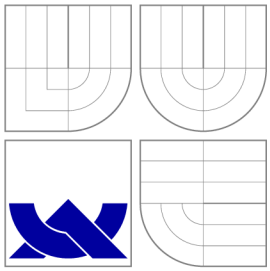
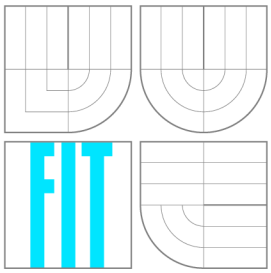


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE HLEDÁNÍ CESTY PRO ROBOTA

VISUALISATION OF PATH-FINDING FOR ROBOT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ZBYNĚK ROUBALÍK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAROSLAV ROZMAN
SUPERVISOR

BRNO 2009

Abstrakt

Tato práce se zabývá hledáním cesty robota za pomocí Bug algoritmů a Potenciálových polí. Jsou zde popsány jednotlivé Bug algoritmy, je zde vysvětlen princip Potenciálových polí a způsob hledání cesty v těchto polích. Součástí této práce jsou také java applety sloužící pro demonstraci funkčnosti a principu těchto algoritmů, je zde uveden návrh i popis implementace a ovládání těchto appletů.

Abstract

This thesis deals with methods of path-finding for robot, rather path-finding with Bug algorithms and Potential fields. The thesis descibes individual Bug algorithms just as theoretical principles of Potential fields and planning paths in this fields. To better demonstration of these algorithms were created java applets. Design, implementation and control over these applets is also described in this thesis.

Klíčová slova

Bug algoritmy, Bug1, Bug2, Tangent Bug, Potenciálová pole, Záplavové vyplňování, Harmonická potenciálová pole, robotika, hledání cesty

Keywords

Bug algorithms, Bug1, Bug2, Tangent Bug, Potential fields, Wave-front algorithm, Harmonic potential field, robotics, path-finding

Citace

Zbyněk Roubalík: Vizualizace hledání cesty pro robota, bakalářská práce, Brno, FIT VUT v Brně, 2009

Vizualizace hledání cesty pro robota

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jaroslava Rozmana.

. . . . . .
Zbyněk Roubalík
20. května 2009

Poděkování

Děkuji svému vedoucímu práce, panu Ing. Jaroslavu Rozmanovi za cenné rady, které mi pomohly při vypracování této bakalářské práce.

Obsah

Kapitola 1

Úvod

Cílem této bakalářské práce je přiblížení problematiky hledání cesty pro robota pomocí Potenciálových polí a Bug algoritmů. V práci jsou vysvětleny základní pojmy nutné k pochopení zadaného tématu, dále pak popis obou přístupů k hledání cesty robotem, tj. kapitola věnující se *Bug algoritmům* a kapitola popisující *Potenciálová pole*. Přesněji řečeno kapitola zabývající se *Bug algoritmy* popisuje použití algoritmu *Bug1*, *Bug2* a *Tangent Bug*, kapitola věnující se *Potenciálovým polím* popisu hledání cesty pomocí *Algoritmu záplavového vyplňování* a *Harmonických potenciálových polí*. Dále je v této práci popsán návrh, implementace a ovládání java appletů, které jsou vytvořeny k předvedení funkčnosti a objasnění principu těchto algoritmů.

Tato práce se sice zabývá hledáním cesty pro robota, ale můžeme si zde nastínit některé další oblasti spojené s robotem, které se hledání cesty nepřímo dotýkají.

Jednou z těchto oblastí je *mapování*, což znamená, že robot je schopen mapovat prostředí, ve kterém se pohybuje. To znamená, že předem nezná prostředí, ve kterém se pohybuje a sestavuje jeho mapu obsahující důležité okolní body, které umožňují robotovi pozdější lokalizaci v prostředí.

Další důležitou oblastí je právě *lokalizace*, což je schopnost robota určit svoji aktuální pozici na mapě podle vzhledu okolního prostředí. Robot musí umět tuto pozici při pohybu správně aktualizovat. K tomuto slouží například metoda *Kalmanův filtr* nebo *Monte Carlo Lokalizace*.

Kapitola 2

Základní pojmy

Tato kapitola definuje pro čtenáře pojem *robot* a uvádí ho do tématiky hledání cesty pro robota.

2.1 Robot

Robot je samostatně pracující stroj, vykonávající určené úkoly [11]. Pro řízení robotů se využívají počítače, které zpracovávají vstupní data ze *senzorů* a rozhodují o jeho dalším stavu. *Senzor* je zařízení, které dokáže měřit nějakou vlastnost vnějšího prostředí [5]. Roboty dělíme na:

- **stacionární roboty** – nemohou se pohybovat z místa na místo, např. průmyslové manipulátory. Většinou jsou složeny z ramena a chapadla, které zajišťuje polohování uchopeného předmětu.
- **mobilní roboty** – mohou se pohybovat v prostoru z místa na místo. Dělí se do dvou podskupin:
  - **dálkově ovládané** – pracují podle instrukcí operátora, často jsou zcela bez inteligence.
  - **autonomní** – na základě instrukcí vykonávají nějakou úlohu, využívají prvky umělé inteligence, dokáží reagovat na vlivy vnějšího prostředí. Všechny tyto úkoly dokážou zvládnout bez pomoci člověka.

V této práci se budeme zabývat právě autonomními roboty, protože dokážou samostatně zpracovávat podněty okolí a reagovat na ně.

2.2 Hledání cesty

Jednou z klíčových úloh, které musíme v souvislosti s roboty řešit, je jejich pohyb v prostoru. Proto musí být robot schopen si naplánovat nebo vyhledat cestu k zadanému cíli. Většinou platí, že robot vykonávající tuto cestu, se do cíle může dostat několika různými způsoby. Reálně nás v nejčastějších případech zajímají následující cíle:

- jak dostat robota z výchozí do cílové pozice,
- jak se má robot vyhnout překážkám, které blokují cestu,
- jak najít nejkratší možnou cestu k cíli,
- jak najít cestu pokud možno co nejrychleji.

Existují algoritmy, které splňují některé nebo i všechny tyto body, nebo naopak nesplňují žádný z nich [7].

2.2.1 Informované hledání cesty

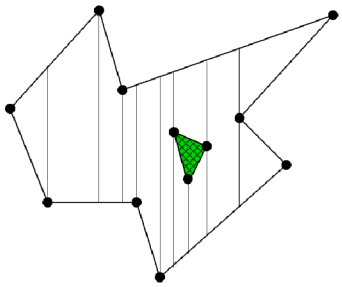
Při hledání cesty robot potřebuje znát místo kam se pohybuje a také má kompletní znalost o okolním prostředí. Robot nejprve na základě mapy vypočítá svoji trasu a poté se začne pohybovat k cíli. Často se pro znázornění prostoru využívají grafy. To znamená, že cesty po kterých se robot může pohybovat zaznačíme jako hrany grafu a průsečíky těchto hran označíme jako uzly tohoto grafu.

Jednou z kategorií algoritmů vyhledávajících cestu je *Exaktní plánování*. To znamená, že pokud existuje cesta k cíli, tak ji algoritmus spadající do této kategorie vždy nalezne. Některé algoritmy mohou být při hledání cesty naopak méně přesné a spolehlivé, např. *pravděpodobnostní algoritmy* [3].

Mezi informované metody patří i *Potenciálová pole*, která si podrobně představíme později. Zde jen krátce zmíníme některé další možné přístupy:

Lichoběžníková dekompozice

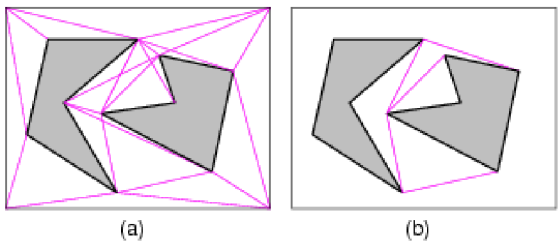
Tento algoritmus dělí volný prostor na lichoběžníky. Pokud je robot a cíl v jednom lichoběžníku, může jít robot jednoduše k cíli. V opačném případě musí najít seznam lichoběžníků, přes které se dostane do cíle, hledá lichoběžníky, které mají společnou hranu a vrcholy.



Obrázek 2.1: Lichoběžníková dekompozice [3]

Graf viditelnosti

Hledá nejkratší cestu mezi pozicí robota a cíle, které společně s vrcholy překážek tvoří vrcholy grafu. Pokud se dva vrcholy grafu vidí, tak se spojí hranou. Cesta je pak hledána přes tyto hrany.



Obrázek 2.2: Graf viditelnosti [3]

Voronoi diagramy

Při použití této metody se pro každou překážku určí body, které jsou k dané překážce nejblíž. Tyto body se spojí a vzniknou tak uzavřené oblasti. Hranice těchto oblastí se nazývají *Voronoi hrany*. Tyto hrany určují místo, ze kterého je vzdálenost k oběma příslušným překážkám stejná. Robot se pak pohybuje po těchto hranách a udržuje si tak maximální vzdálenost k překážkám a tím pádem i volnou cestu k cíli [12].

Obrázek 2.3: Voronoi diagram [12]

Pravděpodobnostní algoritmy

Tento typ algoritmů si dokáže poradit s náročnějšími omezeními na robota. Ten může být nekonvexní, může mít omezenou možnost pohybu nebo může mít určitou dynamiku pohybu (omezená akcelerace, setrvačnost). Pro představu to může být např. robotické otáčející se rameno, autíčko které nemůže jezdit do stran apod. Pro jednoznačný popis pravděpodobnostního plánování cesty je výhodné si definovat *konfigurační prostor* robota, jako n dimenzionální prostor, kde n je počet parametrů jednoznačně definujících pozici robota. Dále pak *volný konfigurační prostor*, což je prostor všech konfigurací, kde robot nekoliduje s žádnou překážkou. Algoritmus se skládá ze dvou fází: generování cesty a hledání cesty v grafu. Nejprve se dle určité pravděpodobnosti náhodně hledají body v prostoru a pokud leží ve volném konfiguračním prostoru, tak se přidají jako vrcholy do grafu. Poté se hledá v tomto grafu cesta, která spojuje pozici robota s cílovou pozicí [4].

2.2.2 Neinformované hledání cesty

Robot nemá žádné informace o okolním prostředí, musí si vše zjišťovat sám. Tím pádem musí řešit obtížnou úlohu – mapování a současnou lokalizaci. Jedním z nejstarších a nejznámějších vyhledávacích metod pracujících v neznámém prostředí jsou *Bug algoritmy*.

Kapitola 3

Bug algoritmy

Bug algoritmy se zabývají hledáním cesty ve zcela neznámém prostředí. Patří k těm nejstarším a nejjednodušším algoritmům, které využívají senzory a mají zároveň prokazatelné výsledky. Bug algoritmy můžeme definovat jako soubory algoritmů pro jednoduché roboty, sloužící k nalezení cesty z počátečního bodu do cílového bodu v prostoru. Tento prostor může obsahovat jednu nebo i více libovolných překážek. Mezi tyto algoritmy patří např. Bug1, Bug2 a Tanget Bug.

Pro zjednodušení se u těchto algoritmů předpokládá, že robot je pouze bod ve zkoumaném prostoru. Pokud robot pro detekci překážek využívá dotekový senzor nebo senzor s nulovým dosahem, použije se Bug1 nebo Bug2 algoritmus. Jestliže využívá senzor s konečným nenulovým dosahem, tak je výhodné použít Tanget Bug algoritmus.

Bug algoritmy se dají poměrně přímočaře implementovat, navíc je zaručeno, že pokud existuje řešení, vždy ho naleznou. Tyto algoritmy vyžadují, aby robot znal svou pozici na mapě. Dále by pak měl být schopen určit směr a vzdálenost k cíli z bodu, kde se právě nachází. Jediné chování, které by měl robot ovládat je pohyb směrem k cíli a pohyb podél překážky.

3.1 Algoritmus Bug1

Algoritmus Bug1 představuje to nejjednodušší a nejpřímější řešení problému navigace a pohybu robota v prostoru. Základní myšlenka je taková, že se robot pohybuje směrem k cíli dokud nenarazí na překážku, tu celou objede a nalezne tak nejlepší pozici pro další pohyb do cílového bodu.

