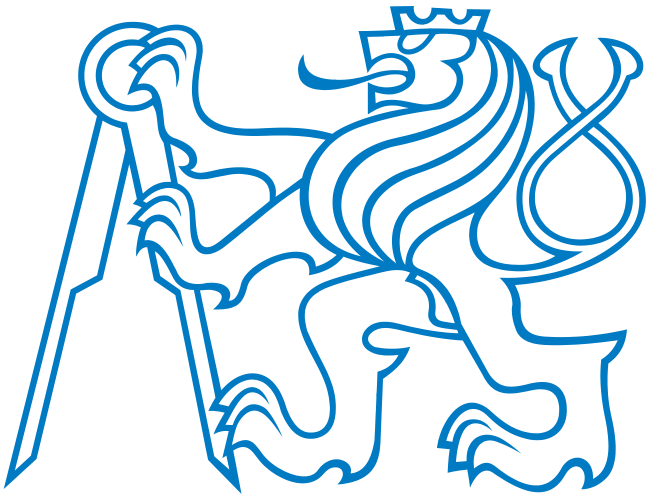
České Vysoké Učení Technické v Praze
Fakulta Elektrotechnická

BAKALÁŘSKÁ PRÁCE



Petr Vaněk

Plánování pohybu technikami RRT

Katedra kybernetiky
Vedoucí bakalářské práce: **Ing. Jan Faigl**

Praha, 2010

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, SW, projekty atd.) uvedené v přiloženém seznamu.

V Praze dne 28. 5. 2010 .....

.....
podpis

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Petr V a n ě k

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný

Obor: Kybernetika a měření

Název tématu: Plánování pohybu technikami RRT

Pokyny pro vypracování:

1. Seznamte se s úlohou plánování pohybu metodou náhodných rychle rostoucích stromů (Randomized Rapidly Growing Trees – RRT) [1].
2. Seznamte se s aplikací konceptu realizovatelnosti viability v metodách RRT [2].
3. Seznamte se s knihovnami MPNN, RAPID používané pro plánování pohybu a vizualizačním prostředím VTK.
4. Implementujte základní varianty algoritmu RRT: RRT, RRTConnect, RRTBidirect, RRT-Blossom a rozšířenou variantu s uvažováním konceptu viability RRT-BlossomVF.
5. Implementované algoritmy porovnejte s algoritmem KPIECE [3] z knihovny OMPL (Open Motion Planning Library).
6. Algoritmy aplikujte pro řešení problému Alpha Puzzle [4] případně jiných problémů.

Seznam odborné literatury:

[1] Steven M. Lavalle: Planning Algorithms. Cambridge University Press, May 2006.

[2] Maciej Kalisiak: Toward More Efficient Motion Planning with Differential Constraints. PhD thesis, University of Toronto, 2007.

[3] Ioan Alexandru Sucan and Lydia E. Kavraki: Kinodynamic Motion Planning by Interior-Exterior Cell Expolaration. In International Workshop on the Algorithmic Foundations of Robotics, Guanajuato, Mexico, 2008.

[4] Motion planning puzzles – http://parasol.tamu.edu/dsmft/benchmarks/mp

Vedoucí bakalářské práce: Ing. Jan Faigl

Platnost zadání: do konce zimního semestru 2010/2011

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

L.S.

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 3. 2. 2010

Abstrakt

Bakalářská práce se zabývá problémem plánování trajektorií mobilního robotu technikami RRT (Rapidly-exploring Ranodm Trees). Kromě základní RRT techniky jsou představeny modifikace RRT-Bidirect, RRT-Connect, RRT-Blossom a RRT-Viability. Tyto algoritmy byly implementovány a experimentálně ověřeny v úloze plánování trajektorie pro dva základní modely robotů (diferenciální a *car-like*) v různých prostředí. V práci je také uvedena plánovací technika KPIECE, která je technice RRT velmi podobná, avšak místo náhodného vzorkování je použito deterministického výběru buňky (části stavového prostoru), ve které je teprve náhodně vzorkováno. Mimoto byly popsané RRT techniky aplikovány v úloze *Alpha Puzzle*, který představuje standardní problém používaný pro porovnání různých plánovacích technik. Charakter této úloh vyžaduje 3D vizualizace plánovací scény, proto je v práci stručně představena použitá vizualizační knihovna VTK. V závěru práce jsou shrnuty dosažené výsledky a nastíněny možné směry dalšího studia plánovacích technik.

Abstract

The thesis deals with the motion planning problem for a mobile robot, particularly using RRT techniques (Rapidly-exploring Random Trees). Beside the basic RRT variant, several modifications are presented: RRT-Bidirect, RRT-Connect, RRT-Blossom a RRT-Viability. These algorithms have been implemented and experimentally verified in a set of motion planning tasks for two basic models of robots (differential and car-like) and in various environments. In addition, the KPIECE planning technique is described. KPIECE is similar to RRT, but instead of direct random sampling it uses a deterministic selection of a cell (a part of the state-space) in which a random configuration is sampled. The described RRT techniques have been applied in the *Alpha Puzzle* task, which represented a standard problem used in motion planning techniques benchmarking. The nature of this task requires 3D visualization, thus the used visualization library VTK is briefly described in the thesis. A summary of the results and possible future work is presented in the conclusion.

Především bych chtěl poděkovat panu Ing. Janu Faiglovi za jeho věcné připomínky, rady, korekce a pomoc se zpracováním bakalářské práce, jehož zásluhou nabyla konečné formy, která je zde předvedena.

Dále bych chtěl poděkovat svým rodičům za trpělivost a ochotu podporovat mě při studiu.

Obsah

Seznam obrázků

Názvosloví

$\mathcal{W}$	Libovolný prostor např. $\mathbf{R}^2$.
$\mathcal{O}$	Překážka.
$\mathcal{A}$	Pevný robot.
$\mathcal{A}(q)$	Robot v nějaké konfiguraci.
q	Konfigurace robotu.
$\mathcal{C}$	Konfigurační prostor.
$\mathcal{C}_{obs}$	Prostor překážek.
$\mathcal{C}_{free}$	Volný prostor.
X	Stavový prostor.
X_{obs}	Prostor překážek jako podmnožina stavovém prostoru.
X_{free}	Volný stavový prostor.
x_{init}	Počáteční stav. Stav ze kterého je hledána trajektorie.
x_{goal}	Cílový stav. Stav do kterého je hledána trajektorie.
x_{rand}	Náhodně vygenerovaný stav.
x_{near}	Nejbližší stav k nějakému jinému stavu.
δ	Vzdálenost v používané metrice.
u	Konkrétní hodnota akčního vstupu robotu.
$\mathcal{U}$	Všechny možné akční vstupy robotu.
ε	Okolí cílového bodu.
T	Strom.
$\mathcal{E}(X)$	Projekce stavového prostoru.

Úvod

Plánování pohybu je část robotiky, která se zabývá prohledáváním stavového prostoru. Pro tento účel je možné si představit stavový prostor jako sadu možných transformací, které je robot schopen vykonat. Takový stavový prostor reprezentuje možné konfigurace robotu, např. x, y, φ, a v literatuře věnované plánování pohybu [7] je označován jako *konfigurační prostor* a značen $\mathcal{C}$.

Tento prostor přináší důležitou abstrakci pro řešení plánování pohybu komplikovaných modelů a transformací. Touto abstrakcí může být vyřešeno mnoho problémů, které spolu na první pohled nemusí navzájem souviset jak geometricky tak i kinematicky. Tyto problémy mohou být i přes jejich odlišnost vyřešeny identickou plánovací technikou. Z tohoto důvodu je tato úroveň abstrakce velice důležitá. Příkladem může být plánování pohybu pro dosažení cíle automobilem nebo umisťování molekul v medikamentech založených na požadovaných biologických strukturách při návrhu léků [15].

V souvislosti s reálným prostředím je možné konfigurační prostor rozdělit na *prostor překážek* [9], ve kterém je robot v kolizi s překážkou a pro který platí

$$\mathcal{C}_{obs} \subseteq \mathcal{C}. \tag{1}$$

Překážkou může být libovolný objekt, ve kterém by mohlo dojít k poškození robotu, nebo konfigurace, ve kterých by se robot neměl nacházet. Za předpokladu, že nějaký prostor $\mathcal{W}$ obsahuje překážky, $\mathcal{O} \subset \mathcal{W}$ a je zde nějaký pevný robot, $\mathcal{A} \subset \mathcal{W}$ jehož konfiguraci můžeme popsat jako $q \in \mathcal{C}$, kde např. $q = (x, y, \varphi)$ pro $\mathcal{W} = \mathbf{R}^2$, potom prostor překážek můžeme vyjádřit jako

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} | \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}. \tag{2}$$

Volný prostor je podprostor konfiguračního prostoru, ve kterém robot není v kolizi s žádnou překážkou a platí pro něj

$$\mathcal{C}_{free} = \mathcal{C} \backslash \mathcal{C}_{obs}. \tag{3}$$

Za předpokladu, že $\mathcal{C}$ je topologický prostor a $\mathcal{C}_{obs}$ je uzavřená množina, potom $\mathcal{C}_{free}$ je otevřená množina. Tato definice zahrnuje i takové konfigurace, při kterých se může robot libovolně přiblížit k překážce, dokud je v $\mathcal{C}_{free}$. Tedy pokud se $\mathcal{A}$ dotýká $\mathcal{O}$ a platí tato rovnice

$$\mathcal{O}^O \cap \mathcal{A}(q)^O = \emptyset \Leftrightarrow \mathcal{O} \cap \mathcal{A}(q) \neq \emptyset, \tag{4}$$

kde $\mathcal{O}^O$ značí vnitřek množiny $\mathcal{O}$, potom je $q \in \mathcal{C}_{obs}$. V praktické robotice může být myšlenka takového přiblížení k překážce nesmyslná [9]. V plánování se pak uvažuje množina reprezentující volný prostor jako uzavřená.

Plánování pohybu je prováděno v *nekonvexním prostoru* (tj. prostor, ve kterém nemůžeme dva jeho libovolné body propojit úsečkou tak, aby byla bez zbytku součástí tohoto prostoru), a proto je snahou plánovacích technik hledat takovou *trajektorii*, která bude v $\mathcal{C}_{free}$. V literatuře bývá často pojem trajektorie zaměňován s cestou. V této práci je cesta chápána jako určitá sekvence bodů propojených hranami, zatímco trajektorie je časové uspořádání stavů dynamických systémů. Tedy hlavní rozdíl mezi trajektorií a cestou je takový, že trajektorie je funkcí času. Při plánování pohybu je úkolem najít pro robot takovou trajektorii, na které není povoleno, aby byl v kolizi s jakoukoliv překážkou. To znamená, že pro libovolnou konfiguraci q_i na této trajektorii platí

$$q_i \in \mathcal{C}_{free}. \tag{5}$$

Tedy základním *plánovacím problémem* je najít robotu trajektorii z nějakého počátečního stavu x_{init} do cílového stavu x_{goal} aniž by byli porušeny zákonitosti pohybu robotu a další omezení jako je kolize s překážkami, rovnováha (např. u kola) nebo limity ohybu kloubů. Trajektorie splňující tyto omezení je označována jako *uskutečnitelná trajektorie*, která může být navíc ještě omezena dobou potřebnou na projetí této trajektorie nebo rychlostním profilem robotu.

Plánování pohybu se nevyužívá pouze pro pohyb robotu v reálných prostředích, ale i ve virtuálních, jako jsou počítačové hry, při generování počítačových animací do filmů a pro digitální návrh technických zřízení. Uplatnění nalezli i ve vzdálenějších odvětvích, jako je návrh léků, a při návrhu složení bílkovin.

Z předchozího textu vyplývá, že plánování pohybu může být použito v různých typech úloh. Pro každou úlohu může být vhodná jiná plánovací technika. Jednou z nich je RRT (Rapidly-exploring Random Tree) technika, která může být označena jako univerzální plánovací technika. Studiu několika RRT variant je věnována tato bakalářská práce.

Kapitola 1

Rychle náhodně rostoucí stromy

V říjnu 1998 vydal Steven M. LaValle článek o plánovací technice RRT (Rapidly-exploring Random Tree) [8], která ihned vzbudila zájem odborné komunity. Technika RRT je v článku představena jako náhodně vytvořená datová struktura, navržená pro širokou škálu problémů plánování trajektorií. Tato technika sdílí několik vlastností s (v té době již dobře známou) technikou PRM (Probabilistic RoadMap). Obě jsou navrženy s jednoduchou heuristikou, obě rychle a rovnoměrně prohledávají stavový prostor [10] a navíc jsou obě základní techniky relativně jednoduché, což ulehčuje analýzu složitosti. Výhodou techniky RRT je, že nepotřebuje žádná propojení mezi jednotlivými konfiguracemi, nebo stavy, kterých PRM typicky potřebuje tisíce. V neposlední řadě může být RRT více efektivní než PRM při plánování trajektorií pro *holonomní* roboty.

Vznik RRT byl motivován potřebou *kinodynamického* plánování trajektorií pro *neholonomní* roboty. Neholonomní robot je takový, u kterého nemůžeme kompletně zintegrovat rovnici popisující jeho konfiguraci a pohyb $f(q, \dot{q}, t) = 0$. Naopak pojem holonomní označuje robot u něhož je tuto rovnici možné zintegrovat [9].

Kinodynamické plánování se snaží uvažovat kinematická a dynamická omezení, jakými jsou vyhýbání překážkám, rychlost nebo akcelerace [2]. Při kinodynamickém plánování musí být vyřešeny problémy kinematických omezení, kterými mohu být například mechanická omezení kloubů a dynamická omezení, jako jsou omezení rychlosti nebo zrychlení. Tedy základním problémem je najít takovou trajektorii, která začíná v počátečním stavu, končí v cílovém stavu a přirozeně jsou při tom splněny všechna požadovaná kinodynamická omezení. Pohyb po této trajektorii je proveden v časovém intervalu $\langle 0, T_f \rangle$.

