy mají jednotkovou kapacitu.

Obrázek 3.4: Nerozlišený MAPF a časově rozšířená síť

Definice (Problém lineárního součtové přiřazení)**.** *Nechť je dána matice* $C \in \mathbb{R}^{+n \times n}$*. Problémem lineárního součtového přiřazení nazýváme nalezení* n *prvků matice* C *tak, aby z každého řádku i každého sloupce matice byl vybrán právě jeden prvek a aby celková suma těchto prvků byla co nejmenší možná.*

Pojďme si představit i alternativní definici tohoto problému. Tuto alternativní definici využívá níže uvedený Maďarský algoritmus.

Definice (Alternativní definice problému lineárního součtového přiřazení)**.** *Buď na vstupu dána matice* $C \in \mathbb{R}^{+n \times n}$*. Nyní definujme bipartitní graf s partitami* $U = (u_1, u_2, ..., u_n)$ *a* $V = (v_1, v_2, ..., v_n)$ *představujícími řádky a sloupce. Dále pro každou dvojici* $(i, j) : i \in \hat{n}, j \in \hat{n}$ *nechť existuje hrana v bipartitním grafu* (u_i, v_j) *s váhou* $c_{i,j}$*. Cílem lineárního součtového přiřazení je nalézt takové perfektní párování mezi partitami* U, V *tak, aby celková váha všech hran byla co nejmenší.*

3.2.2.1 Maďarský algoritmus

Jedním z algoritmů řešících problém LSAP s časovou komplexitou $\mathcal{O}(n^3)$ je Maďarský algoritmus [18], který nyní představíme.

Maďarský algoritmus se skládá ze tří funkcí. První funkce není specifická přímo pro Maďarský algoritmus, ale používá se u většiny LSAP algoritmů k předzpracování vstupní matice a nalezení částečného řešení.

Algorithm 3 Preprocessing matice C

```
1: function LSAP_PREPROCESSING(Matice C)
2:     C̄ ← [0]_{n×n}                                  ▷ Nulová matice n × n
3:     row_red ← col_red ← [0]_n
4:     row_ass, col_ass = [−1]_n
5:     for all i ∈ î do
6:         row_red[i] ← min{c_{i,j} : j ∈ î}
7:     end for
8:     for all j ∈ î do
9:         col_red[j] ← min{c_{i,j} − row_red[i] : i ∈ î}
10:    end for
11:    for all i ∈ î do                                ▷ Částečné řešení
12:        for all j ∈ î do
13:            c̄_{i,j} ← c_{i,j} − row_red[i] − col_red[j]
14:            if row_ass[j] = −1 ∧ c̄_{i,j} = 0 then
15:                row_ass[j] = i
16:            end if
17:        end for
18:    end for
19:    for all j ∈ î do
20:        for all i ∈ î do
21:            if col_ass[i] = −1 ∧ c̄_{i,j} = 0 then
22:                col_ass[i] = j
23:            end if
24:        end for
25:    end for
26:    return (row_red, row_ass, col_red, col_ass)
27: end function
```

$$C = \begin{pmatrix} 3 & 3 & 10 & 3 \\ 2 & 5 & 6 & 6 \\ 5 & 2 & 5 & 2 \\ 3 & 5 & 10 & 4 \end{pmatrix} \qquad \bar{C} = \begin{pmatrix} \underline{\mathbf{0}} & 0 & 4 & 0 \\ 0 & 3 & \underline{\mathbf{1}} & 4 \\ 3 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \end{pmatrix}$$

V případě uvedené matice C vychází výše uvedená redukovaná matice $\bar{C}$ s těmito dalšími proměnnými. Výsledek vektoru pro redukci matice C po

řádcích row_{red} vychází $[3, 2, 2, 3]$, redukce po sloupcích col_{red} vychází $[0, 0, 3, 0]$. Částečné řešení (vyznačené v matici $\hat{C}$ podtrženými tučnými čísly) row_{ass} pro řádky je $[0, 2, -1, -1]$, pro slouce vychází $row_{col} = [0, -1, 1, -1]$.

Další funkce Maďarského algoritmu se nazývá *Augmentace*. Tato funkce se volá v každém kroku Maďarského algoritmu k rozšíření známého řešení o jeden vrchol.

Algorithm 4 Augmentace stromu daného kořenem *row_vertex*

```
1:  function HUNGARIAN_AUGMENT(row_vertex, Matice C, row_red, row_ass,
    col_ass, predecessors)
2:      min_labeled_cost ← [∞]_n
3:      labeled_cols ← labeled_unscanned_cols ← scanned_rows ←
    scanned_cols ← Ø
4:      unlabeled_cols ← n̂                          ▷ Indexová množina n̂
5:
6:      sink ← −1
7:      i ← row_vertex
8:      while sink < 0 do
9:          scanned_rows += i
10:         for all col ∈ unlabeled_cols do
11:             reduced_cost ← c_{i,col} − row_red[i] − col_red[col]
12:             if reduced_cost < min_labeled_cost[col] then
13:                 predecessors[col] ← i
14:                 min_labeled_cost[col] ← reduced_cost
15:                 if reduced_cost = 0 then
16:                     labeled_cols += col
17:                     labeled_unscanned_cols += col
18:                     unlabeled_cols −= col
19:                 end if
20:             end if
21:         end for
```

Nejprve algoritmus popisuje inicializaci proměnných. Důležitější je ovšem cyklus while. V něm dochází k detekci předchůdců pro jednotlivé sloupce a označení sloupce, pokud $\bar{c}_{i,j} = 0$.

```
22:         if labeled_unscanned_cols = ∅ then
23:             δ ← min(min_labeled_cols[i] : ∀i ∈ unlabeled_cols)
24:             for all row ∈ scanned_rows do
25:                 row_red += δ
26:             end for
27:             for all col ∈ scanned_cols do
28:                 col_red −= δ
29:             end for
30:             for all unlabeled ∈ unlabeled_cols do
31:                 min_labeled_cost[unlabeled] −= δ
32:                 if min_labeled_cost[unlabeled] = 0 then
33:                     labeled_cols += unlabeled
34:                     labeled_unscanned_cols += unlabeled
35:                     unlabeled_cols −= unlabeled
36:                 end if
37:             end for
38:         end if
39:         any_col ← col ∈ labeled_unscanned_cols          ▷ Náhodný prvek
40:         scanned_cols += any_col
41:         labeled_unscanned_cols −= any_col
42:         if row_ass[any_col] < 0 then
43:             sink ← any_col
44:         else
45:             i ← row_ass[any_col]
46:         end if
47:     end while
        return sink
48: end function
```

V druhé části funkce *Augment* se nejdříve zjistí, zda stále existují označené nezpracované sloupce. Pokud ne, přidají se. Na závěr cyklu while se vybere náhodný prvek z označených a ještě nezpracovaných prvků. Pokud zjistíme, že row_{ass} pro tento sloupec není ještě definován, funkce končí a vrátí tento sloupec. V opačném případě cyklus while pokračuje.

S předchozími dvěma funkcemi můžeme nyní představit celý algoritmus. Maďarský algoritmus nejprve zavolá funkci *lsap_preprocessing*, která vypočítá redukci řádků a sloupců, které transformují matici C na jednodušší matici $\bar{C}$ a částečné řešení. Poté pro každý nepřiřazený sloupec zavolá funkci *hungarian_augment*, která obohatí výsledný strom řešení o daný sloupec.

Algorithm 5 Maďarský algoritmus s komplexitou $\mathcal{O}(n^3)$

```
1: function HUNGARIAN_LSAP(Matice C)
2:     partial_results ← LSAP_PREPROCESSING(C)
3:     pred = [−1]_n                                   ▷ Pole předchůdců
4:     unassigned ← {∀i ∈ n̂ : col_ass[i] < 0}          ▷ Sloupce bez přiřazených
   řádků
5:     while |unassigned| > 0 do
6:         row_vertex ← row_vertex ∈ unassigned        ▷ Náhodný prvek
7:         unassigned −= row_vertex
8:         sink ← HUNGARIAN_AUGMENT(row_vertex, C, partial_results, pred)
9:         col ← sink
10:        repeat                  ▷ Projdi celý strom přes pole předchůdců
11:            row ← pred[col]
12:            partial_results.row_ass ← row
13:            tmp ← partial_results.col_ass[row]
14:            partial_results.col_ass[row] = col
15:            col ← tmp
16:        until row ≠ row_vertex
17:    end while
18:    return partial_results.row_ass
19: end function
```

3.3 Propojení vrcholů grafů

Důležitou částí teoretické přípravy je diskuze o tom, jakým způsobem propojit vrcholy prostorového grafu, aby byl zajištěn kompromis mezi propojeností grafu (čím větší propojenost, tím lepší můžeme získat výsledky) a složitostí (čím více hran mezi vrcholy, tím obecně multi-agentní algoritmy hledání cest mohou trvat déle na výpočet).

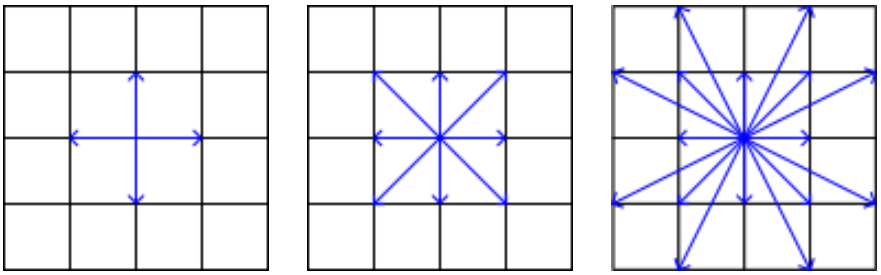
3.3.1 2^k sousedství

2^k sousedství [19] je způsob, jakým propojit vrcholy grafu v rovinné mřížce (viz obr. 3.5). Tento způsob zobecňuje klasické propojení s čtyřmi (obr. 3.5a) nebo osmi (obr. 3.6) sousedy. Sousedství $\mathcal{N}_{2^k}$ pro $k \geq 2$ je definováno pomocí kvadrantů $\mathcal{Q}_{k-2}$. Kvadrant $\mathcal{Q}_0$ je určen takto: $\mathcal{Q}_0 \coloneqq \langle(1,0),(0,1)\rangle$. Nechť pro $\forall i \in \mathbb{N}_0 : \mathcal{Q}_i = \langle a_0, a_1, ..., a_n\rangle$. $\mathcal{Q}_{i+1} \coloneqq \langle b_0, b_1, ..., b_{2n}\rangle$ je formálně zkonstruováno takto: $\forall j \in \hat{n}$:

$$b_{2j} \coloneqq a_j$$
$$b_{2j+1} \coloneqq a_j + a_{j+1}$$

Ukázka několika prvních kvadrantů:

$$\mathcal{Q}_0 = \langle (1,0),(0,1) \rangle$$
$$\mathcal{Q}_1 = \langle (1,0),(1,1),(0,1) \rangle$$
$$\mathcal{Q}_2 = \langle (1,0),(2,1),(1,1),(1,2),(0,1) \rangle$$
$$\mathcal{Q}_3 = \langle (1,0),(3,1),(2,1),(3,2),(1,1),(2,3),(1,2),(1,3),(0,1) \rangle$$



(a) 4 - sousedství (k = 2) (b) 8 - sousedství (k = 3) (c) 16 - sousedství (k = 4)

Obrázek 3.5: Různé hodnoty 2^k sousedství

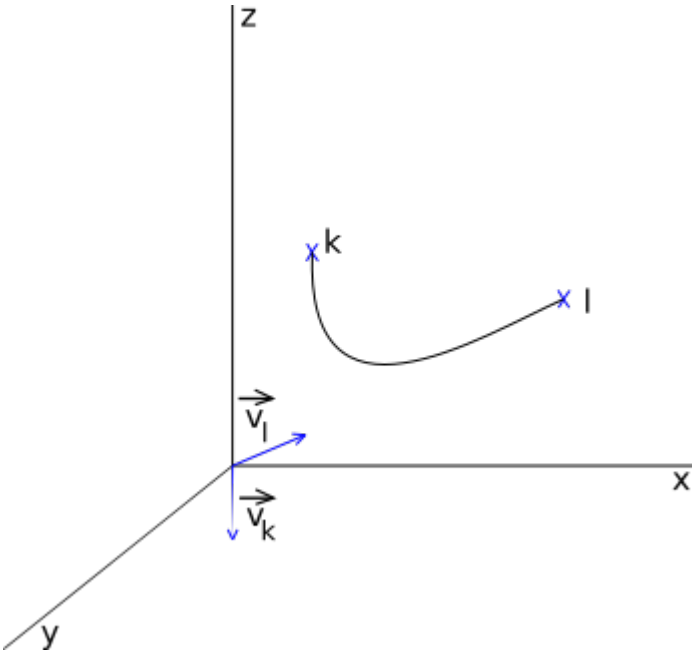
3.3.2 Interpolační křivky

Po částech polynomiální interpolační křivky [20] se často využívají při plánování dronových trajektorií [14] [3]. Specificky kubické splajnové křivky s parametrem t_n mezi body v prostoru k se směrovým vektorem $\vec{v_k}$ a l se směrovým vektorem $\vec{v_l}$ pro $\forall t \in \mathbb{R} : 0 \leq t \leq t_n$ s nulovými druhými derivacemi jsou pro i-tou souřadnici prostoru určeny takto:

$$x_i(t) = a_i + b_i t + c_i t^2 + d_i t^3 \tag{3.6}$$
$$a_i = k_i \tag{3.7}$$
$$b_i = v_{k_i} \tag{3.8}$$
$$c_i = 3\frac{l_i - k_i}{{t_n}^2} - \frac{2v_{k_i} + v_{l_i}}{t_n} \tag{3.9}$$
$$d_i = 2\frac{k_i - l_i}{{t_n}^3} + \frac{v_{k_i} + v_{l_i}}{{t_n}^2} \tag{3.10}$$