3.1.1 Popis

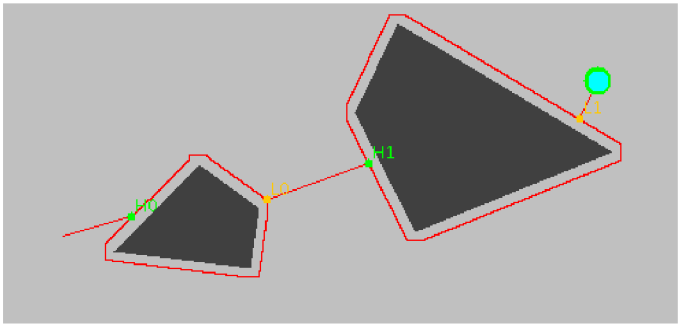
Už jsme zmínili, že robot představuje bod v rovině a zná přesně souřadnice své pozice, dále má dotykový senzor, který dokáže odhalit překážku pokud se jí robot „dotkne“. Robot by měl také umět změřit vzdálenost $d(x, y)$ mezi dvěma libovolnými body x a y. Konečně také můžeme předpokládat, že prostor, ve kterém se robot pohybuje, je ohraničený.

Startovací bod označíme jako S, cílový jako G. Budiž $L_0 = S$ a úsečka m takovou, která spojuje jednotlivé segmenty L_i do G. Počátečně je $i = 0$.

Robot u Bug1 využívá dva druhy chování:

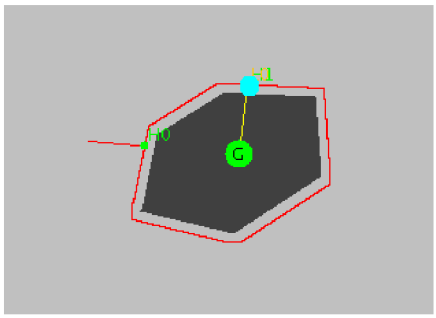
- pohyb směrem k cíli
- pohyb podél překážky

Během pohybu k cíli se robot pohybuje podél úsečky m k bodu G, dokud nedorazí do cíle nebo nenarazí do překážky. Jestliže narazí do překážky, uznačí si tento bod jako H_i (bod dotyku,



Obrázek 3.1: Algoritmus Bug1 úspěšně nalezl cíl

angl. hit point). Např. první takovýto bod si označí jako H_0, robot začne objíždět překážku dokud se nedostane znovu do bodu H_0. Poté určí z trasy vykonané okolo překážky nejkratší bod k cíli a označí si ho jako L_i (bod opuštění, angl. leave point), tomto případě je $i = 0$. Z tohoto bodu L_0 se začne robot opět pohybovat směrem k cíli G. Jestliže úsečka m vedoucí z bodu L_0 do G protíná aktuálně objížděnou překážku, tak neexistuje cesta k cíli. K tomuto protnutí musí dojít okamžitě po opuštění bodu L_0, jinak se i inkrementuje a celá procedura objíždění překážky se opakuje pro nové body L_i a H_i, dokud robot nedorazí do cíle nebo určí, že cíl je nedosažitelný.



Obrázek 3.2: Algoritmus Bug1 ohlásil, že cíl je nedostupný

3.1.2 Pseudokód

Následující pseudokód ukazuje hledání cesty robota při použití algoritmu Bug1.

Algoritmus 3.1.1. Algoritmus Bug1 [2]

Vstup: Robot s dotykovým senzorem jako bod v rovině
Výstup: Cesta k cíli G nebo řešení, že cesta k cíli neexistuje.

1: **while true do**
2: **repeat**
3: Pohybuj se od bodu L_{i-1} směrem k cíli G.
4: **until** není dosažen G **or** narazilo se do překážky v bodě H_i.
5: **if** Je dosaženo cíle **then**
6: Konec.
7: **end if**
8: **repeat**
9: Pohybuj se podél překážky
10: **until** není dosažen G **or** je znovu dosaženo bodu H_i.
11: Urči bod L_i, který je nejblíže k cíli G.
12: Jdi do určeného bodu L_i.
13: **if** Robot se pohybuje směrem k cíli do překážky **then**
14: Konec - cíl je nedosažitelný.
15: **end if**
16: **end while**

3.1.3 Zhodnocení

Pro zhodnocení tohoto algoritmu je výhodné použít odhad maximální možné délky cesty k cíli. Pro výpočet tohoto odhadu můžeme použít následující vzorec:

$$Length_{Bug1} \leq d(S,G) + 1.5 \sum_i p_i \tag{3.1}$$

kde $d(S,G)$ udává přímou vzdálenost ze startu do cíle a p_i udává obvod i-té překážky. Tento obvod musíme ještě vynásobit koeficientem 1.5, protože robot nejprve objede celou překážku a hledá při tom nejlepší bod L. Poté se musí do tohoto bodu dostat a délka této části trasy může být právě nejvíce polovina obvodu dané překážky.

Minimálním odhadem možné délky cesty je právě vzdálenost $d(S,G)$, tato možnost nastane za situace, že robot při cestě k cíli nenarazí na žádnou překážku.

Algoritmus Bug1 provádí *úplné prohledávání* (angl. exhaustive search) pro nalezení optimálního L, tj. bodu opuštění. To vyžaduje, aby robot obešel celou překážku a nalezl nejlepší bod L. Tuto metodu je výhodné použít, pokud je překážka členitá, potom dává algoritmus Bug1 nejlepší výsledky.

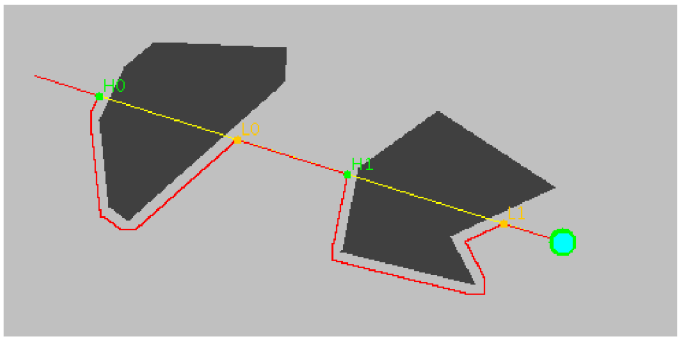
3.2 Algoritmus Bug2

U tohoto algoritmu také předpokládáme, že robot je bod v prostoru a pro detekci překážek je vybaven dotykovým senzorem nebo senzorem s nulovým dosahem. Podobně jako Bug1 využívá i algoritmus Bug2 dva druhy chování robota:

- pohyb směrem k cíli
- pohyb podél překážky

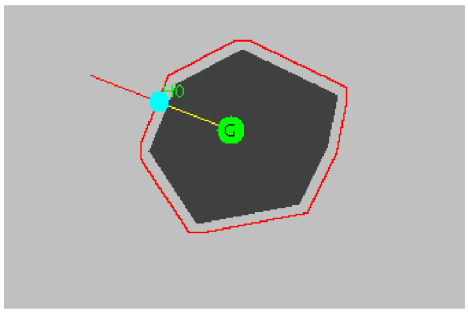
3.2.1 Popis

V základě jsou si tyto algoritmy podobné, přesto mají několik odlišností. Během pohybu k cíli se robot pohybuje podél úsečky m, která u algoritmu Bug2 zůstává po celou dobu pevně daná. Tato úsečka vede ze startovacího bodu S do cíle G. Robot se rozjede z bodu S podél úsečky m směrem k cíli, dokud se nedostane úspěšně do cíle nebo narazí na překážku. Pokud na ninarazí, tak si tento bod označíme podobně jako u Bug1 algoritmu – H_i. Poté robot začne tuto překážku objíždět na libovolnou stranu.



Obrázek 3.3: Algoritmus Bug2 úspěšně nalezl cíl

Také pohyb okolo překážky má jiné chování než u Bug1 algoritmu. Robot pokračuje v objíždění překážky, dokud se nedostane do takového bodu na úsečce m, který je blíž k cíli G než k bodu H_i, tj. k bodu prvotního doteku s překážkou. Tento bod si označíme jako L_i a robot se opět začne pohybovat k cíli. Narazí-li na další překážku, tak se tato procedura opět opakuje. Jestliže se ale opět dostane do původního bodu H_i, znamená to, že neexistuje cesta, která by vedla k cíli.



Obrázek 3.4: Algoritmus Bug2 ohlásil, že cíl je nedostupný

3.2.2 Psedokód

Následující pseudokód popisuje princip Bug2 algoritmu.

Algoritmus 3.2.1. Algoritmus Bug2 [2]

Vstup: Robot s dotykovým senzorem jako bod v rovině
Výstup: Cesta k cíli G nebo řešení, že cesta k cíli neexistuje.

```
1:  while true do
2:      repeat
3:          Pohybuj se od bodu L_{i-1} směrem k cíli G
              podél úsečky m.
4:      until není dosažen G or
            narazilo se do překážky v bodě H_i.
5:      Otoč se doleva (nebo doprava).
6:      repeat
7:          Pohybuj se podél překážky
8:      until není dosažen G or
            je znovu dosaženo bodu H_i or
            narazilo se na úsečku m v takovém bodě M,
            kde M! = H_i and
            d(M,G) < D(M,H_i) (robot je blíž k cíli) and
            se robot pohybuje k cíli a ne do překážky.
9:      Nastav: L_{i+1} = M
10:     i++
11: end while
```

3.2.3 Zhodnocení

I při hodnocení tohoto algoritmu můžeme využít odhad maximální cesty k cíli, který má poněkud složitější výpočet než algoritmus Bug1:

$$Length_{Bug2} \leq d(S,G) + 0.5 \sum_i n_i p_i \tag{3.2}$$

kde $d(S,G)$ je opět vzdálenost startu a cíle, p_i obvod i-té překážky a n_i je počet bodů opuštění i-té překážky. Za předpokladu, že přímka ze startovní pozice do cíle protíná i-tou překážku právě n_i krát, tak můžeme určit, že daná překážka obsahuje maximálně n_i bodů L. Z tohoto počtu, ale musíme odečíst polovinu, protože polovina těchto bodů nejsou platnými body opuštění dané překážky – kdyby se robot pohyboval z tohoto bodu směrem k cíli, narazil by hned do dané překážky. Jinými slovy můžeme právě tuto polovinu považovat za body dotyku H, můžeme to vidět např. na obrázku 3.3. Takže v nejhorším případě objede robot skoro celý obvod překážky pro každý bod opuštění L.

Minimálním odhadem možné délky cesty je podobně jako u Bug1 vzdálenost $d(S,G)$.

Algoritmus Bug2 používá oportunistický přístup (angl. opportunistic approach) k nalezení nejlepšího bodu opuštění L. Pokud algoritmus najde takové L, které je lepší než všechny, které nalezl před tím, tak ho použije. Takový algoritmus se nazývá *chamtivý* (angl. greedy). Tento algoritmus dává uspokojivé výsledky, pokud jsou překážky jednoduché – nemají složitý tvar.

3.3 Algoritmus Tangent Bug

Tangent Bug je vylepšením algoritmu Bug2, které dokáže najít kratší cestu k cíli za využití senzoru s konečným nebo nekonečným dosahem a s rozsahem 360°. Tento senzor dokáže určit, zda-li se v jeho dosahu nachází nějaká překážka.

