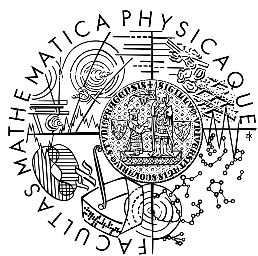


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Bc. Jan Pacovský

Navigace jednotek v počítačových hrách za pomoci toků v sítích

Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot, Ph.D.
Studijní program: Informatika
Studijní obor: Umělá inteligence

Praha 2019

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 8.5.2019 Podpis autora

Velice děkuji Mgr. Jakubu Gemrotovi, Ph.D. z Katedry softwaru a výuky informatiky za vedení mé diplomové práce, za jeho pomoc a čas, který mi ochotně věnoval. Za veškerou podporu děkuji také své rodině a svým přátelům, bez nichž bych si své studium nedokázal představit.

Název práce: Navigace jednotek v počítačových hrách za pomoci toků v sítích

Autor: Bc. Jan Pacovský

Katedra: Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Tato diplomová práce se zabývá navigací jednotek v real-time strategických hrách a zaměřuje se především na problematiku navigace větší skupiny jednotek herním světem, v němž existují zúžená místa. V nich často dochází k výraznému zdržení, kterému by se dalo předejít, pokud by se skupina rozdělila a využila též alternativních cest. V této práci je navrženo řešení tohoto problému pomocí toků v sítích. Převedením herní mapy na síť a její následnou analýzou nalezneme alternativní cesty k přetíženému zúžení, které plánovači cest umožní naplánovat jednotkám ve skupině různé trasy. Předkládaná metoda byla testována v simulátoru, který byl za tímto účelem zvlášť navržen, a porovnávána s jinými způsoby navigace běžně užívanými v počítačových hrách. Závěrečné vyhodnocení ukazuje, že navigace jednotek pomocí toků v sítích dokáže tento problém úspěšně vyřešit.

Klíčová slova: počítačové hry; navigace jednotek; toky v sítích; real-time strategie

Title: Navigation of Units in Video Games Using Flow Networks

Author: Bc. Jan Pacovský

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: This thesis investigates the navigation of units in real-time strategy computer games and is focused on the task of navigating a bigger group of units through a game world that contains narrows passages. The narrow passages causes significant delays that could be avoided by splitting the group and taking some part of it through an alternative way. In this thesis, a solution using flow networks is proposed. The map is transformed into a flow network which then allows the planner to find an alternative route to the fully occupied narrow passage. The proposed method was tested in a simulator specifically designed for this task and compared with the conventional methods of navigation in computer games. The final evaluation shows that the navigation of units using flow networks can successfully solve this problem.

Keywords: video games; navigation of units; flow networks; real-time strategy

Obsah

Úvod **3**

1 Formulace problému **6**
1.1 Abstrakce počítačové hry 6
1.1.1 Mapa 7
1.1.2 Jednotky 7
1.1.3 Skupiny 7
1.1.4 Naplánování cest jednotkám ve skupině 8

2 Toky v sítích **9**
2.1 Definice 9
2.1.1 Definice toků 9
2.1.2 Aplikace definic sítě 10
2.2 Vybrané problémy 10
2.2.1 Problém maximálního toku 10
2.2.2 Dynamické toky – toky v průběhu času 11
2.2.3 Transshipment 11
2.2.4 Dělitelné toky 11
2.2.5 Problém maximálního toku s minimální cenou 12
2.2.6 L-Bounded tok 12
2.2.7 Problém nejrychlejšího toku 12

3 Analýza souvisejících problémů **14**
3.1 Reprezentace herního světa 14
3.1.1 Fyzická reprezentace mapy 14
3.1.2 Logická reprezentace grafu 15
3.2 Plánování 17
3.2.1 Hledání cesty v grafu 18
3.2.2 Skupinové hledání cesty 22
3.3 Exekuce plánu jednotkami 22
3.3.1 Flow field 23
3.3.2 Řízení jednotek 25
3.3.3 Způsoby řízení skupiny 27
3.3.4 Hledání nejbližších jednotek 27

4 Simulátor **29**
4.1 Funkce simulátoru 29
4.2 Mapy 29
4.3 Navigace 30
4.3.1 Plánování 30
4.3.2 Exekuce 31

5 Navrhovaná metoda **34**
5.1 Představení hlavních částí algoritmu 34
5.2 Dekompozice mapy 34
5.3 Vytvoření grafů 36

5.3.1 Středový graf . . . 36
5.3.2 Bezstředový graf . . . 36
5.3.3 Délka hran . . . 36
5.3.4 Porovnání vlastností grafů . . . 38
5.4 Převod na síť . . . 39
5.5 Hledání cest . . . 39
5.5.1 Problém dekompozice toku . . . 39
5.6 Přiřazení jednotek nalezeným cestám . . . 44
5.6.1 Optimalizace . . . 44
5.6.2 Přidělení plánu jednotkám . . . 46
5.6.3 Převod seznamu bran na plán cesty pro jednotku . . . 48

6 Evaluace **50**
6.1 Porovnávané metody plánování . . . 50
6.1.1 Popis chování obou způsobů navigace při exekuci . . . 50
6.2 Výběr map . . . 51
6.3 Konfigurace experimentu . . . 51
6.3.1 Provedení experimentu . . . 51
6.4 Čítače . . . 52
6.4.1 Význam čítačů . . . 52
6.4.2 Další položky souboru . . . 53
6.5 Testované scénáře . . . 55
6.5.1 Měření . . . 55
6.6 Vyhodnocení . . . 55

7 Dokumentace **61**
7.1 Uživatelská dokumentace . . . 61
7.1.1 Požadavky ke spuštění . . . 61
7.1.2 Textový režim . . . 61
7.1.3 Grafický režim . . . 61
7.1.4 Překlad . . . 62
7.1.5 Pokročilá nastavení . . . 63
7.2 Programátorská dokumentace . . . 63
7.2.1 Dekompozice mapy . . . 64

Závěr **66**

Seznam použité literatury **67**

A Přílohy **70**
A.1 Ovládání programu . . . 71

Úvod

Tématem této práce je navigace jednotek v počítačových hrách za pomoci toků v sítích, a to konkrétně v tzv. real-time strategiích (RTS). Jedná se o oblíbený žánr strategických počítačových her, v němž veškeré hráčovy úkony a počítačem řízené odezvy probíhají v reálném čase. Tím se liší od tzv. tahových her založených na principu jednotlivých tahů, respektive herních kol. Herní svět v RTS se typicky skládá z neměnné, dvourozměrné mapy, v čase ubývajících surovin a hráči vytvářených budov a jednotek. Za příklad takovýchto her můžeme považovat herní série Command & Conquer, Starcraft či Age of Empires.

Navigace jednotek v počítačových hrách reálného času představuje specifickou problematiku s vlastní množinou problémů a dělí se na dvě části. První částí je vytvoření plánu a druhou je jeho reálné vykonání (exekuce). Plánování má za úkol nalézt cestu ze startu do cíle. Cesta je posloupnost míst v herním světě určující trasu jednotky. Exekuce vykonává samotný pohyb jednotek na předem naplánovaných cestách.

Plánování cesty pro jednu jednotku je problémem studovaným od dob prvních počítačů. Dnes je hledání cesty pomocí GPS navigace dostupné v každém chytrém telefonu. V této práci se budeme zabývat navigací skupiny jednotek, tedy tento dobře známý problém zobecníme přidáním jednotek, které půjdou mezi stejným startem i cílem. Již ani pro jednu jednotku není bez další specifikace jasné, co lze považovat za nejlepší cestu, pro více jednotek toto platí tím spíše. Z hlediska jediné jednotky lze nejlepší cestu definovat podle zvolených parametrů, jak to umožňuje například běžná navigace pro automobily s nabídkou nejkratší uražené vzdálenosti, nejrychlejší cesty nebo nejekonomičtější cesty. Ve skupině jednotek přibude minimalizace čekání, kdy se jednotka nepohybuje vůbec nebo se nepohybuje plnou rychlostí, minimalizace celkového času, minimalizace sumy jednotlivých časů. Tato práce se bude zabývat minimalizací celkového času.

Hráči své jednotky ovládají jednotlivě nebo po celých skupinách. Hry řízení skupiny jednotek zpravidla řeší jedním ze dvou extrémů. Buď celou skupinu považují za jeden celek, nebo na všechny jednotky nahlíží samostatně, takže ignorují skutečnost, že se vlastně jedná o skupinu. V této práci se držíme uvnitř těchto extrémů. Jednotky budou uvažovány jako samostatné entity, formace nebudou fixní, ale plánování bude probíhat i na úrovni celé skupiny.

V prvním případě se často řeší formace skupiny, která se stává fixní a jednotkám není dovoleno se odchýlit od polohy své skupiny, respektive se od ní oddělit. Ve druhém případě se skupinové plánování zanedbá. Hledání cesty probíhá jakoby se jednalo o nezávislé jednotky. Důsledkem plánování pohybu skupiny skrze plánování jednotlivých jednotek je neoptimální chování, které se nejvíce projevuje ve zúženích. Důvodem je skutečnost, že optimální cesta pro každou jednotku zvlášť není optimální pro všechny jednotky dohromady. Jednotky totiž nemohou být na stejné pozici ve stejný časový okamžik, jinak budou vzájemně kolidovat.

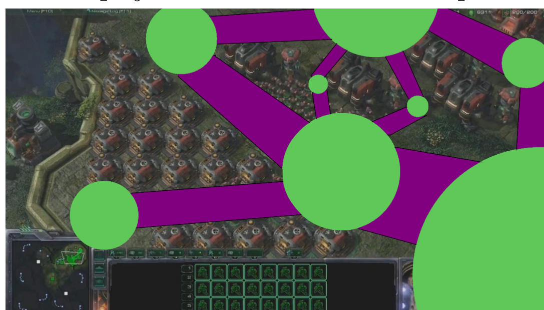
V místě, kde se široké prostranství začíná zužovat a stává se z něj úzký průchod, často dochází k jevu, při němž jednotky na konci velké skupiny zbytečně čekají, dokud se jim neuvolní cesta, protože nemohou všechny projít najednou, ačkoli by mohly využít i jiných alternativních cest mimo dané zúžení.

Pokud však herní mapa jinou možnou cestu nenabízí, nelze tuto problémovou situaci nijak lépe vyřešit. Jestliže existuje alternativní cesta, může být východiskem rozdělení skupiny. Určitou část skupiny by bylo možné poslat po alternativní cestě, čímž by se doba strávená na cestě zkrátila. Právě urychlení cesty pro skupiny jednotek je úlohou, na niž se tato práce zaměřuje a kterou má řešit.

Nabízí se otázka, zda ji lze řešit pomocí toků v sítích, které jsou v rámci teorie grafů předmětem studia teorie sítí. Mapy s vyznačenými zúženými místy, ve kterých je navíc určen počátek a konec pohybu, se totiž nápadně podobají tokům v síti. Proto se předkládaná práce pokouší uplatnit výsledky z tohoto oboru při hledání cest pro koordinovaný pohyb skupiny mezi dvěma vrcholy v rámci grafu. Díky tomu vyhledáme plán pohybu jednotek v celé skupině.

Skupinou jednotek nebudeme rozumět pouze blízkou množinu jednotek, které se spolu pohybují a mají společný cíl pohybu, nýbrž i množinu jednotek začínajících ve stejné oblasti a směřujících do stejného cíle. Rozvolněním podmínky vzájemné blízkosti jednotek v průběhu vykonávání plánu umožníme plánovači

(a) Skupina jednotek procházející úzkým průchodem. Dochází ke zdržení, protože většina jednotek čeká, až na ně přijde řada. Malá část však přetéká do vedlejší cesty.



(b) Navigační graf reprezentující stejnou situaci. Zelené oblasti představují vrcholy a fialové spoje hrany. Navigace ve hře zvolila druhou cestu zleva. Široké cesty na krajích zůstávají zcela nevyužité.

Obrázek 1: Starcraft 2 [2010]

rozprostřít jednotky po mapě.

Protože navigace v počítačových hrách má kromě plánovací části i část exekuční, naprogramujeme herní prostředí, v němž budeme srovnávat naše řešení založené na tocích v síti s navigačními metodami běžně používanými v počítačových hrách. Cílem této práce je tedy naprogramovat simulátor, vytvořit vhodný algoritmus založený na tocích v síti a vyhodnotit jeho přínos.

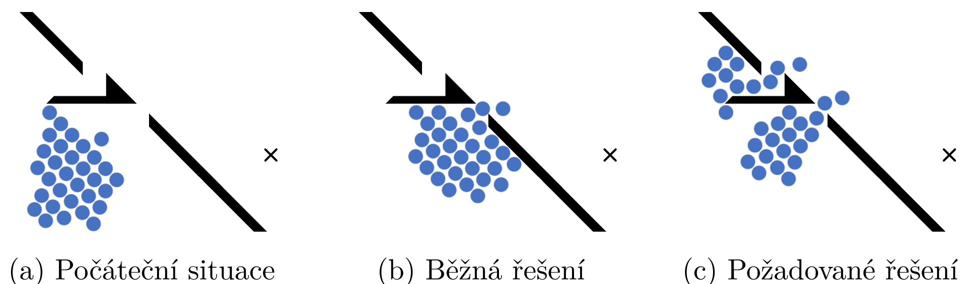
Obrázek 2: Snímek obrazovky ze hry Empire Total War [2009]. Na obrázku je vidět uvízlá část vyslané skupiny koní v úzkém průchodu mezi domem a kládami. Příliš mnoho jednotek se snaží projít úzkým místem, třebaže existují alternativní, navíc širší a volnější cesty. Příčinou této chyby nastalé při exekuci může být neoptimální rozvržení cest jednotek ve fázi plánování.

Tato práce je členěna do několika částí. Nejprve si zadefinujeme problém a určíme, jaké požadavky klademe na simulátor. Ve druhé kapitole se seznámíme s oblastí toků v síti, abychom mohli kvalifikovaně rozhodnout, jakým směrem by se měla ubírat implementace. Ve třetí kapitole analyzujeme problémy spojené s navigací v počítačových hrách. Čtvrtá kapitola popisuje námi implementovaný simulátor s jeho funkcemi a vysvětluje, jaká jsme provedli implementační rozhodnutí. V páté kapitole rozebíráme detailněji naši metodu. Šestá kapitola se zabývá tím, jak jsme prováděli experimenty, jakým způsobem jsme je vyhodnocovali a seznamuje s výsledky našeho testování (evaluace). Sedmá kapitola obsahuje uživatelskou dokumentaci, jež provází přípravou vlastního experimentu i jeho spuštěním, a programátorskou dokumentaci, v níž je představena architektura programu. Poslední kapitolou je závěr s konečným vyhodnocením našeho úkolu.

1. Formulace problému

Cílem práce je prozkoumat, zda je možné použít ideje z teorie grafů (přesněji toků v sítích) k řízení skupiny jednotek v RTS hrách. Jelikož se v této práci snažíme nalézt řešení, jak navigovat větší skupinu jednotek mezi dvěma místy v mapě, zaměříme se na případy, ve kterých existuje mezi místem, v němž se nachází skupina, a cílem, více než jedna cesta. Požadujeme, aby místa od sebe byla dostatečně vzdálena nebo oddělena tak, aby se nevyskytovala v jedné oblasti. Skupina by měla být dostatečně veliká, aby pokryla alespoň nejužší místo cesty. Proto by se nemělo jednat o volné prostranství nebo příliš širokou plochu. V ní by nemělo smysl hledat více než jednu cestu.

Protože navigace v počítačových hrách řeší i provedení plánu, je potřeba otestovat kvalitu řešení. Za tímto účelem by bylo vhodné použít stávající hru a naprogramovat vyhledávání v jejím kontextu, to však bohužel není možné, protože vhodné hry jsou proprietární a nikoli open source. Proto bude nutné připravit i testovací prostředí.



(a) Počáteční situace (b) Běžná řešení (c) Požadované řešení

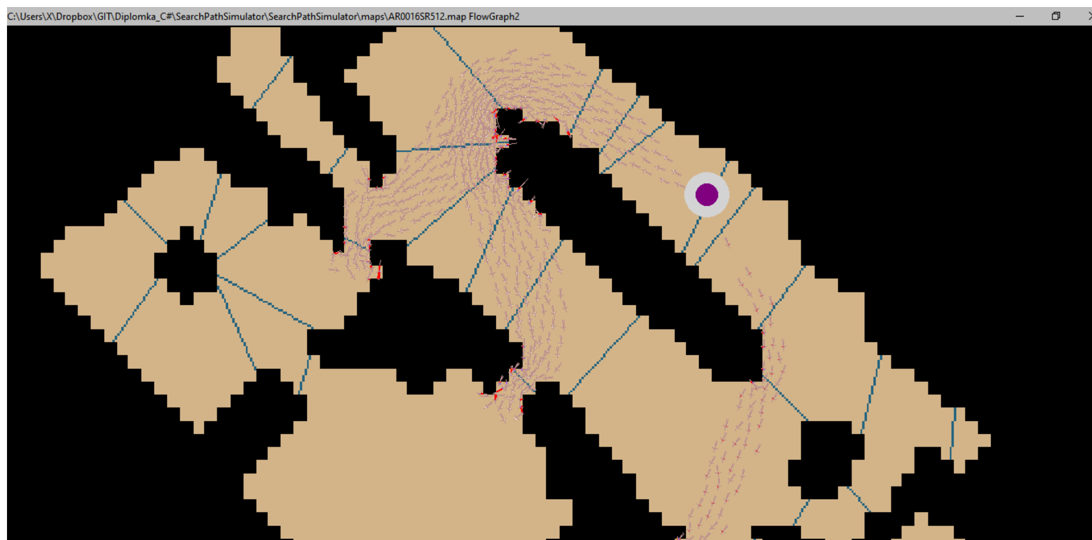
Obrázek 1.1: Pro dostatečně velkou skupinu je rychlejší poslat část jednotek po alternativní cestě, jinak jednotky vzadu čekají, dokud jednotky před nimi neprojdou zúžením. Modře jsou zobrazeny jednotky, černě překážky, křížek označuje pozici cíle.

1.1 Abstrakce počítačové hry

K tomu, abychom mohli porovnání provést, je nutné mít k dispozici prostředí simulující běžnou hru, v němž bychom prováděli experimenty. Nepotřebujeme, aby obsahovala velké množství herních prvků. Postačí, aby se jednalo pouhou abstrakci počítačové hry umožňující simulovat ty části hry, které jsou relevantní pro plánování pohybu jednotek a jeho provedení. Těmito částmi jsou především hráčem ovládané jednotky a samotné herní prostředí.

Simulátor bude poskytovat informace o průběhu navigace při použití různých metod, aby se na základě jeho výstupu mohlo přistoupit k vyhodnocení těchto algoritmů. Dále by měl umožnit přípravu testu, který bude moci být opakovaně vykonán, a to i s různými metodami. Výstupy z těchto testů by měl umět uchovat pro umožnění pozdějšího vyhodnocení.

V následujících odstavcích pojednáme o aspektech běžných RTS her ve vztahu k tomuto testovacímu simulátoru.



Obrázek 1.2: Simulátor během simulace, barevný terč označuje cíl jednotek.

1.1.1 Mapa

Žánr preferuje dvoudimenzionální herní prostředí. Samotné prostředí se může skládat z více typů políček libovolných tvarů i velikostí. Většina her však používá pravidelná n-úhelníková políčka stejné velikosti. My pro jednoduchost implementace zvolíme čtvercová políčka, která jsou bez mezer poskládána vedle sebe a tvoří celou herní mapu. Pokud by mapa nebyla obdélníková, lze ji na obdélník upravit přidáním dalších políček. Proto se naše uvažované herní prostředí bude skládat z obdélníkové mapy tvořené stejně velkými čtvercovými políčky.

Ve skutečných hrách se může vyskytnout více typů polí. Kromě základních polí volného prostoru a neprůchodného prostoru (skály) se může jednat o pole s vodou či brody. V našem případě by další typy polí pouze zvýšily složitost modelu. Jednotky (lodě, vozy, koně) by ve svých definicích musely mít určeno, na která pole (a případně na jak dlouhou dobu) mohou vstoupit a jakým dalším způsobem je daná pole ovlivňují (rychlost či doba, po které dojde k utonutí). Proto nám postačí, že budeme užívat mapy pouze se dvěma typy polí, která jednotkám určují pouze to, zda se na daném políčku mohou či nesmí vyskytnout.

1.1.2 Jednotky

I když se ve hrách objevují jednotky různých tvarů, pracuje se s nimi jako s konvexními objekty. Pro náš simulátor zvolíme jednotky kruhové, jednotkové velikosti, které mají svou pozici definovanou pomocí neceločíselných souřadnic, a tudíž se mohou vyskytovat na více políčcích najednou. Důsledkem tohoto rozhodnutí je, že se pohyb jednotek bude více jevit jako spojitý.

Uživatel simulátoru bude moci stanovit umístění jednotlivých jednotek, tím i jejich počet a jejich cílovou pozici.

1.1.3 Skupiny

Skupina jednotek je neprázdná množina jednotek, kterou hráč ovládá simultánně. Jednotky se nacházejí na určitých výchozích pozicích a je-li hráčovým

záměrem je přesunout na jinou pozici, obdržíme od něj jednu pozici cíle platnou pro celou skupinu. Protože interakce mezi skupinami jde mimo rámec této navigace, není třeba uvažovat o existenci jiných skupin či jednotek na mapě, a tak veškeré jednotky budou náležet do jedné skupiny. To znamená, že bude existovat právě jeden cíl pro všechny jednotky.

1.1.4 Naplánování cest jednotkám ve skupině

Plánování pohybu ve hře zajišťuje plánovač (planner). Jeho vstupem je mapa, skupina jednotek a její nově požadovaná pozice. Stejně tak v simulátoru tento plánovač pohybu každé jednotce vytyčí její vlastní trasu. Samotné vykonání postoupí hernímu enginu. Zatímco plánovač bude pro každý z porovnávaných algoritmů jiný, následující herní engine i vše ostatní bude pro jednotky sledující předem naplánovanou trasu identické. Toto nám umožní otestovat chování algoritmů, a poté je porovnat.

Na otázku, zda lze na mapu nahlížet jako na graf a zda náš problém připomíná některé problémy řešené v teorii sítí, si podrobněji odpovíme v následující kapitole.

2. Toky v sítích

V této sekci se seznámíme s existujícími modely sítí, abychom analyzovali problémy, které bychom mohli pomocí daných modelů v naší úloze řešit. Zajímá nás, zda by tyto sítě mohly nalézt své uplatnění v počítačových hrách. Nakonec u každého modelu zhodnotíme, nakolik je relevantní pro naši úlohu hledání cest pro skupinu jednotek. Podrobnější využití toků v sítích je popsáno v páté kapitole.

2.1 Definice

Toky v sítích jsou studovány přinejmenším od padesátých let 20. století, kdy Ford a Fulkerson [1] představili algoritmus na řešení maximálního toku.

2.1.1 Definice toků

Definice 1 (Graf)**.** *Graf je dvojce množin vrcholů a hran* $G = (V,E)$*. Hrana spojuje dva vrcholy* $\forall e \in E : e = (a,b)\ ;\ a,b \in V$ *. V orientovaném grafu záleží na uspořádání vrcholů v hraně.*

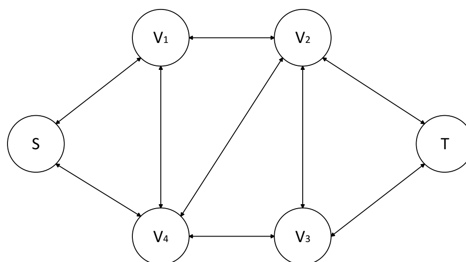
Definice 2 (Síť)**.** *Síť je uspořádaná pětice* (G,s,t,c,l)*. Kde* G *je orientovaný graf,* $s,t \in V$ *zdroj a stok (source, terminal) a funkce* $c : E \to \mathbb{R}^+$ *určuje kapacitu hran,* $l : E \to \mathbb{R}^+$ *určuje délku hran.*

Definice 3 (Kirchhoffův zákon – zákon zachování toku)**.** *Pro všechny vrcholy s výjimkou zdroje a stoku platí, že součet příchozích toků se rovná odchozím. (Tok ve vrcholu nevzniká, ani se neztrácí).*

$$\forall v \ \in V \ \setminus \{s,t\} \sum_{(u,v)^+ \in E} f_{(u,v)} = \sum_{(v,u)^- \in E} f_{(v,u)}.$$

Definice 4 (Tok v síti)**.** *Tok je funkce* $f : E \to \mathbb{R}_0^+$*, splňující Kirchhoffův zákon a je shora omezen kapacitou jím použitých hran* $\forall (u,v) \in E : f(u,v) \leq c(u,v)$*.*

Definice 5 (Saturace cesty)**.** *Hrana je saturovaná, pokud se její kapacita rovná toku.* $(u,v) \in E : f(u,v) = c(u,v)$*. Cesta je saturovaná, pokud alespoň jedna hrana z cesty je saturovaná.*



Obrázek 2.1: Příklad sítě

2.1.2 Aplikace definic sítě

Nejdříve ověříme, zda lze aplikovat definici sítě 2 na náš problém. Dekomponujme mapu na diskrétní oblasti, každé oblasti bude odpovídat jeden vrchol. Sousedící oblasti spojíme hranou. Tímto dostáváme graf, ve kterém nám zatím chybí zdroj a stok. Zdrojem označme vrchol, v němž je výchozí pozice skupiny a stokem vrchol reprezentující oblast, v níž se nachází pozice cíle. Dále nám chybí funkce kapacity a délky. Kapacitou neboli maximálním možným průtokem hran označme maximální počet jednotek, které jsou schopny přes spoj oblastí projít za časový úsek. Tento časový úsek bude odpovídat době potřebné k uražení vzdálenosti rovnající se velikosti jednotky dělené maximální rychlostí jednotek. Délkou hrany označme vzdálenost mezi středy oblastí. Tímto jsme ukázali existenci převodu mapy na graf. Kirchhoffovy zákony zjevně budou fungovat, pokud nebudou přibývat ani ubývat jednotky.