Daným předpisem jsou splněny také následující podmínky:

$$x_i(0) = k_i \tag{3.11}$$
$$x_i(t_n) = l_i \tag{3.12}$$
$${x_i}'(0) = v_{k_i} \tag{3.13}$$
$${x_i}'(t_n) = v_{l_i} \tag{3.14}$$
$${x_i}''(0) = {x_i}''(t_n) = 0 \tag{3.15}$$



Obrázek 3.6: Ukázka interpolační křivky mezi body k a l se směrovými vektory $\vec{v_k}$ a $\vec{v_l}$

Kapitola 4

Vlastní přínos

Předchozí kapitola se zabývá zejména rozborem již existující literatury na příbuzná témata. Oproti tomu v této kapitole již zadefinujeme problém dronového displeje, předvedeme zcela nové vylepšení algoritmu CBS a na závěr této kapitoly provedeme experimentální vyhodnocení tohoto algoritmu.

4.1 Definice problému

V této sekci popíšeme všechny nutné předpoklady, které potřebujeme k definici problému.

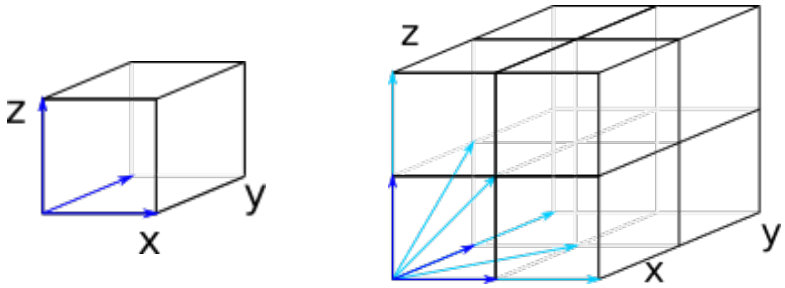
4.1.1 Zobecněné 2^k sousedství

Abychom mohli zadefinovat displej, potřebujeme nejprve zobecnit 2^k sousedství do 3-dimenzionálního prostoru. Stejně jako v případě 2^k sousedství nejprve zadefinujeme problém pro jeden kvadrant a poté zobecníme pro celý prostor.

Definice (Zobecněné 2^k sousedství)**.** *Definujme kvadrant $\mathcal{Q}_0$ jako set trojic $\mathcal{Q}_0 \coloneqq \{(1,0,0),(0,1,0),(0,0,1)\}$. Potom pro každé $i \in \mathbb{N}$ definujme kvadrant $\mathcal{Q}_i$ jako $\mathcal{Q}_i \coloneqq \{i \oplus j : \forall (i,j) \in \mathcal{Q}_{i-1} \times \mathcal{Q}_{i-1}\}$, kde operace $\oplus$ značí sčítání po složkách.*

Zobecněné sousedství $\mathcal{N}_k : k \in \mathbb{N} \wedge k \geq 2$ definujeme pomocí kvadrantů $\mathcal{Q}_{k-2}$, tedy $\mathcal{N}_k \coloneqq \bigcup_{m \in \mathbb{M}} m \otimes \mathcal{Q}_{k-2} : \mathbb{M} \coloneqq \bigcup_{(a,b,c)} : a,b,c \in \{1,-1\}$. Množinu $\mathbb{M}$ jsme zadefinovali jako všechny možné 3-složkové kombinace čísel 1 a -1. V tomto kontextu znamená operace $\otimes$, že první operand (trojice čísel) přenásobí po složkách všechny prvky množiny (druhý operand).

Ilustrujme si nyní, jakých hodnot nabývá několik prvních kvadrantů. Kvadranty $\mathcal{Q}_0$ 4.1a a $\mathcal{Q}_1$ 4.1b jsou pro lepší představu představeny na obrázcích.



(a) Zobecněné sousedství pro kvadrant $\mathcal{Q}_0$ (b) Zobecněné sousedství pro kvadrant $\mathcal{Q}_1$

Obrázek 4.1: Zobecněné 2^k sousedství

$$\mathcal{Q}_0 = \{(1,0,0),(0,1,0),(0,0,1)\}$$
$$\mathcal{Q}_1 = \{(1,0,0),(0,1,0),(0,0,1),(0,0,2),(0,2,0),(1,0,1),(1,1,0),(0,1,1),(2,0,0)\}$$
$$\mathcal{Q}_2 = \{(1,0,0),(0,1,0),(0,0,1),(0,0,2),(0,2,0),(1,0,1),(1,1,0),(0,1,1),(2,0,0),(0,3,0),(2,0,2),(0,2,1),(1,1,1),(0,0,4),(2,1,0),(1,0,3),(3,0,0),(0,1,2),(2,0,1),(1,2,0),(1,0,2),(0,0,3),(3,1,0),(0,2,2),(0,1,3),(3,0,1),(4,0,0),(1,1,2),(0,4,0),(1,2,1),(1,3,0),(0,3,1),(2,2,0),(2,1,1)\}$$

Jak je vidět, počet sousedů pro zobecněné 2^k sousedství roste rychle. Zmíníme, že pro $\mathcal{Q}_0$ máme 3 sousedy na kvadrant, v $\mathcal{Q}_1$ jich je 9, v $\mathcal{Q}_2$ 34. Další tři kvadranty $\mathcal{Q}_3$, $\mathcal{Q}_4$, $\mathcal{Q}_5$ mají 164, 968 a 6544 sousedů. Hlavním smyslem 2^k sousedství je najít rovnováhu mezi počtem sousedů každého grafu a rychlostí algoritmu. Větší sousedství obecně pomáhá nalézt kratší řešení, čímž potenciálně může přispět k rychlejšímu algoritmu s lepšími výsledky. Pokud je ale sousedů příliš mnoho, může se stát, že algoritmus musí projít velkým množstvím hran a to může algoritmus zpomalit. Proto v praktických případech je dobré zvolit menší k jako parameter. Vliv volby k na kvalitu a dobu běhu změříme v experimentální části této práce.

Dále zmíníme, že narozdíl od klasického 2^k sousedství (sekce 3.3.1) tato definice obsahuje i lineárně závislé souřadnice (například $(1,0,0)$ a $(2,0,0)$ v kvadrantu $\mathcal{Q}_1$). Pro definici tohoto problému nám to nevadí.

4.1.2 Displej

V tuto chvíli máme již zadefinovaný a vysvětlený způsob propojení vrcholů. Toho využijeme při následující definici displeje.

Definice (Displej)**.** *Displej je graf $D = (V, E)$, který definujeme pomocí trojice čísel $(l, w, h) \in \mathbb{N}^3$ (trojice značí délku, šírku a výšku), parametru $k \in \mathbb{N}$*

$\wedge\ k \geq 2$ *a parametru* d*. Trojice* (l, w, h) *značí celkovou délku, šířku a výšku displeje. Každý vrchol je jednoznačně určen pomocí souřadnic* $(x, y, z) \in \mathbb{N}^3 : x < l, y < w, z < h$*. Hrany mezi vrcholy jsou určeny pomocí zobecněného* 2^k *sousedství daného parametrem* k*. Vzdálenost mezi vrcholy, kde se právě jedna souřadnicová složka liší přesně o 1, je rovna* d*.*

Displej si můžeme představit jako $l \times w \times h$ krychliček v prostoru, kde délka strany krychličky je rovna d a vrcholy se nachází uprostřed těchto krychlí. Jinými slovy tyto krychličky představují prostorové pixely.

4.1.3 Obarvení vrcholů

Aby tyto pixely byly užitečné, měli bychom zajistit jejich obarvení. Toho dosáhneme pomocí několika následujících definic.

Definice (Rámec)**.** *Buď dán graf displeje* $D = (V, E)$*. Definujme funkci* $\varphi : V_n \to \mathcal{C}$ *kde* $V_n \subseteq V(D) : |V_n| = n$ *je podmnožina vrcholů displeje o velikosti* n*. Množina* $\mathcal{C} \coloneqq \mathbb{N} \bigcup \{\perp\}$ *obsahuje množinu všech přirozených čísel a znak* $\perp$*, který značí chybějící prvek. Tato množina* $\mathcal{C}$ *představuje množinu všech barev. Této funkci říkáme rámec displeje* D*.*

Jelikož jsou pixely v našem displeji reprezentovány pomocí dronů (alias agentů), hodí se nám definovat funkci pro přiřazení agentů k pixelům. Všimneme si, že v naší definici nepožadujeme striktně přiřazení všech agentů (až na nějaké výjimky, které vyjasníme později).

Definice (Přiřazení agentů k pixelům)**.** *Nechť je dána množina agentů* $A = (a_0, a_1, a_2, ..., a_{n-1})$ *a nechť je dán rámec displeje* $\varphi : V_n \to \mathcal{C}$*. Dále mějme graf displeje* $D = (V, E)$*. Dále mějme dánu podmnožinu agentů* $\mathcal{A} \subseteq A$*. Potom definujme funkci přiřazení agentů k pixelům jako* $\psi : \mathcal{A} \to V_n$*.*

Nás především zajímá, jak se pixely mění v čase. K tomuto účelu zadefinujeme tzv. *sekvenci rámců*, která nám říká, jak se mění umístění pixelů a dronů v čase.

Definice (Sekvence rámců)**.** *Nechť je dán graf displeje* $D = (V, E)$ *a množina agentů* A *o velikosti* $|A| = n$ *a nechť je dáno* m *rámců a funkcí přiřazujících agenty k pixelům. Potom sekvenci dvojic* $F \coloneqq \langle(\varphi_0, \psi_0), (\varphi_1, \psi_1), ..., (\varphi_{m-1}, \psi_{m-1})\rangle$ *nazýváme sekvencí rámců při splnění těchto podmínek:*

$$\forall i \in \hat{m} : F_i = (\varphi_i : V_{n,i} \to \mathcal{C}, \psi_i : \mathcal{A}_i \to V_{n,i}) : V_{n,i} \subseteq V(D) \wedge |V_{n,i}| = n, \mathcal{A}_i \subseteq A \tag{4.1}$$

$$\forall i \in \{0, m-1\} : V_{n,i} \subseteq \{V(D) : (x, y, 0) \in V(D)\} \tag{4.2}$$

$$|\mathcal{A}_0| = |A| = n \tag{4.3}$$

Rámec F_0 *nazýváme iniciální rámec a rámec* F_{m-1} *nazýváme finální rámec.*

Rozeberme nyní jednotlivé podmínky. Podmínka 4.1 pouze upřesňuje, jak vypadají jednotlivé rámce F_i a definuje, že rámec φ_i a přiřazení ψ_i používají stejnou podmnožinu vrcholů $V_{n,i}$.

Další podmínka 4.2 značí, že pro iniciální a finální umístění se všichni agenti musí vměstnat do vrcholů s nulovou z-souřadnicí. Tento požadavek je uzpůsobený tomu, že používáme drony jako agenty, které musejí vzlétnout z podlahy a na konci přistát. Tato podmínka nám tak nepřímo dává horní limit na počet agentů: $n \leq w \cdot l$. Nutno podotknout, že pokud bychom chtěli použít menší displej s více drony než je možné, dá se tento požadavek obejít tím, že zajistíme větší prostor na podlaze (neboli zvětšíme délku nebo šířku displeje) a původní chtěný displej bude ve skutečnosti podčást tohoto většího displeje.

Poslední podmínka 4.3 požaduje, aby počáteční umístění zahrnovalo všechny agenty, a tím pádem abychom mohli rozlišit jednotlivé drony. Mimoto dále zmíníme, že v praktickém případě je dobré nastavit $|\mathcal{A}_i| = 0 : i > 0$. Tím de facto zaručíme, že tím budeme řešit nerozlišenou verzi MAPF instance a necháme algoritmus sám se rozhodnout, jakým způsobem nejlépe určit přiřazení vzhledem k celkové vzdálenosti.