3.3.1 Popis

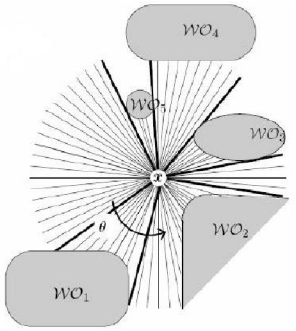
Senzor robota můžeme popsat *funkcí vzdálenosti* $\rho : \mathbb{R}^2 \times S^1 \rightarrow \mathbb{R}$. Střed robota je situován v $x \in \mathbb{R}^2$ a paprsky senzoru vychází z tohoto bodu. Pro každý $\theta \in S^1$ je hodnota funkce $\rho(x, \theta)$ rovna vzdálenosti k nejbližší překážce $\mathcal{WO}_i$ z bodu x pod úhlem θ. Formálněji to můžeme zapsat takto [2]:

$$\rho(x, \theta) = \min d(x, x + \lambda[\cos\theta, \sin\theta]^T), kde\ x + \lambda[\cos\theta, \sin\theta]^T \in \bigcup_i \mathcal{WO}_i \tag{3.3}$$

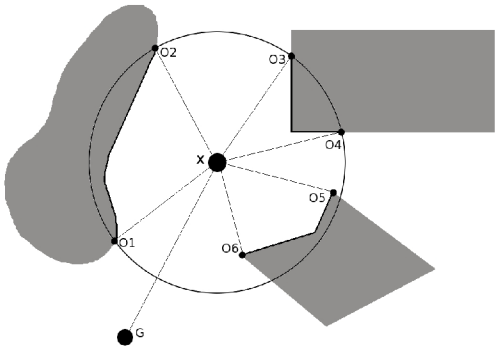
Takto definovaná funkce má nekonečný dosah, protože však mají senzory v reálné situaci omezený dosah, definujme si funkci $\rho_R : \mathbb{R}^2 \times S^1 \rightarrow \mathbb{R}$, která má stejné hodnoty jako ρ, pokud jsou překážky v dosahu senzoru. Jestliže funkce vrátí nekonečno, znamená to, že pod daným úhlem θ není v dosahu senzoru R žádná překážka. [2]

$$\rho_r(x, \theta) = \begin{cases} \rho(x, \theta) & \rho(x, \theta) < R \\ \infty & jinak \end{cases} \tag{3.4}$$

U algoritmu Tangent Bug je předpoklad, že robot dokáže rozpoznat všechny nespojitosti pro aktuální pozici, můžeme to vidět na obrázku 3.5. Definujeme proto pro bod $x \in \mathbb{R}^2$ *interval spojitosti* (angl. interval of continuity), který obsahuje takové body $x + \rho(x, \theta)[\cos\theta, \sin\theta]^T$ v prostoru, kde je funkce $\rho_R(x, \theta)$ konečná a spojitá vzhledem k θ. To znamená, že pod tímto úhlem θ nalezl senzor robota překážku. Krajní body těchto intervalů se nachází tam, kde funkce $\rho_R(x, \theta)$ přestává být spojitá, jako důsledek toho, že pod úhlem θ nenalezl senzor ve svém dosahu žádnou překážku. Tyto body značíme jako O_i, jak je např. znázorněno na obrázku 3.6.



Obrázek 3.5: Tenké čáry jsou hodnoty $\rho_R(x, \theta)$ pro $x \in \mathbb{R}^2$, silné značí nespojitosti intervalů [2].



Obrázek 3.6: Body O_i znázorňují, kde přestává být funkce $\rho_R(x, \theta)$ spojitá. Silná čára na hranicích překážek znázorňuje interval spojitosti [2].

Podobně jako ostatní Bug algoritmy i Tangent Bug využívá dva typy chování robota:

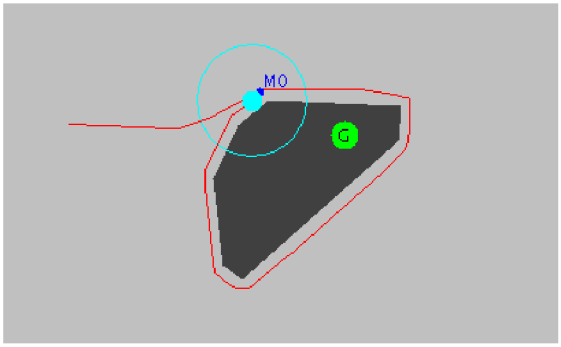
- pohyb směrem k cíli
- pohyb podél překážky

Nicméně jsou tato chování jiná, než jak je známe u Bug1 nebo Bug2 algoritmu. Pohyb směrem k cíli navádí robota k cíli, ale zároveň ho i může navádět okolo překážky. Stejně tak se může robot při pohybu okolo překážky dostat do fáze, kdy se podél překážky nepohybuje.

Pohyb robota směrem k cíli má sám o sobě dvě části. Nejprve se robot pohybuje po přímce směrem k cíli, dokud nerozpozná překážku ležící mezi ním a cílem. To znamená, že přímka spojující pozici robota a cíl, protíná interval spojitosti. Pohybuje se dopředu, až bude schopen urči koncové body intervalu spojitosti O_i u dané překážky. Poté se robot pohybuje směrem k takovému O_i, které nejvíce zmenšuje předem zvolenou heuristickou funkci vzdálenosti k cíli. Tato funkce může být např. $d(x, 0_i) + d(O_i, G)$[2], která sčítá vzdálenost robota od bodu O_i, ke kterému se pohybuje, se vzdáleností od tohoto bodu k cíli.

Množina $\{O_i\}$ se pravidelně aktualizuje podle toho, jak se robot pohybuje. Jakmile robot už nemůže dále zmenšovat heuristickou funkci vzdálenosti k cíli, tak přejde do pohybu podél překážky, tento bod si také můžeme označit jako lokální minimum M této heuristické funkce vzdálenosti. Robot pokračuje v pohybu okolo překážky stejným směrem jako při předchozím pohybu k cíli. Pohybuje se směrem k bodu O_i na objížděné překážce a zároveň aktualizuje hodnoty d_{reach} a $d_{followed}$. Hodnota $d_{followed}$ je nejkratší vzdálenost mezi cílem a hranicí překážky, která byla doposud senzorem robota prozkoumána. Hodnota d_{reach} je pak vzdálenost mezi cílem a nejbližším bodem objížděné překážky, který je v dosahu senzoru robota.

Jakmile je hodnota d_{reach} menší než $d_{followed}$, robot ukončí objíždění překážky a vydá se opět směrem k cíli.



Obrázek 3.7: Algoritmus Tangent Bug ohlásil, že cíl je nedostupný

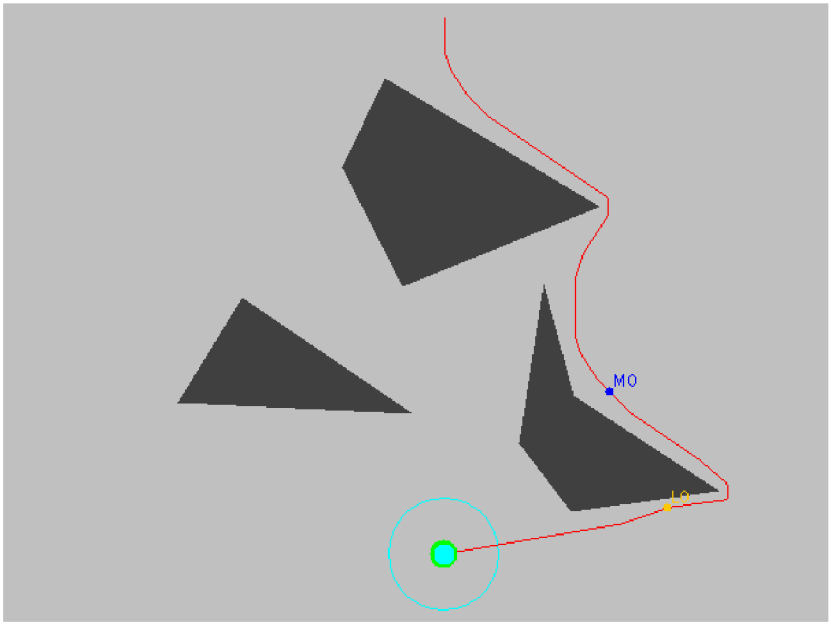
3.3.2 Psedokód

Následující pseudokód ukazuje chování robota při použití Tangent Bug algoritmu. Pro lepší porozumění algoritmu si ještě definujme bod T, který leží na kružnici se středem x a poloměrem R a zároveň leží na úsečce spojující bod x a cílový bod G. T tak vlastně představuje nejbližší bod k cíli, který dokáže robot zjistit v dosahu svého senzoru.

Algoritmus 3.3.1. Algoritmus Tangent Bug [2]

Vstup: Robot se senzorem jako bod v rovině
Výstup: Cesta k cíli G nebo řešení, že cesta k cíli neexistuje.

```
while true do
    repeat
        Pohybuj se k bodu n ∈ {T, O_i},
          tak aby se zmenšovala hodnota d(x,n) + d(n,G)
    until není dosažen G or směr pohybu,
        který zmenšuje d(x,n) + d(n,G) začal zvětšovat
        d(x,G) (robot nalezl lokální minimum)
    Zvol takový směr objíždění překážky,
        který bude pokračovat v pohybu ve stejném směru
        jako poslední pohyb směrem k cíli.
    repeat
        Aktualizuj d_reach a d_followed a {O_i}.
        Pohybuj se směrem ke zvolenému bodu a {O_i}.
    until není dosažen G or
        robot objel celou překážku ⇒ G je nedostupný or
        d_reach < d_followed
end while
```



Obrázek 3.8: Algoritmus Tangent Bug úspěšně nalezl cíl

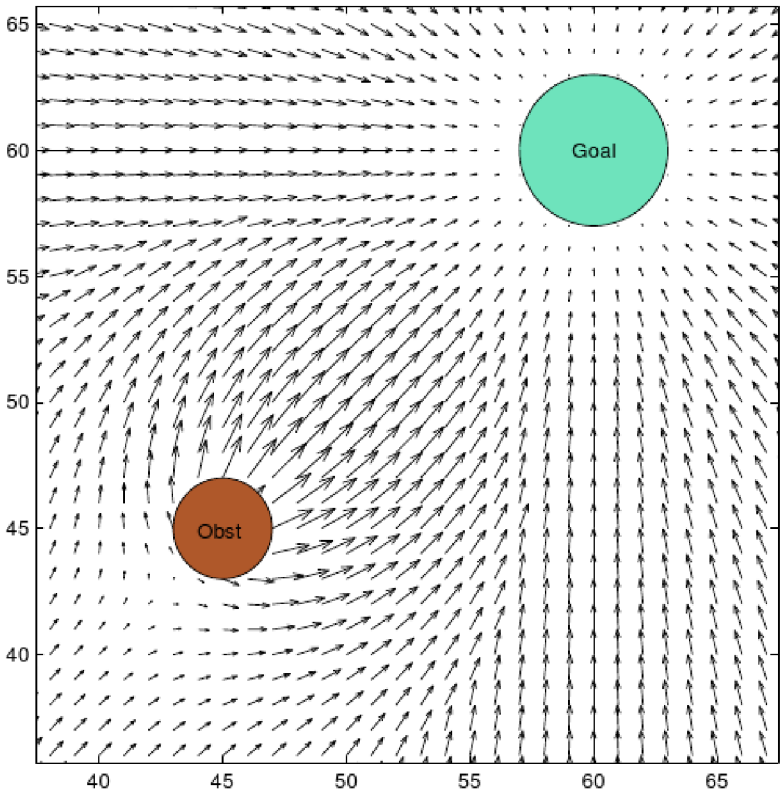
3.3.3 Zhodnocení

U tohoto algoritmu je odhad délky trasy velmi obtížný, protože si robot různě zkracuje cestu při pohybu okolo překážek, např. při nájezdech k vrcholům překážky. Právě díky těmto zkracováním trasy, objížděním překážky jen tolik, kolik je v dané situaci nutné a hledáním co nejvýhodnější cesty k cíli, se stává tento algoritmus v praxi použitelnějším než předcházející bug algoritmy. Použitím nějaké složitější heuristické funkce, pro hledání ideálního směru pohybu okolo překážky, by tento algoritmus mohl dávat ještě zajímavější výsledky.

Kapitola 4

Potenciálová pole

Pro hledání cesty pro robota ve známém prostředí můžeme využít potenciálová pole. Navigace robota pomocí potenciálových polí je založena na tom, že si můžeme představit, že okolí působí na robota silami, které ho různě přitahují a odpuzují (podobně jako třeba magnetické pole). Působení těchto sil pak usměrňuje robotův pohyb prostředím, pomáhá mu vyhýbat se překážkám, přibližovat se k cíli apod. Reprezentace těchto jednotlivých sil v prostředí pak můžeme považovat za potenciálové pole.



