VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

PLÁNOVÁNÍ CESTY V REÁLNÉM ČASE

REAL-TIME PATH PLANINNG

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Zdeněk Bartozel
AUTHOR

VEDOUCÍ PRÁCE — RNDr. Jiří Dvořák CSc.
SUPERVISOR

BRNO 2019

ABSTRAKT

Práce se zabývá plánováním cesty a pohybu holonomního robota v dynamickém prostředí. Cílem práce je implementace několika algoritmů založených na Rapidly-explored random tree algoritmu a jejich porovnání v navrženém dynamickém prostředí.

ABSTRACT

The thesis deals with the path planning and movement of the holonomic robot in a dynamic environment. The aim of this work is implementation of several algorithms based on Rapidly-explored random tree algorithm and their comparison in designed dynamic environment.

KLÍČOVÁ SLOVA

Realtime, plánování cesty, Rapidly-exploring random tree, RRT, holonomní plánování

KEYWORDS

Real-time, motion planing, Rapidly-exploring random tree, RRT, holonomic planing

BIBLIOGRAFICKÁ CITACE

BARTOZEL, Zdeněk. *Plánování cesty v reálném čase,* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky.

PODĚKOVÁNÍ

Za dokončení této práce a tím snad i tohoto studia, bych chtěl poděkovat své rodině, svým přátelům a spolužákům, kteří byli velkou oporou na této cestě. A také děkuji vedoucímu své práce, RNDr. Jiřímu Dvořákovi CSc., za cenné podněty k této práci.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, zpracoval jsem ji samostatně pod vedením RNDr. Jiřího Dvořáka CSc. a s použitím literatury uvedené v seznamu literatury.

V Brně dne 24. 5. 2019 ……

Zdeněk Bartozel

OBSAH

1 ÚVOD

Plánování pohybu je přítomné v každé aplikaci, kde jsou žádoucí různé úrovně autonomie. Dané systémy jsou podmíněny množinou diferenciálních omezení, počátečním a konečným stav, souborem překážek a cílovým regionem. Problém plánování pohybu je vlastně snahou o nalezení takového vstupu do systému, který řídící systém dovede z původního stavu, do stavu cílového. Typickým zástupcem těchto systémů, jsou autonomní roboty. Ty směřujícím ke stále kompaktnějším řešením.

A právě jednou z největších výzev dnešních autonomních systémů je zaujmout takovou strategii, která efektivně minimalizuje cenu zvolené cesty a zároveň i riziko srážky. Tento úkol je o to více aktuální v době, kdy se tyto systémy hojně nasazují do reálného prostředí. A zvyšují se tedy nároky kladené na jejich efektivitu a bezpečnost.

Jako příklady můžeme uvést autonomní obslužné systémy ve skladech, obslužné létající roboty a samozřejmě autonomní vozidla. Z příkladů je patrno, že tyto systémy jsou dnes nasazovány v každém typu prostředí. Tedy i prostředí, ve kterém snadno přichází do kontaktu s lidmi, zvířaty a jinými roboty. Proto jsou také požadována stále rychlejší a výpočetně efektivnější softwarová řešení, která robotu umožňují navigaci v tomto prostředí.

Tato práce si bere za cíl představení velmi rozšířeného a využívaného algoritmu používaného k plánování cesty Rapidly-exploring Random Tree, dále také jako RRT a jeho variant. A jejich simulaci v dynamickém prostředí. Protože ty se dnes jeví jako velmi univerzálním a vhodným nástrojem pro reálné aplikace.

1.1 Kontext a motivace

I když se může zdát, že z pohledu dnešních softwarových řešení a hardwarových možností, se problém hledání cesty jeví jako uspokojivě popsaný a vyřešený. Žádný z dosud navržených přístupů neposkytuje dostatečně univerzální odpověď, pro různé druhy prostředí a typy robotů. Dnešní velmi rozšířená řešení jsou naopak většinou dobře aplikovatelná na konkrétní problémy daného prostředí a robota, ve kterém poskytují uspokojivé výsledky [1].

Což znamená, že při výpočetních možnostech daného robota, algoritmus nalezne optimální, nebo alespoň uspokojivé suboptimální řešení, a to ve vyhovujícím časovém intervalu. Avšak při změně parametrů hledání, mohou i řešení využívající nejmodernějších algoritmů trvat desítky sekund. Například algoritmus pracující velmi dobře v nízko dimenzionálním konfiguračním prostředí je často mnohonásobně pomalejší v prostředí s více dimenzemi. Nebo když má při plánování pohybu robota zahrnout i kinodynamická omezení [2], která každý reálný robot má.

Proto také rozdělujeme hledání cesty podle omezení robota, tedy na holonomního a neholonomního robota [3]. Jako holonomní je systém označován, pokud všechny jeho

vazby jsou holonomní, a omezení závisí pouze na souřadnicích systému a času. Tedy omezení nezávisí na rychlosti nebo hybnosti systému.

Holonomnost je tedy v robotice vyjadřována jako poměr řiditelných stupňů volnosti vůči jejich celkovému počtu. Pokud je tento poměr jedna ku jedné, pak je robot holonomní. Naopak pokud je počet řiditelných stupňů volnosti menší než celkový počet stupňů volnosti, pak je robot neholonomní. Jinými slovy, robot je neholonomní když, jsou-li na něj kladena diferenciální omezení, která nejsou plně integrovatelná

Plánování cesty stále velmi aktivně zkoumanou disciplínou, a to i když se s výzkumem začalo už v počátcích zkoumání konceptu umělé inteligence. Jeho význam v posledních desetiletích stále roste. Důvodem je značný zájem o autonomní robotické systémy, a tudíž i jejich nárůst. Velké využití tyto systémy nachází v armádním [4][5], nebo průmyslovém prostředí. A v posledních dekádách se dostávají i do běžného civilního života.

Charakteristickým rysem armádních aplikací je snaha minimalizovat lidskou přítomnost v situacích bezprostředně ohrožujících lidský život, nebo v situacích, které překonávají lidské možnosti. Příkladem toho je nepřetržitý letecký monitoring zvolené oblasti. Nebo stále aktivněji vyvíjené autonomní pozemní jednotky. V průmyslové oblasti si tyto systémy kladou za cíl snížit potřebu lidské práce a tím i její náklady. To zároveň vede k snížení ceny výrobku a optimálnímu využití lidského času a kreativního potenciálu jednotlivců. Nahrazovány jsou především méně náročné a méně kvalifikované pozice s velkou mírou stereotypu. A v civilním prostředí je snaha o nahrazení monotónních, ale stále nezbytných úloh, jako vysávání, sekání trávy a mnohé jiné. Z toho je zřejmé, že většina právě probíhajícího výzkumu se zaměřuje na vývoj autonomních systémů, které stále méně vyžadují přímý zásah člověka.

K dispozici je mnoho přístupů k řešení problému [6], avšak není zde žádný algoritmus splňující všechny požadavky na plánování cesty. Jako nízkou výpočetní náročnost, spolehlivost, robustnost, bezpečnost, hladkost cesty, její optimalitu a jiné.

Příkladem jsou algoritmy, které jsou velmi rychlé, tedy výpočetně méně náročné [2], ale zpravidla neposkytují dostatečně hladkou cestu, která je většinou požadována. Ta je v takovém případě výsledkem doprovodných vyhlazujících algoritmů, které ale ovlivňují výpočetní čas algoritmu. Na druhou stranu algoritmy jejichž řešením je hladká cesta [6] a jsou zároveň rychlé a efektivní, tak jejich využití se omezuje na použití ve dvoj, nebo troj rozměrném konfiguračním prostoru a jen stěží jsou aplikovatelná na vícerozměrné typy úloh. Konfigurační prostor narozdíl od pracovního, který může být maximálně tří rozměrný, může být i více rozměrný. Tento prostor se dá popsat jako množina všech možných konfigurací robota, která je dána počtem stupňů volnosti robota.

Protože mobilní platformy často nemají neomezený výpočetní výkon, a naopak často mají velmi omezenou životnost baterie. Je velká snaha vyvíjet stále efektivnější přístupy. Právě tak, aby co možná nejefektivněji pracovaly s technickými možnostmi robotických platforem.

2 PLÁNOVÁNÍ CESTY

V této kapitole je formalizován problém hledání cesty, jako nezbytný podklad pro zbylou práci. Jsou zavedeny a představeny základní pojmy a myšlenky, které se vyskytují skrze celou práci.

2.1 Plánovací prostředí

Plánování cesty zahrnuje získání dat z předkládaného prostředí, a na jejich základech následně výpočet trajektorie cesty, která vede z počátečního bodu do bodu konečného. A to v závislosti na předem stanovených kritériích a parametrech daného přístupu.

Pro zohlednění těchto kritérií musíme zavést ohodnocovací funkci. Například může zohledňovat délku cesty, dobu potřebnou k jejímu zvládnutí, nutnost vyhnout se dynamickým překážkám, bezpečnost anebo mnohá další kritéria.

V reálných aplikacích je následným krokem plánování pohybu. To zahrnuje již naplánovanou trajektorii a na jejímž základě dochází k výpočtu pohybu agenta z počátečního bodu do bodů následujících.

Plánování cesty lze velmi dobře rozdělit podle prostředí ve kterém se má agent pohybovat. Nejzákladnějším rozdělením je rozdělení na prostředí statické a dynamické.

Problémy plánování cesty agenta ve známém statickém prostředí, jsou v dnešní době považovány za dostatečně vyřešené [8]. Navigační systémy jsou v takovém prostředí ucelené a optimalizované. To znamená, že jsou schopné naplánovat průchozí cestu, tak abychom dosáhli cíle. A jsou schopné, pro danou metriku nalézt cestu nejkratší. Pro takové prostředí zavádíme pojem „Known space assumption“ (KSA) [9] a předpokládáme, že celé prostředí, ve kterém se robot vyskytuje je známé, a že robot má k dispozici mapu, která byla vygenerována již předem. To znamená, že prostředí neobsahuje chyby, které vznikají v reálných aplikacích špatným namapováním reálného prostředí.

Z pohledu míry nejistoty v prostředí, je opakem takto známého prostředí, prostředí dynamické. To je pro plánování pohybu a cesty agenta velkou výzvu. Protože částečně zmapované prostředí a dynamické překážky, u nichž se v čase může měnit tvar, poloha a orientace, jsou velmi náročné jak výpočetně, tak paměťově.

Dynamická prostředí mohou být dále rozdělena podle toho, jak predikovatelné jsou jejich změny v čase. Pokud máme perfektní znalost o dráze a dynamice pohyblivých překážek, pak říkáme, že takové prostředí je deterministické. Nedeterministická prostředí jsou taková, která sebou nesou aspoň nějakou míru nejistoty.

Hledání cesty v takovém prostředí znamená zaručit, že nalezená cesta je v daný časový interval průchozí, bezpečná, a že v žádné pozici na této cestě nehrozí agentu kolize.

Pro tento typ plánování jsou především vhodné realtime, nebo anytime strategie plánování. Tyto přístupy jsou blíže uvedeny později. Pro toto prostředí je vhodné zavést následující dva pojmy. V této práci se bude agent pohybovat ve známém prostředí s dynamickými překážkami. Což znamená, že mapa, jež má agent k dispozici neobsahuje chyby mapování, které se u reálných aplikací vyskytují [10]. A že cíl, je vždy dosažitelný.

2.2 Prostředí znázorněná grafem

Jeden z možných přístupů, jak řešit problém hledání cesty, je reprezentovat prostředí ve formě grafu. Nalezení cesty se potom dá zobecnit na úkol hledání nejkratší cesty v grafu. Takový graf $G(V, E)$, nazývaný také jako mapa cesty (roadmap), kde množina uzlů je označovaná jako $V = \{v_1, v_2, v_3, ...\}$ a množina hran jako $E = \{e_1, e_2, e_3, ...\}$ a hrana je definována jako $e = \{v_a, v_b\}$.

Platí, že hrana mezi vrcholy vzniká v případě, když takové propojení zajišťuje průchozí cestu. Při hledání optimální cesty musí být hrana ještě ohodnocena dle dané metriky.

Jednou ze stěžejních otázek těchto grafových přístupů je otázka adekvátního pokrytí. Tedy jak moc je potřeba dané prostředí fragmentovat, tak abychom maximalizovali šanci na nalezení optimálního řešení a zároveň minimalizovali paměťové a výpočetní požadavky [11].

Protože se tato práce zabývá metodami, které k uchování informací využívají grafové struktury, zmíníme zde v krátkosti některé datové struktury, které jsou vhodné pro tento typ úloh.

Vhodné zvolení datové struktury je pro všechny typ úloh, které mají charakter prohledávání grafu, naprosto stěžejní. Protože má přímý vliv na časovou náročnost algoritmu. Časová náročnost, jak je uvedeno dále, je pro realtimové algoritmy jedna ze stěžejních otázek. Se špatně navrženou datovou strukturou může doba zpracování neúměrně narůst.

Některé datové struktury jsou více vhodné pro prohledávání předem daných prostředí, například struktura KD-tree, která neumožňuje postupné vkládání do stromu. Jiné se lépe hodí pro čtvercově, nebo hexagonálně rozdělené mapy B-tree.

Jak je podrobněji uvedeno níže, zkoumané algoritmy patří do skupiny „Sampling based“ algoritmů. Jako každá skupina algoritmů, tak i tato má svá specifika. Prvním je, že jednotlivé vrcholy stromu nemají pevný počet sousedících vrcholů. Počet je určován pouze zvolenou hustotou sítě. Například pokud by byla zadána mapa s čtvercovou mřížkou a algoritmus by pracoval s touto mapou, pak by měl každý vrchol jasně daný maximální počet vrcholů. Tedy čtyři, popřípadě osm.

Druhou vhodnou vlastností je, zda daná datová struktura dokáže vracet prvky ze stanoveného okolí bodu. S touto vlastností pracuje algoritmus RRT velmi často.

Pro takový typ úloh, jsou vhodné datové struktury označované jako „prostorové indexy založené na mřížce“ (grid-based spatial indices) [12]. Jejich typickými zástupci

jsou struktury jako KD-tree [13], nebo R-tree [14]. V této práci je požita struktura R-tree, která nabízí velmi snadnou testovací implementaci a velmi vysokou efektivitu, pro tento typ úloh. Protože body ve struktuře jsou uspořádány tak, že je velmi rychle zajištěno prohledávání v okolí bodu.

2.3 Formulace problému hledání cesty

Formulace problému hledání cesty je velmi jasně popsána v literatuře. Jako základní formulaci problému můžeme uvést například "The piano movers problem" [15]. Vzhledem ke geometrii vozidla **A** a překážkám **B**, je plánování pohybu agenta problémem nalezení cesty $\boldsymbol{T(s)}$, v prostoru $\boldsymbol{X}$, z počáteční polohy $\boldsymbol{T(0)} = \boldsymbol{x}_{init}$, až po konečnou polohu $\boldsymbol{T(1)} = \boldsymbol{X}_{goal.}$ A to tak, že zvolená konfigurace **A** a **B** na cestě $\boldsymbol{T(s)}$ není v kolizním stavu. $\boldsymbol{X}_{goal}$ označuje množinu bodů v nejbližším okolí cílového bodu, při jejímž dosažení se uvažuje, že robot dosáhl cílového bodu.

Při formalizaci problému plánování cesty využijeme definici, která byla představena v práci Karamana S. a Frazolli E. [7], protože tato přelomová práce je předlohou naprosté většiny nově uváděných modifikací RRT. Na plánování cesty se díváme, jako na prohledávání x-rozměrného metrického prostoru $\boldsymbol{X} = (0, 1)^d$, kde $d \in \mathbb{N}$, $d \geq 2$. Prostředí $\boldsymbol{X}$ je nejběžněji reprezentováno jako trojrozměrný euklidovský prostor $\mathbb{R}^3$.

V prostoru $\boldsymbol{X}$ zavedeme množinou bodů $\boldsymbol{X}_{\text{obs}}$. A předpokládáme, že $\boldsymbol{X}_{\text{obs}} \in \boldsymbol{X}$ a je použita k reprezentaci všech nedosažitelných bodů prostředí $\boldsymbol{X}$. Tudíž její pozice nemohou být použity k plánování. Také předpokládáme že množina $\boldsymbol{X}_{\text{obs}}$ není předem k dispozici. Potom platí, že $\boldsymbol{X}_{\text{free}} = \boldsymbol{X}/\boldsymbol{X}_{\text{obs}}$, kde $\boldsymbol{X}_{\text{free}}$ je množina bodů použitelná pro plánování. Z toho plyne, že počáteční podmínkou hledání je, že $\boldsymbol{x_{init}}, \boldsymbol{X_{goal}} \in \boldsymbol{X_{free}}$. Problém hledání cesty je potom takový, že hledáme cestu, která vede k cíli. Pokud taková neexistuje, algoritmus musí reportovat selhání.

2.3.1 Hledání optimální cesty

V takto definovaném problému hledání cesty však nejsou nijak zohledněny preference hledání. Jinými slovy není zavedena žádná metrika, podle které bychom byli schopni určit optimalitu výsledného řešení. Z tohoto důvodu zavádíme ohodnocovací funkci. Ta má za úkol každou nalezenou průchozí hranu kladně ohodnotit, tak abychom při průchodu grafem mohli najít optimální řešení. Optimální cestou ze všech nalezených je taková, která ze sumy cest vedoucích k cíli má nejnižší cenu. Nabízí se například ohodnocení podle délky cesty, času, nebo energie potřebné k jejímu zdolání.