4.1.4 MAPF problém ve spojitém čase a prostoru

Poslední věc, kterou potřebujeme vyřešit před definicí problému dronového vzdušného displeje je úprava definice MAPF pro spojitý čas a prostor. Ačkoli problém multi-agentního vyhledávání cest byl již definován jinými autory pro spojitý čas [10], je naše úprava specifická, a proto ji definujeme v této kapitole.

Definice (MAPF problém ve spojitém čase a prostoru)**.** *Nechť je dán graf prostředí $G = (V, E)$, vzdálenostní funkce $\lambda : E(G) \to \mathbb{R}^+$ a nechť existují zobrazení $\gamma_v : V(G) \to \mathbb{R}^3$ a $\gamma_e : E(G) \to \mathbb{R}^3$, která mapují vrcholy a hrany grafu do 3-rozměrného prostoru. Dále nechť je dána množina agentů $A = \{a_0, a_1, ..., a_n\}$ a trojice (r_l, r_w, r_h), které značí poloměry ochranného elipsoidu kolem každého agenta. Nechť je dáno číslo α_{max}, které značí maximální akceleraci agenta a nechť je dána maximální rychlost $v_{max} \in R$. Nechť je dáno prosté počáteční umístění agentů $\varphi_s : A \to V$ a prosté finální umístění $\varphi_f : A \to V$. Označme $v_{i,u} \in \mathbb{R}$ jako vstupní rychlost do vrcholu $u \in V(G)$ pro agenta $a_i : i \in \hat{n}$.*

Každý agent a_i může vykonávat tyto aktivity:

1. *Čekat po určitou dobu v libovolném vrcholu u, pokud vstupní rychlost je nulová $v_{i,u} = 0$.*
2. *Akcelerovat nebo brzdit s akcelerací $\alpha \in \langle -\alpha_{max}, \alpha_{max} \rangle$.*
3. *Pokračovat po trajektorii konstantní rychlostí v.*

Označme $\mathcal{E}_i = (e_0, e_1, ..., e_{k-1})$ jako sekvenci akcí, kterou vykonává agent a_i a nechť pro každou akci $e_j : j \in \hat{k}$ existuje její časová složka, kterou označíme $e_j.time$ a pro kterou platí: $e_j.time \in \mathbb{R}^+$. Dále označme $\mathcal{P}_i = (v_0, v_1, ..., v_{l-1}) : \forall j \in \hat{l} : v_j \in V(G) \wedge \forall j \in \hat{l} \backslash \{l-1\} : (v_j, v_{j+1}) \in E(G)$ jako trajektorii agenta a_i. Funkce $\gamma_a : (t, \mathcal{P}_i, \mathcal{E}_i) \rightarrow (x, y, z) \in \mathbb{R}^3 : \frac{x^2}{{r_l}^2} + \frac{y^2}{{r_w}^2} + \frac{z^2}{{r_h}^2} = 1$, kde $t \in {\mathbb{R}_0}^+$ značí okupaci prostoru agentem a_i v čase t. Označme $v_{i,t}$ jako okamžitou rychlost agenta v čase t. Definujme počáteční rychlost agentů $v_{i,0} \coloneqq 0 : i \in \hat{n}$.

Problémem multi-agentního hledání ve spojitém čase a prostoru cest je najít umístění agentů $(\varphi_s = \varphi_0, \varphi_1, \varphi_2, ..., \varphi_m = \varphi_f)$ (které se dá transformovat na množinu trajektorií jednotlivých agentů $\Phi = \{\mathcal{P}_0, \mathcal{P}_1, ..., \mathcal{P}_{n-1}\}$), množinu sekvencí akcí pro každého agenta $\sigma = \{\mathcal{E}_0, \mathcal{E}_1, ..., \mathcal{E}_{n-1}\}$ a čas $\tau \coloneqq max_{\forall i \in \hat{n}}(\sum_{\forall e_j \in \mathcal{E}_i} e_j.time)$, za splnění podmínek:

$$\forall t \in \langle 0, \tau \rangle, \forall i, j \in \hat{n} : \gamma_a(t, \mathcal{P}_i, \mathcal{E}_i) \bigcap \gamma_a(t, \mathcal{P}_j, \mathcal{E}_j) \neq \emptyset \Rightarrow i = j \tag{4.4}$$

$$\forall i \in \hat{n} : v_{i,\tau} = 0 \tag{4.5}$$

$$\forall i \in \hat{n}, \forall t \in \langle 0, \tau \rangle : 0 \leq v_{i,t} \leq v_{max} \tag{4.6}$$

Jinými slovy jde o nalezení cest pro vícero agentů, když bereme v potaz jejich spojitý čas, velikost agentů a nutnost zrychlení či zpomalení při změně rychlosti agentů.

Popíšeme dopodrobna podmínky, které musí problém multi-agentního vyhledávání ve spojitém čase a prostoru splňovat. Podmínka 4.4 zajišťuje, že žádní dva různí agenti ve stejnou dobu neokupují stejnou část prostoru, a tedy nedochází ke srážkám. Další podmínka 4.5 je podmínka zastavující, která nutí agenty, aby ve své finální pozici setrvali a měli nulovou rychlost. Poslední podmínka 4.6 znemožňuje agentům překročit nejvyšší povolenou rychlost.

4.1.5 Problém dronového vzdušného displeje

Nyní máme všechny potřebné informace k tomu, abychom mohli zadefinovat problém displeje.

Definice (Problém dronového displeje)**.** *Nechť je dán graf displeje $D = (V, E)$, parameter d značící vzdálenost sousedících vrcholů, množina agentů (dronů) $A = \{a_0, a_1, ..., a_{n-1}\}$ a sekvence rámců $F = \langle f_0, f_1, ..., f_{m-1} \rangle$. Dále budiž dány vlastnosti dronů - číslo α_{max}, které značí nejvyšší možnou akceleraci, a trojice $(r_l, r_w, r_h) \in \mathbb{N}^3 \wedge \forall r \in \{r_l, r_w, r_h\} : r \leq \frac{d}{2}$, která označuje poloměr agentů (délka, šířka a výška). Buď dán časový parametr prodlevy $\tau \in \mathbb{R}^+$ (počet sekund), který značí, jak dlouho mají drony zůstat na pozici.*

Problém dronového displeje je následující:

1. *Najít pro následné dvojice rámců $(f_0, f_1), (f_1, f_2), ..., (f_{m-2}, f_{m-1})$ řešení MAPF instance ve spojitém čase a prostoru (které reflektují přiřazená*

místa agentů k pixelům, pokud je přiřazení dáno) a časy $t_0, t_1, ..., t_{m-2}$, kde $\forall i \in (0, 1, ..., m-2)$ je t_i celkový čas řešení MAPF instance pro dvojici rámců (f_i, f_{i+1}).

2. *Určit pro každou dvojici $\forall i \in (0, 1, ..., m-2) : (f_i, f_{i+1})$ funkci $\rho_i : (A, t) \to \mathcal{C}$ pro $t \in (0, t_i), \mathcal{C} \in \mathbb{N} \bigcup \{\perp\}$.*

3. *Z předchozích dvou výsledků určit pro každého agenta časový plán, který reflektuje nalezená řešení a po dosažení každého rámce vyčká po dobu τ na místě. Během doby čekání agent barvu nemění a barva odpovídá požadované barvě dané zadáním pro konkrétní rámec a pozici pixelu, který v daný okamžik agent představuje.*

Aby bylo možné si tento problém lépe představit, připravili jsme opět obrázky s příkladem. Obrázek 4.2a ukazuje možné zadání úlohy uživatelem. Displej začíná rámcem f_0, kde máme určeného žlutého agenta a červeného agenta, kteří mají nulovou z-tovou souřadnici. Rámce f_1 a f_2 již mohou využívat pixely z celého displeje a není potřeba manuálně určovat pozice agentů. Rámec f_3 je finální, agenti musí opět skončit na nulové z-tové souřadnici.

Ilustrace řešení problému dronového displeje se nachází v obrázku 4.2b. Na něm vidíme, že algoritmus našel řešení a agentům a_0 a a_1 přiřadil pixely a našel bezkolizní cestu. Během přesunu se barva agenta postupně mění. Agenti po dané trajektorii i zrychlují a zpomalují - toto ovšem není na ilustraci zachyceno.

4.2 Řešení problému

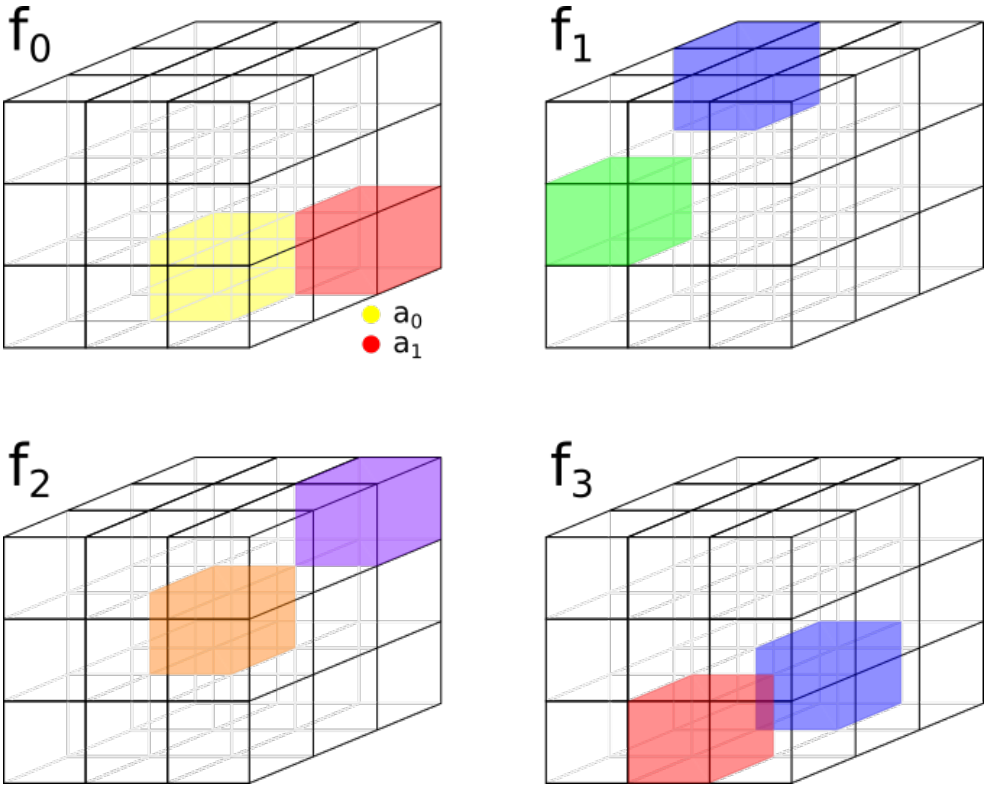
Problém dronového displeje byl již zadán, zbývá již jen ho vyřešit. Proto v této sekci představíme nový algoritmus, který je založený na Conflict Based Search algoritmu. Náš algoritmus má více abstraktních vrstev, které popíšeme shora dolů. Tato sekce popisuje zejména část problému, která řeší výpočet cesty mezi rámci. Nalezení barevného přechodu a individuálního plánu pro agenta popisujeme zejména v následující kapitole o vizualizaci dronového displeje.

4.2.1 Algoritmus dronového displeje - nejvyšší úroveň

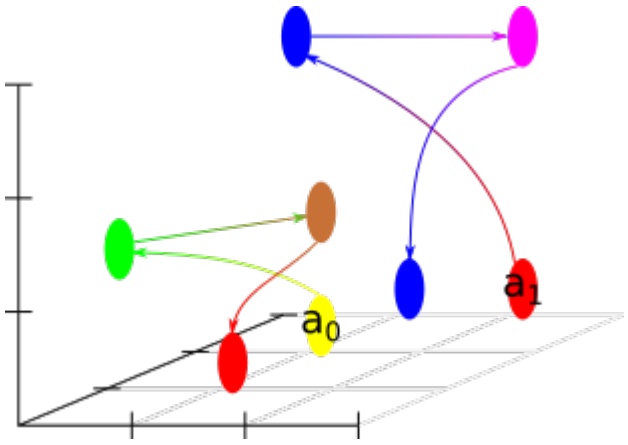
Nejvyšší vrstva algoritmu je jednoduchá a zabývá se především převáděním nerozlišených instancí multi-agentního vyhledávání cest na rozlišené a opakovaným spouštěním algoritmu hledáním cest pro každý rámec.

