VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

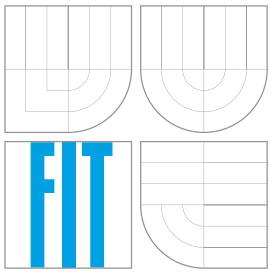
PLÁNOVÁNÍ CESTY V MAPĚ OPENSTREET

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAN DOBEŠ
AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PLÁNOVÁNÍ CESTY V MAPĚ OPENSTREET

PATH PLANNING IN OPENSTREET MAP

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN DOBEŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAROSLAV ROZMAN, Ph.D.

BRNO 2014

Abstrakt

Tato práce se zabývá možnostmi plánování cesty v OpenStreet mapě a Robotickém operačním systému (ROS framework), zkoumá existující řešení, poté vhodně implementuje knihovnu s vlastními plánovacími algoritmy a dalšími vylepšeními. Nakonec ověří funkčnost knihovny.

Abstract

This thesis investigates possible route planning methods in OpenStreet map and Robot operating system (ROS framework), investigates already existing solutions and then implements well integrated own library with own planning algorithms and other improvements. Finally, implemented features are verified.

Klíčová slova

ROS, OpenStreet, mapa, plánování cesty, vizualizace, navigace, rviz

Keywords

ROS, OpenStreet, map, path planning, visualisation, navigation, rviz

Citace

Jan Dobeš: Plánování cesty v mapě OpenStreet, bakalářská práce, Brno, FIT VUT v Brně, 2014

Plánování cesty v mapě OpenStreet

Prohlášení

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

.....
Jan Dobeš
20. května 2014

Poděkování

Rád bych poděkoval panu Ing. Jaroslavu Rozmanovi Ph.D. za vedení při tvorbě této práce.

Obsah

Kapitola 1

Úvod

S plánováním cesty se dnes setkáváme prakticky na každém rohu. Využití má v mnohých odvětvích lidské činnosti, od nejtriviálnější navigace v prostoru reálného světa, po hledání cest v abstraktních strukturách pro nějaký vědecký projekt. Algoritmy určené pro toto plánování nám umožňují získat informace, kudy vést trasu, na kterou můžeme mít různé požadavky, např. aby byla optimální, co nejkratší nebo co nejrychlejší. Tato práce se zabývá spoluprácí volně dostupné *OpenStreet* mapy reálného světa a frameworku *ROS*. Práce zkoumá již dostupná řešení pro plánování cest a použití *OpenStreet* mapy. Dále implementuje knihovnu spolupracující s existujícími komponenty, která nabízí nové nebo alternativní komponenty.

Ve druhé kapitole jsou popsány technologie a nástroje, které jsem použil pro řešení projektu, jejich spolupráce a příklady použití. Lze tu najít informace o *Robotickém operačním systému* a jeho knihovnách vhodných k plánování cest v mapě. Dále o *OpenStreet* mapě, způsobu, jak ukládá data a jak ji lze získat. Oboje spojuje stručný příklad, jak tyto dvě technologie mohou spolupracovat a jak výsledek můžeme vizualizovat.

Ve třetí kapitole se řeší již dostupná řešení, jak moc jsou vyhovující a zda by nebylo vhodnější je implementovat znovu. Poté se definuje, co všechno se bude implementovat, s důrazem na spolupráci s existujícími knihovnami v *ROS*. Také se zde částečně nastiňuje implementace, když se navrhuje struktura implementovaného balíku a jeho funkce.

Čtvrtá kapitola se zabývá detaily implementace balíku, plánovacích algoritmů a jednotlivých nodů. Následně probíhá ověření na sadě úloh, zda implementované struktury opravdu dělají to, co mají.

Kapitola 2

Použité technologie a nástroje

V této kapitole se zaměřuji na stručné zasvěcení do problematiky technologií a nástrojů použitých pro řešení práce. Dále popisuji způsob, jakým tyto technologie spojit. V zásadě jsou klíčové jen 2 technologie – *ROS* a *OpenStreet* mapa, ale především pak *ROS* zastřešuje širokou oblast komponent pro různé účely.

Implementace vlastních komponent bude probíhat v jazyce *C++*, který je dán zadáním. Stejného výsledku by se ale dalo dosáhnout i v jazyce *Python*, který patří také mezi podporované v *ROS*.

2.1 ROS - Robotický operační systém

Robotický operační systém (*ROS*) je softwarový framework primárně určen pro vývoj softwaru na ovládání robotů. Nejedná se tedy o operační systém v pravém slova smyslu, ale spíše o speciální prostředí s vlastními koncepty usnadňující tvorbu aplikací. Framework obsahuje podpůrné nástroje a knihovny, které se hodí pro různé druhy projektů. Mimo knihovny, které jsou zahrnuty ve standardní instalaci, existuje řada dalších knihoven třetích stran. Jedná se o open source projekt šířený pod licencí BSD. V současnosti je instalace frameworku primárně podporována na *unixových* operačních systémech, zejména pak na distribuci *Ubuntu*, která je jako jediná označena jako podporovaná, ostatní distribuce a operační systémy jsou označeny jako *experimentální*. Pro experimentální platformy je často nutné si celý framework ručně zkompilovat ze zdrojových kódů a ani výsledek nemusí být zcela funkční. Dále je pak instalace *ROS* oficiálně podporována na určitých modelech robotů.

2.1.1 Historie

[9] Vývoj začínal v roce 2007 pod názvem *switchyard* a byl vyvíjen *Stanford Artificial Intelligence Laboratory*, později téhož roku vývoj přechází pod laboratoř *Willow Garage*. V roce 2009 vychází první verze frameworku v dnešní podobě – *ROS 0.4*, o rok později verze *1.0*, dále stručný seznam hlavních verzí.

- 22. ledna 2010 - ROS 1.0
- 1. března 2010 - Box Turtle
- 3. srpna 2010 - C Turtle

- 2. března 2011 - Diamondback
- 30. srpna 2011 - Electric
- 23. dubna 2012 - Fuerte
- 31. prosince 2012 - Groovy
- 4. září 2013 - Hydro
- zatím nevydáno - Indigo

Jednotlivé verze nejsou mezi sebou plně kompatibilní. Jedná se především o experimentální software, který sdružuje řadu menších komponent, které se zejména v rané části vývoje často mění.

Tato práce vychází z nejnovější verze k podzimu 2013 – *Hydro*.

2.1.2 Proč použít ROS?

Použití frameworku *ROS* je součástí zadání této práce, nicméně existují další důvody, proč je výhodné jej využít.

Důvodem proč vytvořit aplikaci pro plánování cesty právě nad tímto frameworkem může být jeho celková provázanost s roboty a jejich ovládáním. Kód aplikace lze poté snadno modifikovat např. aby cestu vizualizoval na počítači nebo pomocí dalších knihoven přímo pracoval na robotovi. Zde by mohl cestu v mapě převádět na instrukce pro navádění daného robota, například kde má zrychlit, zpomalit, zatočit a podobně.

Projekt *ROS* má dobrou podporu komunity a má k dispozici i mnoho různých knihoven třetích stran, které jsou často v podobě samostatně spustitelných modulů a nabízí k dispozici nějaké služby nebo publikují do nějakého topicu.

2.1.3 Základní princip

Jak již bylo řečeno, *ROS* není operační systém ve smyslu, že by se zaváděl do operační paměti hned po startu daného zařízení, ale běží až na jiném běžícím univerzálním operačním systému. ROS je od tradičního operačního systému oddělené prostředí, ale přináší vlastní koncepty pro abstrakci hardwaru, procesů a komunikace mezi nimi nebo správů komponent. Procesy tu jsou modelovány ve formě tzv. **nodů** – samostatné jednotky provádějící požadovaný kód a komunikující s ostatními nody. Tyto nody se umisťují do, zpravidla podle využití pojmenovaných **balíků**, které se dále mohou sdružovat do tzv. **stacků**.

Například stack `navigation` obsahuje mimo jiné balíky `move_base` a `navfn`, každý obsahuje 1 node pojmenovaný stejně jako balík.

Základní součástí běžícího systému je `roscore`, což je kolekce nodů, programů a dalších prerekvizit, které jsou nutné k běhu dalších nodů v tomto prostředí.[7] `roscore` si lze představit jako druh serveru, přes který ostatní nody mohou komunikovat s ostatními.

Více informací o činnosti jednotlivých komponent na příkladu v sekci 2.4.

2.1.4 Zprávy

Zprávy se používají při komunikaci mezi nody. Důležitý je jejich typ. Ten je vždy na nejnižší úrovni složen z primitivních typů (bool, int, float…), ale pro praktické použití se typy skládají do polí a složitějších pojmenovaných struktur. Podle konvence se typy zpráv definují v podadresáři balíku `msg/`, každá zpráva je pak v samostatném souboru `*.msg`.

```
# Way-point element for a geographic map.

uuid_msgs/UniqueID id   # Unique way-point identifier
GeoPoint   position     # Position relative to WGS 84 ellipsoid
KeyValue[] props        # Key/value properties for this point
```

Příklad zprávy typu `geographic_msgs/WayPoint`, která reprezentuje bod v OSM, sdružuje jeho ID, souřadnice a atributy.

Zprávy se sdružují do samostatných balíků určených jen pro zprávy, například `geographic_msgs`, ale není to vyžadováno, jedná se spíše o dobrý programátorský zvyk pro snažší znovupoužití určitých zpráv v jiných balících.

2.1.5 Komunikace mezi nody

Ros využívá ke komunikaci mezi nody 2 koncepty.

- topic
- služba s parametry

Topic komunikace je v podstatě pojmenovaný komunikační kanál, který přenáší zprávy určitého typu. Z tohoto kanálu můžeme buď odebírat zprávy nebo je do něj zapisovat. Komunikace probíhá mezi N zapisovateli a N příjemci.

Služby si lze představit jako volání funkce v standardtním procedurálním programovacím jazyce. Jedna strana požaduje od druhé odpověď na otázku, a ta ji odešle odpověď. Služba má stanovené vstupní a výstupní zprávy. Dobře navržená služba by měla obsahovat oba typy zpráv, pokud neobsahuje vstupní nebo výstupní parametry, bývá výhodnější využít topic komunikace.