Nejčastějším je ohodnocení podle vzdálenosti mezi vrcholy grafu, to vede k nalezení nejkratší vzdálenosti z bodu $\mathbf{x}_{\text{init}}$ do $\boldsymbol{X}_{\text{goal}}$.

K hledání nejkratší cesty v grafu se využívá Dijkstrův algoritmus [16], popřípadě A* [17] a jeho modifikace.

2.4 Přístupy ke zpracování časově závislých algoritmů

Pro řešení časově závislých úkolů, tedy úkolů jejichž správné vyřešení je podmíněno časovým limitem. Jsou navrženy dva frameworky, které se s takovými úkoly vypořádávají. Prvním přístupem je anytime a druhým je realtime. V této podkapitole se pokusíme tyto přístupy objasnit a vymezit mezi nimi rozdíl.

2.4.1 Anytime algoritmy

Než bude popsána problematika realtimových algoritmů, je nutné v krátkosti shrnout i jiný přístup, který je také využitelný k plánování cesty a pohybu agenta v reálném prostředí. Právě proto, že přístup k mezivýsledkům a časovým závislostem pod úkolů je v obou případech rozdílný.

Anytime přístup je vhodné aplikovat v případech, kdy je řešení problému závislé na množství času potřebného k nalezení řešení. Takový problém je následně označen jako časově závislý (time dependent). Právě na řešení takových problémů byl Anytime framework navržen [18].

Realtimové algoritmy mají pevně dané časové intervaly pro jednotlivé části celého algoritmu. V těchto intervalech je od každého pod úkolu požadován výstup. Oproti tomu, u Anytime algoritmů je možné požadovat mezivýsledky v předem nespecifikovaných časových okamžicích. To znamená, že algoritmus může být kdykoli přerušen a výsledky které jsou do té doby ohodnoceny nejlépe, jsou předány jako výsledné. Po odevzdání výsledků stále probíhá optimalizace cesty. Cesta, která je aktuálně požívána je pravidelně nahrazována lepším řešením, pokud takové existuje.

Z popisu lze odvodit obecný přístupe v konstrukci anytime algoritmu. Na začátku je vygenerováno počáteční řešení, které je pravděpodobně vysoce neoptimální. A následně je snaha nacházet jiná, další řešení s více optimálním charakterem.

Kdykoli během tohoto generování může být algoritmus vyzván k navrácení nejlepšího nalezeného řešení. Vazba mezi užitečností a časovými nároky je nazývána jako “performance” profil. Anytime algoritmy se snaží zabezpečit, že užitečnost výsledku s časem roste. Pokud je náročnost úkolu taková, že algoritmus selhává při nalezení počátečního řešení, jeho performance profil může být neakceptovatelný.

Existuje celá řada implementací dynamických anytime algoritmů, jako například Anytime-D* [19], Anytime-DRRT [20], Anytime-PRM [21].

2.4.2 Realtimové systémy

Zjednodušeně řečeno, realtimový systém je takový, v němž výsledek operace závisí nejen na dosažení správných výsledků, ale také na časovém limitu, ve kterém je potřeba těchto výsledků dosáhnout [22].

Realtimové systémy musí garantovat časovou odezvu na sledovanou událost, a to před uplynutím předem stanoveného časového limitu, často také nazývaného jako deadline. Konkrétněji lze říci, že plánovací realtimové algoritmy jsou spojené s omezeným množstvím času mezi akcemi [23], jak je také ukázáno v této kapitole.

Realtimové systémy mohou být kategorizovaný podle následků, které pomalé vyhodnocení systému způsobí.

- Hard realtime systémy – jsou takové, ve kterých vyhodnocení po daném deadlinu může vést až ke katastrofě. Takové systémy se objevují v autech, letadlech, medicínských aplikacích a v případě řízení energetických zařízení. Je jasné, že pozdní vyhodnocení u uvedených aplikací je fatální a může vést až ke smrti.
- Firm real-time systémy – akceptují i pozdní splnění úkolu, ale s danou informací pracují tak, že takový výsledek nemusí být stále validní.
- Soft realtime systémy – dokáží tolerovat výsledky s časovým zpožděním, ale to se také odráží na výkonu systému.

V případech, kdy je plánování pohybu považováno za úkon ovlivňující bezpečnost, nebo za kritickou součást softwaru, je k němu přistupováno jako k hard-realtimovému. Toto plánování je pak zatíženo časovými požadavky na nalezení bezpečné a průchozí cesty.

Podobný přístup ale selhává v prostředí, kde deadline nemůže být pevně daný, nebo je jen velmi těžko odhadnutelný. To znamená, že na počátku cesty nejsou známy výpočetní nároky pro danou situaci. Takové úlohy se nazývají jako nepřesné (imprecise).

Snahou při tomto typu plánování je, aby systém byl v první řadě naplánovatelný a měl dobře rozdělené využití výpočetních zdrojů. V reálných aplikacích se využívá prioritního přístupu, kdy úkoly s vyšší prioritou mají ve zpracování přednost. Tento přístup samozřejmě může vést k interferenci mezi jednotlivými úkoly. U Hard-realtimové aplikace se využívá prioritního přístupu nazývaného “Rate Monotonic scheduling algorithm” [24]. Ten zaručuje, že pokud akce zmešká svůj deadline, jsou ovlivněny pouze úkoly s nižší prioritou.

2.4.3 Časová náročnost algoritmu

Podstatným úkolem při navrhování plánovacího algoritmu je porozumět jednotlivostem, které ovlivňují časovou náročnost algoritmu. Znalost časových intervalů potřebných k vykonání jednotlivých pod úloh celého algoritmu, je stěžejní, pro základní rozhodnutí, zda požadovaný úkol je za stávajících podmínek řešitelný. Časová náročnost se běžně vyjadřuje O-notací. Algoritmy s lineární, nebo kvadratickou náročností jsou označovány takto O(1), O(N), O(N^2), kde N je velikost množiny vstupů.

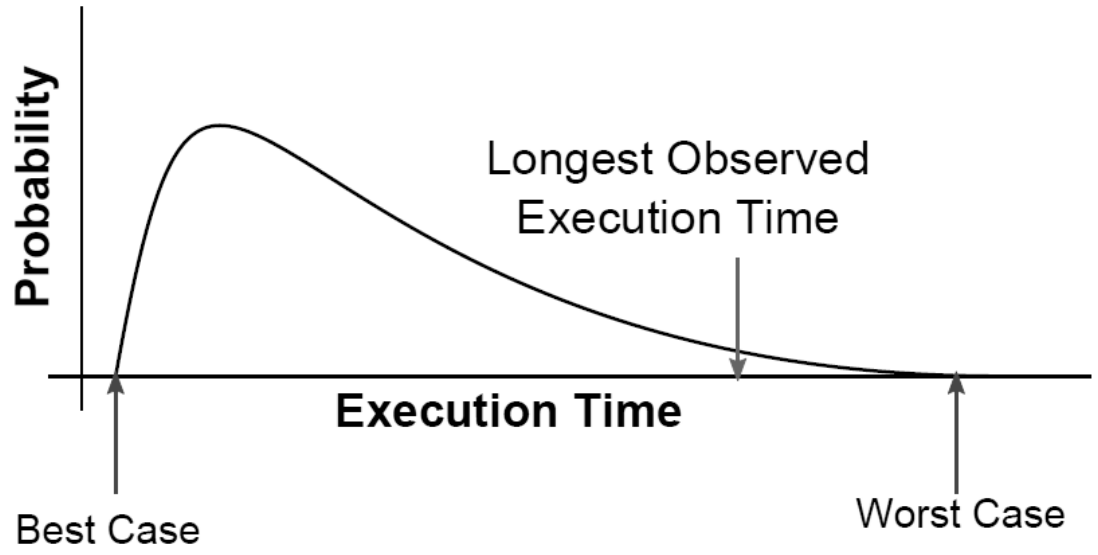
Časová náročnost plánovacího algoritmu v dynamickém prostředí je vázána na množství překážek, které agent musí zahrnout do plánování a na velikosti zkoumaného prostoru.

Pro některé části algoritmu je deadline navržen zvlášť. Toto je dobře viditelné v kapitole zabývající se realtimovými implementacemi RT-RRT. Výsledná suma těchto dílčích časových intervalů pomáhá k určení celkového trvání jednoho běhu algoritmu. To je doba od spuštění algoritmu, až po okamžik, kdy jsou k dispozici výsledky cyklu.

S časovým intervalem potřebným k dokončení úkolu je spojen velmi důležitý pojem "nejhorší možný čas potřebný ke zpracování úkolu" (WCET, worst case execution time). Ten je u reálných aplikací nutné důkladně odvodit, nebo změřit.

Jedním z nejtěžších úkolů realtimových systémů, je právě správné nastavení a odhad doby potřebné na zpracování úkolu. Podcenění exekučního času může vést ke kritické chybě systému a příliš velký časový interval je také nevyhovující. Protože špatné přidělování výpočetního času, může mít za následek neschopnost systému odpovídat na rychlé vnější podněty.

Výsledný interval je přímo úměrný výpočetní síle daného hardwaru a softwarové implementaci. Modelační metody, které mají za úkol odhadnou WCET, by měly v ideálním případě odhadnout čas o něco horší, než je jeho reálná hodnota.



Obr. 1: Pravděpodobnostní křivka distribuce exekučního času [1]

Na Obr. je zachycena pravděpodobnostní křivky, která ilustruje časový průběh sledovaného algoritmu se zadanými vstupy pro x opakování. Toto je nejpřímější postup, jak časovou náročnost algoritmu odhadnout, ale v případech netriviálních softwarových úloh je v podstatě nepoužitelný.

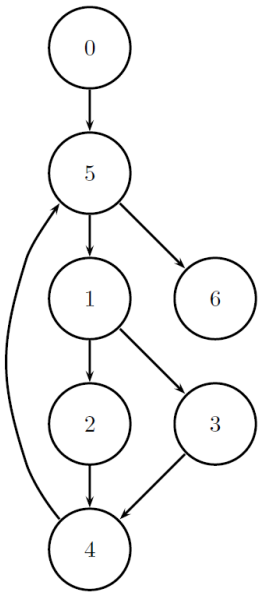
Jiným přístupem je rozdělení měřeného algoritmu na logické části. Výsledný interval pak odpovídá sumě všech naměřených časových intervalů. Jinými slovy testování pak probíhá pro každý logický celek samostatně. Následně je čas exekuce WCET odhadnut jako suma maximálních pozorovaných časů. Tento přístup již v minulosti byl použit pro odhadnutí časů v realtimových algoritmech [25].

Rozdělení metod pro měření doby trvání WCET netriviálních úloh je následující.

- Statická analýza, ta analyzuje softwarový kód a vytváří předpoklad na základě toho, jak metody tohoto modelu komunikují s hardwarem. U vyšších programovacích jazyků, je možné odhadnout WCET pomocí časových schémat, které berou v úvahu časovou náročnost jednotlivých částí kódu, jako například složené výrazy, iterační smyčky, podmínky a jiné.
- Metody založené na měření, rekonstruují tok programu a asociují exekuční čas s reálnými výstupy modelu. Byly navrženy metody [26] pro odhad WCET viz. obr.2, tak že exekuční čas je modelován jako pravděpodobnostní funkce, díky které jsme schopni odhadnout hraniční hodnoty WCET. Pro komplexní úlohy bylo navrženo rozšíření tohoto přístupu, které umožňuje sestavení podobné funkce i pro komplexní úlohy [27].
- Hybridní metoda, kombinující oba zmíněné přístupy.

Čím více zkušebních cyklů je provedeno, tím vyšší je pravděpodobnost nalezení WCET. Obecně se dá říct, že tento přístup vede k nalezení pouze hodnoty blízké skutečnému WCET.

Jednou z metod, jak modelovat exekuční čas softwaru je Control Flow Graph [28], kde cesta skrze graf reprezentuje možnou sekvenci instrukcí, které mohou být vykonány. CFG je direct graf *G(V, E)*, kde uzly *V* reprezentují jednotlivé instrukce a hrany *E* reprezentují tok mezi uzly.



Obr. 2: Asociace v CFG [1]

Pokud je výpočetní model příliš jednoduchý, pak nebere v potaz detailní hardwarovou platformu. Naopak sestavení komplexní modelu je časově náročné a nevýhodné při častých změnách řešení.

2.4.4 Bezpečná navigace v realtimovém prostředí

Bezpečná navigace představuje způsob řízení, který plánuje trajektorii agenta, nebo systému tak, aby bránil kolizím s překážkami. V případě autonomního robota tyto překážky mohou být buď statického, nebo dynamického charakteru.

Realistické prostředí vždy obsahuje i překážky dynamického charakteru. Kvůli nim je nutné v pravidelných časových intervalech zkontrolovat průchodnost cesty a případně ji přeplánovat. Tento přístup se nazývá dynamické plánování pohybu robota (dynamic motion planning).

Na plánování pohybu robota ve statickém prostředí se dá dívat, jako na plánování v otevřené smyčce. Protože celý proces plánování cesty i pohybu proběhne jen jednou. V takovém prostředí totiž nejsou předpokládány žádné změny v době pohybu robota. Opakem toho je právě plánování dynamické, které probíhá v uzavřené smyčce. Tedy probíhá opakovaně a výsledná cesta je vždy navržena na základě aktuálního stavu prostředí. Na jednu iteraci můžeme nahlížet jako na snímání, plánování a řízení.

Snímání zahrnuje jak mapování zkoumaného prostředí, tak robotovu lokalizaci (Simultaneous Localisation and Mapping (SLAM)) [29]. Mapování prostředí je proces, při kterém dochází k virtualizaci procházeného prostředí, a to na základě senzorických měření. Lokalizace je proces hledání a určování polohy a pozice robota.

Plánování cesty následně využívá výsledek mapování a lokalizace a vytváří bezpečnou, bezkolizní dráhu, která je vypočtena na základě aktuální polohy překážek. Během plánování nemohou být do procesu plánování zahrnuty nové informace.

Schopnost cestu bezpečně naplánovat závisí na kvalitě snímání, plánování, řízení a schopností těchto procesu interagovat. V reálných aplikacích interakce mezi těmito složkami vždy zahrnuje i časové zpoždění.

Jak bylo uvedeno výše, v realtimových aplikacích tyto části cyklu pracují v rámci časových intervalů, za který musí být schopny vytvořit požadované výsledky.

3 PŘEHLED SOUČASNÉHO STAVU POZNÁNÍ

V této kapitole jsou uvedeny některé z algoritmů, které měly na vývoj plánovacích algoritmů největší vliv. Kapitola je rozdělena podle typu plánování a skupin algoritmů.

3.1 Reaktivní plánování

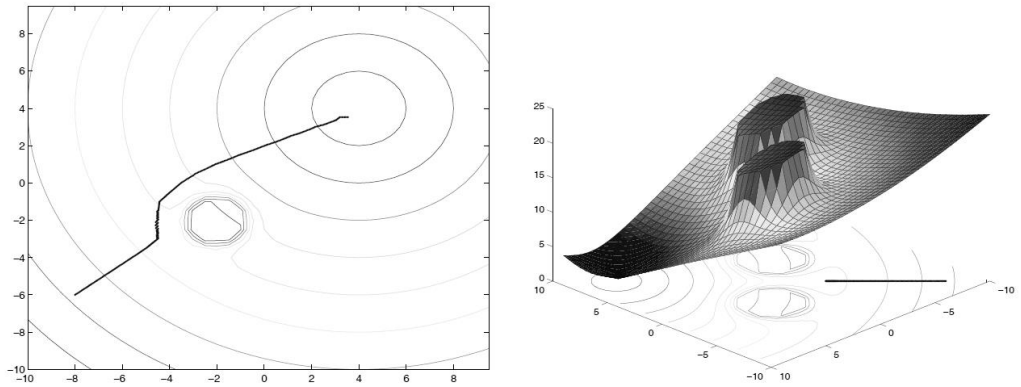
Plánování cesty a pohybu agenta v reálném prostředí je výpočetně velmi náročné. V době, kdy tato výpočetní kapacita nebyla jednoduše k dispozici byl navržen přístup, s ohledem na to, aby agent mohl pracovat i v dynamickém prostředí. Ten se nazývá reaktivní plánování a pracuje tak, že plánování cesty a mapování prostoru se omezuje pouze na agentovo blízké okolí.

Tento přístup agentovi umožňoval rychle měnit zvolenou trasu robota. To proto, že snímače na těle robota byly schopny zaznamenat směr potenciálního nebezpečí. Po vyhodnocení byl předán potřebný pokyn přímo aktuátoru, který reagoval tak aby nedošlo ke kolizi. Jednalo se o prioritní přístup, kde celé řízení agenta bylo rozděleno do několika kategorií. Podle jejich vlivu na bezpečnost a funkci jim byla přidělena priorita. Vedlejším efektem takového přístupu bylo, že řízení nebylo kompletní a robot mohl cíl minout.

