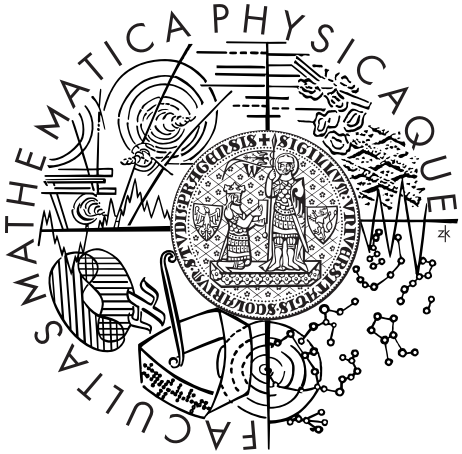
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Tomáš Hurt

Zemědělský robot R4Farmer

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.
Studijní program: Informatika
Studijní obor: Obecná informatika

Praha 2012

Rád bych zde poděkoval vedoucímu své bakalářské práce RNDr. Martinu Pergelovi, Ph.D. za cenné rady, připomínky a trpělivost při vedení mé práce.

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

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

V ..... dne ..... Podpis autora

Název práce: Zemědělský robot R4Farmer

Autor: Tomáš Hurt

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D., Kabinet software a výuky informatiky

Abstrakt: Práce má za cíl nejprve popsat navrhnutý systém autonomního robotického zemědělce R4Farmer a následně se zaměřit na specifickou softwarovou část projektu týkající se plánování trasy robota při zpracovávání pole. Bylo třeba vytvořit dostatečně kvalitní nástroje pro modelaci pole a nastavení parametrů programu robota a dát k dispozici simulační a vizualizační nástroje pro budoucí ladění systému bez rizika hospodářských škod. Významnou část práce tvoří návrh plánovacích algoritmů, které budou určovat samotný program robota na poli. Celý projekt má sloužit jako studie realizovatelnosti a funkčnosti takového systému, a to jak teoreticky, tak i při reálném nasazení v terénu.

Klíčová slova: robot, zemědělství, pole, automatizace

Title: Farming robot R4Farmer

Author: Tomáš Hurt

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D., Department of Software and Computer Science Education

Abstract: The aim of the thesis is to propose a robotic farmer emulator. After designing this toolkit, we focus on algorithms solving particular problems related to the use of such a robot. Mainly we attach the question of a route planning during the field processing (while ploughing, seeding and irrigating). The project should help with a design of a real robot which is being designed separately by other colleagues. It should help with the study of feasibility and eliminate particular risks of economic loss (caused by move of a real robot on a field - or outside of it).

Keywords: robot, farming, agriculture, field, automatization

Obsah

Úvod

Historie zemědělství je stará jako lidstvo samo. Její kořeny sahají až do doby neolitu, kdy se tehdejší společnost sběračů a lovců rozhodla přejít k usedlému způsobu života založenému na zemědělské výrobě, tedy na cíleném pěstování rostlin a chovu zvířat. Samotné zemědělství zůstalo dominantním prvkem hospodářství států až do příchodu průmyslové revoluce v 18. a 19. století. Lidskou sílu a dobytek postupně nahradily mechanizované stroje, které zefektivnily celý proces získávání obživy a v důsledku silně podpořily vývoj lidské společnosti jako takové.

Potřeba zajištění obživy pro obyvatelstvo však nikterak neklesá ani v dnešní době, právě naopak.

Ikonou dnešního zemědělství je traktor s pluhem. Budoucnost však dost možná přinese mnohem **autonomnější**, **efektivnější** a **ekologičtější stroje** – roboty, kteří veškerou zemědělskou práci budou schopni vykonávat samostatně, bez zásahu člověka.

O nalezení efektivní implementace takového systému se pokusíme v následující práci. Problém je přirozeně velmi komplexní a skýtá spoustu záludností. Projekt, který si představíme, nazvaný **R4Farmer**, nemá sloužit jako univerzální zemědělský stroj do všech podmínek. Do jakých podmínek bude určen, jaké budou jeho cíle, možnosti, či jaké budou výhody oproti traktoru, si povíme hned v úvodní kapitole této práce.

Je nutné zdůraznit, že tato bakalářská práce se bude zabývat pouze **vymezenou softwarovou částí projektu**, konkrétně určením programu robota (tj. co má na poli dělat a jak se chovat), jinou část softwaru a hardwarovou část projektu má na starost kolega Ing. Aleš Blažek v rámci své doktorské práce na ZČU v Plzni (podrobnosti opět v úvodní kapitole této práce).

Celý projekt má sloužit jako **studie realizovatelnosti a funkčnosti** takového systému, a to jak teoreticky, tak i při reálném nasazení v terénu. V první fázi bude jistě nezbytné vytvořit **emulační prostředí**, které bude sloužit pro účely vývoje a ladění systému bez rizika hospodářských škod.

Jak již bylo zmíněno, podrobnostem projektu, přesné **specifikaci** jeho možností, způsobu práce robota, do jakých podmínek bude určen a stanovení jeho cílů a možných problémů, bude věnována úvodní kapitola 1.

Kapitola 2 bude blíže pojednávat již o samotné **implementaci** softwarové části a řešení problémů s tím spojených a v kapitole 3 se pak dostaneme k samotnému plánování práce robota na poli, z teoretického i implementačního hlediska. Navrhneme několik **algoritmických přístupů** a v závěru přímo porovnáme jejich výkonnost na různých typech vstupních dat.

Kapitola 4 je pak stavěna jako podrobná uživatelská příručka a také jako programová dokumentace.

V závěru práce zhodnotíme dosažené výsledky a zmíníme potenciální možnosti rozšiřitelnosti softwarové části projektu.

0.1 Teoretický základ

Vzhledem k tomu, že se v této práci budeme pohybovat převážně v informatických vodách, jistě nebude od věci si před uvedením do problému krátce zavést několik obecných informatických pojmů a postupů, které v dalším textu budeme využívat.

Algoritmy a jejich složitost

Algoritmus je návod či postup pro vyřešení daného typu úlohy. Pojem algoritmu se nejčastěji objevuje při programování, kdy se jím myslí teoretický princip řešení problému. Běžným kritériem pro hodnocení kvality algoritmu je vyjma jednoduchosti, efektivity a elegance také určení časové a prostorové složitosti. [1]

Časová a prostorová složitost algoritmu určují závislost času výpočtu na velikosti vstupu, resp. závislost paměťových nároků na velikosti vstupu. Půjde nám o náročnost algoritmu při nejméně příznivém vstupu velikosti n, což můžeme zapsat jako funkci $f(n)$.

Přesné určení funkce $f(n)$ může být velmi složité, proto se při hledání funkce zaměřujeme na dominantní složky funkce, které pro velká n reprezentují podstatnou část času výpočtu a funkci $f(n)$ nahradíme jednodušší funkcí $g(n)$, která bude první funkci **asymptoticky ohraničovat**. Přitom se použije tzv. **O-notace** nebo **Θ-notace** [1].