2.2 Vybrané problémy

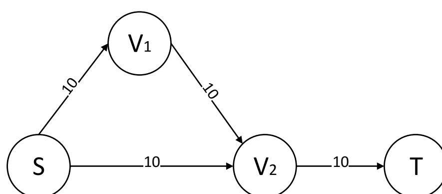
Seznamme se nyní s vybranými problémy, které by nám při řešení naší úlohy mohly poskytnout lepší vhled či řešení.

2.2.1 Problém maximálního toku

Úvodní duální problém při studiu toků je problém maximálního toku či minimálního řezu. Řez je podmnožina hran $R \subseteq E$ taková, že rozděluje množinu vrcholů na dvě disjunktní části, v níž se v jedné nalézá zdroj a v druhé stok. Kapacita řezu odpovídá maximálnímu toku.

Pro libovolnou síť odpovídá problém maximálního toku na otázku, jaké největší hodnoty může mít tok v této síti mezi zdrojem a stokem. Řešení tohoto problému má časovou složitost $\mathcal{O}(|V| * |E|)$ [2].

V námi definovaném převodu odpověď vyjadřuje množství jednotek, které je síť schopná najednou doručovat, tedy propustnost mapy. Maximálního toku sítě lze dosáhnout použitím různých hran (viz obrázek 2.2). A tedy nemáme k dispozici informaci o tom, jaké hrany máme pro plánování použít. Navíc nevyužíváme znalosti délky hran.



Obrázek 2.2: V řešení problému maximálního toku může tok hrany (S,V_2) nabýt libovolné hodnoty od 0 do 10. Tímto ilustrujeme, že použité hrany a hodnoty toku na těchto hranách mohou být v řešení problému maximálního toku různé.

2.2.2 Dynamické toky – toky v průběhu času

Dynamické toky se v literatuře vyskytují pod dvěma názvy: „Dynamic flows“ a „Flows over time“. Zabývají se proměnlivým prostředím. Lze jimi řešit sítě, u kterých se v čase mění jednotlivé váhy hran, nebo tok trvá nenulový čas. V našem případě přesun jednotek, který trvá nějak dlouho.

Dynamická síť

Řešení Forda a Fulkersona [3] je založeno na diskretizaci času. Používá v čase rozvinutou síť, kde se pro každý časový okamžik vytvoří kopie celé sítě. Ale aby se síť mohla rozvinout, je potřeba, aby délky hran byly celočíselné. Na rozvinutou síť se k řešení používá přímo algoritmus pro statický tok. Předem je nutné zvolit parametr času T, pro který se spočítá optimální tok v čase T, my ovšem celkový potřebný čas T předem neznáme.

Spojitý čas pro dynamickou síť

Dynamické sítě lze řešit ve spojitém čase, ale na základě výzkumu Onura Özgüna a Yamana Barlase [4], kteří zkoumali v jakém případě se vyplatí použít spojitou simulaci, víme, že rozdíl spojité simulace oproti diskrétní simulaci pro náš problém nepřinese kýžený užitek. Naše simulace tedy bude moci pracovat v malých diskrétních krocích.

2.2.3 Transshipment

Transshipment je zobecněním toků, ve kterém je rozšířena množina zdrojů a stoků na více než jeden. Tímto bychom mohli lépe namodelovat výchozí pozici jednotek. Bylo by možné modelovat každou jednotku vlastním vrcholem a ten hranou spojit s vrcholem, který nyní označujeme jako zdroj za současného odznačení původního zdroje. Alternativně lze toto řešení použít pouze v případě, pokud by se jednotky nacházely na rozhraní oblastí. V takovém případě bychom opět odznačili zdroj a jako nový zdroj bychom přidali vrchol každé části skupiny, nacházející se v různých oblastech. Takto vzniklé zdrojové vrcholy bychom spojili s oblastí, ve které se jednotky nachází. Těmito způsoby by se nám mohlo podařit získat přesnější plán.

2.2.4 Dělitelné toky

V následujících odstavcích si představíme příbuzné problémy, které všechny spojuje problematika dělení toku.

Nedělitelný tok

Problém nedělitelného toku se zabývá hledáním největšího toku, který se na cestě mezi zdrojem a cílem nerozdělí. Nezdá se, že by nám tento problém mohl něčím přispět, vždyť usilujeme o pravý opak, ale na mapě s velkým počtem úzkých a krátkých cest k cíli by širší, ale delší cesta vyřešila problém uvíznutí také. Navíc, pokud bychom tento algoritmus opakovaně volali na síť, ze které

bychom vždy odstranili saturované hrany, mohlo by to dobře aproximovat optimální řešení.

K-dělitelný tok

Pro náš cíl je možná zajímavější formulace „k-splittable“ toku, který umožňuje omezit štěpení skupiny na předem známý, nebo dokonce hráčem nastavitelný počet.

K-balancovaný tok

Z opačného pohledu přichází „k-balanced flow“, což je tok, který musí být dekomponovatelný na k-tice hranově disjunktních cest. Náš problém nepotřebuje inherentně disjunktní cesty, stačí, když budou disjunktní v blízkosti zúžení.

2.2.5 Problém maximálního toku s minimální cenou

Tento problém je obdobný hledání maximálního toku s tím rozdílem, že z maximálních toků se vybere ten s minimální cenou. Tato změna je však zásadní. Doposud jsme se seznámili s problémy, které neřešily délku toku. Problém maximálního toku s minimální cenou umožňuje do ohodnocení hrany zakódovat délku cesty. Vyřešením tohoto problému dostaneme podmnožinu hran, které byly použity. Na rozdíl od maximálního toku již víme, že daný tok využívá nejkratší cesty, ale i těch může být více, takže nejednoznačnost, kterou jsme ilustrovali obrázkem 2.2, stále přetrvává. (Stačí nastavit stejnou cenu mezi vrcholy S a V_2 pro dvojci hran i pro samotnou hranu.)

Pokud sítí pouštíme postupně se zvyšující tok, dosáhneme posloupnosti toků, které jsou pro danou velikost toku vůči ceně optimální. Pokud se pak zaměříme na jednotlivé toky v posloupnosti, porovnáním se sousedními lze vidět, kudy tok přibývá.

Tento problému lze řešit v polynomiálním čase [5], [6], jeho složitost při opakovaném hledání nejkratší cesty je $\mathcal{O}(m * log\ n\ (m + n\ log\ n))$.

Složitost hledání nejkratší cesty pro Dijkstův algoritmus s použitím Fibonacciho haldy je $\mathcal{O}(m + n\ log\ n))$, kde m a n jsou počty hran a vrcholů. A toto hledání se provádí maximálně $\mathcal{O}(n * log\ n)$.

2.2.6 L-Bounded tok

Dalším problémem, ve kterém se řeší délka toku je „L-Bounded flow“ [7], [8]. Existuje varianta pracující s implicitní jednotkovou délkou hran (lze řešit v polynomiálním čase) a složitější varianta (NP- těžká), která každé hraně přiřazuje její délku. L-Bounded tok hledá maximální tok, který může být dekomponován na toky s cestami s délkou maximálně L, čímž by nám mohl pomoci omezit prohledávanou oblast.

2.2.7 Problém nejrychlejšího toku

„Quickest flow problem“ [9], [10], [11] je problém, který řeší jak co nejrychleji poslat určité množství toku sítí, ze zdroje do stoku. Opět se jedná o NP-těžký

problém. To znamená, že pro účely počítačových her je tato složitost příliš vysoká. Ale pokud by hodnoty na hranách byly celočíselné, lze problém řešit v čase $\mathcal{O}(nm\ log(n^2/m)\ log(nL))$, kde L je nejvyšší hodnota délka hrany.

3. Analýza souvisejících problémů

V této kapitole se nejprve seznámíme, v jakých strukturách hledáme cestu, a poté, jakým způsobem ji hledáme, tedy se způsoby reprezentace grafu a herní mapy metodami vyhledání nejkratších cest. Následně se zaměříme na způsob, jak se ve hrách provádí pohyb, a na chyby, které při něm nastávají.

Plánování pohybu se v RTS hrách dělí na dvě části: plánování cesty a její provedení (v robotice by přibyla ještě část lokomoce). Jelikož se v této práci zaměřujeme na real-time hry, je nutné usilovat o minimalizaci času. To při vývoji her vede až k tomu, že se v assembleru optimalizují kritické části, aby v nich nebyla žádná zbytečná instrukce. V sekci o vylepšeních vyhledávacích algoritmů si představíme metody, které se v současné době používají při předzpracování, aby následné vyhledání bylo co nejrychlejší.

3.1 Reprezentace herního světa

Mapu je potřeba z textového souboru převést do nějaké efektivnější reprezentace. Herní svět lze vyjádřit pomocí grafu. Existují různé typy grafů reprezentující stejný svět. Zvolená reprezentace ovlivní rychlost výpočtu, dokonce i algoritmy, které budeme moci později použít. To platí i pro uložení grafu do paměti. V našem programu budeme chtít využít graf k reprezentaci mapy i pro strukturu, ve které bude probíhat hledání cest.

3.1.1 Fyzická reprezentace mapy

Nyní se zaměříme na reprezentaci herního světa grafem, a to pomocí matic, polí (1D matic), gridu či grafu objektů.

Matice souslednosti je čtvercová matice, ve které pozice $[i,j]$ vyjadřuje váhu hrany jdoucí z i-tého do j-tého vrcholu.

Seznam následníků obsahuje dvě pole celých čísel. Vrcholy i hrany jsou očíslovány. Pole vrcholů obsahuje na i-té souřadnici index do pole hran, na jehož pozici jsou následníci daného vrcholu.

Matice incidence je obdélníková matice $m \times n$. Každá hrana má ve svém sloupci právě na dvou řádcích patřících vrcholům, mezi kterými vede, nenulovou hodnotu. Hodnota může vyjadřovat směr hrany v orientovaném grafu nebo v neorientovaném váhu hrany.

Gridová reprezentace

Pro reprezentaci podkladové mapy zvolíme tuto reprezentaci. V našem případě je mapa dvojrozměrnou, plně spojenou souřadnicovou sítí (2D grid), takže každé políčko má 8 sousedů, z toho 4 sousedé leží na úhlopříčce. Tím, že známe rozměry mapy a způsob spojení grafu, nemusíme tyto informace nikde explicitně

uchovávat. Na pozici $[i,j]$ je uložená binární hodnota určující, zda na dané pole smí jednotka vstoupit či nikoli.

K uložení grafu sloužícího pro vyhledání cest je tato reprezentace nevhodná, protože zde je implicitně uchována struktura grafu.

Objektově orientovaný přístup reprezentace

Protože předem neznáme, jaký největší možný stupeň vrcholu se může vyskytnout, je zapotřebí uložit všechny následníky do nějaké rostoucí struktury, například zvětšujícího se pole. Poté se již moc neliší od seznamu následníků. Objektově orientovaný přístup k uložení grafu není pro velké grafy příliš vhodný z výkonnostních důvodu. Na druhou stranu umožňuje přímočaré uložení dalších informací přímo k dané hraně, a zpřehledňuje tak implementaci. Díky této skutečnosti a předpokladu, že graf pro vyhledávání cest bude dostatečně malý, jsme zvolili tuto reprezentaci.

3.1.2 Logická reprezentace grafu

Nejpřirozenější způsob reprezentace grafu mapy je pomocí gridové reprezentace zmíněné v podsekci 3.1.1. Ostatní používané způsoby jsou komplikovanější na předzpracování, ale poskytují lepší vlastnosti. Mezi ně patří Navmesh a Waypoint navigační grafy. Výhoda předzpracování spočívá v dramatickém snížení velikosti stavového prostoru. Například pro reprezentaci prázdné místnosti v mapě o rozměrech 100×100 není třeba mít 10 000 stavů, ale postačí jich jen několik. Nevýhodou těchto metod je nutnost předzpracování mapy. V případě, kdy hráč může ovlivňovat průchodnost políček (např. tím, že na nich postaví budovu), je nutné tyto grafy komplikovaně aktualizovat.

Navigační graf

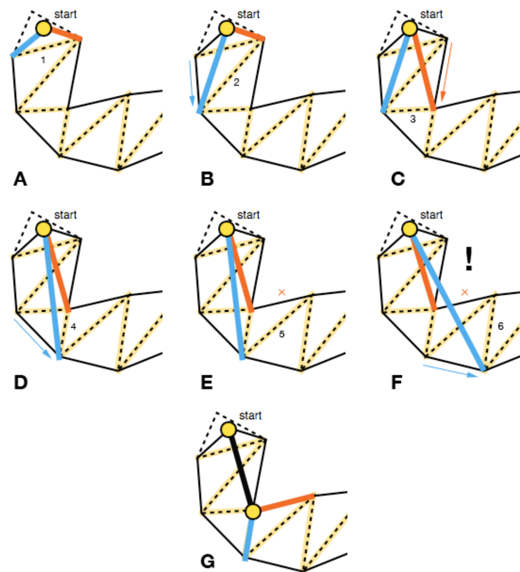
Navigační graf je tvořen navigačními body (waypoints) a hranami. Navigační body a hrany se umisťují do herního světa do volného prostoru s cílem, aby jednotky chodily po hranách a procházely právě těmito body. Lze je generovat automaticky, ale často bývají do mapy ručně umisťovány mapovými designéry. Tento graf umožňuje zrychlit plánování v mapě a docílit realističtěji působícího pohybu než po nejkratší cestě. Pokud například chceme simulovat chůzi humanoida v objektu, je zřejmé, že humanoid držící se při zdi nevypadá příliš přirozeně. Naopak cesta, která drží humanoida blíže středu chodby, působí reálněji. Počítačové hry se proto snaží o navození živěji působící scény, která hráče jako diváka vtáhne do děje. Negativem však je, že se všechny jednotky snaží projít vždy stejnými body, a to nepůsobí reálně, především proto, že se jednotky pohybují po stejné dráze.

NavMesh

NavMesh je zkratka z anglického Navigation Mesh, což doslova znamená navigační pletivo či síťovina. Jedná se o způsob reprezentace herního světa pomocí konvexních n-úhelníků pokrývajících souvislé plochy. Celá plocha herního světa se jimi pokryje a vyhledávání se pak provádí na dvou úrovních. Lokální, uvnitř n-úhelníku, je triviální, protože vnitřek neobsahuje žádnou překážku, takže se

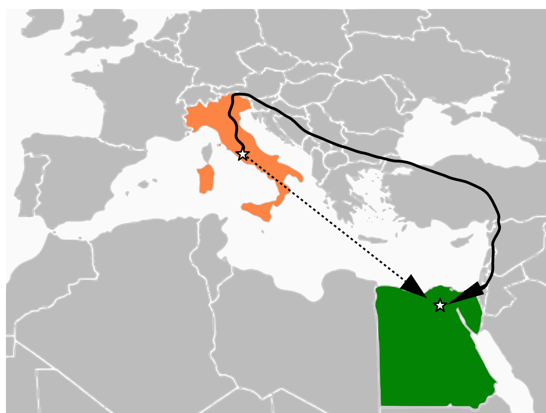
(a) Waypoint (b) Navmesh

Obrázek 3.1: Zobrazení navmesh a waypoint na stejné mapě.



Obrázek 3.2: Záznam výpočtu Funnel algoritmu

lze pohybovat přímo. Globální se, pokud bychom přidali vždy do středu oblasti jeden waypoint, od waypoint hledání neliší. Obecně to však neplatí a vyhledávací algoritmus v mapě je nutné přizpůsobit na míru (viz [12], [13], [14]). Po vyhledání nejkratší cesty víme, jakými plochami dojít k cíli, ale stále je potřeba dovyhledat cestu přes jednotlivé plochy (na obrázku 3.1 by to byla oranžová spojnice bodů). K dovyhledání cesty se používá Funnel algoritmus [15], který využívá znalosti toho, že oblasti jsou konvexní, a tak stačí kontrolovat pouze jejich rohy. Pro trojúhelníkový NavMesh Funnel algoritmus hlídá, zda se nepřekřížily úhly ve směru vedoucímu k cíli. Na obrázku 3.2 je taková situace zobrazena v kroku `F`. V tomto kroku se aktuální vrchol označí jako dílčí cíl na cestě mezi startem a cílem, a z tohoto bodu se dále pokračuje v hledání.



Obrázek 3.3: Příklad vstupu, u kterého relaxace plánování vrací zcela chybný výsledek. V relaxovaném problému, ve kterém ignorujeme mapu, hledáme nejkratší cestu autem z Říma do Káhiry bez použití jiného dopravního prostředku (trajektu). Přerušovaná čára symbolizuje řešení relaxačního problému, zatímco spojitá skutečného problému. Řešení relaxovaného problému nás bude směrovat vzdušnou čarou na jihovýchod, i když ve skutečnosti nejkratší cesta vede opačným směrem.

3.2 Plánování

Plánování je informatická disciplína umělé inteligence, která se zabývá i řešením obecného problému (General Problem Solving). Složitost těchto problémů je často mimo třídu NP-úplných problému, obecně jsou tyto problémy dokonce nerozhodnutelné. Plánovací problém obsahuje vstupní stav prostředí, popis možných akcí a cílové podmínky. Plánování probíhá v prostoru plánů, ve kterém se hledá podmnožina splňující zadaná kritéria. Plánovací algoritmy tento velký prostor systematicky prohledávají, využívají nejrůznější techniky vedoucí ke zredukování jeho velikosti nebo k prioritnímu procházení míst prostoru, kde se pravděpodobněji vyskytuje řešení.

Hledání cest v grafu je speciálním případem řešení obecného problému. V této sekci se blíže podíváme na některé techniky vedoucí ke zrychlení nalezení řešení. Plánovací problém hledání nejkratší cesty obsahuje: mapu, startovní pozici, možné pohyby jednotky, cílovou pozici a navíc ze všech cest, které toto splňují, hledáme nejkratší možnou cestu. Těch však díky symetrii může být více.

Běžným přístupem řešení úloh v plánování, ve kterém se často k vyřešení problému uvažuje obdobný, ale zjednodušený problém, je relaxace problému. Zjednodušený problém, ve kterém došlo ke zmírnění omezujících podmínek, se vyřeší pro každou možnost, mezi kterými se algoritmus procházející prostor plánů aktuálně rozhoduje a na základě řešení relaxovaných problémů rozhodne, kam směřovat prohledávání v původním problému. Příkladem může být heuristika používaná v A^* algoritmu, kde algoritmus hledá nejkratší cestu k cíli v zadané mapě. Relaxace tuto mapu přehlíží a směruje vyhledávací algoritmus přímým směrem k cíli, jako by v mapě existovalo přímé spojení mezi všemi políčky. Taková relaxace ignorováním mapy není neomylná, jak je vidět na obrázku 3.3.

3.2.1 Hledání cesty v grafu

V plánování se hojně využívá hledání nejkratší cesty. Nejkratší cesta se hledá v grafu, jenž reprezentuje prohledávaný prostor. Mapa se do tohoto prostoru převádí z toho důvodu, aby se zrychlilo nebo jinak vylepšilo vyhledávání. V gridu většinou existuje mnoho nejkratších cest, ale algoritmy vrátí pouze jednu. V počítačovém průmyslu někdy dokonce nevadí, pokud nedostaneme nejkratší cestu.

V tomto oddíle se budeme zabývat pathfinding algoritmy. V herním průmyslu byl dříve de facto standardem Dijkstrův algoritmus, poté ho nahradil $A*$ a jeho další různá rozšíření $IDA*$, hierarchické plánování a další.

Dijkstrův algoritmus

Nejjednodušší metoda vyhledávání v grafu je prohledávání do šířky, rovněž známá pod názvem Breadth first search nebo zkráceně BFS. Pokud tento algoritmus, jehož předpokladem je, že všechny vzdálenosti v grafu jsou stejné, rozšíříme o možnost mít na hranách různé délky, zobecníme ho na Dijkstrův algoritmus. Dijkstrův algoritmus existuje ve více verzích. V jedné z nich se hledají cesty ze zadaného vrcholu do všech ostatních vrcholů. V alternativní verzi algoritmu se toto hledání ukončí, když je spočítána vzdálenost do koncového vrcholu. Vstupem algoritmu jsou tři parametry: orientovaný graf G, počáteční vrchol s a koncový vrchol t. Označme zápisem $d(a,b)$ funkci, která každé hraně mezi libovolnými vrcholy a a b přiřazuje její délku.

Idea hledání nejkratší cesty Dijkstrova algoritmu je následující: Během výpočtu se do prioritní fronty OTEVŘENÉ ukládají všechny vrcholy v, na které se již narazilo, ale zatím se neuzavřelo, jaká je jejich minimální vzdálenost k počátečnímu vrcholu s. V této frontě jsou vrcholy setříděny dle jejich právě známé minimální vzdálenosti k vrcholu s. Vzdálenost $h(v)$ je na začátku nastavena všem vrcholům, kromě vrcholu s, na nekonečno. Algoritmus z fronty odebere první vrchol a dále se zaměří na jeho sousedy. Již uzavřené vrcholy algoritmus dále nezpracovává. Do fronty přidá zatím neotevřené vrcholy. Neuzavřeným vrcholům, které se již ve frontě nachází a zmenšila se jim nejkratší cesta k vrcholu s, upraví jejich minimální vzdálenost. Takto se postupuje v závislosti na variantě algoritmu, dokud se z fronty neodebere koncový vrchol, nebo dokud se fronta nevyprázdní. To může nastat i v prvním případě, pokud by s a t leželo v různých komponentách souvislosti.

```
OTEVŘENÉ ← s

Dokud OTEVŘENÉ obsahuje nějaký vrchol:
    v ← OTEVŘENÉ
    Pro všechny sousedy n vrcholu v:
        Pokud h(n) > h(v) + d(v,n):
            h(n) := h(v) + d(v,n)
            OTEVŘENÉ ← n
```

Pseudokód 1: Idea Dijkstrova algoritmu.

Algoritmus A^*

Algoritmus A^* vylepšuje Dijkstrův algoritmus o heuristickou funkci, která má za úkol usměrňovat hledání správným směrem.[16] Nejběžněji používanou heuristikou je již zmíněná relaxace problému ignorováním mapy (viz obr. 3.3).

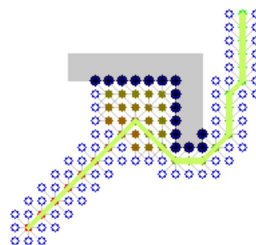
Rozdílné chování algoritmů můžeme ilustrovat na situaci, ve které v prioritní frontě budou pouze dva vrcholy, a to stejně ohodnocené. Původní algoritmus z nich vybere víceméně náhodně, ale A^* vybere ten vrchol, který se heuristické funkci jeví blíže k cíli. Rozdílem oproti Dijkstrovu algoritmu je změna pořadí zpracování jednotlivých vrcholů s nadějí, že s lepší heuristikou se dosáhne výsledku rychleji, což obecně neplatí.[17] Pseudokód 1 rozšíříme o výpočet heuristické funkce $g(v)$ a řádek s výpočtem vzdálenosti $h(n)$ změníme jako v ukázce pseudokódu 2.

$$h(n) := h(v) + d(v,n) + g(v)$$

Pseudokód 2: Hlavní trik A^*, $h(n)$ nyní obsahuje odhad vzdálenosti.

Heuristická funkce Pozastavme se na chvíli u výběru heuristické funkce. Čím je daná funkce přesnější, tím lépe vede algoritmus k řešení. Prozkoumejme krajní případy. Pokud bychom měli neomylnou funkci, stačilo by expandovat vrcholy pouze po cestě v grafu. Jestliže by funkce vracela vždy konstantu, výhoda A^* by se ztratila a výpočet by byl ekvivalentní Dijkstrtovu algoritmu. A pro funkci odpovídající (záměrně) zcela opačně by se před nalezením skutečně nejkratší cesty prohledal celý vyhledávací prostor.

Algoritmus A^* je optimální (najde správné řešení) pouze pro přípustnou heuristiku. Aby heuristika byla přípustná, nesmí být její odhad nikdy vyšší než je skutečná cena dosažení koncového vrcholu. Dále je vhodné, aby byla konzistentní (platí v ní trojúhelníková nerovnost), jinak by mohlo docházet k opětovnému otvírání již uzavřených stavů.



Obrázek 3.4: Výsledná suboptimální cesta nalezená A^* algoritmem používající konzistentní, ale nepřípustnou heuristiku.

Pro úsporu výpočetního času se však v počítačových hrách používají i heuristiky, které nejsou ani přípustné. Vede to k suboptimálním řešením, tedy k cestám, které do cíle vedou, ale nejsou nejkratší (viz obr. 3.4). Tato na první pohled kritická vada hledání nejkratší cesty často nemá žádné negativní důsledky, protože v závislosti na konkrétní doméně se může jednat o následující důvody: rozdíl v délkách obou cest je vzhledem k jejich celkové délce zanedbatelný, pohyb jednotky do cíle se nedokončí, nejedná se o statický cíl, tedy se jeho pozice mění,

část, starající se o exekuci pohybu, si před nevhodným místem vynutí přepočet cesty z aktuální pozice.

Algoritmus IDA*

Algoritmus IDA* zapouzdřuje vícero volání upraveného A* algoritmu, který rozšiřuje o parametr omezení hloubky a kontrolu před přidáním vrcholu do fronty OTEVŘENÉ. Vrchol n přidá pouze v případě, že jeho hodnota $h(n)$ je menší, než je současné omezení hloubky. Výsledek může znamenat, že cesta pro zadanou hloubku neexistuje. IDA* cyklicky volá takto upravený A*, dokud nedospěje k výsledku. Výhoda IDA* oproti A* spočívá v omezení spotřeby paměti.

```
h ← optimistický odhad hloubky řešení
Dokud existuje možnost, že existuje cesta:
    v ← A*(h)
    Pokud v obsahuje cestu nebo cesta neexistuje:
        Vrať řešení.
    Jinak (skončilo se kvůli omezení hloubky):
        h++
```

Pseudokód 3: Pseudokód IDA* algoritmu.