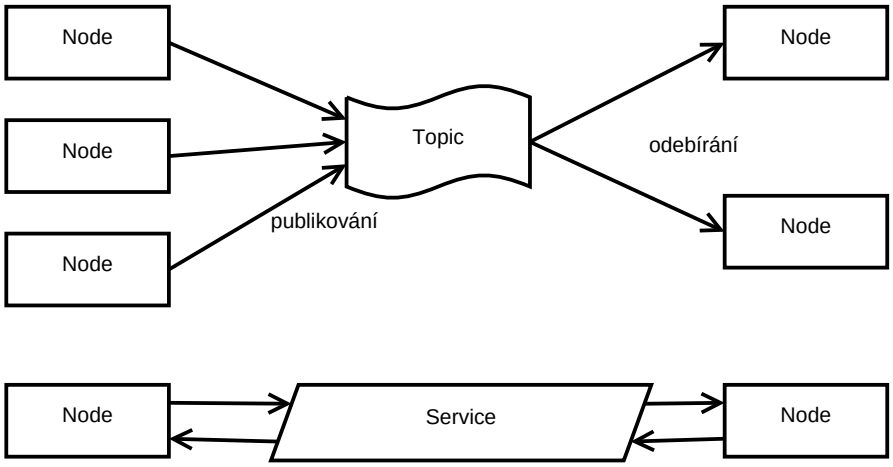
Služby se v balíku umisťují do adresáře `srv/` a jsou v souborech `*.srv`.

```
# Get a plan to traverse a route network from start to goal.
#
# Similar to nav_msgs/GetPlan, but constrained to use the route network.

uuid_msgs/UniqueID network      # route network to use
uuid_msgs/UniqueID start        # starting way point
uuid_msgs/UniqueID goal         # goal way point
---
bool            success         # true if the call succeeded
string          status          # more details
RoutePath       plan            # path to follow
```

Příklad definice služby `geographic_msgs/GetRoutePlan`. Nad oddělovačem jsou zprávy určitého typu jako parametry, dole jsou zprávy, které jsou vráceny.

Podobně jako zprávy se definice služeb mohou umisťovat do samostatných balíků, kde plní funkci rozhraní, jehož funkci jiný balík implementuje. Výsledná služba, která se volá je pak libovolně pojmenovaná, název závisí na nodu, který její rozhraní implementuje.



Obrázek 2.1: Znázornění rozdílu komunikace mezi topics a službami.

2.1.6 Správa balíků – nástroj catkin

Catkin je oficiální automatizovaný nástroj ROS pro překlad balíků, knihoven, generování hlavičkových souborů a dalších věcí usnadňujicích implementaci, překlad a distribuci balíků. Cílem je nahradit starší nástroj `rosbuild`. Využívá nástroj *CMake*.

Důvodem k použití takové nástroje je velká rozmanitost projektů implementovaných v ROS, které jsou implementovány v různých jazycích, mají závislosti na dalších projektech a další komplikace, díky kterým by manuální vývoj ROS modulu byl kontraproduktivní. S tímto nástrojem stačí doplnit požadované informace o projektu a jeho závislostech do konfiguračních souborů a catkin se při překladu postará o zbytek. Například ze zpráv a služeb definovaných v souborech, které jsou popsány výše, se vygenerují hlavičkové soubory nutné pro referenci na zprávu/službu při implementaci modulu.

Vývoj vlastních modulů probíhá v pracovním adresáři catkinu, který je strukturován do podadresářů se přeloženými binárními soubory, zdrojovými kódy a pomocnými nástroji. Nejdůležitější pro vývojáře je adresář se zdrojovými soubory – `src/` Každý modul má vlastní adresář s konfiguračními soubory pro popis balíku a samotnými zdrojovými texty.

2.1.7 Správa parametrů

ROS obsahuje vlastní službu určenou ke správě parametrů, aby se vývojáři vyhnuli zpracovávání parametrů z příkazové řádky. Na pozadí spuštěného ROS běží parametrový server. V jakémkoliv okamžiku lze číst, zapisovat, měnit nebo mazat pojmenované parametry. Parametrový server lze ovládat nástroji s příkazové řádky, z *API* programovacího jazyka nebo *launch* souborů[1].

2.2 Relevantní komponenty z ROS

Pro tento projekt jsou důležité především balíčky obsahující metody pro plánování cesty. Užitečné jsou i balíčky pomáhající pracovat s *OpenStreet* mapou, tudíž by nemuselo být potřeba vytvářet vlastní parser mapy pro tento účel.

[1] XML soubory definující seznam komponent, které se provedením tohoto souboru spustí

2.2.1 Navigation stack

Jedná se o soubor balíčků které umí zpracovat vše od zpracování informací ze senzorů, správu mapy, až po výstupní rychlostní příkazy pro robota. Navigace probíhá v 2D prostoru. Plánování cesty je tu prováděno ve dvou fázích – globálně a lokálně. Pro naše účely je důležité globální plánování, zde se vyhodnocuje trasa v prostoru s překážkami, lokální plánovač naopak řeší až problém, jakou trajektorií robot dosáhne cíle, když už zná cestu. Navigation stack je k dispozici pod BSD licencí.

Globální plánovače v tomto stacku nalezneme 2 – `carrot_planner` a `navfn`.

`carrot_planner` je jednoduchý plánovač, který se snaží co nejvíce přiblížit cíli pomocí obcházení překážek a není vhodný pro použití s mapou, kde uzly tvoří síť. `navfn` je potřebám práce bližší, vytváří si mapu prostředí, pro hledání nejkratší cesty v ní používá *Dijkstrův algoritmus* (v současnosti, v budoucnu možná podpora pro $A*$[6]).

2.2.2 osm_cartography

Balík obsahující nody pro práci s *OpenStreet* mapou. Vstupem je mapa v *XML* formátu, ta se převede do formátu zpráv popsaném v balíku `geographic_msgs`. Mapu je možné dále převést do formátu *RViz* souřadnic. 2.2.5 Tento balík není součástí základní instalace *ROS* a je k dispozici pod BSD licencí.

2.2.3 route_network

Balík spolupracující s `osm_cartography`. Stejnojmenný node `route_network` získá pomocí funkcí z balíku `osm_cartography` mapu ve formátu `geographic_msgs` a extrahuje z ní pouze relevantní informace – pouze sjízdné cesty. Tuto zjednodušenou mapu lze vizualizovat také pomocí *RViz* souřadnic.

Je tu i node `plan_route`, který nabízí službu pro plánování trasy. Využívá algoritmus $A*$ s heuristikou vyhodnocující přímou vzdálenost z bodu k cíli.[8]

2.2.4 geographic_msgs

Tento balík obsahuje struktury a funkce, jejichž použití je popsáno v předchozích dvou podsekcích. `osm_cartography` a `route_network` umožňují, aby se do struktur definovaných zde dala převést *OpenStreet* mapa.

2.2.5 RViz

Nástroj pro 3D vizualizaci. V tomto projektu se využije především za účelem rychlého ověření, jak dobře dané algoritmy fungují. Vzhledem k textové reprezentaci jednotlivých objektů v mapě by bylo obtížné jinak než graficky ověřovat řešení.

2.3 OpenStreet mapa

OpenStreet mapa je komunitní projekt, který vytváří a udržuje svobodná mapová data po celém světě. Mapu vytváří komunita různými způsoby – z leteckých snímků, pomocí GPS, digitalizováním papírových map atd. Mapová data jsou šířena pod *Open Data Commons Open Database License*[4]. Projekt *OpenStreetMap* vznikl v roce 2004 a v současnosti je spravován neziskovou organizací *OpenStreetMap Foundation*[3].

Veškerá mapová data jsou volně ke stažení, ale při reálném použití se kvůli velikosti dat mapy stáhne pouze určitý výřez. Stažená mapová data jsou uložena v *XML* formátu. Získat mapová data lze více způsoby. Jsou k dispozici připravené archivy s určitými regiony, pro účel ověření této práce je ale vhodnější stáhnout pouze určitý obdélníkový výřez pomocí *API*, které je ve formátu webové adresy s parametry souřadnic krajních bodů, tuto adresu lze sestavit a poté stáhnout i přímo z webového rozhraní mapy. Stahování přes *API* je omezené určitou velikostí výřezu, pro výřezy větších oblastí je nutné použít tzv. *XAPI*[5], kde je adresa odlišná.

`http://api.openstreetmap.org/api/0.6/map?bbox=11.54,48.14,11.543,48.145`

Příklad adresy pro stažení výřezu mapy pomocí *API*.

2.3.1 Formát mapy

Stažená mapová data jsou ve formátu *XML*[11] – jednotlivé objekty v ní jsou čitelné pro člověka i v této formě, ale jejich vztah a pozice v prostoru je těžko představitelná.

V mapě najdeme 4 druhy primitiv – *body*, *cesty*, *relace* a *atributy*.

Každý bod obsahuje svůj identifikátor, souřadnice pozice, dále další volitelné informace jako např. jméno jeho autora, verzi nebo čas, kdy byl změněn.

Cesty spojují jednotlivé body do skupin. V tomto objektu definujeme jednotlivé body a také další atributy, např. typ cesty, její jméno, její směr apod.