Pro účel kinodynamického plánování je vhodné definovat *stavový prostor* X, ve kterém je každý stav $x \in X$ definován jako $x = (q, \dot{q})$, pro $q \in \mathcal{C}$. Výhodou tohoto stavového prostoru je, že má v sobě zahrnuty konfigurační i rychlostní parametry. V podstatě je to tečna v konfiguračním prostoru [10]. Obdobně jako v rovnici (3), je volný stavový prostor definován jako $X_{free} = X \backslash X_{obs}$. V tomto stavovém prostoru je robot řízen nějakými vstupními parametry. Potom jeho pohyb může být popsán diferenciální rovnicí ve tvaru

$$\dot{x} = f(x, u), \tag{1.1}$$

kde $u \in \mathcal{U}$ a $\mathcal{U}$ je množina všech možných řídicích vstupů. Zde se předpokládá, že f je spojitá funkce. Nový stav x_{new} lze získat z počátečního stavu x a řídicího vstupu

$u \in \mathcal{U}$ integrací funkce f přes časový interval Δt. Například Eulerovo integrací bude získán nový stav z rovnice

$$x_{new} \approx x + f(x, u)\,\Delta t. \tag{1.2}$$

Pro větší přesnost může být použita integrační metoda Runge-Kutta.

Od svého vzniku byla technika RRT úspěšně aplikována v celé řadě problémů plánování trajektorií pro roboty s kinodynamickými vlastnostmi [`http://msl.cs.uiuc.edu/rrt/projects.html`]. Tato technika může být považována jako generátor monotónních trajektorií nelineárních systémů se stavovými omezeními. Samotné RRT bývá nedostačující k vyřešení plánovacích problémů, a proto je používáno jako součást různých plánovacích systémů. Díky těmto vlastnostem se dá tato technika aplikovat na různé druhy plánovacích úloh.

V dalších částech této kapitoly je popsán základní algoritmus a jeho adaptace, které vylepšují některé jeho vlastnosti.

1.1 Základní technika

Technika RRT vytváří stromovou datovou strukturu reprezentující jednotlivé navzorkované stavy. Tento strom je vytvořen ze stavů, které jsou do něho postupně přidávány. Základním principem techniky RRT je vygenerování náhodného stavu x_{rand} a k tomuto stavu je snahou rozšířit strom. Je-li nalezen stav, kterým je možné strom rozšířit, je po přidání do stromu, změřena jeho vzdálenost od cílového stavu x_{goal}. V případě, že je tato vzdálenost menší než zvolené ε, podařilo se nalézt výslednou trajektorii, která je ze stromu snadno rekonstruovatelná. Pokud je tato vzdálenost větší než zvolené ε následuje opakování generování náhodného stavu a rozšiřování stromu. Stavy ve stromu musí splňovat $x_i \in X_{free}$, zároveň přechod ze jednoho stavu do následujícího stavu stromu musí respektovat příslušná omezení pohybu robotu (rychlosti, zrychlení). U této techniky je obtížné, aby výsledná trajektorie vedla přesně z počátečního stavu x_{init} do cílového x_{goal}, proto vhodná volba tolerančního pásma ε zvyšuje rychlost nalezení cesty.

Algoritmus 1 naznačuje základní princip RRT techniky. Kořen stromu je vytvořen z počátečního stavu x_{init}. Po této inicializaci následuje hlavní cyklus algoritmu RRT. Počet iterací tohoto cyklu může být zadán jako maximální počet iterací nebo jako časové kvantum, ve kterém musí být plánovací úloha vyřešena. Druhý případ bývá součástí plánování v reálném čase, kde musí být plánování provedeno v určitém časovém intervalu.

Při kinodynamickém plánování pohybu narážíme na několik problémů. Plánování bývá prováděno v prostoru s mnohem větším počtem dimenzí než je dvou nebo tří dimenzionální eukleidovský prostor. To může způsobit komplikace při výběru nejbližšího stavu, jelikož čím je počet dimenzí vyšší, tím je časová náročnost jeho výběru složitější. Proto zde není klíčové mít absolutně nejbližší stav. V anglické literatuře je problém výběru nejbližšího stavu označován jako *nearest neighbour*. Pro efektivní nalezení nejbližšího stavu ve více dimenzionálním prostoru je možné použít strukturu KD-stromu, jejíž využití v RRT představila Anna Yerhova v článku [16]. Další komplikací je, že strom musí růst ve volném stavovém prostoru. To může způsobit

Algoritmus 1: RRT

Vstup: Počáteční stav x_{init}, cílový stav x_{goal}.
Výstup: Výsledná trajektorie pokud je nalezena.

```
T = treeInit(x_init)   // inicializace stromu; z x_init je vytvořen kořen stromu
for i = 0 to max do
    x_rand = getRandomState()        // generování náhodného stavu
    x_new = growTree(T, x_rand)
    if x_new ∧ δ(x_new, x_goal) < ε then // δ vypočítá vzdálenost mezi dvěma stavy v používané metrice
        return extractSolution(x_new)   // extrahuje ze stromu výslednou trajektorii
return failed
```

problém při průjezdu místy, kde je robot těsně obklopen překážkami. V anglické literatuře se tento problém nazývá *Narrow Passage*. Problém hledání nejbližšího souseda je vlastně problém nalezením vhodné metriky. Z praktického pohledu však vyřešení tohoto problému nepomůže, neboť výpočet ideální metriky je stejně namáhavý jako vyřešení původního plánovacího problému [10].

V algoritmu 2 je naznačen způsob jakým je expandován strom. Ve stromu je nalezen k náhodně vygenerovanému stavu x_{rand} jeho nejbližší soused x_{near}. Z tohoto stavu se vyhledává stav s nejkratší vzdáleností k x_{rand} funkcí *pickCtrl*. Je-li takový stav nalezen, bude tento stav přidán do stromu. Pro lepší představu je výsledek přidání stavu do stromu znázorněn na obrázku 1.1.

Algoritmus 2: growTree

Vstup: Strom T, náhodný stav x_{rand}.
Výstup: Nejlepší expandovaný stav směrem k x_{rand}.

```
x_near = findNearestNeighbour(T, x_rand) // nalezení nejbližšího souseda
x_best = pickCtrl(x_near, x_rand)
if x_best then
    T → addPoint(x_best)             // přidá stav do stromu
    return x_best
return ∅
```

Způsob výběru nejvhodnějšího stavu (funkce pickCtrl) je uveden v algoritmu 3. Zde je snahou expandovat stav x_{near} přes všechny vstupy z množiny $\mathcal{U}$. Expandované stavy, které vedou na kolizi s překážkou, nejsou dále uvažovány. Dále je zde vhodné vyřadit ty stavy, které nesplňují dynamická omezení. Výstupem algoritmu je stav expandovaný z x_{near} po aplikování řídicího vstupu $u \in \mathcal{U}$, se vzdáleností k náhodnému stavu menší než je vzdálenost mezi x_{near} a x_{rand}.

Výhodou takovéto implementace výběru dalšího stavu je, že strom rychle expanduje v několika málo směrech a teprve až po několika iteracích se strom začne

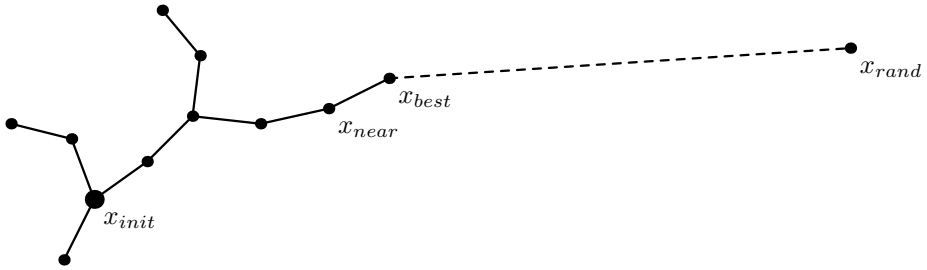
Algoritmus 3: pickCtrl

Vstup: Nejbližší stav x_{near}, náhodný stav x_{rand}.
Výstup: Nejlepší expandovaný stav směrem k x_{rand}.

```
1  d_min = δ(x_near, x_rand)        // δ vypočítá vzdálenost mezi dvěma stavy
   v používané metrice
2  foreach u ∈ U do
3      x = expand(x_near, u)        // expanduje stav x_near pomocí řídicího
       vstupu u
4      if isInCollision(x) then // kontrola zda je x v kolizi s překážkou
5          continue
6      d = δ(x, x_rand)
7      if d < d_min then
8          d_min = d
9          x_best = x
10 if x_best then
11     return x_best
12 return ∅
```

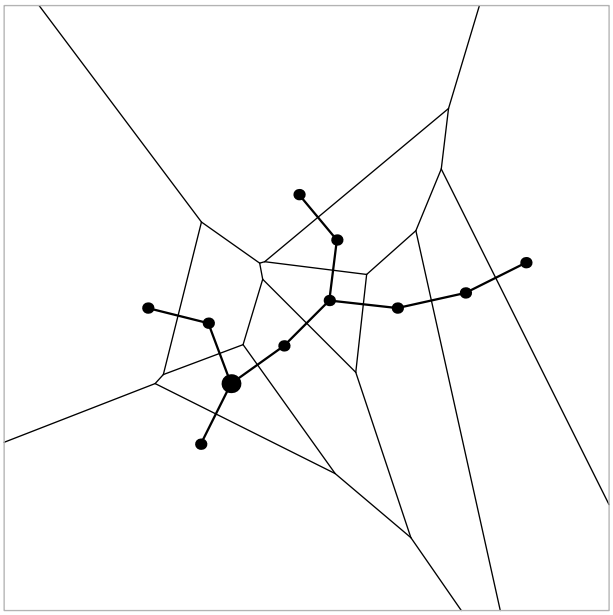
více větvit. Tato vlastnost umožňuje rychlé a efektivní prohledávání stavového prostoru.

Postup tvoření RRT stromu může být dobře znázorněn Voroného diagramem. Na obrázku 1.2 je takový diagram pro strom z obrázku 1.1. Veliké Voroného buňky odpovídají stavům na okraji stromu. Jelikož nejvhodnější stavy se pro náhodný stav vybírají metodou nejbližšího souseda, budou s vysokou pravděpodobností vybrány právě ty stavy ležící ve velikých Voroného buňkách. Tato vlastnost umožňuje, že strom roste především do neprozkoumaných částí prostoru. Po přidání nových stavů do stromu jsou veliké Voroného buňky štěpeny na menší, a tím je zaručeno rovnoměrné prohledávání prostoru.



Obrázek 1.1: Znázornění růstu stromu z kořene x_{init}. Nejprve je vygenerován náhodný stav x_{rand}. Dále je k tomuto stavu nalezen nejbližší soused x_{near} a ten je expandován směrem k náhodnému stavu. Nejlepší expandovaný stav x_{best} je přidán do stromu. *Obrázek je adaptován z [6]*

Přestože je základní RRT technika plánování pohybu úplná, může být nalezení cílového stavu časově náročné. Mnohem lepších výsledků může být dosaženo, pokud bude základní technika obohacena informací o umístění základního stavu.



Obrázek 1.2: Voroného diagram pro strom z obrázku 1.1. Z tohoto obrázku je vidět, že na okrajích stromu jsou Voroného buňky větší.

1.1.1 Goal Bias

To, že je algoritmus informován o cílovém stavu se v anglické literatuře nazývá *Goal Bias*. Informování se provádí tak, že každou k-tou iteraci je namísto generování náhodného stavu použit stav cílový. Podle prostředí, ve kterém je plánování prováděno, je vhodné určit jaká bude míra informovanosti. V prostředích s malým počtem překážek nebo úplně bez překážek může vést vysoká míra informovanosti k rychlému nalezení cílového stavu. Naopak v prostředích s velikým počtem překážek by vysoká míra informovanosti mohla způsobovat opakované expandování stavů do překážek. Vhodná volba míry této informovanosti může výrazně zkrátit dobu nalezení cílového stavu. Jelikož nelze přesně určit jak bude vhodná určitá míra informovanosti v nějakém prostředí, je její volba složitá, a proto záleží na zkušenostech operátora, který tuto hodnotu do plánovače vkládá.

Tato modifikace základní techniky může v řadě případů přispět k mnohem lepším výsledkům, ale existují prostředí, ve kterých by naopak mohla uškodit. Jedním příkladem takových prostorů může být plánování trajektorie pro přejezd mezi dvěma místnostmi oddělených chodbou, jako je například znázorněno na obrázku 1.3. Pro takové prostředí se spíše hodí plánování trajektorií technikou, ve které paralelně rostou dva stromy. Tato technika je popsána v následujícím oddíle.

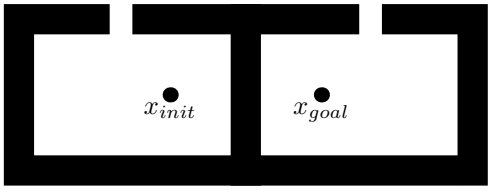
1.2 RRT-Bidirect

Jednou z možných modifikací základní RRT techniky je paralelní růst dvou stromů a v literatuře bývá označována jako RRT-Bidirect. První strom má kořen v x_{init} stejně jako u základního algoritmu RRT. Druhý strom začíná v x_{goal} a je snahou nalézt výslednou trajektorii vhodným propojením obou stromů. Tato technika

zlepšuje vlastnosti především při prohledávání prostorů podobných obrázku 1.3. Je to tím, že čím jsou oba stromy rozrostlejší tím poskytují jeden druhému mnohem větší možnost nalezení dvou společných stavů, které jsou od sebe vzdáleny méně než zvolené ε. Tím, že cílem je nalezení vhodného stavu z větší množiny stavů, je pravděpodobnější, že bude výsledná trajektorie nalezena dříve.