Algoritmus je popsaný ve funkci *display_solve* 6. Na vstupu je dána definice displeje, rámce s pixely, sada agentů. Ostatní parametry jako rozměry agentů, maximální rychlost i akcelerace a jiné jsme označili pod souhrnným názvem *display_constraints*.

Pokud není určeno finální umístění agentů pro jednotlivé rámce, tak během iterací voláme maďarský algoritmus na určení chybějících pozic. Jak už jsme



(a) Tento obrázek popisuje možné uživatelské zadání dronového displeje pro $F = (f_0, f_1, f_2, f_3)$ a agenty $A = (a_0, a_1)$. Rozměry displeje jsou $3 \times 3 \times 3$. V počátečním rámci jsou agenti určeni jednoznačně, v dalších rámcích je určení agentů nepovinné, v tomto případě úplně chybí.



(b) Na obrázku popisujeme řešení instance problému dronového displeje dané obrázkem 4.2a pro agenty a_0 a a_1. Algoritmus našel přiřazení pixelů jednotlivým agentům a cestu mezi těmito pixely. Při pohybu se barva postupně mění. Akceleraci agentů tento obrázek nezobrazuje.

Obrázek 4.2: Problém dronového displeje

zmínili v předchozí kapitole, tento postup nám umožňuje aproximativně převést nerozlišený problém MAPF na rozlišený. Díky tomu umíme kombinovat nejen rozlišené a nerozlišené hledání v rámci jedné instance (nebo dokonce rámce) problému dronového displeje, ale též můžeme využít nové varianty algoritmu CBS popsané v dalších sekcích.

Algorithm 6 Algoritmus řešící problém dronového displeje

```
1: function DISPLAY_SOLVE(displej D = (V, E), rámce F, agenti A, display_constraints)
2:     res ← []
3:     ψ ← ψ_0                                   ▷ Počáteční přiřazení agentů
4:     for i ← 1 to n − 1 do
5:         A_m ← A\A_i                           ▷ Nepřiřazení agenti
6:         if A_m ≠ ∅ then                       ▷ Vypočte přiřazení agentů
7:             C = [0]_[|A_m|×|A_m|]
8:             row ← 0
9:             for all v_I ∈ ψA_m do
10:                col ← 0
11:                for all v_F ∈ V_{i,n}\ψ_i(A) do
12:                    C_{row,col} ← EUCLIDIAN_DISTANCE(v_I, v_F)
13:                    col ← col + 1
14:                end for
15:                row ← row + 1
16:            end for
17:            row_ass ← HUNGARIAN_LSAP(C)
18:            ψ_next ← MERGE_ASSIGNMENTS(ψ_i, row_ass)  ▷ Vytvoří přiřazení všech agentů k dalším pixelům
19:        else
20:            ψ_next ← ψ_i
21:        end if
22:        res.append(CSPT_CBS(D, display_constraints, ψ, ψ_next))
23:        ψ ← ψ_next
24:    end for
25:    return res
26: end function
```

4.2.2 Continuous Spacetime Conflict Based Search

Od této sekce dál popisujeme de facto novou variantu algoritmu CBS, kterou nazýváme anglicky *Continuous Spacetime Conflict Based Search*, český překlad je Hledání pomocí konfliktů ve spojitém času a prostoru. Zkratkou jej nazýváme CSPT_CBS.

Stejně jako algoritmus CBS má i varianta CSPT_CBS úroveň algoritmu, která pracuje na abstraktnější úrovni. V rámci teoretické přípravy této části říkáme vysokoúrovňová 2. Tato část je totožná s originální verzí CBS, proto se již na ní pouze odkážeme.

4.2.3 Hledání konfliktů

V klasickém CBS algoritmu probíhá detekce konfliktů tak, že algoritmus projde po krocích cesty jednotlivých agentů a porovná, zda ve stejném kroku 2 agenti nejsou přítomni ve stejném vrcholu (případně hraně, pokud toto zadání vyžaduje). Ve spojitém CSPT_CBS algoritmu musíme využít jiný způsob. Hlavní finta spočívá v tom, že pro účely hledání konfliktů diskretizujeme spojitý prostor displeje na 3-dimenzionální mřížku. Tato mřížka se skládá z krychlí a vrcholy displeje se nachází uprostřed každé této krychle.

Při hledání konfliktů algoritmus prochází trajektorie jednotlivých agentů. Nejprve určíme, jakými krychlemi trajektorie agenta prochází a také si držíme informaci, v jakém časovém intervalu tyto krychle okupuje. Algoritmus pro každý vrchol drží setříděnou mapu výskytů těchto časových intervalů. Pokud dojde k tomu, že se dvěma různým agentům protne časový interval v nějaké krychli, dostaneme kandidáta na konflikt.

Přesné hledání konfliktů je možné, ale není kritické. Podle autorů podobného algoritmu CCBS [10] trvá detekce konfliktů signifikantní porci času. Pro jednodušší implementaci a rychlejší algoritmus jsme si oproti problému dronového displeje dovolili udělat tato zjednodušení:

1. Místo ochranného elipsoidu obklopujeme agenty ochranným kvádrem se stejnými poloměry stran jako u požadovaných elipsoidů. Tyto kvádry využíváme pro detekci průniku s mřížkovými krychlemi.

2. Kandidáta na konflikt rovnou považujeme za konflikt, i když konfliktem být nemusí.

Upřesnění hledání konfliktů a hledání vztahů mezi lepší kvalitou výsledku a délkou výpočtu je předmětem dalšího zkoumání. Implementace přesnější detekce je možná buď analyticky, kde je nutné brát v potaz pozice, rychlost, zrychlení a čas, anebo přibližně pomocí numerických metod.

Jakmile dojde k nalezení konfliktu, stejně jako v případě CBS jej přidáme do množiny konfliktů a zapamatujeme si, v jakém časovém intervalu, v jaké části mřížky a mezi kterými agenty došlo ke srážce.

Celý algoritmus je popsán níže (viz 7).

4.2.4 Hledání cest

Další modifikovanou částí algoritmu je hledání cest. Protože tato modifikace je komplikovanější, podrozdělíme si tuto sekci na 2 části, kde první část se zabývá

Algorithm 7 CSPT_CBS detekce konfliktů

1: **function** CSPT_FIND_FIRST_CONFLICT(displej $D = (V, E)$, agenti A, trajektorie agentů $\mathcal{P}$, akce agentů $\mathcal{E}$)
2: $conflict\_map \leftarrow \{\}$
3: **for all** $p_i \in \mathcal{P}$ **do** ▷ Pro každou část trajektorie p_i agenta a_i
4: $node\_map \leftarrow conflict\_map[p_i.coordinates]$
5: (I, a_c) ←FIND_INTERSECTING_TIME_INTERVAL($node\_map$, $p_i.time\_interval$) ▷ Najdi protínající interval a agenta, se kterým se tento interval protíná
6: **if** $I = \emptyset$ **then**
7: $node\_map[p_i.time\_interval] \leftarrow a_i$
8: **else**
9: **return** (a_i, a_c, I, $p_i.coordinates$) ▷ Konflikt mezi agenty a_i, a_c v časové intervalu I v krychli o souřadnicích $p_i.coordinates$
10: **end if**
11: **end for**
12: **end function**

zejména modifikací trajektorií mezi sousedními vrcholy vzhledem k pohybu agenta a ta druhá se víc zabývá samotným algoritmem.

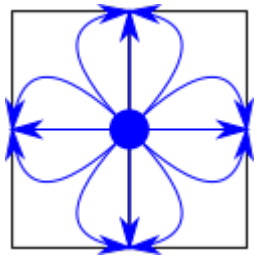
4.2.4.1 Vliv pohybu agenta na dostupné sousedy

V našem algoritmu jsme zakomponovali vliv momenta agenta na trajektorie mezi sousedními vrcholy. Tento vliv je namodelován velmi zjednodušeně, a to tak, že nastává právě jeden ze dvou případů, a to v závislosti na vstupní rychlosti v_{Iu} agenta a na vstupním směru $\vec{s_u}$ do nějakého vrcholu u.

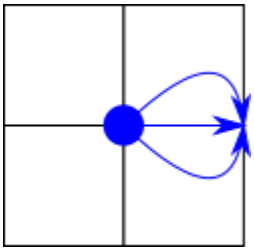
- Pokud vstupní rychlost v_{Iu} je nulová (agent čeká) (viz ilustrace 4.3a), agent si může vybrat, jakého souseda w navštíví v příštím tahu a z jakého směru $\vec{s_w}$ dorazí do vrcholu w.
- Pokud je vstupní rychlost v_{Iu} kladná (viz ilustrace 4.3b), nemá agent na vybranou a musí pokračovat ve zvoleném směru. Může si ovšem vybrat, z jakého směru $\vec{s_w}$ dorazí do vrcholu w.

Další úprava ohledně křivky trajektorie závisí na vstupní rychlosti v_{Iw} do následujícího vrcholu w.

- Jestli brzdíme na nulovou rychlost v_I z vrcholu u do vrcholu w, tak se agent pohybuje po přímce.
- Naopak když agent pokračuje z vrcholu w dál, je trajektorie dána po částech polynomiální interpolační křivkou (viz sekce 3.3.2). Parametr t_n je



(a) Možné volby sousedů při nulové rychlosti $v_{I\,u}$.



(b) Možné volby trajektorie při nenulové rychlosti $v_{I\,u}$.

Obrázek 4.3: Vliv vstupní rychlosti na trajektorie agentů a dostupnost sousedů pro vrchol u a vstupní směr $\vec{s_u} = (1, 0)$. Pro zjednodušení jsme pohyb ilustrovali v 2-dimenzionální mřížce.

dán uživatelem na vstupu. V této práci jej označujeme pod názvem *curvature*. Tato metoda nám vrátí předpis křivky v prostoru. My však potřebujeme znát i její délku – k tomu se hodí další uživatelem definovaný parametr zvaný *precision*. Ten říká, na kolik částí máme křivku rozdělit, poté délku křivky aproximujeme sumou vzdáleností mezi jednotlivými body.

4.2.4.2 Algoritmus hledání cest v CSPT_CBS

V této sekci popíšeme pseudokód algoritmu hledání cest (viz algoritmus 8). Tento kód je sice založený na algoritmu A*, jinak se ale dramaticky liší. Co mají zcela jistě společné, je existence heuristické funkce $h(node)$, u které platí, že cena cesty z vrcholu $node$ je alespoň $h(node)$. My definujeme funkci $h(node) := \varepsilon * \frac{||node.next||}{v_{max}} + \varepsilon * \frac{goal\_distance}{v_{max}}$, kde $\varepsilon \in {\mathbb{R}_0}^+$ je inflační parametr, $||node.next|| \in {\mathbb{R}_0}^+$ je vzdálenost příštího vrcholu od současného (ten v našem algoritmu známe dopředu, viz předchozí sekce), $goal\_distance \in {\mathbb{R}_0}^+$ je euklidovská vzdálenost následujícího vrcholu k cílovému.

Algoritmus prochází jednotlivé nody v hlavním cyklu while a v každé iteraci vybere z fronty ten, kde součet aktuální ceny a odhadované ceny je nejmenší. Na začátku každé iterace se zavolá funkce *find_neighborhood*. Ta najde sousední vrcholy v závislosti na momentu a aktuální rychlosti. Mechanismus výběru sousedů jsme si již nastínili v předchozí sekci.

Algoritmus dále pokračuje for cyklem, kde se nejprve snažíme najít zrychlení po dané trajektorii, aby agent dorazil do následujícího vrcholu co nejrychleji. Tím se zabývá funkce *adjust_acceleration*. Ta předpokládá, že agent se pokusí zrychlit až na nejvyšší možnou rychlost a pokud brzdíme, tak ke konci zastaví na nulu. Pokud dojde ke kolizi, pokusí se zpomalit. Když by došlo ke kolizi agentů na stejném místě, algoritmus snahu ukončí a vrátí konflikt. Dále

stojí za zmínku, že tato funkce má specifický kus kódu, pokud řešíme i zastavení. Zde uvedeme příkladem, že se může stát, že kdyby agent akceleroval moc dlouho, nebude mít dostatečně dlouhou brzdnou dráhu. To řešíme tak, že spočteme maximální délku akcelerace, aby byla brzdná dráha dostatečně dlouhá k zastavení. Veškeré vzorečky v této funkci jsou odvozeny z Newtonových zákonů akcelerace.

Hlavní smyčka dále pokračuje řešením nastalého konfliktu, na který reagujeme zejména tehdy, když se agent teprve začíná pohybovat z nulové rychlosti - tehdy totiž můžeme agentovi nakázat, aby ještě chvíli počkal na místě. V opačném případě daný node zahodíme a nebudeme se jím již dále zabývat. Abychom neměli příliš mnoho nodů ve frontě, které čekají ve stejném vrcholu, děláme ještě to, že vždy vybereme pouze ten nejkratší čas (viz řádky 19-24 v algoritmu). Nody, ve kterých agenti čekají, přidáme do fronty až pouze po zpracování všech sousedů současného nodu.