Relace stružují body, cesty a další relace do větších celků a nastavují další atributy, např. určení omezení skupiny cest pro určitá vozidla, určení, ze kterých cest je možné jezdit na které atd.

```
<?xml version=,,1.0‘‘ encoding=,,UTF-8‘‘?>
<osm version=,,0.6‘‘ generator=,,CGImap 0.0.2‘‘>
 <bounds minlat=,,54.088‘‘ minlon=,,12.248‘‘ maxlat=,,54.091‘‘ maxlon=,,12.252‘‘/>
 <node id=,,298884269‘‘ lat=,,54.09017‘‘ lon=,,12.248263‘‘ user=,,Somebody‘‘ .../>
 <node id=,,261728686‘‘ lat=,,54.09063‘‘ lon=,,12.244192‘‘ user=,,Somebody‘‘ .../>
 <node id=,,1831881213‘‘ lat=,,54.09006‘‘ lon=,,12.253938‘‘ user=,,Somebody‘‘ ...>
  <tag k=,,name‘‘ v=,,Some Place‘‘/>
  <tag k=,,traffic_sign‘‘ v=,,city_limit‘‘/>
 </node>
 ...
 <way id=,,26659127‘‘ user=,,Somebody‘‘ uid=,,55988‘‘ ...>
  <nd ref=,,292403538‘‘/>
  <nd ref=,,298884289‘‘/>
  ...
  <nd ref=,,261728686‘‘/>
  <tag k=,,highway‘‘ v=,,unclassified‘‘/>
  <tag k=,,name‘‘ v=,,Some street‘‘/>
 </way>
 <relation id=,,56688‘‘ user=,,Somebody‘‘ uid=,,56190‘‘ visible=,,true‘‘ ...>
  <member type=,,node‘‘ ref=,,294942404‘‘ role=,,‘‘/>
  ...
  <member type=,,node‘‘ ref=,,364933006‘‘ role=,,‘‘/>
  <member type=,,way‘‘ ref=,,4579143‘‘ role=,,‘‘/>
  ...
  <member type=,,node‘‘ ref=,,249673494‘‘ role=,,‘‘/>
  <tag k=,,name‘‘ v=,,Bus way‘‘/>
  <tag k=,,network‘‘ v=,,VVW‘‘/>
```

```
  <tag k=,,ref‘‘ v=,,123‘‘/>
  <tag k=,,route‘‘ v=,,bus‘‘/>
  <tag k=,,type‘‘ v=,,route‘‘/>
 </relation>
 ...
</osm>
```

Způsob popisu *OSM* ve formátu *XML*.

2.4 Ukázka použití OSM v ROS

V této část je popsán příklad, jak lze staženou mapu vložit do *ROS* a případně ji vizualizovat. Předpokladem je nainstalovaný *ROS* s balíky `osm_cartography`, `route_network`, `geographic_msgs`, `geodesy` a staženou mapu, resp. její výřez ve formátu *XML*.

Začneme spuštěním jádra systému.

```
$ roscore
```

K načtení mapy ze souboru využijeme nody z balíku `osm_cartography`, k tomuto účelu slouží node `osm_server`. Ten implementuje funkci `get_geographic_map`, která přijme parametrem cestu k souboru mapy a vrátí zpět data ve formátu `geographic_msgs`. Spustíme tedy `osm_server`

```
$ rosrun osm_cartography osm_server
```

Nyní můžeme zavoláním služby tohoto nodu získat mapová data ve zpracovatelném formátu. Pokud bychom chtěli mapu vizualizovat nástrojem *RViz*, musíme spustit `viz_osm`, který sám zavolá `osm_server`, získá mapu, převede ji na pole vizualizačních *RViz* bodů, které následně publikuje jako topic `visualization_marker_array`.

```
$ rosrun osm_cartography viz_osm _map_url='file:///cesta/k/mape/map.osm'
```

Teď už máme zdroj *RViz* bodů. Ještě než spustíme *RViz*, je pro tento druh zobrazení publikovat nějaký vztažný bod k mapě, kdybychom to neudělali, mapu bychom neviděli. Hodnoty je nutné získat ze zpráv, které publikuje vizualizér, jsou to souřadnice v UTM formátu.

```
$ rosrun tf static_transform_publisher ? ? ? ? ? ? ? map local_map 100
                        $ rosrun rviz rviz
```

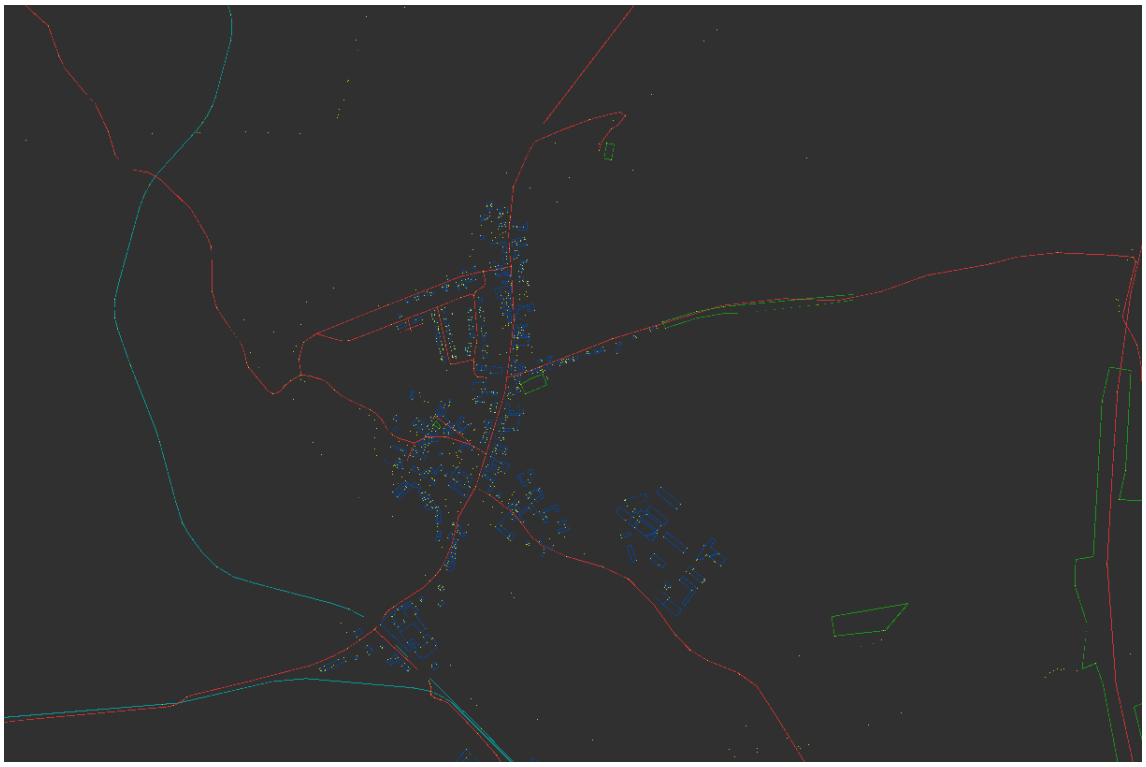
Na obrázku je patrné, že v běžně stažené mapě je mnoho pro navigaci zbytečných objektů – domy, řeky, lesy aj. Bylo by dobré tyto objekty vyfiltrovat již ze vstupního *XML* souboru nebo použít funkce z balíku `route_network`, které tuto funkcionalitu nabízí.

Ukončeme běžící `osm_cartography viz_osm` vizualizující celou mapu a spusťme `route_network`, který vyfiltruje pouze sjízdné cesty. Pokud dále spustíme `viz_routes`, můžeme si výsledek prohlédnout v nástroji *RViz*.

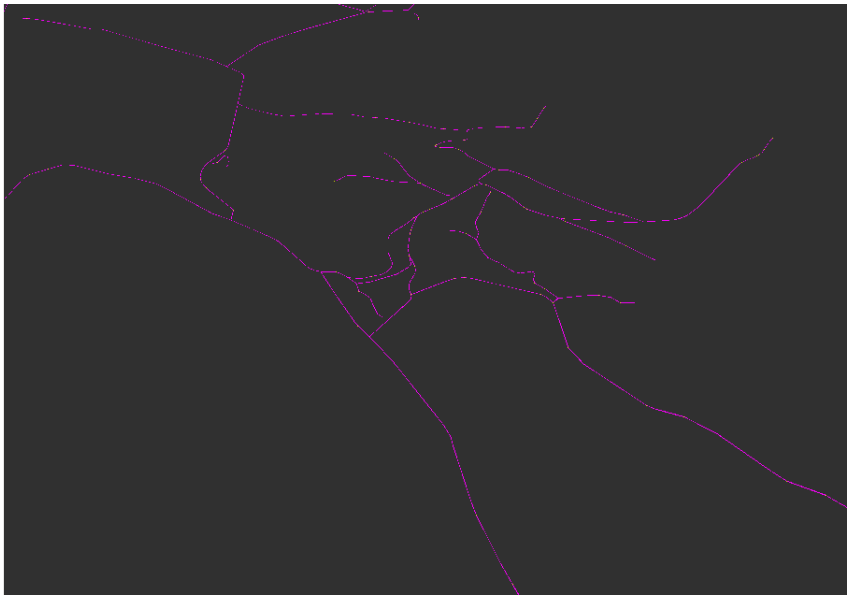
```
$ rosrun route_network route_network
  $ rosrun route_network viz_routes
```

Rozhraní pro plánování trasy – node `plan_route` nabízí službu pro plánování cesty `get_route_plan`.

```
                  $ rosrun route_network plan_route
$ rosservice call /get_route_plan 'route_network.uuid' 'source.uuid' 'goal.uuid'
```



Obrázek 2.2: Příklad vizualizace celé mapy nástrojem *RViz*.



Obrázek 2.3: Příklad vizualizace pouze cest nástrojem *RViz*.

2.5 Plánovací algoritmy

2.5.1 Dijkstrův algoritmus

Dijkstrův algoritmus slouží pro hledání všech nejkratších cest v grafu – mapa zjednodušena pouze na křižovatky a cesty mezi nimi, resp. uzly a hrany, hrany jsou nějakým způsobem ohodnoceny, nejčastěji délka hrany. Pokud uvažujeme možnost jednosměrných hran, jedná se o *orientovaný* graf. Podmínkou algoritmu jsou v grafu nezáporně ohodnocené hrany, v případě abstrakce reálných cest je tato podmínka vždy splněna. Nalezené cílové cesty jsou posloupnost uzlů, v kterém pořadí jimi procházíme.

Dijkstrův algoritmus nemá cílový bod, vyhodnocují se cesty ke všem uzlům v grafu. Pro vyhledávání cesty z jednoho bodu do druhého je vhodnější podobný algoritmus *uniform cost search – metoda stejných cen*, ten vyhledává pouze cestu mezi dvěma body. V praxi tyto dva algoritmy často splývají, protože jejich princip je podobný.

Algoritmus využívá prioritní frontu, kam ukládá uzly a jejich ohodnocení – nejkratší vzdálenost, jak se k nim z počátečního uzlu dostat. Uzly s nejmenším ohodnocením jsou ve frontě první. V počátečním stavu jsou ve frontě všechny uzly, počáteční uzel má ohodnocení *0*, ostatní mají nekonečno. Každý uzel si uchovává informaci, kdo je jeho aktuální předek, na konci se z těchto referencí sestaví nejkratší cesta k uzlu.