První zmínku o této technice vydal S. M. LaValle a J. J. Kuffer v článku [10]. V tomto článku je tato technika implementována tak, že je vygenerován náhodný stav x_{rand} a k němu je snahou rozšířit strom s kořenem v x_{init} o stav x_i. Podaří-li se to, hledá se ke stavu x_i stav ze stromu s kořenem v x_{goal} se vzdáleností menší než zvolené ε. Pokud je takový stav nalezen, je také nalezena výsledná trajektorie. Není-li nalezen, je snahou rozšířit ke stavu x_{rand} strom s kořenem v x_{goal} o stav x_g. Je-li toto rozšíření úspěšné, je snahou ve stromu s kořenem v x_{init} najít stav, který se nachází v kratší vzdálenosti než zvolené ε od stavu x_g. Podaří-li se takový stav naleznout, je nalezena i výsledná trajektorie. V případě nezdaru je tento postup opakován v další iteraci.

V této práci je však uvedena implementace, kterou použil M. Kalisiak ve své disertační práci [4]. Rozdíl mezi implementacemi je především v tom, jak se rozrůstají oba stromy. Zatímco v implementaci podle S. M. LaValle je snahou oba stromy rozšířit směrem k x_{rand}, je v implementaci M. Kalisiaka k náhodnému stavu x_{rand} rozšiřován o stav x_i pouze jeden strom a je-li tento rozšiřující stav nalezen, je snahou k němu rozšířit druhý strom.



Obrázek 1.3: Dva pokoje oddělené chodbou, toto prostředí je vhodné pro plánování růstem dvou stromů

Detailnější znázornění této implementace je uvedeno v algoritmu 4. Nejdříve jsou stromy inicializovány tak, že ze stavů x_{init} a x_{goal} jsou vytvořeny kořeny těchto stromů. Poté, obdobně jako u základní techniky, následuje hlavní cyklus. Na začátku tohoto cyklu je vygenerován náhodný stav. K tomuto stavu je snahou rozšířit strom T_A. Podaří-li se nalézt rozšiřující stav x_{newA}, je po přidání do stromu T_A použit jako stav, ke kterému je snahou rozšířit druhý strom. Pokud je i v tomto kroku nalezen rozšiřující stav x_{newB}, je po přidání do stromu T_B změřena jejich vzájemná vzdálenost, a je-li menší než zvolené ε, je nalezena výsledná trajektorie. Je-li větší, pokračuje algoritmus vzájemným prohozením stromů a další iterací, ve které je postup opakován.

Hlavním problémem u této modifikace RRT techniky je propojení obou stromů. Jelikož bývá plánování prováděno pro neholonomní kinodynamický systém záleží na tom, jak budou stromy spojeny. To může být díky nevhodně zvolené metrice velice problematické. Přesto se často tento problém řeší tak, že se oba stromy spojí, aniž by byli dodrženy kinodynamické podmínky. Pokud jsou oba stromy propojené je vhodné na spoj aplikovat aproximační funkci, která zajistí, že kinodynamický robot

Algoritmus 4: RRTBidirect

Vstup: Počáteční stav x_{init}, cílový stav x_{goal}.
Výstup: Výsledná trajektorie pokud je nalezena.

```
1  T_A = treeInit(x_init)
2  T_B = treeInit(x_goal)
3  for i = 0 to max do
4      x_rand = getRandomState()
5      x_newA = growTree(T_A, x_rand)        // shodné s algoritmem 2
6      if x_newA then
7          x_newB = growTree(T_B, x_newA)
8          if x_newB ∧ δ(x_newA, x_newB) < ε then
9              return extractSolution(x_newA, x_newB)
10     T_A, T_B ← T_B, T_A
11 return failed
```

bude schopen projet tuto část trajektorie. Teoreticky by zde neměla být ani žádná tolerance ε, ale tím by bylo nalezení výsledné trajektorie časově velice náročné.

Dalším problém je, že tato technika nemůže být použita pro libovolný robot. Robot musí být ve směru jízdy osově symetrický nebo musí mít při jízdě dopředu i dozadu stejné kinodynamické vlastnosti, příkladem může být diferenciální robot. Algoritmus je už méně vhodný pro robot s kinodynamickými vlastnostmi jako automobil.

1.3 RRT-Connect

Dalším rozšířením základní RRT techniky je takzvané RRT-Connect. Slovo Connect (spojit) zde vyjadřuje to, že po vygenerování náhodného stavu x_{rand}, je snahou k tomuto stavu rozšiřovat strom dokud se k němu nepřiblíží nebo dokud není nalezeno výsledné řešení, a nebo pokud nedojde ke kolizi s překážkou. Strom se k náhodnému stavu x_{rand} rozšiřuje rychle, protože po prvním rozšíření tohoto stromu je znám nejbližší stav, ze kterého bude strom rozšiřován. Pokud je zvolena vhodná datová struktura stromu, může být vyhledání nejvhodnějšího souseda provedené také v konstantním čase [6]. Implementace této techniky je znázorněna v algoritmu 5.

Nejprve je strom inicializován přidáním počátečního stavu. Poté je vygenerován náhodný bod x_{rand} a k němu se snažíme rozrůstat strom dokud tento stav není dosažen nebo nedojde-li ke kolizi s překážkou, a nebo není nalezen cílový stav x_{goal}. Pokud je nalezen cílový stav ukončí se prohledávání rekonstrukcí trajektorie. V opačném případě pokračuje algoritmus generováním náhodného stavu a rozrůstáním stromu dokud nepřekročí limit maximálního počtu opakování těchto kroků. V tomto případě je hledání neúspěšné.

Dále lze toto rozšíření aplikovat i na RRT-Bidirect naznačené v algoritmu 4 a to jak na jeden strom tak i na oba. Díky této vlastnosti existuje několik variant této rozšiřující techniky. M. Kalisiak je označuje ve své disertační práci [4] jako RRTCon-

Algoritmus 5: RRT-Connect

Vstup: Počáteční stav x_{init}, cílový stav x_{goal}.
Výstup: Výsledná trajektorie pokud je nalezena.

```
1 T = treeInit(x_init)
2 for i = 0 to max do
3     x_rand = getRandomState()
4     while x_new do
5         x_new = growTree(T, x_rand)
6         if x_new ∧ δ(x_new, x_goal) < ε then
7             return extractSolution(x_new)
8 return failed
```

Con, RRTConExt, RRTExtCon a RRTExtExt, kde Con znamená spojit ve smyslu rozšiřovat strom k nějakému stavu a Ext znamená rozšířit ve smyslu jednoho rozšíření stromu směrem k nějakému stavu. V názvu varianty značí první indikátor způsob růstu prvního stromu, tedy stromu který se rozšiřuje směrem k náhodnému stavu a druhý indikátor značí způsob růstu druhého stromu, který se rozšiřuje směrem ke stavu, kterým byl rozšiřován první strom. Pro holonomní roboty je vhodná varianta RRTConCon, protože oba stromy se rychle rozrůstají a nezáleží jak budou oba stromy propojeny. Ve většině případů uvedených v [4] však bývá použita varianta RRTExtCon, ve které první strom prohledává prostor, tedy rozrůstá se směrem k náhodně vygenerovanému stavu a druhý strom roste ve směru k prvnímu dokud je není možné propojit.

Tato technika může dosahovat lepších výsledků v prostředích bez překážek nebo s malým počtem překážek, zejména je-li algoritmus vhodně informován o cílovém stavu.

1.4 RRT-Blossom

Další plánovací techniku založenou na principu RRT navrhl M. Kalisiak a nazval jí RRT-Blossom [4]. Hlavním rysem této techniky je to, že při rozšiřování stromu není hledán pouze jeden nejlepší stav, ale jsou zahrnuty i vzdálenější stavy ovšem bez těch, které by již zasahovali do prohledaného prostoru. Takto rozšířený stav připomíná kvítek a odtud pochází název této techniky. Výběr stavů, které budou přidány do stromu, je řízen podmínkou

$$\exists x \in T \;\mid\; \delta\left(x, x_{leaf}\right) < \delta\left(x_{parent}, x_{leaf}\right), \tag{1.3}$$

kde T je strom již navzorkovaných stavů a δ je vzdálenost mezi dvěma stavy. Zde je nutné zmínit, že tato podmínka může být nevýhodná pro neholonomní roboty. Pokud dynamická omezení povolují expandování stavu tak, že mezi dvěma nejvzdálenějšími expandovanými stavy se stejným rodičovským stavem, je vzdálenost menší než mezi expandovaným a rodičovským stavem, bude do stromu přidán vždy právě ten, který bude expandován jako první, a pokud bude expandování prováděno vždy ve stejném pořadí bude se robot pohybovat pořád stejným směrem.

Tato technika může být aplikována jak na prohledávání jedním stromem tak i na prohledávání dvěma stromy. Tedy podle typu prohledávání je funkce popsaná v algoritmu 6 použita v algoritmu 1 při prohledávání jedním stromem nebo v algoritmu 4 při růstu dvou dvou stromů.

Algoritmus 6 popisuje, jakým způsobem je u této techniky rozšiřován strom. Nejdříve je nalezen nejbližší stav k x_{rand} a z něho je snahou rozrůst strom.

Algoritmus 6: growTree

Vstup: Strom T, náhodný stav x_{rand}.
Výstup: Nejlepší expandovaný stav směrem k x_{rand}.

```
1 x_near = findNearestNeighbour(T, x_rand)
2 x_new = nodeBlossom(T, x_near, x_rand)
3 return x_new
```

V algoritmu 7 je popsán způsob jakým se rozrůstá stav x_{near}. Tento stav se rozrůstá přes všechny vstupy z množiny $\mathcal{U}$. Expandované stavy, které vedou na kolizi a ty stavy, které nesplňují podmínku (1.3) nejsou dále uvažovány. Z nově expandovaných stavů je navrácen ten, který je nejblíže k x_{rand}.

Algoritmus 7: nodeBlossom

Vstup: Strom T, nejbližší soused x_{near}, náhodný stav x_{rand}.
Výstup: Nově expandovaný stav nejblíže k x_{rand}.

```
1 foreach u ∈ U do
2     x = expand(x_near, u)
3     if isInCollision(x) then
4         continue
5     if regressionp(T, x_near, x) then
6         continue
7     T → addPoint(x)
8 return nový uzel nejblíže k x_rand
```

Algoritmus 8 reprezentuje rovnici (1.3) avšak místo procházení všech stavů ve stromu je zde vybrán nejbližší sousední stav x_{near} směrem k x_{new} a vzdálenost mezi těmito stavy je porovnána se vzdáleností mezi stavy x_{parent} a x_{new}. Výsledek tohoto

Algoritmus 8: regressionp

Vstup: Strom T, rodičovský stav x_{parent}, nový stav x_{new}.
Výstup: Informace o tom, zda byla podmínka splněna či nikoliv.

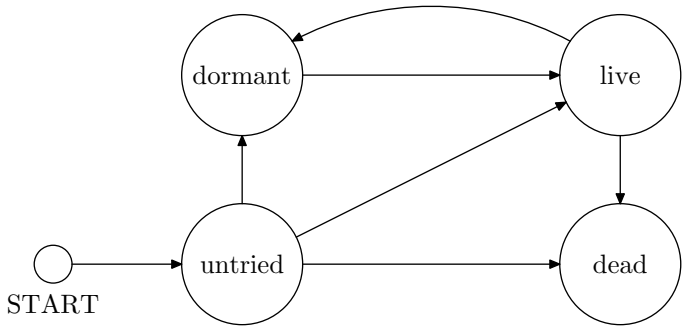
```
1 x_near = findNearestNeighbour(T, x_new)
2 if δ(x_near, x_new) < δ(x_parent, x_new) then
3     return true
4 return false
```

porovnání je návratovou hodnotou algoritmu.

Chování tohoto algoritmu sleduje koncept rychlého prohledávání prostoru především tím, že během jedné iterace se strom může rozrůst o více stavů. Algoritmus je avšak velice závislý na typu robotu, pro který je plánování prováděno. Dále může mít tato technika nepříjemné vlastnosti v prostředích s úzkými průjezdy. Pokud se strom přiblíží do místa s úzkým průjezdem, může být podmínkou (1.3) vyřazen i stav, který vede k vyřešení tohoto problému. Tyto nepříjemné vlastnosti se snaží odstranit nadstavbou této techniky nazvaná RRT-Viability.

1.5 RRT-Viability

Varianta techniky RRT nazvaná RRT-Viability je především rozšíření techniky RRT-Blossom a jejím autorem je M. Kalisiak [4]. V literatuře bývá tato modifikace označována také jako RRT-BlossomVF (RRT-Blossom with Viability Focus). Slovo viability znamená v překladu životaschopnost a důvodem tohoto označení je následující rozlišování uzlů ve stromu. Každý uzel je ohodnocen jednou ze čtyř hodnot z množiny {*untried, dormant, live, dead*}, jejíž prvky lze přeložit do češtiny jako {*nevyzkoušený, uspaný, živý, mrtvý*}. Na obrázku 1.4 jsou graficky znázorněny možné přechody mezi jednotlivými ohodnoceními.



Obrázek 1.4: Možné přechody mezi jednotlivými ohodnoceními jednoho uzlu ve stromu

Kromě tohoto ohodnocení uzlů je uzel rozšířen o dodatečné informace o tom, který stav je jeho předek, stavech následujících, ohodnocení hran propojující tento uzel s jeho následníky a seznam hran, které tento stav blokují. Hrany se ohodnocují ze stejné množiny hodnot jako ohodnocení uzlů. Například hrana vedoucí na kolizi je označena jako *mrtvá*. To má výhodu především z implementačního hlediska, jelikož nemusí být vytvářen objekt uzlu jehož stav je v kolizi. Navíc počet hran vedoucí z nějakého uzlu je předem znám, je stejný jako počet prvků množiny $\mathcal{U}$, tím je možné určit, zda byly vyzkoušeny veškeré expanze tohoto uzlu. Ohodnoceními je vylepšeno chování prohledávání stavového prostoru pokud stavy pro přidání do stromu nesplňují podmínku (1.3). Především tím, že stavy nesplňující podmínku nejsou úplně vyřazeny z uvažování, ale jsou ohodnoceny jako *uspané*. Pokud se totiž v průběhu expanze stromu ukáže, že předtím upřednostněný stav vede na kolizi, může být takto *uspaný* stav *oživen*. Tímto procesem tak nejsou vyloučeny z uvažování potenciální vhodné stavy k expanzi stromu.