3.1.1 **Metoda umělého potenciálového pole** (Artificial potential fields methods) [30]

Algoritmy pracují nad souřadnicově znázorněným prostředím a řeší problém plánování, jako analogii pohybu částice v elektromagnetickém poli. Robot je zobrazen jako elektricky nabitý bod, cíl je naopak znázorněn jako bod s opačným potenciálem. Takže robot je k cíli přitahován. Překážky jsou modelovány jako stejně nabité body, a ty se tudíž odpuzují. Metoda je v konceptu jednoduchá a snadno implementovatelná, hlavním problémem je uvíznutí v lokálním minimu a oscilace kolem něj.

Nejvýznamnější modifikací tohoto algoritmu je Randomised potential fields [31]. Ten v případě, že detekuje uvíznutí v lokálním minimu provede sérii náhodných kroků, která může vést k uvolnění agenta. Kompletnost takového řešení je, ale stále odkázána na délku náhodného kroku.



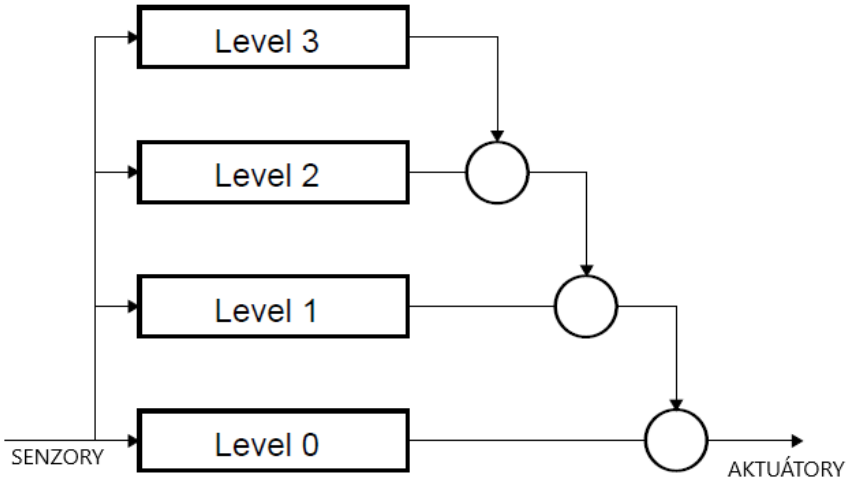
Obr. 3: Potential field planning. Na obrázcích je vizualizováno potenciálové pole s vozidlem sledujícím cestu pomocí obrysového grafu. První obrázek znázorňuje cestu k cíli. Na druhém je znázorněno uvíznutí v místním minimum, které brání v dosažení cíle. [1]

3.1.2 Subsumption

Je jedním z prvních přístupů dovolujících mobilním robotickým platformám plně fungovat v dynamickém prostředí. Byl inspirován biologickými systémy, které rovněž reagují na své nejbližší okolí [32].

Tato strategie je založena na rozdělení komplexního problému řízení robota do řady jednodušších, navzájem nezávislých úkolů. Jednotlivé vrstvy zapojené do řízení robota, např. detekce kolize, plánování cesty, zkoumání prostoru a jiné, jsou prioritně ohodnoceny. Spouštěny jsou konkurenčně, a je upřednostněna taková funkce jež má vyšší prioritu. Typicky jsou takové funkce založeny na výsledcích generovaných ze senzorů.

Tento návrh robotického systému je pokusem postavit robota z nejnižších vrstev, teda od senzoriky a postupovat po vrstvách nahoru k sofistikovanějším způsobům řízení. Systém implementuje schopnost reakce na neznámé vnější podněty, založené na přímé zpětné vazbě mezi senzory. Narozdíl od pokusu modelovat robotovo nejbližší okolí a možné interakce s ním.



Obr. 4: Příklad znázornění základní architektury subsumption řešení. Je vidět, že každá úroveň má přístup k datum z aktuátorů a má možnost ovlivňovat aktuátory. Kruhové body znázorňují, že část s vyšší prioritou může potlačit vliv částí podřízených.

Největší výhodou systému, je pokus o rozhodovací paralelismus a možnost iterativního nastavování systému na základě výsledků zkoušek v cílovém prostředí. Limitací této architektury je na druhou stranu její následné škálování. Výpočetní náročnost pro sestavení bezpečné a bezkolizní strategie roste proporcionálně s množstvím funkcí, která je robot schopen provádět [33].

3.2 Deliberativní plánování

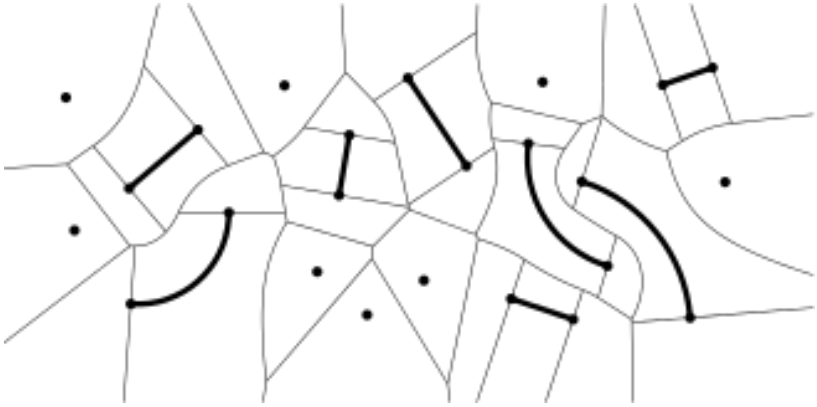
Deliberativní plánování je přístup, který se soustředí na plánování cesty z počátečního až k jejímu konečnému bodu. Což je přesným opakem reaktivního plánování, které se soustředí na kontrolu a přizpůsobení se svému nejbližšímu okolí. Typicky, deliberativní plánování generuje graf, nebo strom, ve kterém popisuje strukturu problému. Tento přístup je v dnešních systémech používány nejčastěji. Ve chvíli, kdy je stromová, nebo grafová struktura prostředí k dispozici, úkol se zjednodušuje na nalezení nejkratší cesty v grafu. Uplatňují se například Dijkstrův algoritmus [15], nebo A* [16] a jiné.

3.2.1 Geometrické metody

Prostředí je popsáno jako soubor polygonů a z jejich vlastností jsou cesty vypočítány jako posloupnost geometrických primitiv, jako jsou čáry, oblouky, splajny, … Nejběžnější postupy jsou založeny na přístupu tzv. vizuálního plánu “visibility roadmap” [33].

- **Voronoiův diagram**

Typickým zástupcem je zobecněný Voronoiův diagram [33]. Tento přístup velmi intuitivní a vhodný pro plánování optimální cesty ve 2D prostoru, jeho rozšíření pro další rozměry, ale není triviální a optimalita takového řešení není zaručena. Tento přístup není vhodný pro plánování v dynamickém prostředí.

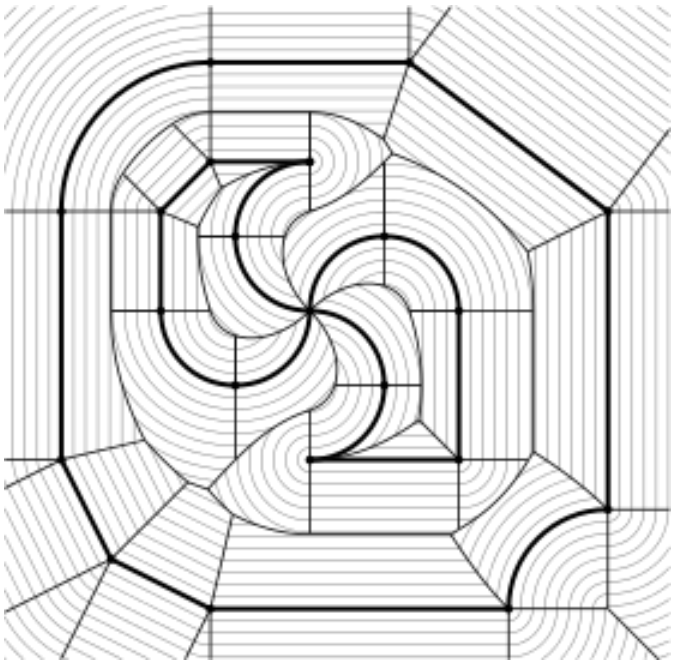


Obr. 5: Voronoi diagram přímkových segmentů a oblouků [36]

Rozšířením Voronoiových diagramů je Delaunayova triangulace prostředí [35]. Triangulace se snaží maximalizovat nejmenší úhel v trojúhelníku. Delaunaiho trojúhelníky jsou přímo spjaty s rozdělením prostředí do Voroniových diagramů. Protože každé těžiště trojúhelníku v trojúhelníkové síti je vrcholem Voroniova polygonu. Jedná se o jednu z nejpoužívanějších triangulační algoritmů. Hlavní nevýhodou je, že výsledkem triangulace může být "zvláštní" trajektorie.

Obr. 6: Znázorňující rozdílné rozdělení vstupních dat mezi Dalaunay triangolací (uprostřed) a Voronoivým grafem (vpravo) [36]

Jak je znázorněno na obrázku níže, v moderních výpočetních přístupech založených na Voroniových diagramech není problém zpracovávat efektivně nejen body a přímky, ale i oblé křivky [36].



Obr. 7: Zobrazení rádiusů pomocí Voronoi diagramů [36]

3.2.2 Metody založené na prohledávání grafových a stromových struktur

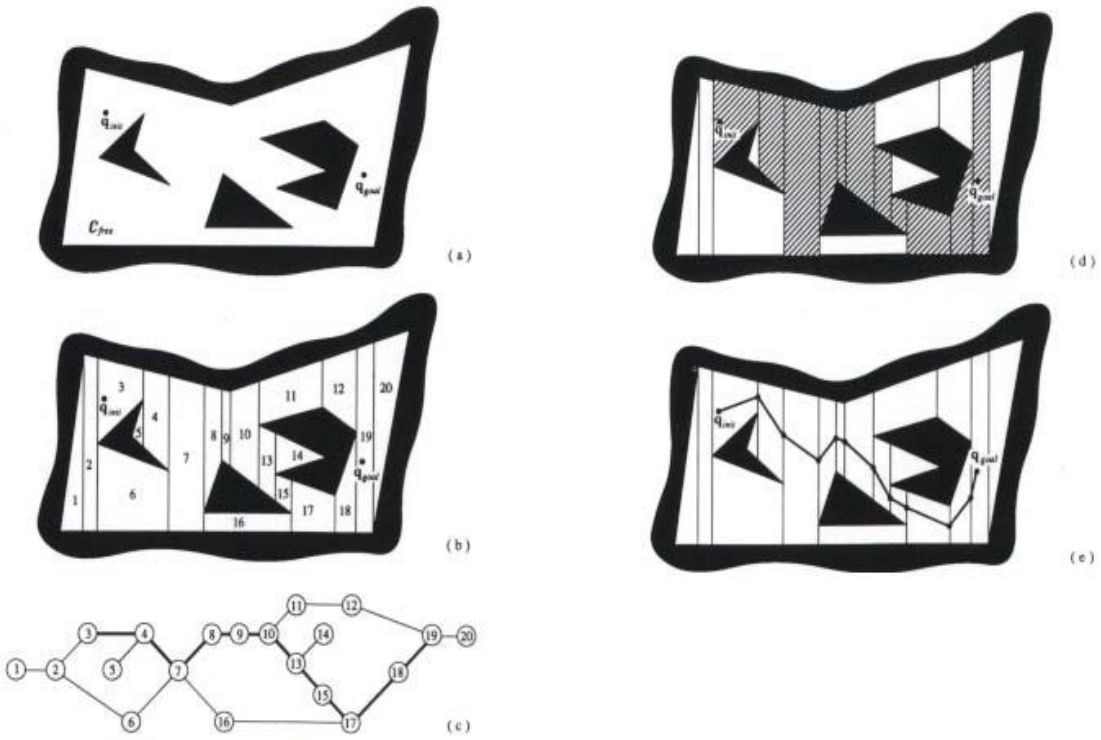
Tento přístup je v posledních letech zkoumán nejaktivněji. V podstatě se jedná o nejjednodušší způsob, jakým prohledávat namapované prostředí a zvýraznit v ní prostor dostupný k plánování. Namapované prostředí je reprezentováno jako stromová, nebo grafová struktura *G(V, E)*. Každý vrchol *V* je ohodnocen stavem, ve kterém se v daném okamžiku *t* nachází. Může nabývat stavů *free*, nebo *obs*. Hrany grafu *E* reprezentují průchozí cestu mezi vrcholy *V* a jsou ohodnoceny podle předem zvolené metriky. Konečná cesta se potom sestavuje s požadavkem na minimální cenu cesty, z počátečního do konečného stavu.

3.2.3 Metody dekompozice prostředí

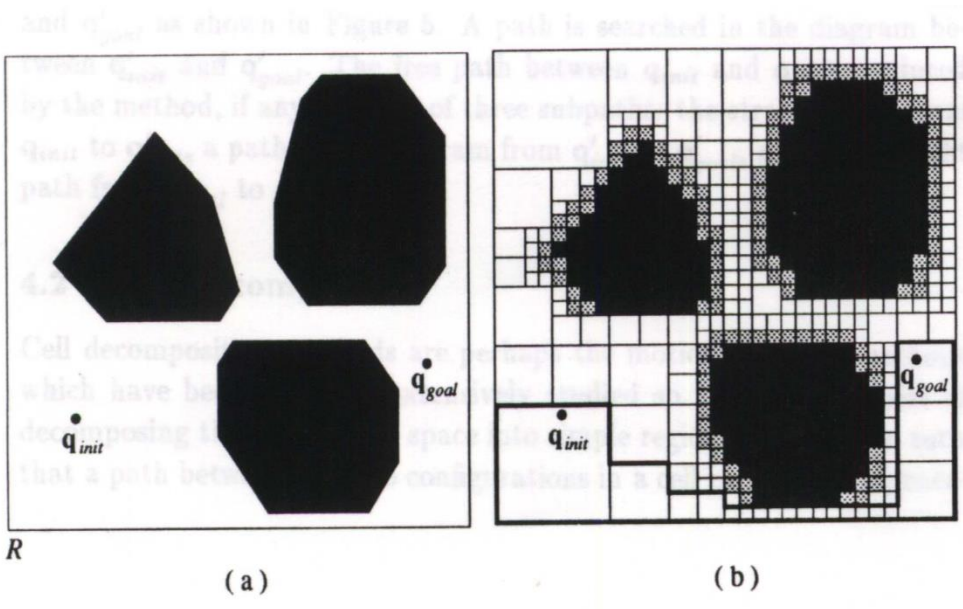
Dekompozice prostředí [37] je přístup, který vycházející z toho, že celé namapované prostředí lze rozdělit do buněk. Následně je z těchto buněk poskládána cesta robota.

Dekompozice může být rozdělena na přesnou, nebo přibližnou. Z uvedených obrázků je patrný rozdíl obou přístupů.

Přesná dekompozice je matematicky náročnější a vede k přesnějšímu rozdělení prostředí. Ale při vhodně zvolené velikosti buněk u dekompozice přibližné, není výsledek takřka rozlišitelný.



Obr. 8: Přesná dekompozice [37]



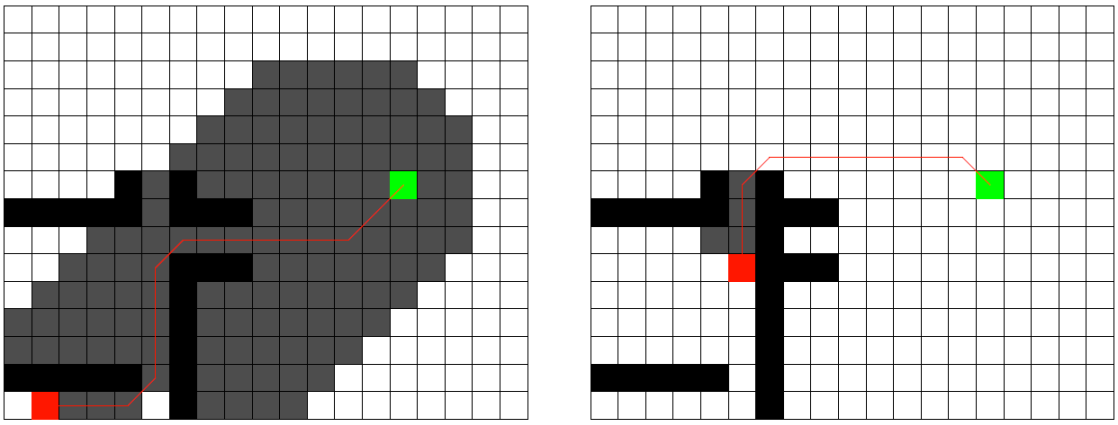
Obr. 9: Nepřesná dekompozice [37]

Nepřesná dekompozice prostředí je na obr.10. Jedná se iterativní proces, při kterém jsou buňky rozdělovány, dokud nedojde ke splnění dekompozičních podmínek.

Zjevným omezením tohoto přístupu je, že takový proces prostředí zkresluje. Přesnost konečného řešení potom závisí na zvolené konečné velikosti buněk. Vyšší hustotě buněk je přímo úměrná zvýšená výpočetní náročnost. Ta se váže jak k sestavování grafu, tak k jeho prohledávání. Velikost grafu prostředí G exponenciálně roste podle počtu dimenzí prohledávaného prostředí.

Říkáme, že výsledek těchto metod je přibližně konečný (approximately complete), protože nevhodné nastavení mřížky nemusí vést k nalezení průchozí cesty.