Nakonec přichází na řadu ošetření finálního vrcholu a přidání dalšího vrcholu do fronty. V klasickém A* algoritmu se díváme, zda již do vrcholu existuje nějaká cesta a pokud ano, tak node do fronty nepřidáváme. V našem algoritmu jsme tuto podmínku trochu upravili – přidali jsme výjimku, že když jsme souseda nalezli v této iteraci, můžeme jej přidat do fronty znova. To je proto, že tímto způsobem můžeme do sousedního vrcholu dorazit po jiné trajektorii a v dalších iteracích algoritmu máme různé možnosti toho, kudy se dál vydat.

4.2.5 Dávková heuristika

Během vzletu a přistání dochází k tomu, že větší množství agentů, kteří jsou jinak rozprostření v prostoru, se může nacházet ve stěsnaném prostoru. To pak může ovlivňovat výkon algoritmu kvůli velkému množství kolidujících agentů. Abychom zmírnili tento problém, naimplementovali jsme tzv. dávkovou heuristiku.

Tato heuristika, kterou značíme h_b, spočívá v tom, že během těchto kritických přechodů spustíme najednou menší množství agentů. Dávková heuristika je parametrizovaná pomocí parametrů $h_{bc} \in \mathbb{N}$, $h_{b\Delta t} \in \mathbb{R}^+$ a $h_{bd} \in \mathbb{R}^+$. Takto popořadě tyto parametry ovlivňují maximální počet agentů v dávce, zpoždění dávky a minimální povolenou vzdálenost mezi agenty u startovních a cílových pozic.

Řekněme také, že pro displej $D = (V, E)$ je $\mathcal{S} : (u, v) \to O$, kde $u, v \in V(D)$ a $O \in \{true, false\}$ je řadící funkcí. Označme pro libovolný vrchol $w \in V(D)$ trojici (z_w, y_w, x_w), která reprezentuje výškovou, šířkovou a délkovou souřadnici vrcholu w. Pro vzlet volíme řadící funkci $\mathcal{S} \coloneqq (z_u, y_u, x_u) < (z_v, y_v, x_v)$ (odshora dolu), pro přistání naopak $\mathcal{S} \coloneqq (z_u, y_u, x_u) > (z_v, y_v, x_v)$ (zdola nahoru).

Nyní v bodech zhruba představíme postup heuristiky:

Algorithm 8 CSPT_CBS hledání cest

```
1: function CSPT_FIND_FIRST_PATH(displej D = (V, E), počáteční umístění
   φ_i, finální umístění φ_f, konfliktní set C)
2:     queue ← {node(φ_i)}                ▷ Seřazeno podle node.cost + h(node)
3:     visited ← {φ_i}
4:     while queue ≠ ∅ do
5:         node ← queue.pop()
6:         new_visited ← {}
7:         wait_map ← {}
8:         neighborhood ← FIND_NEIGHBORHOOD(node)
9:
10:        for all neighbor ∈ neighborhood do
11:            neigh_node ← node(neighbor.coord)
12:            neigh_node.prev ← node
13:            conflict ← ADJUST_ACCELERATION(node, neighbor,
   neigh_node, C)
14:
15:            if conflict ≠ ∅ then                ▷ Řešení konfliktu
16:                if node.v_I = 0 ∧ conflict.Δt < ∞ ∧ node.coord ≠ φ_i then
17:                    wait_node ← wait_map[neighbor.coord]
18:                    time ← node.cost + conflict.Δt
19:                    if wait_node = ∅ ∨ time < wait_node.cost then
20:                        wait_node ← node(node.coord)
21:                        wait_node.prev ← node
22:                        wait_node.cost ← time
23:                        wait_map[neighbor.coord] ← wait_node
24:                    end if
25:                end if
26:                continue
27:            end if
28:
29:            if neighbor = φ_f then              ▷ Dorazili jsme do cíle
30:                if neighbor.stop then
31:                    return neigh_node
32:                end if
33:                continue
34:            end if
```

```
35:             if visited[neighbor.coord] = ∅ ∨ new_visited[neighbor.coord] ≠ ∅ then   ▷ Přidání nodu do fronty
36:                 visited.insert(neighbor.coord)
37:                 new_visited.insert(neighbor.coord)
38:                 queue.insert(neigh_node)
39:             end if
40:         end for
41:
42:         for all (_, node) ∈ wait_map do          ▷ Čekání přidáme do fronty
43:             queue.insert(node)
44:         end for
45:     end while
46: end function
47:
48: function ADJUST_ACCELERATION(node, neighbor, neighbor_node, conflict_set)
49:     neigh_node.acc ← (0, α_max)                  ▷ Dvojice doba trvání a hodnota
50:     neigh_node.deacc ← (0, −α_max)
51:     if neighbor.stop then
52:         neigh_node.v_I ← 0
53:     else
54:         neigh_node.v_I ← v_max
55:     end if
56:     rest_speed ← v_max                           ▷ Rychlost po skončení akcelerace
57:     conflict_coords ← ∅
58:     repeat
59:         neigh_node.acc.time ← (rest_speed − node.v_I) / acc.acceleration
60:         if neighbor_node.stop then
61:             neigh_node.deacc.time ← − rest_speed / neigh_node.deacc.acceleleration
62:             acc_dist ← ACC_DIST(neigh_node.acc, node.v_I)
63:             deacc_dist ← ACC_DIST(neigh_node.deacc, rest_speed)
64:             rest_time ← (neighbor.dist − acc_dist − deacc_dist) / rest_speed
65:             if rest_time < 0 then                ▷ Není dostatek času k zrychlení
66:                 neigh_node.acc.time ← ACC_DURATION(node, neighbor, neigh_node)
67:                 rest_speed ← neigh_node.acc.time · neigh_node.acc.acceleration + node.v_I
68:             end if
69:         end if
```

```
70:         trajectory ← FIND_TRAJECTORY(node, neighbor, neigh_node)
71:         conflict ← ∅
72:
73:         for all box ∈ trajectory do
74:             new_conflict ← FIND_CONFLICT_DURATION(conflict_set,
    box.coord, box.time_interval)
75:             if new_conflict.duration > conflict.duration then
76:                 conflict ← new_conflict
77:             end if
78:         end for
79:
80:         if conflict ≠ ∅ then                          ▷ Úprava akcelerace
81:             new_conflict ← conflict_coords.contains(conflict.coord)
82:             conflict_coords.insert(conflict.coord)
83:             conflict.Δt ← conflict.time_interval.to − node.cost
84:             neigh_node.acc.acceleleration ← 2·(conflict.Δt−node.v_I·conflict.Δt) / conflict.Δt²
85:             rest_speed ← neigh_node.acc.acceleleration · conflict.Δt +
    node.v_I
86:         end if
87:     until conflict ≠ ∅ ∧ new_conflict
88:     return conflict
89: end function
90:
91: function ACC_DIST(acc, v_I)
92:     return 1/2 · (acc.time² · acc.acceleleration + 2 · v_I · acc.time)
93: end function
94:
95: function ACC_DURATION(node, neighbor, neighbor_node)
96:     acc ← node.acc.acceleleration
97:     deacc ← node.deacc.acceleleration
98:     a ← acc · deacc − acc²
99:     b ← 2 · node.v_I · acc2 − 2 · node.v_I · acc1
100:    c ← −node.v_I² − 2 · neighbor.dist · deacc
101:    D ← b² − 4 · a · c
102:    return max{(−b±√D)/(2·a)}
103: end function
```

1. Setřídíme agenty podle jejich cíle φ_f a řadící funkce $\mathcal{S}$.
2. Dokud máme agenty s nepřiřazeným zdržením, tak pro $i = 0, 1, 2, ...$ opakuj následující kroky pro dávky indexované číslem i.
   a) Dokud je aktuální dávka nenaplněná, projdi zbývající agenty.
      i. Zkontroluj, zda euklidovská vzdálenost aktuálně procházeného agenta od všech ostatních v dávce je při startu a v cíli větší než h_{bd}.
      ii. Pokud ne, přeskoč aktuálního agenta.
      iii. Přiřaď agentovi zdržení $i \cdot h_{b\Delta t}$.

4.3 Experimentální vyhodnocení

Výše zmíněný algoritmus jsme implementovali v jazyce C++, který jsme otestovali na datech vygenerovaných náhodným generátorem. Algoritmus je dostupný na fakultní stránce gitlabu [21]. Vygenerovaná data jsou ve formátu CFG, algoritmus tento formát umí zpracovat a obohatí vygenerovaný konfigurační soubor o výsledky, které lze zobrazit ve vizualizačním programu. Ke konfiguraci algoritmu je potřeba znát velké množství možných parametrů. Než přejdeme k samotnému měření, představíme si jejich kompletní seznam.

4.3.1 Parametry algoritmu a náhodného generátoru

Nyní představíme všechny parametry algoritmu:

- **length** – představuje počet pixelů, které se vejdou do displeje na délku.
- **width** – představuje počet pixelů, které se vejdou do displeje na šířku.
- **height** – představuje počet pixelů, které se vejdou do displeje na výšku.
- **k** – je parametr do algoritmu zobecněného 2^k sousedství, které určuje míru propojení jednotlivých pixelů. Požadujeme, aby parametr $k \geq 2$.
- **precision** – je parametr pro numerický výpočet délek křivek mezi vrcholy. Výpočet je aproximován pomocí sumy vzdáleností mezi body na křivce, jejichž počet je dán tímto parametrem. Čím větší je číslo, tím větší je přesnost.
- **curvature** – je číselný parametr, který ovlivňuje, jak vypadají křivky mezi vrcholy.
- **grid_size** – tímto parametrem označujeme vzdálenost mezi jednotlivými vrcholy dronového grafu. Tento parametr se uvádí v milimetrech $[mm]$.

- **drone count** – je počet dronů, které se pohybují v každém rámci.
- **maximum speed** – označuje maximální rychlost dronů v jednotkách $[m \cdot s^{-}1]$.
- **maximum acceleration** – označuje maximální akceleraci nebo brždění dronů v jednotkách $[m \cdot s^{-}2]$.
- **drone length radius** – je délkový poloměr dronu v milimetrech $[mm]$. Hodnota nesmí být větší než je polovina délky velikosti mřížky.
- **drone width radius** – je šířkový poloměr dronu v milimetrech $[mm]$. Hodnota nesmí být větší než je polovina délky velikosti mřížky.
- **drone height radius** – je výškový poloměr dronu v milimetrech $[mm]$. Hodnota nesmí být větší než je polovina délky velikosti mřížky.
- **stop penalty** – pokud je nastaven na kladnou hodnotu, algoritmus penalizuje zastavení agenta na místě daným počtem milisekund $[ms]$.
- **wait time** – značí, po jakou dobu mají drony zůstat na místě, jakmile dorazí na místo určení, v milisekundách $[ms]$.
- **distance multiplier** – je hyperparametr pro nízkoúrovňové hledání cest. Násobí hodnotu heuristiky pro odhad vzdálenosti daným číslem. Čím větší číslo, tím více algoritmus preferuje expandování vrcholů, které jsou blíže k cíli.
- **batch size** – pokud má tento hyperparametr kladnou hodnotu, probíhá vzlet a přistání dronů po dávkách s maximálním počtem dronů daných tímto parametrem.
- **batch delay** – je časový parametr v milisekundách $[mm]$, který zpožďuje jednotlivé dávky dronů.
- **batch distance** – určuje, že v dané dávce nemají startovat drony, jejichž startovní nebo cílová vzdálenost je menší než dané číslo.

Podobně jako algoritmus dronového displeje lze parametrizovat hodnoty i u generátoru náhodných dat:

- **frames count** – parametr, který určuje počet rámců, pro které se vygeneruje náhodné umístění pixelů.
- **drone density** – je parametr mezi 0 až 1, který určuje, jaká je hustota vygenerovaných dronů. Hodnota 0 značí, že umístění barevných pixelů je rozeseto rovnoměrně po celém displeji, hodnota 1 značí, že drony jsou umístěné blízko sebe v centru displeje.
- **fixed drones** – značí, kolik dronů má přesně určenou pozici v rámcích.

4.3.2 Měření rychlosti algoritmu a kvality výsledku

V této sekci popíšeme, jaké jsou naměřené hodnoty pro zvolené hodnoty parametrů. Upřesníme, že na výpočet každé úlohy byl alokovaný maximální čas 3 minuty a velikost paměti 6 GB. Testování probíhalo paralelně na vícejádrovém počítači (jedno jádro na úlohu) na operačním systému Ubuntu 20.04. Rychlost procesorového jádra byla 2.6 GHz.