Obrázek 4.1: Potenciálové pole vygenerované robotem v prostoru, kde se nachází překážka *Obst* a cíl *Goal* [6].

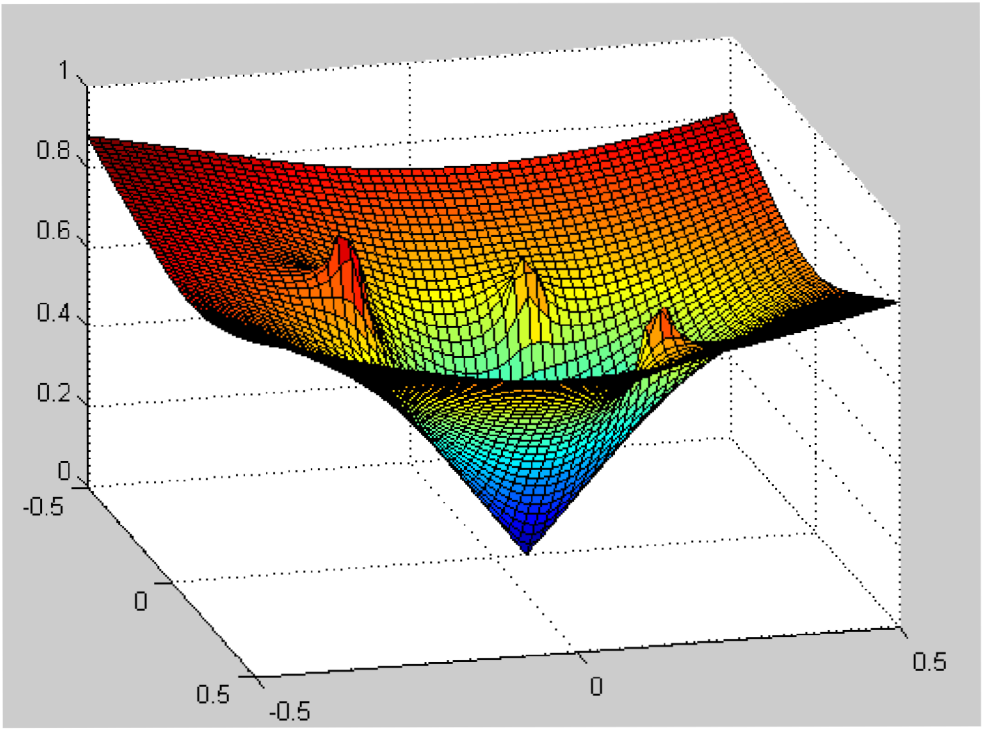
4.1 Úvod

V této části si definujeme některé pojmy, které jsou důležité pro pochopení tématiky potenciálových polí.

4.1.1 Potenciálová funkce

Potenciálová funkce je funkce U, na jejíž hodnotu můžeme nahlížet jako na energii v daném bodě. Směr působení síly této energie určuje *gradient*. Formálně:

$$U : \mathbb{R}^m \rightarrow \mathbb{R} \tag{4.1}$$



Obrázek 4.2: Potenciálová funkce, cíl leží uprostřed a tři lokální maxima značí oblasti, kde se nachází překážky [8].

4.1.2 Gradient

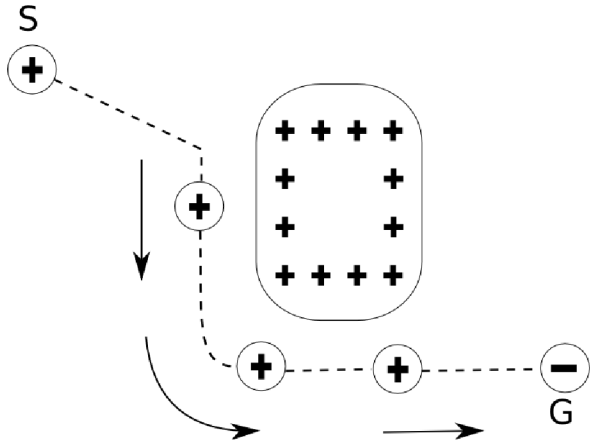
Podle [10] je gradient v obecném smyslu slova směr růstu. V kontextu potenciálových polí ho můžeme považovat za vektor, který míří ve směru nejvíce zvětšujícím U na dané pozici. Při zápisu se používá operátor nabla: ∇.

$$\nabla U(q) = [\frac{\partial U}{\partial q_1}(q), \ldots, \frac{\partial U}{\partial q_m}(q)] \tag{4.2}$$

4.1.3 Pohyb v potenciálových polích

Pro využití gradientu můžeme definovat *pole vektorů*, což je takové pole, kde je každému bodu přiřazen vektor. Poté můžeme říct, že *gradientní pole vektorů* přiřazuje gradient dané funkce každému bodu [2].

Můžeme tak říci, že pohyb robota pomocí potenciálové funkce se dá připodobnit pohybu v gradientním poli vektorů. Na gradienty tak můžeme nahlížet jako na síly působící na kladně nabitého robota, který je přitahován záporně nabitým cílem. Překážky na cestě mohou mít také kladný náboj, to způsobí, že kladně nabitý robot je odpudivými silami veden dál od překážek. Tato kombinace kladných a záporných sil zajistí bezpečný pohyb robota prostředím (obrázek 4.3).



Obrázek 4.3: Záporné síly přitahují kladně nabitého robota k cíli, kladné ho odpuzují od překážek.

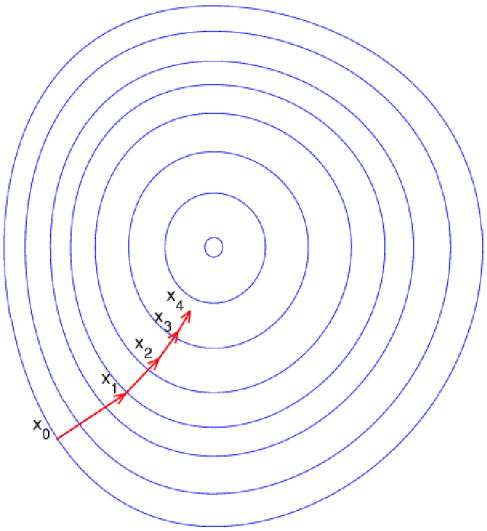
4.1.4 Sestupný gradient

Potenciálová pole si můžeme představit jako prostor, kde se robot pohybuje z bodu vyšší hodnoty potenciálové funkce do bodu hodnoty nižší. Robot se tak vlastně pohybuje ve směru znegovného gradientu potenciálové funkce, tato metoda se nazývá *sestupný gradient* (angl. gradient descent)[2].

Metoda sestupného gradientu se hojně využívá při optimalizaci různých problémů. Spočívá v tom, že se robot ze své počáteční pozice posune o malý krok ve směru oproti gradientu. Tak se robot dostane do nové pozice a celý proces se znovu opakuje. Formálněji je popsán tento algoritmus v následujícím pseudokódu, kde $q(i)$ odpovídá hodnotě bodu q v i-té iteraci tohoto algoritmu. Hodnota $\alpha(i)$ představuje velikost kroku v i-té iteraci. Volba kroku $\alpha(i)$ může být provedena např. podle vzdálenosti robota od cíle. Existuje však více možností volby $\alpha(i)$, které však pro naši tématiku nejsou podstatné.

Algoritmus 4.1.1. Algoritmus sestupného gradientu [2]

Vstup: Výpočet gradientu $\nabla U(q)$ v bodě q
Výstup: Sekvence bodů $\{q(0), q(1), \ldots, q(i)\}$.
1: $q(0) = S$
2: $i = 0$
3: **while** $\nabla U(q(i)) \neq 0$ **do**
4: $q(i+1) = q(i) + \alpha(i)\nabla U(q(i))$
5: $i++$
6: **end while**



Obrázek 4.4: Ilustrace metody sestupného gradientu [9]. Jednotlivé iterace jsou zde zaznačeny jako $x1, x2, \ldots, x4$.

4.1.5 Kritický bod

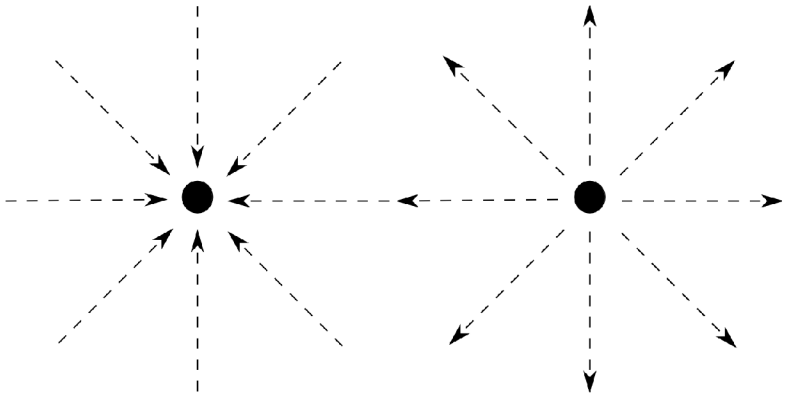
Robot se přestane pohybovat, jakmile dosáhne bodu, kde gradient nemá žádnou hodnotu. To znamená, že dosáhl bodu q^*, kde $\nabla U(q^*) = 0$. Tento bod se nazývá *kritický bod* potenciálové funkce U. Kritický bod může být lokální maximum, lokální minimum nebo sedlový bod dané funkce.

Pro metody, který využívají sestupný gradient je zřejmé, že kritický bod bude vždy lokální minimum. Protože sestupný gradient vždy snižuje U, tak se robot nemůže dostat do lokálního maxima. Jediná možnost je, že by robot svůj pohyb začínal v lokálním maximu, ale pak sestupný gradient zajistí, že se robot z tohoto bodu bezpečně dostane. Stejně tak je nestabilní sedlový bod a není možnost, že by tam mohl robot uváznout.

Protože už víme, že jediný kritický bod, kde by mohl robot ukončit svůj pohyb, je lokální minimum, můžeme tak předpokládat, že právě v lokálním minimu se nachází cílový bod.

4.2 Přitažlivé a odpudivé potenciály

Existuje mnoho metod jak pomocí potenciálových funkcí hledat cestu robota. Některé mohou být sice výpočetně velmi rychlé, ale všechny tyto metody kromě přitažlivých a odpudivých potenciál trpí jedním problémem – lokální minimum těchto funkcí nemusí odpovídat cílovému bodu [2]. To znamená, že tyto metody nemusí robota dovést do cíle a tím pádem robot nesplní požadovaný úkol. Pro řešení tohoto problému můžeme využít potenciálová pole ve spojení s nějakým algoritmem na prohledávání prostoru, což je ale výpočetně drahé, nebo použijeme takové metody, které mají pouze jedno lokální minimum – což jsou právě přitažlivé a odpudivé potenciály.



Obrázek 4.5: Pole přitažlivých vektorů (vlevo) a odpudivých vektorů (vpravo).

Přitažlivé a odpudivé potenciály patří k nejjednodušším potenciálovým funkcím, jsou založeny na jednoduché myšlence: cíl robota přitahuje a překážky ho odpuzují. Součet těchto přitažlivých U_{att} a odpudivých U_{rep} sil nám dává příslušnou potenciálovou funkci.

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{4.3}$$

4.2.1 Přitažlivé potenciály

Přitažlivé potenciálové pole (angl. attractive potential) U_{att} by mělo monotónně zvyšovat svou hodnotu úměrně se vzdáleností od cíle. Můžeme to zaručit použitím *kónické potenciály*, kdy U_{att} definujeme jako váženou vzdálenost k cíli, $U(q) = \zeta d(q, G)$. Parametr ζ představuje váhu přitažlivé potenciály a gradient si definujeme jako $\nabla U(q) = \frac{\zeta}{d(q,G)}(q - G)$. To znamená, že vektor představující gradient směruje od cíle a v každém bodě má určitou velikost ζ. Poté pokud robot bude startovat v libovolném bodě a bude se pohybovat proti směru gradientu, dostane se vždy do cíle.