Důležitou limitací tohoto přístupu je, že není vhodná na typy úloh s kinodynamickými omezeními. Jinou možností, jak přistupovat k dekompozici prostředí, je převést celý namapovaný prostor na mřížku. Ta má běžně čtvercový, nebo šestiúhelníkový charakter. Tento způsob trpí stejně, jako výše zmíněná metoda dekompozice, zvolenou konečnou velikostí buňky.



Obr. 10: Plánovácí v prostředí rozdělené do přesně dané mřížky. [40]

Typickými zástupci této skupiny hledání mohou být Dijkstra [15], nebo A*[16]. popřípadě jejich modifikace D*[39], která efektivně využívá informace použité v předchozím hledání.

3.2.4 A*, D*, D*-lite

Jeden z hlavních a dodnes stále používaných algoritmů patřících do skupiny, které využívá mřížkovou dekompozici prostředí je A*. Ten ale není příliš vhodný pro aplikace v dynamickém prostředí. Protože pokud v grafu dojde k přehodnocení ceny cesty v některých uzlech, což je pro dynamické prostředí běžné, přeplánování vyžaduje nové hledání skrze celý graf. Speciálním případem A*, který se snaží s touto situací pracovat je algoritmus D*, který si i při změně v grafu uchovává informace z předešlého hledání. Ty potom používá při novém hledání a tím zvyšuje jeho efektivitu.

Algoritmus D* a jeho různé deriváty, byl použit jako základní navigační algoritmus při mnoha důležitých praktických aplikacích. Za zmínku určitě stojí použití ve vozítku Mars rover.

Nejdůležitějším rozšířením D* je D*-lite [40]. Ten vykazuje stejné charakteristiky jako D*. Jeho hlavní výhodou je, že je jednodušší pro pochopení, aplikaci a i jeho možná rozšíření.

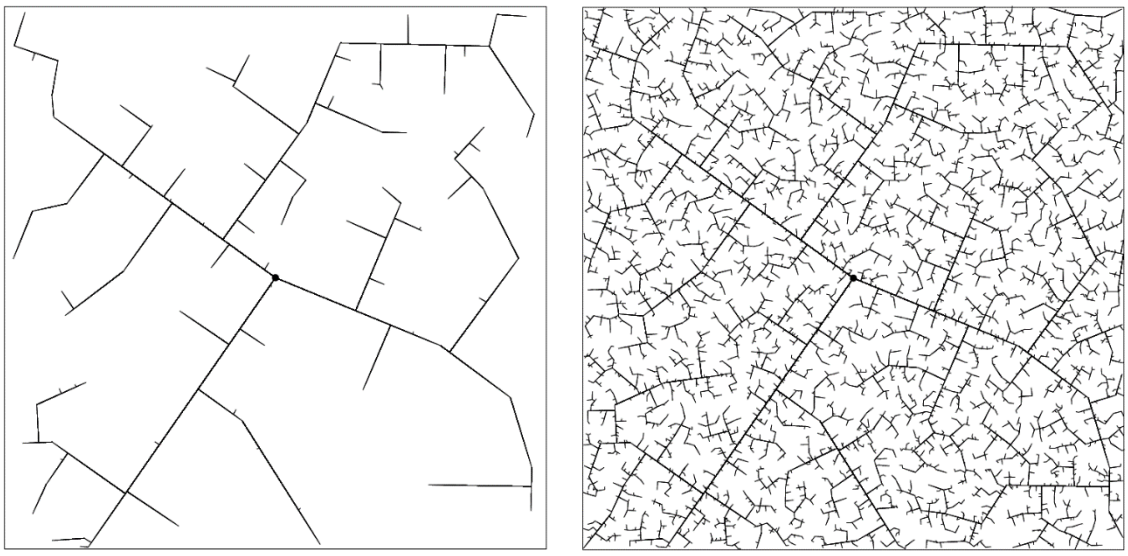
Z praktických experimentů vyplývá, že v případech, kde nastávají značné změny v grafu, je praktičtější začít plánovat od začátku.

3.2.5 Vzorkovací metody

Tyto metody používají skrytou reprezentaci prostředí, tedy narozdíl od metod založených na dekompozici prostředí. Algoritmus v namapovaném prostředí náhodně generuje body a snaží se ho co nejefektivněji pokrýt.

Tyto metody těží z toho, že v cyklu algoritmu není žádný čas vynaložen na přímou rekonstrukcí překážek, nebo samotného prostředí. To je zobrazeno pouze symbolicky ve formě buněk, která náleží do skupiny X_{free}, nebo X_{obs}. To jim přináší velkou časovou a výpočetní úsporu proti jiným komplexnějším řešením.

Nejpoužívanější přístupy, z této skupiny, jsou algoritmy založené na Rapidly exploring random trees (RRT) [2] a Probabilistic Roadmap (PRM) [41].



Obr. 11: Jedná se o prostředí pokrývané stromem algoritmu RRT. Na levém obrázku je pokrytí při 45. iteraci, na pravém je při 2345. iteraci. Jedna iterace znamená jeden maximálně jeden nový bod. [2]

Body, které algoritmus v prostoru vygeneruje a náleží do množiny X_{free}, jsou vloženy do stromu [2], nebo grafu [41]. Hrana k tomuto vrcholu je vytvořena podle specifických kroků každého algoritmu. I podobné algoritmy, jako RRT a RRT*, mají rozdílné přístupy při ustavování průchozí cesty mezi vrcholy. Procedura přidání vrcholu

do grafu sebou nese několik operací. Ohodnocení cesty mezi body, zjištění kolizí na cestě, zjištění nejvhodnějšího rodiče atd... Za zmínku určitě stojí, že rozdíl mezi asymptoticky optimálním RRT* a RRT je právě v proceduře přidávání vrcholu do grafu. A stejně tak i nejmodernější realtimové algoritmy založené na RRT, jako RRT^X [43], nebo CL-RRT# [44] se ve svých přístupech velmi výrazně věnují časové a výpočetní optimalizaci přidávání nových hran do stávajícího řešení.

Takto získaný graf, svými vrcholy a hranami reprezentuje prohledávaný prostor a slouží jako podklad k řešení původního problému hledání cesty. Za nejlepší je považováno takové řešení, které obsáhne všechny důležité uzly daného prostředí, a tedy je schopno dosáhnout optimálního řešení s nejmenším možným množstvím hran a uzlů.

Nejjednodušší strategie hledání, využívají uniformního generování bodů přes celé X [2]. Takto generovaný strom je popsán jako „Space filing tree“ [45]. Ty komplexnější pak v sobě kombinují více přístupů, proto aby generování bodů směřovalo efektivněji k cíli. Nejlépe tak, že se prohledávaný prostor co nejvíce zúží. Příkladem využívajícím alespoň dvou typů prohledávání je algoritmus Informed-RRT [43]. Ten v první fázi generuje body obdobně jako RRT, které je popsáno níže, po nalezení cílového bodu se jeho strategie změní. Algoritmus generuje body jen v oblasti elipsy. Ta je tvořena dvěma ohnisky x_{init} a x_{goal} a její šířka je dána tak aby co nejtěsněji obalovala nalezenou cestu. Je matematicky dokázáno [43], že kratší cesta, pokud existuje, musí ležet uvnitř této elipsy. Struktura stromu v této oblastí je z optimalizované verze RRT, tedy RRT*.

Pokusíme-li se vzorkovací metody zobecnit, pak se jejich základní úloha dá definovat takto. Je to snaha o nalezení minimálního potřebného množství uzlů a jejich pozic v daném prostoru, tak aby řešení mohlo být kompletní a optimální, respektive suboptimální.

Na těchto základních algoritmech je postavena většina moderních metod z kategorie vzorkovacích algoritmů. Jejich hlavním problémem je stochastický charakter. U těchto základních algoritmů, cesta nalezená takto náhodně generovanými body není optimální a ani hladká.

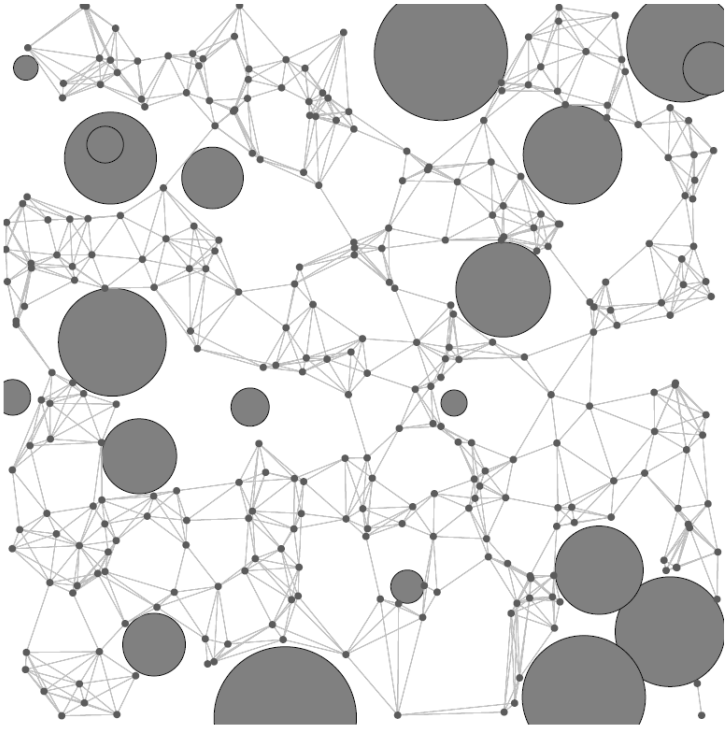
V práci [7] jsou navrženy úpravy těchto algoritmů RRT*, PRM* a RRG. Zmíněný problém je v ní odstraněn. Tato práce je pro vzorkovací metody zlomová, protože v ní navržené algoritmy zaručují hladkou a asymptoticky optimální řešení. S množstvím vzorků blížícím se nekonečnu, algoritmus RRT* zaručuje dokonce řešení optimální. A to jen s minimálním navýšením časových požadavků oproti původním algoritmům.

Proto je také většina nově uváděných algoritmů nástavbou těchto tří algoritmů. Ty dnes patří k nejvíce prakticky používaným přístupům. Tato práce není jediná, která nabízí asymptoticky optimální řešení. Takové řešení nabízí i algoritmus RRT# [42], který k problému přistupuje jinak než RRT*. Ale jeho přístup je na RRT* z velké části postaven.

3.2.6 Probabilistic Roadmap – PRM

Tento algoritmus byl uveden v roce 1996 [41] a je možná nejvíce známou formou algoritmů založených na náhodném vzorkování. Nejzákladnější implementace toho algoritmu přesně odpovídá nahoře uvedenému popisu.

Tedy graf *G(V, E)* je tvořen náhodným vzorkováním v namapovaném prostoru. Pokud je vygenerovaný vzorek x_{rand} označen jako X_{free}, je označen jako vrchol *V* a vložen do grafu. Následně se algoritmus vytvoří hranu mezi všemi vrcholy z nejbližšího okolí x_{rand}. Pokud je hran vyhodnocena jako průchozí, je vložena do grafu. Vzorkování probíhá, dokud není dosaženo potřebné hustoty sítě.



Obr. 12: Příklad grafu vygenerovaného ve 2D prostředí pomocí PRM [7]

```
1 V ← ∅; E ← ∅;
2 for i = 0, ..., n do
3     x_rand ← SampleFree_i;
4     U ← Near(G = (V, E), x_rand, r) ;
5     V ← V ∪ {x_rand};
6     foreach u ∈ U, in order of increasing ‖u − x_rand‖, do
7         if x_rand and u are not in the same connected component of G = (V, E) then
8             if CollisionFree(x_rand, u) then E ← E ∪ {(x_rand, u), (u, x_rand)};
9 return G = (V, E);
```

Obr. 13: Hlavní smyčka algoritmu [7]

Ve chvíli, kdy je graf prostředí vygenerován, jeho vyřešení je poměrně rychlé. V tomto bodě je možné nejkratší cestu nalézt, jako nejkratší cestu grafu [16].

Využití vhodných datových struktur, jako například KD-trees, v těchto případech značně zvyšuje efektivitu prohledávání.

Faktor, který značně limituje implementaci PRM pro dynamické prostředí, je že je potřebná plná znalost prohledávaného prostředí, a to včetně polohy dynamických překážek. To vyžaduje buď periodické generování nového grafu, podle aktuálního stavu mapy, nebo testování jednotlivých hran na jejich průchodnost. Časová náročnost algoritmu může být značná, záleží převážně na množství překážek a velikosti prohledávaného okolí.

3.3 Současná realtimová řešení

Jak již bylo uvedeno, do popředí se stále více dostávají aplikace, které jsou nasazovány do reálného prostředí. Tedy dynamického prostředí s jistou mírou neurčitosti. Tato nejistota může být způsobena šumem v senzorech, nebo od objektů v prostředí. Protože jejich chování není vždy možné předem určit. Přestože se reálné aplikace dneška zaměřují především na plánování cesty robota v dynamickém prostředí, a i přes velkou oblibu RRT algoritmů, není k dispozici mnoho modifikací tohoto přístupu. Která by byly přímo navrženy pro dynamické prostředí. Většinou se i u reálných úloh setkáme s aplikací RRT*, RRT# algoritmů, ty jsou ale původně navrženy do prostředí statického.

V této kapitole uvedeme jednu realtimovou implementaci RRT z poslední doby, která je cíleně navržena na práci v dynamickém prostředí. Ale vykazuje i velmi dobré výsledky, srovnatelné s RRT* a RRT#, v prostředí statickém.

3.3.1 RRT^{X}

Tento algoritmus [43] je speciálně navržený pro plánování a přeplánování pohybu robota v dynamickém prostředí s nepředvídatelnými překážkami. Tedy s takovými, které se mohou nečekaně objevit, zmizet, nebo změnit směr pohybu. V uvedené práci je algoritmus srovnáván i ve statickém prostředí s algoritmy RRT* a RRT# a i v tomto prostředí dosahuje velmi dobrých výsledků.

Rozdílem oproti těmto algoritmům je, že plánování cesty agenta, a tedy i kořen stromu, začíná v X_{goal}. Výhodou tohoto přístupu je, že kořen stromu se nemusí přesouvat při pohybu robota. A kdykoli je detekována překážka na nalezené cestě, provede se kaskádovitý převin stromu a znova se hledá nejkratší cesta.

```
1  V ← {v_goal} ;
2  v_bot ← v_start ;
3  while v_bot ≠ v_goal do
4      r ← shrinkingBallRadius() ;
5      if obstacles have changed then
6          updateObstacles() ;
7      if robot is moving then
8          v_bot ← updateRobot(v_bot) ;
9      v ← randomNode(S) ;
10     v_nearest ← nearest(v) ;
11     if d(v, v_nearest) > δ then
12         v ← saturate(v, v_nearest) ;
13     if v ∉ X_obs then
14         extend(v, r) ;
15     if v ∈ V then
16         rewireNeighbors(v) ;
17         reduceInconsistency() ;
```

Obr. 14: Hlavní smyčka RRT^X [43]

Popis algoritmu:

- Vstupem algoritmu RRT^X je prohledávaný prostor X a náhodná sekvence vzorků S
- V prvních dvou krocích se inicializují pozice $x_{goal} = v_{goal}$ a $x_{start} = v_{start}$
- Hlavního smyčka algoritmu s podmínkou „*while* $x_{goal} \neq x_{start}$“ .
- Dalšími kroky je sada podmínek, která kontroluje pohyb agenta a překážek v prostředí. Popřípadě aktualizuje jejich polohu v grafu.
- Na řádku 9 je vygenerován x_{Rand} a na řádku 10 je nalezen jeho nejbližší „soused“ $x_{Closest}$.
- Na řádku 11 je ověřeno, zda není vzdálenost mezi x_{Near} a x_{Rand} příliš velká. Pokud ano dojde k její korekci.
- V dalším kroku se ověří, zda x_{Rand} nenáleží do X_{Obs}, pokud nenáleží, tak x_{Rand} je vloženo do množiny bodů a mezi tímto bodem a jeho rodičem je vytvořena hrana
- Pokud byl bod přidána mezi vrcholy, pak dojde k úpravě nejbližšího okolí a redukci vzniklých nekonzistencí

Zbytek algoritmu je velmi podrobně uveden v práci o RRT^X, popřípadě jeho praktická realizace je k vidění na videích, které se dají najít na internetu.