Algoritmus 9: growTree

Vstup: Strom T, náhodný stav x_{rand}.
Výstup: Nejlepší expandovaný stav směrem k x_{rand}.

```
1 x_near = nearestNeighbour(T, x_rand)
2 x_new = nodeBlossom(T, x_near, x_rand)
3 return x_new
```

Postup růstu stromu je znázorněn v algoritmu 9 a skládá se ze dvou základních kroků podobně jako výchozí modifikace RRT-Blossom. Nejprve je vyhledán nejbližší stav x_{near} k náhodnému stavu x_{rand}, ze kterého se bude dále strom rozšiřovat. Tato technika může být použita při prohledávání jedním stromem i při prohledávání dvěma stromy, proto může být algoritmus 9 použit jako příslušná funkce v algoritmech 1 a 4.

Ohodnocení uzlů je zejména uvažováno při výběru nejbližšího souseda k novému náhodnému stavu x_{rand}. Schema výběru je znázorněno v algoritmu 10. Ze stromu je vybrán takový uzel stavu, který má nejkratší vzdálenost k x_{rand} a přitom není *mrtvý* ani *uspaný*. Pokud je strom v *deadlocku*, může být návratovou hodnotou algoritmu i *uspaný* uzel. V *deadlocku* se strom může nacházet ve dvou případech. První případ je takový, že všechny uzly stromu jsou mrtvé. Potom mezi startovním a cílovým bodem neexistuje trajektorie. Zde však budeme *deadlockem* nazývat případ, kdy jsou všechny nemrtvé uzly *uspané*.

Algoritmus 10: nearestNeighbour

Vstup: Strom T, náhodný stav x_{rand}.
Výstup: Nejbližší soused k x_{rand}.

```
1  d_min = ∞
2  foreach n ∈ T do
3      if n is dead then
4          continue
5      workable_states ← {untried, live}
6      if deadlock then
7          workable_states ← workable_states, dormant
8      if n.status ∩ workable_states = ∅ then
9          continue
10     d = δ(n, x_rand)
11     if d < d_min then
12         d_min = d
13         n_min = n
14 return n_min
```

Způsob jakým je strom rozšiřován z uzlu n se stavem x_{near} znázorňuje algoritmus 11. Z tohoto stavu je snahou expandovat nové stavy přes řídící vstupy z množiny $\mathcal{U}$. Ty nové stavy, které vedou na kolizi, nejsou dále uvažovány a hrany, které vedou

do těchto stavů jsou označeny jako *mrtvé*. Pokud není strom v *deadlocku* zjišťuje se, zda nově expandovaný stav není blokován stavem již dříve přidaným do stromu. Pokud je blokován, je hrana vedoucí do tohoto stavu označena jako *živá* a dále je přidaná jako blokovaná k uzlu, který jí blokuje. Není-li blokován, je uzel tohoto stavu přidán do stromu jako *živý*. Je-li strom v *deadlocku* bude do stromu přidán i uzel jehož stav je blokovaný jiným stavem. Poté co je uzel expandován přes všechny řídicí vstupy $u \in \mathcal{U}$ je výsledek tohoto expandování z uzlu n rozšíření do stromu a nakonec je navrácen nově expandovaný stav, který je nejblíže k x_{rand}.

Algoritmus 11: nodeBlossom

Vstup: Strom T, uzel n nejbližšího sousedního stavu x_{near}, náhodný stav x_{rand}.

Výstup: Nově expandovaný stav nejblíže k x_{rand}.

```
1  x ← n.x
2  foreach u ∈ U do
3      x_new ← sim (x, u)
4      if isInCollision(x_new) then
5          n.edge_status[u] ← dead
6          continue
7      if !deadlock then
8          n_blk ← regressionp(T, x, x_new)
9          if n_blk then
10             n.edge_status[u] ← live
11             n_blk.block_edges ← n_blk.block_edges + (n, u)
12             continue
13     T ← addPoint(x_new)
14     n.edge_status[u] ← live
15 propagate_status(n)
16 return nově expandovaný stav nejblíže k x_rand
```

Algoritmus 12 popisuje způsob jakým je reprezentována podmínka (1.3). Existuje-li nemrtvý uzel splňující tuto podmínku, je algoritmem navrácen. V opačném případě je navrácena 0.

V algoritmu 13 je naznačen způsob jakým je ve stromu rozšiřováno ohodnocení nově přidaných stavů. V podstatě se zde provádí to, že pokud jsou všichni potomci ohodnoceny jako *uspané* bude uspán i rodič. Pokud je alespoň jeden potomek *živý* bude jako *živý* označen i rodič. Jsou-li všichni potomci *mrtví* je takto ohodnocen i rodič. Navíc, je-li umrtven stav, který blokoval nějaké jiné stavy, jsou tyto stavy označeny jako nevyzkoušené a ohodnocení těchto odblokovaných stavů je rozšířeno do stromu. Tato ohodnocení se rekurzivně šíří až ke kořeni stromu. Toto rekurzivní šíření je zastaveno pokud má rodičovský stav již takové ohodnocení, jaké by mu bylo přiřazeno podle následníků.

Tímto šířením ohodnocení stavů stromem, lze zjistit jestli je stromu v *deadlocku* tím, že je uspaný jeho kořen. V případě, že je kořen stromu *mrtvý*, neexistuje trajek-

Algoritmus 12: regressionp

Vstup: Strom T, rodičovský stav x_{parent}, nový stav x_{new}.
Výstup: Uzel stavu splňující podmínku (1.3) pokud existuje a není *mrtvý*.

```
for n ∈ T do
    if n.status = dead then
        continue
    if δ(n, x_new) < δ(x_parent, x_new) then
        return n
return 0
```

Algoritmus 13: propagate_status

Vstup: Uzel n.

```
while n do
    s_n = dead
    for u ∈ U do
        s_e ← n.edge_status[u]
        if s_e = live then
            s_e ← n.children[u].status
        if s_e ∈ {untried, live} then
            s_n = live
        if s_e = dormant then
            s_n = dormant
    if n.status = s_n then
        return
    if s_n = dead then
        for (n_b, u) ∈ n.block_edges do
            if n_b.edge_status[u] = dormant then
                n_b.edge_status[u] ← untried
                propagate_status(n_b)
    n ← n.parent
```

torie mezi startovním a cílovým stavem, jelikož je *mrtvý* i zbytek stromu. Vlastnost rozhodnutí o neexistenci řešení má ze zde uvedených modifikací plánovací techniky RRT pouze tato technika. Její hlavní výhodou oproti ostatní zde uvedeným modifikacím je lepší chování při plánování trajektorií úzkými průjezdy, což je dále ukázáno v kapitole 4 věnované experimentům.

Kapitola 2

KPIECE

V této kapitole je stručně popsaná plánovací technika KPIECE, jejíž použití pro plánování pohybu robotu PR2 (Personal Robot 2) bylo prezentováno v článku [11]. Název KPIECE vychází z anglického (Kinodynamic Motion Planning by Interior-Exterior Cell Exploration) a do češtiny by se dal přeložit jako kinodynamické plánování pohybu explorací vnitřních a vnějších buněk [13]. Jako vnitřní je buňka označena v případě, že má 2n sousedních buněk, ale přitom se nezapočítávají sousedé na diagonále. Vnější a vnitřní buňky jsou pro tento algoritmus charakteristické. Tyto buňky jsou používány pro efektivní vyhledávání stavu, který bude expandován a je snahou expandovat především vnější buňky. Technika KPIECE vzorkuje stavový prostor podobně jako RRT, avšak při vzorkování je kladen větší důraz na výběr nové konfigurace. V případě RRT je nová konfigurace, ke které je strom expandován volena zcela náhodně. V KPIECE technice je výběr buňky deterministický a stav pro další růst stromu je následně volen náhodně z takto vybrané buňky.

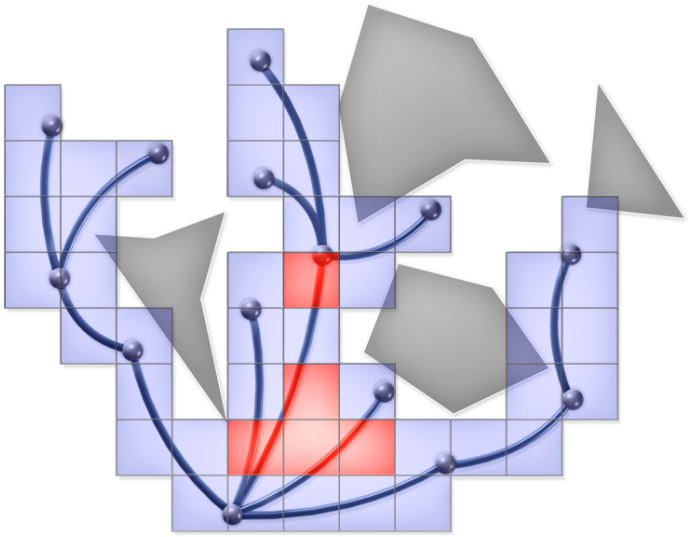
V následující části je popsán základní princip techniky KPIECE. Část 2.2 je věnována diskusi projekci stavového prostoru do planárního prostoru, ve kterém jsou organizovány buňky, a jež je nezbytnou součásti algoritmu KPIECE. V části 2.3 je pak uveden vlastní algoritmus KPIECE tak jak byl prezentován v [13].

2.1 Základní princip techniky KPIECE

Základní myšlenkou techniky KPIECE je sledování pokrytí prohledávaného prostoru náhodnými vzorky tak, aby bylo pokud možno vzorkováno v místech, které jsou potenciálně vhodná k nalezení řešení. Sledování pokrytí prostoru může být časové náročné, proto autoři tohoto algoritmu využívají rozdělení prostoru na buňky. Efektivity této struktury je dosaženo především tím, že se jedná o 2D mřížku (grid), případně 3D mřížku. Příklad mřížky a stromu je zobrazen na obrázku 2.1, vnitřní buňky jsou zobrazeny červeně a vnější buňky modře. Jelikož však plánování typicky probíhá v prostor dimenze mnohem vyšší je použita projekce konfiguračního prostoru vyšší dimenze do roviny.

Je-li k dispozici vhodná projekční funkce $\mathcal{E}(X)$, kde X je stavový prostor, je možné růstu stromu techniky KPIECE popsat následovně.

1. Výběr buňky c, s preferencí vnějších buněk (např. bias v rozmezí 70 % až 90 %).



Obrázek 2.1: Strom a buňky v technice KPIECE, *obrázek převzat z [11]*

2. Výběr stavu s z c podle poloviny-normální distribuce.
3. Výběr náhodné vzorku x ze stavového prostoru X.
4. Rozšíření stromu z s k x až k nějakému stavu x'.
5. Je-li pohyb z s do x' realizovatelný
   - přidání stavu x' do stromu a
   - aktualizace penalizace buňky c ($c.score = c.score \cdot \alpha$), kde $0 < \alpha < 1$,
6. jinak
   - aktualizace penalizace buňky c ($c.score = c.score \cdot \beta$), kde $0 < \beta < \alpha$.

Uvedený postup je opakován dokud není dosaženo cíle, nebo definovaného časového intervalu pro růst stromu.

2.2 Projekce stavového prostoru do roviny

Výběrem vhodné projekční funkce se autoři zabývají v článku [1], ve kterém jsou studovány různé náhodně zvolené funkce spolu s manuálně nalezenými funkcemi, pokud možno co nejlépe odpovídající konkrétnímu problému. V článku jsou uvažovány náhodné lineární projekce

$$\boldsymbol{V} = (\boldsymbol{v}_1, \ldots \boldsymbol{v}_k), \boldsymbol{v}_i \in \mathbf{R}^n. \tag{2.1}$$

Stav $x \in X$ z prostoru dimenze n je transformován na stav p z prostoru dimenze k, $p \in \mathbf{R}^k$. Tedy

$$p = \boldsymbol{V}^T x, \tag{2.2}$$

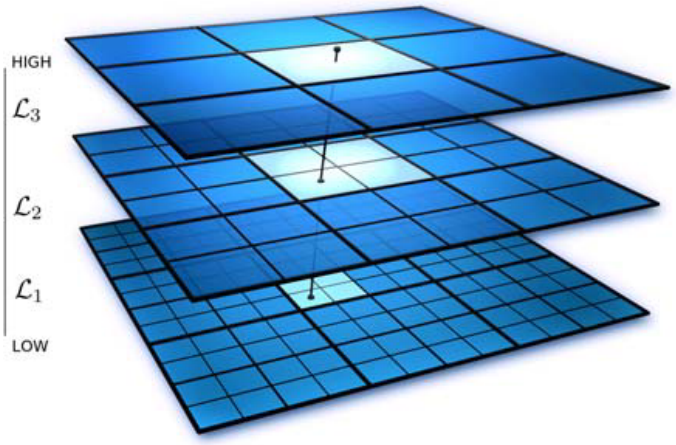
pro případ sloupcových vektorů.

Autoři použili následující způsob výběru náhodné transformace pro $k = 2$ a $k = 3$. Nejdříve generují N náhodných projekcí, která následně ohodnotí a setřídí vzestupně podle ohodnocení. Ohodnocení je založeno na opakovaném volání algoritmu KPIECE, který využívá příslušnou projekci, pro zvolené testovací úlohy. Pokud je nalezeno algoritmem KPIECE řešení dvakrát, je projekce označena jako validní a průměrná doba řešení je použita jako ohodnocení projekce v tomto konkrétní běhu. Celkové ohodnocení příslušné projekce je pak počítáno z několika běhů. Tímto způsobem byly vybrány tři projekce: s nejvyšším, nejnižším hodnocením a medián. Tyto projekce pak byly následně použity v porovnání s uživatelsky definovanou projekcí, speciálně navrženou pro konkrétní úlohu, v řadě dalších úloh s třemi různými modely robotů.