Algoritmy Hierarchického plánování

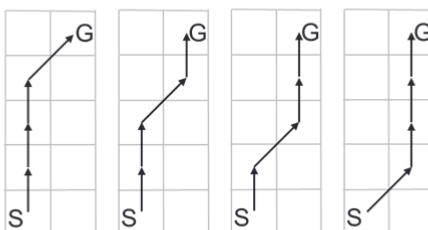
Hierarchické plánování vylepšuje A* o plánování na různých abstraktních úrovních namísto stavového prostoru. Při předzpracování se mapa rozdělí na menší oblasti. Úrovní může být více, kdy se oblasti mohou dělit na ještě menší části. Příkladem takového dělení může být technika *maxdegree*, využitá v [18], kde vytvoření hierarchie probíhá tak, že se dohromady spojují sousední stavy v určité vzdálenosti pod stav s největším stupněm. To se opakuje tak dlouho, dokud všechny stavy nejsou přiřazeny pod nějaký abstraktní stav. Dále se pro každou novou úroveň tento proces opakuje, dokud nezbude pouze jeden uzel.

Mezi nejmenšími částmi se spočítá optimální vzdálenost. Pro výpočet vzdáleností případných vyšších úrovní se použije předchozí úroveň. Při samotném hledání se neprochází přes jednotlivá políčka, ale přes jednotlivé oblasti. Nejprve se do abstraktního grafu přidá start a cíl. V tomto grafu se provede standardní A* algoritmus, cesta získaná přes vyšší úrovně abstrakce se převede do nižších, až se převede do stavového prostoru, ve kterém je posledním krokem uhlazení cesty. Takto nalezená cesta však nemusí být nejkratší možná. Například algoritmus hierarchického plánování [19], který nalezne cestu mezi startem a cílem řádově rychleji, nalezne cestu, která od nejkratší cesty není delší o více než 1%. Další nevýhodou je, že při skupinovém hledání cesty obdrží jednotky ze stejné oblasti skoro stejné cesty.

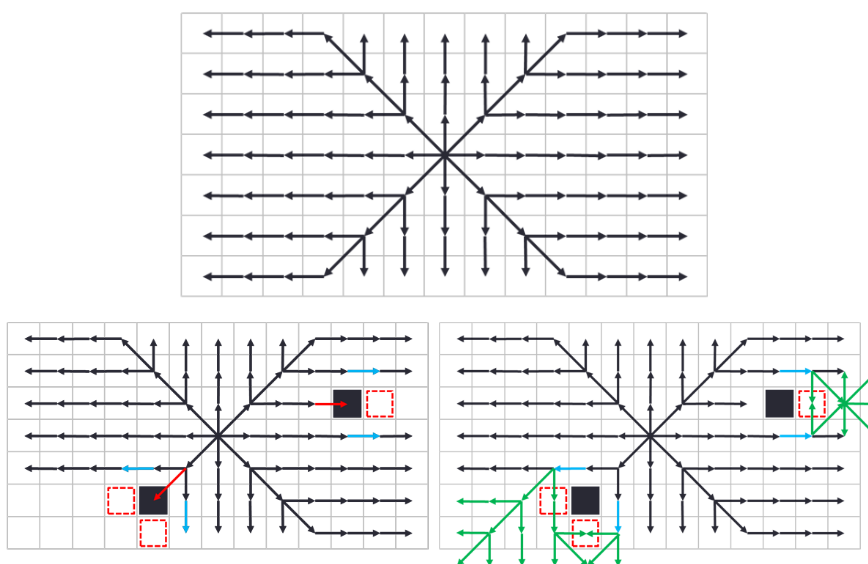
Jump point search + goal bounding box algoritmus

$JPS + GBB$ [20] vznikl kombinací dvou nezávislých algoritmů. Prvním je JPS+ algoritmus, který funguje pouze na gridech a zajišťuje, že se při výpočtu nenavštěvují redundantní cesty. Na obrázku 3.5 je příklad takových cest. Toho

dociluje pomocí speciální vyhledávací strategie, která je zobrazena na obrázku 3.6. Pokud není v cestě překážka, postupuje algoritmus po úhlopříčkách a odděluje horizontální či vertikální cesty. Narazí-li však na zeď, musí změnit své chování, jinak by některá políčka nebyla nalezena.



Obrázek 3.5: Všechny optimální cesty mezi políčky S a G.



Obrázek 3.6: Speciální vyhledávací strategie *JPS+* algoritmu. Černé políčko je zeď, modré šipky reprezentují místa, ze kterých se nově oddělí prohledávání. Červené čtverce jsou políčka, která byla kvůli blízké zdi nedosažitelná a zelené šipky označují, jaká políčka budou navštívena díky této změně strategie.

Druhá část *JPS+* si předzpracuje celou mapu a pro každé políčko spočítá všem osmi směrům vzdálenost ke zdi nebo ke speciálně přidaným značkám, které se nacházejí u míst, kde se algoritmu otvírá možnost vydat se směrem, který byl předtím blokován. Při samotném běhu se skáče především mezi těmito speciálními políčky namísto procházení všech políček.

Druhým algoritmem je Goal bounding box algoritmus, který zabraňuje prohledávání do zaručeně špatného směru. Před samotným hledáním proběhne preprocesing, při kterém se z každého políčka pustí upravený Dijkstrův algoritmus, který si navíc zapamatuje, z jakého políčka přišel. Tím vytvoří strom cest. Poté tyto jednotlivé cesty zabalí do obdélníkových bounding boxů. Při běhu algoritmu se před přidáním souseda do prioritní fronty otestuje, zda je možné, že cesta, na kterou se prohledávací algoritmus chystá vstoupit, vede nejrychleji k cíli.

3.2.2 Skupinové hledání cesty

Naivním přístupem k plánování skupiny jednotek je ignorování faktu, že se jedná o skupinu. Tento nejjednodušší přístup známý ze starších her je vhodný jen pokud skupiny nepřesahují počtem několik jednotek. Navigace probíhá tak, že si nejprve každá jednotka naplánuje vlastní cestu. A pokud nejkratší cesta povede přes nějaká úzká hrdla nebo se bude táhnout kolem velké překážky, pak se s velkou pravděpodobností nejkratší cesty jednotlivých jednotek budou překrývat. Navigace následně při exekuci tyto překrývající se cesty bude chtít využít současně, což povede k seřazení jednotek. Tento přístup označíme za nekoordinovaný pohyb skupiny.

Hra Age of Empires 2 jako jedna z prvních použila koordinovaný pohyb skupiny. Hra omezuje velikost skupiny na pouhých 40 jednotek. V rámci této skupiny se jednotky srovnají podle svých typů a utvoří řady, ve kterých za sebou postupuje jízda, pěchota, lučištníci a nakonec obléhací stroje. Jednotlivé jednotky udržují pozici vůči řadě, ve které se nacházejí a vůči ostatním řadám. Rychlost všech jednotek se řídí nejpomalejší jednotkou v rámci této skupiny. Toto řešení funguje dobře, pokud se skupina nedostane k úzkému místu. Úzká místa řeší hra podobným způsobem jako změnu cíle skupiny, dočasně vypne jednotkám jejich kolizní rádius, což umožní libovolnému množství jednotek být na stejném místě. Poté se jednotky znovu snaží zformovat do formace. Později vydané hry upravují toto chování povolením penetrace jednotky pouze do určité míry nebo přidávají další vylepšená chování jednotek, které si snaží vyjít vstříc tím, že pustí jinou jednotku.

Novější herní titul Company of Heros obchází problém plánování cest jiným způsobem. Nehledá nejkratší cestu k cíli, ale po cestě hledá nejrůznější úkryty, mezi nimiž se vojáci, kteří se navzájem kryjí, přesouvají. Další omezení spočívá v tom, že není možné ovládat jednotlivé jednotky, ale pouze skupinu jako celek. Skupina je velmi malá a její velikost je pevně stanovena.

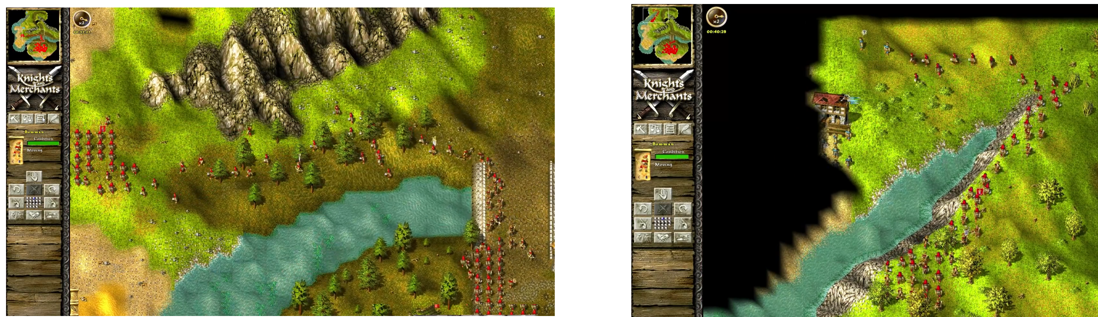
Dalším způsobem řešení navigace skupiny je metoda Leader following, ve které pozice jednotek ve formaci není fixní. Právě jedna jednotka ze skupiny je určena jako vedoucí. Té se jako jediné jednotce naplánuje cesta. Pohyb ostatních jednotek je řešen až v exekuční fázi, ve které se ostatní jednotky snaží být vedoucímu nablízku, ale zároveň mu dávají přednost a uhýbají.

3.3 Exekuce plánu jednotkami

Druhou částí navigace, přímo navazující na první část, ve které byla nalezla vhodná cesta mezi požadovanými místy v mapě, je exekuce. Ta provede připravený plán – přemístí jednotky z jednoho místa v mapě do druhého pomocí naplánované cesty.

Zatímco plánování cesty proběhne na začátku vykonání hráčovy instrukce pouze jednou (popř. pouze jednou pro každou jednotku), exekuční část navigace je spouštěna pro každý krok během celého průběhu hraní hry. Tato část zajišťuje každý, téměř nepatrný, pohyb jednotek. Řídí je, aby se držely svých vytyčených tras, zajišťuje, aby jednotky nekolidovaly mezi sebou navzájem, s okrajem mapy, zdmi, ani s ostatními objekty herního světa. Na exekuci lze nahlížet jako na lokální plánovač pohybů jednotek s malým časovým výhledem. Samotný pohyb

Obrázek 3.7: Age of Empires 2, skupina tvořená řadami.



Obrázek 3.8: Knights and Merchants. Červené jednotky pochodují seřazeny za sebou, protože nechtějí opustit vytyčenou trasu.

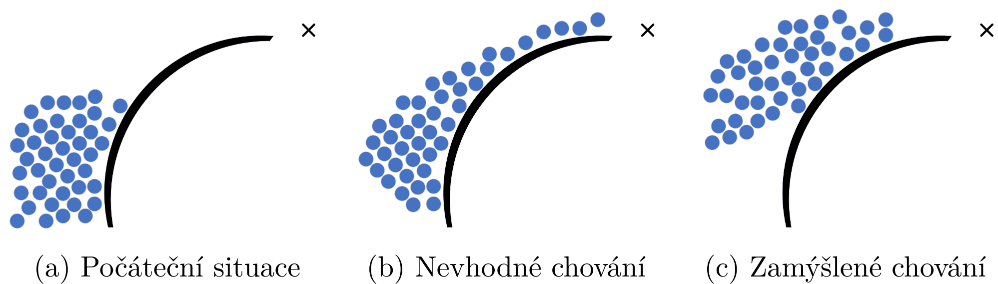
jednotek je pouhá iluze, během jednoho kroku hry se všem jednotkám spočítají nové pozice, na které se jednotky na hráčově obrazovce vykreslí. Vše probíhá diskrétně, mnohokrát za sekundu.

Při exekuci dochází k různým chybám. Některé jsou velmi patrné, a proto se s nimi počítá už při návrhu architektury her. Jednotky v sobě mají implementované kontrolní mechanismy proti uvíznutí, například restart navigace. Rovněž pokud již kolize nastane, tak ji taktně vyřeší. Jiné chyby jsou dány nastavením sil působících na jednotky (obrázky 3.8 – 3.11).

3.3.1 Flow field

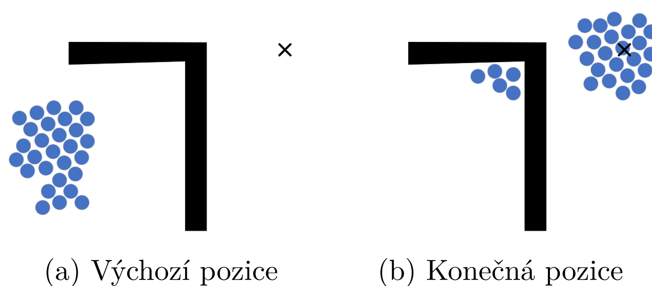
Flow field [21] je pojem zastřešující množství příbuzných metod, využívající předpočítané vektorové pole, kterého navigace využívá až při exekuci. Význam hodnot a způsob, jakým se využívají, se v různých metodách liší. S jedním takovýmto polem jsme se již setkali v sekci 3.2.1 u algoritmů vylepšujících A^*, kde je součástí algoritmu $JPS+$, ve kterém se toto předpočítané pole používá při vyhledávání nejkratších cest. V tomto případě mělo dokonce každé políčko uloženo osmirozměrný vektor.

Použijeme-li upravený Dijkstrův algoritmus, který spustíme z cílového políčka, jehož hlavní úprava spočívá právě ve vyplňování vektorového pole na základě toho,



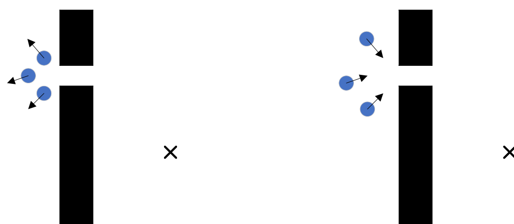
(a) Počáteční situace (b) Nevhodné chování (c) Zamýšlené chování

Obrázek 3.9: Chování jednotek, které téměř vůbec nechtějí opustit svou vytyčenou trasu. Modře jsou zobrazeny jednotky, černě překážky, křížek označuje pozici cíle.



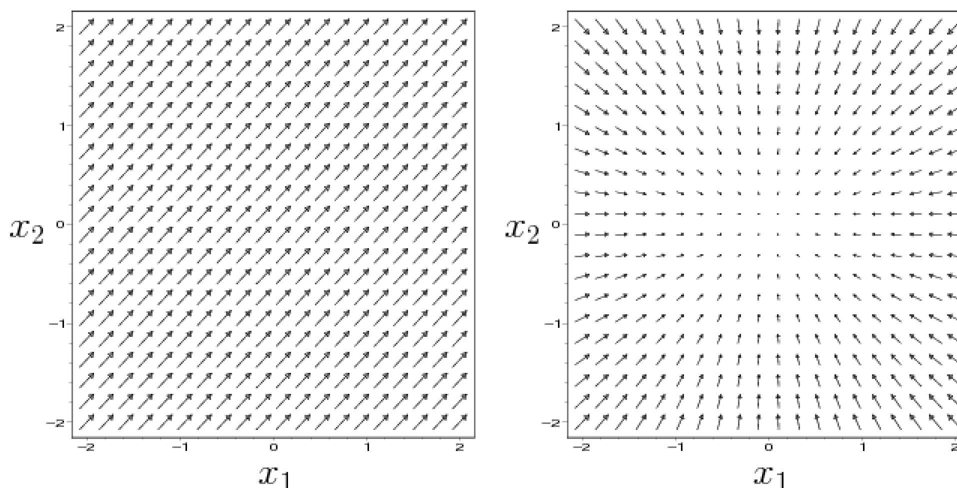
(a) Výchozí pozice (b) Konečná pozice

Obrázek 3.10: Uvíznutí jednotek, které se nechaly vytlačit ze své vytyčené trasy příliš daleko.



Obrázek 3.11: Špatné chování jednotek, které si vzájemně dávají přednost. V cyklech se od sebe vzdálí a opět se k sobě přiblíží.

odkud se na nové políčko přišlo, dostaneme speciální flow field. Tento flow field bude mít tu vlastnost, že všechny vektory, na které se lze dívat jako na šipky, budou ukazovat na sousední políčko, které je ze sousedních nejblíže k cíli. Díky tranzitivitě budou tyto šipky ukazovat směrem k cíli. Druhou úpravou algoritmu, kterou přidáme, je chování u políček mapy, na než jednotky mají zakázaný vstup. Šipky v jejich blízkosti pootočíme směrem od nich. A na těchto políčcích šipky nastavíme k nejbližším povoleným políčkům. Jednotka, která by svůj pohyb řídila tímto polem, by pak měla bez ohledu na startovní pozici dojít do cílového políčka.



Obrázek 3.12: Příklad vektorových polí. Vlevo: konstantní vektorové pole. Vpravo: do středu přitahující kulové pole. Obrázek převzat z [22]

3.3.2 Řízení jednotek

Metody „Steering Behaviors“[23] slouží k řízení pohybu jednotek během exekuce. Jedná se o reaktivní řízení na základě senzorických vstupů z prostředí. Ty jsou lokálního charakteru. Jednotky při exekuci mnohdy nemají žádné informace o tom, jak vypadá okolí, od něhož jsou vzdáleny několik svých délek.

Hry pracují s fyzikálním modelem jednotky. Jednotky mají velikost, pozici, hybnost (rychlost a směr pohybu), někdy dokonce hmotnost, a tím získanou setrvačnost, třecí koeficienty či poloměr zatáčení. Steering určuje sílu, jakou se má na takovýto fyzikální objekt působit.

Cílem použití těchto příbuzných metod je řídit jednotku jen pomocí jejího malého okolí nezávisle na ostatních jednotkách. Všechny následující metody počítají sílu a směr, jakým se má na jednotku působit. Většina z nich operuje s chtěnou hybností a k té se snaží aktuální hybnost jednotky upravit tak, aby jí dosáhly. Od aktuální hybnosti odečtou požadovanou hybnost. Zmíníme-li v následujících odstavcích požadovanou hybnost, tuto operaci tím implicitně míníme také.

Seek cílem tohoto chování je pohyb směrem k zadané souřadnici, chtěná hybnost vede k ní. Generuje maximální velikost síly.

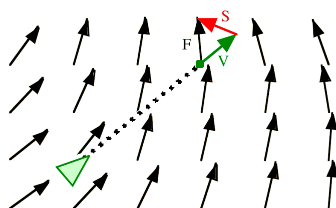
Arrival se snaží na zadané souřadnici zastavit, pokud je dostatečně daleko, chová se jako Seek, ale čím blíže je k cíli, tím větší silou působí proti směru pohybu až do úplného zastavení.

Separation chování má za úkol držet jednotky od sebe, aby nekolidovaly, generuje od každé jiné jednotky v okolí odpudivou sílu nepřímo úměrnou její vzdálenosti.

Alignment snaží se držet stejný směr a rychlost s okolními jednotkami na základě rozdílu směru a rychlosti pro jednotky v okolí spočítá průměrnou hybnost, které se snaží dosáhnout.

Coheason / Flock centering se snaží držet se davu, spočítá ve svém okolí těžiště okolních jednotek a usiluje ho dosáhnout.

Flow following chování následuje předpočítaný tok (flow field), jenž je uložen ve vektorovém poli. Na každé souřadnici je vektor, který se pro jednotku stojící na dané souřadnici stane její chtěnou hybností.



Obrázek 3.13: Vygenerovaný steering `S` k dosažení pozice, `F` vektor chtěné hybnosti, `V` současná hybnost.

Path following chování následování předpočítané cesty zajišťující, že jednotka nevyjede z pomyslných hranic mezi sousedními dílčími cíli.

Obstacle avoidance má za úkol zabránit kolizi se stěnou. Jednotka vysílá paprsky (raycasting), které když zasáhnou stěnu, vyvolají nepřímo úměrnou sílu opačným směrem podle vzdálenosti, ve které zasáhnou cíl.

Collision avoidance snaží se zabránit kolizi s ostatními jednotkami. Pro jednotky ve svém okolí spočítá jejich i svou budoucí pozici, a pokud by hrozila kolize, vygeneruje úhybnou sílu. Metody RVO a z ní vycházející ORCA, HRVO (Reciprocal Velocity Obstacle, Optimal Reciprocal Collision Avoidance a The Hybrid RVO) umožňují bez další komunikace (například o zvolené změně směru) jednotkám provádět úspěšné úhybné manévry [24], [25], [26].

Složená chování Pomocí těchto jednoduchých způsobů chování lze dohromady skládat komplikovanější modely chování.

3.3.3 Způsoby řízení skupiny

Skupinu lze řídit globálně, ale není to nutné. Postačí řídit jednotlivé jednotky a zajistit, že se jejich rychlost ani směr nebude příliš lišit od blízko se vyskytujících jednotek, že s nimi nebude kolidovat. Tak dosáhneme podobného chování, jaké je vidět u hejna ryb či ptáků – flocking. Flocking je příkladem složeného chování, jedná se o kombinaci collision avoidance, alignment a coheason [27].

Leader following

Skupina si zvolí jednotku, která bude jejím vedoucím. Konkrétní implementace výběru nebo přiřazení vedoucího skupině se může lišit, ale pro fungování metody to není příliš podstatné. Vedoucí skupiny jako jediný obdrží plán cesty, zbytek jednotek pouze ví, kdo je vedoucím, a snaží se držet se k němu v limitu minimální i maximální vzdálenosti. Pokud by nebyl limit na minimální vzdálenost stanoven, vedoucí by mohl uvíznout v davu svých následovatelů. S nastaveným limitem odpuzuje nejbližší jednotky ze své cesty a ty mu tak vytvářejí volnou cestu.

Flocking lze použít rovněž tehdy, když se jednotky snaží nebýt příliš blízko k jakékoli jiné jednotce pomocí separation. Pomocí alignment se snaží směřovat stejným směrem a pomocí flock centering se posouvat do volného prostoru uvnitř skupiny.

Formace

Obdobnou metodou pohybu jako je leader following je formace. Podobají se v tom, že se každá jednotka snaží zaujmout pozici ke zbytku skupiny. Skupině se spočítá centrum a orientace. A protože zde všechny jednotky mají pevně určenou relativní pozici vzhledem ke skupině, tak ve spojení s informacemi o centru skupiny a jejím směru mají i určenou absolutní pozici. [28]

Metoda se chová při změně směru hráčem vizuálně zajímavě. Do té doby ve formaci uzamčené jednotky se rozpustí a v případě potřeby se i zmenší kolizní rádius, aby se co nejrychleji dostaly do nově vypočtené formace.

3.3.4 Hledání nejbližších jednotek

Pro vyhledávání kolizí jednotek je vhodné použít nějakou datovou strukturu, která bude o jednotlivých jednotkách v každém kroku vědět, kde se nachází. Nejznámějším zástupcem takovýchto struktur je Quad tree [29].

Rozmysleme si, jak se budou jednotky v mapě pohybovat. Na počátku pohybu jsou všechny na jednom místě, poté se vydávají na cestu. Všechny jednotky mohou sdílet společný plán, ale může se stát, že každé jednotce bude přidělen vlastní jedinečný plán. Pravděpodobně se však skupina bude pohybovat pospolu.

I pro naší metodu, kde se jednotky náležející jednomu plánu budou pohybovat ve vzájemné blízkosti po celou dobu pohybu a s jednotkami vykonávající jiný plán do kontaktu budou přicházet do okamžiku rozdělení nebo krátce před dosažením cíle.

V obou případech budou jednotky na mapě rozloženy tak, že většina mapy bude prázdná a bude existovat pouze několik málo míst, kde se jednotky koncentrují. A nich se projeví výhoda testování pouze těch jednotek, které jsou blízko.

Naše struktura pro kolize bude vypadat následovně: mapa je 2D grid, my ho proložíme jiným s menším počtem polí a každé pole bude mít seznam jednotek, které se v něm nalézají. Tento seznam lze enumerovat, pokud se naimplementuje vhodně velkou hashovací tabulkou, tak přidání, přístup i smazání jednoho prvku zabere $\mathcal{O}(1)$. V každém kroku, kdy jednotka překročí práh, se odebere z jedné tabulky a přidá se do druhé. Tento přístup navíc začne získávat na smyslu, zamyslíme-li se nad operacemi, které budeme potřebovat, a sice výpočet separation force, kdy potřebujeme tyto blízké jednotky enumerovat.

Metody, které v našem simulátoru využíváme, jsou blíže popsány ve dvou následujících kapitolách.

4. Simulátor

Nedílnou součástí řešení je program, který nám umožní simulovat herní prostředí. V této kapitole se seznámíme s jeho vnitřním fungováním a metodami navigace, které implementuje.

4.1 Funkce simulátoru

Primárním účelem simulátoru je simulovat herní logiku navigace, sekundárním je umožnit ohodnocení provedené simulace.

Program nám umožňuje vybrat si herní mapu, její konkrétní dekompozici na disjunktní oblasti, pozici cíle a počet a pozice jednotek. Pro vytvoření testu jsou právě tyto parametry nezbytnou součástí konfigurace. Tuto konfiguraci lze uložit a opakovaně spouštět, případně i editovat a uložit jako novou konfiguraci testu (testy jsou v simulátoru immutable). Simulátor dále uchovává potřebné informace z průběhů testů pro všechny typy navigací k pozdějšímu zpracování.

4.2 Mapy

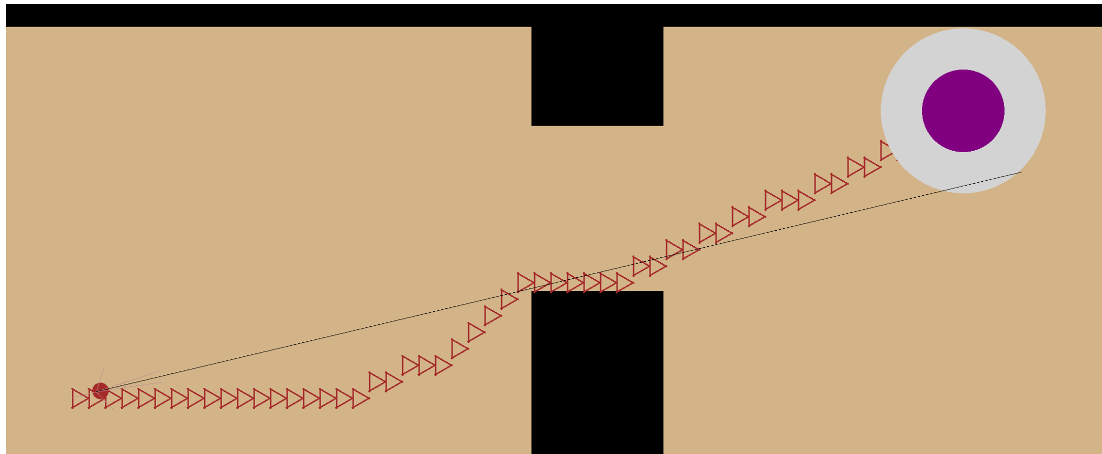
Náš simulátor potřebuje k zevrubnému otestování navigace různá herní prostředí. Abychom toho dosáhli umožňujeme načítání různých map. Tyto mapy si buď můžeme připravit sami, nebo můžeme použít již existující. Formát map jsme zvolili kompatibilní s formátem, který zvolil Nathan Sturtevant, pro svůj benchmark plánovacích algoritmů na gridových mapách [30]. Tím máme k dispozici velké množství map, ať již map náhodně generovaných, map bludišť či map, které byly použity ve skutečných hrách.