Podle [2] se při implementaci této metody mohou objevit problémy s nespojitostí od určité vzdálenosti od cíle. Proto je lepší použít takovou metodu, která postupně snižuje velikost gradientu, tak jak se robot přibližuje k cíli. Nejjednodušší je použít potenciálovou funkci, jejíž hodnota roste kvadraticky se vzdáleností od cíle G:

$$U_{att}(q) = \frac{1}{2}\zeta d^2(q, G) \tag{4.4}$$

a gradient si vyjádříme jako:

$$\nabla U_{att}(q) = \nabla(\frac{1}{2}\zeta d^2(q,G)), \tag{4.5}$$

$$\nabla U_{att}(q) = \frac{1}{2}\zeta \nabla d^2(q,G), \tag{4.6}$$

$$\nabla U_{att}(q) = \zeta(q-G) \tag{4.7}$$

Takto je gradient definován jako vektor z bodu q směřující od cíle a má velikost úměrnou vzdálenosti od cíle. To znamená, že čím je robot blíže k cíli, tím se k němu přibližuje pomaleji, protože tam má gradient menší velikost. Naopak ve velkých vzdálenostech od cíle je to nežádoucí, protože rychlost by byla moc vysoká, proto se podle [2] doporučuje použít kombinaci kónického a kvadratického potenciálu. To znamená, že ve vzdálenosti od cíle větší než zvolený práh d^*_G by robota přitahoval k cíli kónický potenciál a ve vzdálenosti menší nebo rovné než d^*_G kvadratický potenciál. Formálně tato funkce a gradient:[2]

$$U_{att}(q) = \begin{cases} \frac{1}{2}\zeta d^2(q,G) & d(q,G) \leq d^*_G \\ d^*_G \zeta d(q,G) - \frac{1}{2}\zeta (d^*_G)^2 & d(q,G) > d^*_G \end{cases} \tag{4.8}$$

$$\nabla U_{att}(q) = \begin{cases} \zeta(q-G) & d(q,G) \leq d^*_G \\ \frac{d^*_G \zeta(q,G)}{d(q,G)} & d(q,G) > d^*_G \end{cases} \tag{4.9}$$

4.2.2 Odpudivé potenciály

Odpudivé potenciálové pole (angl. repulsive potential) $U_{rep}(q)$ udržuje robota v potřebné vzdálenosti do překážek. Podle názvu je nejspíše jasné, že jsou protikladem přitažlivých potenciál, takže v jejich vlastnostech nacházíme určitou analogii, samozřejmě ale obrácenou. To znamená že odpudivé síly závisí na vzdálenosti robota od překážky – čím je blíž k překážce, tím větší je odpudivá síla, která robota navádí od této překážky. Podle [2] se odpudivé potenciály obvykle definují jako vzdálenost od nejbližší překážky $D(q)$, formálně:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{D(q)} - \frac{1}{Q^*})^2 & D(q) \leq Q^* \\ 0 & D(q) > Q^* \end{cases} \tag{4.10}$$

$$\nabla U_{rep}(q) = \begin{cases} \eta(\frac{1}{Q^*} - \frac{1}{D(q)})\frac{1}{D^2(q)} & \nabla D(q) \quad D(q) \leq Q^* \\ 0 & D(q) > Q^* \end{cases} \tag{4.11}$$

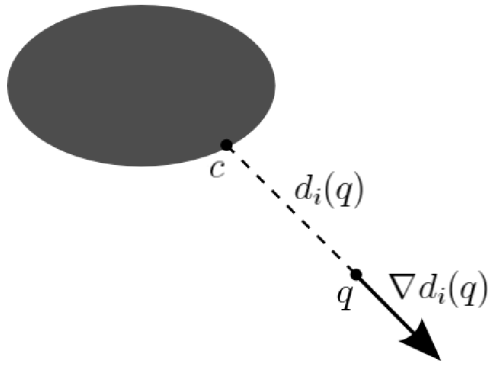
Q^* je určitý zvolený práh, který dovoluje ignorovat překážky, které jsou velmi vzdálené. Parametr η představuje váhu potenciály, podobně jako parametr ζ u přitažlivých potenciál. Tak jak je uvedeno v [2], může se při implementaci této metody stát, že trasa začne oscilovat mezi překážkami, protože daný bod leží ve stejné vzdálenosti od více překážek. Proto je lepší nepočítat tuto potenciálovou funkci jen pro nejbližší překážku, ale pro všechny překážky v dosahu a výslednou potenciálovou funkci definovat jako součet těchto dílčích funkcí. Můžeme ji pak definovat jako:

$$U_{rep}(q) = \sum_{i}^{n} U_{rep_i}(q) \tag{4.12}$$

a gradient:

$$\nabla d_i(q) = \frac{q - c}{d(q, c)}, \tag{4.13}$$

kde i představuje index právě jedné překážky z celkového počtu n, které jsou v dosahu a c představuje nejbližší bod na překážce k bodu q.

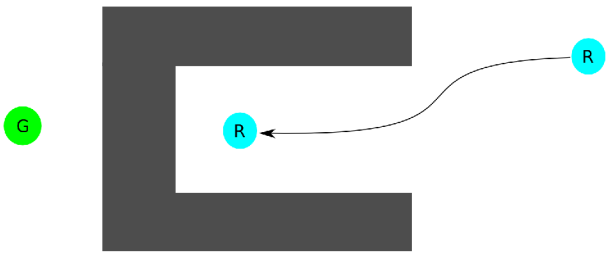


Obrázek 4.6: Obrázek znázorňuje vzdálenost $d_i(q)$ od překážky a gradient odpudivé potenciály z bodu q.

4.2.3 Problém lokálního minima

Při pohybu robota v potenciálových polích může často dojít k situaci, že robot uvázne v bodě, který je lokální minimum a není zaručeno, že právě v tomto minimu leží cíl. Jak můžeme vidět na obrázku 4.7, robot je přitahován cílem a postupně se přibližuje k překážce, která ho začne ovlivňovat. Horní část této překážky ho začne tlačit dolů a spodní ho začne tlačit nahoru, robot se tak pohybuje přesně mezi těmito částmi překážky směrem k cíli. Poté však začnou působit odpudivé síly té části překážky, která stojí mezi robotem a cílem. V určitém místě se síly působící směrem k cíli G a odpudivé síly od překážky vyrovnají a robot uvázne v bodě q, kde $\nabla U(q) = 0$, ale tento bod $q \neq G$ – robot není v cíli.

Podle [2] můžeme tento problém vyřešit použitím metody, která se jmenuje *Randomized Path Planner*. Tato metoda funguje tak, že robot postupuje směrem k cíli za použití různých potenciálových funkcí a pokud uvázne v lokálním minimu, tak se snaží z tohoto minima dostat sérií náhodných kroků. Většinou se pomocí těchto kroků podaří dostat z lokálního minima, a robot tak může znovu pokračovat klasickým způsobem k cíli.



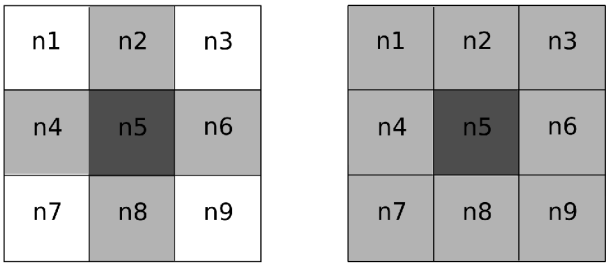
Obrázek 4.7: Robot uváznul v lokálním minimu.

4.3 Plánování na mřížce

Při použití potenciálových polí se často používá abstrakce okolního prostoru na mřížku. Celý prostor je pak diskretizován na jednotlivé body – pixely, které dohromady tvoří mřížku. Mřížku si tak můžeme představit jako dvourozměrné pole, kdy jednotlivé buňky (prvky) této mřížky jsou právě tvořeny pixely. Těmto buňkám pak můžeme přiřazovat hodnotu a využívat je tak při různých algoritmech.

Při použití různých metod a algoritmů si často můžeme zvolit, jakým způsobem spolu jednotlivé buňky sousedí. Každá buňka může mít buď čtyři sousední nebo osm sousedních buněk, tak jak je to zobrazeno na obrázku 4.8.

Pokud se za sousední považují jen čtyři buňky, má to tu výhodu, že můžeme použít *Manhattanskou vzdálenost*. Je to vzdálenost mezi dvěma body, měřená podél os vedoucích z těchto bodů, za předpokladu, že tyto osy spolu svírají pravý úhel. Takže například vzdálenost bodu $a1$ ležícího na souřadnicích $(x1, y1)$ od bodu $a2$ ležícího na $(x2, y2)$ se vypočítá jako $|x1 - x2| + |y1 - y2|$ [1].



Obrázek 4.8: Obrázek vlevo zobrazuje situaci, kdy za sousedící buňky se považují jen ty, které spolu sousedí celou hranou. Naproti tomu obrázek vpravo zobrazuje situaci, kdy se za sousedící považují ty buňky, které spolu mají společný alespoň jeden vrchol.

4.3.1 Výpočet vzdálenosti

Pro samotné hledání cesty je potřeba, aby robot uměl spočítat přitažlivé a odpudivé potenciály, potažmo aby uměl počítat vzdálenosti potřebných bodů. Při přitažlivých potenciálách by výpočet neměl být problém, protože je potřeba vypočítat vzdálenost mezi cílem a pozicí robota – obě tyto hodnoty známe.

Problém nastává u odpudivých potenciál, protože robot potřebuje vypočítat vzdálenosti k překážkám. Pokud by se robot pohyboval v prostředí a byl vybaven senzorem podobně jako u *Tangent Bug* algoritmu, mohl by pomocí tohoto senzoru zjistit lokální minima k překážkám ve svém dosahu. Zde je ale výpočet limitován dosahem tohoto senzoru, proto tato metoda nedává moc dobré výsledky. Jak jsme si již uvedli, u potenciálových polí se často využívá plánování na mřížce, kde můžeme využít *Brushfire algoritmus*, který dává dobré výsledky.

Brushfire algoritmus

Anglický název tohoto algoritmu můžeme volně přeložit jako Algoritmus šíření ohně. Tento algoritmus používá mřížku k určení vzdálenosti od překážek, a tím k určení odpudivé potenciálové funkce. Můžeme ho definovat tak, že na začátku všem buňkám mřížky, ve kterých leží alespoň část překážky, přiřadíme hodnotu jedna a ostatním buňkám přiřadíme hodnotu nula. Výstupem to-

hoto algoritmu bude mřížka, ve které hodnota každé buňky bude značit její vzdálenost od nejbližší překážky. Z těchto hodnot pak můžeme vypočítat odpudivou potenciálovou funkci a gradient.

Poté co přiřadíme počáteční hodnotu buňkám, ve kterých leží překážky, začneme určovat hodnotu dalších buněk. Všem buňkám, které mají hodnotu nula a sousedí s buňkami s hodnotou jedna, přiřadíme hodnotu dvě. Poté všem buňkám, které mají hodnotu nula a sousedí s buňkami s hodnotou dvě, přiřadíme hodnotu tři. Tento princip opakujeme, dokud nejsou ohodnoceny všechny buňky – žádná buňka nemá hodnotu nula. Při určování sousedních buněk si můžeme zvolit jestli budeme za sousední považovat čtyři nebo osm buněk, viz obrázek 4.8. Po dokončení tohoto algoritmu bude hodnota v každé buňce odpovídat vzdálenosti k nejbližší překážce. K tomu abychom mohli vypočítat odpudivou potenciálovou funkci, potřebujeme ještě znát gradient. Ten určíme tak, že u každé buňky vybere takovou sousední buňku, která má nejnižší hodnotu (pokud je takových buněk víc, vybereme libovolnou). Pak můžeme říci, že gradient je vektor, který míří na právě zvolenou sousední buňku.