Nechť $f(x)$ a $g(x)$ jsou funkce definované na nějaké podmnožině reálných čísel. Potom

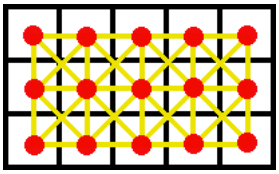
$$f(x) \in O(g(x)), \text{ jestliže } \exists(C > 0), x_o : \forall(x > x_o)|f(x)| \leq |Cg(x)| \text{ a}$$
$$f(x) \in \Theta(g(x)), \text{ jestliže } \exists(C, C' > 0), x_o : \forall(x > x_o)|Cg(x)| < |f(x)| < |C'g(x)|.$$

Grafy

(Neorientovaný) graf je dvojice $G = \langle V, E\rangle$, kde V je neprázdná množina **vrcholů** (někdy také uzlů) a $E \subseteq \{\{u, v\}|u, v{\in}V, u{\neq}v\}$ je množina dvouprvkových množin vrcholů, (neorientovaných) **hran**.

Orientovaný graf je dvojice $G = \langle V, E'\rangle$, kde V je neprázdná množina vrcholů a $E' \subseteq \{(u, v)|u, v{\in}V, u{\neq}v\}$ značí množinu uspořádaných dvojic vrcholů, tj. orientovaných hran. Pokud $(u, v){\in}E$, pak řekneme, že v grafu existuje hrana z vrcholu u do vrcholu v.

Graf je **souvislý**, pokud pro každé dva jeho vrcholy $x, y{\in}V$ v něm existuje cesta z x do y. **Kružnice v grafu** je posloupnost $(v_0, e_1, v_1, e_2, \ldots, e_{t-1}, v_{t-1}, e_t, v_0)$, kde $v_0, v_1, \ldots, v_{t-1}$ jsou navzájem různé vrcholy grafu G, a $e_i = \{v_{i-1}, v_i\} \in E$ pro $i = 1, 2, \ldots, t-1$, a také $e_t = \{v_{t-1}, v_0\} \in E$. [5]



Obrázek 1: Čtvercová mřížka

V dalším textu budeme často užívat čtvercovou mřížku, což bude graf s vrcholy odpovídající každému ze čtverců této mřížky, přičemž každé dva vrcholy budou

mít mezi sebou hranu, jestliže budou v mřížce sousedy (včetně příčných směrů). Příklad čtvercové mřížky ilustruje obrázek 1.

Stromy

Široce užívanou hierarchickou datovou strukturou je **strom**. Formálně jde o souvislý orientovaný acyklický graf, kde každý z vrcholů má nula a více potomků a nejvýše jednoho rodiče. Nejvýše postavený vrchol (bez rodiče) se nazývá **kořen** stromu, naopak nejnižší vrcholy (bez potomků) se nazývají **listy**. Nejčastěji se setkáme s tzv. **binárními stromy**, kde každý uzel má nejvýše dva potomky. [8]

Prohledávání do šířky

Často budeme využívat algoritmu prohledávání grafu do šířky (*Breadth-First Search*, zkráceně BFS), označován jako tzv. **algoritmus vlny**, či jeho modifikací. Tento algoritmus nejprve prochází všechny sousedy startovního vrcholu, poté sousedy sousedů atd. až projde celou komponentu souvislosti. Časová složitost algoritmu je $O(|V| + |E|)$, kde V je množina vrcholů a E je množina hran grafu. Paměťová složitost algoritmu odpovídá $O(|V|)$. [6]

Uveďme si jeho pseudokód:

```
1) vytvoř frontu Q, vlož do Q počáteční vrchol v, označkuj vrchol v
2) dokud není Q prázdná:
3)    m ← vezmi prvního z fronty Q
4)    je-li m hledaným vrcholem:
5)      vrať m
6)    pro všechny hrany e incidentní s vrcholem m:
7)      f ← opačný vrchol hrany e
8)      pokud není f označkován:
9)        označkuj f, vlož f do fronty Q
```

Prohledávání do hloubky

Prohledávání do hloubky (anglicky *Depth-First Search*, zkráceně DFS) je grafový algoritmus pro procházení grafů metodou **backtrackingu**. Pracuje tak, že od aktuálně zpracovávaného vrcholu získá prvního následníka, kterého ještě nenavštívil, stejným způsobem zpracuje tento vrchol atd. Pokud narazí na vrchol, z nějž už nelze dále pokračovat (nemá žádné následníky nebo byli všichni navštíveni), vrací se zpět backtrackingem. Časová a paměťová složitost algoritmu je stejná jako u prohledávání do šířky, tj. $O(|V| + |E|)$, kde V je množina vrcholů a E je množina hran grafu. Paměťová složitost odpovídá $O(|V|)$. [6]

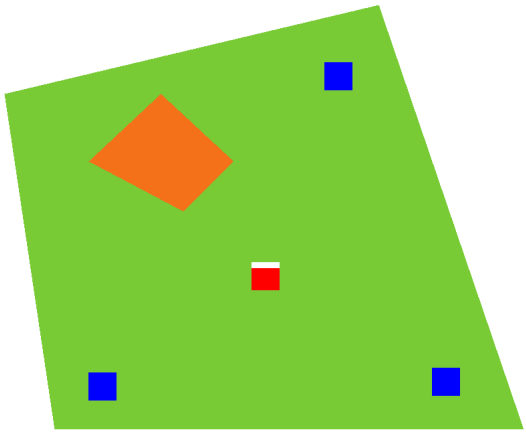
Pseudokód je stejný jako při prohledávání do šířky, pouze místo fronty používáme zásobník a dochází tak k jinému pořadí zpracovávání vrcholů. Často je však algoritmus implementován rekurzivně:

```
1) označ všechny vrcholy jako nenavštívené
2) DFS(v):
3)    označ v jako navštívený
4)    zpracuj v
5)    pro všechny vrcholy w sousedící s v:
6)      pokud w je nenavštívený, pak DFS(w)
```

0.2 Barevné značení

V průběhu této práce budeme na obrázcích ilustrovat jednotlivé postupy. Často se budou týkat zpracovávání samotného pole, a proto bude vhodné si předem osvětlit **význam barev jednotlivých elementů pole**. Analogicky se toto barevné rozlišování bude dodržovat i přímo v programu, kde je však navíc po celou dobu uvedena legenda.

Typické pole ukazuje obrázek 2. Základ pole je vymezen elementy se světle zelenou barvou, oranžové elementy pak značí překážku v poli a modré elementy budou značit tzv. body energetické obsluhy (viz dále). Robot v poli bude mít vždy červenou barvu, přičemž jeho aktuální otočení je indikováno bílým pruhem (předek robota).



Obrázek 2: Pole

1. R4Farmer – specifikace

V této kapitole se krátce seznámíme s celkovým konceptem projektu R4Farmer a přesně stanovíme možnosti a cíle projektu.