V závěru článku autoři hodnotí možnost automatického nalezení projekční funkce a shrnují dosažené výsledky tím, že je možné použít náhodnou lineární projekci, přestože netvrdí, že je lineární projekce dostačující. Uvedený závěr je obzvlášť důležitý z pohledu univerzálního použití algoritmu KPIECE, kde je obzvlášť výhodné, aby uživatel nebyl nucen nastavovat příslušnou projekci manuálně. Automatickou volbou projekce je algoritmus KPIECE více použitelným, neboť efektivního rozdělení na buňky lze dosáhnout ve 2D nebo 3D prostoru.

2.3 Algoritmus KPIECE

Algoritmus KPIECE byl přestaven v [13]. Algoritmus sleduje základní myšlenku uvedenou v části 2.1, avšak pro zvýšení rychlosti výběru vhodné buňky je použito hierarchické struktury, viz obrázek 2.2. Kromě této struktury autoři také v článku zmiňují, že jednotlivé buňky na příslušných úrovní vytvářejí až tehdy, jsou-li potřeba, čímž dosahují snížení paměťových nároků.



Obrázek 2.2: Vrstvy hierarchického rozdělení na buňky v algoritmu KPIECE, *obrázek převzat z [13]*

Hlavním cílem použití buněk je odhad pokrytí stavového prostoru stromem. Autoři proto definují míru pokrytí buňky p z nejnižší vrstvy $\mathcal{L}_1$ jako součet dob jed-

notlivých plánovaných pohybů v buňce p. Pro vyšší vrstvy pak jako počet vytvořených buněk na nižší úrovni.

Při vlastním rozšiřování stromu jsou vzorkovány jednotlivé vrstvy a lze tak hovořit o řetízku buněk (*cell chain*). Vnější buňky jsou preferovány a výběr je založen na takzvané důležitosti buňky (*importance*), která je počítána jako:

$$importance(p) = \frac{\log(i) \cdot score}{\mathcal{S} \cdot \mathcal{N} \cdot \mathcal{C}},$$

kde i je pořadové číslo iterace, ve které byla buňka vytvořena, hodnota *score* je inicializována na 1 a v průběhu růstu stromu je aktualizována, $\mathcal{S}$ udává kolikrát byla buňka p vybrána pro expanzi, $\mathcal{N}$ je počet vytvořených sousedních buněk (ve stejné úrovni) a $\mathcal{C}$ míra pokrytí buňky. Další část expanze stromu se skládá z výběru stavu, náhodné vzorku stavového prostoru, růstu k takovému náhodnému vzorku a sleduje tak základní princip. Více úrovní mřížky však vyžaduje aktualizaci hodnoty *score* na všech úrovní. Pro úplnost je algoritmus KPIECE uvedený v [13] přetisknut v obrázku 2.3.

Algorithm 1. KPIECE($q_{start}, N_{iterations}$)

1: Let μ_0 be the motion of duration 0 containing solely q_{start}
2: Create an empty `Grid` data-structure G
3: G.ADDMOTION(μ_0)
4: **for** $i \leftarrow 1 ... N_{iterations}$ **do**
5: Select a cell chain **c** from G, with a bias on exterior cells (70% - 80%)
6: Select μ from **c** according to a half normal distribution
7: Select s along μ
8: Sample random control $u \in U$ and simulation time $t \in \mathbb{R}^+$
9: Check if any motion $(s,u,t_\circ)$, $t_\circ \in (0,t]$ is valid (forward propagation)
10: **if** a motion is found **then**
11: Construct the valid motion $\mu_\circ = (s,u,t_\circ)$ with $t_\circ$ maximal
12: If $\mu_\circ$ reaches the goal region, **return** path to $\mu_\circ$
13: G.ADDMOTION($\mu_\circ$)
14: **end if**
15: **for** every level $\mathscr{L}_j$ **do**
16: $P_j = \alpha + \beta \cdot$ (ratio of increase in coverage of $\mathscr{L}_j$ to simulated time)
17: Multiply the `score` of cell p_j in **c** by P_j if and only if $P_j < 1$
18: **end for**
19: **end for**

Obrázek 2.3: Schema algoritmu KPIECE, *převzato z [13]*

Implementace plánovací techniky KPIECE je součástí volně dostupné knihovny `OMPL` (Open Motion Planning Library) [12]. Algoritmus je v této knihovně implementován s podporou více vláken a využívá tak možností moderních procesorů, které obsahují více výpočetních jader a program tak může běžet skutečně paralelně. I přesto, že je implementace volně dostupná, není její použití přímočaré a jednoduché, zejména z důvodu závislosti na použitých strukturách, které jsou součástí softwarového balíku *ROS*. Při praktickém použití to znamená, že je nutné se seznámit se základními

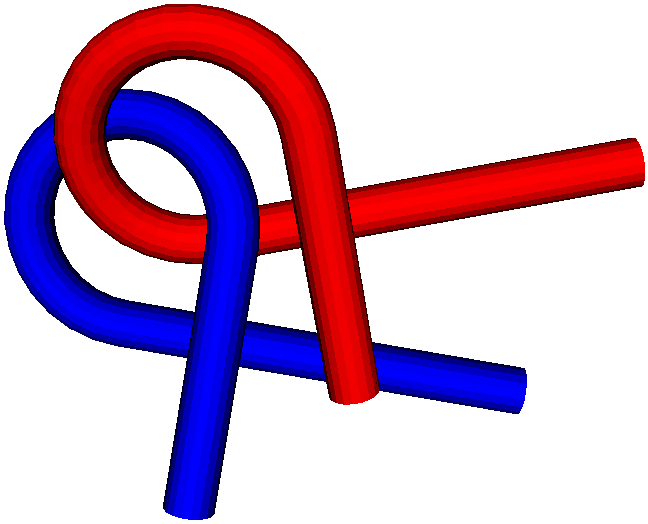
použitými strukturami a přizpůsobit jim vlastní program. Jelikož je ROS relativně komplexní softwarový produkt zaměřený na aplikaci řízení robotu, který poskytuje hardwarovou abstrakci, nízkoúrovňové řízení, základní funkce a podporu komunikace mezi jednotlivými moduly, je dostupná dokumentace relativně rozsáhlá a její nastudování může trvat nezanedbatelnou dobu.

Kapitola 3

Alpha Puzzle

Dobrým způsobem jak porovnávat a testovat plánovací techniky, je spouštět je na známých a složitých úlohách, neboť tak lze relativně jednoznačným způsobem zaručit stejné počáteční podmínky pro různé algoritmy. Jednou z takových úloh hlavolamu známý pod názvem *Alpha Puzzle*, který je součástí souboru dostupných standardní problémů [`http://parasol.tamu.edu/dsmft/benchmarks/mp`].

Hlavolam*Alpha Puzzle* se skládá ze dvou trubiček ohnutých do tvaru řeckého písmene α, které jsou vzájemně propleteny do sebe. Cílem úlohy je nalézt trajektorii pro jednu z trubiček tak, aby došlo k oddělení propletených trubiček, viz obrázek 3.1. Tedy problém je možné si představit tak, že jedna trubička představuje překážku a



Obrázek 3.1: Vizualizace hlavolamu*Alpha Puzzle* ve VTK, modrou barvou je znázorněna část hlavolamu představující překážku, červenou barvou je označen robot, každá trubička je složena z 1008 trojúhelníků

druhá reprezentuje robot. K vyřešení hlavolamu je potřeba obě části od sebe oddělit, což v tomto konkrétní případě znamená, že jsou obě části velmi blízko sebe. Tato úloha je proto náročná, zejména s ohledem na řešení problému *narrow passage*.

3.1 Přístup k řešení

Úloha*Alpha Puzzle* představuje problém ve 3D a ve své základní variantě jde pouze o nalezení nějaké realizovatelné trajektorie. Proto postačuje uvažovat pouze šesti dimenzionální konfigurační prostor, tři parametry (x, y, z) popisující pozici a tři parametry (φ, ψ, θ) popisující natočení. Algoritmy popsané v kapitole 1, ale i algoritmus KPIECE popsaný v kapitole 2, představují univerzální plánovací procedury, proto může být de-facto použit kterýkoliv z nich. S ohledem na *narrow passage* problém u úlohy*Alpha Puzzle* byla pro demonstrační účely RRT techniky zvolena modifikace RRT-Connect, která se pro tento typ úloh hodí.

3D charakter scény vyžaduje při ladění algoritmu a také 3D vizualizaci scény. Kromě vizualizace, je také nutné zvolit vhodný souřadný systém a zjistit správné provedení příslušných transformací, které reprezentují pohybu objektu v prostoru a jeho natočení je reprezentováno maticí rotace. Použitý způsob vizualizace je popsán v následující části a popis nalezeného řešení je uveden v části 3.3.

3.2 Vizualizace

Možností jak vizualizovat třírozměrnou scénu je několik, od přímé použití některé základní knihovny, například OpenGL, až po sofistikované nástroje poskytující širokou paletu možností konfigurace vizualizované scény. Výhodou vyšších nástrojů je odstínění uživatele od základních zobrazovacích primitiv, který se tak může plně soustředit na řešení vlastního problému a přenechat vizualizaci specializovaným procedurám poskytující vyšší uživatelský komfort. Jednou z takových možných knihoven je i VTK (Visualization Tool Kit) [5]. Knihovna VTK je volně dostupná a je nabízena jako takzvaný kitware, tedy jako jedna ze sady vizualizačních nástrojů vyvíjených firmou Kitware, Inc. Hlavní výhodou kitware produktů je, že jsou poskytovány jako open source, z čehož plyne výhoda pořizovací ceny. Neméně důležitým aspektem open source je snadná modifikace, proto lze nalézt řadu uživatelských rozšíření, které se v některých případech stávají součástí nových verzí knihovny VTK. VTK je dostupné pro více platforem a jeho rozhraní je vytvořeno pro jazyky C++, Tcl, Perl, Python a Java.

Základní koncept použití VTK je založen na vizualizaci datových struktur, na které je možné aplikovat různé procedury a tím dosáhnout požadovaného způsobu zobrazení atributů uložených v příslušných datových strukturách. V případě vizualizace scény při plánování nejsou vyžadovány specifické vizualizační vlastnosti a je tak postačující co možná nejjednodušší způsob vizualizace příslušného robotu a prostředí.

V knihovně VTK lze takového přístupu vizualizace docílit prostřednictvím instance třídy `vtkActor`, která reprezentuje příslušný 3D objekt, nad kterým je možné provádět operace rotace a translace. Výhodou tohoto objektu je, že tyto operace jsou realizovány prostřednictvím OpenGL přímo na grafické kartě, proto mohou být velmi efektivní a nezatěžovat hlavní CPU. Tedy pro nějaký model objektu použitý v plánovací části algoritmu postačuje vytvoření příslušného VTK objektu a pohyb tohoto objektu je pak možné realizovat voláním metod `SetPosition(x, y, z)`, `RotateZ(angle)`, `RotateY(angle)`, `RotateX(angle)` nebo `SetOrientation(phi,`

psi, theta). Promítnutí požadovaných změn na pozici a natočení objektu je pak provedeno voláním metody Render().

```
vtkActor *createActor(const collision::Model& model) {
   vtkCellArray      *tMash            = vtkCellArray::New();
   vtkPoints         *points           = vtkPoints::New();
   vtkPolyData       *trianglePolyData = vtkPolyData::New();
   vtkPolyDataMapper *mapper           = vtkPolyDataMapper::
      New();
   vtkTriangle       *triangle;
   for (unsigned int i = 0; i < model.size(); i++) {
      points->InsertNextPoint(model[i].v1[0], model[i].v1[1],
         model[i].v1[2]);
      points->InsertNextPoint(model[i].v2[0], model[i].v2[1],
         model[i].v2[2]);
      points->InsertNextPoint(model[i].v3[0], model[i].v3[1],
         model[i].v3[2]);
      triangle = vtkTriangle::New();
      triangle->GetPointIds()->SetId(0, i * 3);
      triangle->GetPointIds()->SetId(1, i * 3 + 1);
      triangle->GetPointIds()->SetId(2, i * 3 + 2);
      tMash->InsertNextCell(triangle);
   }
   trianglePolyData->SetPoints(points);
   trianglePolyData->SetPolys(tMash);
   mapper->SetInput(trianglePolyData);
   vtkActor *actor = vtkActor::New();
   actor->SetMapper(mapper);
   return actor;
}
```

Listing 3.1: Postup vytvoření vizualizovaného objektu

Příklad vytvoření objektu vtkActor je zobrazen ve výpise 3.1. V úloze*Alpha Puzzle* jsou jednotlivé objekty reprezentovány jako množiny trojúhelníku. Ty jsou ve VTK reprezentovány jako množina bodů (vtkPoints) resp. množina jednotlivých relací mezi body tvořící trojúhelník (vtkTriangle), které jsou uloženy v obecném seznamu buněk (vtkCellArray). Před vytvořením objektu vtkActor jsou body a relace uloženy ve společné struktuře vtkPolyData, pro kterou je vytvořen takzvaný *mapper*. Ten je následně použit při vlastním vytvoření objektu vtkActor.