Nyní algoritmus běží ve smyčce dokud fronta není prázdná. Vybere se první prvek z fronty (opustí frontu) – při prvním průchodu počáteční uzel, postupně se otestuje pro všechny jeho potomky, jestli platí *ohodnocení aktuálního uzlu + ohodnocení hrany ¡ ohodnocení potomka*. Pokud ano, nastaví se ohodnocení uzlu potomka na *ohodnocení aktuálního uzlu + ohodnocení hrany* a jeho předka na aktuální uzel. Nyní se vybere opět první uzel ve frontě.

Po skončení algoritmu je k dispozici každý uzel ohodnocen nejkratší vzdáleností, kterou je možné se k němu dostat a pomocí referencí na předka lze tuto cestu rekonstruovat.

Složitost algoritmu závisí na způsobu implementace prioritní fronty a vyhledávání v ní.

2.5.2 A*

Algorimtus je principiálně podobný *Dijkstrově algoritmu* s rozdílem, že se nevyhodnocuje celý graf, ale pouze se v něm hledá cesta mezi dvěma body. Patří mezi informované metody prohledávání stavového prostoru, protože jsou známa ohodnocení hran a při hledání je vybrána vždy nejnadějnější možnost. U neinformovaných (slepých) metod se naopak pouze slepě prohledávají podle určitého řádu uzly, dokud není nenalezen cíl. Při vybírání, který následující uzel je nejvhodnější se využívá heuristických funkcí.

Ty mají za cíl optimalizovat rychlost hledání cesty, nehledá se pouze na základě délek hran, ale nejoptimálnější nebo nejlevnější cesta podle různých faktorů, které vezmeme v heuristické funkci v úvahu. Nejjednodušší heuristikou při vyhledávání v mapě bývá přímá vzdálenost z bodu k cíli.

Algoritmus je *úplný* i *optimální* – Vždy nalezne cíl, pokud existuje, a cesta k němu je nejlepší možná.

```
f(x) = g(x) + h(x)
```

Funkce `f` pro řazení v prioritní frontě je dána funkcí `g` ohodnocující cestu po hranách, ale také funkcí `h` představující heuristiku.

2.5.3 UCS – metoda stejných cen

Zjednodušení algoritmu *A**, kdy neuvažujeme heuristiky a vhodný následující uzel hledáme jen na základě délek hran – stejně jako u *Dijkstrova algoritmu*.

Algoritmus je *úplný* i *optimální* stejně jako *A**, jen je potřeba k jeho provedení prozkoumat více uzlů – možných odboček po cestě, které se jeví nadějně. Algoritmus se řadí mezi slepé metody.

```
f(x) = g(x)
```

2.5.4 Greedy Search – hladové vyhledávání

Zjednodušený algoritmus *A**, kde se tentokrát nehledí na délky hran, ale jen na heuristiku – například přímou cestu k cíli.

Algoritmus je *úplný*, ale není *optimální*.

```
f(x) = h(x)
```

Kapitola 3

Existující řešení a návrh implementace

Nyní jsme schopni s již implementovanými komponentami dosáhnout plánování cesty. Tato kapitola se zabývá, jakým způsobem pro tento účel komponenty poskládat a jak je toto řešení efektivní. Dále určuje, co bude naším vlastním řešením – které části implementujeme a které využijeme ve stavu, jakém jsou k dispozici.

3.1 Existující možnosti použití

S balíky popsanými v 2.2 jsem byl schopen staženou *OSM* načíst do *ROS* pomocí nodu `osm_server`. Odtud lze získat mapu ve formátu zprávy `geographic_msgs/GetGeographicMap`. Tu lze ještě vyfiltrovat prostřednictvím nodu `route_network` tak, aby obsahovala informace jen o cestách – zpráva typu `geographic_msgs/RouteNetwork`. A nyní se nodem `plan_route` dá zjistit nejkratší cesta voláním služby `get_route_plan` (Služba využívá algoritmu A*).

Tato služba ale umožňuje pouze vyhledat cestu v mapě z bodu A do bodu B, navíc přijímá pouze parametry ve formátu `UniqueID`, jejichž zadávání není uživatelsky přívětivé.

Výše popsané balíky jsou implementovány v jazyce *Python*, což na uživatelské úrovni ničemu nevadí, ale náš projektový balík by měl být implementován pouze v *C/C++*. Proto nebudou upravovány již existující balíky nebo od nich přebírán zdrojový kód. Pouze bude využíváno některých jejich komponent.

Naopak balíky z *navigation* stacku se ukázaly pro účel implementace knihovny s metodami plánování cesty jako nevhodné. Navigation stack je zaměřen na zpracování senzorických dat robota a jejich následný převod na navigační pokyny pro platformu robota. Zde implementované metody plánování cesty probíhají nad pravděpodobnostní mapou, která se sestavuje právě ze senzorických dat[2].

3.2 Co a jak implementovat

Základem implementace bude vlastní plánovač, který bude schopen pracovat s různými typy vstupů, bude nezávislý na použitém plánovacím algoritmu a bude vylepšovat existující řešení o další funkce. Jako vhodné se jeví možnost specifikace cesty pomocí více bodů. Zvolil jsem dvě množiny bodů, jednu pro body, kterým se chceme vyhnout, a druhou pro body, kterými chceme zaručeně projít.

Tyto funkce by neměly záviset na použitém plánovacím algoritmu, ten vždy pouze implementuje jednotné rozhraní, které stanovuje, že má hledat cestu mezi dvěma vstupními body.

Konečnou nalezenou cestu by bylo vhodné názorně vizualizovat použitím *RViz*, služba `plan_route` vrací cestu ve formě zprávy `geographic_msgs/RoutePath`, potřebujeme implementovat metodu umožňující cestu zvýraznit mezi ostatními cestami. Balík `route_network` sice obsahuje node `viz_plan`, který je k tomu určený, ale není v současné verzi zdokumentovaný a při bližším zkoumání funkčnosti se ukázalo, že pouze neustále plánuje cesty mezi dvěma náhodnými body, které pak vizualizuje.

3.2.1 Plánovací algoritmy

Plánovací algoritmy budou implementovány tři – *A** (2.5.2), *UCS* (2.5.3) a *Greedy Search* (2.5.4). Mohl by být implementován jakýkoliv algoritmus, ale tyto tři podobné algoritmy byly, díky jejich zohledňovaní prostorových vlastností, určeny jako nejvhodnější.

```
mějme prioritní frontu OPEN a seznam CLOSED
do fronty OPEN vlož počáteční bod

dokud není fronta OPEN prázdná prováděj:
- vyber z fronty OPEN uzel s nejnižším ohodnocením f(x)
- je-li tento uzel cílový, vyhledávání bylo úspěšné, vrať cestu k tomuto bodu
- vlož bezprostřední následníky do fronty open i s jejich ohodnocením f(x),
  ale jen ty, které nejsou na seznamu CLOSED
  a zároveň nejsou už v OPEN s nižším ohodnocením
- aktuální uzel vlož na seznam CLOSED

cyklus skončil, cestu nelze najít
```

Společný pseudokód implementovaných plánovacích algoritmů. [1]

3.2.2 Vylepšení rozhraní služeb

Dosud bylo možné koncové a cílové body definovat pouze ve formátu `UniqueID`, které je zejména při volání služby ručně z příkazové řádky značně nepohodlné. Lepší by bylo službu volat s běžnými *WGS84* souřadnicemi nebo identifikačními čísly waypointů z OSM.

3.2.3 Rozšířená specifikace cesty

Je možné specifikovat body, kterým se v mapě vyhnout a kterými naopak projít.

Aby se dosáhlo vynechání určitých bodů, z mapy cest ve formátu `geographic_msgs/RouteNetwork` jednoduše odstraníme ty segmenty cesty, které tyto body mají jako počáteční nebo koncové.

Většinou to jsou alespoň 4 různé segmenty – 2 segmenty do bodu a od bodu a stejně i v opačném směru, protože u obousměrných cest jsou definovány segmenty s oběma kombinacemi počátečního i koncového bodu. Dále to přirozeně neplatí pro koncové body slepých ulic.

U množiny bodů, kterými projít, je to složitější. Tyto body jsou bez pořadí, nejedná se tedy o sekvenci, kdy bychom šli od bodu k bodu v přesně stanoveném pořadí. Je nutné najít pořádí, v kterém když projdeme, tak cesta bude nejkratší.

Plánovač bude implementovat konceptuálně nejjednodušší řešení – zkusíme naplánovat všechny kombinace pořadí bodů a vrátíme jen to nejkratší. Toto řešení je nejvhodnější pro malý počet bodů, protože s každým přidaným bodem do této množiny počet prohledávaných řešení velmi roste – `n!` (1, 2, 6, 24...).

3.2.4 Vizualizace cesty

Bylo rozhodnuto o vytvoření vlastního vizualizéru, který bude barevně vyznačovat pouze požadovanou cestu. Jako nejvhodnější se ukázala implementace, kdy vizualizér odebírá topic, jehož zprávy jsou ve formátu `geographic_msgs/RoutePath`, které reprezentují jednu cestu a vizualizér bude zobrazovat jen tu poslední přijatou do doby než přijme další. Zápis do tohoto topicu není dosud nikde implementovaný, musí probíhat buď ručně nebo musí být implementován v nějaké obalující funkci, která získá plán cesty a poté jej pošle na vizualizaci.

3.2.5 Přehled použitých zpráv a služeb z jiných balíků

- `geographic_msgs/BoundingBox`

  Zpráva reprezentující hranice mapy, které určují dva body – nejvíce jihozápadní a nejvíce severovýchodní.

- `geographic_msgs/GeoPoint`

  Zpráva představující bod v prostoru definovaný *WGS84* souřadnicemi (zeměpisnou šířkou a délkou) a nadmořskou výškou (tu ve zpracovaných mapách neuvažujeme).

- `geographic_msgs/WayPoint`

  Zpráva obohacující `geographic_msgs/GeoPoint` o další atributy, výsledek reprezentuje WayPoint v OSM.