1.1 Projekt R4Farmer

Cílem projektu R4Farmer je vývoj robotického zemědělce pro rytí, zavlažování a osévání pole. Projekt je koncipován jako **studie a následně realizace prototypu** robota, který by v budoucnu mohl nahradit zemědělskou činnost týkající se setí obilovin pro potravinový průmysl a užitných rostlin pro biopaliva.

Zamýšlené nasazení projektu R4Farmer jsou pole Afriky. Cílem je využít dostatek solární energie v této oblasti, **minimalizovat náklady** a vybudovat stroj, který bude **autonomně** schopný pracovat v **extrémních podmínkách**.

Projekt je rozdělen do tři fází:

R4Farmer 1/S – vývoj software pro vývojový a testovací kit,
R4Farmer 1/H – vývoj a realizace HW robota,
R4Farmer 1 – spojení software a hardware do konečné podoby.

Tato **bakalářská práce** se bude věnovat vymezené části R4Farmer 1/S, která bude mít na starosti **plánování práce** robota na poli, tj. bude sloužit jako **master program** určující pohyb v terénu pole. Zbytek projektu zahrnující subsystém představující jádro řízení robota (řešení navigace a plánování trasy ve 3D terénu na základě požadavků master programu) i realizaci hardwarové části robota bude mít na starosti kolega Ing. Aleš Blažek, v rámci své doktorské práce na ZČU v Plzni. Součástí hardware budou navigační rutiny, které zajistí přísun informací pro výše uvedený master program. V první fázi se počítá se zkonstruováním zmenšeného modelu robota velikosti cca. 1,2 × 0,8 m. Plánovaný termín dokončení první fáze projektu je v průběhu roku 2013.

1.2 Specifikace projektu – možnosti a cíle

Jak již bylo zmíněno, projekt by měl být nasazen na polích Afriky. Pro pohon a své fungování bude užívat **elektřinu** – pro tento účel budou v polích využívány tzv. **body energetické obsluhy**, které budou sloužit pro dobíjení akumulátorů. Právě na těchto bodech bude možno využít solární energii pro dobíjení.

Kapitolou samou pro sebe jsou africké extrémní podmínky, co se týká teplot. Robot musí mít dostatečně **odolnou konstrukci** pro zvládání vysokých a nízkých teplot. Na druhou stranu jde o největší výhodu robota oproti použití klasického traktoru v těchto destinacích: obsluha traktoru by byla velmi zatěžována, bylo by nutná instalace klimatizace, sluneční clony apod. Robot pochopitelně nebude vyžadovat přímou obsluhu. Další výhodou oproti traktoru by měly být **nižší náklady na provoz**. Elektrické motory sice jsou slabší a pomalejší, avšak jsou tišší a mohou tak pracovat i v noci. Robot sice bude dělat pomaleji než běžné traktory, nicméně o to **vytrvaleji**. Díky nižší síle a rychlosti, lze použít menší

pevnost konstrukce než traktor, slabší výkon je pak dotažen skrze vytrvalost. Odhad je takový, že nahradit **jeden traktor** bude znamenat nasadit zhruba **dva roboty**.

Robot jako takový nebude potřebovat další přídavné nástroje, které by za sebou táhl. Pracovní nástroje zajišťující rytí, setí a zavlažování budou přímo na spodku robota.

Co se týká výpočetní kapacity robota, v tuto chvíli se počítá s integrovaným počítačem o výkonnosti běžné stolní sestavy.

Následuje výčet klíčových prvků plánované implementace.

1.2.1 Pole

Prvotním úkolem programu bude dát uživateli možnosti a dostatečně silné nástroje pro vytvoření půdorysu pole. Pole může mít libovolný tvar a rozměry, od velikostí v řádu několika metrů po několik kilometrů. Rozumí se, že budou k dispozici nástroje pro **dostatečně přesné modelování** daných polí. Pole jako taková se bude dělit maximálně do čtverečků rozměrů 5×5 cm (zvolená maximální přesnost). Do polí bude umožněno **umístit překážky** libovolných rozměrů.

1.2.2 Body energetické obsluhy

Zásadním omezením pro náš projekt bude fakt, že robot nebude mít dostatek energie pro vykonání celého programu, ale pouze pro určitou část. Robot se bude muset včas dostavit na místo označené jako bod energetické obsluhy, kde dojde k **doplnění energie** akumulátorů. Tyto body budou zároveň sloužit pro **doplnění osiva a vody** – to v případě, kdy bude robot obstarávat setí a zavlažování pole.

Body energetické obsluhy budou mít omezené kapacity osiva i vody, mohou mít nastavitelné rozměry a mohou být rozmístěny libovolně po poli, dle přání uživatele.

1.2.3 Robot

Robot bude mít určenou startovní pozici v poli, nastavitelné rozměry – včetně rozměrů a polohy jednotlivých nástrojů na spodní části robota (ty budou horizontálně vycentrované), kapacitu nádrže pro vodu a osivo a také **energetickou náročnost** jednotlivých operací. Takovými **operacemi** se myslí elementární pohyb robota o 5 cm, otočení o 45 a 90 stupňů (to probíhá na místě stylem *tanku*), dále energetická náročnost se zapnutým nástrojem (tj. např. rycí nástroj) a rovněž je třeba počítat s tím, že spotřeba bude (lineárně) záviset na aktuálním množství osiva/vody v nádrži robota.

Naprosto zásadním úkolem programu bude zajistit, aby se robot při svých toulkách přírodou nedostal do situace, kdy mu **dojde veškerá energie** uprostřed pole, a on tak nebude schopný dojet na bod energetické obsluhy pro doplnění energie. Důraz je kladen na to, aby veškerá práce, kterou robot provede, byla co nejefektivnější, tj. po celou dobu budeme chtít **minimalizovat spotřebovanou energii**.

Uživatel před odstartováním robota zvolí požadované operace (rytí, setí či zavlažování).

1.2.4 Neočekávané překážky

Uživatel sám může určit překážky v poli, libovolných tvarů a rozměrů, při modelování pole. V terénu se ovšem mohou vyskytovat rovněž neočekávané překážky (kupříkladu stožár elektrického napětí; v zásadě však může jít opět o překážku libovolného tvaru i rozměrů). Případné neočekávané překážky oznámí sám robot, který má nainstalované **detekční senzory**. Robot se musí s detekovanou překážkou náležitě vypořádat a pamatovat si její pozici do budoucna.

1.2.5 Simulační nástroje

Pro účely simulace a ladění robota bude nutné vytvořit **emulátor práce robota**. V tomto simulačním prostředí bude možno testovat práci robota, bez rizika jakýchkoli hospodářských škod, před skutečným nasazením v terénu.

1.2.6 Cíl programu robota

Cílem je po vymodelování pole a nastavení všech parametrů nechat robota **zpracovat dané pole**. Zadaná práce musí být provedena co **nejefektivněji** vzhledem k energetické spotřebě a s důrazem na to, že program jako takový **neselže** – tzn. robot se *nezapomene* včas dobít a nenechá se zaskočit neočekávanými překážkami v poli.