Úlohy jsme měřili tak, že jsme hýbali jedním parametrem a všechny ostatní jsme zafixovali. Každou hodnotu jsme měřili na 30 různých náhodných instancích. Pokud dále nebude uvedeno jinak, zvolili jsme tyto hodnoty parametrů: rozměry displeje $30 \times 30 \times 15$, $k = 3$, $precision = 100$, $curvature = 1$, velikost mřížky 1 metr, 50 dronů, rychlost $4m \cdot s^{-1}$, akcelerace $0.01m \cdot s^{-2}$. Poloměry dronu jsou $0,3 \times 0,3 \times 0,4$ metrů. Penalizace za zastavení na místě je 100 milisekund, čekání na pozicích mezi rámci jsou 3 sekundy. Inflační hyperparametr (distance multiplier) je nastavený na 10, dávkový parametr vypnutý. Rámců bylo 12 (z toho 10 prostorových + 2 vzlet a přistání), hustota 0 a zafixovaných dronů 0.

4.3.2.1 Počet dronů

První metriku, kterou jsme vyzkoušeli, byl vliv počtu dronů na výkonnost algoritmu. Zároveň jsme chtěli porovnat vliv dávkové heuristiky na výkonnost a kvalitu. Heuristiku jsme vyzkoušeli na stejném datasetu s nastavením velikosti dávky na 30, zpožděním dávky na 3 sekundy a vzdáleností rovnou 2.

V tabulce 4.1 se nachází změřené údaje pro tuto metriku. Levý sloupec obsahuje měřené hodnoty o počtech dronů. Následují dva slouce, které udávají průměrnou dobu běhu na instanci pro úspěšně vypočtené instance. Velké písmeno H značí, že udávaná hodnota byla naměřena pro variantu s heuristikou. Další dva sloupce začínající na písmenka SR udávají success rate pro obě varianty instancí. To značí poměr úspěšně spočtených instancí oproti celkovému počtu. Poslední sloupec je poměr, který říká, kolikrát je heuristické řešení v průměru horší než řešení nalezené bez heuristiky. V této hodnotě měříme sumu cen všech agentů.

Tabulka 4.1: Závislost doby běhu na počtu dronů

#	Doba běhu	Doba běhu H	SR	SRH	Poměr kvality
10	0,30 s	x	100,00 %	x	x
25	0,70 s	x	100,00 %	x	x
50	1,47 s	1,67 s	100,00 %	100,00 %	1,45
100	4,41 s	3,48 s	96,67 %	100,00 %	2,34
150	17,99 s	5,16 s	100,00 %	100,00 %	3,31
200	59,46 s	7,27 s	63,33 %	100,00 %	4,37

Pro první dva parametry jsme tuto heuristiku nepoužili. To je proto, že pro takto malé instance by ze své podstaty dávkování neuplatnilo. Jak vidíme podle naměřených hodnot, průměrná doba pro větší instance dramaticky stoupá. Pro opravdu velké instance máme i velký propad úspěšnosti algoritmu. Tato negativa můžeme zmírnit pomocí heuristiky, zaplatíme ale násobně dražším řešením.

4.3.2.2 Konektivita

Další parametr, který jsme zkoumali, je konektivita vrcholů grafu displeje. Údaje z těchto běhů jsme zanesli do tabulky 4.2.

Tato tabulka obsahuje čtyři sloupce: hodnota konektivity, průměrná doba běhu, úspěšnost (success rate) a poměr kvality. Poměr kvality značí kolikrát je kvalita průměrně horší oproti hodnotě na prvním řádku. Pokud je hodnota menší než jedna, máme ve skutečnosti lepší výsledek než předtím.

Tabulka 4.2: Závislost doby běhu na zvolené konektivitě vrcholů

K	Doba běhu	Success rate	Poměr kvality
2	0,24 s	96,67 %	1,00
3	1,64 s	96,67 %	0,81
4	1,62 s	96,67 %	0,81
5	1,62 s	96,67 %	0,81
6	1,75 s	96,67 %	0,81

Podle naměřených dat na našich instancích došlo ke změně pouze mezi konektivitou 2 a konektivitou 3. Konektivita 2 vypočte řešení rychle, ovšem řešení je lehce horší než s konektivitou 3. Změny v dalších hodnotách jsou pouze marginální a nedošlo ke zlepšení výsledku. Zdá se, že pro velikost 6 mírně začíná opět růst exekuční čas.

4.3.2.3 Hustota dronů

Třetím zkoumaným parametrem je vliv hustoty dronů na dobu výpočtu algoritmu. V tabulce 4.3 najdeme naměřené hodnoty pro sloupce hustoty, průměrné doby běhu a úspěšnosti (success rate).

Tabulka 4.3: Závislost doby běhu na hustotě dronů

Hustota	Doba běhu	Success rate
0,1	1,60 s	100,00 %
0,2	1,64 s	100,00 %
0,3	1,60 s	100,00 %
0,4	1,67 s	100,00 %
0,5	1,72 s	100,00 %
0,6	1,80 s	100,00 %
0,7	2,20 s	100,00 %
0,8	5,70 s	96,67 %
0,9	41,64 s	70,00 %
1,0	68,39 s	16,67 %

Naměřili jsme, že pro hustší instance strmě stoupá exekuční čas a stejně tak rychle klesá úspěšnost algoritmu.

4.3.2.4 Fixované drony

Poslední sledovaný parametr určuje, kolika dronům jsme v každém rámci uložili přiřazené místo. Tabulka 4.4 obsahuje opět 3 sloupce. V prvním je hodnota parametru, následuje průměrná doba běhu a úspěšnost. Nejlepších výsledků dle tabulky dosahuje hodnota parametru 0.

Tabulka 4.4: Závislost doby běhu na počtu fixovaných dronů

Zafixované drony	Doba běhu	Success rate
0	1,71 s	100,00 %
5	5,67 s	86,67 %
10	28,07 s	86,67 %
15	19,33 s	76,67 %

4.3.3 Interpretace výsledku

Z naměřených dat vyplývá, že se jedná o kvalitní algoritmus, který řeší sériově několik instancí multi-agentního hledání cest pro nemalé množství agentů. Je ovšem nutné vzít v potaz povahu dat – náhodná data pravděpodobně nebudou vypadat jako reálné konfigurace. Takovým příkladem může být instance o 105 dronech, které dohromady vytvářejí obraz draka. Náhled takové instance je k dispozici v příští kapitole. Bez správně nastavených hyperparametrů tato instance neskončí v časovém limitu 3 minuty a 6 GB. Bohužel tato instance na testovaném počítači neskončí ani v časovém limitu 20 minut a veškeré paměti počítače (32 GB). Se správně zadanými hyperparametry ovšem algoritmus vrátí řešení za 0,3 sekundy.

Věříme ovšem, že lze k těmto statistikám přihlédnout a všimnout si, co pozitivně a negativně ovlivnilo výkonnost algoritmu. Velkým problémem jsou zcela jistě husté instance. Tam dochází k tomu, že jak jsou droni k sobě víc nahustění, je větší pravděpodobnost konfliktu. Obdobný vliv má zvětšování počtu dronů v instanci, zde však není nárůst tak vysoký jako v předchozím případě. Dále nedoporučujeme fixovat drony na pozice.

Překvapením pro nás bylo, že až na čísla 2 a 3 neměla jinak volba konektivity prakticky vliv na rychlost a přesnost výsledku. To je možná tím, že dimenze displeje nebyly dostatečně rozsáhlé, aby se změna projevila. Každopádně i tak doporučíme zvolit velikost 2 nebo 3 – menší čísla mají přinejmenším menší paměťové nároky a jak je vidět, v praxi to stačí.

Potěšilo nás, že naše dávková heuristika má signifikantní vliv na rychlost algoritmu a umožňuje spočítat instance, které klasickým způsobem spočítat nejdou. Z toho také vyplývá, že naše hypotéza, která tvrdí, že algoritmus tráví podstatnou část výpočtem vzletu a přistání, je správná. Urychlením této části se drasticky zrychlí celý algoritmus, zejména pro instance s větším počtem dronů.

Kapitola 5

Vizualizační část

V této kapitole popíšeme program Vizualizér dronového displeje. Tento program umožňuje simulovat pohyb dronů v prostoru pomocí algoritmu dronového displeje.

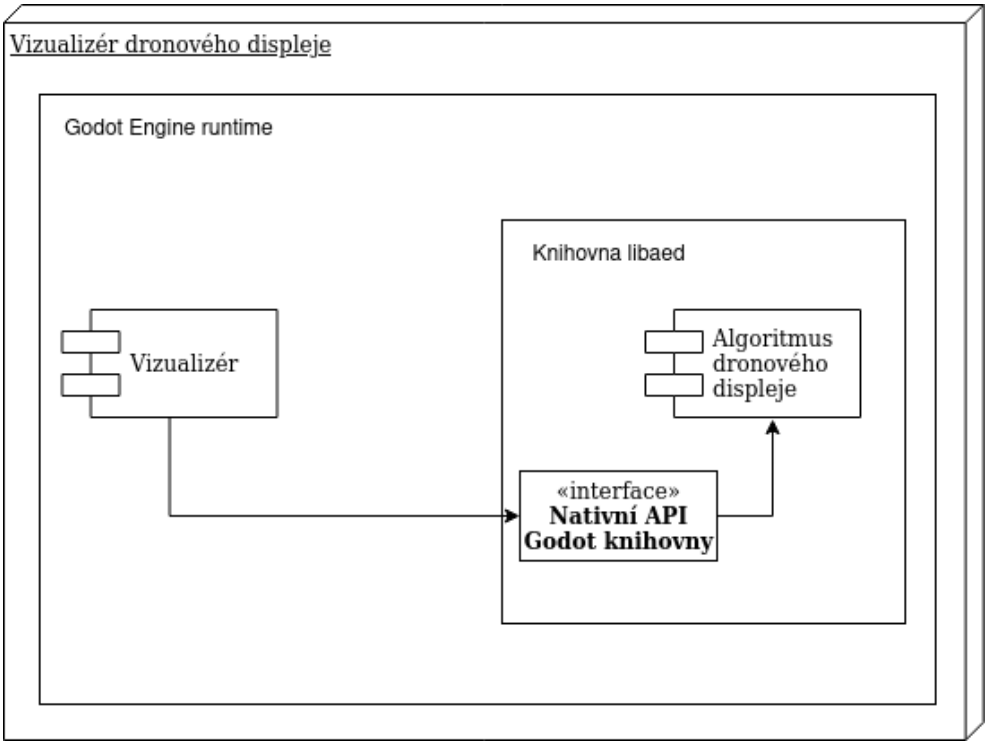
5.1 Architektura

V této sekci popíšeme architektonické schéma programu. Pro zobrazení objektů v 3-dimenzionálním prostoru jsme použili technologii Godot Engine [22]. Godot Engine je open-source framework, který bežně umožňuje zdarma vyvíjet 2D nebo 3D hry. Zvolili jsme ho i proto, že umožňuje vyvíjet a kompilovat projekty pro různé operační systémy. V předchozí kapitole jsme představili algoritmus dronového displeje, který jsme naimplementovali v jazyce C++. Tento framework nám navíc umožňuje takto napsaný kód přepoužít v rámci knihovny pro náš projekt.

Nyní již ke schématu, které představíme na ilustraci 5.1. V následujících bodech popíšeme jednotlivé moduly.

- **Godot Engine runtime** - Jak jsme již zmínili, náš program využívá framework Godot Engine, který umožňuje rozhýbat a zobrazit celý displej. Tento runtime běží po celou dobu na pozadí. Godot umožňuje integraci pomocí statických scén a skriptů, které naopak umožňují naprogramovat interaktivní část aplikace. Skriptovat v Godotu se dá několika různými způsoby: pomocí jazyka GDScript (vlastní jazyk inspirovaný Pythonem), jazyka C#, VisualScript a umožňuje integraci C a C++ knihoven pomocí mechanismu GDNative. Tyto jazyky se v rámci projektu dají kombinovat.

- **Vizualizér** - Vizualizér je náš modul obsahující scény a skripty pro Godot Engine. Tato část definuje, jak vypadá prostředí, poskytuje uživatelské rozhraní k načítání a ukládání a editaci konfigurace displeje,



Obrázek 5.1: Schéma popisující Vizualizér dronového displeje. Součástí programu je knihovna libaed, kterou modul Vizualizér využívá pro výpočet trajektorií. Veškeré části běží v rámci Godot Enginu.

kterou ukládá do konfiguračního souboru. Zobrazuje pohyb dronů a jejich barvy. Pro výpočet problému dronového displeje se obrací přes API libaed knihovny.

- **Knihovna libaed** umožňuje integraci vizualizační části s výpočetním algoritmem. Navenek vystavuje API, které jde zavolat z Godot skriptů. Vnitřně se pak tato knihovna stará o překlad datových struktur, které používá Godot do interních struktur algoritmu dronového displeje, který poté zavolá. Po výpočtu transformuje výsledky algoritmu a sestaví plán trajektorie každému agentu.