Se znalostí gradientu a vzdáleností od překážky už není problém vypočítat odpudivou potenciálovou funkci. Nyní můžeme ve spojení s výpočtem přitažlivé potenciálové funkce (obvykle vzdálenost k cíli) určit potenciálové pole.

Tuto metodu nemusíme nutně omezovat jen na dvourozměrné pole, algoritmus bude fungovat i při vyšších dimenzích. Jen musíme náležitě upravit sousedství buněk, kdy je zřejmé, že při vyšších dimenzích už nebude buňka představovat čtverec, ale bude to například krychle (tři rozměry). Počet sousedních buňek se tak navýší ze čtyř na šest respektive z osmi na dvacet šet [2].

4.3.2 Hledání cesty

V předchozích částech jsme si uvedli všechny důležité poznatky, které jsou nutné k tomu abychom porozuměli potenciálovým polím. Známe všechny podstatné zákonitosti výpočtů potenciálových funkcí, problém spojený s lokálním minimem atd., proto teď můžeme přejít k samotnému problému hledání cesty.

Pokud si shrneme požadavky na algoritmus použitý pro hledání cesty, měly by to být tyto:

- pokud existuje cesta k cíli, měl by ji najít
- musí se umět vyhnout překážkám
- musí se umět vypořádat s nejednoznačnostmi, tzn. pokud by podle daného algoritmu měl cíl ležet v minimu, musí toto potenciálové pole obsahovat pouze jedno minimum.

Dále je určitě vhodné, ale ne nutné, aby daný algoritmus nebyl výpočetně náročný a aby vyhledal pokud možno co nejkratší cestu k cíli. Tyto dva nepovinné požadavky se však často navzájem vylučují, proto se většinou musíme spokojit pouze s jedním z nich.

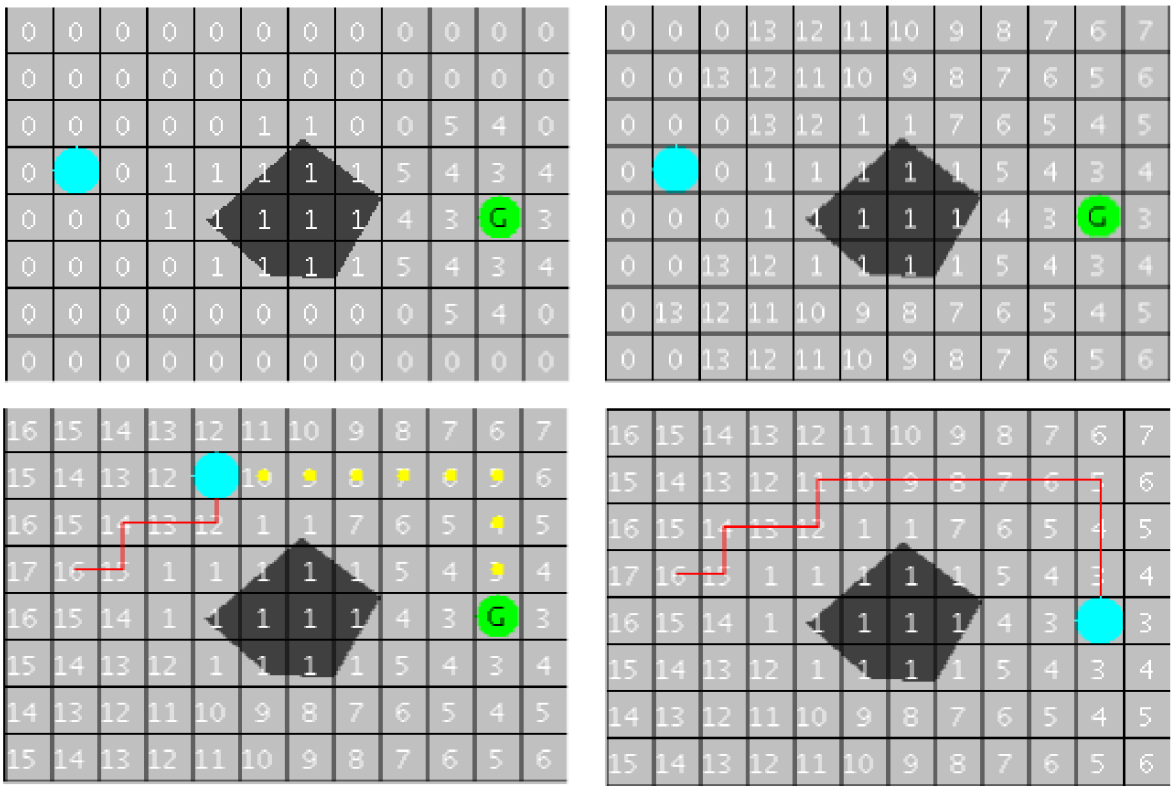
Algoritmus záplavového vyplňování

Tento algoritmus (angl. Wave-Front algorithm) splňuje výše zmíněné požadavky, protože dokáže jednoduše vyřešit nejtěžší požadavek – a to problém lokálního minima. Za určitou nevýhodu lze považovat to, že tento algoritmus můžeme použít pouze, pokud je prostor reprezentován mřížkou. Tento prostor ale může být i vícedimenzionální.

Algoritmus záplavového vyplňování je určitou modifikací *Brushfire algoritmu*, kdy buňkám obsahujícím překážku přiřadíme hodnotu jedna a neobsazeným buňkám hodnotu nula. Dále buňce, která odpovídá cíli, dosadíme hodnotu dva [2].

V prvním kroku dosadíme hodnotu tři buňkám, které mají hodnotu nula a sousedí s cílem. V dalším kroku dosadíme buňkám s hodnotou nula, které sousedí s buňkami o hodnotě tři, hodnotu

čtyři. Takto analogicky pokračujeme, dokud nedosáhneme buňky, na které je startovní pozice robota. Poté určíme ze startovní pozice cestu k cíli pomocí metody *sestupného gradientu*, to znamená že vždy hledáme sousedící buňku s hodnotou o jedna nižší. Takže např. pokud start leží na buňce s hodnotou 56, tak další buňka na cestě k cíli bude s hodnotou 55, další bude 54 atd. Jestliže je buněk s touto hodnotou více, libovolně z nich vybereme jednu. Tím, že hledáme vždy hodnotu o jedna nižší je také zaručeno, že nás tento algoritmus nepovede do překážky, protože buňky na kterých leží překážky mají hodnotu jedna, a hodnotu o jedna větší, tzn. hodnotu dva má pouze buňka kde leží cíl. Dále také můžeme říct, že už z principu tohoto algoritmu je jasné, že se nám vždy podaří najít buňku s nižší hodnotou, samozřejmě do té doby než se robot dostane do cíle. Pokud budeme



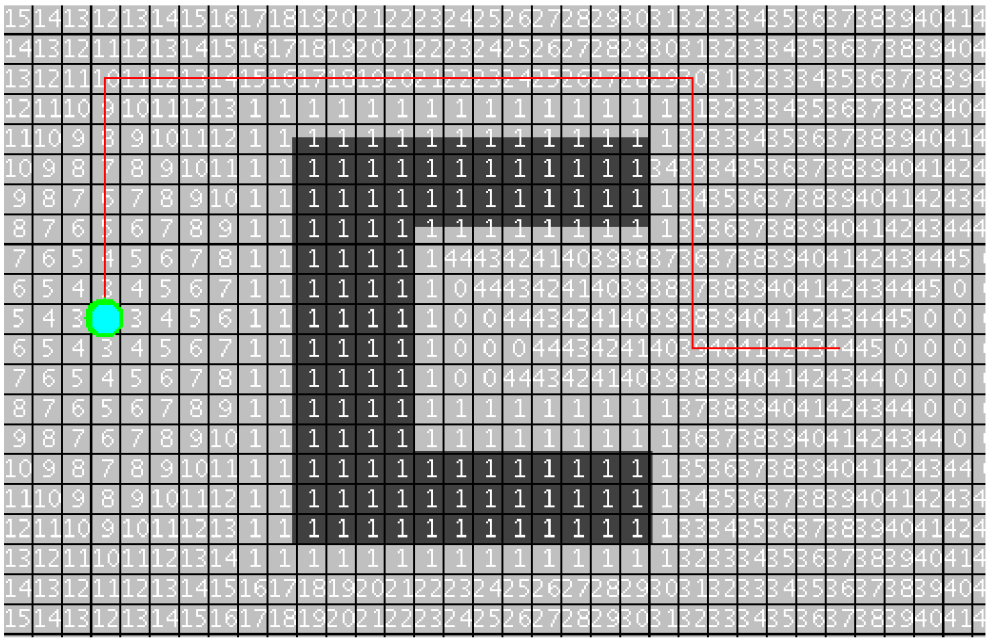
Obrázek 4.9: Na jednotlivých obrázcích můžeme vidět postup Algoritmu záplavového vyplňování.

za sousední buňky považovat jen ty, které mají společné hrany, tak můžeme říct, že všechny buňky se stejnou hodnotou mají stejnou (*Mannhattanskou*) vzdálenost k cíli.

Tato metoda je velmi výhodná, protože tvoří pouze jedno minimum a to v cíli, dále také dokáže najít nejkratší cestu k cíli. Toto má ale za následek to, že při prohledávání velkého prostoru je hodně náročná na paměťový prostor, protože si musí pamatovat velké množství hodnot. Při prohledávání vícedimenzionálních prostorů tato náročnost ještě více naroste.

Harmonická potenciálová pole

Dalším zajímavým řešením hledání cesty je metoda harmonických potenciálových polí, tato metoda má pouze jedno minimum ležící v cílové pozici. Podle [13] můžeme tuto metodu přirovnat k šíření tepla v prostředí, kdy zdroj šíření tepla je cíl a naopak překážky své okolí mrazí. Takže si potenciálové pole můžeme definovat tak, že buňka, ve které leží cíl má nejvyšší hodnotu (tj. jedna) a překážky



Obrázek 4.10: Na obrázku můžeme vidět důkaz toho, že si Algoritmus záplavového vyplňování dokáže poradit s lokálním minimem, které neleží v cíli.

mají nejnižší (nula), ostatní buňky mají hodnotu definovanou tak, že stejné množství tepla přijmou i odevzdají. Pokud máme takto nadefinované všechny buňky, robot může najít cestu k cíli tak, že postupuje ze své pozice směrem k cíli takovým směrem, kde je přírůstek hodnoty nejvyšší.

Podle [13] můžeme rovnovážný stav příjmu a výdaje tepla pro každou buňku vypočítat pomocí následující rovnice:

$$(U(q[x-1,y]) - U(q[x,y])) + (U(q[x+1,y]) - U(q[x,y])) +$$

$$+ (U(q[x,y-1]) - U(q[x,y]))(U(q[x,y+1]) - U(q[x,y])) = 0, \tag{4.14}$$

kde $U(q[x,y])$ je hodnota buňky q na souřadnicích x a y. Protože musíme tuto rovnici vypočítat pro každou buňku, je výhodné si sestavit lineární soustavu rovnic a pro výpočet použít iterační numerickou metody. Můžeme využít například Gauss-Seidelovu metodu. Sestavíme si rovnici, kterou budeme postupně iterovat, můžeme ji definovat takto:

$$U(q[x,y]) = \frac{1}{4}(U(q[x-1,y]) + U(q[x+1,y]) + U(q[x,y-1])U(q[x,y+1])), \tag{4.15}$$

což znamená, že hodnota každé buňky se rovná průměru součtu hodnot okolních hodnot.

Tato metoda nabízí spolehlivé řešení problému hledání cesty, tato vlastnost je ale vykoupena poměrně velkou výpočetní náročností.

Kapitola 5

Popis implementace appletů

Součástí této práce je i zpracování zde uvedených vyhledávacích metod pomocí java appletů. Tyto applety by měly pomoci k lepšímu pochopení tématiky a dokázání funkčnosti těchto vyhledávacích metod.