- `geographic_msgs/RouteSegment`

  Zpráva znázorňující jeden díl cesty v jednom směru, obsahuje počáteční a koncový bod ve formě reference (`uuid_msgs/UniqueID`) na ID `geographic_msgs/WayPoint`.

- `geographic_msgs/RouteNetwork`

  Tato zpráva představuje síť cest, kterou získáme pročištěním celé mapy od zbytečných objektů. Zprávy tohoto typu produkuje balík `route_network`, jehož implementace pracuje se všemi typy cest stejně. Zpráva je složena z metadat identifkujících síť, ohraničujícího pole `geographic_msgs/BoundingBox`, pole bodů `geographic_msgs/WayPoint[]` a pole segmentů `geographic_msgs/RouteSegment[]`.

- `geographic_msgs/RoutePath`

  Zpráva představuje cestu v síti cest. Obsahuje referenci na určitou síť cest a pole referencí na segmenty, kterými cesta vede.

- `uuid_msgs/UniqueID`

  Univerzální unikátní identifikátor[10], celkem 128-bitů, je možné jej zapsat více způsoby, například zde je definován jako pole 16 osmibitových čísel. Pro praktické použití a snadnější zápis se převádí na řetězec hexadecimálních čísel.

```
[23, 75, 63, 177, 77, 41, 80, 100, 187, 117, 41, 82, 223, 252, 138, 17]
174b3fb1-4d29-5064-bb75-2952dffc8a11
```

Dva způsoby zápisu UniqueID, první je způsob implementace v ROS, druhý je zápis kanonickou formou.

- `geographic_msgs/GetRoutePlan`

  Rozhraní služby pro získání plánu hledané cesty. Požaduje referenci na počáteční a koncový bod, vrací nalezenou cestu a indikátory úspěšného nalezení. Definice této služby je popsána v 2.1.5.

3.2.6 Struktura implementované knihovny

Knihovna bude sdružovat různé nástroje pro práci s OSM, proto její název bude obecný – `osm_tools`.

Funkce budou obstarávat 3 nody – `planner`, `osm_tools` a `viz_plan`.

- `planner`

  Node zajišťující plánování trasy, spravuje jednotlivé plánovací algoritmy, které uživatel specifikuje pomocí parametrového serveru (2.1.7). Nabízí službu `osm_tools/GetRoutePlanExt`, která vychází z `geographic_msgs/GetRoutePlan`, ale specifikuje navíc body, kterým je se třeba vyhnout a kterými je třeba projít.

- `osm_tools`

  Tento node nabízí implementované rozhraní dalších služeb, které mají za cíl zjednodušit a urychlit plánování cesty a poté její vizualizaci. Implementuje službu `osm_tools/VizRoutePlanExt`, která kontaktuje `planner`, aby naplánoval cestu, tu však nevrací a místo toho ji posílá nodu `viz_plan` k vizualizaci.

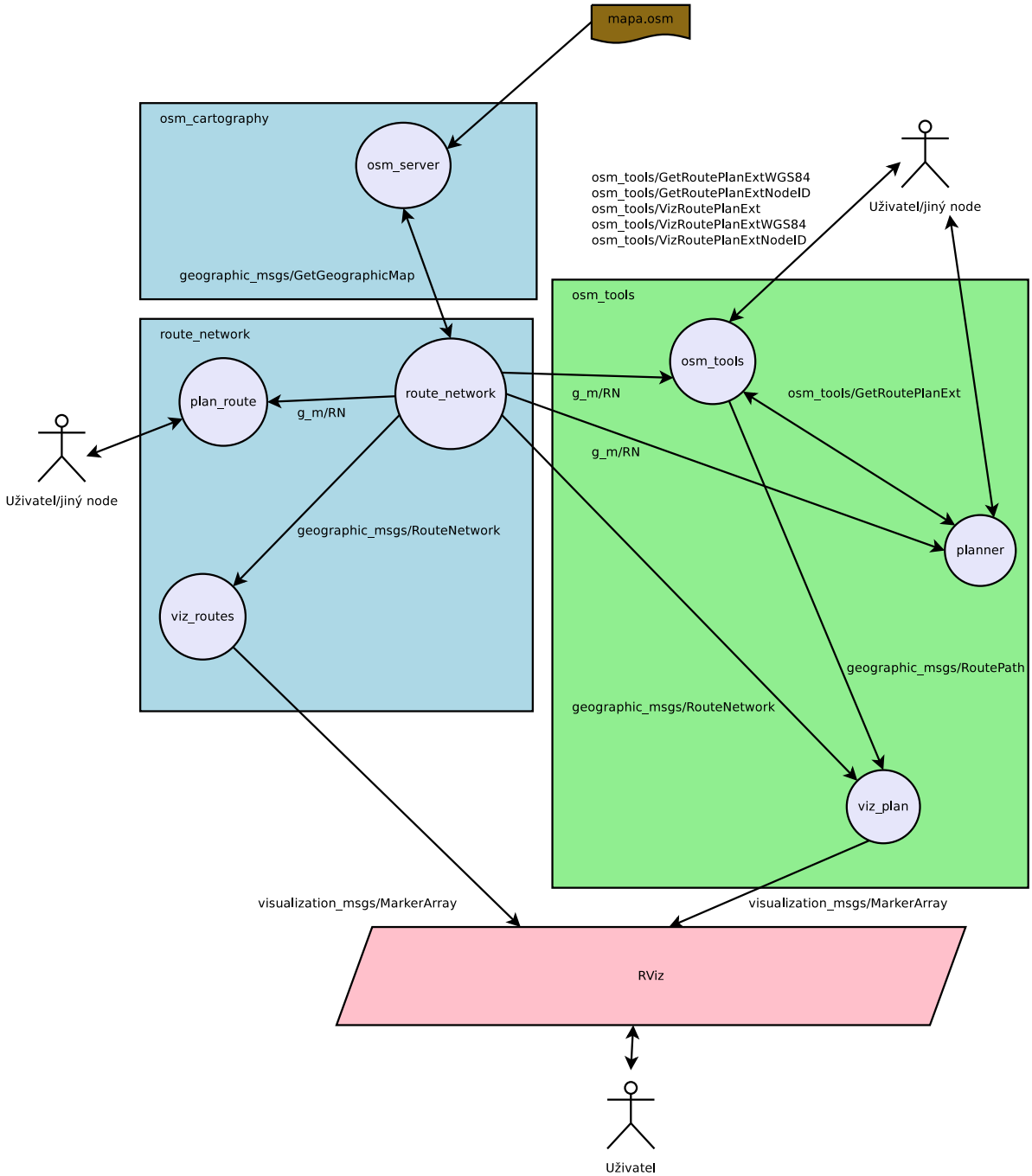
  Dále implementuje služby
  `osm_tools/GetRoutePlanExtWGS84` a
  `osm_tools/VizRoutePlanExtWGS84`, které se od předchozích liší tím, že specifikují klíčové body ve formátu *WGS84* – ve zprávě `geographic_msgs/GeoPoint`.

  Analogicky implementuje služby
  `osm_tools/GetRoutePlanExtNodeID` a
  `osm_tools/VizRoutePlanExtNodeID`, jejichž parametry jsou specifikovány ID (čísly) uzlů v *OSM*.

  Také tu existují samostatné služby pro funkce, kterých se ve výše popsaných službách využívá – `osm_tools/GetClosestWayPoint` a `osm_tools/GetWayPoint`. První služba vrací nejbližší `geographic_msgs/WayPoint` k zadanému `geographic_msgs/GeoPoint` bodu. Druhá služba vrací `geographic_msgs/WayPoint`, který odpovídá zadanému ID nodu v *OSM*.

- `viz_plan`

  Node implementující vizualizaci naplánovaných cest. Viz 3.2.4.



Obrázek 3.1: Znázornění spolupráce jednotlivých balíků a nodů.

Kapitola 4

Implementace a ověření

V předchozí kapitole jsem definoval, které nody a služby budou vytvořeny. V této kapitole jsou popsány detaily implementace jednotlivých částí i balíku jako celku.

Implementace probíhá v programovacím jazyce *C/C++*, který patří mezi podporované pro psaní modulů pro *ROS*, interně je v *ROS* tato podpora implementována balíkem `roscpp`.

Balík implementovaný touto prací je pojmenován `osm_tools` a je spravován nástrojem *catkin*.

4.1 Konfigurace balíku

Vývoj balíku probíhá v pracovním adresáři catkinu. Catkin vyžaduje mít závislosti a další informace definované v konfiguračních souborech `CMakeLists.txt` a `package.xml`.

V souboru `CMakeLists.txt` jsou definovány závislosti na jiných balících pod položkami:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
  message_generation
  visualization_msgs
  unique_id
  geodesy
  geographic_msgs
)
...
catkin_package(
  CATKIN_DEPENDS unique_id
)
```

Protože je použit adresář s hlavičkovými soubory `include` a určitá část balíku se přeloží do samostatné knihovny, je nutné o tom dát vědět správci balíků:

```
include_directories(include)
```

Je nutné se odkázat na všechny definované soubory se službami a zprávami a také definovat generované zprávy:

```
add_service_files(
  FILES
  GetClosestWayPoint.srv
  GetWayPoint.srv
  GetRoutePlanExt.srv
  GetRoutePlanExtWGS84.srv
  GetRoutePlanExtNodeID.srv
  VizRoutePlanExt.srv
  VizRoutePlanExtWGS84.srv
  VizRoutePlanExtNodeID.srv
)
...
generate_messages(
  DEPENDENCIES
  std_msgs
  geographic_msgs
  uuid_msgs
  visualization_msgs
)
```

Nakonec je nutné říct, které knihovny a spustitelné nody se mají přeložit a kde jsou jejich zdrojové texty:

```
add_executable(viz_plan src/viz_plan.cpp)
target_link_libraries(viz_plan ${catkin_LIBRARIES})

add_executable(osm_tools src/osm_tools.cpp)
target_link_libraries(osm_tools ${catkin_LIBRARIES})

# Planning Algorithms
add_library(common src/plan_algorithms/common.cpp)
target_link_libraries(common ${catkin_LIBRARIES})
add_library(astar src/plan_algorithms/astar.cpp)
target_link_libraries(astar ${catkin_LIBRARIES} common)
add_library(ucs src/plan_algorithms/ucs.cpp)
target_link_libraries(ucs ${catkin_LIBRARIES} common)
add_library(greedy src/plan_algorithms/greedy.cpp)
target_link_libraries(greedy ${catkin_LIBRARIES} common)


add_executable(planner src/planner.cpp)
target_link_libraries(planner ${catkin_LIBRARIES} common astar ucs greedy)
```

V souboru `package.xml` jsou už pak obecnější informace o balíku jako celku a na kterých ostatních balících závisí v době překladu a v době běhu.

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>unique_id</build_depend>
```

```
<build_depend>geographic_msgs</build_depend>

<run_depend>roscpp</run_depend>
<run_depend>unique_id</run_depend>
<run_depend>geographic_msgs</run_depend>
<run_depend>osm_cartography</run_depend>
<run_depend>route_network</run_depend>
```

4.1.1 Verze použitých balíků

Další balíky: `geodesy`, `geographic_msgs`, `osm_cartography`, `route_network`, `uuid_msgs` a `unique_id` byly při vytváření balíku také v pracovním adresáři, protože součástí standardní instalace verze *Hydro* nebyly jejich aktuální verze nebo tam nebyly obsaženy vůbec, proto pochází z *git* repozitářů s verzemi aktuálními k době začátku práce na balíku.