5.2 Uživatelské rozhraní

Tato sekce popisuje možnosti uživatelského rozhraní. Ukážeme, jak se program používá a představíme ukázky.

Obrázek 5.2: Ukázka úvodního menu po spuštění programu.

5.2.1 Úvodní menu

Úvodní menu 5.2 obsahuje 3 tlačítka:

- **New Simulation** - otevře dialog k vytvoření a editaci simulace nového displeje.
- **Load Simulation** - otevře dialog k načtení simulace.
- **Exit** - ukončí program.

5.2.2 Nastavení displeje

Po vytvoření nové simulace nebo po načtení existující uživatel uvidí uživatelské rozhraní pro editaci displeje. Toto rozhraní je rozděleno do dvou tabů, mezi kterými může uživatel přepínat.

První tab zvaný **Display Settings** 5.3a se zabývá nastavením parametrů. Parametry jsou seskupeny podle kategorií, které v bodech vysvětlíme. Popis všech parametrů lze nalézt v předchozí kapitole v sekci experimentálním vyhodnocení algoritmu 4.3.

- **Graph** - umožňuje konfigurovat hodnoty samotného displeje, například jeho dimenze, propojení vrcholů nebo velikost mřížky.

- **Drones** - obsahuje nastavení dronů, jako jejich počet, velikost a další vlastnosti.

- **Frames** - obsahuje jedinou hodnotu k nastavení, a to množství času, po které budou droni čekat v rámci, než dojde k přesunu do dalšího rámce. Zbylé nastavení se nachází v dalším tabu.

- **Hyper parameters** - jsou parametry, jejichž správným nastavením lze zajistit rychlejší výpočet výsledku.

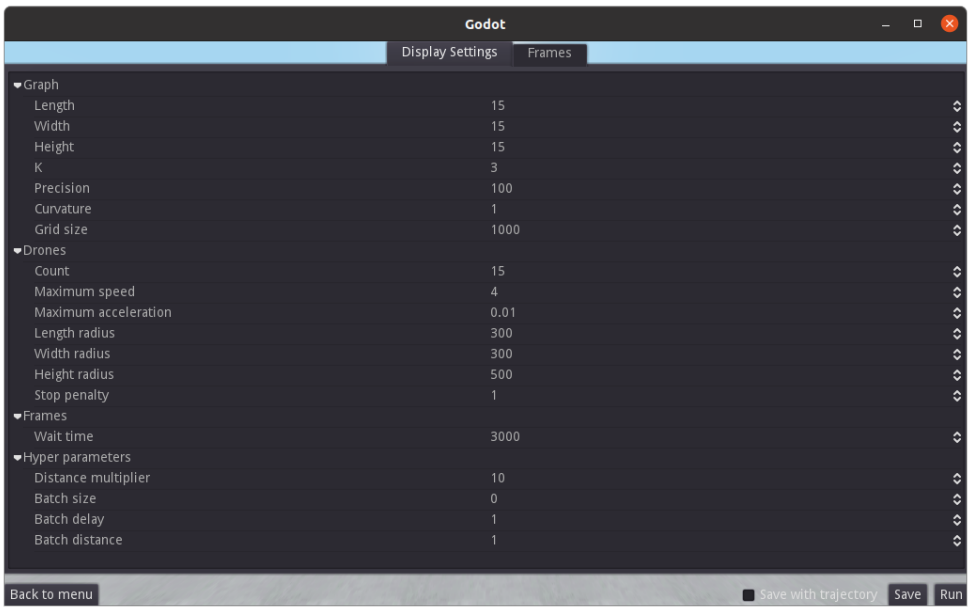
Druhý tab se jmenuje **Frames** a slouží k nastavení rámců a obrazového výstupu. Tento tab obsahuje tyto prvky:

- **Horní lišta** – horní lišta obsahuje ovládací prvky k manipulaci se samotnými rámci. Popořadě zleva doprava zde nalezneme prvky k přepínání aktuálního rámce, vyčištění, duplikaci a smazání aktuálního rámce. Dále zde nalezneme tlačítka k vložení nových rámců před a za aktuální rámec.

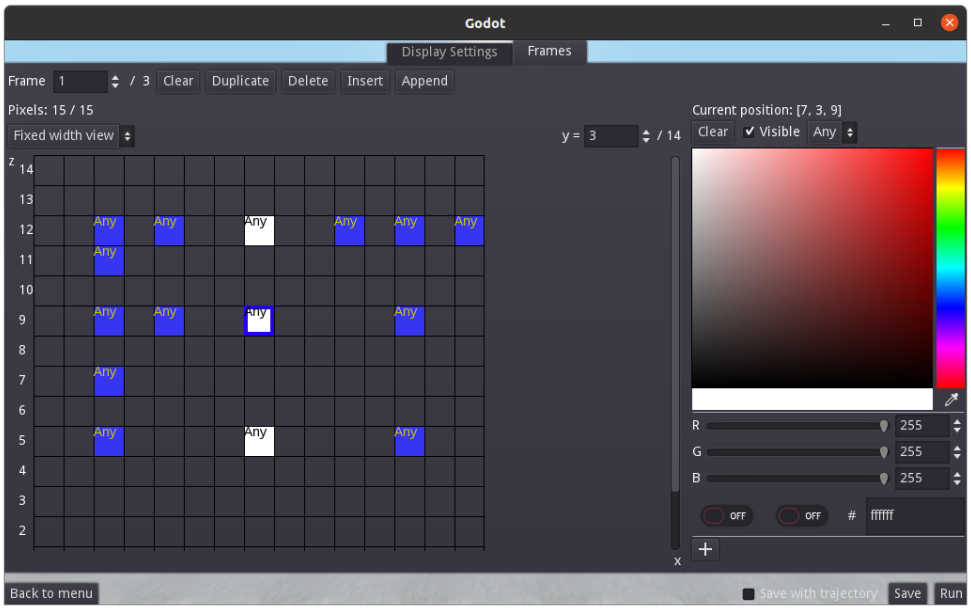
- **Levá část** – zde nalezneme také několik ovládacích nebo informačních prvků. Nejprve zde máme ukazovátko, které informuje uživatele o počtu aktuálně nastavených pixelů a kolik je jich třeba celkem nastavit v daném rámci. Levé části ovšem dominuje 2-dimenzionální mřížka, která umožňuje vybírat konkrétní umístění. Ovšem mřížka displeje je 3-dimenzionální, tedy ukázaná mřížka je pouze řez z jedné dimenze. Na tento řez si můžeme vybrat libovolnou dimenzi. Aby šly nastavit všechny pixely v rámci, máme zde i přepínátko, které umožňuje vybrat si souřadnici řezu.

- **Pravá část** – pravá část je rozdělena na tři řádky. První je informační, ukazuje souřadnici vybraného pixelu, připadně zobrazuje informaci, že žádný pixel není vybraný. V druhém řádku nalezneme ovládací prvky umožňující smazání barvy, vypnutí barvy a nastavení agenta, který má být přítomen v prostoru pro daný pixel. Zbytek pravé části vyplňuje selektor barev, který umožňuje výběr libovolné barvy. Tyto barvy si i umí uložit do palety, aby uživatel nemusel použití stejné barvy naklikávat vícekrát.

Oba taby mají společnou spodní lištu, která obsahuje tlačítka k navrácení do hlavního menu, tlačítko uložení a tlačítko spuštění programu. Je zde i přepínátko, které určuje, zda se má s konfigurací displeje uložit i výsledek. Tento výsledek lze v příštím běhu načíst i s konfigurací, a pokud se konfigurace mezitím nezmění, Vizualizér nebude tento výsledek znovu počítat.



(a) Nastavení parametrů a hyperparametrů dronového displeje.



(b) Konfigurace pixelů

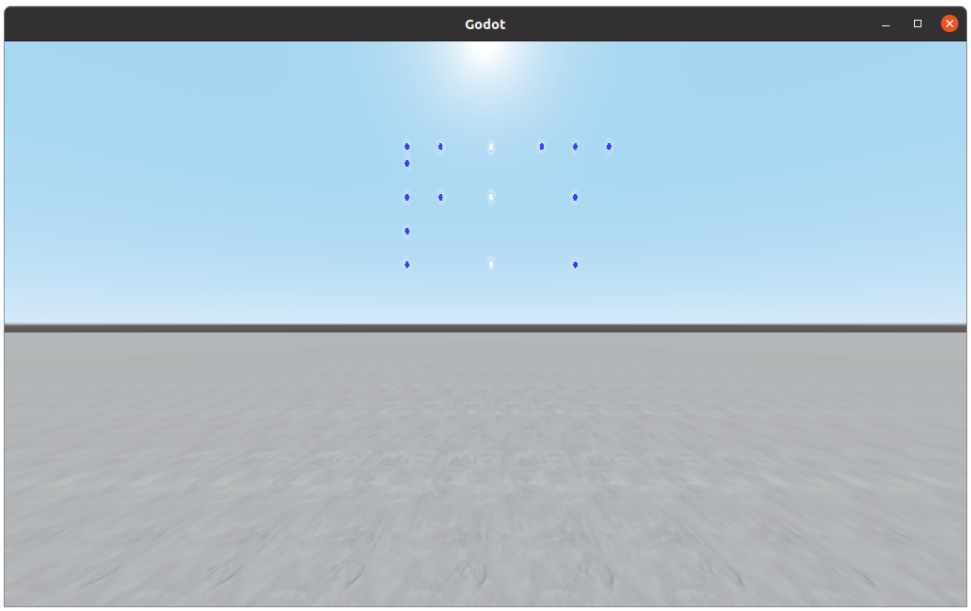
Uživatelské rozhraní pro konfiguraci pixelů dronového displeje. Vlevo obsahuje mřížku pro výběr pixelu, vpravo obsahuje selektor pro výběr barvy.

Obrázek 5.3: Různé možnosti nastavení displeje

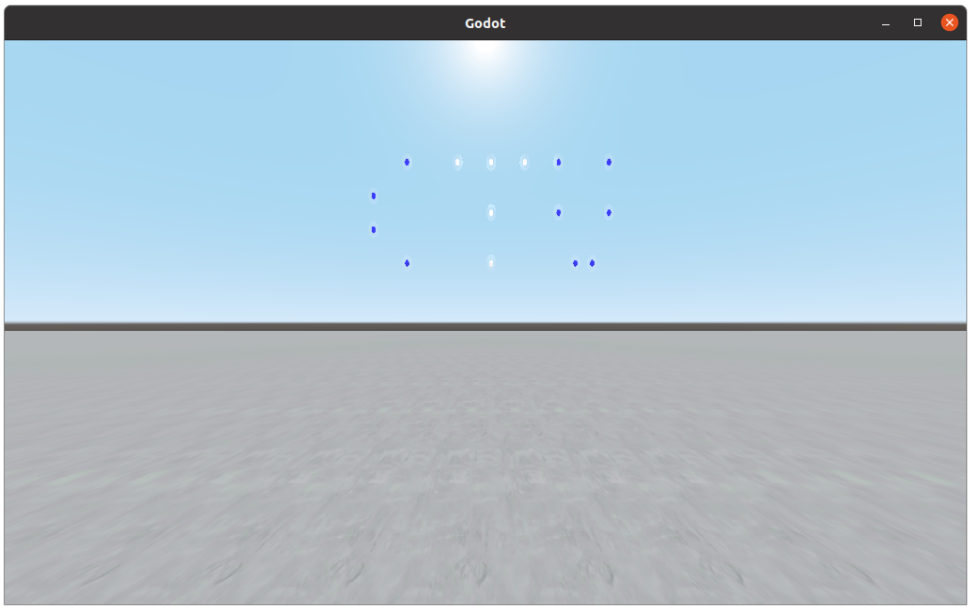
5.2.3 Displej

Po stisknutí tlačítka **Run** dojde ke spuštění simulace. V té se můžeme pohybovat pomocí šipek a záběr kamery lze upravit pomocí pohybu myši doleva a doprava. Jednotlivé drony jsou reprezentovány pomocí bílého průhledného elipsoidu, jehož poloměry byly nastaveny v sekci nastavení dronů. Simulaci lze ukončit stisknutím tlačítka escape. Po stisknutí vyskočí opět nastavení simulace. Opětovným stisknutím tlačítka Run dosáhneme znovuspuštění stejné simulace.

Pro představu, jak taková simulace vypadá, jsme si připravili rovnou dvě ukázky (obě lze nalézt v elektronické příloze této práce k samostatnému vyzkoušení). V první ukázce dochází ke změně nápisů, kde se drony přeskupí z nápisu FIT 5.4a do nápisu CTU 5.4b. V té druhé jsme si připravili vizualizaci draka, na který je potřeba 105 dronů. Ten jsme zachytili zepředu 5.5a a také ze strany 5.5b, kde je lépe vidět prostorový rozměr draka.