Pokud bychom chtěli do simulátoru přidat vlastní mapu, je možné si ji buď napsat, protože mapa samotná je textový soubor, jehož detailní popis formátu souboru mapy nalezneme v benchmarku v sekci „Map File Format Description“, nebo použít skript `covnert_image_to_map.sh`[1], kterým lze převést libovolný černobílý obrázek podporovaného formátu na mapu. Vytvářet si vlastní mapy či znát formát vstupního souboru mapy není pro obsluhu programu potřeba.

Přestože tento formát obsahuje více typů polí, než jsme v úvodním rozboru požadovali v 1.1.1, upravujeme ji, aby obsahovala pouze dva typy polí. Přes pole typu brod a volné pole tak mají jednotky povoleno chodit, přes ostatní mají průchod zakázán. Políčka se zakázaným průchodem budeme označovat jako zdi a doplňková políčka jako volná políčka.

Velikost pro většinu map z benchmarku nepřekračuje rozměry 512 × 512 políček, nejmenší mapy mají 8 × 8 a největší mapa, o níž víme, dosahuje 1260 × 1104. S přihlédnutím k rozměrům z benchmarku jsme z implementačních důvodů omezili velikost největší povolené mapy na 4096 × 4096. Implementační důvody tohoto rozhodnutí jsou rozebrány v programátorské dokumentaci (7.2).

[1] Skript používá program IrfanView pro převod do přenositelného textového souboru `.pbm`, který převádíme na text. Návod pro nastavení IrfanView, aby ukládal dočasný soubor do ASCII místo do binárního formátu, je uvnitř skriptu, který je součástí přílohy.



Obrázek 4.1: Jednotka následující cestu vytvořenou pomocí A* algoritmu. Trojúhelníky reprezentují plán jednotky, černou úsečkou je zvýrazněn směr pohybu jednotky. Tato úsečka prochází středem jednoho trojúhelníku, jedná se o nejvzdálenější bod plánu, jehož všechny předcházející body jednotka již viděla a je jejím cílem, dokud se neobjeví vzdálenější bod s přímou viditelností.

4.3 Navigace

V našem simulátoru jsme implementovali dva způsoby navigace, které se liší jak plánováním, tak exekucí. Nejprve představíme metodu navigace pomocí vektorového pole, poté představíme metodou, ve které se jednotky pohybují podle předpřipraveného plánu. V následující kapitole pro ni dodáme i další dvě různé implementace plánovací části založené na tocích v síti, ale exekuční část ponecháme shodnou.

4.3.1 Plánování

Pro navigaci, která následuje plán, máme unifikované rozhraní vstupů i výstupů z plánovacího algoritmu. Vstupem navigace je mapa s pozicemi jednotek a cíle. Výstupem je cesta pro každou jednotku.

Vektorové pole

Naší první baseline metodou navigace je metoda využívající vektorové pole. Fungování je detailněji popsáno v 3.3.1, zde jen uvedeme, že se v gridu z pole na jehož souřadnicích má cíl střed pomocí vyhledávání do šířky vyplní všechna dosažitelná volná políčka vektory směrujícími k cíli. Zdem se vyplní vektor ve směru nejbližšího volné pole. Vektorové pole pro tuto metodu je sdíleno všemi jednotkami skupiny, a proto se vyhledává pouze jednou a jednotky je referencují. V exekuční fázi se používá flow following steering.

Následování plánu – Nekoordinovaný A^*

Stejně jako u vektorového pole i v druhé baseline metodě navigace jednají jednotky na sobě zcela nezávisle jak v plánovací, tak i v exekuční fázi.

Každá jednotka po obdržení cílové souřadnice skupiny naplánuje svou vlastní cestu pomocí $A*$ algoritmu. Výsledkem je posloupnost políček, kterými má projít, aby dorazila k cíli.

Metrika V A* algoritmu se na výpočet vzdáleností v 2D gridu se nabízelo použít manhattanskou metriku vzdáleností, ale pro naše řešení jsme zvolili metriku vedoucí na cesty obsahující nižší počet bodů, kterými se má projít (manhattanská metrika vyrábí schodovou cestu, námi vybraná metrika vede k použití cest s diagonálními částmi). Vybrali jsme přesnější Eukleidovské přiblížení pro 8 nejbližších sousedů políčka 2D gridu, kde sousední políčka mají hodnotu jedna, kromě políček na úhlopříčce, které mají hodnotu $\sqrt{2}$.

Komunikace plánování a exekuce

Tyto části navigace jsou nezávislé a komunikují spolu pouze předáním cesty, případně předáním vektorového pole, a v opačném směru, když dojde k uvíznutí jednotky, exekuční engine může požádat pro konkrétní jednotku o nový plán.

4.3.2 Exekuce

Jednotky pro oba typy navigace se liší pouze v tom, jak získávají požadovaný vektor hybnosti. Ostatní vlastnosti jednotek jsou shodné. Jednotky jsou implementované pomocí steeringů. Z nichž jsou naiplementované Seek, Separation, Alignment, Obstacle Avoidance, Collision avoidance, Flow following a Path following. Jednotky navigace se liší v tom, že jedna využívá Flow following a druhá Path following. Flow following je implementováno tím způsobem, jak jsme o něm pojednali v analýze.

Path following

Path following je ovšem implementováno jinak, než jak je Raynolds popisuje. Cesta v jeho pojetí je de facto n-úhelník, uvnitř nějž se jednotka může volně pohybovat. V našem pojetí je cesta posloupnost bodů. Samotné fungování se tedy musí lišit.

Aktuální pozice v plánu Jednotka si pamatuje seznam bodů, kterými má projít. Projití bodem však neznamená, že se musí dostat do dostatečně blízkosti k bodu. Stačí, že se její vzdálenost k tomuto bodu bude snižovat a posléze narůstat, a že tento bod ve svém blízkém okolí bodů, kterými má jednotka za úkol projít, bude jeden z nejbližších. Pokud toto nikdy nenastane, zkoumá jednotka, zda se již nedostala za tento bod, a nemá tak budoucí bod k sobě blíže než aktuální. V takovém případně ho v plánu přeskočí. Tímto způsobem zajistíme, že jednotka bude mít informaci o tom, jaký je poslední bod plánu, ke kterému se přiblížila.

Jednotka si pamatuje i poslední nejvzdálenější bod plánu, ke kterému viděla všechny předchozí body a nejvzdálenější bod, který kdy viděla (opět s tím, že viděla i předchozí body). Bod, ke kterému jednotka směřuje, je vždy ten, který vidí a který je od ní v plánu nejvzdálenější (viz obr. 4.1).

Směřování k dílčím cílům plánu Podobně jako v se raycasing využil v obstacle avoidance, využívá se ho i pro zjištění viditelnosti nejvzdálenějšího bodu. Díky tomu je jednotce umožněno, aby nešla za dalším bodem plánu, ale za vzdálenějším. Tím se jednotce otvírá možnost pohybu k cíli pod libovolným úhlem. Zatímco vyhledávání vzdáleností v gridu počítá s horizontálním, vertikálním a diagonálním pohybem, je pohyb jednotky volnější. A tím, že body vycházejí z vyhledávání nejkratší cesty v 2D gridu, může být celková uražená vzdálenost kratší než minimum nalezené A* algoritmem.

Zotavení při selhání exekuce Pokud se jednotka dostane do situace, že nevidí žádný z bodů plánu, které jsou blízké poslednímu přiblíženému bodu nebo nejvzdálenějšímu zapamatovanému, pokouší se jednotka ještě několik kol nalézt svou pozici v plánu a dosáhnout předchozího cíle. Pokud se jí to nedaří příliš dlouho, je nutné ji přeplánovat. Toto se netýká metody s vektorovým polem, protože ta má pro každý bod mapy definovaný směr pohybu.

Náhodný pohyb zabraňuje uvíznutí Proto má jednotka používající Path following navíc jednu techniku řešící zároveň tento problém a zároveň uvíznutí. Všechny jednotky implementují obstacle avoidance, ale kvůli plánování nejkratších cest, které často vedou v těsné blízkosti zdí, hrozilo uvíznutí spíše v jejich blízkosti. A právě u nich jednotce většinu zorného pole vyplňuje zeď, a tak se za ní mohou lehce skrýt její dílčí cíle. Díky tomu, že jednotka kolem sebe vysílá paprsky, které jí slouží jako tykadla, která generují opačnou, nepřímo úměrnou sílu ke kontaktnímu místu, bylo možné přidat paprsek, který náhodně mění svoji délku a vnáší do celého systému stochastičnost.

Flow following

Metoda pohybu založená na vektorovém poli tuto nahodilost neobsahuje, a tak jediný rozdíl při opakovaní experimentu, který se může projevit je, když se jednotky nachází na stejném bodě, poté je generována náhodná odpudivá síla. To však nastává velmi výjimečně (především pouze krok po inicializaci). Z čehož plyne, že simulace pro jednotku pracující s vektorovým polem vydává téměř identický výstupy.

Pohyb jednotek

Jednotky se snaží nekolidovat se stěnou ani s ostatními jednotkami a pomocí separation force udržují k ostatním jednotkám vetší vzdálenost. Pokud jednotky dojdou do úzkého hrdla, steeringové síly je téměř zastaví, ale kdykoli se naskytne příležitost v podobě volného prostoru, do kterého by se mohli posunout, využívají toho a díky recipročnímu vyhýbání do sebe příliš nevrážejí. Pokud se jednotka dostane do zdi, má povoleno se v ní z části pohybovat, pokud směr, kterým se chce ubírat, má v určité vzdálenosti volné pole (opět měřeno raycastem). Jednotky mají tímto způsobem možnost zkrátit si cestu skrz roh políčka.

Zánik jednotky

Jednotka, která vyžaduje přeplánování a pro jejíž aktuální pozici neexistuje cesta k cíli, což nastane pouze v případě, že jsou s cílem v různých komponentách souvislosti, je takováto jednotka odstraněna. Pro účely vyhodnocení má každá jednotka několik čítačů, více se o nich zmíníme až v kapitole o evaluaci, v sekci 6.6. Čítač udržující počet jednotek je o tom informován a je přehrán zvuk ztráty jednotky. Podobně se zachováváme i v situaci, kdy jednotka dosáhne globálního cíle. V takovém případě se však čítače jednotky nezruší, ale uloží. Uvíznutí jednotek se řeší podobným mechanismem, pokud všechny ostatní prostředky nápravy selžou, jsou takové jednotky odebrány.

5. Navrhovaná metoda

Námi navrhovaná metoda existuje v kontextu simulátoru, díky tomu můžeme využít stejnou implementaci exekuční části navigace, kterou obsahuje i nekoordinovaný A*, a tím dosáhneme výsledků, které budou více reflektovat úspěšnost naší metody.

Ve skutečnosti jsme implementovali dvě metody, které se od sebe liší pouze grafem, nad kterým pracují. Jak se od sebe tyto grafy liší je popsáno v části 5.3 vytvoření grafu.

5.1 Představení hlavních částí algoritmu

- Převod mapy na síť
  - Dekompozice mapy
  - Vytvoření grafu
  - Převod na síť
- Vyhledání souběžných množin cest v síti
  - Opakované hledání nejkratší cesty
  - Oprava záporných hran křížením cest
- Přiřazení jednotek jednotlivým cestám
  - Výběr nejrychlejší množiny souběžných množin cest
  - Naplánování počtu jednotek k jednotlivým cestám
  - Přiřazení cesty jednotlivým jednotkám

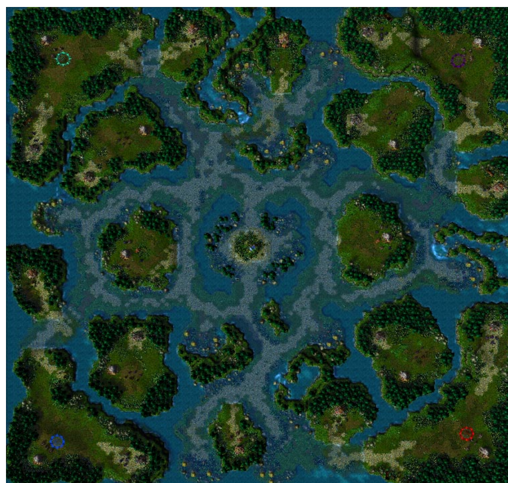
5.2 Dekompozice mapy

Pro převod mapy získané z textového souboru na síť v prvním kroku používáme automatickou dekompozici mapy na jednotlivé zóny oddělené branami. Dekompozice od Kári Halldórssona [31] funguje následujícím způsobem. Nejprve se pro všechna políčka spočítá vzdálenost od nejbližší zdi. Tato hodnota se interpretuje jako hloubka terénu. Tímto se vytvoří reliéfová mapa, která se postupně od nejhlubších údolí zaplavuje vodou. Hranice, kde se setkají jednotlivá jezera, se označí jako místo spoje. Tato místa dále nazýváme branami. Dekompozice pokračuje tím, že se všechna volná políčka zaplavená z jednoho pramene označí stejným číslem.

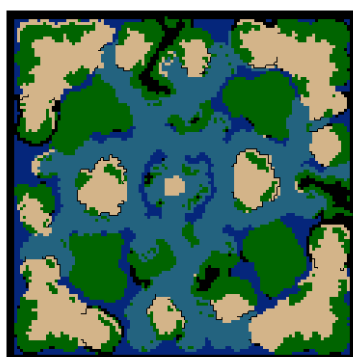
Chtěli bychom, aby brána ležela vždy mezi políčky s různými čísly oblastí, avšak za určitých okolností brány vznikají i uvnitř jedné oblasti, proto takovéto brány odstraňujeme[1]. Odstraňujeme i příliš malé brány a další chyby.

Tento způsob dekompozice není jedinou možností, jak dekompozici provést. Alternativně lze, stejně jako to dělají herní společnosti pro Navmesh, nechat designéry mapy, aby ke každé mapě připravili navigační data. Nehledě na zdroj dekompozice, je nutné ji převést na graf. Tento graf může být opět součástí připravených navigačních dat. Naše řešení ale umožňuje přepočítání dekompozice během hry, čímž můžeme do navigace zpracovat i hráčem vytvořené budovy či jiné dynamické objekty.

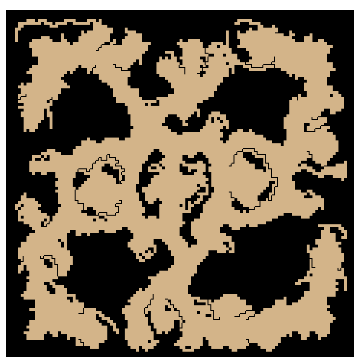
[1] pouze pro jeden z grafů



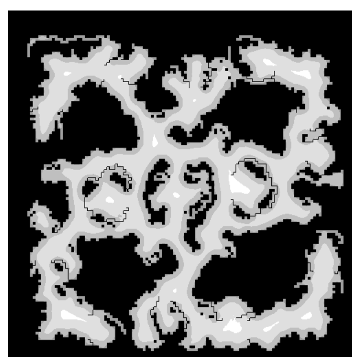
(a) Mapa Mystic Isles ze hry Warcraft III



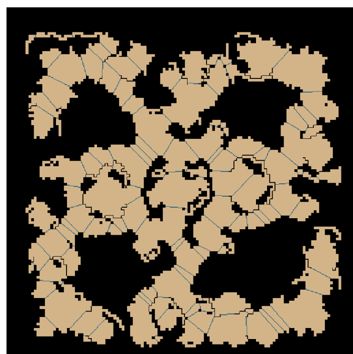
(b) Vstupní soubor



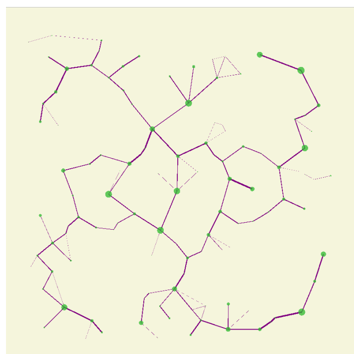
(c) Binární mapa



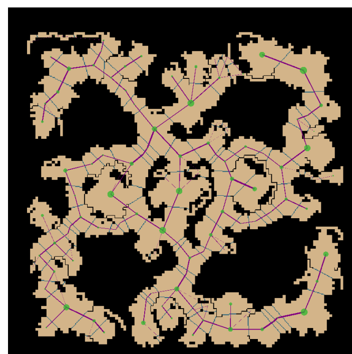
(d) Analýza zaplavením



(e) Nalezené brány



(f) Graf sítě



(g) Graf sítě v mapě

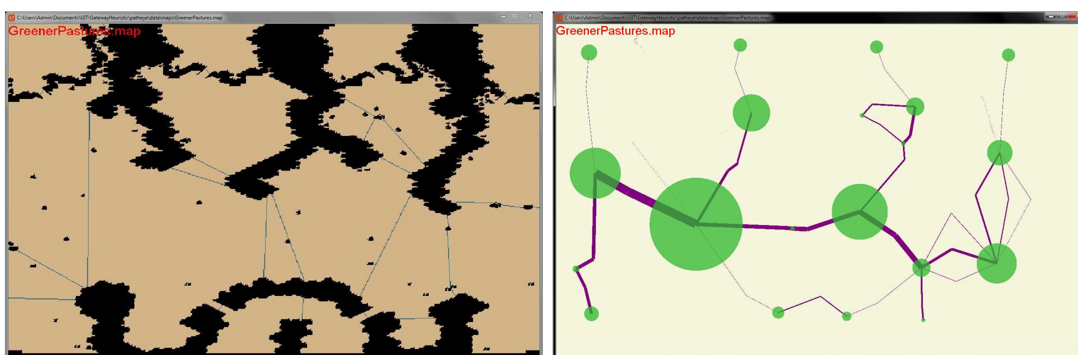
Obrázek 5.1: **Zpracování vstupního souboru** – `mysticisles.map` z benchmarku pro hledání cest v gridových mapách [30]. V simulátoru zobrazený vstupní soubor, ve kterém byla pro každý typ políčka použita jiná barva `(b)`. Počet typů políček se zredukuje pouze na dva typy na základě jejich průchodnosti: na volná pole, nebo zdi `(c)`. V takto upravené mapě se provádí analýza, při níž se zaplavuje mapa vodou `(d)`. Podle rozdílu výšek vodních hladin se určují pozice bran `(e)`. Brány dělí původní mapu na oblasti, které jsou mezi sebou propojené právě těmito branami. Takto rozdělenou mapu lze převést na grafovou reprezentaci `(f)`. Grafová reprezentace promítnutá zpět na mapu `(g)`.

5.3 Vytvoření grafů

Pro naši metodu vytváříme dva druhy grafů s odlišnými vlastnostmi. Budeme je označovat jako Středový a Bezstředový graf (pro přehlednost psáno s velkými počátečními písmeny). Vstupem pro vytvoření grafu je mapa reprezentovaná 2D gridem, kde jednotlivá pole mají hodnotu číslo oblasti, do níž náleží, druhým vstupem jsou koncové souřadnice bran oddělující oblasti.

5.3.1 Středový graf

Začneme tím, že oblastem spočítáme těžiště, do něj umístíme vrchol reprezentující střed oblasti. Dále pro každou bránu vytvoříme vrchol v jejím středu a propojíme je s vrcholy ve středech oblastí, jimž brána tvoří hranici. Dostáváme první reprezentaci mapy pomocí grafu, kde vrcholy jsou středy oblastí a brány a hrany jsou spojnice mezi středem a bránami v oblasti. Tento graf v simulátoru používá jednotka „FlowGraph1“. Šířka hran mezi středem a libovolným vrcholem uvnitř brány se rovná šířce této brány. Díky tomu, že brány obsahují vrchol, lze obě oblasti spojit více hranami a stále se tak nejedná o multigraf.



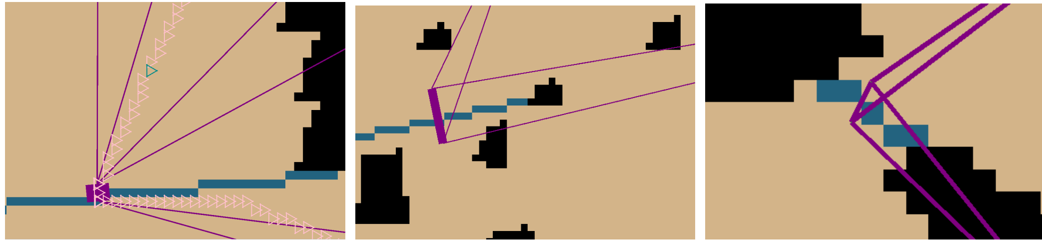
Obrázek 5.2: Screenshoty simulátoru. Vlevo: Mapa se zobrazenými branami. Vpravo: Středový graf vzniklý převodem z této mapy.

5.3.2 Bezstředový graf

Bezstředový graf vytvoříme tak, že na každou stranu brány umístíme vrchol, a tyto vrcholy spolu spojíme. Po vytvoření všech vrcholů bran spojíme všechny vrcholy nacházející se v jedné oblasti, každý s každým (viz obr. 5.3). Tento graf v simulátoru používá jednotka „FlowGraph2“. Šířka hran procházející branou je určena šířkou brány a šířka hran v rámci oblasti je určena jako minimum z šířek vrcholů, jež spojuje. Šířka vrcholu se rovná šířce hrany jdoucí mimo oblast, tedy šířce hrany jdoucí skrze bránu.

5.3.3 Délka hran

Výpočet šířky bran je jednoduchý, mezi oběma konci brány se nenachází zeď a pohybujeme se ve dvourozměrném eukleidovském prostoru, takže šířka je vzdáleností těchto bodů.



Obrázek 5.3: Detail spojení bran Bezstředového grafu. Hrany grafu vedou skrz středy bran, vlevo je vidět důsledek – neoptimální cesta pro jednotku. Všechny brány v rámci jedné oblasti jsou mezi sebou plně spojeny.

Délku hran lze počítat různými způsoby. Nejdříve je potřeba rozhodnout, zda budeme měřit vzdálenost mezi mezi centry bran, krajními body, nebo mezi všemi body. A pokud mezi všemi body, zda budeme chtít minimální, maximální nebo nějak průměrnou vzdálenost. Pokud bychom znali přesnou trasu, mezi kterými body se jednotky budou pohybovat, tak je zřejmé, že nás zajímala minimální vzdálenost mezi body, které jednotka při cestě použije. Poté je potřeba rozhodnout, jak budeme tuto vzdálenost měřit, problémem totiž jsou nekonvexní oblasti.

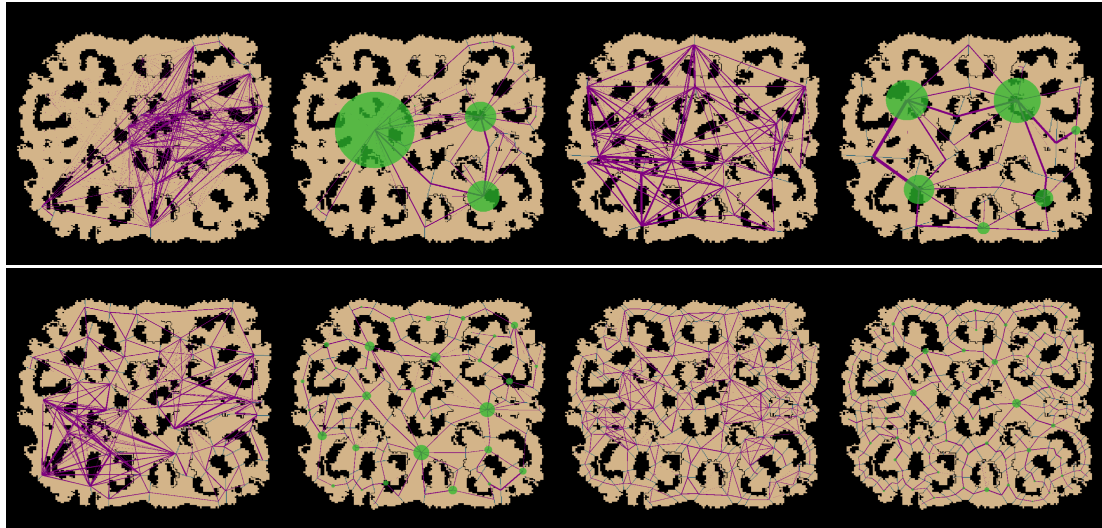
Tato rozhodnutí je potřeba dělat po analýze výstupů dekompozice, která často produkuje velmi široké brány (viz obr. 5.6c a 5.3). Pokud bychom v tomto případě zvolili středy bran, výsledná vzdálenost cesty mezi branami by byla delší, než by byla reálná vzdálenost uražená jednotkou.

Dekompozice však rovněž produkuje nadbytečné brány a brány, které svírají úhel, takže alternující konce bran jsou k sobě blíže než jejich opačné konce viz 5.6e, kde je na obrázku vidět bíle zobrazená cesta jednotky, které vzdálenost k boční bráně a zpět vyšla kratší než přímá cesta, tedy pro tento výpočet neplatí trojúhelníková nerovnost. Zde by naopak výpočet vzdálenosti mezi středy potlačil tento zjevný problém.

Naimplementovali jsme obě možnosti výběru bodů pro vzdálenost, a přesto se na základě našich testů se zdá, že na většině map větší chybu působí středy bran oproti nejkratším cestám. Možná je to způsobeno pravděpodobností vzniku těchto jevů, čím delší cesta, tím větší pravděpodobnost jejího zkrácení, ale na dlouhé cestě také pravděpodobně bude více chyb exekuce, takže se do určité míry tyto chyby navzájem vyruší.