2. R4Farmer – implementace

V této kapitole se zaměříme na samotnou implementaci vymezené softwarové části projektu R4Farmer, dle specifikace v kapitole 1. Pozornost bude věnována spíše programátorskému hledisku a řešení problémů při implementaci. Detailní seznámení s programem z ryze uživatelského hlediska i detailní programovou dokumentaci nabídne kapitola 4.

V prvé řadě bude cílem vytvořit program s jednoduchou obsluhou pro uživatele, kterému mu dá možnost vymodelovat pole a následně sledovat proces zpracovávání daného pole robotem. Algoritmům zaměřeným na plánování trasy robota při obdělávání pole se bude věnovat kapitola 3.

2.1 Programovací jazyk, knihovny

Program je implementován jako standardní „okenní“ aplikace za použití programovacího jazyka **C++** a multi-platformních freeware open-source knihoven **wxWidgets**[1]. Dané GUI bylo zvoleno s cílem dát uživateli k dispozici dostatečně pohodlné prostředí pro modelování pole a vizualizaci práce robota. Jazyk C++ byl upřednostněn před alternativami (Java, C#) zejména z důvodu rychlostního a z důvodu potenciálního fungování na odlišné platformě (unixové systémy).

Primární platforma je v tuto chvíli MS Windows, přičemž zdrojové kódy jsou psány v prostředí MS Visual Studia 2008. Základní portabilita pro unixové systémy sice byla zajištěna – program je zkompilovatelný a spustitelný – nicméně aplikace zde nebyla dosud řádně odladěna.

Co se týká dalších knihoven, pro složitější operace nad polygony je použita externí freeware knihovna *Clipper* od autora Anguse Johnsona[2] a pro generování Voroného diagramu volně šiřitelná C++ implementace Shana O'Sullivana[3] původního algoritmu navrženého Stevenem Fortunem[4].

2.2 Konfigurační modul

Program je rozdělen na dva hlavní moduly – **Konfigurační** a **Pracovní modul**. Důvodem této modularizace je snaha o maximální oddělení dvou logicky odlišných celků programu. Prvně jmenovaný modul zajišťuje **modelování pole** a nastavení všech parametrů projektu, zatímco druhý slouží pro samotné **plánování práce robota** a rovněž jako **emulátor práce robota**, včetně uživatelské **vizualizace**.

2.2.1 Modelování pole

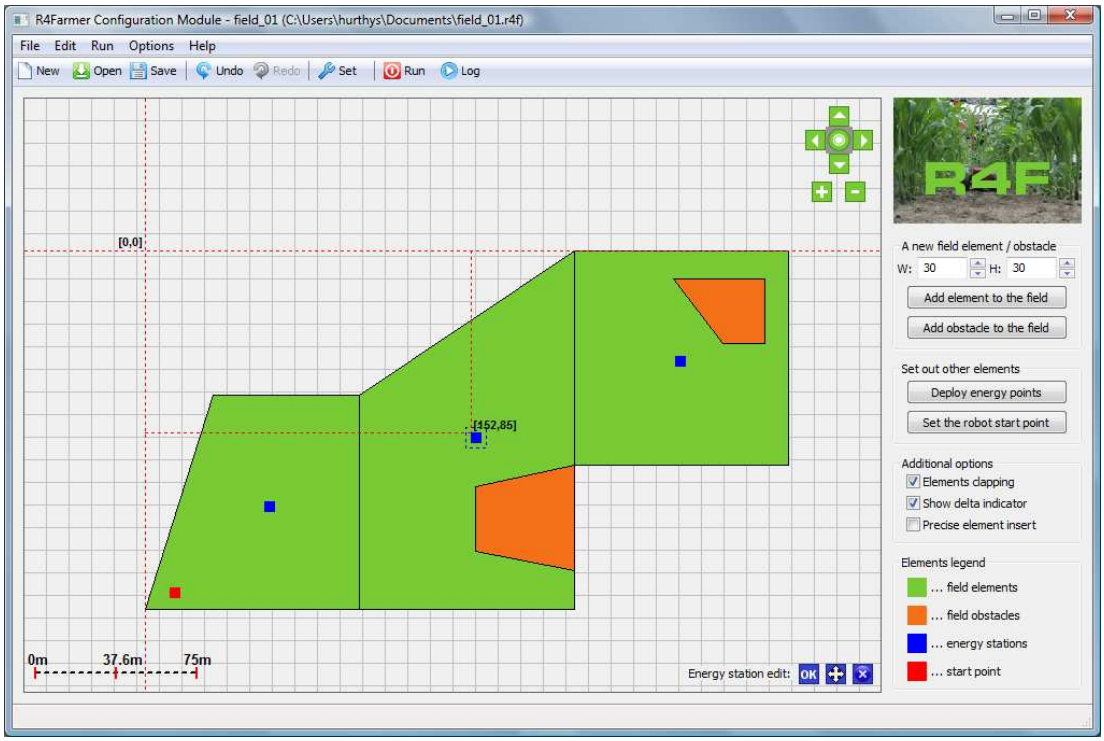
Modelovací plocha nabízí jednoduchou navigaci a zoomování, **pravý editovací panel** pak vložení jednotlivých elementů pole, překážek, nastavení startovní pozice robota a nastavení bodů energetické obsluhy (viz obrázek 3). Každý

[1] `http://www.wxwidgets.org/`
[2] `http://www.angusj.com/delphi/clipper.php`
[3] `http://www.skynet.ie/~sos/mapviewer/voronoi.php`
[4] `http://ect.bell-labs.com/who/sjf/`

z přidaných prvků lze dále editovat a posouvat. **Elementy pole** (dále jako field elementy) i **překážkové elementy** (dále jako baulk elementy) se přidávají do modelovací plochy pouze jako obdélníky zadaných rozměrů, nicméně lze je následně editovat táhnutím za roh příslušného elementu do v zásadě libovolného tvaru (bez křížení hran). Svislé a vodorovné hrany jednotlivých polygonů na sebe lze jednoduše *přilepovat*, při dostatečném přiblížení se k sobě *doklapnou*. Dostatečnou přesnost modelování zajišťuje ukazatel relativní pozice vůči souřadnicovému počátku.

Field elementy i baulk elementy jsou v celém průběhu programu reprezentovány shodně jako vektorová data – přesněji jako obecné čtyřúhelníky, které se dají k sobě napojovat a skládat přes sebe. **Výsledné pole** pak vznikne sjednocením všech field elementů a následným odečtením všech baulk elementů. Pro tyto operace nad polygony jsou použity externí knihovny *Clipper*.