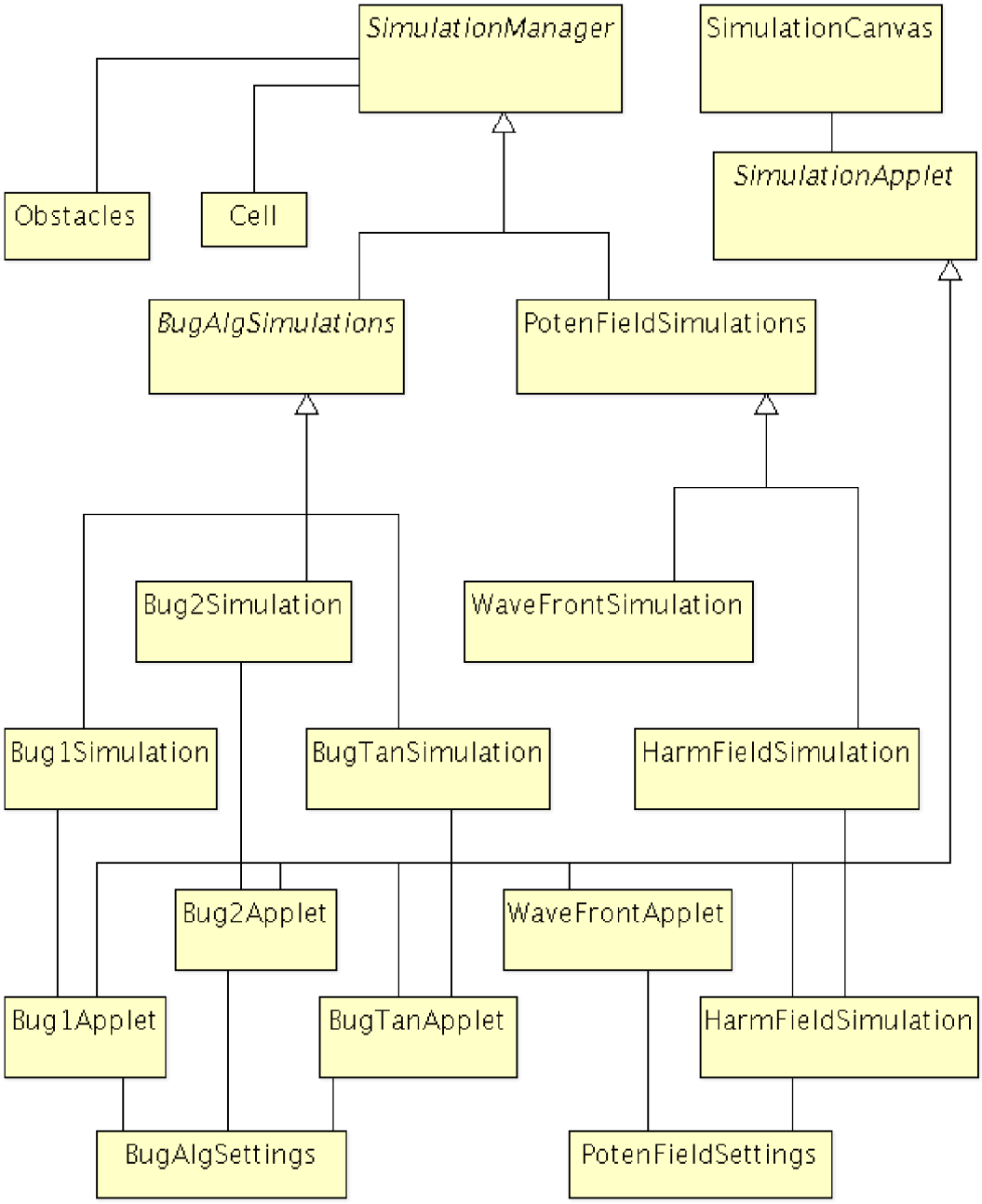
Na začátku této kapitoly si představíme implementační detaily appletů, popíšeme si základní třídy a samotné řízení a zobrazování simulace. Dále si popíšeme detaily implementace jednotlivých metod, jak příslušných *Bug algoritmů* tak i *Potenciálových polí*. Poté si popíšeme uživatelské rozhraní a ovládání appletů.

5.1 Návrh a implementační detaily

Při navrhování appletů jsem se snažil o co nejlepší návrh z hlediska možného budoucího rozšíření a zapracování dalších algoritmů, tzn. že jsem se snažil využít výhod objektově orientovaného programování jako je například dědičnost. Zároveň jsem se snažil tento návrh neudělat moc složitý, aby následná implementace nebyla obtížná. Applety jsou napsány v jazyku *Java* za použití standardních knihoven ve verzi 1.6.0.13. Při implementaci jsem se bohužel setkal s některými omezeními standartních knihoven, hlavně co se týče práce s grafickými komponentami. Některé metody nedávaly moc přesné výsledky, např. metody pro zjištění, jestli grafický objekt obsahuje bod o zadaných souřadnicích apod. Některé tyto problémy si ještě popíšeme později.

5.1.1 Hierarchie tříd

Třídy použité při vytváření těchto appletů patří do balíčku `pathFindingApplets`, hierarchii těchto tříd můžeme vidět na následujícím 5.1. Nyní si popíšeme jednotlivé skupiny těchto tříd.



Obrázek 5.1: Hierarchie tříd

Simulační třídy

Základní simulační třídou je abstraktní třída `SimulationManager`, která obsahuje parametry a metody společné pro všechny odvozené třídy. Uchovává pozice robota, cíle, překážek. Dále jsou v této třídě definované různé fáze simulace, tyto fáze jsou definované jako výčtový typ `RobotState`, může to být například fáze `FOLLOWING_OBSTACLE` nebo `MOVE_TO_GOAL`, podle těchto fází je pak řízena simulace. Tato třída je odvozená od třídy `java.lang.Thread`, která umožňuje spouštět instance této třídy jako samostatné vlákno. Tuto třídu rozšiřují dvě

třídy, které se už více specializují na použité metody, jsou to: `BugAlgSimulations` a třída `PotenFieldSimulations`.

Třída `BugAlgSimulations` obsahuje metody, které využívají *Bug algoritmy*, je to například metoda `exploreRobotRange()`, sloužící algoritmu *Tangent Bug* pro prozkoumání oblasti robota a nalezení překážek, nebo metoda `followObstacleOneStep()`, která provede jeden krok při objíždění překážky robotem u algoritmů *Bug1* a *Bug2*. Samotný algoritmus jednotlivých bug algoritmů je implementován v třídách odvozených od této společné třídy, jsou to třídy `Bug1Simulation`, `Bug2Simulation` a `BugTanSimulation`.

Třída `PotenFieldSimulations` naproti tomu obsahuje metody použité u potenciálových polí, jako je například metoda `initObstaclesCellValues()`, která nastaví hodnoty buněk, na kterých leží překážky. Algoritmus záplavového vyplňování je pak implementován v třídě `WaveFrotSimulation`, která je potomkem této třídy.

Třída pro grafické zobrazení

Pro zobrazení simulace, vykreslení překážek, robota apod. jsem rozšířil standartní třídu jazyka Java `java.awt.Canvas`. Takto vzniklá třída `SimulationCanvas` umožňuje pomocí metod `paintRobot()`, `paintObstacles()` a dalších, vykreslovat překážky, robota či průběh simulace, která probíha v simulační třídě. Tato třída implementuje rozhraní `MouseInputListener` z balíčku `javax.swing.event`, proto je možné zachytávat kliknutí uživatele na tento canvas (plátno). Tuto vlastnost využijeme pokud uživatel mění startovní pozici robota, pozici cíle nebo přidává či maže vybrané překážky. Třída `SimulationCanvas` v sobě neuchovává žádné informace, slouží pouze k nezávislému vykreslování nebo k předávání vstupu od uživatele simulační třídě.

Třídy pro definování appletů

Vzhled a rozvržení prvků na appletu je implementováno v abstraktní třídě `SimulationApplet`, která je potomkem třídy `javax.swing.JApplet`. Je zde definované a implementované obslužné menu včetně jednotlivých položek, dále má tato třída metodu `updateStatusInfo()`, která slouží k aktualizaci informačního statusu, který uživatele informuje o simulaci. Pod menu je zde přidán `SimulationCanvas`, který slouží pro samotné zobrazovaní simulace. V této třídě jsou také definovány abstraktní metody `initSimulation()`, `resetSimulation()` a `showSettings()`, které musí být implementované v potomcích této třídy, protože jsou závislé na typu právě probíhající simulace. První dvě metody se starají o korektní inicializaci a reset simulace a třetí metoda `showSettings()` by měla obsahovat kód, který zobrazí uživateli možné nastavení pro zvolenou simulaci.

Potomky této třídy jsou pak applety jednotlivých simulovaných algoritmů, v těchto třídách jsou jen korektně překryty požadované metody a nastavena základní simulační třída. Patří sem například třídy `Bug1Applet`, `WaveFrontApplet` a analogicky applety dalších implementovaných algoritmů.

Pomocné třídy

Zde si uvedeme další třídy využité při simulaci, první je třída `Obstacles`, která rozšiřuje `java.util.ArrayList<java.awt.Polygon>`, což je vlastně seznam překážek nebo-li objektů typu `Polygon`, protože každá překážka je reprezentovaná jedním grafickým objektem `Polygon`. Pomocí metod této třídy, jako je například metoda `defaultObstacles()` nebo `spiralObstacles()`, můžeme snadno nastavit rozmístění překážek, startu a cíle do přednasta-

vených scénářů, což je dostupné uživateli přes menu. Poslední metodou, kterou zde zmíníme, je metoda `addArea()`, která slouží k přidávání překážek, které nakreslí uživatel.

Další třídou je `Cell`, která je potomkem třídy `java.awt.geom.Rectangle2D.Float`. Tato třída reprezentuje jednu buňku v mřížce, pokud je použit algoritmus, který využívá mřížku. Uchovává v sobě svou pozici a také přiřazenou hodnotu.

Poslední dvě třídy `BugAlgSettings` a `PotenFieldSettings` rozšiřují třídu `JDialog` z balíčku `javax.swing`. Obsahují metodu `createGUI()`, která zobrazí uživateli okno s možnostmi nastavení dané simulace. Uživatel zde může měnit například velikost robota, dosah senzoru robota u algoritmu *Tangent Bug* apod. Třída `BugAlgSettings` slouží pro zobrazení nastavení pro simulace *Bug algoritmů* a třída `PotenFieldSettings` slouží pro zobrazení nastavení pro simulace *Potenciálových polí*.

5.1.2 Řízení a zobrazení simulace

Už jsme si představili jednotlivé třídy, proto si teď můžeme popsat samotný princip fungovaní appletů, řízení a zobrazení simulace.

V první fázi dochází k inicializaci appletu, spustí se metody patřící třídě `SimulationApplet` a to metoda `createGUI()` a `initSimulation()`. První se postará o vykreslení a zobrazení prvků na appletu a druhá inicializuje příslušnou simulaci, například při simulaci algoritmu *Bug1*: spustí se instance třídy `Bug1Applet`, která spustí inicializační metody: vykreslí menu, canvas a vytvoří v novém vlákně instanci simulační třídy `Bug1Simulation`. Tato simulační třída po inicializaci čeká až tuto simulaci spustí uživatel přes menu, v tomto okamžiku se mohou měnit parametry simulace, jako je například rozmístění překážek, pozice startu a cíle apod. Po spuštění simulace probíhají jednotlivé fáze daného algoritmu a je postupně překreslován canvas. Protože simulace běží ve svém vlastním vláknu, není překreslování appletu nějak ovlivňováno danou simulací. Jakmile je dokončena simulace algoritmu, dochází k ukončení běhu simulačního vlákna. Uživatel může opět spustit novou simulaci přes menu, to vyvolá metodu `resetSimulation()`, kdy se vytvoří opět nové vlákno se simulačním algoritmem.

Při výpočtech simulace je robot považován pouze za bod v prostoru, protože jsem však při zobrazení simulace chtěl, aby byl robot dobře viditelný, reprezentuje ho kolečko o větším průměru. Proto bylo potřeba vyřešit, aby robot při míjení překážek do nich nezasahoval, ale aby se jim korektně vyhnul i se svou větší velikostí. Proto jsem se rozhodl zvětšit hranice překážky o poloměr robota, ale zobrazuji původní nezvětšené překážky. V knihovnách Javy jsem bohužel nenašel žádnou funkci, která by korektně zvětšovala všechny typy překážek, proto jsem ve třídě `SimulationManager` implementoval metodu `enlargeObstaclesBorders(int size)`, která překážky zvětší o zadanou velikost pomocí upraveného algoritmu *dilatace obrazu*.