Měření vzdálenosti

Na základě testů jsme tedy zvolili hledání nejkratší cesty mezi všemi body bran, ale stále zbývá rozhodnout, jak tuto vzdálenost měřit. Zda stejně jako šířku hrany, což by silně podměřovalo, pokud by neexistovala přímá viditelnost mezi branami. Nebo zda ji určit jako počet políček, který odpovídá nejkratší cestě nalezené A* algoritmem, případně vzdáleností této cesty podle jeho metriky (viz 4.3.1). Takováto cesta však naopak bude příliš dlouhá (viz obr. 4.1). Řešení, ke kterému jsme se přiklonili, kombinuje předešlé přístupy. Na cestě nalezené A* algoritmem postupně hledáme přímo viditelné úseky, mezi nimiž počítáme vzdálenost stejně jako u šířky a tyto vzdálenosti sčítáme (na výše uvedeném obrázku by se jednalo o součet vzdálenosti od startu k vyznačenému trojúhelníku a od něj do cíle). Takto spočítaná vzdálenost se velmi blíží skutečnosti, díky



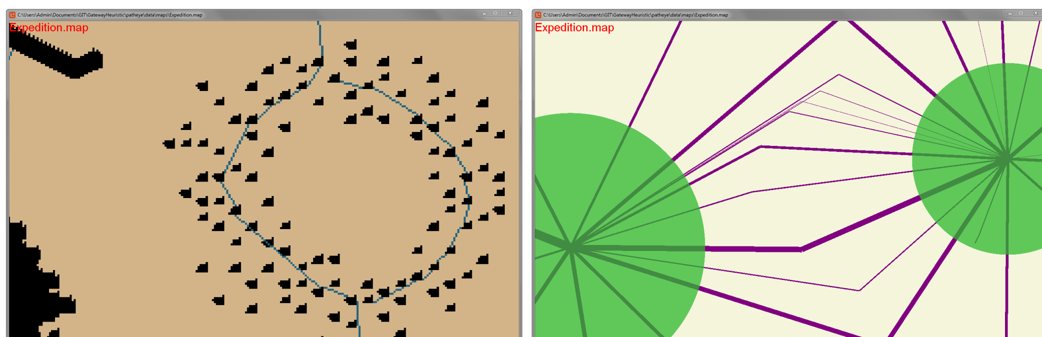
Obrázek 5.4: Porovnání dvojic grafů založených na stejné dekompozici mapy ukázané na mapě Blastedlands. Vlevo je vždy Bezstředový graf, vpravo Středový.

způsobu pohybu jednotky.

5.3.4 Porovnání vlastností grafů

Středový graf je rovinný, je odolnější vůči špatné dekompozici, obsahuje méně hran, ale při dobré dekompozici dává horší výsledky než Bezstředový graf, a navíc nerespektuje vzdálenosti v oblasti, takže plány, které vytváří se někdy kříží. Jeho velkým problémem jsou oblasti, které mají střed uvnitř zdi. Toto se častěji stává pro dekompozice, které obsahují málo hran. Cesty této oblasti pak nemohou mít správně určenou vzdálenost, a tak hledání selže.

Bezstředovému grafu nevadí brány uvnitř jedné oblasti, ani pokud se zvolí výpočet vzdálenosti splňující trojúhelníkovou nerovnost či nikoli. V případě, že není splněna, používá tok tyto brány ke svojí navigaci.



Obrázek 5.5: Detail mapy s vyobrazenými branami. Samotná vrstva (multi)grafu sítě.

5.4 Převod na síť

Graf zorientujeme zduplikováním hran. Nastavíme kapacitu hrany jako její šířku a cenu jako její délku. Přidáním zdroje a stoku dokončíme převod na síť. Zde se metoda převodu mezi oběma typy grafu liší. Středovému grafu, v němž se jednotky i cíl nacházejí uvnitř oblasti reprezentované vrcholem, pouze tyto vrcholy označíme jako zdroj a stok. V Bezstředovém grafu spočítáme centrum skupiny, na dané místo přidáme nový vrchol, který označíme jako zdroj a spojíme ho se všemi branami v oblasti. Analogicky řešíme i stok.

5.5 Hledání cest

Nyní máme hotovou síť, v níž můžeme začít hledat tok. Problém maximálního toku s minimální cenou (viz 2.2.5) lze řešit pomocí zlepšujících cest „augmented paths“, my s naším problémem budeme postupovat obdobně. V grafu vyhledáme nejkratší cestu, tuto cestu saturujeme. Saturace cesty nejprve najde maximální tok, který lze po dané cestě poslat, a o jeho velikost snižuje kapacitu hran v této cestě a hranám jdoucím opačným směrem kapacitu naopak navyšuje. Nulová kapacita hrany je téměř ekvivalentní tomu, jako by byla z grafu odebrána (rozdíl se projeví u nového toku opačným směrem, neodebrané hraně lze kapacitu navýšit oproti odebrané, která se musí znovu přidat). Takto saturovanou cestu si uložíme i s její kapacitou a pokračujeme v hledání nejkratších cest, dokud existuje cesta ze zdroje do stoku.

Tímto způsobem získáme posloupnost zlepšujících se cest. Pokud bychom v podkladovém grafu sečetli hodnoty toku odpovídajících hran těchto cest, dostali bychom řešení maximálního toku s minimální cenou. Čili jsme vytvořili dekompozici tohoto toku na jednotlivé cesty. Tyto cesty však mohou tok dekomponovat pro naše účely nevhodným způsobem.

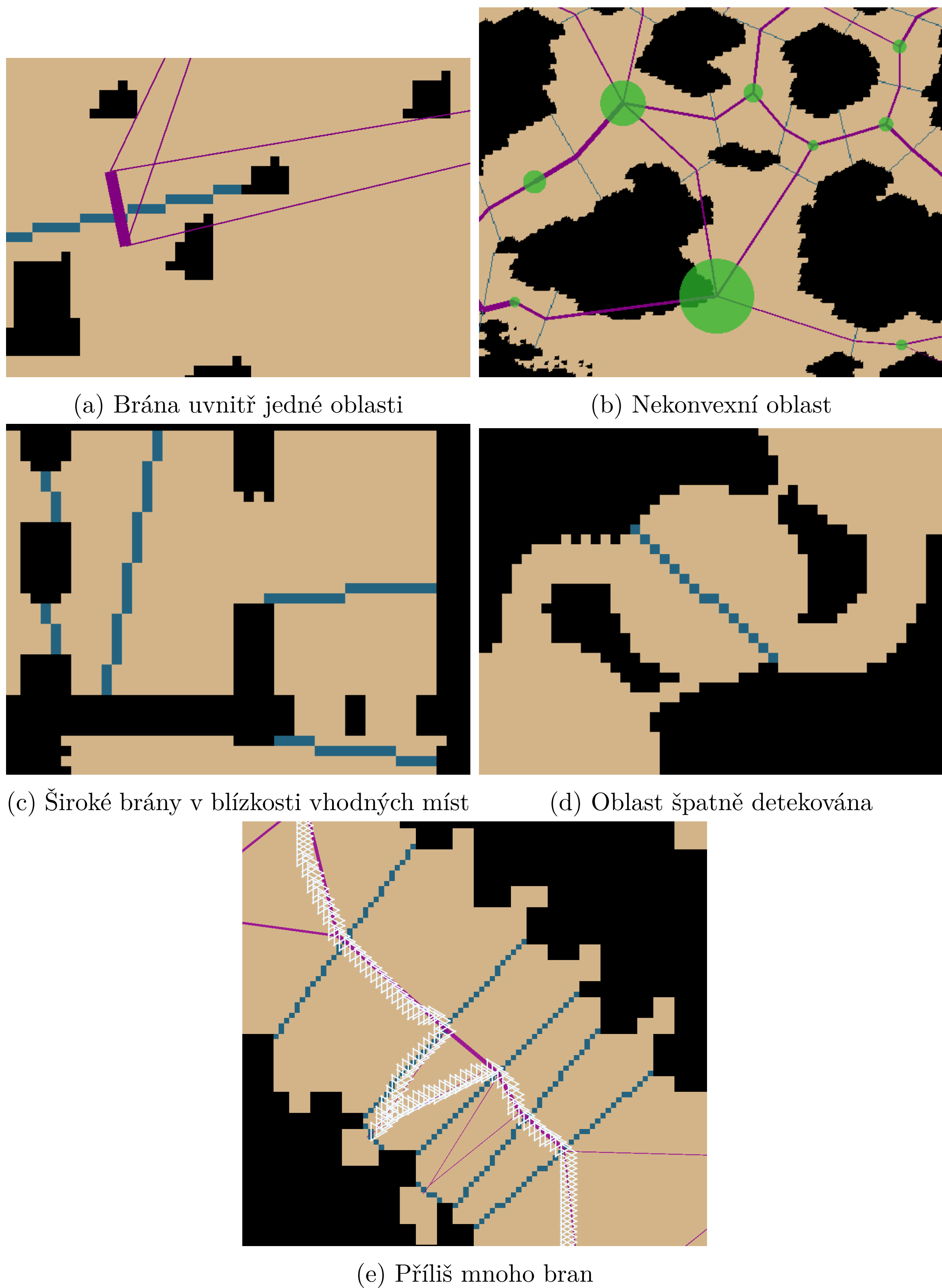
5.5.1 Problém dekompozice toku

Pro lepší ilustraci toho, proč je tato dekompozice nevhodná, k ní zkusíme dojít jiným způsobem. Mějme mincost solver, který budeme používat k hledání částečných řešení a tato průběžná řešení tohoto problému si budeme ukládat. Nechť solver umí nastavit velikost toku[2]. A nechť je náš problém řešitelný v celých číslech. Začneme s tokem velikosti jedna a dále postupně zvětšujeme tok vždy o jedna, dokud nenalezneme maximální tok. Po nalezení maximálního toku získáme porovnáním využití kapacit sousedních toků informaci, které hrany jednotkový tok využil. Odečítáme-li vždy starší tok od novějšího ($tok_{n+1} - tok_n$), může nastat situace, že některé hrany budou záporné.

Záporný čas

Je způsoben tím, že tok znovu využívá již využitou hranu proti směru dosavadního toku. Díky tomu, že posloupnost hran ve správném pořadí reprezentuje cestu

[2]Pro nastavení velikosti toku není nutné mít solver, který to implicitně umí, stačilo by namodelovat vhodným způsobem síť – nahradit zdroj novým, spojeným s původním zdrojem hranou, které by se měnila velikost kapacity.



(a) Brána uvnitř jedné oblasti

(b) Nekonvexní oblast

(c) Široké brány v blízkosti vhodných míst

(d) Oblast špatně detekována

(e) Příliš mnoho bran

Obrázek 5.6: Příklady selhání dekompozice mapy při vyhledání bran vedoucí k suboptimálním výsledkům.

ze zdroje do stoku a tím, že cena hran reprezentuje její délku, dostáváme zápornou délku hrany. Délka hrany navíc reprezentuje ve spojitosti s rychlostí jednotky čas, vyjadřuje, jak dlouho bude trvat cesta z jednoho vrcholu sítě do druhého. Dostáváme se tedy k zápornému času, vlastně jsme se dostali k cestování v čase (do minulosti). Z hlediska toku je toto v pořádku, protože hranou již cestuje protisměrný tok a cesta hranou oběma směry trvá stejně dlouho a tok přenáší homogenní médium. Odpovídá to tomu, pokud bychom při exekuci navzájem vyměnili jednotky, nacházející se na obou koncích hrany nebo jim pouze vyměnili plány. Touto výměnou by se identita jednotek toku změnila, ale situace by odpovídala řešení v tocích. My ovšem takovouto možnost nemůžeme připustit, protože toto řešit nenáleží exekuci. Navíc by pak ani nebylo možné použít unifikovanou exekuční část navigace.

Záporné hrany v cestě

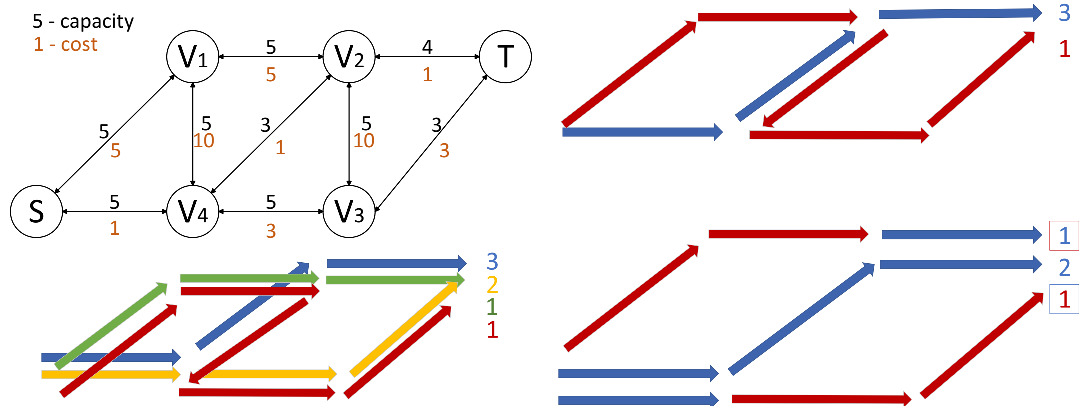
Těchto záporných hran se tedy budeme muset zbavovat. Křížením se záporná hrana anuluje. Výběrem množiny možných cest ke křížení a výběru nejlepší její podmnožiny provázejí následující sekce. Příklad dosavadního postupu naší metody a křížení je na obrázku 5.7.

Záporné hrany hledáme rovnou při hledání zlepšující cesty. Pokud narazíme na cestu obsahující záporné hrany, pak tuto cestu rozdělíme na alternující části podle toho, jakým směrem v daných částech teče tok. Více hran jdoucích za sebou může mít tok směřující zpět. Nejnižší možný počet alternujících segmentů jsou tři. Cesty, se kterými pracujeme, vznikly jako nejkratší cesty, proto nemohou obsahovat více průchodů libovolným vrcholem, a tedy ani zdrojem či stokem. Z toho plyne, že části cesty u obou těchto vrcholů musí mít dopředný tok. Při existenci alternujících části musí mít alespoň jednu souvislou část hran s tokem v opačném směru. Situace, že zlepšující cesta bude obsahovat více oddělených zpětných toků nastat může, tím bude i více alternujících částí, ale celkový počet alternujících částí nikdy nebude sudý.

Odstranění záporných hran

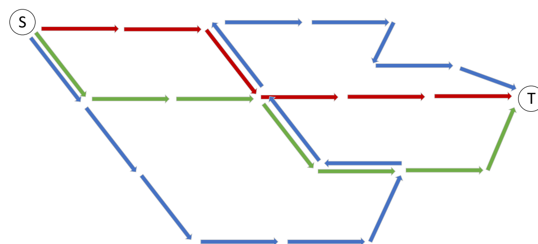
Došli jsme tedy k tomu, že počet alternujících částí bude vždy tři anebo více, a proto je budeme zpracovávat po těchto trojicích. Množině všech nalezených cest říkejme koexistující cesty. Tato množina má tu vlastnost, že všechny v ní obsažené cesty mohou být saturovány najednou bez toho, aby došlo k přesaturování libovolné hrany v podkladovém grafu, a to přesto, že mezi sebou mohou sdílet hrany. Saturací cesty zde myslíme poslat tok až o velikosti uložené kapacity, tedy ne saturaci v podkladovém grafu. Z množiny koexistujících cest vybereme všechny cesty, které obsahují záporný segment trojce uvnitř své cesty v dopředném směru.

Používáme heuristiku, která preferuje delší cesty, proto nejprve hledáme cesty obsahující oba krajní vrcholy záporného segmentu. Pokud součet toků cest obsahující tyto vrcholy bude větší nebo roven zápornému toku, můžeme přejít k tomu, že z těchto cest vybereme cesty, které budeme křížit. Pokud však žádná cesta neexistuje nebo pokud je součet toků těchto cest menší než tok, který chceme poslat skrz tuto hranu, je potřeba záporný segment rozdělit na více segmentů. Ilustrujme obrázkem 5.8, kdy tato situace může nastat. Je potřeba, aby zpětný tok bez použití dopředného toku opět pokračoval v protisměrném toku. Oba segmenty po



Obrázek 5.7: Vlevo nahoře: Síť s vyznačenou kapacitou a cenou všech hran. Vlevo dole: Vyobrazení postupu naší metody, která opakovaně hledá nejkratší cesty, ty saturuje a v případě, že nalezne záporné hrany, tak se jich zbaví křížením. Nejprve nalezne modrou, žlutou, zelenou a červenou cestu. Vpravo nahoře: Protože červená cesta obsahuje zápornou hranu, křížíme ji s cestou, která tuto hranu používá. Vpravo: Vznik nových cest díky křížení. Původní modrá cesta na rozdíl od červené nezanikla díky tomu, že měla dostatečnou kapacitu. Součet toku modro-červené, modré a červeno-modré cesty je stále 4, jako před křížením.

dělení je třeba doplnit prázdným dopředným segmentem tak, abychom zachovali invariant o lichosti počtu alternujících toků. Dělení záporného toku na části má smysl pouze pro vrcholy, pro které existuje i jiný než protisměrný tok. V našem případě se jedná o vrchol, do kterého směřují šipky všech barev. Vrchol na zelené cestě o jednu hranu níž nemá cenu zkoušet dělit. [3]



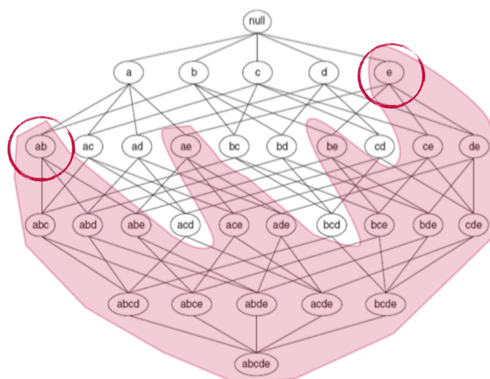
Obrázek 5.8: Příklad toků v síti, kde modrá cesta obsahuje tři protisměrné hrany, dvě na zelené cestě a jednu na červené.

Tato situace bude nastávat výjimečně. Opačná situace je však častá. A sice součet toků je větší roven toku, který chceme poslat. A množina cest, které se našly, obsahuje alespoň jeden prvek.

Výběr podmnožiny cest s dostatečným tokem V tomto odstavci se budeme věnovat výběru podmnožiny cest, které mají společnou trasu přes místo, kterým v opačném směru chce téci nová cesta. Vyřešme nejprve triviální případy, kdy množina je rovna jedné. Kde se celkový tok rovná požadovanému opačnému

[3]zde se implementace odlišuje a používá méně komplexní implementaci, která hledá pouze krajní body, tedy toto dorozdělení není součástí vzorové implementace, protože na našich testovacích mapách z benchmarku nebyl takto komplikovaný algoritmus potřeba.

toku. A nebo obecněji, kde součet toků cest bez libovolné nedosahuje požadované velikosti toku, ale s ní je celkový tok větší či roven požadovanému. V těchto případech je potřeba zvolit všechny cesty. V posledním případě může část toku po odečtení záporného toku zbýt. Pro rozdělení tohoto zbytkového toku existuje řada strategií. Naší zvolenou strategií je vybrat pouze jednu cestu, které zbude část celkového toku. Motivací je snížit prohledávaný prostor.



Obrázek 5.9: Pokud dvojce cest `a`,`b` a cesta `e` mají již dostatečnou kapacitu, zmenší se prohledávaný prostor o růžovou oblast.

Pro netriviální případy generujeme možné kombinace cest. Podmnožiny mají částečné uspořádání, to lze zachytit Hassovým diagramem. Generování podmnožin děláme chytře, abychom nemuseli enumerovat všechny podmnožiny množiny. K tomu použijeme poznatek z Apriori algoritmu pro Data mining metody. Ve svazu existuje vlastnost „The downward closedness“, která říká, že pokud již něco jednou splňuje podmínky, tak je nepřestane splňovat jejich dalším zvolňováním. Tímto způsobem vygenerujeme všechny podmnožiny, které mají v součtu tok větší roven toku, ale nemají podmnožinu se stejnou vlastností (viz obr. 5.9). Z těchto množin poté vybereme tu, která má nejmenší rozdíl skutečného a požadovaného toku, ideálně pokud by se rovnaly, dojde ke kompletnímu odstranění všech cest.

Křížení cest V grafu vždy existuje maximálně jedna záporná cesta, a to ta právě nalezená. Křížíme vždy dvojici cest, které mají společnou část cesty, ale v opačných směrech. Jak křížení cest vypadá představuje obrázek 5.7.

Samotné křížení pro obě nově vznikající cesty napojí jejich první části až po vstupní vrcholy do záporného segmentu a poté spojí se zbytkem cesty za výstupním vrcholem záporného segmentu. Tím vzniknou dvě nové cesty, které se přidají do koexistujících cest a původním cestám se upraví velikost toku, případně se odeberou. Pokud je záporných segmentů více, tak se nově zkřížená cesta, obsahující zbytek záporných segmentů, znovu ponechá křížit, dokud existují záporné segmenty. Křížení vždy sníží velikost toku záporného segmentu nebo ho odebere úplně. Z trojce alternujících segmentů se tak stane první segment anebo kompletní cesta. Tím se postupně odstraní všechny záporné segmenty všech záporných cest.

Nyní jsme se dostali do stavu, kdy máme dekompozici, která neobsahuje záporné hrany, a můžeme ji tak použít k dalšímu plánování.

5.6 Přiřazení jednotek nalezeným cestám

Vždy, když se našla záporná hrana, uložili jsme si před křížením množinu koexistujících cest. Po přidání poslední nejkratší cesty si uložíme i aktuální koexistující množinu. A optimalizaci provádíme na těchto koexistujících množinách cest. Každé koexistující množině spočítáme kolik kroků potrvá simulace při použití dané množiny, z nich vybereme nejkratší a její plán cest použijeme.

Cesta je posloupnost bran, kterými má jednotka na cestě k cíli projít. Brány mají souřadnice v gridu a my mezi nimi pomocí $A*$ algoritmu nalezneme nejkratší cesty. Tyto cesty v gridové mapě převedeme, aby více odpovídaly skutečným drahám jednotek během exekuce. Jejich délku zkombinujeme spolu se znalostí maximální možné rychlosti jednotek, a tím dostaneme dolní odhad potřebného času (t) k uražení cesty mezi zdrojem a stokem skrze brány dané cesty pro jednu jednotku. Předpokládáme, že brány budou umístěné do nejužších míst mapy. Proto by se minimální šířka území v dráze měla rovnat šířce nejužší bráně dané cesty. Na základě tohoto předpokladu a znalosti velikosti jednotek spočítáme propustnost (p) cesty, vyjadřující, kolik jednotek pojme cesta ve svém nejužším místě, potažmo nejvyšší možný počet jednotek, které najednou mohou dokončit tuto cestu.

Řešíme optimalizační úlohu, jak rozdělit jednotky cestám tak, aby se minimalizoval celkový čas průchodu. Tedy minimalizujeme maximum délky trvání průchodu všech přiřazených jednotek přes všechna možná rozdělení (r) počtu jednotek k jednotlivým cestám. Pro m cest a n jednotek optimalizujeme výraz[4]:

$$\min_{r \in R} \left(\max_{i \in \{1,\dots,m\}} \left(\frac{r_i}{p(i)} + \operatorname{sgn}(r_i) * t(i) \right) \right), \text{ kde}$$

$$r = \begin{pmatrix} r_1 \\ \vdots \\ r_m \end{pmatrix} \quad r_i \in \mathbb{N}_0, \quad R := \left\{ r \middle| \sum_{i=1}^{|r|} (r_i) = n \right\}$$

Pro nás zajímavým výstupem této optimalizace není jen ono minimum, ale i rozdělení `r`, určující počet jednotek přiřazený jednotlivým cestám.

5.6.1 Optimalizace

Pro předem setřízenou posloupnost cest bude výpočet této optimalizace zabírat pouze lineární čas ($\mathcal{O}(n)$). Pokud by byla nesetříděná, je potřeba ji setřídit, časová složitost by tedy narostla na $\mathcal{O}(n * log(n))$. Zadefinujeme si funkce, potřebné k výpočtu optimalizace. Jedná se o čtyři funkce: tranzit cesty, tranzit množiny cest, počet potřebných kroků k přepravě určitého množství jednotek po jedné cestě a nakonec počet kroků k přepravě skrze množinu cest.

Nejprve začneme výrazem pro tranzit cesty `c` v i-tém kroku, vyjadřující celkový počet jednotek, které po cestě `c` dojdou do cíle do i-tého kroku:

$$T_c(i) = \begin{cases} (i - t) * p & \text{pro } i \geq t \\ 0 & \text{pro } i < t \end{cases}$$

[4] Funkce signum ve výrazu figuruje pro to, aby eliminovala $t(i)$.

Tranzit množiny cest `C` v i-tém kroku je součtem tranzitů jednotlivých cest:

$$T_C(i) = \sum_{c \in C} T_c(i)$$

Počet kroků k přepravě `x` jednotek po cestě `c` odpovídá počtu kroků, které skupina potřebuje, aby prošla místem dané šířky a počtu kroků, které jedna jednotka urazí po cestě:

$$K_c(x) = \left\lceil \frac{x}{p} \right\rceil + t$$

Zbývá vyřešit, jak spočítat počet kroků k přepravě `x` jednotek v množině cest. Zavedeme pojem virtuální cesty, abychom mohli použít méně složitý algoritmus. Virtuální cesty budou obsahovat délku, šířku a počáteční stav počtu jednotek `j`. Navíc si budou pamatovat, z jakých (virtuálních) cest jsou složeny. Upravme nyní výše uvedené vzorce pro použití s virtuální cestou. Vzroce platné pro virtuální cestu jsou tedy:

$$T_c(i) = \begin{cases} (i-t) * p + j & \text{pro } i \geq t \\ 0 & \text{pro } i < t \end{cases}$$

$$K_c(x) = \begin{cases} \left\lceil \frac{x-j}{p} \right\rceil + t & \text{pro } x \geq j \\ t & \text{pro } x < j \end{cases}$$

Nyní již přistoupíme k samotnému algoritmu. Nejprve je potřeba sloučit cesty stejných délek do virtuálních cest s šířkou cesty danou součtem šířek dílčích cest. Nyní máme pro každou délku maximálně jednu cestu této délky. Setřídíme množinu cest podle délky a postupně od nejkratší cesty rozšiřujeme virtuální cestu.