Poznámka: Vymodelované pole bude rovina, což v reálu typicky nenastává. 3D navigace v terénu se však bude řešit až v řídícím subsystému (není součástí této práce). Nicméně plánovací program by například měl vědět o zvýšené spotřebě při jízdě do kopce. Diference mezi reálnou spotřebou a plánovanou teoretickou bude korigována v rámci měření kapacity akumulátorů robota. Robot pošle zpětnou vazbu plánovacímu programu a dochází k případnému přepočítání trasy robota. Význam barev jednotlivých elementů uvádí legenda v pravém panelu.



Obrázek 3: Modelování pole v Konfiguračním modulu

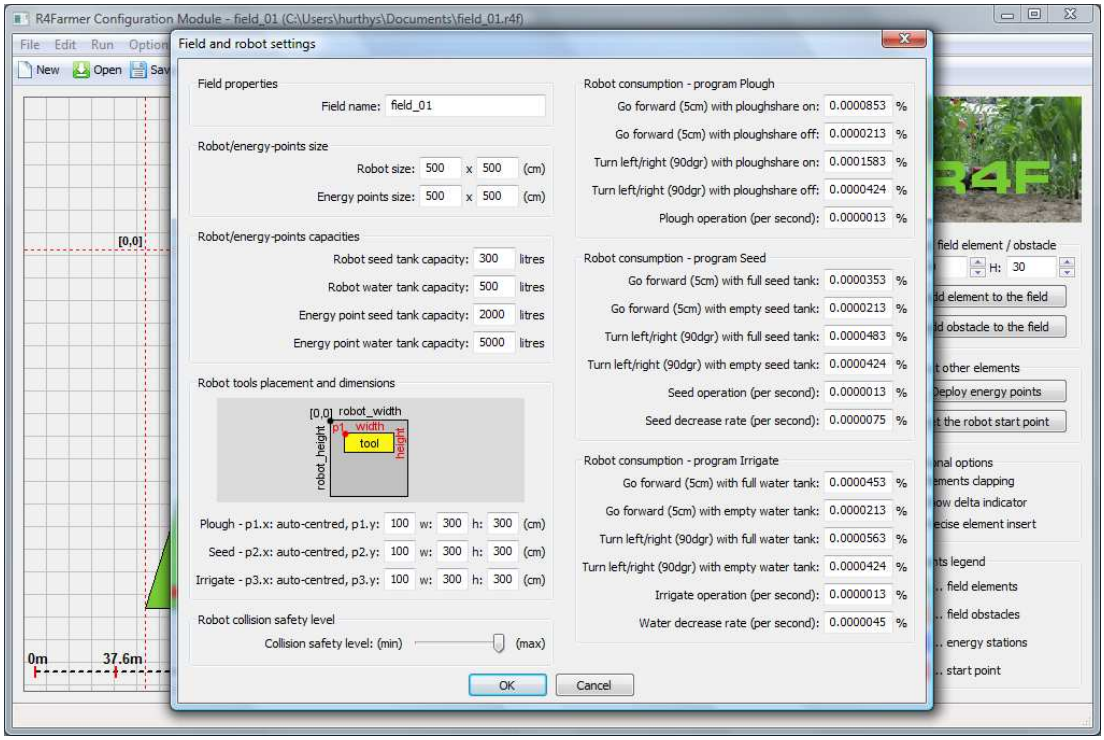
2.2.2 Parametry projektu

Další důležitou částí Konfiguračního modulu je nastavení **všech parametrů projektu**, od rozměrů robota po energetickou náročnost jednotlivých operací

(obr. 4). Před odstartováním robota (spuštěním Pracovního modulu) je **kontrolováno nastavení** všech nezbytných vlastností projektu, tj. nastavení startovního bodu robota, alespoň jednoho bodu energetické obsluhy a také kompaktnost pole, tzn. nejsou dovoleny izolované field elementy.

Uživatel navolí **požadované operace** (rytí/setí/zavlažování), zvolí **algoritmus** a v případě simulace může nastavit rovněž **parametry emulovaného robota**, tj. rychlost provádění jednotlivých operací a polohy neočekávaných překážek v poli (v případě reálného nasazení se tyto parametry pochopitelně nenastavují, tam neočekávané překážky znamenají skutečnou detekci nějaké překážky pomocí senzorů robota).

Nastavit lze také zaznamenávání **logu** činnosti robota. Celý průběh práce se pak dá jednoduše zrychleně přehrát pomocí dostupného **přehrávače logu** (de facto pouze speciální režim Pracovního modulu, viz dále).



Obrázek 4: Konfigurační modul – nastavení parametrů pole a robota

2.3 Pracovní modul

Druhým modulem programu je Pracovní modul, který provádí nejzásadnější úkony, a sice samotné **plánování práce robota**. Rovněž slouží pro účely **emulování práce robota** a poskytuje **vizualizaci** doposud vykonané práce. Takováto vizualizace je nezbytná pro další vývoj a ladění akcí robota.

Robot je odstartován stisknutím tlačítka *Launch* na pravém panelu Pracovního modulu. Přerušit celou operaci lze prostým zavřením okna Pracovního modulu – robot se vypne a všechna pracovní vlákna se korektně ukončí. V momentě, kdy robot **úspěšně dokončí** zvolený program, vydá zprávu o ukončení práce, spolu se souhrnem celkem spotřebované energie, osiva a vody a uplynulým čas od za-

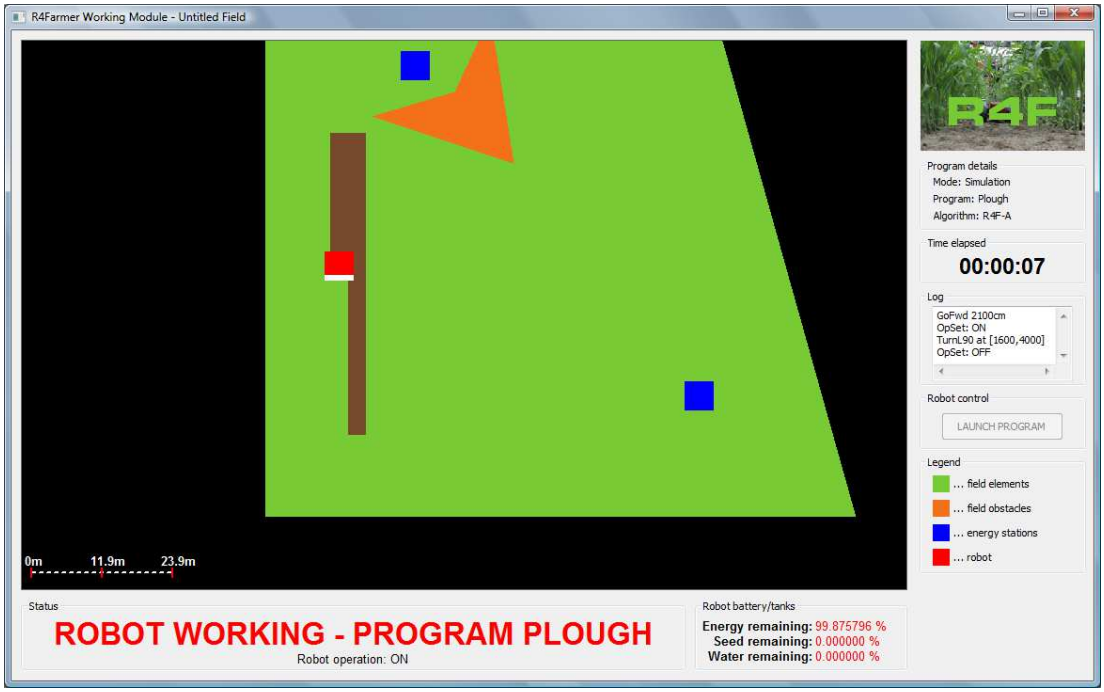
počnutí operace po její úspěšné dokončení. V případě **selhání** programu robota se vydá oznamovací zpráva s důvodem tohoto selhání.