5.1.3 Implementace zkoumaných algoritmů

V této části si popíšeme detaily implementace jednotlivých algoritmů a představíme si zejména nejdůležitější metody. Pokud nebude uvedeno jinak, metody použité u *Bug algoritmů* jsou definované v třídě `BugAlgSimulations` a metody použité u *Potenciálových polí* jsou definované v třídě `PotenFieldSimulations`.

Algoritmus Bug1

Při tomto algoritmu používá robot v podstatě dva druhy chování: pohyb směrem k cíli, který je implementován v metodě `moveRobotToGoalOneStep()` a pohyb okolo překážky implementovaný v metodě `followObstacleOneStep()`. Po tom co robot objede celou překážku je

potřeba, aby se co nejkratší cestou přemístil do bodu opuštění překážky, dosáhne toho za pomoci metody `moveRobotToLeavePtOneStep()`.

Algoritmus Bug2

Tento algoritmus využívá pro pohyb okolo překážky a pohyb k cíli stejné metody jako předchozí algoritmus. Za zmínku stojí metoda `isFoundLeavePoint()`, která kontroluje jestli robot při objíždění překážky nedosáhl bodu opuštění.

Algoritmus Tangent Bug

Pro pohyb robota k cíli a okolo překážky využijeme metody `moveRobotToGoalOneStepTngt()` a `followObstacleOneStepTngt()`. Tento algoritmus je výpočetně náročnější, protože robot musí v každé pozici zkontrolovat prostor, kam dosahuje jeho senzor a podle toho přepočítávat svou cestu, tato kontrola se děje za pomoci metody `exploreRobotRange()`. Tato metoda prozkoumá okolní prostor a vybere nejlepší směr pohybu robota.

Algoritmus záplavového vyplňování

Tento algoritmus využívá plánování na mřížce, mřížku inicializujeme pomocí metody `initGrid()` a nastavíme hodnoty mřížky pomocí metody `initObstaclesCellValues(int value)`, tj. buňky ve volném prostoru na hodnotu nula a buňky na překážkách na hodnotu specifikovanou parametrem `value`, v tomto případě na jedna. Zde se objevily další problémy se standardními knihovnami Javy, metoda `intersects()` sloužící ke zjištění, zda-li daný grafický objekt obsahuje čtverec definovaný parametrem, nevrací vždy přesné výsledky. Což se v tomto případě projevilo tak, že při určování zda-li hledaná buňka zasahuje do překážky, označila tato metoda i buňky, které už leží vně překážky. Což vedlo k tomu, že překážky byly v mřížce označeny na větším prostoru, než jaký doopravdy zabíraly. Na samotný algoritmus však tento nedostatek vliv neměl. Algoritmus záplavového vyplňování je implementován v metodě `computeWaveFrontOneStep()`, poté se vyhledá cesta k cíli metodou `computeLineToGoalOneStep()` a robot se do cíle přesune.

Harmonická potenciálová pole

Podobně jako u předchozího algoritmu inicializujeme mřížku, nastavíme hodnototy buněk na překážkách, tentokrát však nastavíme parametr `value` na nula. Výpočet hodnot dalších buněk a tím i výpočet potenciálového pole provádíme pomocí metody `computeGaussSeidelForCell()`, což je vlastně implementace Gauss-Seidelovy metody pro náš případ. Po spočítání celého pole, vyhledáme pomocí metody `computeLineToGoalOneStepHarm()` cestu k cíli a robot se tam přesune. Tato metoda je výpočetně velmi náročná, proto je maximální počet buněk v mřížce omezen.

5.2 Popis uživatelského rozhraní appletů

Nyní si popíšeme uživatelské rozhraní appletů a také jejich ovládání. Applet se skládá z menu, z oblasti pro informování uživatele o stavu simulace a z plochy kde je vykreslována simulace, tak jak je můžeme vidět na obrázku 5.2. Hlavní menu se skládá z pěti vysouvacích nabídek:

- **Simulation** - obsahuje tlačítka pro start, reset a nastavení simulace
  - **Start simulation** - tlačítko pro start simulace

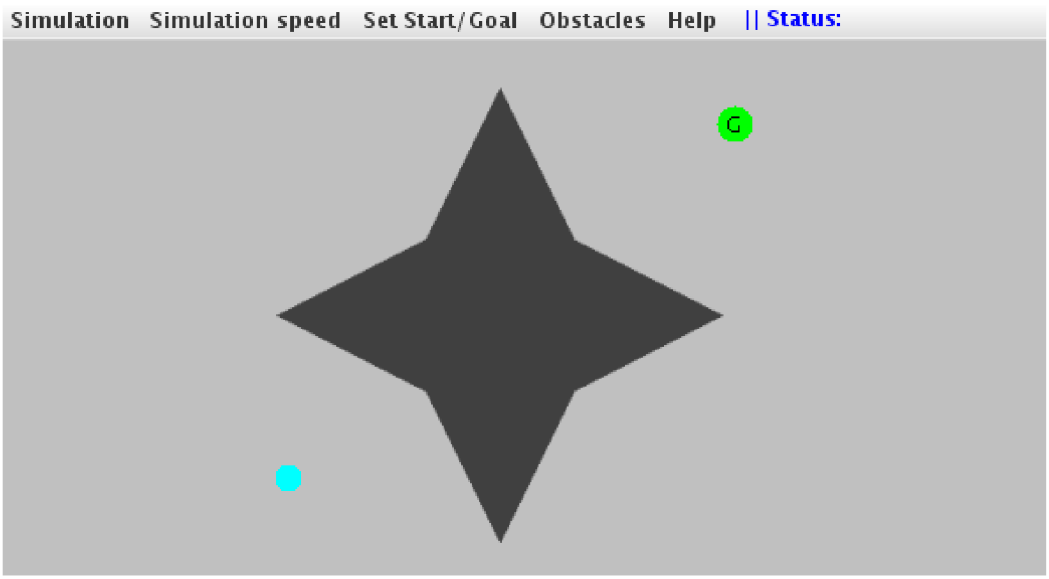
  - **Reset simulation** - tlačítko pro reset simulace
  - **Simulation settings** - tlačítko pro otevření okna s nastavením simulace

- **Simulation speed** - tato nabídka slouží pro nastavení rychlosti
  - **Instant simulation** - toto tlačítko slouží pro spuštění simulace bez prodlevy
  - nabídka obsahuje také posuvník pro pohodlné nastavení rychlosti simulace

- **Set Start/Goal** - nabídka pro přesunutí startu a cíle, je aktivní pouze před začátkem simulace
  - **Set Start** - po kliknutí na tlačítko se kliknutím na plochu zvolí pozice startu
  - **Set Goal** - po kliknutí na tlačítko se podobně nastaví i pozice cíle

- **Obstacles** - nabídka umožňující uživateli přidávat vlastní překážky, odstraňovat překážky nebo zvolit jeden z přednastavených scénářů rozmístění překážek
  - **Add Obstacle** - kliknutím na plochu začíná uživatel kreslit hranu překážky, dalším kliknutím tuto hranu uloží a může kreslit další hranu, kreslení překážky ukončí dvojklikem
  - **Delete Obstacle** - kliknutím na zvolenou překážku ji odstraní
  - **Default scenario** - standardní scénář rozmístění překážek
  - **Simple scenario** - scénář obsahující jednoduché překážky
  - **Star scenario** - scénář obsahující nekonvexní překážku
  - **Spiral scenario** - scénář obsahující překážku ve tvaru spirály

- **Help** - nabídka obsahující nápovědu a informace o aplikaci
  - **Help** - tlačítko které vyvolá nápovědu
  - **About** - tlačítko které zobrazí informaci o aplikaci



Obrázek 5.2: Uživatelské rozhraní appletu.

Kapitola 6

Závěr

V této práci jsem se zabýval problematikou hledání cesty robota za použití dvou rozdílných přístupů. Bug algoritmy slouží pro hledání cesty ve zcela neznámém prostředí, naproti tomu potenciálová pole slouží pro hledání cesty ve známém prostředí. Pro demonstraci těchto algoritmů jsem vytvořil java applety.

V první části této práce jsem uvedl některé základní pojmy týkající se robotiky, v dalších kapitolách popisuji principy a podrobnosti Bug algoritmů a Potenciálových polí. Samostatná kapitola je věnována návrhu a implementaci appletů včetně popisu jejich ovládání z hlediska uživatele.

Problematika hledání cesty robota je zde diskutována především z hlediska použití těchto algoritmů v dvourozměrném prostoru. Užití těchto algoritmů ve vícerozměrném prostoru je zde nastíněno jen okrajově, protože je to velmi rozsáhlá oblast, která si zaslouží další zpracování v nějakém projektu navazujícím na tuto práci.

V rámci této práce jsem také vytvořil webovou prezentaci, kde jsou popsány jednotlivé metody a kde jsou umístěny applety. Adresa tohoto webu je `http://www.stud.fit.vutbr.cz/~xrouba02/bakalarka/`. Na webu stejně jako na přiloženém CD je umístěna i textová část práce, programová dokumentace a zdrojové soubory appletů.

Literatura

[1] BLACK, P. E.: *Manhattan distance* [online]. 2006, [rev. 2006-05-31], [cit. 2009-04-30]. URL http://www.itl.nist.gov/div897/sqg/dads/HTML/manhattanDistance.html

[2] CHOSET, H. M.; HUTCHINSON, S.; LYNCH, K. M.; aj.: *Principles of Robot Motion*. MIT Press, 2005, ISBN 0-262-03327-5.

[3] DLOUHÝ, M.: *Exaktní plánování* [online]. 2003, [rev. 2003-11-07], [cit. 2009-04-30]. URL http://robotika.cz/guide/exactplan/cs

[4] DLOUHÝ, M.: *Pravděpodobnostní plánování* [online]. 2003, [rev. 2003-12-05], [cit. 2009-04-30]. URL http://robotika.cz/guide/probplan/cs

[5] DLOUHÝ, M.; WINKLER, Z.: *Co je to robot?* [online]. 2005, [rev. 2005-01-22], [cit. 2009-04-30]. URL http://robotika.cz/guide/robot/en

[6] GOODRICH, M. A.: *Potential Fields Tutorial* [online]. [cit. 2009-05-10]. URL http://www.ee.byu.edu/ugrad/srprojects/robotsoccer/papers/goodrich_potential_fields.pdf

[7] KOLOMAZNÍK, J.: *Implementace A* algoritmu na konkrétní problém orientace v prostoru budov* [online]. 2005, [rev. 2005-11-28], [cit. 2009-04-30]. URL http://nlp.fi.muni.cz/uui/referaty2005/kolomaznik_jan/Implementace%20A%20Star%20-%20text.pdf

[8] Wikipedie: *Navigation function* [online]. 2008, [rev. 2008-03-20], [cit. 2009-04-30]. URL http://cs.wikipedia.org/wiki/Navigation_function

[9] Wikipedie: *Gradient descent* [online]. 2009, [rev. 2009-05-06], [cit. 2009-05-10]. URL http://en.wikipedia.org/wiki/Gradient_descent

[10] Wikipedie: *Gradient* [online]. 2009, [rev. 2009-03-24], [cit. 2009-04-30]. URL http://cs.wikipedia.org/wiki/Gradient

[11] Wikipedie: *Robot* [online]. 2009, [rev. 2009-03-11], [cit. 2009-04-30]. URL http://cs.wikipedia.org/wiki/Robot

[12] Wikipedie: *Voronoi diagram* [online]. 2009, [rev. 2009-04-22], [cit. 2009-04-30]. URL http://en.wikipedia.org/wiki/Voronoi_diagram

[13] WINKLER, Z.: *Plánování na mřížce* [online]. 2003, [rev. 2003-12-03], [cit. 2009-04-30]. URL http://robotika.cz/guide/gridplan/cs