Začneme s prázdnou virtuální cestou. V `n`-té iteraci spočítáme její tranzit `y` v kroku odpovídajícím délce `n+1` cesty:

$$y = T_v(t(c_{n+1}))$$

Pokud je daný počet ostře menší než `x`, jež je vstupním parametrem ($K_C(x)$), uložíme získaný počet `y` do virtuální cesty jako novou počáteční hodnotu, k šířce přičteme šířku `n+1` cesty a nová délka bude odpovídat délce `n+1` cesty.

Takto postupujeme tak dlouho, dokud nalezené `y` nedosáhne (nepřekročí) počet `x` nebo dokud nesloučíme všechny cesty do jedné. V obou případech dostáváme jednu (virtuální) cestu `v`, a tím jsme převedli tento problém na předchozí, který již umíme řešit. $K_C(x) = K_v(x)$, kde v je takto získaná virtuální cesta. Tedy můžeme použít předchozí výraz:

$$K_v(x) = \begin{cases} \left\lceil \frac{x-j}{p} \right\rceil + t & \text{pro } x \geq j \\ t & \text{pro } x < j \end{cases}$$

Stále řešíme optimalizační úlohu, jak rozdělit jednotky cestám tak, aby se minimalizoval celkový čas průchodu. Nyní již umíme množině cest spočítat nezbytný počet kroků k přepravě daného počtu jednotek $i = K_C(x)$, i jak pro tyto cesty a nezbytný počet kroků spočítat, kolik jednotek dojde po které cestě ($T_c(i)$).

Díky zaokrouhlování může nastat situace, že součet jednotek přiřazený jednotlivým cestám bude vyšší, než skutečný počet jednotek (ale o jeden krok kratší

cesta by měla již méně přiřazených jednotek). V takovém případě jsme v naší implementaci snižovali počet přiřazený nejpočetnějším cestám tak, aby se obě hodnoty rovnaly.

5.6.2 Přidělení plánu jednotkám

Ze všech koexistujících množin jsme vybrali tu s nejmenším počtem kroků. Pro jednotlivé plány cest již známe počet jednotek, který jim náleží. Nyní musíme vybrat, který plán dostane každá jednotka. Toto rozdělení plánu jednotkám není triviální. Existují různé strategie přiřazení.

Náhodný výběr

Výpočetně nejjednodušší je postupně jednotkám přiřazovat plán, dokud se nenaplní počet jednotek, které mají jít jeho cestou, a poté se posunout k následujícímu plánu. Tento přístup má za následek neoptimální chování v exekuční fázi. Pořadí při postupném procházení jednotek je totiž náhodné. Konkrétně jsou jednotky přiřazovány v pořadí, v jakém vznikly, a pokud byly vytvářeny v GUI pomocí hromadného vytváření, je jejich pořadí zcela nezávislé na pozici, na které se nacházejí. V našem případě je tento postup postupného přiřazování ekvivalentní náhodnému výběru jednotky ze zbývajících, ještě nevybraných jednotek. Důsledkem je, že sousedící jednotky na počátku nemají stejné cíle a snaží se skrze sebe prodírat a vzájemně se brzdí, až kolidují. Tomuto přístupu budeme říkat v evaluaci náhodný výběr.

Pro přiřazení jednotek cestám by bylo vhodné použít strategii, která by zabránila vzniku tohoto jevu nebo by ho alespoň minimalizovala.

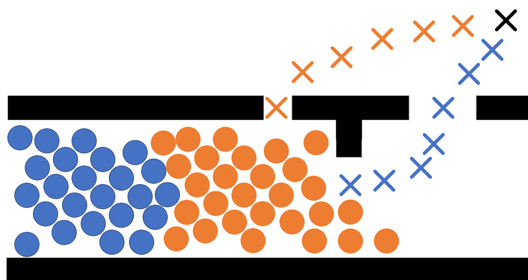
Skrytá závislost nezávislých skupin

Naivní heuristika přiřazující cesty, potažmo první brány, ve kterých se cesty liší, každou jednotku přiřazuje vždy k nejbližší bráně, která stále požaduje jednotky. Tato heuristika bude fungovat, ale není vždy optimální.

Jedna z problémových konfigurací pro tuto heuristiku je ilustrována obrázkem 5.10. Při něm jednotky čekají za jinou skupinou, na které měly být nezávislé, dokud tato skupina neprojde úzkým místem nebo se alespoň z části neuvolní volná cesta k dosažení prvního kroku v jejich plánu. Pokud by navíc cesta skrze vzdálenější cestu byla delší, pak se oproti strategiím používajícím první bránu (což znamená první cestu) celkový čas dosažení cíle všemi jednotkami naopak zhorší. Poté, co dorazí první skupina, nastane prodleva a až za chvíli začnou docházet jednotky ze druhé skupiny.

Nezávislé heuristiky

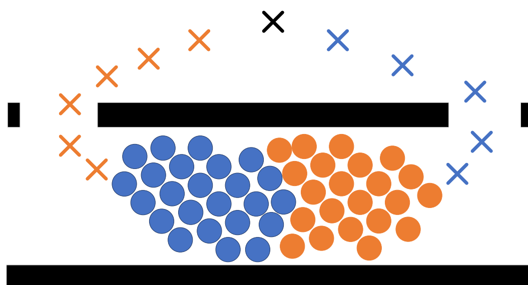
Tento jev řeší heuristika, která přiřadí jednotky k plánu jiným způsobem. Pro zjednodušení popisu fungování heuristiky předpokládejme, ve shodě s uvedeným obrázkem, pouze dva alternativní plány, které se navíc liší již svou první branou. Rovněž zanedbáme přesný počet přiřazení jednotek k jednotlivým cestám. Každé jednotce se spočítá vzdálenost k oběma branám a přiřadí se k té bráně, jež je vzdálenější (ve skutečnosti k plánu majícímu v prvním kroku tuto bránu).



Obrázek 5.10: Přiřazení plánu jednotkám má vliv na celkovou dobu simulace. Pokud plán oranžových jednotek povede skrze oranžovou cestu a modrých skrze modrou, musely by modré jednotky čekat, dokud se jim neuvolní cesta. Alternativní přiřazení jednotek by toto čekání snížilo.

Tímto způsobem by se opravdu přiřadily některé oranžové jednotky k modré bráně, ale pouze pokud by se na řadu dostaly před vyčerpáním kapacity plánu. Pro dosáhnutí kýženého přiřazení jednotek k branám rozdílné barvy (až na malý počet jednotek způsobený různou kapacitou plánů), bude potřeba zajistit pořadí přiřazování jednotek.

Upravme tedy pořadí přidělování jednotek následovně: vždy najdeme nepřiřazenou jednotku, která je nejbližší k libovolné první bráně plánu. Takto nalezenou jednotku přiřadíme k plánu, jehož brána je nejvzdálenější. Tímto dostáváme heuristiku, která řeší případ z obrázku 5.10. Ale právě tato heuristika je zástupcem heuristik, které při nevhodném rozložení mohou způsobit při exekuci uvíznutí – deadlock –, jak ilustruje obrázek 5.11 Jedná se o případ, ve kterém dvě části skupiny dostaly přiřazený plán, vedoucí přes počáteční pozice jednotek druhé části, navíc se nachází v místě, ve kterém se jednotky nemohou nijak vyhnout. Tato heuristika tedy také vytváří skrytou závislost.



Obrázek 5.11: Chybné přiřazení plánu jednotkám vedoucím k uvíznutí (deadlock). Jednotky vlevo mají podcíle vpravo a opačně. Bez dostatečného prostoru k vyhnutí skupin nemá exekuce možnost, jak se z této situace dostat. V hrách se obdobné situace řeší restartem navigace.

Zvolená heuristika

Zdá se, že pro každou konkrétní heuristiku lze najít protipřípad, ve kterém nefunguje optimálně nebo je příliš výpočetně náročná (no free lunch theorem). Původní problém i následující řeší zvolená heuristika. Její nevýhodou je však větší časová složitost.

Každé dva plány se liší alespoň jednou branou, jinak by se jednalo o jeden plán s kapacitou odpovídající jejich součtu. A takovéto plány slučujeme již při jejich případném vzniku. Navíc graf všech prvních bran jednotlivých plánů a bran na cestě k nim tvoří strom (kořenem je oblast, ve které se nacházejí jednotky).

Plány sdílející bránu budeme rozvrhovat dohromady. Poté budeme rozvrhovat jednotky přiřazené pouze této bráně a přiřazujeme tedy pouze jednotky jdoucí alespoň z části stejným směrem. Proto na tento zbytek lze použít již zmiňovanou heuristiku „nejprve přiřadit jednotky vzdálenější bráně“.

Každé bráně spočítáme vzdálenost k nejbližším j jednotkám, kde j je počet jednotek, který má danou branou projít. Pro všechny jednotky vyhodnotíme, kolik bran je má ve svých nejbližších jednotkách.

Může nastat jeden ze tří případů: žádná, právě jedna a více než jedna. V případě, že právě jedna brána danou jednotku má mezi svými nejbližšími, ji přiřadíme této bráně. V případě jednotek, kterou mají v nejbližších více než jednu bránu, změříme vzdálenosti k branám a vybereme jednotce tu nejbližší. Pro zbývající jednotky, které nebyly nejblíže opakujeme výběr tak dlouho, dokud nebudou přiřazeny všechny jednotky.

5.6.3 Převod seznamu bran na plán cesty pro jednotku

Plán, se kterým jednotka pracuje, obsahuje souřadnice v mapě. Zde ale stále pracujeme s tím, že plán obsahuje seznam bran, kterými má jednotka projít a poslední položkou plánu není brána, ale pozice cíle.

Brána je množina indexů v gridovém poli, obdobně i cesta jednotky je množina bodů v gridových souřadnicích. Problém, který zde řešíme, není jen nalezení cesty mezi branami (to jsme již udělali pro měření jejich vzdáleností), ale kterými konkrétními body brán povede výsledná cesta.

Jelikož oblasti nejsou konvexní, nejen svým vnějším tvarem, ale objevují se překážky i uvnitř oblastí, není zaručena přímá viditelnost mezi branami. Proto předat jednotce pouze seznam bran k projití nestačí. Pokud by dekompozice byla konvexní (nebo ideálně, pokud by byla na trojúhelníky), znamenalo by to, že brány by tvořily celou stěnu a my bychom mohli použít vyhledávání, které používá Funnel algoritmus. Díky němu by bylo možné najít přímou cestu jdoucí přes několik oblastí, čímž by se zkrátila výsledná nalezená cesta. Takto nalezená cesta je vidět na obrázku 3.1, kde mezi body A a B vede oranžová spojnice přímo. Připomenutí samotného fungování algoritmu je na obrázku 3.2. Jelikož ale oblasti konvexní nejsou, musíme se spokojit pouze s přibližnou cestou, která ovlivní kvalitu řešení.

Výslednou cestu plánujeme po částech vždy mezi středy sousedních bran. Tuto cestu plánujeme pomocí A* algoritmu a skládáme ji dohromady. Tento přístup vede cestu nalezenou pomocí A* u velmi širokých bran, daleko od optimální cesty, jak lze i vidět na levém obrázku 5.3. Avšak díky poměrně chytrému path followingu (viz sekce 4.3.2 s obrázkem 4.1) se v exekuční části jednotek tato

nepřesnost neprojevuje tolik, jako když by byl path following řešen průchody ve fixní vzdálenosti od bodů cesty. Na druhou stranu mají jednotky i ve větší skupině tendenci držet se během exekuce více u středů bran, tím mají volný prostor k manévrování a dochází tak méně ke kolizím jak s ostatními jednotkami, tak se stěnou.

První brány

Poslední rozdíl mezi grafovými reprezentacemi, který se posléze projeví v plánování je přeskočení navigace k prvnímu vrcholu. U Středového grafu přeskočíme cestu k prvnímu vrcholu, jenž je středem oblasti, ve které se nacházejí jednotky (vrchol samotný se může vyskytovat daleko od jednotek). Tak jednotky rovnou zamíří k druhému vrcholu – k bráně. Bezstředový graf tímto netrpí, protože se jednotky nacházejí v těsné blízkosti vrcholu.

Jednotkám neplánujeme cestu k první bráně (první brána je první brána v plánu po aplikaci výše uvedeného). Máme předpoklad, že je viditelná ze současné pozice jednotek, což nemusí platit, jak je vidět např. na obrázku 5.6b. Pokud by zde jednotky v největší oblasti byly nahoře a chtěly jít doleva, potřebovaly by v rámci této oblasti nejprve dojít ke středu oblasti, aby se toto urychlení neaplikovalo[5].

Každá jednotka dostane přidělený plán cesty, a tím plánování končí a o zbytek se postará exekuční část navigace.

[5]V simulátoru je možné toto chování zajistit, pokud se v souboru `UnitPathAssignment` nedefinuje preprocesorová direktiva `ConvexRegions`

6. Evaluace

V této kapitole se seznámíme s testy provedenými v simulátoru, který jsme naimplementovali právě pro provedení těchto experimentů. Potřebujeme vyhodnotit námi objevený a naimplementovaný plánovací algoritmus skupinové navigace v jeho obou variantách a porovnat ho se stávajícími algoritmy s nekoordinovaným přístupem.

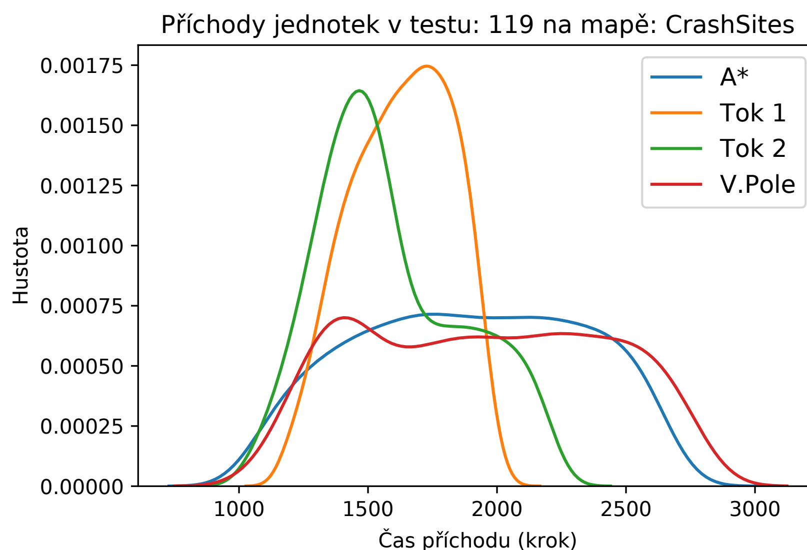
6.1 Porovnávané metody plánování

Testujeme čtyři metody. Námi naimplementované metody používají dva typy grafů. Metodu se Středovým grafem označme jako Tok 1 a metodu s Bezstředovým grafem jako Tok 2. Metody založené na následování plánu jsou Tok 1, Tok 2 a nekoordinovaný A^*. Poslední metoda je založená na Vektorovém poli. Naše metody jsou tedy Tok 1 a Tok 2.

6.1.1 Popis chování obou způsobů navigace při exekuci

Pokud se jednotka metody sledující plán dostane daleko od oblasti, v níž má naplánovanou cestu, a začne být výhodnější použít jinou cestu než setrvat na stávající cestě, i přesto zůstává na své původní cestě a snaží se o návrat. Jednotky se navzájem mohou vytlačovat ze svých plánovaných cest. Pokud se jednotka dostane do situace, že nevidí na žádný svůj dílčí cíl nebo pokud je cíl příliš vzdálený, tak se ztratí. V takových případech dojde k přeplánování.

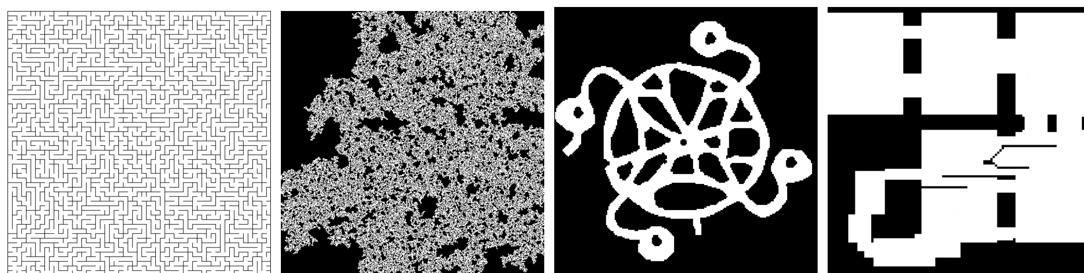
Pokud jednotka metody využívající Vektorové pole bude ve stejné situaci, půjde po nejkratší cestě do cíle. K přeplánování nikdy nemůže dojít, protože tato metoda nevyužívá plán cesty.



Obrázek 6.1: Zajímavý výsledek, na kterém je vidět plató indikující dosažení maximální kapacity hrany, která ovlivňuje i část Toku 2.

6.2 Výběr map

Testy budeme provádět na různých mapách, které z větší části poskytne již zmíněný benchmark 4.2. Mapy, které používáme pro porovnání, byly vybrány na základě následujících kritérií. Musely mít prostor na umístění skupiny jednotek, čímž se eliminovaly bludišťové a náhodné mapy. Dalším kritériem byla existence alternativních cest s dostatkem volného prostoru pro umístění takového počtu jednotek, aby zahltily nejužší místo cesty. Posledním kritériem bylo, aby na dané mapě správně fungovala dekompozice mapy, což se díky více možnostem dekompozice neukázalo být velkým problémem, protože i přibližná dekompozice funguje dostatečně dobře. Mapy z benchmarku jsme prošli a vyfiltrovali pouze ty mapy, na kterých se objevují místa splňující naše požadavky. Mapy, které jsme sami vyrobili pro účely testování, s vlastnostmi, které zvýrazní chování jednotek, nazvěme syntetické mapy. Syntetické mapy jsou druhým typem map, které při testech používáme.



Obrázek 6.2: Různé typy map. Zleva: bludiště, náhodná mapa, mapa ze hry a syntetická mapa.

6.3 Konfigurace experimentu

Jak jsme již předeslali, pro vytvoření experimentu je potřeba zvolit mapu a dále pak vybrat vhodný parametr pro dekompozici mapy, který ovlivňuje podobu grafu. Zejména je potřeba dbát na velikosti bran, jejich počet a vzájemnou konfiguraci v místech, kterými by mohly jednotky při simulaci procházet. Určit kolik jednotek, z jakého místa a kam mají jít. Poté tuto konfiguraci uložit. Jak se tyto operace dají v programu vykonat je popsáno v uživatelské dokumentaci v sekci příprava experimentu 7.1.3. Špatně zvolená dekompozice ovlivňuje výsledek, proto je potřeba ji zvolit vhodně. Příklady problémů, které nastávají při dekompozici, jsou zobrazeny na obrázcích 5.6.

6.3.1 Provedení experimentu

Experimenty provádíme v našem simulátoru v dávkovém režimu, bez obrazového výstupu. Na příkazové řádce předáme soubor s konfigurací běhu a názvem metody, která se má testovat. Pro porovnání všech metod navigace je potřeba opakovaně spustit simulátor pro každou metodu s příslušnou hodnotou parametru. V tomto režimu se program po skončení simulace a uložení výsledků sám

vypne. Protože je chování všech metod v exekuční fázi navigace nedeterministické, a u našich metod může k nedeterminismu při shodě u křížení cesty dojít také v plánovací fázi, provádíme testy vícekrát.

Testy lze alternativně spouštět s grafickým výstupem, a to následujícím způsobem: Po spuštění simulátoru (v grafickém módu), načteme mapu a uloženou konfiguraci experimentu, zvolíme typ jednotky a spustíme simulaci. Po doběhnutí jednotky program vypíše potvrzení o doběhnutí a můžeme ho buď vypnout, nebo pokračovat v testování. Tento způsob jsme však pro finální testování nevyužili.

6.4 Čítače

Měření provádíme v rámci simulátoru, kde existuje několik globálních čítačů, přičemž každá jednotka navíc po dobu své existence během simulace obhospodařuje několik vlastních čítačů. Simulace začíná ve stavu, kdy všechny jednotky existují a jsou na správných pozicích, po celou dobu běhu žádná další jednotka nepřibude. Cíl je na správném místě a byla již provedena dekompozice mapy. Na začátku simulace jsou všechny čítače nastavené na nulu (kromě počtu jednotek). Po spuštění se v každém kroku každé jednotky aktualizují její čítače. Především ten, jenž určuje, kolik kroků strávila jednotka na cestě do cíle. Poté, co jednotka dorazí do cíle, se zastaví aktualizace jejích čítačů a jednotka se ze simulace odebere. Po doběhnutí poslední jednotky se stav čítačů všech jednotek předá ke statistickému zpracování. Pro každý čítač se vypočítá průměrná hodnota, medián, minimum, maximum, směrodatná odchylka, rozptyl první a třetí kvantil a suma.

6.4.1 Význam čítačů

V následujícím textu si postupně rozebereme význam jednotlivých čítačů a přidržíme se názvů, jaké má výsledný `CSV` soubor:

Začneme čítačem `wallCollisions`. Tento čítač počítá, kolikrát se daná jednotka ocitla ve zdi. Přesněji řečeno pro to, aby jednotka hodnotu tohoto čítače navýšila, je zapotřebí, aby na konci aktuálního kroku její vzdálenost ke stěně od středu jednotky byla menší než její poloměr. Zvýší se tedy pouze o 1 i v případě, že koliduje s několika stěnami najednou.

Obdobně se i čítač `unitCollisions` zvýší pouze o 1, i když by jednotka kolidovala s více jednotkami.

Čítač `numberUpdates` se zvyšuje v každém kroku simulace, jakmile se na jednotku dostane řada ve zpracování. Pokud by jednotka začínala v cíli, skončí s hodnotou 1.

Čítače `notFullmove`, `cantMove` i `distanceTravelled` se počítají při aktualizaci pozice jednotky. Pokud se jednotka nehne o vzdálenost odpovídající její maximální rychlosti, je navýšen první. Pokud navíc je rychlost menší než malé epsilon, je navýšen i druhý. Třetí z čítačů se jako jediný nenavyšuje pouze po jedné. Navyšuje se v každém kroku, přičítá k doposud uražené celkové vzdálenosti rozdíl souřadnic středů jednotky před a po aktualizaci pozice.

Čítač `Replans` je globální pro všechny jednotky. Tento čítač se navýší pokaždé, když jednotka detekuje, že se ztratila a během uživatelem nastavitelného počtu kroků zůstává ztracenou. V takovém případě si pro sebe jednotka vyžádá od plánovače novou cestu do cíle. Tím přestává plnit původní plán a vydává se k cíli

po nejkratší cestě nalezeným algoritmem $A*$, i když je součástí testu pomocí jiné metody. Toto chování odpovídá tomu, jak se v hrách řeší problémy s uvíznutím jednotek. Přeplánování není dostupné pro metodu navigující pomocí vektorového pole.

6.4.2 Další položky souboru

Soubor obsahuje informace o samotném experimentu: jeho konfiguraci, umístění konfiguračního soboru a jeho parametrech.

`UnitType` je parametr určující, která ze zvolených metod simulace běžela.

`FloodParam` Je celé číslo v rozsahu 0-10 určující dekompozici mapy.

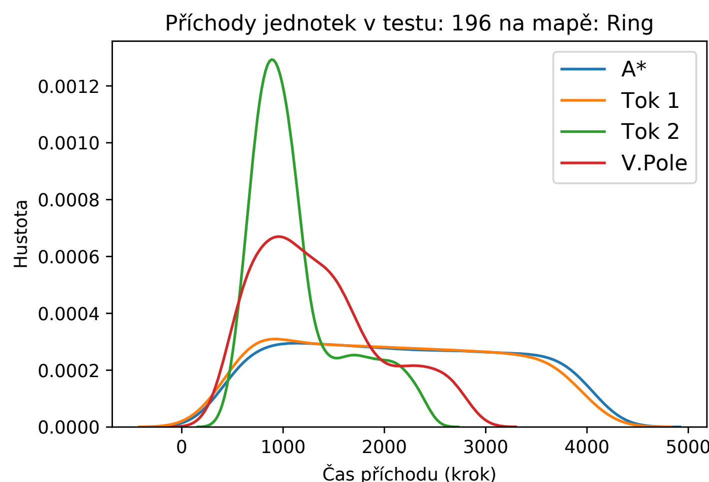
Posledními hodnotami, které se mohou lišit mezi jednotlivými testy stejné konfigurace, je dvojce `FinishedUnits` a `NotFinished`. Díky existenci časového limitu nebo hlídání dlouhé prodlevy doražení jednotek se může stát, že některé jednotky nestihnou včas dorazit do cíle. Speciálně ty, které se ztratily a uživatel v globálním nastavení programu zvolil nepřeplánovávat jednotky. Doběhnuté a nedoběhnuté jednotky doplňuje hodnota celkového počtu jednotek `Units`.

Binární hodnota `TimerCutoff` signalizuje, zda byl běh ukončen, protože došlo k vyčerpání uživatelem stanoveného času na simulaci či, pokud pro dostatečně velký počet kol nedošla žádná jednotka a zároveň jich na mapě zbylo jen několik. Tyto simulace, ve kterých navigace selže, se ukončují automaticky po vyhodnocení chyb a uložení výsledků.