Neméně důležitou součástí vizualizace je vytvoření vlastního vizualizačního okna a vizualizačního kontextu. Ve VTK je tato část velmi snadná a je k dispozici několik možností jak scénu inicializovat. Především záleží, zda-li se jedná o interaktivní scénu, ve které může uživatel s prvky manipulovat, nebo o scénu, která je plně ovládána programově. Pro základní vizualizaci scény z pozice objektu při plánování je postačující druhá možnost, která je zobrazena ve výpise 3.2

```
vtkRenderer *renderer = vtkRenderer::New();
vtkRenderWindow* renderWindow = vtkRenderWindow::New();
vtkRenderWindowInteractor* renderWindowInteractor =
```

```
    vtkRenderWindowInteractor::New();

renderer->SetBackground(0.2, 0.2, 0.2);
renderWindow->SetSize(640, 480);

renderWindow->AddRenderer(renderer);
renderWindowInteractor->SetRenderWindow(renderWindow);

renderWindow->Render();
renderWindowInteractor->Render();
```

Listing 3.2: Postup vytvoření vizualizačního kontextu

Příklad přidání objektu do scény a vizualizace příslušné změny objektu je zobrazena ve výpise 3.3.

```
renderer->AddActor(actorRobot); //add robot to the scene
renderWindowInteractor->Render();

actorRobot->RotateZ(1.0); //change robot yaw
renderWindowInteractor->Render();
```

Listing 3.3: Přidání objektu do scény a promítnutí změn v poloze

Další užitečnou funkcionalitu VTK je ukládání obrázků vizualizované scény a to jak do vektorového tak rastrového formátu. K jejich vytvoření se používají třídy `vtkGL2PSExporter` pro vektorové obrázky, `vtkWindowsToImageWriter` a `vtkImageWriter` pro rastrové obrázky. Jejich použití je přímočaré. Do instance jejich objektů se vloží objekt okna, které má být uloženo a nastaví se název a typ souboru. Obrázek se poté uloží voláním metody `Write()`.

Knihovna VTK nabízí širokou škálu vizualizačních nástrojů, které se dají relativně snadno použít. Výrobce poskytuje na internetu rozsáhlou dokumentaci a množství příkladů použití tohoto nástroje. Pokud si uživatel zvykne na práci s touto dokumentací stane se použití vizualizačního nástroje VTK snadné.

3.3 Příklad řešení

Množina řídicích vstupů $\mathcal{U}$ byla vytvořena ze všech možných kombinací rotací a posunutí. Posunutí bylo vzorkováno s krokem 0,2 cm a úhel pootočení 0,05 rad. Pokud by bylo zvoleno větší pootočení nebo posunutí mohlo by se v určitých případech stávat, že by překážka byla přeskočena. Takto byla získána množina 728 různých řídicích vstupů jako kombinace všech pootočení a posunutí t.j. $3^6 - 1$, protože kombinace žádného pootočení ani posunutí není třeba. Řešení bylo nalezeno algoritmem RRT-Connect spuštěném v operačním systému Gentoo s jádrem Linux Vanilla verze 2.6.34-rc1 běžící na stroji s procesorem Intel Core Duo T2050 @ 1,6 GHz s 2GB RAM. Pro výpočet bylo použito pouze jedno jádro použitého procesoru a řešení bylo nalezeno přibližně za 5,5 hodiny a bylo při tom využito až 1,5 GB paměti RAM. Algoritmus byl informován o cílové poloze v každé 500té iteraci.

Pro nalezení výsledného řešení nebylo nutné modifikovat plánovací algoritmus, jelikož není závislý na velikosti dimenze prostoru, ve kterém je plánování prováděno. Pouze byl vytvořen nový robot a nový stavový vektor, které určují stavový prostor robotu a způsob, kterým se v něm bude pohybovat. Tedy modifikace pro plánování trajektorií planárních robotů pro roboty prostorové je přímočaré a může být pro obě skupiny použita identická implementace plánovací techniky.

Kapitola 4

Experimenty

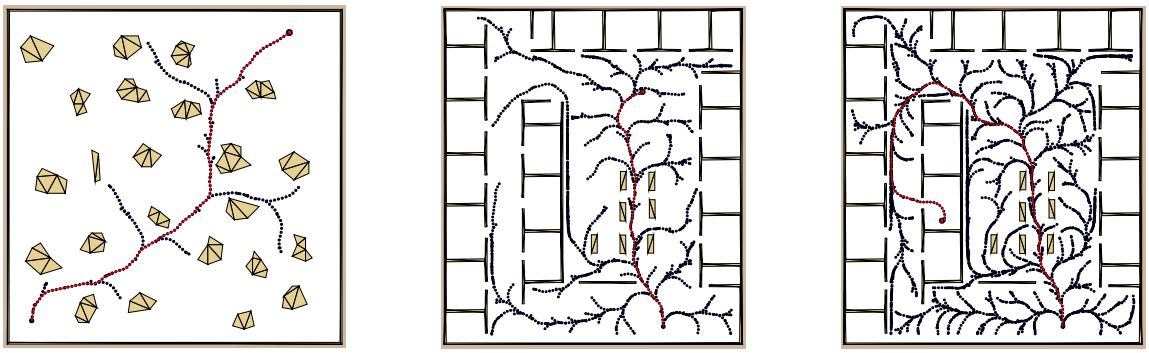
Tato kapitola je věnována prezentaci provedených experimentů, ve kterých jsou vzájemně porovnány jednotlivé modifikace techniky RRT uvedené v kapitole 1. Jednotlivé algoritmy byly opakovaně spuštěny pro různé úlohy, které se liší jak v rozmístění a charakteru překážek, tak také v použitém modelu robotu. Všechny testované algoritmy byly implementovány v C++ a přeloženy překladačem G++ verze 4.3.4. Všechny experimenty byly provedeny v identickém výpočetním prostředí a tedy uvedené požadované časové nároky na řešení jsou přímo porovnatelné. Algoritmy byly spuštěny v operačním systému Gentoo s jádrem Linux Vanilla verze 2.6.34-rc1 běžící na stroji s procesorem Intel Core Duo T2050 @ 1,6 GHz s 2GB RAM. Implementované algoritmy jsou jednovláknové a využívají tak pouze jediného výpočetního jádra použitého procesoru.

Tato kapitola je organizována následujícím způsobem. V části 4.1 jsou popsány jednotlivé testovací úlohy z pohledu modelu prostředí a robotů. Jelikož je rychlost nalezení řešení technikou RRT do značné míry ovlivněna algoritmy detekce kolizí a hledání nejbližšího souseda jsou v části 4.2 stručné popsány použití podpůrné algoritmy, resp. knihovny. Vlastnímu porovnání původní RRT techniky a několika modifikací uvedených v kapitole 1 je věnována část 4.3. V části 4.4 jsou pak uvedeny experimentální výsledky vlivu informovanosti RRT algoritmu (*goal bias*) na rychlost nalezení řešení zvolené úlohy.

4.1 Popis testovacích úloh

4.1.1 Testovací prostředí

Jako testovací prostředí byly zvoleny planární prostředí *jh* a *potholes*. V prostředí potholes byla zvolena startovní konfigurace v levém dolním rohu a cílová v pravém horním, viz obrázek 4.1a. V prostředí jh byly zvoleny dvě konfigurace: jednodušší (*jh*-I) je zobrazena na obrázku 4.1b a složitější (*jh*-II) na obrázku 4.1c.



(a) potholes (b) jh (c) jh

Obrázek 4.1: Zvolená prostředí ve kterých bylo provedeno testování

4.1.2 Modely robotů

Pro testování byly zvoleny dva typy robotů. Diferenciální rovnice popisující jejich pohyb jsou převzaty z [14]. Pro měření vzdálenosti mezi dvěma konfiguracemi robotu byla v obou případech zvolena eukleidovská metrika

$$\delta\left(q_1, q_2\right) = \sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2}. \tag{4.1}$$

Diferenciální robot

Tento robot je zkonstruován tak, že má po stranách dvě kolečka o poloměru r. Kolečka jsou od sebe vzdáleny L a robot je řízen změnou jejich rychlostí, t.j. rychlost pravého kolečka v_r a levého v_l. Točí-li se obě kolečka stejnou rychlostí pohybuje se robot po přímce a naopak, jsou-li tyto rychlosti rozdílné tak robot zatáčí. Konfigurace robotu v prostoru je popsána vektorem $q = (x, y, \varphi)$ a jeho pohyb popisují rovnice

$$\begin{aligned} \dot{x} &= \frac{r}{2}\left(v_r + v_l\right) cos\varphi, \\ \dot{y} &= \frac{r}{2}\left(v_r + v_l\right) sin\varphi, \\ \dot{\varphi} &= \frac{r}{L}\left(v_r - v_l\right). \end{aligned} \tag{4.2}$$

Robot se bude pohybovat po nějaké kružnici, pokud budou rychlosti obou koleček konstantní, ale navzájem různé $v_r \neq v_l$. Tento model robotu je v experimentech značen jako *diff*.

Car-like robot

Car-like robot připomíná vzhledem i chováním automobil. Jako u automobilu jsou vpředu dvě kolečka, kterými lze měnit směr jízdy robotu a vzadu ve vzdálenosti L od předních koleček jsou zadní dvě kolečka. Robot je řízen dopřednou rychlostí u_s a natočením předních koleček u_ϕ. V prostoru popisuje konfiguraci robotu vektor $q = (x, y, \varphi)$. Jeho pohyb popisují rovnice

$$\begin{aligned} \dot{x} &= u_s cos\varphi, \\ \dot{y} &= u_s sin\varphi, \\ \dot{\varphi} &= \frac{u_s}{L} \tan u_\phi. \end{aligned} \tag{4.3}$$

Tento model je v experimentech značen jako *car-like*, resp *car*.

4.2 Podpůrné struktury

4.2.1 Detekce kolizí - RAPID

Pro detekování kolizí byla použita knihovna RAPID [3]. Ta umožňuje detekování kolizí dvou pevných nedeformujících se modelů, které jsou do systému nahrány jako trojúhelníky složené z jednotlivých bodů reprezentující model robotu nebo prostředí. S modelem robotu se v systému pohybuje ortogonální rotační maticí a vektorem posunutí. Systém je navržen tak, že může být zjištěno, které trojúhelníky jsou v kolizi. Pro účely experimentů je postačující detekovat pouze jednu kolizi. V tabulce 4.1 je počet trojúhelníků které v reprezentují robot a počet trojúhelníků, ze kterých je vytvořen model prostředí.

Tabulka 4.1: Počet trojúhelníků reprezentující model robotu a mapy

(a) modely robotů

Robot	**Počet trojúhelníků**
car	2
diff	2

(b) modely prostředí

Mapa	**Počet trojúhelníků**
jh	182
potholes	111

4.2.2 Datová struktura stromu - MPNN

Pro hledání nejbližšího souseda je využita datová struktura knihovny MPNN, kterou navrhla A. Yershova a její popis je uveden v článku [16]. Jednotlivé stavy jsou v této struktuře uloženy v KD-stromu a vzdálenost mezi jednotlivými stavy je určena uživatelsky definovanou metrikou. Knihovna umožňuje zvolit topologii systému, což může být výhodné pokud vektor stavu obsahuje nějaké informace v úhlech. Hlavní výhodou této knihovny je, že dovede rychle a efektivně nalézt jednoho nejbližšího souseda nebo k nejbližších sousedů k nějakému stavu. Počet sousedních stavů, které

budou navráceny, je nutné definovat při inicializaci této knihovny. V provedených experimentech byla použita eukleidovská vzdálenost a jediný nejbližší soused.

4.3 Porovnání algoritmů

Jednotlivé algoritmy jsou porovnány na základě několika sledovaných parametrů. Zejména se jedná o počet iterací potřebných k nalezení řešení. Dále je to počet kolizí, který indikuje jak moc vhodně je strom expandován, a počet expanzí, který indikuje velikost stromu, ve kterém je nalezeno výsledné řešení. V případě varianty RRT-Viability, je však celkový počet uzlů vyšší, neboť při expanzi dochází k přidání více uzlů do stromu, které jsou následně ohodnoceny, např. jsou uspány. RRT technika představuje randomizovaný algoritmus, proto je každý algoritmus spuštěn 100 krát a příslušné ukazatele jsou počítány jako průměrné hodnoty z těchto běhů[1]. Navíc testované algoritmy nemusí vždy nalézt řešení v dedikovaném časovém intervalu, proto je součástí prezentovaných výsledků také počet úspěšných řešení, ve kterých bylo nalezeno řešení. Kromě zmíněných ukazatelů jsou z úspěšných řešení počítány dva kvalitativní ukazatele nalezených řešení: průměrná doba nalezení řešení (značena T) a průměrná délka nalezené cesty, která je značena L.

Jednotlivé testované algoritmy jsou: `rrt` (viz část 1.1), `rrt-bidirect` (část 1.2), `rrt-connect` (část 1.3) a `rrt-viability` (část 1.5). Hodnota max, tedy počet iterací, byla v algoritmech nastavena na 100.000. Pokud do této doby nebylo nalezeno řešení, bylo hledání ukončeno a označeno za neúspěšné. Kromě RRT-Bidirect byla ve všech algoritmech schodně nastaven hodnota informovanosti (goal-bias) na 20. Tedy v každé 20té iteraci byl algoritmus informován o cílové konfiguraci. Na každou hranu bylo použito 6 integračních vzorků, integrovaných Eulerovo metodou. Doba potřebná na ujetí jednoho vzorku byla stanovena na 1, čímž byla usnadněna implementace. Pro oba roboty je uvažována proměnná rychlost, tedy hrana mezi jednotlivými vzorky má délku 1-5 cm. Hledání trajektorie bylo označeno jako úspěšné, pokud se robot přiblížil k cílovému stavu na méně než 14 cm. Výsledky jsou pro jednotlivé prostředí a roboty jsou uvedeny v tabulkách 4.2-4.7 Příklady nalezených řešení jsou zobrazeny na obrázcích 4.2 a 4.3.

Tabulka 4.2: Robot *diff*, prostředí *potholes*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	T [s]	L [m]
`rrt`	994	703	1.697	100	0,42	29,89
`rrt-bidirect`	81	97	322	100	0,12	28,23
`rrt-connect`	993	992	993	100	0,47	35,80
`rrt-viability`	777	418	1.195	100	0,50	29,06

[1]Příslušné směrodatné odchylky jsou uvedeny v příloze B.