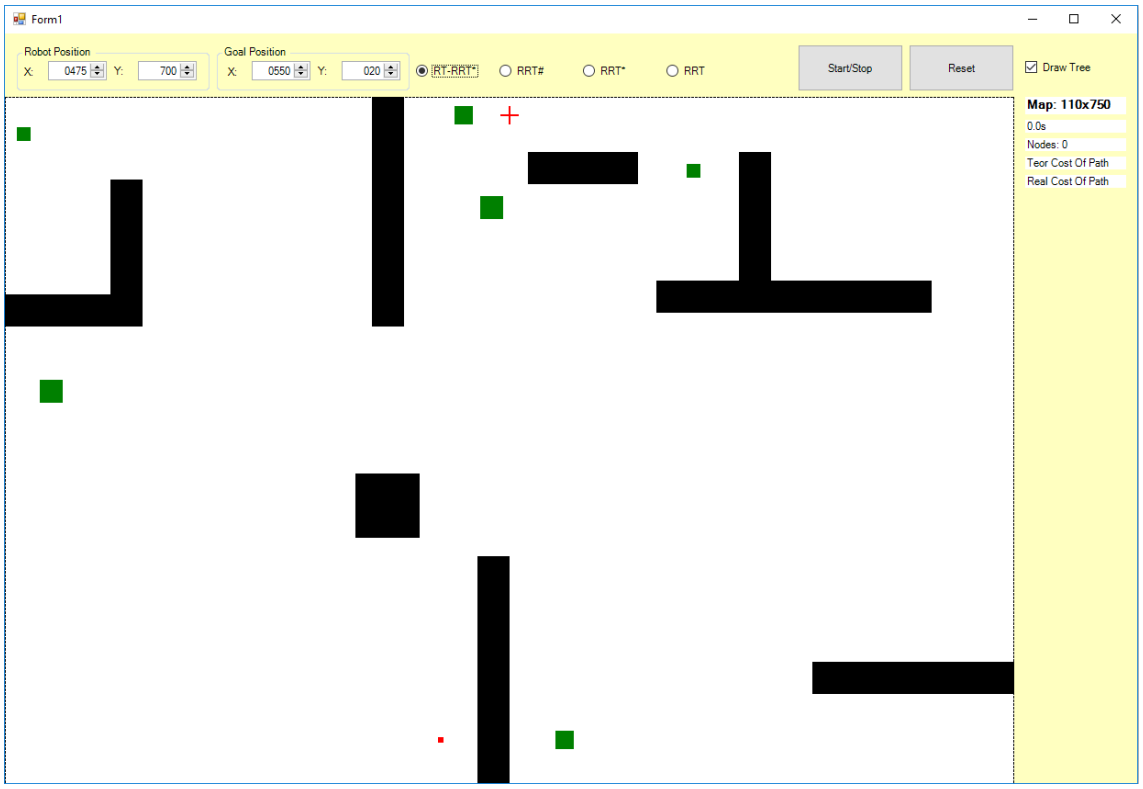
Závěrem ještě můžeme uvést algoritmy CL-RRT# a FND-RRT, které jsou také vhodnými algoritmy pro dynamické aplikace, použitelné i v částečně neznámém prostředí.

4 VLASTNÍ ŘEŠENÍ

Pro vlastní implementaci byly vybrány algoritmy RRT, RRT* a RT-RRT*. První dva zmíněné jsou původně určeny do známého statického prostředí. Proto byly tyto algoritmy upraveny tak, aby byly schopny reagovat i na dynamické změny v prostředí. Tato úprava byla inspirována přístupem RT-RRT*, který je i původně navrhnut jako realtimový algoritmus.

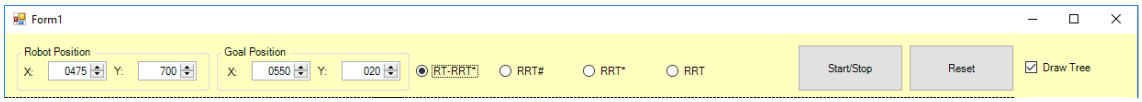
4.1 Testovací prostředí

Pro simulaci těchto algoritmů bylo připraveno testovací prostředí s využitím technologie Windows Forms. Algoritmy byly napsány v jazyce C# a jako datová struktura bylo využito RRT stromů.



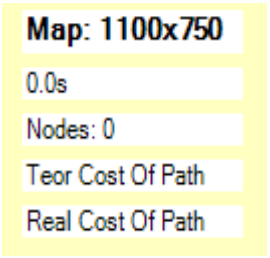
Obr. 15: Simulační prostředí

První část programu, obr.15, je v horní části programového okna. Slouží pro nastavení počáteční a koncové pozice robota, pro výběr algoritmu, start a reset programu. Umožňuje uživateli zvolit, zda chce vizualizovat strom vytvořený pomocí zvoleného algoritmu.



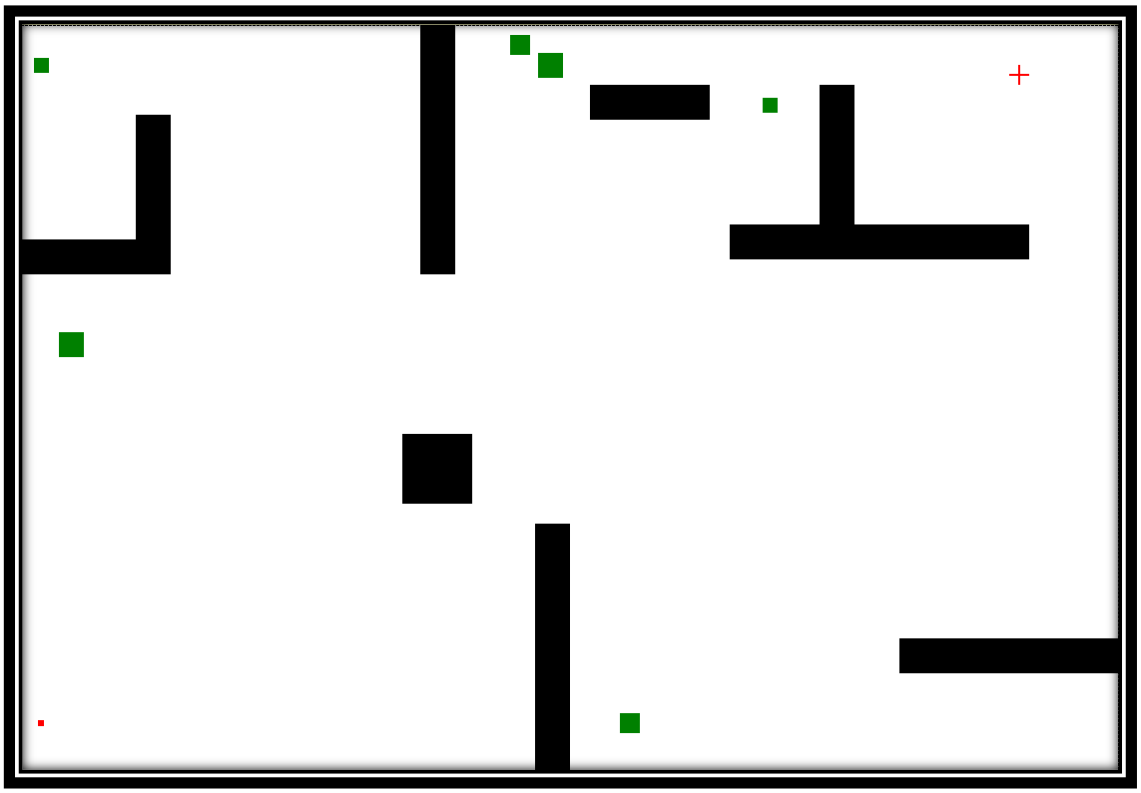
Obr. 16: Uživatelská část programu

Druhá část programu je na pravé straně okna programu a slouží k zobrazování potřebných informací. Ty jsou později využity pro porovnání algoritmů. Zobrazena je v prvním řádku velikost mapy. Na druhém doba dosažení cíle. Třetí a čtvrtý zobrazují cenu nalezené cesty. Třetí zobrazuje cenu cesty z aktuálního polohy robota a čtvrtý reálnou hodnotu, kterou robot při cestě dosáhl.

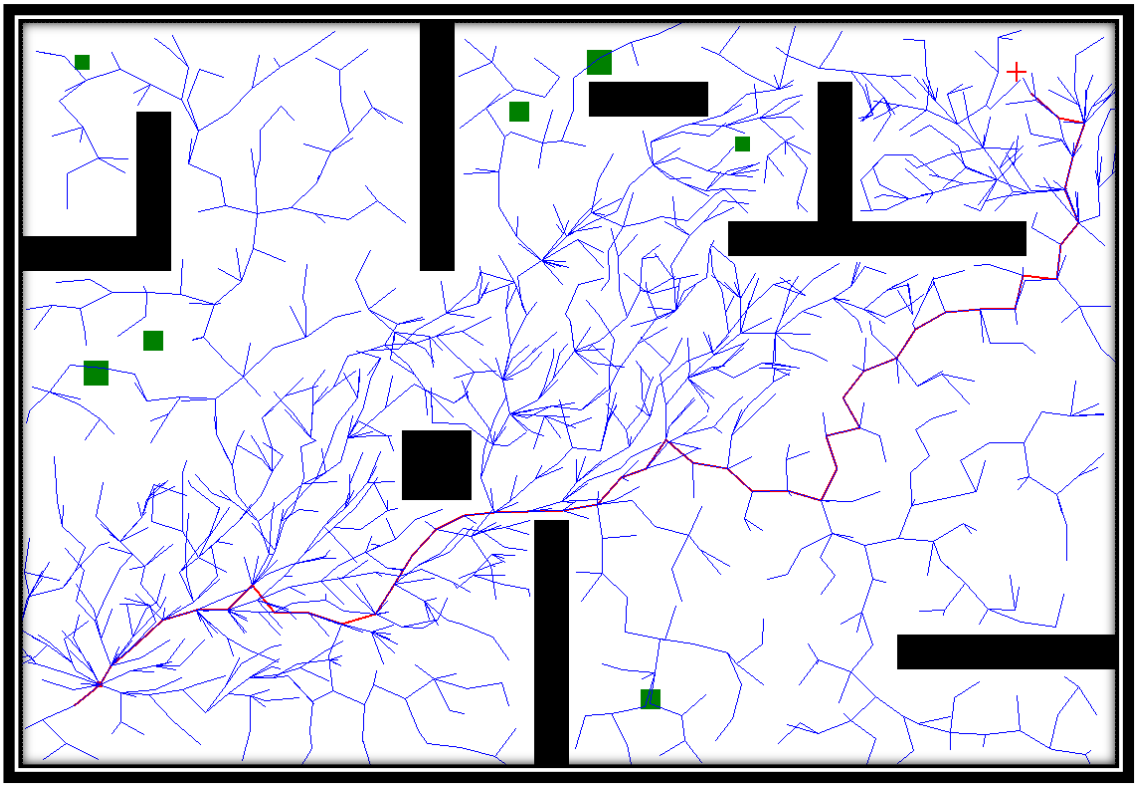


Obr. 17: Informativní část programu

Poslední části je zobrazeno simulační prostředí. Tedy mapa, ve které se agent musí pohybovat.



Obr. 18: Mapa připravená pro simulaci robota



Obr. 19: Vizualizace cesty robota, po vygenerování stromu a nalezení cesty.

Na mapě jsou označeny statické překážky černou barvou a dynamické zelenou. Cílový bod je označen červeným křížem a počáteční bod červeným čtvercem.

Modrou barvou je označen dosud vygenerovaný strom a červenou je v něm označena nalezená průchozí cesta. Ta se s přibývajícími iteracemi postupně mění.

4.2 Algoritmus přeplánování

Procedura přeplánování je pro všechny implementované algoritmy stejná. V pravidelných intervalech je kontrolováno nejbližší okolí cílového bodu. Jsou vybrány všechny body, které se v tomto okolí nalézají. Následně je pro ně určena aktuální cena cesty. Ta je určena tak, že každý vrchol v grafu má nejvíce jednoho rodiče a udržuje si na něj referenci a cenu cesty k němu. Výsledná cena cesty je potom sumou těchto cen, mezi nalezeným vrcholem a kořenem stromu. Kořen stromu ve zmíněných implementacích reprezentuje polohu robota. Pro agenta je potom zvolena taková cesta, která nabízí nejnižší cenu a zároveň se dá označit jako bezkolizní.

Při přeplánování, vycházíme z předpokladu, že pokud jsou v okolí cílového bodu nalezeny vrcholy stromu, pak jednoznačně existuje průchozí cesta k tomuto bobu. A pokud agent pro toto okolí není schopen takovou cestu nalézt, neznamená to, že taková cesta neexistuje, ale že je přehrazena dynamickou překážkou. V takovém případě se nejbližší oblast cílového bodu zvětší tak, aby byl agent schopen cestu nalézt.

Cena takové cesty je pro agenta nevýhodná, proto se stejným přístupem stále snaží najít výhodnější řešení.

Takto nalezená cesta nemusí nutně končit v předem určeném nejbližším okolí cíle, proto je vhodné, kolem koncového bodu každé nově nalezené cesty strom převinout. Tím se i z původně dražší cesty, můžeme dostat k suboptimálnímu řešení, pro stávající podmínky agenta.

Toto přeplánování probíhá před každým pohybem agenta. To pro nás v této práci zajišťuje časovou podmíněnost, která byla zmíněna v úvodu v souvislosti s reálnými aplikacemi.

Algoritmus `UpdateAgentRoot`:

```
public void UpdateAgentRoot()
       lock (Constants._object)
              agent.NextStep = PlanAgentMove();
              if (NextStep != null)
                     IReadOnlyList<Node> nodes =
              tree.FindClosestNodes(Root,MAX_EUC_DISTANCE + 1);

                     foreach(Node node in nodes)
                     if (node.Parent != null && node.Parent.Equals(Root))
                            node.CostToParent = Tree.Distance(NextStep, node);
                            node.Parent =NextStep;
                     RealCostOFPath += Tree.Distance(agent.NextStep, agent.Root);
                     Root.Parent = NextStep;
                     Root.CostToParent = Tree.Distance(NextStep,Root);
                     Root = NextStep;
                     NextStep.Parent = null;
                     IsInGoal = Root.Equals(Goal);
```

Algoritmus implementuje třída, která je děděna všemi uváděnými algoritmy. Tato metoda je volána z hlavního okna aplikace a zajišťuje aktualizaci polohy agenta. Jak je z kódu patrné, pokud je metodou `PlanAgentMove` vygenerován nový krok `NextStep`. Jsou následně všechny uzly, které se se odkazují na současnou polohu agenta, přereferencovány na tuto novou polohu. Současná poloha agenta je označena jako `Root`. Následně je i agentova reference mezi `Root` a `NextStep` zaměněna. To znamená, že kořen stromu se pohybuje s agentem a při agentově pohybu není nutné generovat nový strom.

Algoritmus `PlanAgentMove`:

```
internal Node PlanAgentMove()
	while (true)
		IReadOnlyList<Node> nodes = tree.FindClosestNodes(goal,area);
		if (nodes.Count == 0)
			return null;

		double costOfPath = double.MaxValue;
		List<Node> Path = null;
		foreach (Node node in nodes)
			double cost = Tree.LenghtOfShortestPath(node, root);
			if (node.Equals(root))
				return agent.Goal;

			if (cost < costOfPath && IsPathFreeForMove(node, root))
				Path = Tree.GetPath(node, root);
				 costOfPath = cost;
		if (returnNode != null)
			agent.EstimatedCostOFPath = costOfPath;
			agent.Path = Path;
			return Path[Path.Count - 1];
		else
			goalArea *= 2;
```

Tato metoda má za úkol určit agentův další krok. Pokud jsou nalezeny body v nejbližším okolí cílového bodu. Je pro každý takový bod nalezena cesta. Ta je ohodnocena a je zjištěno, zda je cesta bezkolizní. Pokud je taková cesta nalezena je předána agentovi její cena, všechny body, které obsahuje a následný krok.

Pokud taková bezkolizní cesta v daném okolí cílového bodu neexistuje, je oblast dvojnásobně rozšířena. Jedná se o dočasné řešení situace, protože cena takové cesty je vyšší než jiné, ale zatím nedostupné, možnosti. Protože se střed stromu pohybuje spolu s agentem. Z charakteru stromu je dáno, že všechny vrcholy stromu vedou do jeho kořene. Není pro agenta problém cestu přeplánovat bez nadbytečných kroků tak, aby nově zvolená cesta měla nižší cenu cesty než ta současná. Tedy pokud je taková cesta k dispozici.

Poslední zmíněný bude algoritmus `IsPathFreeForMove`. Ten má za úkol zjistit, zda je cesta průchozí. V současném provedení pracuje tak, že algoritmus prochází celou cestu, ne které generuje v dostatečně malých rozestupech obrys agenta. A u těchto obrazů zkouší, zda nejsou v interferenci s dynamickými překážkami na cestě. Tento způsob je pro praktickou realizaci zbytečně výpočetně nákladný.

Jako případnou optimalizaci tohoto procesu by bylo vhodné použít některý z existujících algoritmů, které řeší kolize robota. Nebo, pro toto řešení vhodnějším způsobem, může být využití možností RRT. Pomocí RRT a souřadnic dynamických překážek zjistit, zda se v okolí překážek nachází nějaký úsek cesty a pokud ano, určit jeho nejmenší možnou vzdálenost. Ta musím být vyšší než nejnižším dovolená.

4.3 Rapidly Exploring random tree – RRT

Práce byla uvedena v roce 1998 [2] a prezentuje algoritmus pro zvládání holonomních i neholonomních omezení. Včetně plánování s vysokým stupněm volnosti.

Je primárně určena pro úkoly s jediným požadavkem. To znamená, že algoritmus funguje v otevřené smyčce a po nalezení průchozí cesty, respektive dosažení požadované hustoty sítě, dále své okolí neprohledává.

Algoritmus je navržen s ohledem na použití minimální množství heuristik a potřebných parametrů. To vede k lepší analýze výkonu a konzistentnějšímu chování. Velkou výhodou je, že algoritmus může být přímo použit na neholonomní i holonomních plánování.