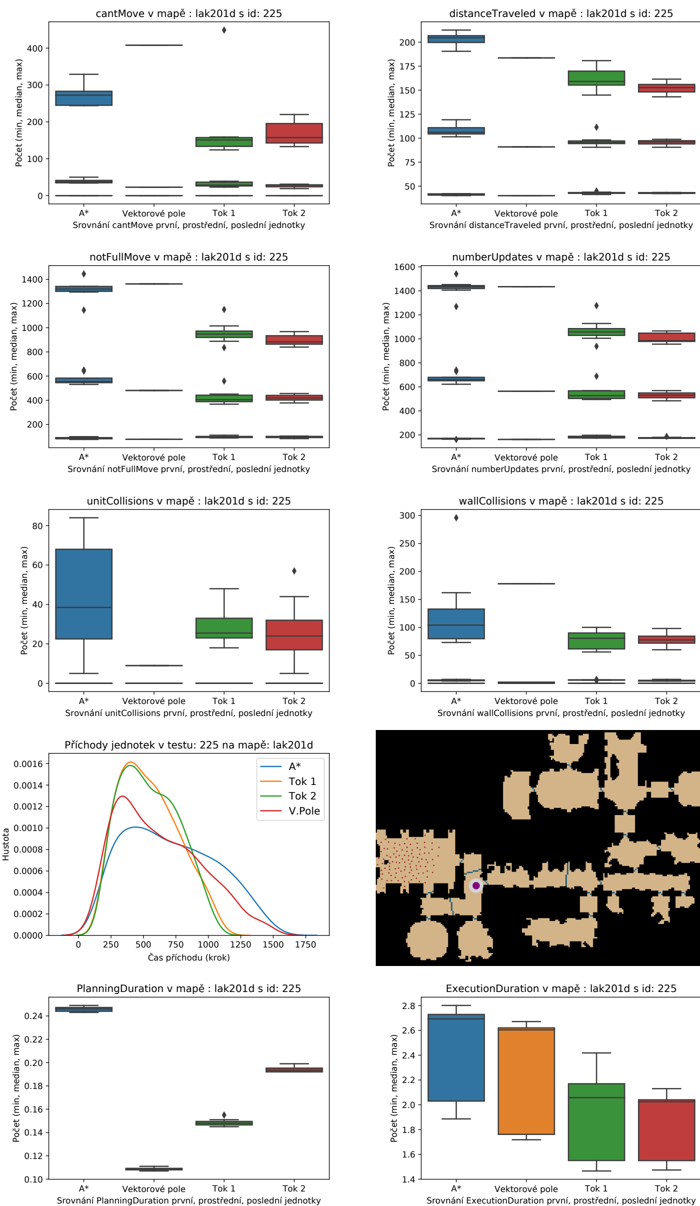
Binární hodnotu `BatchMode` – zda test byl prováděn po celou dobu bez grafického výstupu.

Identifikační číslo testu `ID`. Toto číslo souhlasí s přiloženými soubory, složkami s nafocenými testy, konfiguračními soubory, přiloženými grafy i s ID v tabulkách a obrázcích.

A konečně časy, jak dlouho v simulaci běžela navigace: `PlanningDuration`, `ExecutionDuration`. Připomínáme, že se doba běhu dekompozice se do těchto časů nezapočítává.



Obrázek 6.4: Syntetický test, v němž naše metoda Tok 1 používá na rozdíl od metod Tok 2 a A* i ostatní cesty.



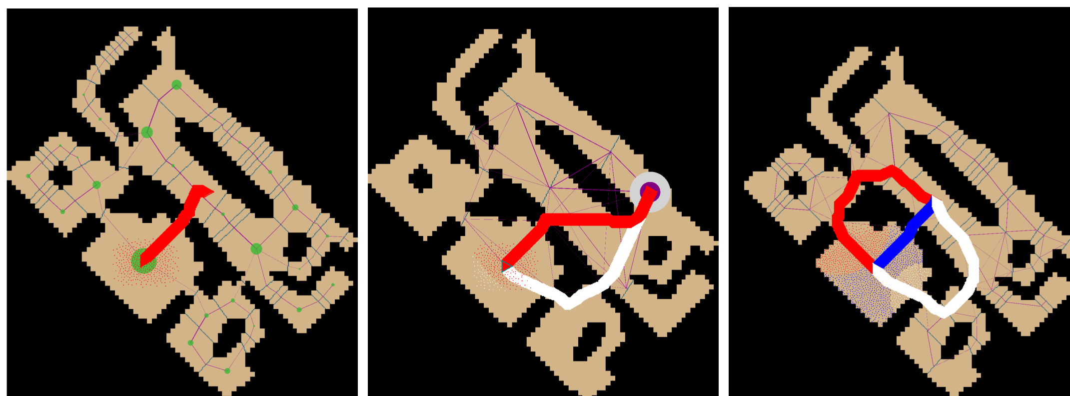
Obrázek 6.3: Ukázka téměř kompletní sady grafů pro experimet s ID 225, včetně mapy a pozic jednotek a cíle. Původní soubory jsou results-225.csv a kombinace souborů arrivalTimes. V grafech, kde se nad sebou nachází tři boxploty, patří každý jedné jednotce stejné metody. Jedná se vždy o první, prostřední a poslední jednotku (dle pořadí, v jakém dosáhla cíle). Na boxplotech je dobře patrná variabilita během jednotlivých běhů. Navigace pomocí vektorového pole v souladu s předpoklady tuto variabilitu má velmi nízkou.

6.5 Testované scénáře

Testujeme tři scénáře. Především testujeme již avizovanou situaci úzkého místa s alternativními cestami. Rovněž však testujeme situace, ve kterých i ostatní metody dělí skupinu a naše metoda by neměla mít velký prostor ke zlepšení celkové rychlosti. Naopak se projeví, vhodnost poměru rozdělení skupiny na části a přiřazení konkrétních jednotek konkrétním plánům. Posledním scénářem je navigace mezi místy, kde alternativní cesta neexistuje nebo by bylo nevhodné ji použít.

6.5.1 Měření

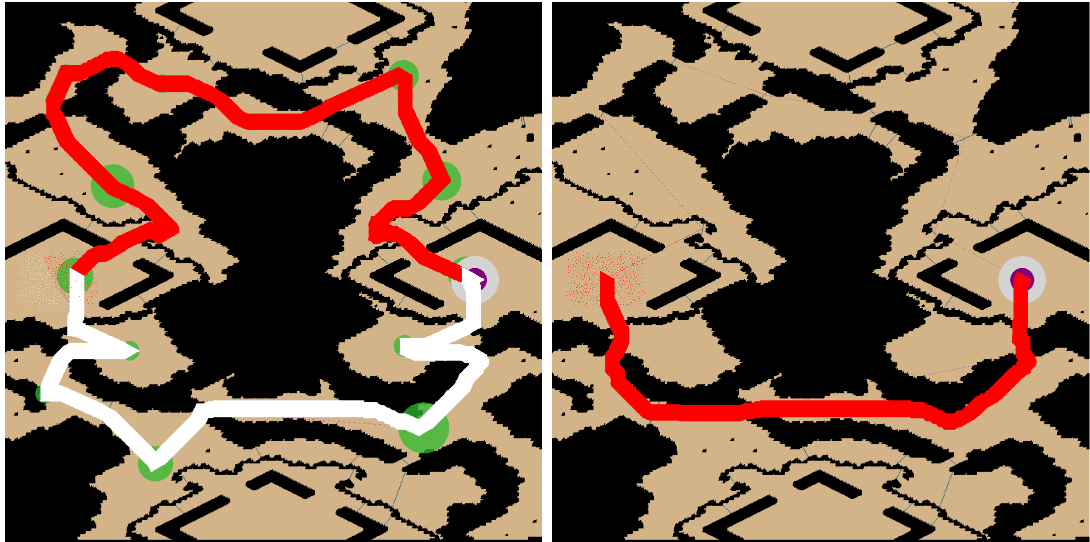
Vytvořili jsme tři sady testů, pro každý testovací scénář jeden. V první sadě jsou testy, ve kterých neočekáváme dosažení výrazného zlepšení oproti ostatním metodám. Ve druhé sadě se nacházejí testy, ve kterých bychom mohli být o něco lepší než alternativní metody. Ve třetí sadě jsou situace, ve kterých bychom měli být nejlepší. Příklady experimentů z jednotlivých sad lze vidět na obrázku 6.5.



Obrázek 6.5: Příklady experimentů patřící do různých testovacích scénářů s uvedenými ID testů. Barvy jednotek odpovídají plánu, ke kterému jsou přiřazeny.
Vlevo: Experiment s id 2 z první sady testů, ve kterém použití alternativní cesty prodlouží celkovou dobu přesunu jednotek.
Uprostřed: Experiment s id 1 z druhé sady testů, ve kterém i ostatní metody dělí skupinu. Dělení naší metodou by však mělo rozdělit skupinu optimálněji.
Vpravo: Experiment s id 3 ze třetí sady testů, ve kterém se projeví výhoda naší metody nejvýrazněji.

6.6 Vyhodnocení

Pro zpracování výsledků je obtížné vybrat objektivní metriku, která by jedním číslem vyjadřovala, která z metod je nejlepší. Důvodem jsou mimo jiné chyby navigace nastávající při exekuci. Výstupy našich měření lze lépe porovnat v grafech, kde z jejich průběhu lze vypozorovat blíže jejich chování. Přesto chceme mít k dispozici i dané číslo. Pro analýzu celkového počtu kroků v exekuci je nejdůležitější počet updatů poslední přišlé jednotky. Ostatní data z čítačů jen poskytují doplňkový obraz na provedenou simulaci. Pro vyhodnocení počtu kroků všech



Obrázek 6.6: Experiment s ID 253, ve kterém Tok 1, kvůli příliš málo dekomponované mapě, použije alternativní cestu která je ve skutečnosti delší. V bíle cestě jsou tři středy, které prodlužují vypočtenou bílou cestu na tolik, že je použita i červená. Vpravo pro porovnání Tok 2 jehož trasa, obdobně jako trasa pro zbývající metody, použije pouze dolní cestu.

jednotek by bylo příhodné použít místo maxima median, průměr či třetí kvartil. Všechny tyto funkce by pomohly v případě, kdy se při exekuci jednotka zasekne na poměrně dlouhou dobu, ale zbytek skupiny již dávno dorazil na místo určení, naopak by však zkreslovaly chování algoritmů projevující se až ke konci simulace, například doražení skupiny jdoucí po dlouhé cestě.

Proto jsme zvolili nakonec maximum s tím, že experimenty opakujeme a používáme median přes všechny provedené běhy simulace, abychom předešli zkresleným výsledkům kvůli náhodné chybě při exekuci. Přesto měřit poslední přišlou jednotku není k našim metodám příliš shovívavé. Protože se na rozdíl od ostatních metod stává, že jednotky jdou různými směry, a tak pravidelně některá z jednotek zůstávavá uvězněna v davu jednotek, které ji odtlačí daleko od startu, a až když se ze skupiny vymaní a skupina odejde, vydá se ztracená jednotky zpět na start, a tak se celkový čas prodlouží, i když většina jednotek mezitím zdárně dorazila (viz graf 6.8). Toto je způsobeno buď až při exekuci, nebo nedokonalým přidělením nalezených cest jednotkám heuristikou.

K vyhodnocení používané hodnoty jsou mediánem počtu kroků simulace pro cca 10 měření každé metody. Počet kroků simulace se rovná maximu kroků jednotek. Na základě naměřených statistik jsme vytvořili souhrnou tabulku 6.7, z níž je zřetelně vidět úspěšnost jednotlivých metod, pro jednotlivé scénáře. Hodnoty v tabulce jsou poměrem vůči nejrychlejší metodě v daném testu. Případy, ve kterých v testech Tok 2 nedopadal dobře, byly většinou způsobeny neoptimální dekompozicí. Ve zbylých případech byla viníkem exekuce. Tok 2 se ukázal lepší metodou než Tok 1, který ale podává na horších dekompozicích lepší výsledky, protože při malých změnách vstupu podává téměř totožné výsledky. Jelikož obsahuje menší počet hran, je i rychlejší. Výpočet naší metody i pro složitější graf je ve většině případů srovnatelně rychlý jako výpočet vektorového pole. Při vyšším počtu jednotek je A* řádově pomalejší. Naše metoda má však téměř konstantní

ID	A*	Tok 1	Tok 2	Vekt
2	1.01	1.00	1	1.01
5	1.33	1.05	1.03	1
6	1.33	1	1.33	1.32
7	1.55	1.48	1.13	1
23	1.00	1	1.01	1.08
51	1.27	1.10	1	1.08
54	1.65	1.05	1.01	1
55	1.05	1.04	1	1.03
56	1.01	1	1.00	1.05
57	1.04	1	1.02	1.08
58	1	1.27	1.04	1.06
59	1.30	1.98	1.17	1
68	1.03	1.02	1	1.27
71	1.11	1.13	1	1.11
72	1.05	1	1.04	1.11
80	1.01	1.01	1	1.06
81	1.57	1.01	1.37	1
86	1.06	1.41	1.17	1
106	1.65	1.21	1.31	1
120	1	2.31	1.01	1.08
121	1.24	1	1.08	1.08
130	1.13	1.01	1	1.04
133	1.00	1.05	1	1.05
136	1.01	1	1.00	1.07
138	1.07	1.03	1	1.05
139	1.02	1.37	1.12	1
202	1.24	1	1.08	1.02
204	1.02	1.04	1.01	1
210	1.10	1.11	1	1.37
214	1.76	1.51	1.24	1
215	1.02	1.01	1	1.07
227	1.08	1	1.08	1.11
229	1	1.01	1.01	1.04
230	1	1.00	1.00	1.04
235	1.43	1	1.11	1.14
240	1.20	1.22	1	1.20
246	1.18	1.01	1	1.30
251	1.13	1.19	1	1.10
252	1	1.06	1.00	1.13
253	1.00	1.13	1	1.07
254	1.20	1	1.13	1.26
259	1.09	1.06	1	1.11
260	1.01	1.01	1	1.07
261	1.03	1.02	1	1.16

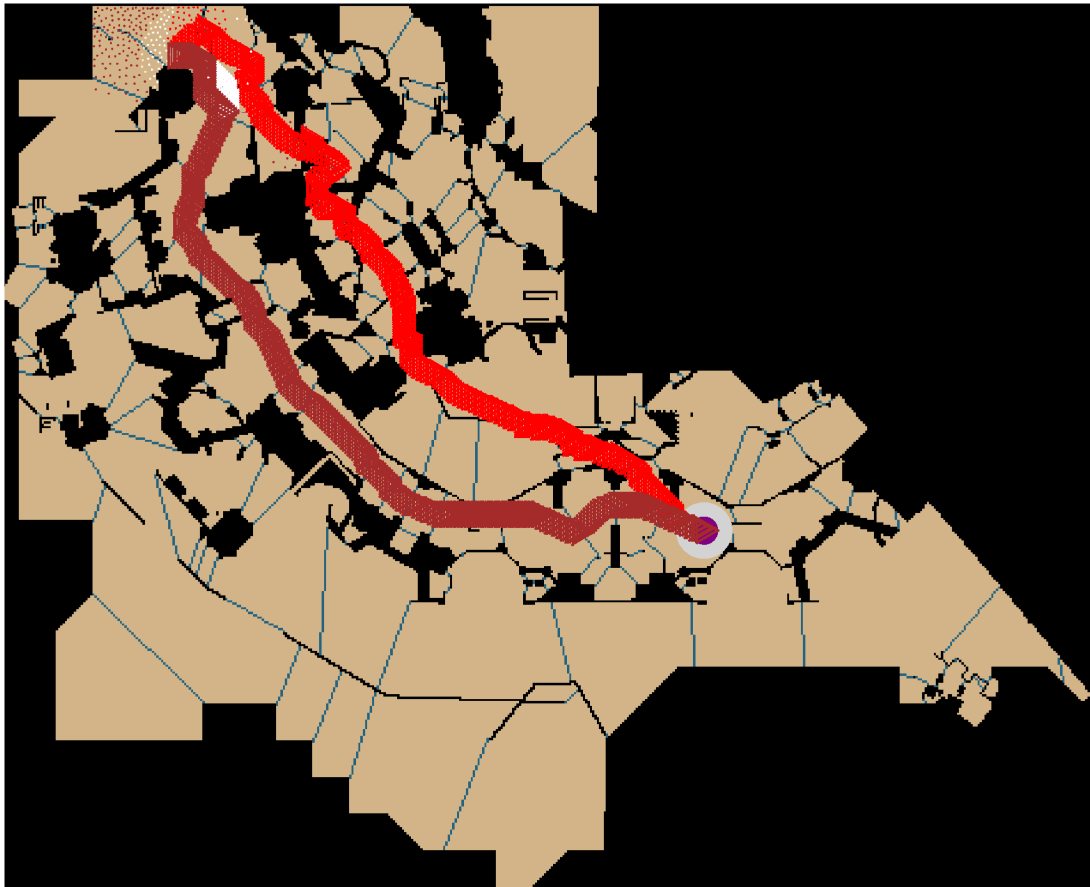
ID	A*	Tok 1	Tok 2	Vekt
0	1.27	1	1.01	1.05
1	1.09	1.09	1	1.14
4	1.30	1.03	1	1.36
8	1.31	1.32	1	1.41
9	1.18	1.23	1	1.47
15	1.09	1.20	1	1.06
27	1.21	1.22	1	1.38
60	1.03	1.05	1	1.64
61	1.08	1.10	1.07	1
62	1.20	1.05	1	1.16
65	1.21	1	1.01	1.23
69	1.16	1	1.04	1.09
70	1.20	1.12	1	1.03
73	1.53	1.08	1	1.08
74	1.10	1.09	1	1.19
77	3.18	1	1.54	1.45
78	1.23	1.03	1.00	1
82	1.39	1.03	1	1.47
84	1.11	1	1.11	1.13
85	1.19	1	1.09	4.13
87	1.24	1.11	1.05	1
95	1.30	1.15	1	1.20
108	1.17	1.18	1	1.24
117	1.12	1	1.17	1.36
119	1.34	1	1.12	1.41
122	1	1.12	1.12	1.11
124	1.40	1.28	1	1.39
129	1.00	1	1.00	1.11
131	1.04	1	1.02	1.27
134	1.36	1	1.28	1.50
135	1.54	1	1.52	1.67
137	1	1.13	1.20	1.09
206	1.25	1.11	1.15	1
208	1.07	1.05	1	1.08
209	1.09	1.02	1	1.06
231	1	1.00	1.00	1.04
238	1.16	1.03	1.01	1
241	1.18	1.24	1	1.18
244	1.30	1	1.29	1.35
250	1.81	1	1.70	1.51
257	1.31	1.36	1	1.12
258	1.19	1.30	1.08	1
264	1.20	1	1.08	1.38
265	1.46	1	1.01	1.15

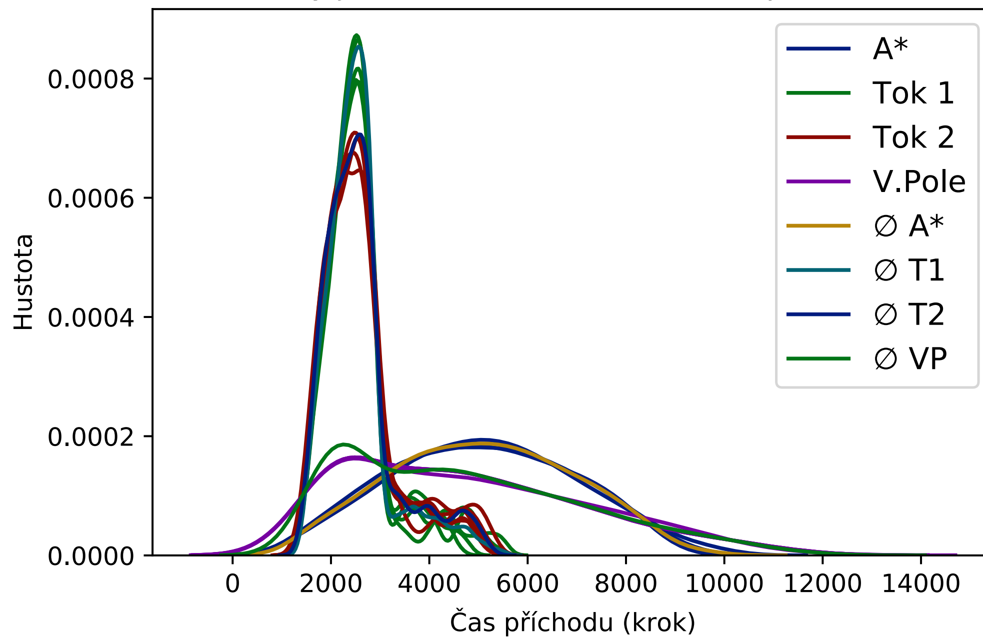
ID	A*	Tok 1	Tok 2	Vekt
3	1.28	1	1.03	1.43
28	1.55	1.14	1	1.84
29	1.82	1.20	1	2.01
30	1.67	1.05	1	2.10
31	1.66	1	1.39	1.97
63	1.53	1.04	1	1.70
64	1.22	1.13	1	1.32
66	1.04	1.05	1	1.08
67	1.17	1.00	1	1.63
79	1.22	1.04	1	1.29
98	1.48	1.16	1	1.45
99	1.44	1.21	1	1.46
128	1.51	1.13	1	5.90
132	1.06	1	1.03	5.51
195	1.53	1.51	1	1.67
196	1.66	1.65	1	1.15
197	3.00	1	1.49	2.03
198	2.25	1	1.20	1.08
199	1.26	1.10	1.06	1
216	1.56	1.36	1	1.05
217	1.23	1.05	1.06	1
218	1.22	1.09	1	1.29
219	1.55	1	1.18	1.41
220	1.43	1.23	1.19	1
221	1.25	1.26	1	1.00
222	1.78	1.15	1.27	1
223	2.28	1	1.57	1.20
224	1.15	1.28	1	1.14
225	1.50	1.09	1	1.52
226	1.27	1.00	1	1.46
228	1.06	1.02	1	1.14
232	1.75	1	1.07	2.17
233	1.14	1.04	1	1.19
234	1.19	1.03	1	1.24
236	1.83	1.12	1	1.09
237	1.83	1.09	1	1.08
239	1.30	1.30	1	1.68
242	1.10	1.01	1	1.11
243	1.23	1.22	1	1.14
245	1.02	1	1.19	1.05
247	1.39	1	1.21	1.59
262	2.21	1	1.66	2.34
263	2.70	1	1.48	2.22
266	1.13	1.40	1	1.16

Obrázek 6.7: Tabulky s porovnáním metod ve třech testovaných scénářích. Námi implementované metody jsou ve dvou prostředních sloupcích. Hodnoty vyjadřují poměr času příchodu poslední jednotky dané metody k vítezné metodě. Barvy indikují o procentuální rozdíl, tmavě zelená do 1%, světle zelená do 10%, oranžová do 50% a čerená nad 50%.
Vlevo: Experimenty scénáře, ve kterém není možné nebo vhodné dělit skupinu.
Uprostřed: Experimenty scénáře, ve kterém i zbylé metody dělí skupinu.
Vpravo: Experimenty scénáře, který se zaměřuje na experimenty, vemli vhodné pro rozdělení skupiny na více částí.

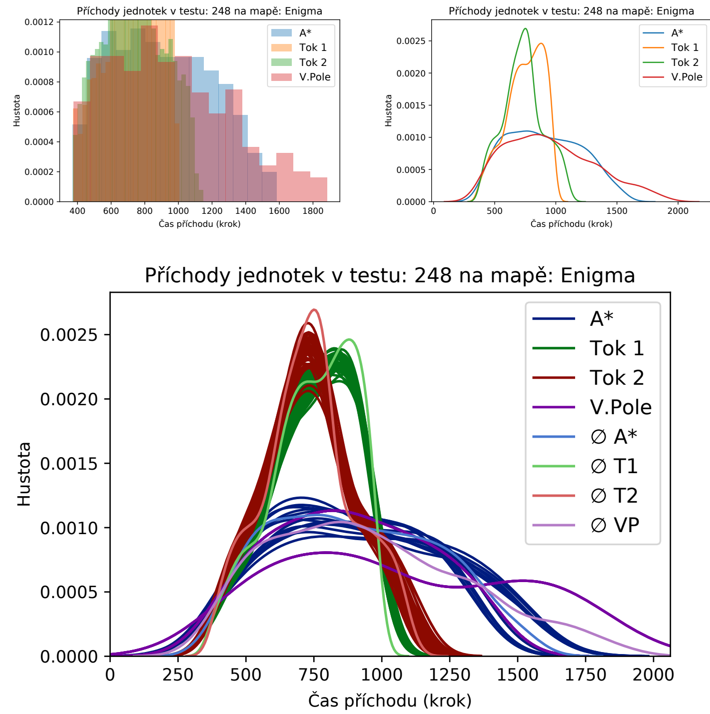
dobu běhu k počtu jednotek, které naviguje. Nehledě na počet jednotek se vyhledají všechny možné cesty a pouze doba běhu výsledné optimalizace a heuristického přiřazení jednotek závisí na jejich počtu.

Náš předpoklad, že řízení skupiny pomocí toků v síti uspěje v porovnání s ostatními metodami, se ukázal být pravdivý, a to ve všech testovaných scénářích. Data jsou součástí přílohy i s vygenrovanými grafy a IPython skriptem, které se nacházejí na přiloženém CD spolu s mnoha sty dalšími měřeními pro různé konfigurace heurisitckých funkcí a jiných nastavení, například nastavení, zda brány mají být spojeny centrem nebo nejbližšími body.