Tabulka 4.3: Robot *car*, prostředí *potholes*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	T [s]	L [m]
`rrt`	2.359	6.169	8.528	97	1,27	28,34
`rrt-bidirect`	72	157	563	100	0,17	27,12
`rrt-connect`	3.413	12.836	12.837	94	2,20	32,92
`rrt-viability`	2.827	1.607	4.434	100	2,31	27,00

Tabulka 4.4: Robot *diff*, prostředí *jh*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	T [s]	L [m]
`rrt`	1.165	1.332	2.497	100	0,70	21,56
`rrt-bidirect`	81	86	313	100	0,12	21,05
`rrt-connect`	72	0	1	100	0,02	18,66
`rrt-viability`	331	244	575	100	0,19	20,36

Tabulka 4.5: Robot *car*, prostředí *jh*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	T [s]	L [m]
`rrt`	3.326	13.187	16.513	93	2,56	20,03
`rrt-bidirect`	69	127	520	100	0,16	21,44
`rrt-connect`	3.854	21.250	21.252	83	1,65	24,20
`rrt-viability`	3.887	3.633	7.520	100	3,34	18,56

Tabulka 4.6: Robot *diff*, prostředí *jh*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	T [s]	L [m]
`rrt`	4.748	7.361	12.110	97	2,68	33,96
`rrt-bidirect`	46	1.383	5.204	100	1,87	33,30
`rrt-connect`	6.664	10.691	10.692	98	4,24	36,06
`rrt-viability`	4.607	4.209	8.815	100	4,37	33,23

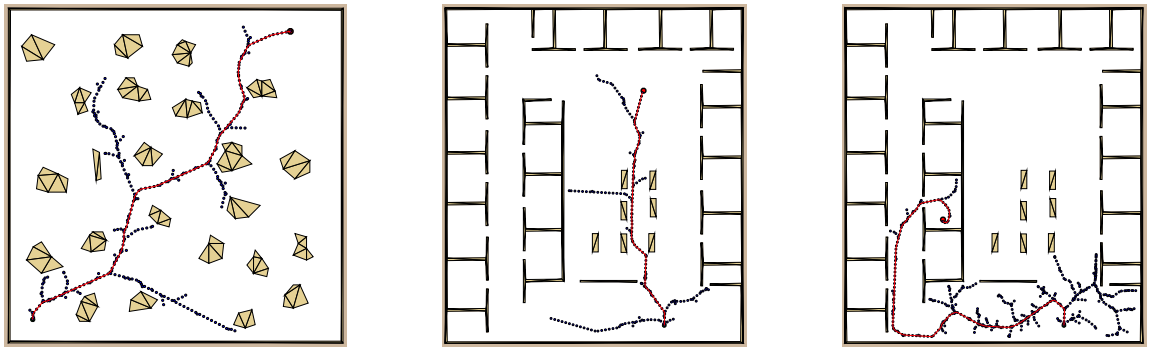
Z provedených experimentů vyplývá, že každý testovaný algoritmus je vhodný pro jiné prostředí a jiný typ robotu. Z hlediska výpočetní náročnosti je nejvhodnějším al-

Tabulka 4.7: Robot *car*, prostředí *jh*

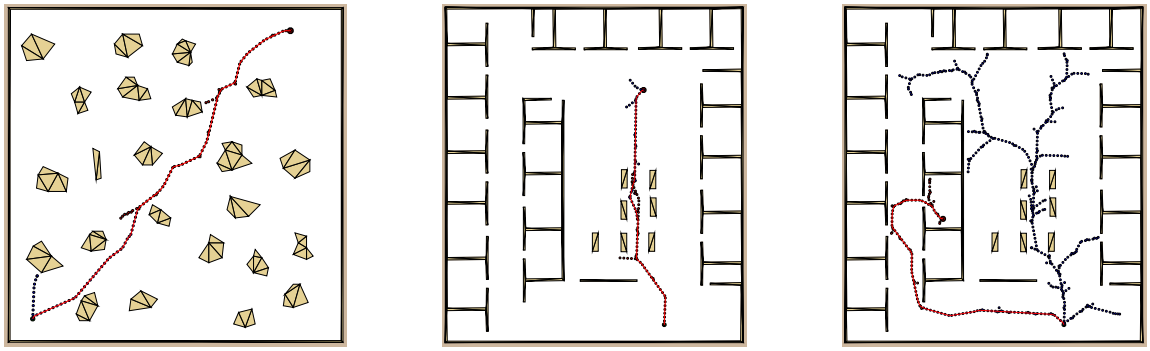
Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	T [s]	L [m]
`rrt`	10.553	54.677	65.231	40	3,32	35,19
`rrt-bidirect`	9	9.776	13.254	12	10,49	33,06
`rrt-connect`	12.795	80.659	80.660	24	6,20	35,19
`rrt-viability`	28.261	29.566	57.827	59	13,85	34,84

goritmem varianta RRT-Bidirect. V tabulce 4.7 je sice u tohoto algoritmu průměrná doba nalezení výsledné trajektorie delší a ani úspěšnost není moc vysoká, ale to je dáno především nevhodnou cílovou konfigurací. Pokud je plánování prováděno v prostředí s malým počtem překážek, jako například experiment vyhodnocený v tabulce 4.4, je velice vhodné použít plánovací techniku RRT-Connect. Plánování je v takovém případě provedeno ve velice krátkém čase a výsledná trajektorie se blíží nejkratšímu řešení. V takových prostředích je také vhodné použít techniku RRT-Bidirect. Z prezentovaných výsledků je dále vidět, že vhodnou informovaností základní techniky je možné dosáhnout také dobrých výsledků.

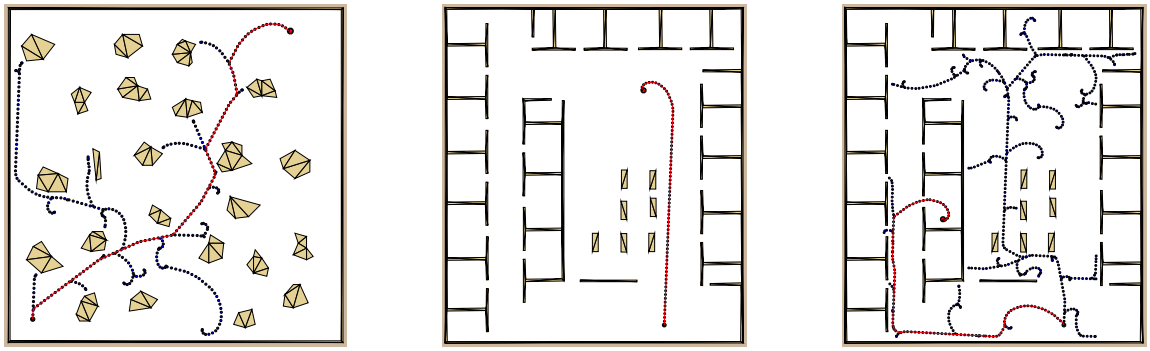
Výhoda ohodnocení stavů v technice RRT-Viability se projevila především u *car-like* robotu při plánování v prostředí *jh*, viz tabulka 4.7. Z porovnaných technik je RRT-Viability v tomto prostředí nejúspěšnějším algoritmem. Doba nalezení trajektorie je sice delší, ale řešení bylo nalezeno s větší úspěšností než v případě ostatních technik. Na druhou stranu má tato technika nevýhodu ve vyšších paměťových nárocích.



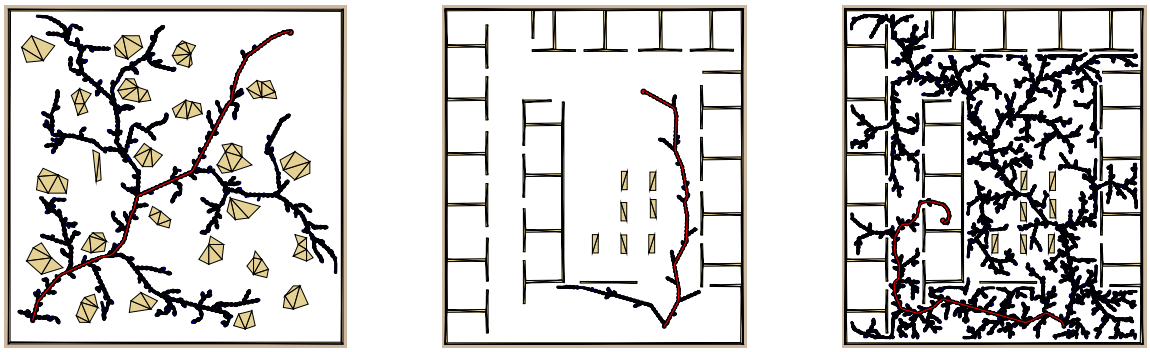
(a) `rrt`, *potholes* (b) `rrt`, *jh*-I (c) `rrt`, *jh*-II



(d) `rrt-bidirect`, *potholes* (e) `rrt-bidirect`, *jh*-I (f) `rrt-bidirect`, *jh*-II

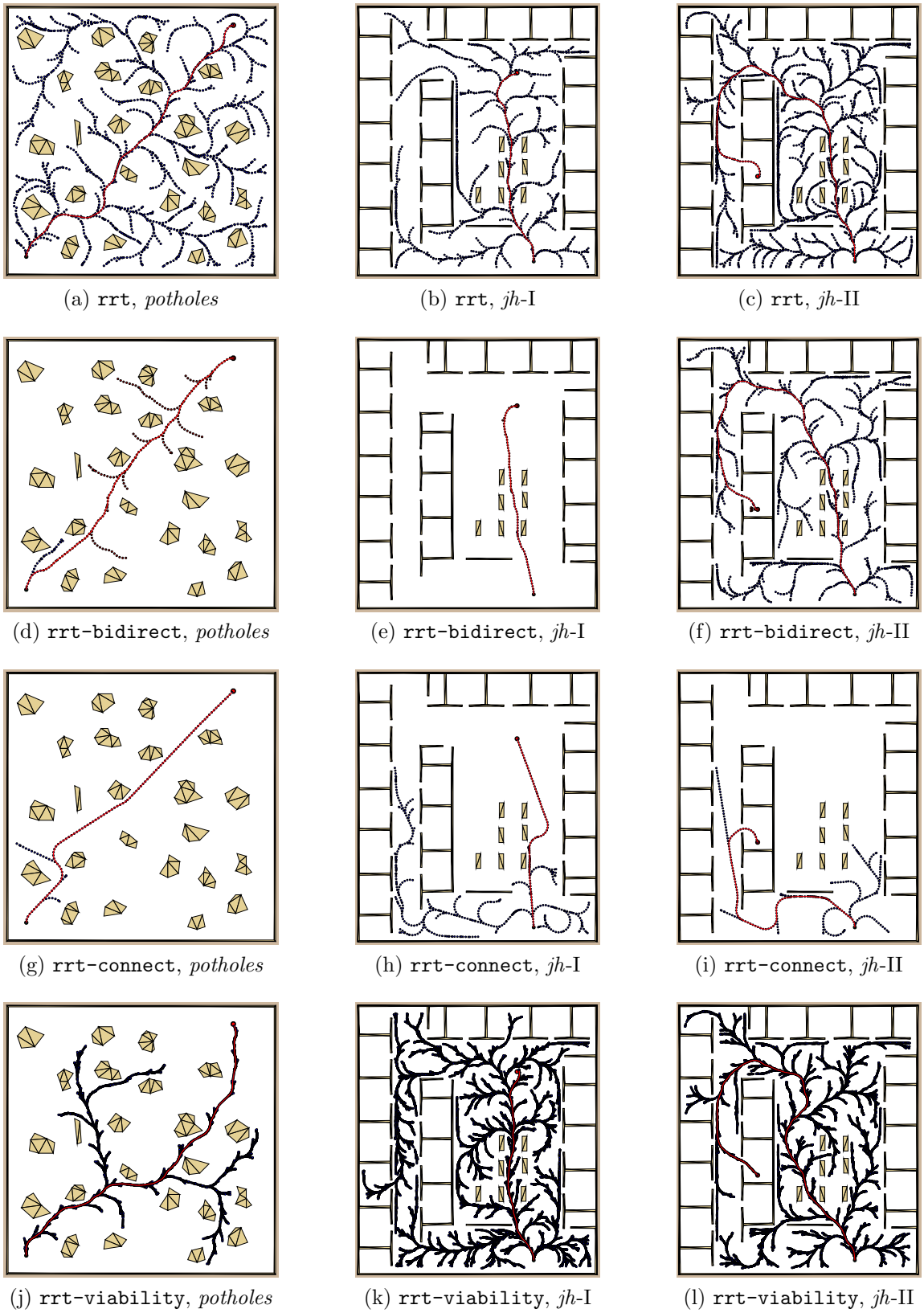


(g) `rrt-connect`, *potholes* (h) `rrt-connect`, *jh*-I (i) `rrt-connect`, *jh*-II



(j) `rrt-viability`, *potholes* (k) `rrt-viability`, *jh*-I (l) `rrt-viability`, *jh*-II

Obrázek 4.2: Příklad řešení pro model robotu *diff*



(a) `rrt`, *potholes* (b) `rrt`, *jh*-I (c) `rrt`, *jh*-II

(d) `rrt-bidirect`, *potholes* (e) `rrt-bidirect`, *jh*-I (f) `rrt-bidirect`, *jh*-II

(g) `rrt-connect`, *potholes* (h) `rrt-connect`, *jh*-I (i) `rrt-connect`, *jh*-II

(j) `rrt-viability`, *potholes* (k) `rrt-viability`, *jh*-I (l) `rrt-viability`, *jh*-II

Obrázek 4.3: Příklad řešení pro model robotu *car*

4.4 Vliv Goal-bias

Dalším experimentem, který byl proveden je studium vlivu informovanosti algoritmu o cílovém stavu (*goal-bias*) na výslednou trajektorii. Experiment byl proveden pouze v prostředí *potholes* a pouze pro jeden robot *car-like*, což je dostačující pro demonstraci vlivu této informace. I zde byl maximální počet iterací nastaven na 100.000 a pro každou hodnotu goal-bias byl algoritmu spuštěn 100 krát. Výsledek je uveden v tabulce 4.8, ve které je uveden počet úspěšných řešení, průměrná doba jejich nalezení (T) a průměrná délka nalezené cesty (L). Hodnota *goal-bias* je značena G_b.