Během každé iterace algoritmu je do stromu přidán bod x_{New} pokud $x_{New} \in X_{Free}$. Následně se provede propojení nejbližšího listu stromu s nově vygenerovaným bodem. Pokud je takové propojení možné, bod je přidán do setu vrcholů s odkazem, hranou, na svého předka.

```
while (agent.NumberOfNodes < numberOfSamples)
       if (this.SearchRunning)
              lock (Constants._object)
                     Node xRand = GenerateNewNode();
                     Node xClosest = tree.FindClosestNode(xRand);
                     Node xNew = Steer(xRand, xClosest);
                     if (MapManager.IsPointObstacle(xNew))
                            continue;

                     if (!MapManager.IsPathFree(xNew, xClosest,
              MapManager.staticObs))
                            continue;

                     xNew.Parent = xClosest;
                     xNew.CostToParent = Tree.Distance(xNew, xClosest);
                     tree.Insert(xNew);
```

Popis algoritmu:

- Při inicializaci stromu, je do něj vložen počáteční vrchol $x_{Init} \in X_{Free}$.
- Ve druhém kroku se algoritmus dostává do smyčky, která má předem předepsané množství kroků K. V tomto algoritmu se K=V.
- Ve třetím kroku je vygenerován náhodný bod x_{Rand}.
- Ve čtvrtém kroku je nalezen nejbližší existující vrchol k tomuto bodu a je ohodnocena vzdálenost k robotu podle zavedené metriky.
- V pátém kroku se určí vektor u mezi body x_{Rand} a $x_{Closest}$. Na tomto vektoru je vygenerován bod $x_{New,}$ mezi $x_{Closest,}$ a maximální dovolenou vzdáleností od tohoto bodu.
- Následně je nový bod x_{New} přídán do stromu T. Také je přidána hrana mezi $x_{Closest}$ a x_{New}.
- Návratovou hodnotou algoritmu je graf $G(V, E)$

Tento přístup má několik hlavních výhod.

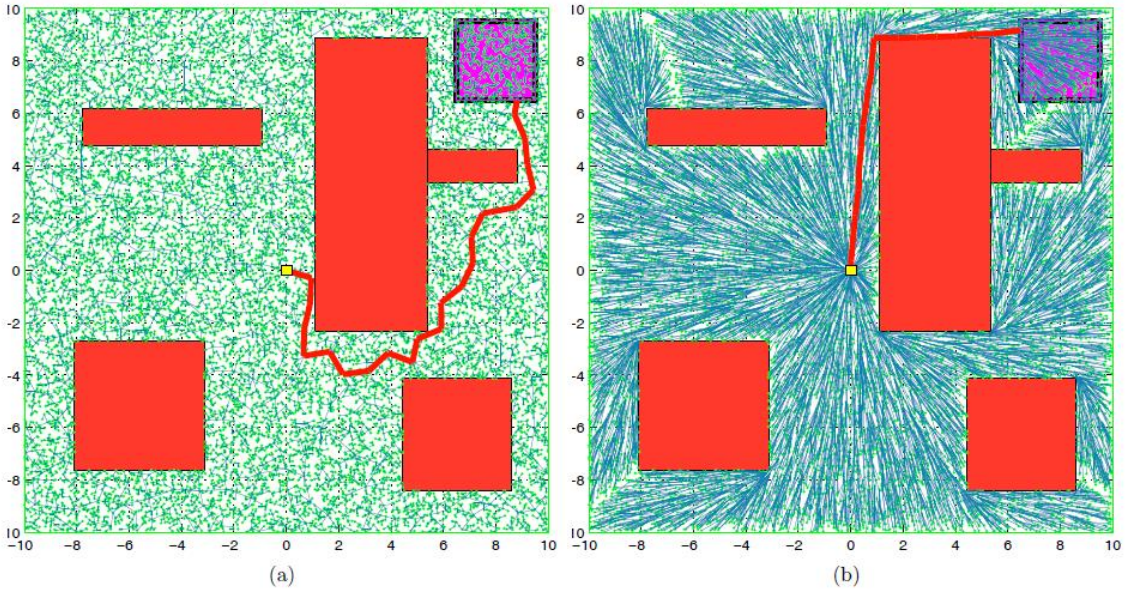
- Výhodou je, že při každé expanzi mohou být brána v úvahu holonomní kritéria, to znamená, že všechny listy, které jsou přidány do stromu jsou dostupné pro dynamiku robota.

- Distribuce uzlů na mapě odpovídá vzorkovacímu rozdělení (uniformnímu rozdělení). To vede ke konzistentnímu pokrytí mapy.
- RRT povede k nalezení cíle i za velmi obecných podmínek.
- RRT je dostatečně jednoduché, což také usnadňuje analýzu výkonu. A i když se cesta jeví zubatě, neobjevují se v ní žádné spirály.

Ideální výkon dostaneme, pokud se podaří navrhnout vhodnou metriku, která zajistí nejkratší možnou cestu mezi dvěma stavy. Navržení takto optimálního přístupu je ale často samo velmi náročné.

Velká nevýhoda spočívá v náhodném generování bodů, takže nalezená cesta může mít klikatý charakter a nalezená cesta nezaručuje svoji optimalitu. Ale jak je uvedeno výše, toto se u RRT* algoritmu podařilo odstranit.

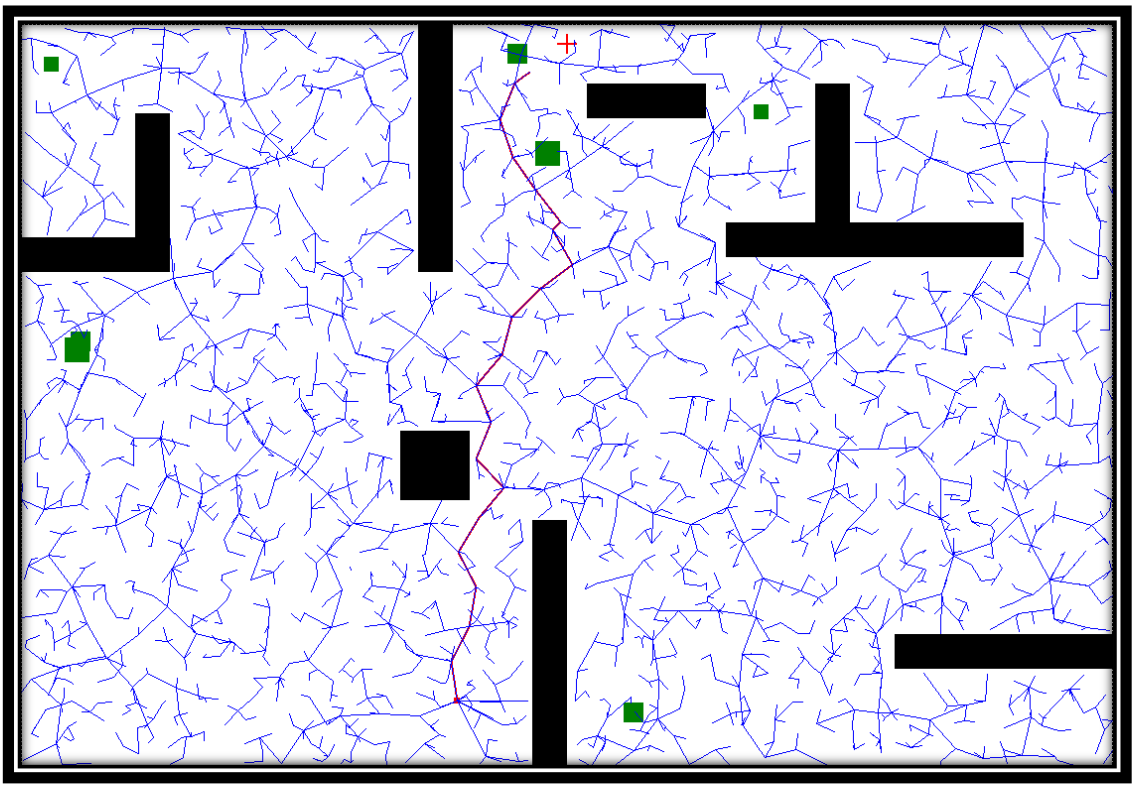
Autoři tohoto algoritmu si již na začátku uvědomovali nedostatky a možná zlepšení tohoto řešení. Například více generovaných stromů v jednom řešení, nebo efektivnější přidávání listu ke stromu



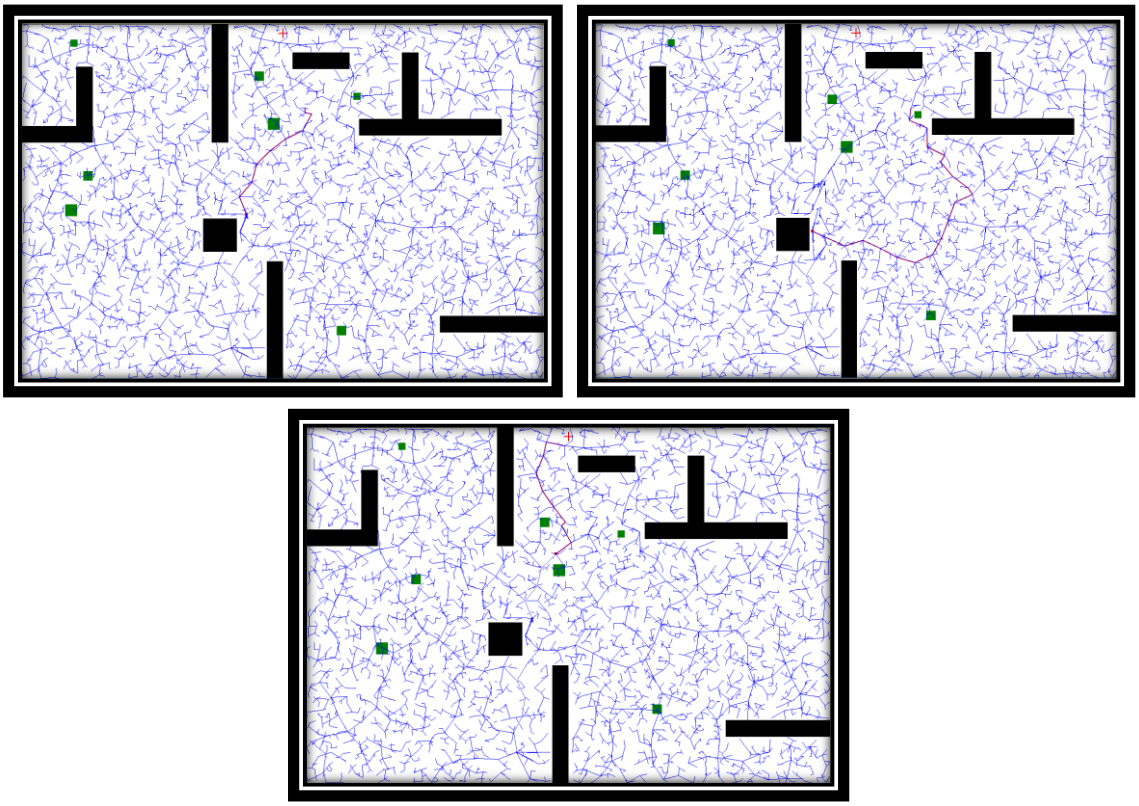
Obr. 20: Porovnání RRT (vlevo) a RRT*, oba se stejným počtem uzlů 20 000. Rozdíl v ceně cesty byl přes 30%. [7]

RRT pracuje ve směru zatím neprozkoumaného prostředí. Je navrženo na postupné zmenšování velkých Voronoiových oblastí na menší. Z experimentů vyplývá, že nalezená cesta není příliš vzdálená té optimální, a že listy stromu nakonec mají uniformní rozdělení skrze celé prostředí.

Simulace RRT:



Obr. 21: Cesta nalezená pomocí RRT



Obr. 22: Na obrázcích je zachyceno přeplánování cesty, podle dostupnosti a ceny cesty pro RRT algoritmus

4.4 Optimal Rapidly Exploring random tree RRT*

Algoritmus RRT* je optimalizovaná verze algoritmu RRT, a byl představena v práci [7]. Úpravy algoritmů, které jsou v ní popsány jsou navrženy jako asymptoticky optimální. Přitom časová náročnost algoritmu, jejíž důkaz je součástí uvedené práce, je s RRT srovnatelná. RRT* je postaven na algoritmu RRG, který původně vychází z RRT, jen místo grafu využívá mnohem efektivnějšího přístupu skrze stromovou strukturu.

Body jsou do stromu přidávány úplně stejným způsobem jako v případě RRT. Rozdílné jsou ale podmínky pro vytvoření hrany mezi těmito body. Hrana je vytvořena podle kritéria.

Hrana může vzniknout mezi vrcholy obsaženými v množině X_{Near} a listem x_{new}. Pokud je to cenově výhodnější než vytvořit hranu s $x_{Closest}$.

Algoritmus stejně jako v případě RRT vrací strom $G(V, E)$.

Hlavní smyčka algoritmu:

```
public override void MainLoop()
       while (this.SearchRunning)
              if (agent.NumberOfNodes < numberOfSamples)
                     lock (Constants._object)
                            Node xRand = GenerateNewNode();
                            Node xClosest = tree.FindClosestNode(xRand);
                            Node xNew = Steer(xRand, xClosest);
                            if (MapManager.IsPointObstacle(xNew))
                                   continue;

                            ConectAlongMinPath(xNew, xClosest);
                            if (!MapManager.IsPathFree(xNew, xNew.Parent,
                            MapManager.staticObs))
                                   continue;

                            tree.Insert(xNew);
                            RewireTree(xNew);
              if(agent.NextStep != null)
                     RewireTree(agent.NextStep);
```

Popis algoritmu:

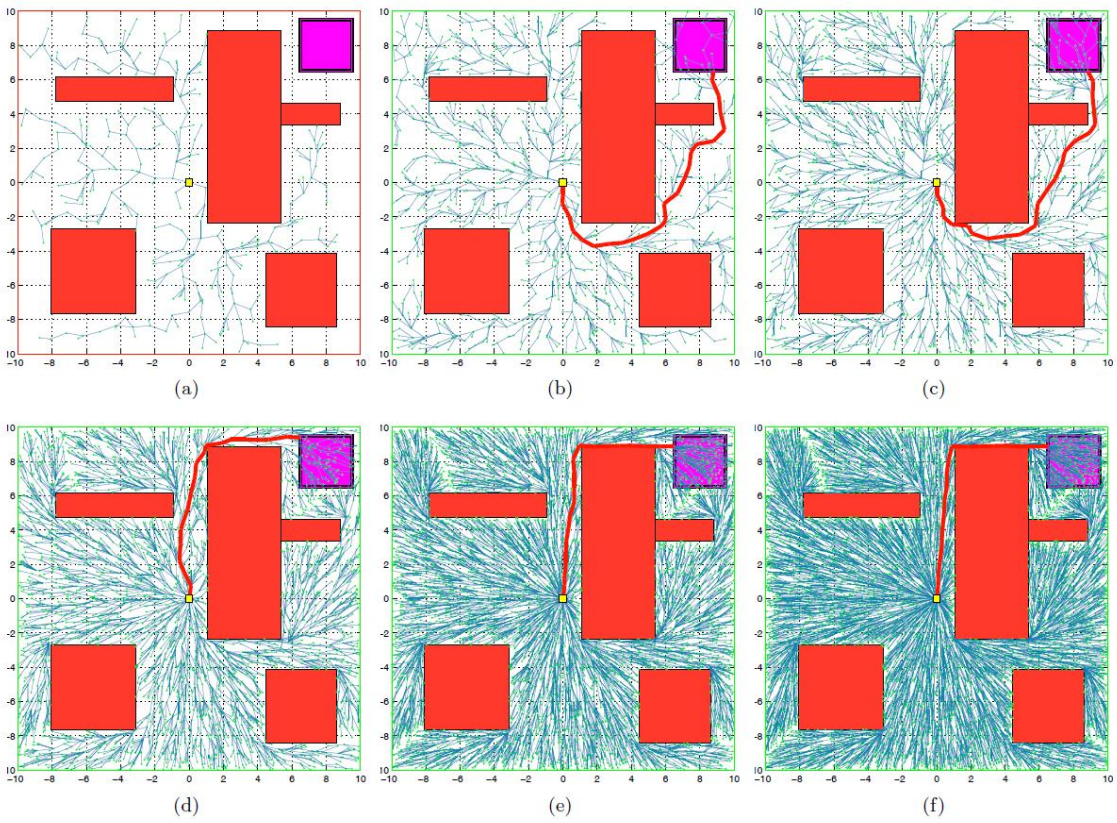
- První je inicializace stromu s počátečním vrcholem.
- Následně cyklus, jehož podmínkou je maximální počet vrcholů v prostředí.
- Na řádku 5-8 je vygenerován v prostoru nový bod x_{Rand}. Je nalezena jeho varianta x_{New}, ta je přídána do stromu i s hranou k nejbližšímu vrcholu $x_{Closest}$, jak je popsáno v RRT
- Následně jsou vybrány body v okolí x_{New}, které jsou označeny jako X_{near}
- V několika krocích je ověřena průchodnost cesty mezi X_{Near} *a* x_{New}. Funkcemi `IsPathFree a IsPointObstacle`.
- V posledních krocích dochází převíjení grafu ve směru nejmenší ceny cesty.