- `geodesy` – 0.3.1
- `geographic_msgs` – 0.3.1
- `osm_cartography` – 0.2.1
- `route_network` – 0.2.1
- `uuid_msgs` – 1.0.3
- `unique_id` – 1.0.3

Kompilace všech balíčků v pracovním adresáři catkinu je proveditelná příkazem `catkin_make` v kořenovém adresáři.

4.2 Implementace plánovacích algoritmů

Použití plánovacích algoritmů je modelováno pomocí objektově orientovaného programování, kdy každý plánovací algoritmus implementuje rozhraní třídy `PlanAlgorithm`. Konkrétní algoritmus od této třídy dědí, využívá pak funkce, které jsou pro všechny algoritmy společné, a implementuje ty potřebné.

Třída `PlanAlgorithm` obsahuje následující `protected`[1] proměnné:

- `std::map<std::string, geographic_msgs::RouteSegment> segments` – mapa všech segmentů, ke které slouží jako klíč `UniqueID` daného segmentu.
- `std::map<std::string, geographic_msgs::WayPoint> points` – mapa všech bodů, ke které slouží jako klíč `UniqueID` daného bodu.
- `std::map<std::string, std::vector<uuid_msgs::UniqueID> > segments_from_point` – pomocná mapa s odkazy na všechny segmenty, které vychází z daného bodu, jehož `UniqueID` slouží jako klíč.

Implementované metody této třídy jsou:

[1] Modifikátor viditelnosti potřebný, aby tyto atributy byly viditelné i v odvozených třídách

- `void prepareData(geographic_msgs::RouteNetwork &network)`

  Metoda, která plní výše popsané struktury daty.

  Zatímco data uložená v `geographic_msgs::RouteNetwork` jsou v lineárních datových strukturách, které jsou nevhodné pro časté vyhledávání, data ve výše definovaných mapách jsou přístupná klíčem. Pole všech prvků se projde jen jednou a další výběr prvků z mapy je rychlý. Nevýhodou jsou vyšší paměťové nároky.

  Mapy je nutné před každým naplněním vyprázdnit, protože běžící instance třídy může pracovat s různými mapami po sobě, v opačném případě by v mapách zůstávala nerelevantní data.

- `double getLength(geographic_msgs::RouteSegment &segment)` – metoda počítá délku daného segmentu, úkol je převeden na vzdálenost mezi dvěma body a předán funkci `getDistance`

- `double getDistance(WayPoint &a, WayPoint &b)` – metoda vyhodnocující vzdálenost mezi dvěma body, využívá se vztah *Pythagorovy věty* pro vzdálenost ve 2D prostoru.

Dále je definována virtuální metoda `virtual double getPlan(...)`, jejíž implementace je na konkrétním algoritmu – odvozené třídě. Metoda vrací celkovou procestovanou vzdálenost.

4.2.1 A*

Vzhledem k tomu, že všechny implementované algoritmy se liší jen v detailech, bude popsána pouze implementace třídy `AStarAlgorithm` vycházející ze pseudokódu definovaného v 3.2.1.

Klíčovou částí je implementace metody `getPlan(...)`, které se dodávají jako parametry síť cest, počáteční a koncový bod a struktura s výslednou cestou, která se má vyplnit.

Po zavolání metody je potřeba naplnit datové struktury aktuální sítí `prepareData(...)`. Poté definujeme datové struktury pro frontu *OPEN*, seznam *CLOSED* a pro uchování zpáteční cesty.

Prvky, s kterými se pracuje, nejsou body, ale segmenty. Při použití bodů by bylo nutné řešit režii navíc, když bychom skládali výslednou cestu, která je složena ze segmentů. Funkce algoritmu zůstala neporušena.

- `std::map<std::string, std::pair<double, double> > open`

  Fronta *OPEN* je implementována jako mapa, klíčem je `UniqueID` daného segmentu, hodnotou je dvojice *cena* a *výsledek heuristické funkce*. Při hledání vhodné struktury v jazyce *C++* by testována i prioritní fronta, kde ale byla obtížná manipulace s prvky uvnitř fronty.

  Pro výběr minimálního prvku slouží samostatná metoda `getMinKey(&map)` vracející `UniqueID` klíč do mapy. Pro seřazování prvků metoda využívá přetěžování[2] funkce `operator()`.

  Protože využíváme místo bodů segmenty, tak se jako první prvek do fronty *OPEN* vloží nepravý segment, který má nastaven body na startovní bod.

[2] Stav, kdy existuje více funkcí se stejným názvem, ale odlišnými parametry a implementací.

- `std::map<std::string, bool> closed`

  Seznam *CLOSED* je implementován jako mapa, kde je klíčem bod a hodnotou bitová informace, která značí, zda byl uzel uzavřen. Výhodou je rychlé nastavení i ověření, zda je bod už na tomto seznamu.

- `std::map<std::string, std::string> plan_map`

  Mapa pro udržení vztahu mezi předchozími a následujícími segmenty, klíčem je aktuální segment, hodnotou je předchozí segment.

  Při rekonstrukci cesty jen procházíme mapou od koncového segmentu k prvnímu.

  Segment se expanduje tak, že se vezme jeho koncový bod, indexuje se s ním do mapy `segments_from_point`, z které se získají následující segmenty.

4.3 Implementace nodů

Každý node je implementován v jednom `*.cpp` souboru. Nody nemají své vlastní hlavičkové soubory.

4.3.1 planner

Tento node spravuje plánovací algoritmy a nabízí službu `GetRoutePlanExt`.

Při spuštění se v parametrovém serveru kontroluje parametr `plan_algorithm`, který má definované následující hodnoty:

- astar
- ucs
- greedy

Podle přijatého parametru se instancuje[3] stejnojmenný plánovač. Pokud je přijata jiná hodnota nebo není definována žádná, použije se plánovač *A**.

Tento node implementuje rozšířené funkce pro specifikaci cesty, jak je popsáno v 3.2.3. Při vytváření množiny segmentů bez bodů, kterým je se v cestě třeba vyhnout, si algoritmus pamatuje původní množinu a po skončení vyhledávání ji zase obnoví. Kdyby se tak nestalo, docházelo by ke ztrátám segmentů v běžícím nodu plánovače.

Posloupností bodů, kterými musí vést cesta, se generují do dvourozměrného pole `all_possible_paths`, jednotlivé cesty se poté vyhodnocují bod po bodu. Stačí jim funkce poskytovaná plánovacím algoritmem, která plánuje jen mezi dvěma body. Nevýhodou je, že se při opakovaných voláních plánovacího algoritmu a zejména při prozkoumávání kombinací mnoha bodů doba potřebná k nalezení optimální cesty rychle zvyšuje. Pro generování kombinací bodů v poli se používá funkce `std::next_permutation`.

Node má připravené rozhraní pro obsluhu více sítí zároveň. Stejně jako ostatní nody přijímá sítě, které produkuje node `route_network`, ale zde se neudržuje jen poslední přijatá. Sítě se ukládají do mapy `nets`, klíčem je `UniqueID` sítě.

[3] Vytvoří se objekt podle definice dané třídy.

4.3.2 osm_tools

Node poskytující především uživatelsky přívětivější služby pro plánování a vizualizaci, což je popsáno v 3.2.6. Node dokáže pracovat najednou jen s jednou sítí, zde to má i výhody – při volání služeb, které node nabízí, není třeba specifikovat `UniqueID` sítě.

Způsob lokalizace daného *WayPointu* z *WGS84* souřadnic je implementován jako lineární procházení polem všech bodů a nalezení toho, který je zadaným souřadnicím ve 2D prostoru nejblíže. Při určování limitní maximální vzdálenosti mezi body se používá vzdálenost mezi dvěma protilehlými rohovými body v obdelníku ohraničujicím mapu.

Pro získání *WayPointu* z jeho ID v OSM je třeba vědět, jakým způsobem jsou *WayPointy* tvořeny, protože *WayPoint* si toto ID přímo neuchovává. Po analýze kódu nodu `osm_server` je patrné, že se používá adresa ve tvaru:

`http://openstreetmap.org/node/123456`

Ta se pomocí funkcí balíku `geodesy` dá převést na `UniqueID`, kterým můžeme už snadno zjistit požadovaný *WayPoint*.

4.3.3 viz_plan

Tento node má na starost publikování *RViz* souřadnic (zpráva `visualization_msgs::Marker`) vytvořených z plánů cest, které odebere z topicu `route_plan`. Podobně jako předchozí nody, vytváří si i `viz_plan` mapy se segmenty a body pro rychlejší vyhledávání.

Výslednou grafickou cestu zapisuje ve formátu zpráv `visualization_msgs::MarkerArray` do topicu `visualization_marker_array`. Jedná se o stejný topic, do jakého zapisuje i `viz_routes` – v *RViz* stačí nastavit odebírání pouze tohoto topicu a bude se zobrazovat mapa i plán v ní vyznačený zároveň.