2.3.1 Vizualizace práce robota

Standardní obrazovka pracovního modulu dovoluje uživateli sledovat **průběh práce** robota, tzn. jeho pohyb po poli a postupné zpracovávání pole vymodelovaného v Konfiguračním modulu. Podobně jako v předchozím modulu, i zde je v GUI implementována **jednoduchá navigace** – uživatel si myší *přitáhne* libovolnou část pole, kterou chce sledovat, a přiblíží si (či oddálí) pohled libovolně kolečkem myši.

V dolním panelu má uživatel k dispozici aktuální informace o **zbývající kapacitě energie** robota i **zbývajících komoditách** (osivo/voda) v nádržích robota. Stejně tak lze při dostatečném přiblížení na bod energetické obsluhy zjistit zbývající komodity v jeho zásobníku.

Pravý panel pak nabízí uživateli detaily o **nastaveném programu** robota (např. setí + zavlažování), zvoleném algoritmu, uplynulém času práce robota a stručný **log** poslední činnosti robota.



Obrázek 5: Pracovní modul – práce robota na poli

Pokud jde o samotné vykreslování pracovní plochy, souřadnice všech vektorových dat (elementy pole, překážky, body energetické obsluhy, poloha robota, ...) jsou přepočítávány dle aktuální hodnoty zoomu. Přesnost těchto souřadnic je v řádech centimetrů – ačkoli v Konfiguračním modulu je při modelování pole nejmenší jednotkou 1 metr, souřadnice jsou v Pracovním modulu přepočítány na centimetry pro dosažení maximální přesnosti při provádění geometrických transformací robota, tj. zejména otáčení (prováděné na místě kolem svého středu stylem *tanku*).

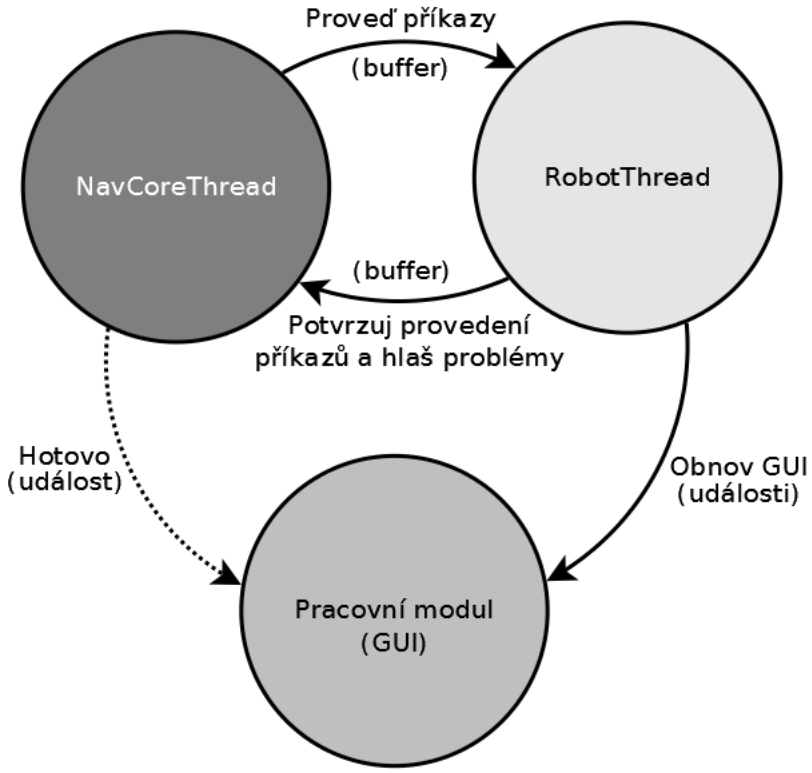
Pokud robot projíždí pole se zapnutým (např. rycím) nástrojem, a tedy zpracovává pole, zůstává za ním, resp. za daným nástrojem na spodku robota, barevná čára značící **zpracované úseky** pole. Pracovní modul si během toho, kdy je spuštěn příslušný nástroj, zaznamenává počáteční a koncové souřadnice každé čáry a následně je periodicky vykresluje, stejně jako zbylé elementy pracovní plochy. Hnědá čára značí zryté pole, žlutá oseté pole a modrá čára zavlažené pole. Zcela nezpracované části pole mají pak standardně zelenou barvu. Jak takové zpracovávání pole může vypadat, ukazuje obrázek 5.

Význam barev ostatních elementů opět uvádí legenda v pravém panelu dole.

2.3.2 Pracovní vlákna a jejich komunikace

Hlavní vlákno programu obsluhuje grafický interface (GUI). Pro průběžný dopředný výpočet pracovního plánu robota při zachování efektivního vykreslování na pracovní plochu v reálném čase bude nutné vytvořit **nové pracovní vlákno**, to nazveme `NavCoreThread`.

Dále bude vhodné oddělit část programu zajišťující plánování budoucích kroků robota od těch částí, které dané příkazy provádějí. To nám umožní plánovat dané kroky nezávisle na tom, zda je následně zpracuje emulovaný či reálně nasazený robot. **Plánování** tedy bude zajišťovat již zmíněný `NavCoreThread`, zatímco **entitu robota** bude reprezentovat vlákno `RobotThread`. Komunikaci mezi všemi třemi vlákny ilustruje diagram na obrázku 6.



Obrázek 6: Komunikace mezi vlákny

Prvně jmenované vlákno, `NavCoreThread`, je v zásadě nejdůležitější částí softwaru projektu R4Farmer – má k dispozici kompletní informace o aktuální konfiguraci robota i pole a právě zde bude probíhat **plánování trasy** robota na základě

několika různých algoritmů (dle volby uživatele). Těmito algoritmy se bude obšírně zabývat kapitola 3. `NavCoreThread` tedy rozhoduje, co se bude dít, a **generuje sekvenci příkazů** pro robota, tzn. posílá je na příchozí buffer `RobotThreadu`.