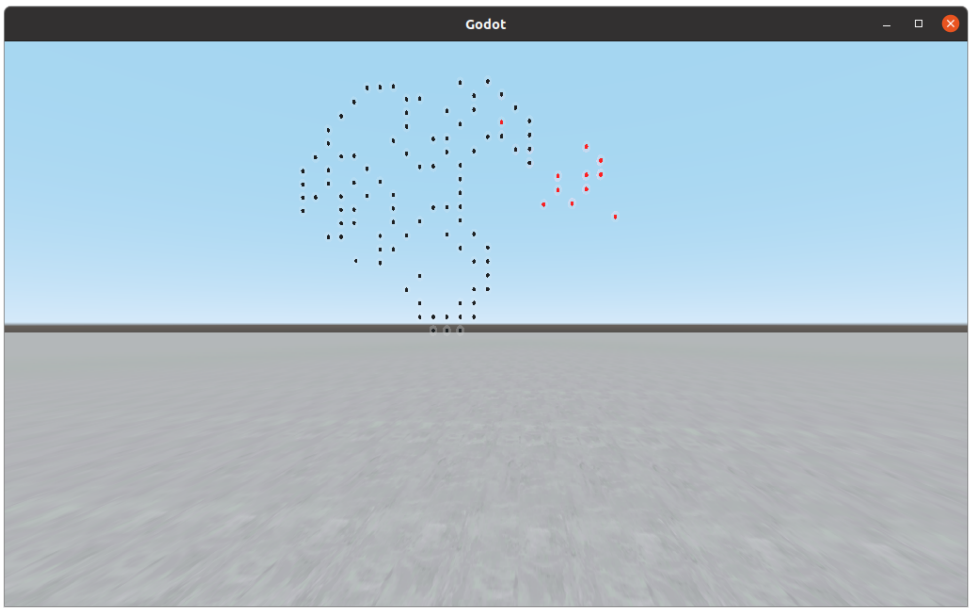


(a) Nápis FIT složený z 14 dronů.

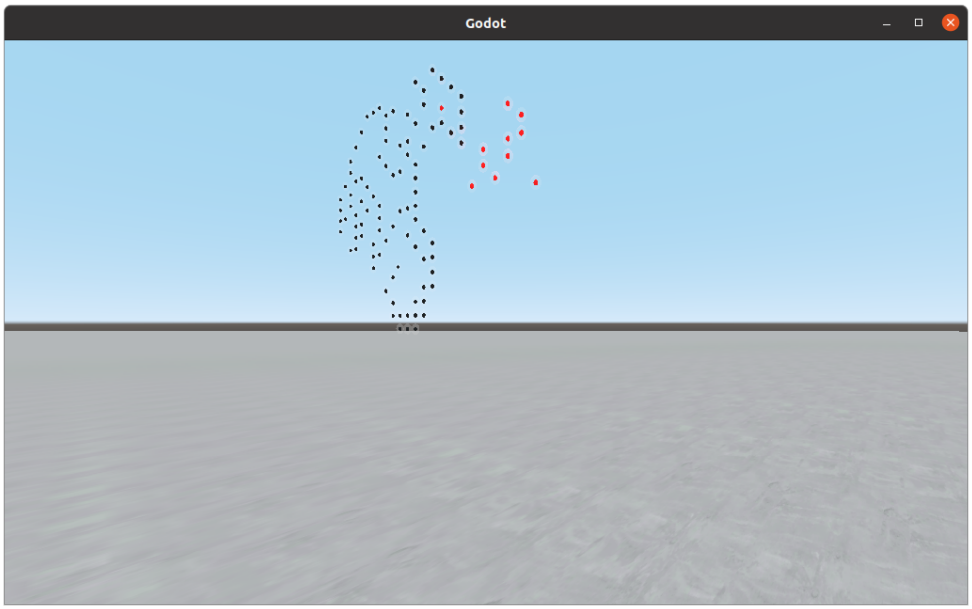


(b) Nápis CTU, který vznikl transformací dronů z předchozího obrázku 5.4a.

Obrázek 5.4: Ukázka s měnícím se nápisem



(a) Obrázek draka, který se skládá z celkem 105 dronů.



(b) Obrázek stejného draka jako v obrázku 5.5a z boku. Na této ilustraci je lépe vidět prostorové rozmístění dronů, které znázorňují oheň.

Obrázek 5.5: Ukázka displeje s postavou draka

Závěr

Cílem práce bylo vytvoření softwarového prototypu, který demonstruje proveditelnost dronového displeje. Tento a všechny dílčí cíle se podařilo splnit.

V rámci práce jsme se zabývali algoritmickými i jinými teoretickými podklady, které nám pomohly k vytvoření definice problému dronového displeje. K vyřešení problému jsme adaptovali existující algoritmus pro hledání multi-agentních cest zvaný CBS (Conflict Based Search) a vymysleli jsme jeho novou variantu CSPT_CBS (Continuous Spacetime Conflict Based Search). Tato varianta se liší oproti originálnímu algoritmu v tom, že bere v potaz spojitý prostor a čas, zrychlující i brzdící agenty a také částečně jejich momentum. V rámci algoritmu dronového displeje jsme také využili existující Maďarský algoritmus. Pro algoritmus jsme dále vymysleli základní heuristiky zrychlující čas výpočtu. Následně jsme algoritmus experimentálně vyhodnotili. V závěru této práce jsme představili vizualizační program s ukázkami, který simuluje drony v displeji.

V budoucnu bychom rádi využili tento algoritmus k demonstraci reálného dronového displeje. Vidíme zde ovšem i potenciál dalších vylepšení algoritmu, na které se z časových důvodů nedostalo. Jedním z takových vylepšení je například implementace přesného kolizního mechanismu a změření jeho vlivu na výkonnost algoritmu. Další zlepšení vidíme ve vymyšlení jiných heuristik, které zrychlí výpočet algoritmu, případně již existujících CBS mechanismů pro rychlejší výpočet. Pozornost si jistě zaslouží část hledání cest, kde jsme nebrali v potaz, zda jsou nastolené trajektorie realisticky dosažitelné. Způsob, jakým jsme zakomponovali momentum do algoritmu, je zcela jistě naivní a také si zaslouží vylepšení.

Literatura

[1] Yamada, W.; Yamada, K.; Manabe, H.; aj.: *ISphere: Self-Luminous Spherical Drone Display*. UIST '17, New York, NY, USA: Association for Computing Machinery, 2017, ISBN 9781450349819, str. 635–643, doi:10.1145/3126594.3126631. Dostupné z: https://doi.org/10.1145/3126594.3126631

[2] Gomes, A.; Rubens, C.; Braley, S.; aj.: *BitDrones: Towards Using 3D Nanocopter Displays as Interactive Self-Levitating Programmable Matter*. New York, NY, USA: Association for Computing Machinery, 2016, ISBN 9781450333627, str. 770–780. Dostupné z: https://doi.org/10.1145/2858036.2858519

[3] Preiss, J.; Honig, W.; Sukhatme, G.; aj.: Crazyswarm: A large nano-quadcopter swarm. 05 2017: s. 3299–3304, doi:10.1109/ICRA.2017.7989376.

[4] AB, B.: Crazyflie 2.0. Navštíveno 2.9.2021. Dostupné z: https://www.bitcraze.io/products/old-products/crazyflie-2-0/

[5] Surynek, P.: A novel approach to path planning for multiple robots in bi-connected graphs. 2009: s. 3616–3619.

[6] Surynek, P.: An Optimization Variant of Multi-Robot Path Planning is Intractable. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, AAAI Press, 2010, str. 1261–1263.

[7] Wagner, G.; Choset, H.: M*: A Complete Multirobot Path Planning Algorithm with Performance. 09 2011, s. 3260–3267, doi:10.1109/IROS.2011.6095022.

[8] Hart, P. E.; Nilsson, N. J.; Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems*

Science and Cybernetics, ročník 4, č. 2, 1968: s. 100–107, doi:10.1109/TSSC.1968.300136.

[9] Sharon, G.; Stern, R.; Felner, A.; aj.: Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, ročník 219, 02 2015: s. 40–66, doi:10.1016/j.artint.2014.11.006.

[10] Andreychuk, A.; Yakovlev, K.; Atzmon, D.; aj.: Multi-Agent Pathfinding with Continuous Time. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, International Joint Conferences on Artificial Intelligence Organization, 7 2019, s. 39–45, doi:10.24963/ijcai.2019/6. Dostupné z: `https://doi.org/10.24963/ijcai.2019/6`

[11] Phillips, M.; Likhachev, M.: SIPP: Safe interval path planning for dynamic environments. *2011 IEEE International Conference on Robotics and Automation*, 2011: s. 5628–5635.

[12] Barer, M.; Sharon, G.; Stern, R.; aj.: Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. *Frontiers in Artificial Intelligence and Applications*, ročník 263, 01 2014: s. 961–962, doi:10.3233/978-1-61499-419-0-961.

[13] Pohl, I.: Heuristic Search Viewed as Path Finding in a Graph. *Artif. Intell.*, ročník 1, 1970: s. 193–204.

[14] Hönig, W.; Preiss, J. A.; Kumar, T. K. S.; aj.: Trajectory Planning for Quadrotor Swarms. *IEEE Transactions on Robotics*, ročník 34, č. 4, 2018: s. 856–869, doi:10.1109/TRO.2018.2853613.

[15] Yu, J.; LaValle, S.: Multi-agent Path Planning and Network Flow. 04 2012, doi:10.1007/978-3-642-36279-8_10.

[16] Dobson, A.; Bekris, K.: Sparse roadmap spanners for asymptotically near-optimal motion planning. *International Journal of Robotics Research*, ročník 33, 01 2014: s. 18–47, doi:10.1177/0278364913498292.

[17] Edmonds, J.; Karp, R.: Theoretical Improvement in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, ročník 19, 01 2003: s. 248–264, doi:10.1007/3-540-36478-1_4.

[18] Rainer Burkard, M. D.; Martello, S.: *4. Linear Sum Assignment Problem*, kapitola 4. s. 73–144, doi:10.1137/1.9781611972238.ch4, `https://epubs.siam.org/doi/pdf/10.1137/1.9781611972238.ch4`. Dostupné z: `https://epubs.siam.org/doi/abs/10.1137/1.9781611972238.ch4`

[19] Rivera, N.; Hernández, C.; Hormazábal, N.; aj.: The 2^k Neighborhoods for Grid Path Planning. *Journal of Artificial Intelligence Research*, ročník 67, 01 2020: s. 81–113, doi:10.1613/jair.1.11383.

[20] Knott, G. D.: *Cubic Polynomial Space Curve Splines*. 2000, ISBN 978-1-4612-1320-8, s. 77–93, doi:10.1007/978-1-4612-1320-8_7. Dostupné z: `https://doi.org/10.1007/978-1-4612-1320-8_7`

[21] Marek, O.: Drone Aerial Display. Dostupné z: `https://gitlab.fit.cvut.cz/marekon9/drone-aerial-display`

[22] Godot Engine. Navštíveno 16.9.2021. Dostupné z: `https://godotengine.org/`

Příloha A

Seznam použitých zkratek a pojmů

BCBS Bounded Conflict Based Search (ohraničené hledání pomocí konfliktů)

CBS Conflict Based Search (hledání pomocí konfliktů)

CCBS Continuous Conflict Based Search (hledání pomocí konfliktů se spojitým časem)

CSPT_CBS Continuous Spacetime Conflict Based Search (hledání pomocí konfliktů ve spojitém čase a prostoru)

GCBS Greedy Conflict Based Search (hladové hledání pomocí konfliktů)

LSAP Linear Sum Assignment Problem (lineární součtové přiřazení)

MAPF Multi-agent Path Finding (multi-agentní hledání cest)

iSphere Self-Luminous Spherical Drone Display (Světelný sférický dronový displej)

SIPP Safe Interval Path Planning (plánování cest pomocí bezpečných intervalů)

POV Persistence of Vision (vytrvalost vidění)

Příloha **B**

Obsah přiloženého CD

- `README.md` ..... stručný popis obsahu CD
- `bin`
  - `cmd` ..... obsahuje aplikaci příkazového řádku
    - `linux` ..... adresář s linuxovou verzí
    - `win` ..... adresář s verzí pro Windows
  - `visualiser` ..... obsahuje vizualizační aplikaci
    - `linux` ..... adresář s linuxovou verzí
    - `win` ..... adresář s verzí pro Windows
- `data` ..... složka uložených konfigurací displejů
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `README.md` ..... detailnější popis struktury zdrojového kódu
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `thesis.pdf` ..... text práce ve formátu PDF