Vytváření pole `MarkerArray` má na starosti metoda `createMarkers()`, která se aktivuje kdykoliv se v topicu `route_plan` objeví nový plán. Nejprve smaže pole vytvořené z předchozího plánu. Poté začne v cyklu vytvářet `Marker` pro každý segment. Je nutné definovat jejich id, typ, měřítko, barvu, životnost, počáteční a koncový bod a další atributy. Souřadnice bodů je potřeba převést z formátu *WGS84* na formát *UTM* pomocí funkcí z balíku `geodesy`.

Každý `Marker` má omezenou životnost, aby byly vizualizovány trvale, je v nodu definovaný `Timer`, který v pravidelných intervalech volá metodu `publishMarkersCallback`. Ta prochází každý `Marker` a aktualizuje jejich timestamp[4], poté celé pole znovu publikuje.

Životnost `Markeru` je nastavena na `ros::Duration(5)`, interval časovače na `ros::Duration(4)`.

Je implementováno, aby segmenty cesty byly zvýrazněny bílou barvou, první segment je červený, poslední segment je zelený. Při plánování složitějších cest s více body může docházet k vracení se po cestě, kde už segmenty zvýrazněny jsou – zobrazené segmenty mohou být překryty segmenty jiné barvy.

4.4 Ověření funkčnosti

V podadresáři `test` balíku `osm_tools` je implementována sada testů ve formě shellových skriptů. *ROS* sice podporuje vlastní **.launch* soubory, ale jejich možnosti nejsou tak rozsáhlé, proto jsem se rozhodl pro tuto variantu.

[4] časové razítko pro evidenci stáří dat

Pro jejich reprodukci na konkrétním stroji je nutné upravit cesty v jednotlivých skriptech a mít nainstalovanou unixovou utilitu *screen*.

Nejdříve je třeba spustit skript `launch-astar.sh` který spustí všechny potřebné nody s plánovačem *A**. Poté je tu skript s několika úlohami na základní ověření funkčnosti – bylo by zbytečné demonstrovat všechny kombinace parametrů, protože jejich použití je repetivní. Testovací skript po každé vykonané úloze pozastaví činnost do stisknutí nějaké klávesy, aby bylo možné zkontrolovat výsledky. Testování probíhá nad mapou parku se sítí cestiček – `park.osm`, jedná se o exportovaný park z *Brna – Lužánky*[5].

Průběh spuštění skriptu `test-park.sh`:

- Ověření úspěšnosti získání bodu z relevantních *WGS84* souřadnic.

  `get_closest_waypoint [49.20642,16.60951,0.0]`

  Výsledkem je platný bod.

```
success: True
status: ''
waypoint:
  id:
    uuid: [1, 66, 64, 220, 35, 66, 89, 90, 146, 130, 197, 116, 9, 180, 164, 50]
  position:
    latitude: 49.2063401
    longitude: 16.609204
    altitude: nan
  props:
    -
      key: created_by
      value: JOSM
```

- Ověření neúspěšnosti získání bodu z nerelevantních *WGS84* souřadnic.

  `get_closest_waypoint [52.3244,17.77447,0.0]`

  Výsledkem je prázdný bod, neúspěšná operace a chybové hlášení.

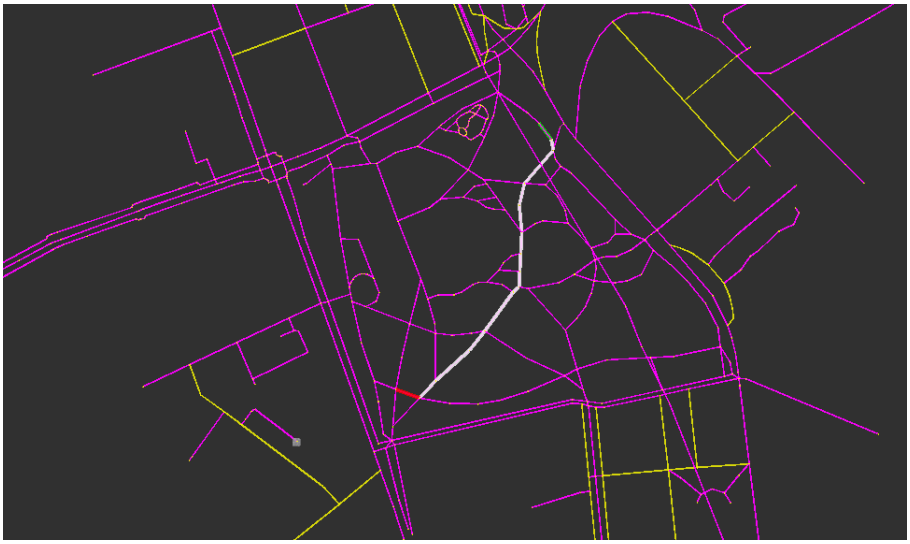
```
success: False
status: Cannot find closest point!
waypoint:
  id:
    uuid: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  position:
    latitude: 0.0
    longitude: 0.0
    altitude: 0.0
  props: []
```

- Ověření úspěšnosti získání bodu z OSM Nodu daného číslem.

  `get_waypoint 56235820`

  Výsledkem je platný bod.

[5] http://cs.wikipedia.org/wiki/Lužánky



Obrázek 4.1: Nejkratší nalezená vzdálenost mezi dvěma body definovanými souřadnicemi *WGS84*.

- Ověření neúspěšnosti získání bodu z neexistujícího OSM Nodu[6] daného číslem.

  `get_waypoint 75232830`

  Výsledkem je prázdný bod, neúspěšná operace a chybové hlášení.

- Ověření neúspěšnosti získání plánu cesty z nerelevantních *WGS84* souřadnic.

  `get_route_plan_ext_wgs [60.20463,16.60693,0.0] [49.20804,16.60958,0.0] [] []`

  Výsledkem je prázdná množina segmentů, neúspěšná operace a chybové hlášení.

- Ověření úspěšnosti získání plánu cesty z relevantních *WGS84* souřadnic.

  `get_route_plan_ext_wgs [49.20463,16.60693,0.0] [49.20804,16.60958,0.0] [] []`

  Výsledkem je množina naplněná segmenty a úspěšná operace.

- Ověření vizualizace plánu cesty s *WGS84* souřadnicemi.

  `viz_route_plan_ext_wgs [49.20463,16.60693,0.0] [49.20804,16.60958,0.0] [] []`

  Výsledkem je úspěšná operace a výstup v *RViz* (viz obrázek 4.1).

- Ověření vizualizace s definovaným průchozím bodem.

  `viz_route_plan_ext_wgs [49.20463,16.60693,0.0]`
  `[49.20804,16.60958,0.0] [] [[49.20634,16.61072,0.0]]`

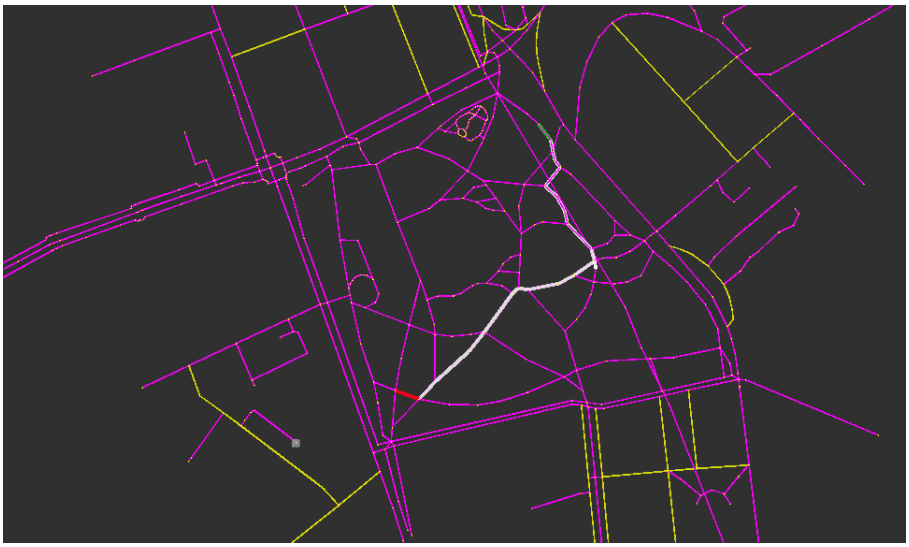
  Výsledkem je úspěšná operace a výstup v *RViz* (viz obrázek 4.2).

- Ověření vizualizace plánu cesty s body definovanými OSM ID.

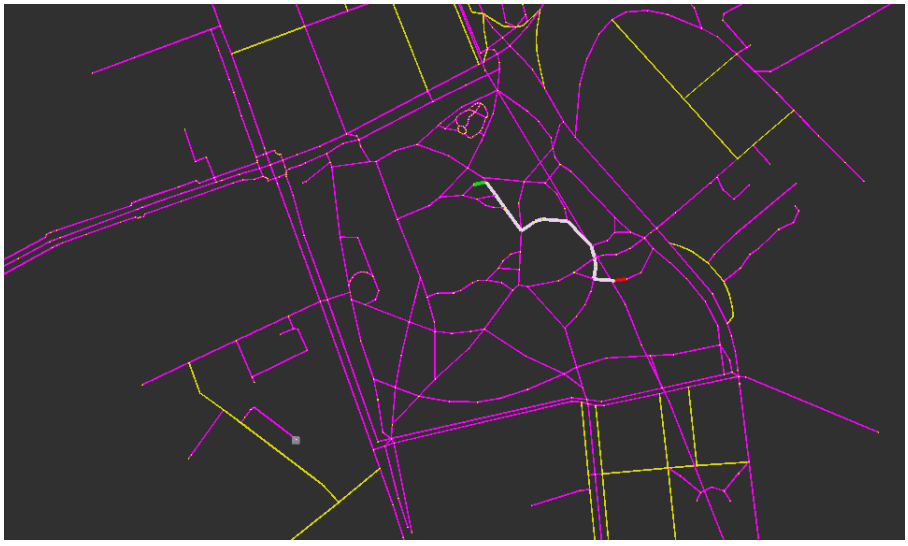
  `viz_route_plan_ext_node 56235757 54747974 [] []`

  Výsledkem je úspěšná operace a výstup v *RViz* (viz obrázek 4.3).