Algoritmus `ConectAlongMinPath`:

```
protected Node ConectAlongMinPath(Node treeNode, Node newNode, List<Node>
nearNodes)

        Node treeNode2 = treeNode;
        double costMin = Tree.LenghtOfShortestPath(treeNode, agent.Root) +
        Tree.Distance(treeNode, newNode);

        foreach (Node node in nearNodes)
                if (agent.Root.Equals(node))
                        continue;
                double distance = Tree.Distance(node, newNode);
                double costNew = Tree.LenghtOfShortestPath(node, agent.Root) +
        distance;
                if (costNew < costMin &&  distance < 50 &&
                MapManager.IsPathFree(node, newNode, MapManager.staticObs))
                        costMin = costNew;
                        treeNode2 = node;
        return treeNode2;
```

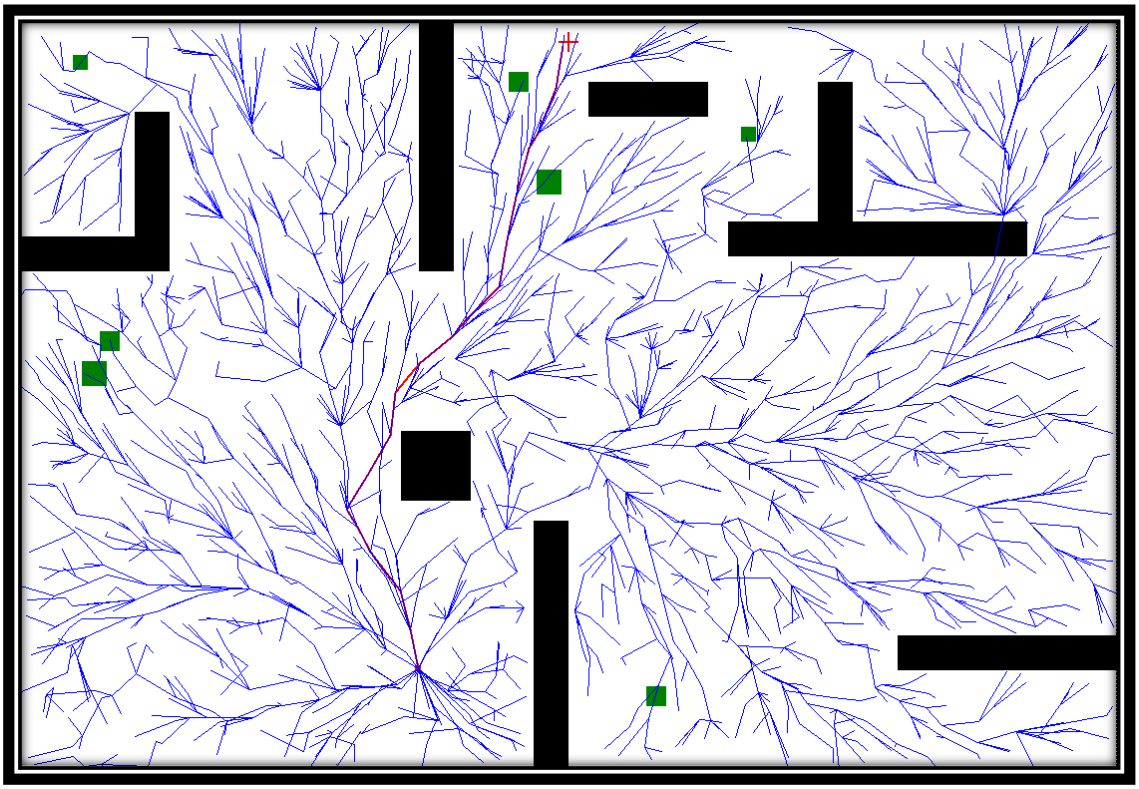
Popis algoritmu:

Algoritmus přijímá dva body. Nový, vkládaný bod a bod jemu nejbližší. V blízkém okolí nového bodu jsou vybrány všechny body. Je porovnána cena cesty ke kořenu stromu. Následně je vybrán bod z množiny blízkých bodů, který splňuje podmínku nejnižší cesty k cíli. Algoritmus `RewireTree` funguje velmi obdobně, jen místo jednoho bodu převíjí strom v okolí všech bodů, které jsou v blízkosti zvoleného bodu. Tato procedura je v cyklu pouštěna bezprostředně po přidání nového bodu.

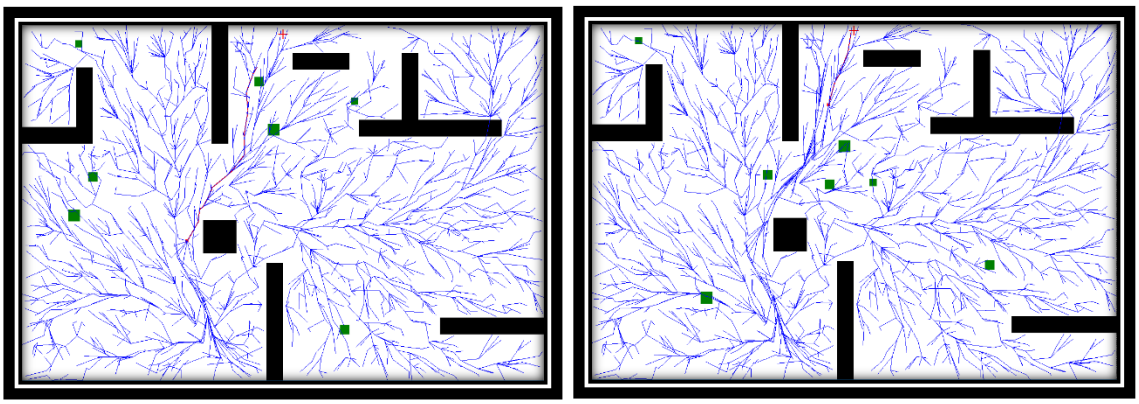


Obr. 23: RRT* po 500 (a), 1 500 (b), 2 500 (c), 5 000 (d), 10 000 (e), 15 000 (f) iteracích. [7]

Simulace RRT*



Obr. 24: Cesta získaná pomocí algoritmu RRT*



Obr. 25: Okamžité přeplánování cesty pří jejím zatarasení

4.5 RT-RRT

Jedná se o první realtimovou implementaci RRT*, nazvanout Real-time RRT*[42]. Tento přístup je kombinací RRT* a Informed-RRT*.

V první fázi je prohledávání prostoru, stejně jako v případě RRT*. Když je cíl nalezen, začíná se agent pohybovat jeho směrem. Během jeho pohybu se algoritmus snaží nalézt řešení bližší optimálnímu. K tomu využívá přístup Informed-RRT*, tedy mezi agentem a Xgoal je vytvořena eliptická oblast. V té generování bodů probíhá intenzivněji, protože v této oblasti je možno nálézt optimální cestu.

Agent i nadále generuje body pro zbytek mapy. To z důvodu, že algoritmus je navržen jako mnoho příkazový (multi query), tedy aby agent při zmně cíle byl schopen reagovat okamžitě. Intenzivnější pokrytí v oblasti kolem agenta je z důvodu lepší reakce na dynamickou překážku.

Hlavní smyčka algoritmu:

```
public override void MainLoop()
	while (!agent.IsInGoal && this.SearchRunning)
		lock (Constants._object)
			Node xNew = ExpandRewire();
			if (xNew != null)
				tree.Insert(xNew);
			RewireFromTreeRoot();
```

Popis algoritmu:

- Hlavní smyčka je podmíněna dosažením cíle anebo uživatelským zastavením programu.
- V ní běží dvě hlavní podprocedury `ExpandRewire` a `RewireFromTreeRoot`.

ExpandAndRewire:

```
private Node ExpandRewire()
	Node xRand = GenerateNewNode(rand.NextDouble());
	Node xClosest = tree.FindClosestNode(xRand);
	Node xNew = Steer(xRand, xClosest);
	if (MapManager.IsPointObstacle(xNew))
		return null;

	if (MapManager.IsPathFree(xClosest, xNew, MapManager.staticObs))
		IReadOnlyList<Node> Xnear = tree.FindClosestNodes(xNew, Density);
	if (Xnear != null && Xnear.Count <= Constants.MaxNumberOfNeighbours)
		AddNodeToTheTree(xClosest, xNew, Xnear);
		Qr.Push(xNew);
	else if (xClosest.Parent != null)
		Qr.Push(xClosest);
		xNew = null;
	RewireRandomNode();
return xNew;
```

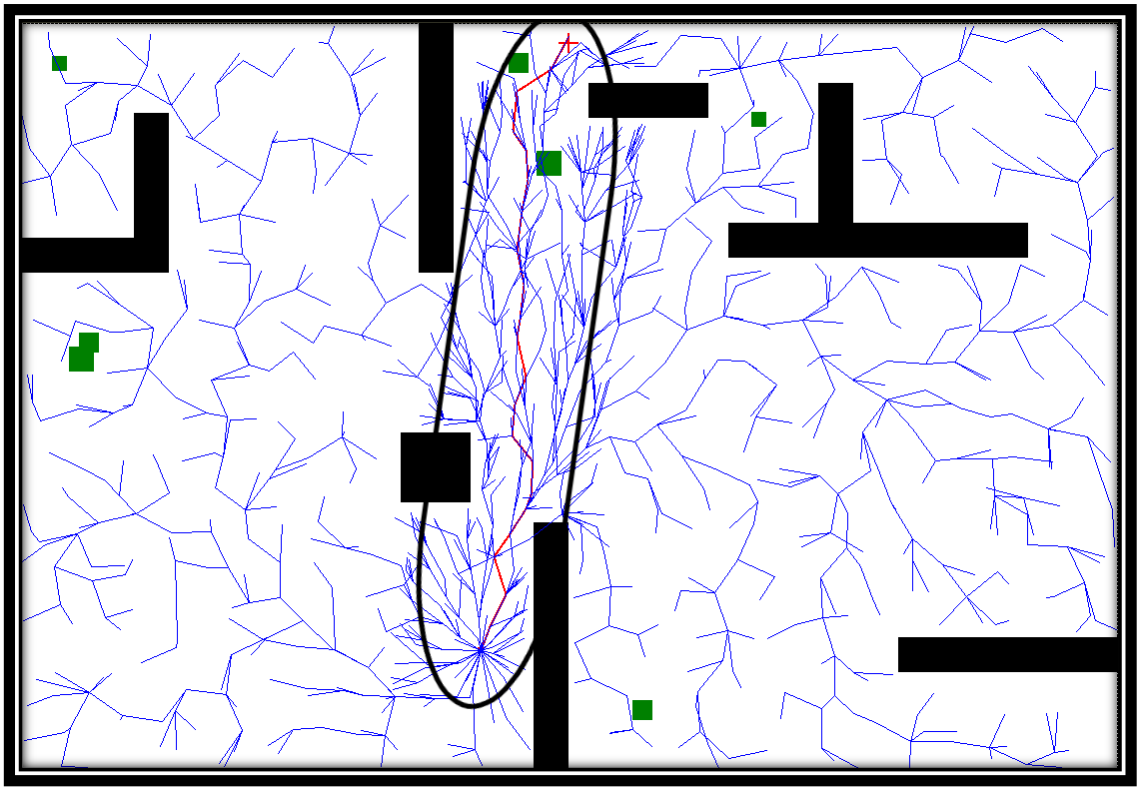
Tento algoritmus se stará o rozšiřování stromu a ověřuje, zda je nově vygenerovaný vrchol x_{New} k dispozici pro plánování cesty. Jeho poslední etapou je, graf v nejbližším okolí tohoto bodu převinout. K tomu slouží metoda `RewireRandomNode`.

RewireFromTreeRoot:

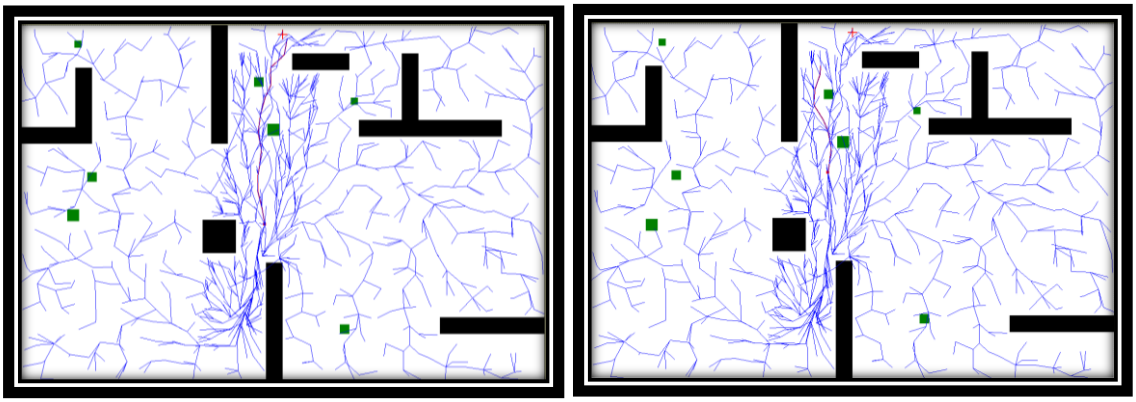
```
protected void RewireFromTreeRoot()
	Queue<Node> Qs = new Queue<Node> { };
	Qs.Enqueue(agent.Root);
	DateTime time = DateTime.Now.Add(new TimeSpan(150));
	while (Qs.Count > 0 && DateTime.Now <= time)
		Node rewiredNode = Qs.Dequeue();

	IReadOnlyList<Node> nodes = tree.FindClosestNodes(rewiredNode, 50);
	foreach (Node node in nodes)
		double distance = Tree.Distance(rewiredNode, node);
		if (distance == 0)
			continue;
		double costOld = Tree.LenghtOfShortestPath(node, agent.Root);
		double costNew = Tree.LenghtOfShortestPath(rewiredNode,agent.Root);
		if (costNew < costOld && MapManager.IsPathFree(rewiredNode, node,
	MapManager.staticObs) && distance < Constants.MAX_EUC_DISTANCE)
			node.Parent = rewiredNode;
			node.CostToParent = distance;
		if (!Qs.Contains(node))
		Qs.Enqueue(node);
```

Algoritmus, podobně jako `RewireRandomNode`, má za úkol převíjet strom. Rozdílem je jen to, že převíjení probíhá z kořenového bodu. Tedy z polohy agenta. Toto převíjení je stejně jako `RewireRandomNode` časově omezené. Předpokládáme totiž, že struktura stromu se narušuje pouze z důvodu pohybu agenta. Tedy v jeho nejbližším okolí.

Simulace RT-RRT*.



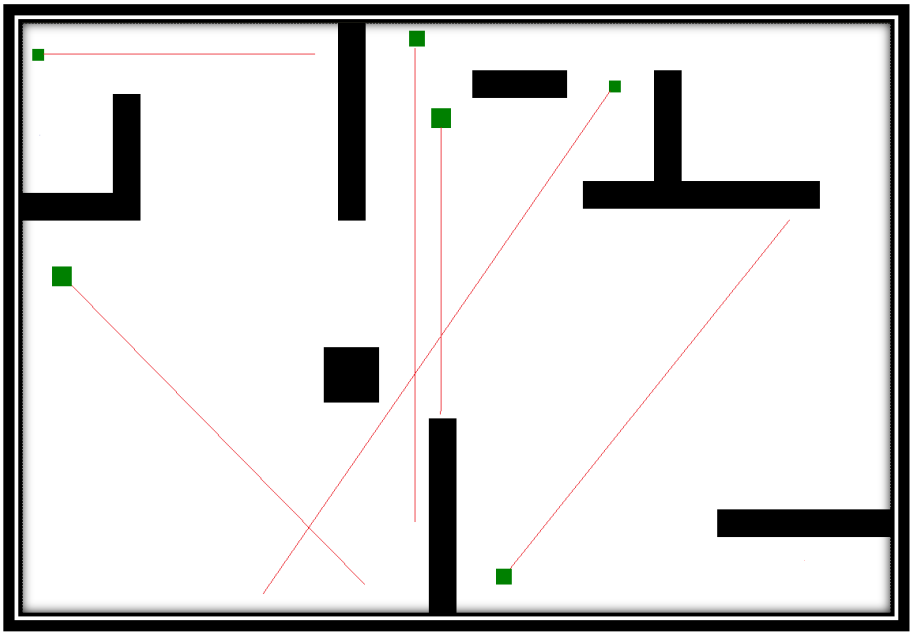
Obr. 26: Cesta získaná pomocí algoritmu RT-RRT*, vyznačená oblast označuje generování bodů v oblasti elipsy, podle algoritmu Informed-RRT*



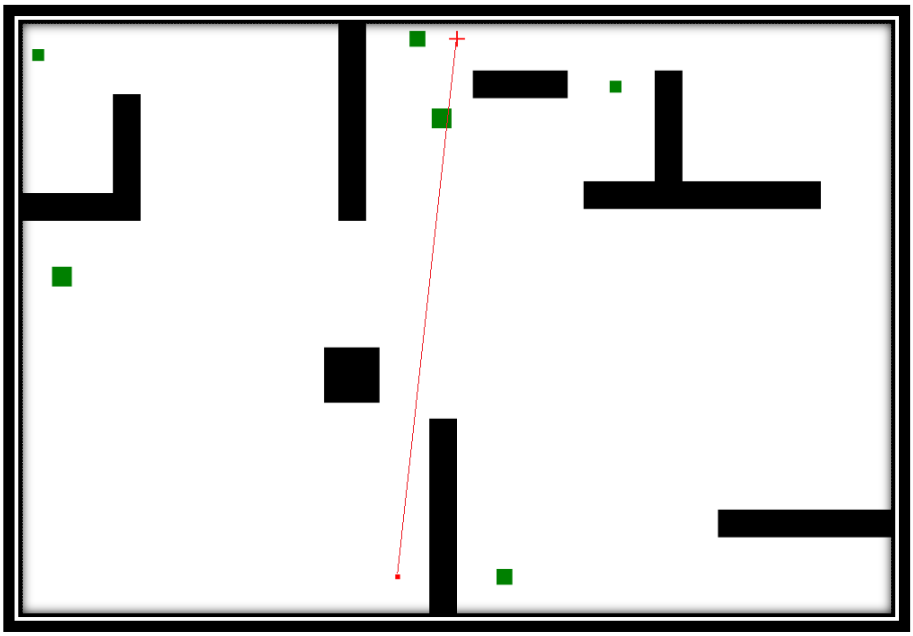
Obr. 27: Na obrázku je zachyceno okamžité přeplánování cesty pří jejím zatarasení.