Vlákno `RobotThread` reprezentuje robota jako takového (emulovaného i reálně nasazeného – více o emulaci v sekci 2.3.3). Jsou mu posílány příkazy od `NavCoreThreadu` (typicky: jeď dopředu o 1 m, zatoč doprava o 45° apod.), které následně **vykonává** a posílá zpět **potvrzení**. Ve stejnou dobu pošle i **echo hlavnímu vláknu** (GUI), aby aktualizovalo situaci na plánku pole na Pracovním modulu (změna pozice robota, úbytek energie apod.). Původně tuto jednostrannou komunikaci s GUI zajišťoval `NavCoreThread` poté, co dostal potvrzení o provedení operace od robota, nicméně hrozilo zde, že pokud by toto vlákno provádělo v době přijmutí potvrzení od robota nějaké rozsáhlé výpočty, nemohlo by včas aktualizovat GUI a to by mohlo z pohledu uživatele na chvíli *zamrznout*. Takto má uživatel k dispozici vždy informace o skutečně aktuální konfiguraci robota.

Robot se tedy pokouší přijmuté příkazy postupně splňovat, a pokud se příkaz podaří splnit, pošle potvrzení na příchozí buffer `NavCoreThreadu`. Může ovšem dojít k situaci, kdy se robotovi **nepodaří provést příkaz**, protože narazil na neočekávanou překážku v poli. V takovém případě toto oznámí `NavCoreThreadu` spolu s dalšími informacemi o detekovaném objektu. Přitom ignoruje další příkazy na příchozím bufferu, dokud mu `NavCoreThread` nepošle zprávu o **zotavení** (mezitím všechny příchozí příkazy jednoduše přeposílá zpět). Obě komunikující vlákna jsou *dohodnutá*, že takovéto oznámení překážky automaticky znamená zastavení pohybu robota i případné vypnutí právě aktivního nástroje. `NavCoreThread` po přijmutí oznámení o neočekávané překážce *upraví svůj vnitřní stav* (co to konkrétně znamená, už záleží na implementaci daného algoritmu), aby byl schopen reagovat na změnu situace na poli. Musí změnit plánovací strategii, poslat `RobotThreadu` zprávu o zotavení a teprve pak může opět začít generovat nové příkazy pro robota. Neočekávaná překážka se po detekování stává pro `NavCoreThread` již očekávanou a počítá s ní při budoucích plánech. Samozřejmě je opět dáno echo vláknu GUI, aby tuto překážku vyznačilo na pracovním plánku pole.

Jakmile `NavCoreThread` **dokončí plánování** trasy robota, pošle zakončující příkaz FINISH a čeká až robot dokončí naplánovanou práci (pokud by během toho došlo k problému, tj. zejména detekce neočekávané překážky, plánování by se pochopitelně znovu spustilo, neboť dříve vygenerovaná trasa již nebude validní). V momentě, kdy `NavCoreThread` obdrží potvrzující zprávu o dokončení od robota, obě vlákna se ukončí, upozorní se GUI, pošle se mu **shrnující zpráva** o celkově spotřebované energii, osivu a vodě a ta se zobrazí uživateli.

Robot má definovanou **množinu validních příkazů**, tj. těch, které zpracovává (příkazy pohybu, zapnutí/vypnutí nástroje, přechod do jiného programu, apod. + několik signálních příkazů). Přijímá i jemu neznámé příkazy, které automaticky posílá zpět `NavCoreThreadu` – typicky se užije pro signální účely vlákna `NavCoreThread`.

Ke komunikaci mezi vlákny `NavCoreThread` a `RobotThread` jsou použity příchozí/odchozí **buffery** na obou stranách, implementovené jako dvě fronty (se zabezpečenou synchronizací). Komunikace `RobotThreadu` s GUI vláknem probíhá výhradně prostřednictvím zasílání zpráv (událostí).

Předpokládá se, že kompletní konfigurace robota (pozice v poli, prováděný program apod.), kterou má vlákno `NavCoreThread` k dispozici, odpovídá reálné

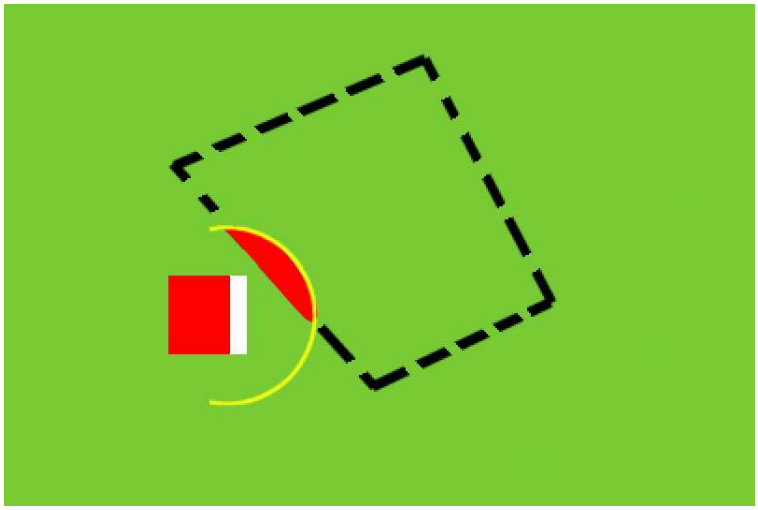
konfiguraci robota na poli. Řešení v praxi důležitých úkonů jako lokalizace (verifikace pozice robota na poli), nepřesnosti detekčních senzorů apod., nejsou součástí této práce.

2.3.3 Emulace robota

Komunikační schéma popsané výše se nebude nijak lišit, ať už robota reálně nasadíme v poli, či bude pouze emulovaný. I díky zvolené modularizaci vláken bude rozdíl pouze v **implementaci vlákna** `RobotThread`, popsaného v sekci 2.3.2.

V případě **emulovaného robota** se vlákno opět bude snažit splňovat přijmuté příkazy, ty však pochopitelně nebudou reálně provedené, ale vyčká se příslušná doba, po kterou by reálně nasazený robot tuto operaci prováděl. Rychlost těchto operací uživatel může definovat v Konfiguračním modulu při nastavování parametrů spouštěného programu, včetně předpokládané doby dobití energie a doplnění komodit v bodech energetické obsluhy.