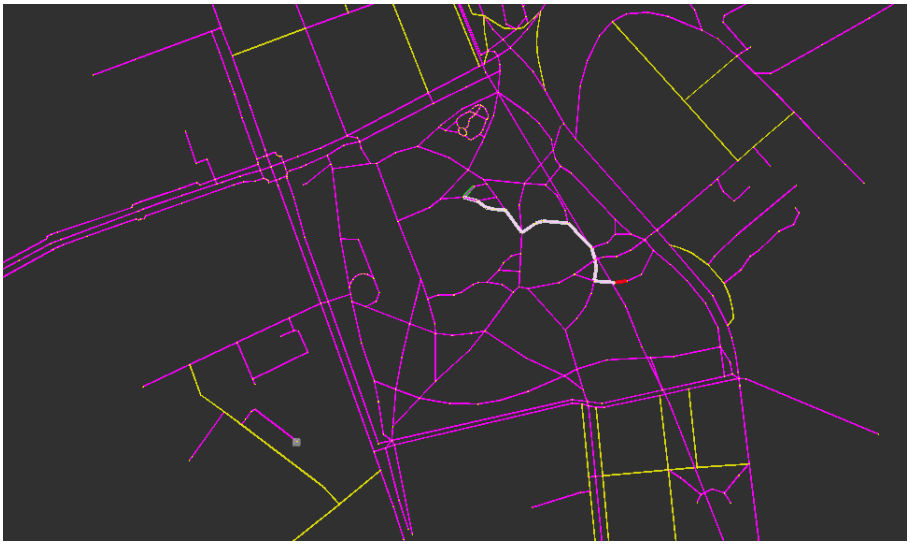
[6] Neexistujícího alespoň v aktuální mapě.



Obrázek 4.2: Nejkratší nalezená vzdálenost mezi třemi body definovanými souřadnicemi *WGS84*.



Obrázek 4.3: Nejkratší nalezená vzdálenost mezi dvěma body definovanými ID Nodů z OSM.



Obrázek 4.4: Nejkratší nalezená vzdálenost mezi dvěma body definovanými ID Nodů z OSM. Je odstraněn jeden bod, kterým by normálně vedla nejkratší cesta.

- Ověření vizualizace plánu cesty s body definovanými OSM ID, je zakázán průchod jedním bodem poblíž koncového bodu.

  `viz_route_plan_ext_node 56235757 54747974 [54747969] []`

  Výsledkem je úspěšná operace a výstup v *RViz* (viz obrázek 4.4).

- Ověření vizualizace plánu cesty s body definovanými OSM ID, přidáme navíc některé body, kterými se musí projít.

  `viz_route_plan_ext_node 56235757 54747974 [54747969] [54747962,26365073]`

  Výsledkem je úspěšná operace a výstup v *RViz* (viz obrázek 4.5).

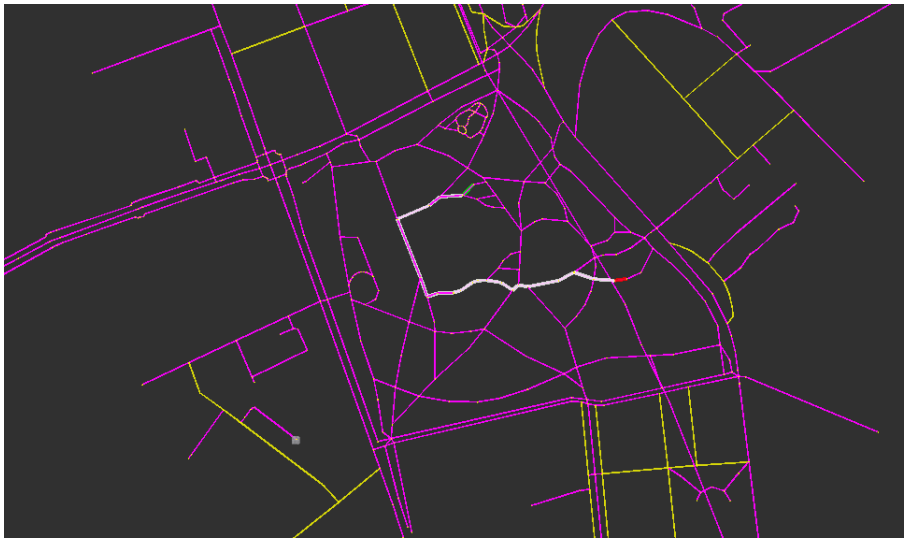
- Ověření vizualizace plánu cesty s body definovanými OSM ID, navíc je zakázán další bod na cestě z minulé úlohy.

  `viz_route_plan_ext_node 56235757 54747974 [54747969,56235839] [54747962,26365073]`

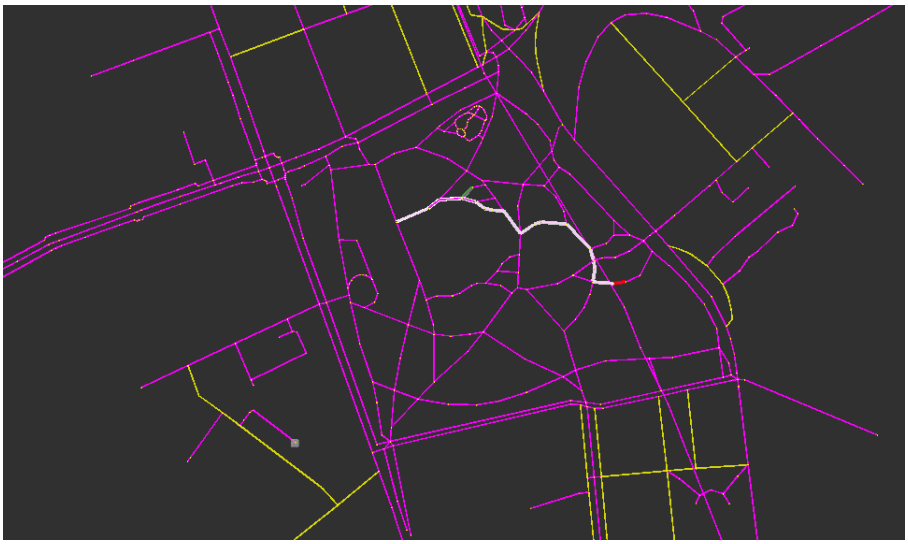
  Výsledkem je úspěšná operace a výstup v *RViz* (viz obrázek 4.6).

Podobně lze tyto úlohy zopakovat s plánovačem *UCS* a *Greedy Search*, spuštěním `launch-ucs.sh` nebo `launch-greedy.sh`.

Při použití s těmito plánovači si lze všimnout rozdílných počtů rozgenerovaných nodů v plánovači, zatímco *UCS* jich má často řádově více než *A**, *Greedy Search* jich má naopak méně, ale zase na druhou stranu nenachází optimální cestu.



Obrázek 4.5: Nejkratší nalezená vzdálenost mezi čtyřmi body definovanými ID Nodů z OSM.



Obrázek 4.6: Kombinace všech parametrů, trasa se již překrývá.

Kapitola 5

Závěr

Podařilo se implementovat knihovnu obohacující funkcemi již existující knihovny. V práci jsou popsány koncepty *ROS* na úrovni nutné k úspěšné implementaci knihoven, které pracují s mapami a plánovacími algoritmy. Potvrdilo se, že v *ROS* nezáleží na implementačním jazyku knihovny, důležité je pouze rozhraní služeb a zpráv, které je stejné.

Byly vyhodnoceny vlastnosti podobných knihoven zabývajících se *OSM* a zejména jejich nedostatky.

Byl implementován plánovač cest, snadno rozšiřitelný o další plánovací algoritmy. K němu bylo implementováno několik nejběžnějších plánovacích algoritmů – *A**, *UCS* a *Greedy Search*. Plánovač také podporuje možnost specifikace dalších bodů, které jsou v cestě významné.

Dále bylo implementováno rozhraní služeb, které ulehčuje použití plánovacích algoritmů.

Nakonec byl implementován vizualizační nástroj pro naplánované trasy na bázi *RViz*, který používají i související knihovny.

Toto vše bylo vyzkoušeno na mapě cest v parku.

Do budoucna se nabízí provázání této knihovny s jinými knihovnami zabývajícími se navigací v prostoru. Pro roboty pohybující se na velkých plochách se plánování v mapě nepochybně hodí. Nebo naopak může nějaký robot automaticky pomocí svých senzorů vytvářet nebo vylepšovat *OSM*, nejprve bude data ukládat do struktur používaných zde, poté se tato data opačným procesem převedou do *OSM*.

Literatura

[1] doc. Ing. František Zbořil, P., CSc. a Ing. František Zbořil: Základy umělé inteligence. studijní opora, 2012 [cit. 2014-04-15].

[2] Kopečný, T.: Pravděpodobnostní lokalizace mobilního robota. bakalářská práce, 2012 [cit. 2014-04-17].

[3] WWW stránky: About OpenStreetMap Blog [online]. `http://blog.osmfoundation.org/about/`, [cit. 2014-02-01].

[4] WWW stránky: Autoská práva a licence [online]. `http://www.openstreetmap.org/copyright`, [cit. 2014-02-01].

[5] WWW stránky: Downloading data [online]. `http://wiki.openstreetmap.org/wiki/Downloading_data`, [cit. 2014-02-01].

[6] WWW stránky: navfn [online]. `http://wiki.ros.org/navfn`, [cit. 2014-02-01].

[7] WWW stránky: roscore [online]. `http://wiki.ros.org/roscore`, [cit. 2014-02-01].

[8] WWW stránky: route_network.planner - route_network 0.2 documentation [online]. `http://docs.ros.org/hydro/api/route_network/html/python/_modules/route_network/planner.html`, [cit. 2014-02-01].

[9] WWW stránky: ROS.org — History [online]. `http://www.ros.org/history/`, [cit. 2014-02-05].

[10] WWW stránky: RFC 4122 [online]. `http://tools.ietf.org/html/rfc4122.html`, [cit. 2014-04-05].

[11] WWW stránky: Extensible Markup Language [online]. `http://www.w3.org/XML/`, [cit. 2014-04-12].

Dodatek A

Obsah CD

Na CD jsou umístěny zdrojové texty implementované knihovny a tato práce včetně zdrojových textů.