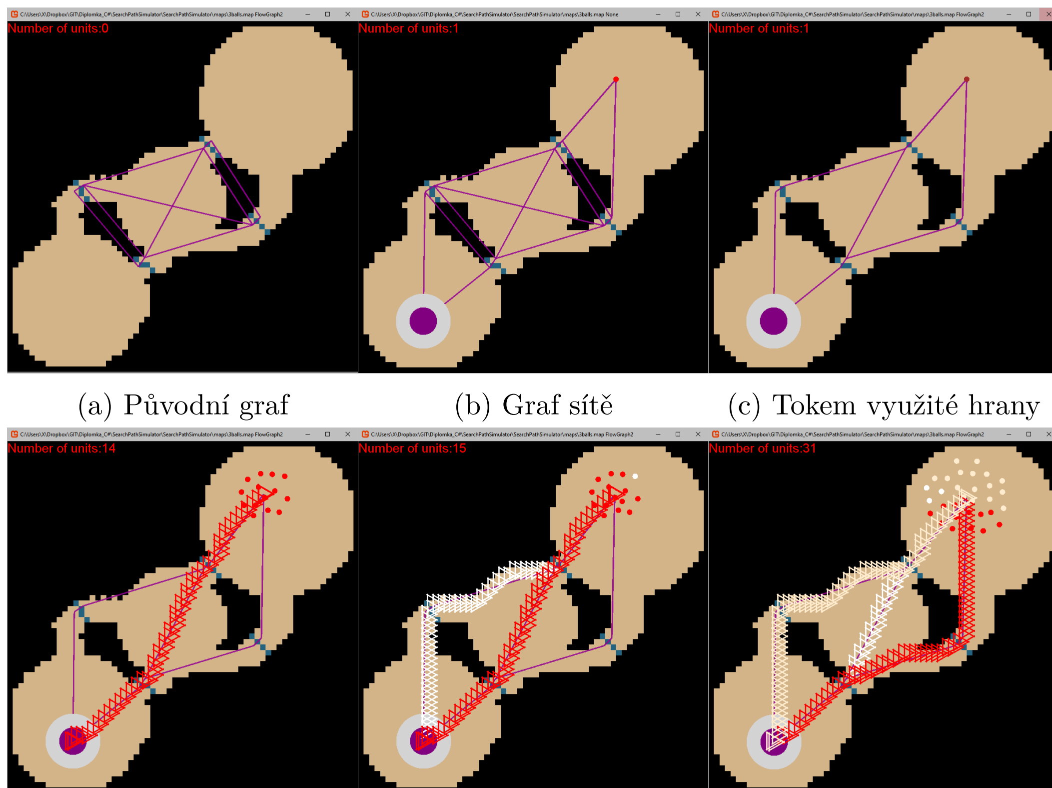




Obrázek 6.8: Experiment s ID 232, na nemž naše metoda dorazí do cíle asi za polovinu času, který potřebují zbylé metody. V grafu našich metod je vidět zpomalení způsobené exekuční částí.



Obrázek 6.9: Výsledky experimetu pro příchody (nejen posledních) jednotek pro test s ID 248. Histogram není pro zobrazení více metod přes sebe příliš přehledný, proto jsme zvolili funkci Kde - Kernel density estimation (jádrový odhad hustoty), která umožňuje přehlednější zobrazení, a to i pro mnoho běhů experimentu.



(a) Původní graf (b) Graf sítě (c) Tokem využité hrany

Obrázek 6.10: Námi implementovaná metoda má očekávané chování, kdy se při navyšování počtu jednotek mění využití jednotlivých cest.

7. Dokumentace

7.1 Uživatelská dokumentace

V této kapitole představíme ovládání programu, požadavky, módy, ve kterých jej lze spustit. Uvedeme i příklad jak vytvořit, spustit, uložit a nahrát experiment.

7.1.1 Požadavky ke spuštění

Program byl testován na 64–bitových operačních systémech Windows 7 a Windows 10. Pro spuštění aplikace je nutné běhové rozhraní .NET[1] ve verzi 4.7.1 (nebo novější). Další potřebné knihovny (MonoGame Framework, MonoGame Extended, OptimizedPriorityQueue) jsou k dispozici skrz balíčkovací systém NuGet a všechny zmíněné knihovny používáme v době psaní v jejich aktuálních verzích[2].

Simulátor spouštíme souborem `SearchPathSimulator.exe` a i z příkazové řádky se spustí v grafickém režimu. Z příkazové řádky lze spustit experiment přímo, bez nutnosti použít menu simulátoru. V takovém případě syntaxe příkazové řádky za názvem spustitelného souboru vyžaduje dva parametry: konfigurační soubor experimentu a název testované metody.

```
SearchPathSimulator.exe  "cesta/k/experimentu.scenario.xml"  "AStar"
```

7.1.2 Textový režim

Jedná se o režim, v němž se nezobrazuje simulace pohybu jednotek, nelze nic nastavovat, přesto se z tohoto režimu lze i v průběhu výpočtu přepnout do grafického režimu. Tento režim je zamýšlen pro vykonání samotných experimentů. Při nich se výsledek zapíše do výstupního souboru a program se poté ukončí. Výhoda tohoto režimu spočívá ve vyšší rychlosti simulace díky tomu, že není nutné vykreslovat herní svět.

7.1.3 Grafický režim

Grafický režim je plnohodnotný režim, ve kterém se program ovládá pomocí klávesnice i myši. Konfigurují se v něm experimenty, které lze načítat, ukládat, spustit, přerušit, krokovat, pozastavit i předčasně ukončit. Experiment lze předčasně ukončit a uložit výsledky pro jednotky, které cíle již dosáhly a statistiky z ostatních jednotek se kromě jejich počtu nijak nezapočítávají. Ovládání jednotlivých úkonů programu je v tabulce s klávesovými zkratkami A.1.

[1] Tento framework je k dispozici ke stažení na stránce: `https://www.microsoft.com/net/download/dotnet-framework-runtime`.

[2] MonoGame Framework verze 3.7.1.189, MonoGame Extended verze 1.1.0 OptimizedPriorityQueue verze 4.2.0

Příprava experimentu

Po spuštění programu je nejprve potřeba vybrat mapu. Mapami listujeme pomocí `Ctrl + (Shift) + Tab`, případně, pokud známe jméno mapy, lze ji načíst přímo pomocí menu dostupného přes klávesu `Enter`. Dalším krokem je volba konkrétního rozdělení mapy na oblasti. Rozděleními listujeme pomocí kláves `N` a `P`. Jednotky lze přidávat po jedné, nebo hromadně. Po jedné se přidávají prostředním tlačítkem myši. Hromadné přidání jednotek do obdélníku vytvořeného pohybem myši se provádí uvolněním levého tlačítka myši při současném držení klávesy `Ctrl`. Cíl se přidá pravým tlačítkem myši, jeho velikost je ovlivnitelná pohybem myši. Velikost cíle mění pouze vzdálenost, ve které jednotka cíle dosáhne. Tedy, pokud by cíl zasahoval na volná pole za zeď mimo oblast ve které se nachází, nebudou jednotky plánovat k tomuto přesahujícímu kusu cíle svou cestu. Stiskem klávesy `F` se takto připravená simulace uloží.

Spuštění uloženého experimentu

V grafickém režimu se připravený experiment otevře vstupem do menu klávesou `Enter`. V něm jsou experimenty roztříděny podle mapy, v níž se odehrávají. V další úrovni jsou experimenty pro danou mapu seřazeny dle svého ID. Zobrazeny jsou i další informace o počtu jednotek, vzdálenosti mezi startovní a cílovou pozicí nebo volitelná poznámka.

Menu primárně slouží k načtení experimentu, ale lze je použít i pro načtení pouhé mapy (bez připraveného experimentu) tak, že se víceúrovňové menu na první úrovni schová klávesou `H` nebo opustí opakovaným stisknutím klávesy `Esc`.

Po načtení připraveného experimentu zbývá vybrat typ jednotky. Ten se vybere stiskem jedné z kláves `A`, `S`, `D` nebo `W`. V tomto pořadí se jedná o metody plánování pomocí: A*, Toku v grafu z center oblastí, Toku v grafu složeném pouze z bran a Vektorového pole. Simulace se spustí klávesou `F5`, přerušit ji lze klávesou `Mezerník`, pro malé počty jednotek je vhodné ji zpomalit stiskem klávesy `F9`.

7.1.4 Překlad

Projekt překládáme standardním způsobem ve Visual Studiu. Nejprve je potřeba přeložit `MapDecomposition` a poté `SearchPathSimulator`. Je možné nastavit, zda chceme konzolovou nebo okenní aplikaci v závislosti na tom, zda potřebujeme vidět i textový výstup programu. Pokud přeložíme program jako okenní aplikaci, pak se při spuštění z příkazové řádky výstup programu do konzole nezobrazuje ani v příkazovém řádku `cmd`, ani v `powershell`. Ale ve Windows 10 s nainstalovanou součástí systému „`Windows Subsystem for Linux`" lze okenní aplikaci spustit z konzole `bash` i s textovým výstupem. Ve Windows 7 i 10 výstup rovněž funguje při použití `bash` z programových balíčků `Git`, `Cygwin` či `Babun`. V případě, že přeložíme aplikaci jako konzolovou, pak je tento výstup vidět ve všech zmíněných programech. Ale při spuštění samotného programu mimo konzoly dojde k otevření nového okna pro právě konzolový výstup.

7.1.5 Pokročilá nastavení

Pokud bychom chtěli v menu s experimenty vidět vlastní poznámky k jednotlivým experimentům, je třeba upravit text tagu `<Comment>` v souboru s experimentem (to je soubor se suffixem `.scenario.xml`).

Některá nastavení se nacházejí v souboru `app.config`, který je umístěn ve složce s programem. Může se však stát, že se vytvoří jiný soubor, který před ním při zpracování nastavení bude mít přednost. V operačním systému Windows by se nacházel uvnitř složky `AppData\Local`. Plná cesta ke složce simulátoru by byla `C:\Users\` jméno uživatele `\AppData\Local\SearchPathSimulator\`. V ní se pak nachází adresářová struktura, na jejímž konci je soubor `user.config`. Pokud by k tomu došlo, můžeme buď pracovat se souborem uvnitř této složky, nebo celou složku smazat. V souboru `app.config` se nachází perzistentní nastavení napříč běhy aplikace. Kromě `SimulationID`, které ovlivňuje názvy ukládaných simulací, obsahuje i `GridSize`. Tímto parametrem jsme schopni upravovat velikost, na jakou se přepočítává mapa. Jednotky se pohybují po této přepočtené mapě, takže rychlost a velikost jednotek je s ním úzce svázaná. Počet políček původní mapy se zvětšuje s druhou mocninou ($velikost\ mapy * \texttt{GridSize}^2$). Tento parametr je celé kladné číslo a standardně má velikost 1. Jeho zvětšení má negativní vliv na rychlost simulace a spotřebu paměti, zvláště pro velké mapy, jejichž velikost je omezena až na 4096×4096 políček. Další zajímavá nastavení jsou `StepsToReplanLost` umožnující nastavit po kolika krocích se ztracené jednotce povolí přeplánování, `MaxSimulationTimeAllowed` je maximální doba běhu programu, poté dojde k jeho ukončení.

7.2 Programátorská dokumentace

Hlavní třídou je třída `Simulator`, která dědí od Monogame třídy Game a obsahuje metody Draw a Update, které jsou periodicky volány. Metoda Draw vykresluje všechny objekty. Pokud je však program spuštěn pouze jako simulace, tak se jen provede `GraphicDevice.Clear()` a žádný objekt se nevykresluje. Metoda Update je MonoGame frameworkem opakovaně volána a každé její zavolání odpovídá kroku v simulaci. Během něj se přepočítá pozice všech jednotek. Frekvence opakování lze ovlivnit (zpomalit) zapnutím fixní délky kroku. Výpočet programu začíná v metodě Main v `Program.cs`, kde se zpracují vstupy a pak už se předá řízení Monogame Frameworku pomocí Simulation.Run().

Jednotky

Abstraktní třída AbstractUnit má potomky Target, Raycasting unit. Tyto třídy obsahují logiku pohybu jednotek a řízení pomocí steeringů. Jsou vyvářeny tovární metodou MakeUnit uvnitř UnitFactory. Při vzniku jednotky se reference na instanci jednotky ukládá do třídy NearbyUnitsPositions, která udržuje informace o pozici všech jednotek a slouží i k vyhledávání jednotek v okolí. Tato třída je alternativou za Quadtree a již jsme ji rozebírali v analýze 3.3.4. Hledání nejbližších jednotek probíhá pomocí proloženého většího gridu, jehož každá buňka obsahuje informace o jednotkách, které se v ní nacházejí. Pro vyhledání jednotek v okolí zvoleného bodu, čehož využívá například výpočet separation force, je

potřeba prohledat i sousední buňky gridu. Simulátor interně umožňuje libovolně zvětšovat mapu, tím se dosáhne zmenšení velikosti jednotek.

Mapa

Mapa je textový soubor s formátem definovaným v benchmarku [30], kde každému písmenu náleží typ buňky. Pro další zpracování nám stačí vědět, zda se jedná o průchozí či neprůchozí buňku. Pro účely vykreslení mapy je ovšem pomocí regionů v souboru `MapTile.cs` definována možnost nekomprimovat prostor buněk na dva typy – stěna, volné políčko –, ale nechat původní prostor nezměněný. Na binárním poli průchodnosti jednotlivých políček se nic nemění. Jediný viditelný rozdíl, který tato změna přinese, se týká vykreslování jednotlivých políček mapy.

Kolekci map sdružuje třída MapManager, která se inicializuje při startu programu. Při případném přidání souboru s mapou během běhu programu je pro její zobrazení nutné restartovat celý simulátor. Mapu zobrazujeme pomocí třídy background. Třída dostane na vstupu třídu Map obsahující již zpracovanou mapu v dvourozměrném poli TileEnum. Pomocí tohoto výčtového typu se rozhodne nejen barva políčka, ale i jeho průchodnost – tedy zda se jedná o zeď. Celou mapu rendrujeme do jedné textury, která má omezení na velikost 4096×4096 pixelů. Jedno políčko mapy vykreslujeme jako jeden pixel a při použití mapu zvětšujeme. Odtud plyne omezení pro velikost map, které umíme zpracovat. Pro vstupy z benchmarku je toto více než dostatečné. V opačném případě by bylo potřeba udělat úpravu například tak, aby se do textur rendrovala vždy pouze část mapy a tyto části by se podle potřeby skládaly dohromady.

7.2.1 Dekompozice mapy

Pro analýzu mapy využíváme knihovnu z našeho projektu `MapDecomposition`. Jedná se o kód doplňující diplomovou práci Kári Halldórssona [32], který jsme lehce upravili, aby vyhovoval našim potřebám. Knihovna je napsána v jazyce C++, voláme ji z Wrapper třídy ChokePointsAnalysis pomocí PInvoke. Tato knihovna analyzuje mapu a jako výstup vrací každému políčku číslo oblasti, do které spadá, a rovněž pozici souřadnice obou konců každé brány. Volání obsahuje tři důležité parametry int `distanceDenominator`, int `wallThreshold`, int `depthThreshold`, které dohromady ve zbytku sestavení skrývá proměnná `floodParameter`, která enumeruje vhodně zvolené parametry této trojce, ovlivňující kvalitu provedené dekompozice.

Simulace

Všechny potřebné funkce a pomocné objekty k uložení a znovu načtení simulace jsou ve třídě `Simulation`. Samotnou simulaci ukládáme za použití pomocného objektu, zachycujícího stav simulace pomocí serializace do xml souborů. Ukládáme pozici jednotek, cíle, o jakou jde mapu, velikost gridu na mapě, jaké hodnoty použít pro vstup pro výpočet pozice bran a několik dalších pomocných proměnných. Namísto binárního souboru používáme textový soubor, z toho důvodu, aby do něj bylo možné zasahovat, nebo aby se dokonce případně dal celý generovat.

Vstup od uživatele

Všechen vstup od uživatele je řešen ve třídách `MouseAction` nebo `KeyAction`. Všechny klávesové zkratky jsou definovány v druhé zmíněné.

Menu

Menu je naimplementováno pomocí tříd `MapManager` a `HelpMenu`. Help menu pomáhá vykreslovat logiku skrytou ve třídě držící kolekci map. Help menu má navíc další úlohu a tou je poskytovat nápovědu uživateli.

Switches

Statická třída `Switches` obsahuje pouze veřejné boolovské hodnoty, ovlivňující chování napříč programem, většinou se jedná o ladící výpisy.

Statistiky

Jednotky volají metody třídy `Counters` pokaždé, když nastane nějaká z událostí, které měříme. Ve třídě `Statistics` se tyto čítače po skončení simulace vyhodnotí, sumární statistiky z nich se zapíší do souboru. Kromě přímého vyhodnocení se zapisuje ještě jeden soubor, který zapíše všem jednotkám krok, ve kterém dorazily do cíle (tento výstup je setříděn od prvního příchodu do posledního).

Plánování

Ve třídách `AStar` a `FlowFieldSearch` je obsažena logika pro vyhledávání ve 2D gridu (jež je součástí třídy `Map`) a pro vyhledávání v síti (třída `FlowGraph`). Pro námi prezentovanou metodu výpočet plánování začíná ve třídě `Planner`, kde se z flow grafu vytvoří množina flow paths a ty se převedou na cestu reprezentovanou množinou bodů, které se již přímo přiřadí jednotkám. Hrany jsou ve flow grafu reprezentované neorientovaně, místo toho abychom měli pro každý směr jednu hranu, máme hranu pouze jednu a směr toku na ní reprezentuje znaménko. Pokud je kladné, teče tok ve směru od vrcholu s nižším ID do vrcholu s vyšším. Tento implementační detail je vidět, pokud v simulátoru zobrazíme graf po nalezení cest pomocí naší metody a zapneme i zobrazení extra informací ke grafu.

V algoritmu A^* používáme knihovnu `PriorityQueue` [33] pro zjištění nejbližší otevřené pozice.

Závěr

Cílem této práce bylo prozkoumat možnosti použití toků v sítích v real-time strategických hrách k navigaci větší skupiny jednotek, a to především v místech mapy obsahujících zúžení, která vedou k výraznému zpomalení přesunu skupiny. Našim úkolem bylo nalézt metodu, která by prostřednictvím toků v sítích umožnila naplánovat jednotkám takové cesty, aby se negativní dopady zúžení minimalizovaly.

Zkoumali jsme různá řešení, která se v počítačových hrách používají. Při analýze přidružených problémů jsme se detailněji zabývali oblastí toků v síti, abychom zjistili, která z metod toků v sítích bude při plánování v RTS hrách nejúspěšnější. Zavedli jsme požadavky na simulátor herního prostředí. Ten jsme v souladu se zadáním práce naimplementovali. Naprogramovali jsme dvě metody, jež využívají toků v síti a zhodnotili jejich úskalí. Spolu s dalšími dvěma metodami obecně používanými v RTS hrách jsme je testovali v našem simulátoru. Na desítkách map jsme provedli stovky experimentů ve třech scénářích. Náš simulátor nám umožnil porovnávat různé algoritmy hledání cest a jejich vykonání. Z evaluace provedených testů vyplývá, že naše řešení založené na tocích v sítích jednoznačně překonává svou kvalitou vybrané metody, které toky v sítích nevyužívají. To ilustrují i přiložené grafy.

Docházíme tedy k závěru, že toky v sítích lze k navigaci velkých skupin v real-time počítačových hrách bezesporu vhodně využít. Naše řešení přineslo výrazné zlepšení s ohledem na zrychlení průchodu jednotek.

Pro zrychlení vyhodnocování, aby metoda ještě lépe plnila požadavky real-time her, by bylo zajímavé prozkoumat možnosti zmenšení prohledávaného prostoru tak, aby se neohrozila možnost nalezení vhodných cest. Použitelnou strategií by mohlo být při prvním nalezení nejkratší cesty spočítat celkový počet kroků potřebný k poslání celé skupiny přes tuto cestu a také horní hranici délky cest, které ještě mohou přispět ke zkrácení celkového času. Omezení délky cesty by však bylo potřeba dělat vhodným způsobem, protože kvůli vzniku hran se zápornou cenou by mohlo dojít ke ztrátě vhodných cest ke křížení. Jednou z možností, jak by se to dalo provést, je v původním grafu pustit hledání do šířky s omezením jako má IDA* a označovat vrcholy, jež má smysl navštívit.

Naše metoda by s malými obměnami v optimalizační části mohla být dobře použita i pro hledání nedělitelného toku a ostatních toků omezujících jeho šířku. Bylo by jistě užitečné vyhodnotit i přínosy těchto strategií.

Seznam použité literatury

[1] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Journal canadien de mathématiques*, 8(0):399–404, jan 1956.

[2] James B. Orlin. Max flows in o(nm) time, or better. In *Proceedings of the 45th annual ACM symposium on Symposium on theory of computing - STOC '13*. ACM Press, 2013.

[3] L. R. Ford and D. R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6(3):419–433, 1958.

[4] Onur Özgün and Yaman Barlas. Discrete vs. continuous simulation: When does it matter?, 2009.

[5] Z. Király and P. Kovács. Efficient implementations of minimum-cost flow algorithms. *Acta Universitatis Sapientiae, Informatica, 4, 1 (2012) 67-118*, 2012.

[6] James B. Orlin. A faster strongly polynomial minimum cost flow algorithm, 1991.

[7] Kateřina Altmanová, Petr Kolman, and Jan Voborník. On polynomial-time combinatorial algorithms for maximum l-bounded flow. 2019.

[8] Georg Baier, Thomas Erlebach, Alexander Hall, Ekkehard Köhler, Petr Kolman, Ondřej Pangrác, Heiko Schilling, and Martin Skutella. Length-bounded cuts and flows. *ACM Trans. Algorithms*, 7(1):4:1–4:27, December 2010.

[9] Rainer E. Burkard, Karin Dlaska, and Bettina Klinz. The quickest flow problem. *ZOR Zeitschrift für Operations Research Methods and Models of Operations Research*, 37(1):31–58, feb 1993.

[10] Lisa Fleischer and Martin Skutella. Quickest flows over time. *SIAM Journal on Computing*, 36(6):1600–1630, jan 2007.

[11] Maokai Lin and Patrick Jaillet. On the quickest flow problem in dynamic networks – a parametric min-cost flow approach. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1343–1356. Society for Industrial and Applied Mathematics, dec 2014.

[12] Xiao Cui and Hao Shi. An overview of pathfinding in navigation mesh. *IJCSNS*, 12(12):48–51, 2012.

[13] Paul Tozour and IS Austin. Building a near-optimal navigation mesh. *AI game programming wisdom*, 1:298–304, 2002.

[14] Paul Tozour. Fixing pathfinding once and for all, 2008.

[15] Douglas Jon Demyen. Efficient triangulation-based pathfinding, 2007.

[16] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[17] Robert C. Holte. Common misconceptions concerning heuristic search. In *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010*, 2010.

[18] Robert C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical a *: Searching abstraction hierarchies efficiently. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, pages 530–535. AAAI Press, 1996.

[19] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of game development*, 1(1):7–28, 2004.

[20] Steve Rabin and Nathan R. Sturtevant. Combining bounding boxes and jps to prune grid pathfinding. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 746–752. AAAI Press, 2016.

[21] Sachin Patil, Jur van den Berg, Sean Curtis, Ming C Lin, and Dinesh Manocha. Directing crowd simulations using navigation fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):244–254, feb 2011.

[22] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[23] Craig Reynolds. Steering behaviors for autonomous characters, 1999.

[24] Jur van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *2008 IEEE International Conference on Robotics and Automation*. IEEE, may 2008.

[25] J. Snape, J. v. d. Berg, S. J. Guy, and D. Manocha. The hybrid reciprocal velocity obstacle. *IEEE Transactions on Robotics*, 27(4):696–706, Aug 2011.

[26] Steve Rabin Ben Sunshine-Hill. *Game AI Pro 3: Collected Wisdom of Game AI Professionals, Chapter 19: RVO and ORCA: How They Really Work*. CRC Press, 2017.

[27] Graham Pentheny. Advanced techniques for robust, efficient crowds. In *Game AI Pro 2*, pages 173–182. A K Peters/CRC Press, apr 2015.

[28] Clodéric Mars and Jérémy Chanut. *Game AI Pro 2: Collected Wisdom of Game AI Professionals, Chapter 20: Hierarchical Architecture for Group Navigation Behaviors*. Jenifer Niles, 04 2015.

[29] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[30] N. R. Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, June 2012.

[31] Kári Halldórsson and Yngvi Björnsson. Automated decomposition of game maps. In *AIIDE*, 2015.

[32] Kári Halldórsson. Using map decompostion to improve pathfinding. Master's thesis, School of Computer Science at Reykjavík University, 2015.

[33] BlueRaja. C# priority queue library. `https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp`.

A. Přílohy

A.1 Ovládání programu

Klávesy	**Funkce**
H / F1	Zobrazit nápovědu
?	Listovat v nápovědě
Enter	Otevřít menu / Načíst mapu nebo experiment
Escape	Opustit menu / Ukončit program po skončení simulace
Backspace	Opustit menu / Jít o úroveň výš v menu
←	Pohyb mapy / Opustit menu / Jít o úroveň výš v menu
→	Pohyb mapy / Načíst mapu nebo experiment (na 1. řádku)
↑	Předcházející položka v menu
↓	Následující položka v menu
F	Uložit simulaci (nutné definovat cíl pro jednotky)
R	Obnovit zoom a pohled
W	Použít algoritmus BFS
A	Použít algoritmus A*
S	Použít algoritmus FlowGraph1
D	Použít algoritmus FlowGraph2
+	Přiblížit pohled
−	Oddálit pohled
N	Následující dekompozice oblastí
P	Předcházející dekompozice oblastí
O	Přepnout typ grafu
1	Přepnout zobrazení cílů jednotky
2	Přepnout zobrazení čísla jednotky
3	Přepnout zobrazení raycast paprsků jednotky
4	Přepnout zobrazení barev jednotky
5	Přepnout zobrazení grafu toku
6	Přepnout zobrazení informací grafu (při zaplém grafu)
7	Přepnout vybranou jednotku a zobrazení jejího okolí
8	Přepnout zobrazení základních informací v rohu obrazovky
9	Přepnout zobrazení gridu
0	Přepnout zobrazení mapy
CapsLock	Odstranit jednotky
F5	Zahájit simulaci (mezerník ji přeruší)
F11	Provádět simulaci v jednotlivých krocích
Mezerník	Přehrávat simulaci po dobu stisku
F6	Zapnout dávkový režim bez vykreslení průběhu
F7	Přepnout vertikální synchronizaci obrazovky
F9	Přepnout fixní délku kroku simulace
Shift + Plus	Zvětšit velikost gridu
Shift + Minus	Zmenšit velikost gridu
Ctrl + Šipky	Zrychlit pohyb
Ctrl + Tab	Načíst další mapu

Kombinace kláves LTM znamená	**Funkce** Levé tlačítko myši
Ctrl+Tab	Přepnout mapy
Prostřední tlačítko myši	Vytvořit jednotku
Ctrl + LTM	Vytvořit jednotky v označené oblasti
Ctrl + Shift + LTM	Vytvořit 100 jednotek
Ctrl + Alt + LTM	Vytvořit 600 jednotek s povolenými kolizemi
Ctrl + Alt + Shift + LTM	Vytvořit 100 jednotek s povolenými kolizemi
LTM + označení jednotek	Přepočítat cestu k cíli pro všechny označené jednotky pomocí A*