4.6 Simulace

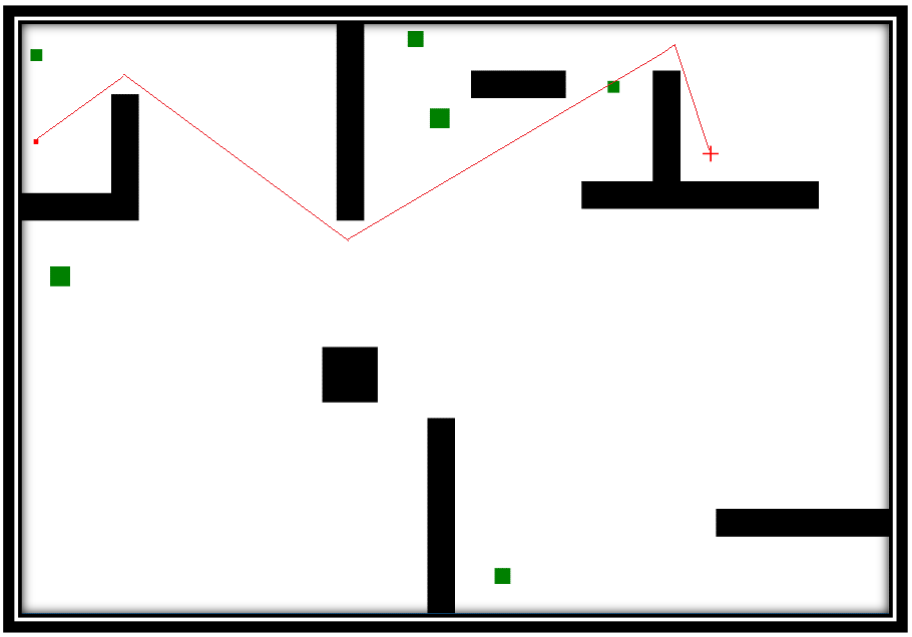
Ve zkušebním prostředí bylo připraveno několik testovacích scénářů a byla provedla simulace. Její výsledky jsou zaznamenány v Tabulce.1. A její zhodnocení je součástí Závěru. Pro algoritmy RRT a RRT* byla v této simulaci stanovena horní hranice 5000 vrcholů. Agent vyrážel k cíli ihned po navázání cesty. Takže k její optimalizaci dochází v jejím průběhu.



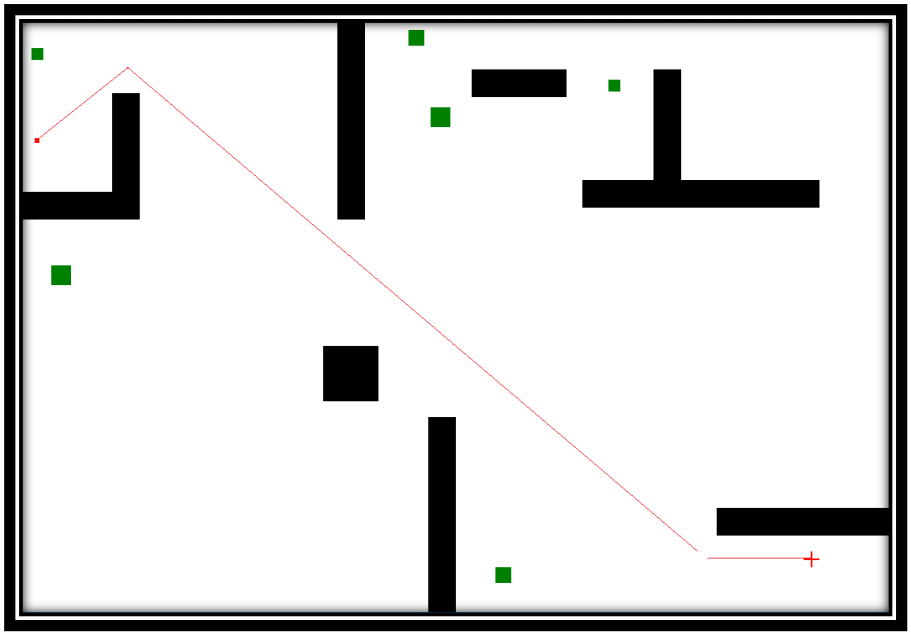
Obr. 28: Trajektorie dynamických překážek.



Obr. 29: První simulační úkol.



Obr. 30: Druhý simulační úkol



Obr. 31: Třetí simulační úkol

Tabulka s výsledky simulace.

Simulace	Ideální cena cesty	Cena cesty pro RT-RRT*		Cena cesty pro RRT*		Cena cesty pro RRT	
	Vzdálenost [-]	Vzdálenost [-]	Čas [s]	Vzdálenost [-]	Čas [s]	Vzdálenost [-]	Čas [s]
1.	700	735	3,8	800	3,5	1130	7,5
2.	1147	1270	6,2	1150	5,9	2090	10,2
3.	1268	1310	6,5	1275	6,4	1604z	15,4

5 ZÁVĚR

Během simulace bylo prokázáno, že navržený přístup přeplánování je aplikovatelný v reálném prostředí. A stejně tak i všechny testované algoritmy.

Z testů vyplývá, že algoritmy RRT* a RT-RRT* jsou pro plánování cesty v dynamickém prostředí stejně vhodné. Dosahovaly stejných časových hodnot i stejné ceny cesty.

Potvrdili jsme, že algoritmus RRT je i v dynamickém prostředí aplikovatelný. Ale jeho reálné použití je velmi sporné. Z porovnávaných algoritmů, má právě RRT nejrychleji vygenerovaný graf. Jeho použití, je ale velmi závislé na dovolené maximální délce kroku a na množství bodů v prostředí. V případě, že dovolený krok je moc dlouhý, může dojít k situaci, že překážka odřízne celou větev stromu a tím znepřístupní agentu část mapy. Kvůli této situaci by bylo nutné vygenerovat nový strom.

Nalezená cesta a ani rychlost dosažení cíle není vzhledem k neoptimalitě řešení nějak překvapivá.

Naopak algoritmy RRT* a RT-RRT* velmi rychle nalezly průchozí cestu a dobře reagovaly na nutnost přeplánování cesty. Cena cesty se i přes nutnost přeplánování blížila optimální.

Z pohledu využití výpočetních prostředků se jeví nejvhodněji RT-RRT*, které k dosažení cíle potřebovalo mnohem méně vygenerovaných bodů. Jeho výhodou je, že elipsa obalující průchozí cestu by se měla rozšiřovat podle šířky nalezené cesty. U časté nutnosti přeplánování toto může být nedostatkem.

Stromová struktura, která byla původně algoritmem vygenerována, se při pohybu robota poškozuje. Tím pádem se postupně komplikuje schopnost přeplánovat pro některé oblasti mapy. Proto by vygenerovaný graf bylo vhodno použít jen pro jedno plánování.

I z tohoto důvodu se jeví, jako vhodnější přístup plánování z cílového bodu, který je po celou dobu plánování statický. Tento přístup by byl vhodnější i pro částečně neznámé prostředí. Protože plánování, ze statického bodu mapy, umožňuje efektivně pracovat s částmi strom, které se v průběhu objevování prostoru projeví, jako nadále nepotřebné, nebo nepoužitelné.

6 SEZNAM POUŽITÉ LITERATURY

[1] WALKER A., Hard real-time motion planning for autonomous vehicles, 2011, Doctor thesis Swinburne University

[2] LAVALLE S. (October 1998). *"Rapidly-exploring random trees: A new tool for path planning"*. Technical Report. Computer Science Department, Iowa State University (TR 98–11).

[3] ŠÍMA, M. Plánování cesty robota ve spojitém prostředí.Brno: Vysoké učení technické v Brně,Fakulta strojního inženýrství, 2006. 63 s. Vedoucí diplomové práce Ing. Petr Krček

[4] BUTTON R. Department of the Navy. The Navy Unmanned Undersea Vehicle (UUV) Master Plan, 2004. ISBN/EAN: 9780833046888;

[5] BOTTON R. Acquisition and Technology Policy Center, and National Defense Research Institute (U.S.). A Survey of Missions for Unmanned Undersea Vehicles. RAND, 2009.

[6] GONZÁLEZ J. Fast Marching Methods in path and motion planning: improvements and high-level applications, Universidad Carlos III de Madrid, 2015

[7] KARAMAN S., FRAZZOLI E. Sampling-based Algorithms for Optimal Motion Planning, International Journal of Robotics Research, Volume 30 Isuue 7, June 2011

[8] DOGLOV D., THRUN S., M. Montemerlo, J. Diebel, Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments, International Journal of Robotics Research, Vol.29, 485-501, 2010.

[9] MARDER-EPPSTEIN E., BERGER E., FOOTE T., GERKEY B., KONILIGE K., The Office Marathon: Robust navigation in an indoor office environment, 2010 IEEE International Conference on Robotics and Automation (ICRA12), 300 -307, Anchorage, Alaska USA, 2010.

[10] GOMÉZ E.Z., Map-building and planning for autonomous navigation of a mobile robot. Center for Research and Advanced Studies of the National Polytechnic Institute, Mexiko, 2015

[11] GERAERTS R,. OVERMARS M. H. Reachability Analysis of Sampling Based Planners. In IEEE International Conference on Robotics and Automation, pages 406–412, 2005

[12] SHEKHAR S., CHWLA S., Spatial Databases: A Tour, Prentice Hall, 2003 (ISBN 0-13-017480-7)

[13] BENTLEY J. L. *(1975). "Multidimensional binary search trees used for associative searching"*. Communications of the ACM. *18 (9): 509.* doi:10.1145/361002.361007.

[14] GUTTMAN A. (1984). *"R-Trees: A Dynamic Index Structure for Spatial Searching"* . Proceedings of the 1984 ACM SIGMOD international conference on Management of data – SIGMOD '84. p. 47. *doi*:*10.1145/602259.602266*. *ISBN 978-0897911283*.

[15] SCHWARTZ J. T., SHARIR M. A., On the 'piano movers' problem I: A case of a twodimensional rigid polygonal body moving amidst polygonal barriers. Commun. Pure Appl. Math., 36:345–398, 1983. , J. T. Schwartz and M. A. Sharir. On the 'piano movers' problem: II. General techniques for computing topological properties of real algebraic manifolds. Advances in applied Mathematics, 4:298–351, 1983.

[16] DIJKSTRA E. W., (1959). *"A note on two problems in connexion with graphs"*. Numerische Mathematik. **1**: 269–271. *doi*:*10.1007/BF01386390*.

[17] HART, P. E.; NILSSON, N. J.; RAPHAEL, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics SSC4*. 4 (2): 100–107. doi:10.1109/TSSC.1968.300136

[18] AUDSLEY N.C., BURNS A., DAVIS R.I., SCHOLEFIELD D.J., WELLINGS A.J., Integrating Optional software Components into Hard Real-time Systems. Software Engineering Journal, 11(3):133–140, 1996.

[19] LIKHACKEY M., FERGUSON D., GORDON G., STENTZ A., THRUN S., Anytime Dynamic A*: An Anytime, Replanning Algorithm. In Proceedings of the International Conference on Automated Planning and Scheduling, 2005.

[20] FERGUSON D., STENTZ A. Anytime RRTs. In IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 5369–5375, 2006.

[21] BERG J., FERGUSON D., KUFFNER J. Anytime path planning and replanning in dynamic environments. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pages 2366–2371, 2006.

[22] A. STANKOVIC J. A., Misconceptions about real-time computing: A serious problem for next-generation systems. Computer, 21(10):10–19, 1988.

[23] KORF R. E., Real-time heuristic search. Artificial Intelligence, 42(2-3):189–211, 1990.

[24] LIU C. L., LAYLAND J. W., Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM, 20(1):46–61, 1973.

[25] HSU D.. Randomized Single-Query Motion Planning in Expansive Spaces. PhD thesis, Stanford University, 2000

[26] EDGAR S. F., Estimation of Worst-Case Execution Time Using Statistical Analysis. PhD thesis, York University, 2002.

[27] PETTERS S. M., Comparison of trace generation methods for measurement based wcet analysis. In 3nd International Workshop on Worst Case Execution Time Analysis, Porto, Portugal, 2003.

[28] PETTERS S. M., Worst Case Execution Time Estimation for Advanced Processor Architectures. PhD thesis, Institute for Real–Time Computer Systems, Technische Universit¨at M¨unchen, Munich, Germany, September 2002.

[29] LEONARD J.J., DURRANT-WHYTE H.F., Simultaneous map building and localization for an autonomous mobile robot. IEEE/RSJ International Workshop on Intelligent Robots and Systems, 3:1442–1447, 1991.

[30] KHATIB O. Real-time obstacle avoidance for manipulators and mobile robots. In Proceedings of the IEEE International Conference on Robotics and Automation, volume 2, pages 500–505, 1985.

[31] BODDY M. S., Anytime Problem Solving using Dynamic Programming. In Proceedings of the Ninth National Conference on Artificial Intelligence, pages 738–743, 1991.

[32] RODNEY A., BROOKS. A robot that walks; emergent behaviors from a carefully evolved network. Neural Comput., 1:253–262, June 1989.

[33] LATOMBE J., Latombe. Robot Motion Planning. Kluwer Academic Publishers, 1991., ISBN - 978-1-4615-4022-9

[34] AURENHAMMER F., *Voronoi diagrams - a survey of a fundamental geometric data structure*, ACM Computing Surveys, pp. 345-405, vol. 23(3), 1991, Topology-Oriented Incremental Computation of Voronoi Diagrams of Circular Arcs and Straight-Line Segments Martin Held, Stefan Huber Comp. Aided Design pp. 327--338, vol. 41(5), May 2009

[35] *DELAUNAY B.*, (1934). *"Sur la sphère vide"*. Bulletin de l'Académie des Sciences de l'URSS, Classe des Sciences Mathématiques et Naturelles. **6**: 793–800.

[36] HELD S., STEFAN H., PALFRADER P., Generalized Offsetting of Planar Structures Using Skeletons, Comp. Aided Design & Appl. pp. 712--721, vol. 13(4), March 2016

[37] Robotic motion planing [cit. 2019-05-24] Dostupné z: https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/robotics/basicmotion.html

[38] STENTZ, ANTHONY (1994), "Optimal and Efficient Path Planning for Partially-Known Environments", *Proceedings of the International Conference on Robotics and Automation*: 3310–3317, CiteSeerX 10.1.1.15.3683

[39] KOENIG, S.; LIKHACHEV, M. (2005), "Fast Replanning for Navigation in Unknown Terrain", *Transactions on Robotics*, 21 (3): 354–363, CiteSeerX 10.1.1.65.5979, doi: 10.1109/tro.2004.838026

[40] KAVRAKI L. E.; ŠVESTKA P.; LATOMBE J.-C.; Overmars, M. H. (1996), "Probabilistic roadmaps for path planning in high-dimensional configuration spaces", *IEEE Transactions on Robotics and Automation*, 12 (4): 566–580, doi:10.1109/70.508439

[41] ARSLAN O., TSIOTRAS P., Use of Relaxation Methods in Sampling-Based Algorithmsfor Optimal Motion Planning. 2014, Electronic ISBN: 978-1-4673-5643-5

[42] GAMMELL D., SRINIVASA S., BARFOOT D., Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic

[43] M. Otte, E. Frazzoli. RRTx: Real-Time Motion Planning/Replanning for Environments with Unpredictable Obstacles, 2014, DOI:10.1007/978-3-319-16595-0_27

[44] Arslan O., BERNTORP K., TSIOTRAS P., Sampling-based Algorithms for Optimal Motion PlanningUsing Closed-loop Prediction, 2016, arXiv:1601.06326

[45] KUFFNER J., LAVALLE S. M.: *Space-filling Trees*, The Robotics Institute, Carnegie Mellon University, CMU-RI-TR-09-47, 2009. *CMU-RI-TR-09-47*

[46] NADERI K., RAJAMAKI J., HAMALAINEN P., RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*, 2015, DOI: 10.1145/2822013.2822036

[47] LAVALLE S. Sampling-Based Motion Planning, 2006, Dostupné z: http://planning.cs.uiuc.edu/