Tabulka 4.8: Vliv Goal-bias, robot *car*, prostředí *potholes*

Metoda	G_b	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	T [s]	L [m]
`rrt`	2	270	939	1.209	100	0,27	26,22
`rrt`	5	591	1.576	2.167	99	0,26	27,02
`rrt`	10	988	2.551	3.539	98	0,34	27,89
`rrt`	20	3.779	12.068	15.847	90	1,45	27,81
`rrt`	50	4.536	13.323	17.859	94	2,83	28,48
`rrt`	100	4.781	12.968	17.749	92	2,37	28,98
`rrt`	200	6.667	19.316	25.984	87	3,35	28,67
`rrt`	500	5.281	13.557	18.838	95	3,26	28,93
`rrt`	1.000	5.754	15.320	21.075	92	3,19	28,88
`rrt`	2.000	5.967	15.878	21.845	92	3,36	28,67
`rrt`	5.000	6.692	17.612	24.304	92	3,98	29,11

Z prezentovaných výsledků je vidět, že pro toto prostředí a tento typ robotu je vhodná větší informovanost. Čím je hodnota *goal-bias* větší tím je průměrná doba nalezení trajektorie delší a průměrná délka nalezené trajektorie je také delší. Počet iterací potřebný k nalezení také roste s hodnotou G_b.

Kapitola 5

Závěr

Cílem této práce bylo seznámení se s úlohou plánování pohybu metodou RRT a jejími rozšířeními. Tyto techniky byli důkladně prozkoumány, implementovány a jejich použití pro různé plánovací úlohy bylo otestováno v několika prostředích a pro různé modely robotů. Pro detekci kolizí bylo nutné se seznámit s knihovnou RAPID, která detekuje kolizi dvou objektů složených z trojúhelníků. Rychlé vyhledání nejbližšího souseda může výrazně zlepšit rychlost nalezení výsledné trajektorie, proto byla k tomuto účelu použita knihovna MPNN využívající strukturu KD-stromu.

Dále je v práci představen algoritmus KPIECE, který nebyl z důvodu předpokládané časové náročnosti implementován. Dostupná implementace algoritmu nebyla plně otestována, neboť adaptace této implementace pro použití vhodné k testování se ukázala jako časově náročná a pravděpodobně by nebyla dokončena v dedikované časové dotaci řešení bakalářské práce.

Každý z uvedených algoritmů je vhodný pro konkrétní prostředí a robot, což je jednak diskutováno v odborné literatuře a také to vyplývá z provedených experimentů. Většina adaptací základní techniky RRT se snaží vyřešit problém úzkých průjezdů, především se jedná o varianty RRT-Bidirect a RRT-Viability. Algoritmus RRT-Connect v modifikaci prezentované v této práci je vhodný spíše pro prostředí s méně překážkami. Z experimentálních výsledků také vyplývá, že základní technika RRT dosahuje dobrých výsledků pokud je vhodně informována o poloze cílového stavu. Technika RRT-Blossom není v experimentech uvedených v části 4.3 prezentována, jelikož v prvotních experimentech se ukázalo, že pro použité modely robotů není tato technika vhodná.

Jednotlivé algoritmy RRT jsou navrženy (implementovány) takovým způsobem, že změna robotu nevyžaduje zásah do samotné plánovací techniky. Pouze se vytvoří příslušná třída reprezentující model robotu, ve které je určen stavový vektor a způsob jakým řídicí vstup mění hodnoty stavu. To je hlavním důvodem, proč je velmi snadné adaptovat tyto algoritmy pro úlohy typu *Alpha Puzzle*. Nepříjemnou vlastností těchto úloh je však velikost vstupního vektoru, který z důvodu vyšší dimenze problému má vyšší paměťové nároky. To se při experimentech projevilo především u algoritmu RRT-Viability, který v jednom uzlu ukládá veliké množství informací což vede na problémy s nedostatkem paměti pro vyřešení úlohy.

Výsledkem plánování je nějaká trajektorie vedoucí z počátečního do cílového stavu. Tato trajektorie je posloupnost bodů jako funkce času, a proto je vhodné tuto

trajektorii vizualizovat, čímž může být ověřena správnost řešení. Především v úloze *Alpha Puzzle*, kde by bylo složité si pod jednotlivými body trajektorie představit způsob jakým byl hlavolam vyřešen, hraje vizualizace důležitou roli. Z tohoto důvodu bylo jedním z cílů bakalářské práce seznámit se s vizualizačním nástrojem VTK, který nabízí sadu nástrojů pro snadnou vizualizaci takové scény jakou je úloha *Alpha Puzzle*. Tato knihovna umožňuje přehrát výsledek hledání trajektorie jako video zobrazující řešení dané úlohy a popřípadě s ním interaktivně manipulovat. Navíc je možné jednotlivé obrázky, nebo i video uložit do některého ze známých formátů jako je například PDF, EPS, PNG, JPG, MPEG a AVI. Použití tohoto nástroje je jednoduché a jelikož je firmou Kitware vyvíjen jako open-source projekt, nabízí se jako vhodné řešení vizualizací v projektech podobných této bakalářské práci. V bakalářské práci byla knihovna VTK použita pro vizualizaci úlohy *Alpha Puzzle*.

Při experimentech byl zjištěn vliv informovanosti o cílové poloze na výslednou trajektorii a na dobu potřebnou k jejímu nalezení. Problémem je však jak tuto hodnotu určit a spíše záleží na citu a znalosti prostředí, ve kterém je plánování prováděno. To vede k zamyšlení, zda by nebylo vhodné měnit tuto informovanost dynamicky podle úspěšnosti rozšiřování stromu.

Zanesení pohybových rovnic reprezentující zde představené roboty nebylo obtížné, ale pokud by se jednalo o složitější model, jehož konfigurační prostor by byl vyšší dimenze, mohla by být reprezentace pohybu tohoto modelu značně složitá. Možným způsobem odstínění tohoto problému je použití knihovny ODE (Open Dynamics Engine), které je možným vhodným tématem dalších prací.

Technika KPIECE byla v této práci pouze představena, podle publikovaných aplikací této techniky se jedná o zajímavý přístup, který poskytuje řešení náročných úloh s více dimenzemi. Z tohoto pohledu je vhodné tuto techniku implementovat a detailně porovnat s již prozkoumanými technikami ve známých plánovacích úlohách jako je *Alpha Puzzle*. Tato technika funguje především díky projekci prostoru vysoké dimenze do nižších jako jsou druhá a třetí dimenze. Proto lze očekávat, že projekce $\mathcal{E}(X)$ by mohla mít také vliv na lepší výsledky technik RRT.

Pole působnosti při zkoumání plánovacích technik je široké, a byla by škoda dále nevyužít vědomosti nabyté řešením této bakalářské práce, proto je mým cílem v započatém studium plánovacích technik pokračovat i v dalších letech mého studia.

Literatura

[1] Şucan, I. A.; Kavraki, L. E.: On the Performance of Random Linear Projections for Sampling-Based Motion Planning. V *IEEE/RSJ International Conference on Intelligent Robots and Systems*, St. Louis, USA, October 2009, s. 2434–2439.

[2] Donald, B.; Xavier, P.; Canny, J.; aj.: Kinodynamic motion planning. *J. ACM*, ročník 40, č. 5, 1993: s. 1048–1066, ISSN 0004-5411.
URL `http://doi.acm.org/10.1145/174147.174150`

[3] Gottschalk, S.; Lin, M. C.; Manocha, D.: OBBTree: A Hierarchical Structure for Rapid Interference Detection. *Computer Graphics*, ročník 30, č. Annual Conference Series, 1996: s. 171–180.

[4] Kalisiak, M.: *Toward More Efficient Motion Planning with Differential Constraints*. Dizertační práce, University of Toronto, 2007.

[5] Kitware, I.: Visualization Toolkit. 2010, [Online].
URL `http://vtk.org`

[6] Kuffner, J. J.; LaValle, S. M.: RRT-Connect: An Efficient Approach to Single-Query Path Planning. V *Proceedings IEEE International Conference on Robotics and Automation*, 2000, s. 995–1001.

[7] Latombe, J. C.: *Robot Motion Planning*. Norwell, MA, USA: Kluwer Academic Publishers, 1991, ISBN 079239206X.

[8] LaValle, S. M.: Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technická zpráva, Computer Science Dept., Iowa State University, oct 1998, tR, 98-11.

[9] LaValle, S. M.: *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.
URL `http://planning.cs.uiuc.edu/`

[10] LaValle, S. M.; Kuffner, J. J.: Randomized Kinodynamic Planning. V *Proceedings IEEE International Conference on Robotics and Automation*, 1999, s. 473–479.

[11] Rusu, R. B.; Sucan, I. A.; Gerkey, B. P.; aj.: Real-Time Perception-Guided Motion Planning for a Personal Robot. V *IEEE/RSJ International Conference on Intelligent Robots and Systems*, St. Louis, 11/10/2009 2009, s. 4245–4252.

[12] Sucan, I.: Open Motion Planning Library. [Online; accessed 6-Červen-2010].
URL http://www.ros.org/wiki/ompl

[13] Sucan, I. A.; Kavraki, L. E.: Kinodynamic Motion Planning by Interior-Exterior Cell Exploration. V *International Workshop on the Algorithmic Foundations of Robotics*, Guanajuato, Mexico, 2008, s. 449–464.
URL http://www.wafr.org/papers/wafr08-sucan.pdf

[14] Vonásek, V.: *Trajectory generation for cooperative inspection task*. Diplomová práce, Czech Technical University in Prague, Czech Republic, 2008, in Czech.

[15] Wikipedia: Drug design — Wikipedia, The Free Encyclopedia. 2010, [Online; accessed 3-May-2010].
URL http://en.wikipedia.org/w/index.php?title=Drug_designi&oldid=359072587

[16] Yershova, A.; LaValle, S. M.: Improving Motion-Planning Algorithms by Efficient Nearest-Neighbor Searching. *IEEE Transactions on Robotics*, ročník 23, č. 1, 2007: s. 151–157.
URL http://dx.doi.org/10.1109/TRO.2006.886840

Příloha A

Obsah CD

Přiložené CD obsahuje zdrojové kódy pro text bakalářské práce ve formátu PDF a zdrojové kódy celého textu pro systém LaTeX. V následující tabulce je popsána struktura CD.

Adresář	Popis
`src`	zdrojové kódy knihovny
`doc`	zdrojové kódy textu bakalářské práce
`logs`	výsledky provedených experimentů
`thesis.pdf`	text bakalářské práce

Tabulka A.1: Adresářová struktura na CD

Příloha B

Výsledky experimentů

Tabulka B.1: Směrodatné odchylky, robot *diff*, prostředí *potholes*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	σ_T [s]	σ_L [m]
`rrt`	1.277	1.050	2.314	100	0,57	2,69
`rrt-bidirect`	23	88	244	100	0,09	2,62
`rrt-connect`	3.422	7.411	7.411	100	2,62	4,94
`rrt-viability`	288	302	564	100	0,25	2,19

Tabulka B.2: Směrodatné odchylky, robot *car*, prostředí *potholes*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	σ_T [s]	σ_L [m]
`rrt`	4.579	16.742	21.226	97	3,16	2,77
`rrt-bidirect`	31	102	408	100	0,12	3,06
`rrt-connect`	6.164	28.797	28.797	94	5,43	4,78
`rrt-viability`	4.550	2.796	7.196	100	4,07	1,95

Tabulka B.3: Směrodatné odchylky, robot *diff*, prostředí *jh*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	σ_T [s]	σ_L [m]
`rrt`	1.911	2.851	4.741	100	1,36	4,37
`rrt-bidirect`	23	89	284	100	0,10	2,38
`rrt-connect`	0	0	0	100	0,00	0,00
`rrt-viability`	158	277	405	100	0,11	2,53

Tabulka B.4: Směrodatné odchylky, robot *car*, prostředí *jh*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	σ_T [s]	σ_L [m]
`rrt`	4.721	24.375	28.997	93	4,63	4,16
`rrt-bidirect`	29	93	414	100	0,12	4,20
`rrt-connect`	5.804	37.858	37.858	83	4,07	6,55
`rrt-viability`	6.663	5.996	12.461	100	5,78	2,24

Tabulka B.5: Směrodatné odchylky, robot *diff*, prostředí *jh*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	σ_T [s]	σ_L [m]
`rrt`	5.050	12.734	17.743	97	2,54	6,60
`rrt-bidirect`	15	1.195	5.188	100	1,85	6,89
`rrt-connect`	7.199	19.067	19.067	98	5,86	6,80
`rrt-viability`	2.343	4.728	6.882	100	3,62	6,59

Tabulka B.6: Směrodatné odchylky, robot *car*, prostředí *jh*

Metoda	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	σ_T [s]	σ_L [m]
`rrt`	5.980	37.553	43.502	40	3,23	5,57
`rrt-bidirect`	3	2.053	13.004	12	7,85	6,85
`rrt-connect`	4.777	36.371	36.371	24	7,00	9,14
`rrt-viability`	22.528	24.441	41.245	59	13,66	5,41

Tabulka B.7: Vliv Goal-bias, směrodatné odchylky, robot *car*, prostředí *potholes*

Metoda	G_b	**počet expanzí**	**počet kolizí**	**počet iterací**	**počet řešení**	σ_T [s]	σ_L [m]
`rrt`	2	270	939	1.209	100	0,18	2,67
`rrt`	5	591	1.576	2.167	99	0,49	2,79
`rrt`	10	988	2.551	3.539	98	0,76	2,62
`rrt`	20	3.779	12.068	15.847	90	3,38	2,36
`rrt`	50	4.536	13.323	17.859	94	5,41	2,31
`rrt`	100	4.781	12.968	17.749	92	3,91	3,01
`rrt`	200	6.667	19.316	25.984	87	4,69	2,66
`rrt`	500	5.281	13.557	18.838	95	4,94	3,04
`rrt`	1.000	5.754	15.320	21.075	92	4,56	3,09
`rrt`	2.000	5.967	15.878	21.845	92	4,91	2,48
`rrt`	5.000	6.692	17.612	24.304	92	4,81	3,01