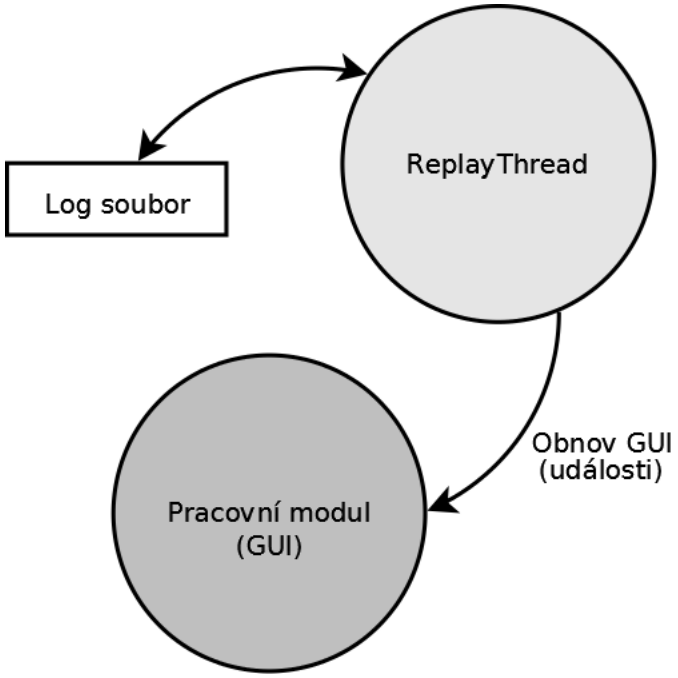
Detekce **neočekávaných překážek** při emulaci robota probíhá tak, že má `RobotThread` k dispozici údaje o své konfiguraci i údaje o neočekávaných překážkách v poli (tj. neočekávaných pro plánovací vlákno `NavCoreThread`). Udržuje si aktuální informace o své pozici, a pokud se k nějaké takové překážce přiblíží, oznámí detekci neočekávané překážky spolu s informací o pozici **nově detekované překážky**. Souřadnice této nové překážky se získají průnikem zvětšené obalové kružnice robota (v závislosti na dosahu senzorů robota) s daným polygonem reprezentujícím neočekávanou překážku v poli. Situaci ilustruje obrázek 7. Pozice neočekávaných překážek se dá opět definovat v Konfiguračním modulu.



Obrázek 7: Detekce neočekávané překážky při emulaci robota

2.3.4 Přehrávač logu

Jak již bylo řečeno, před vypuštěním robota je v Konfiguračním modulu k dispozici možnost ukládání **logu provedené činnosti**. Tento log se ukládá do souboru a lze ho následně přehrát pomocí přehrávače logu (v programu označen jako *log replayer*), který lze spustit přímo z Konfiguračního modulu.



Obrázek 8: Vlákna – přehrávač logu

Přehrávač logu je v principu pouze **odlišný režim** Pracovního modulu – opět sledujeme práci robota na poli, ubývající energii, komodity, uplynulý čas atd. Nicméně vše, co se nyní na obrazovce odehrává je *ze záznamu*, tj. program neprovádí žádné plánování, pouze čte dříve zaznamenanou práci z log souboru.

Pokud srovnáme fungování Pracovního modulu v tomto režimu se standardním fungováním – vlákna `NavCoreThread` a `RobotThread` nahradí jediné vlákno `ReplayThread`, které pouze čte log soubor a posílá události hlavnímu vláknu (GUI), které vše aktualizuje na obrazovku (viz diagram na obrázku 8).

Přehrávač logu dává navíc uživateli možnost nastavit **rychlost přehrávání** pomocí příslušného ovládacího prvku v levém horním rohu pracovní plochy a to i přímo při přehrávání. Přehrávač logu nám tedy dává možnost zrychleně zhlédnout potenciálně velmi dlouhý proces zpracovávání pole robotem a je tudíž vítaným **ladícím nástrojem**.

3. R4Farmer – plánujeme

V následující kapitole se dostaneme k samotnému plánování práce robota probíhající v pracovním vlákně `NavCoreThread` (viz sekce 2.3.2). Navrhneme několik různých algoritmických přístupů a v závěru přímo porovnáme jejich výkonnost na různých typech vstupních dat.

3.1 Definice problému

Zopakujme, co máme k **dispozici**, a co je naším **cílem**. V Konfiguračním modulu jsme si vymodelovali pole pomocí menších elementů a jejich sjednocením dostaneme nějaký obecný polygon. Stejně tak máme seznam polygonů představující překážky v poli, pozice bodů energetické obsluhy a startovní pozici robota. Robot má dané rozměry a na své spodní straně má rozmístěné rycí, secí a zavlažovací nástroje, i u nich známe rozměry a pozici. Známe energetickou náročnost jednotlivých operací robota. Předpokládejme, že je u robota nastaven program setí. Úkolem je efektivně (vzhledem ke spotřebě energie) zpracovat (zasít) celé pole pomocí daného (secího) nástroje. Je třeba hledět na úroveň zbývající energie a osiva a tyto komodity včas doplnit v bodě energetické obsluhy a je třeba vypořádat se s případnými neočekávaným překážkami v poli.

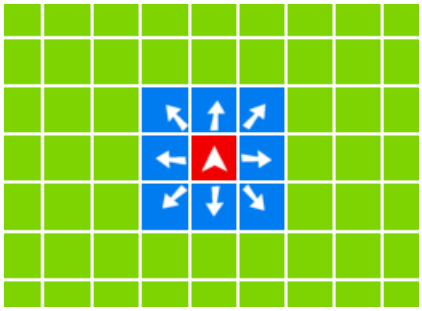
Problémům tohoto typu, kdy je třeba efektivně pokrýt nějakou plochu, se souhrnně říká *coverage planning* a existují v zásadě dva hlavní přístupy – jaké to jsou a proč se příliš nehodí pro náš konkrétní problém se podíváme v podkapitole 3.5.

Zkusíme k problému přistoupit po svém a nejprve představíme spíše experimentální algoritmus (R4F-A), následně jeho modifikovanou verzi (R4F-B), která nám již dá pěkné uniformně zpracované pole, a další modifikaci (R4F-C), jejímž cílem bude další snaha o minimalizaci zdrojů. V závěru provedeme analýzu navržených algoritmů.

3.2 Rozhodovací přístup (algoritmus R4F-A)

Komplexní problém plánování budeme chtít co možná nejvíce zjednodušit. Danou úlohu **zdiskretizujeme**, čímž dostaneme konečnou množinu stavů, na kterých budeme operovat – pole zadané jako polygon se vhodně *rozkrájí* na menší čtverečky, tj. provedeme dekompozici pole na pravidelnou čtvercovou mřížku, daný proces nazveme ***samplováním***. Získané čtverečky (samply) se pokusíme efektivně uložit, abychom mohli sledovat, které z nich již máme zpracované a které nikoli. K samotnému zpracovávání pole dochází **přejezdy** mezi jednotlivými samply. Úkolem bude projet všechny nezpracované samply dekomponovaného pole. Pozor bude třeba dávat na kolize s překážkami. Pro hlídání úrovně energie bude nutné dalšího předzpracování, tentokrát přímo s body energetické obsluhy – kolem každého z nich určíme zóny předpokládané spotřeby pro dojezd k danému bodu, díky kterým se bude moci robot rychle rozhodovat.

Jakmile budeme mít vše připravené, můžeme robota vyslat na pole. V každém kroku se bude na základě určených kritérií rozhodovat, kterým z osmi světových



Obrázek 9: Rozhodovací přístup – možnosti dalšího postupu

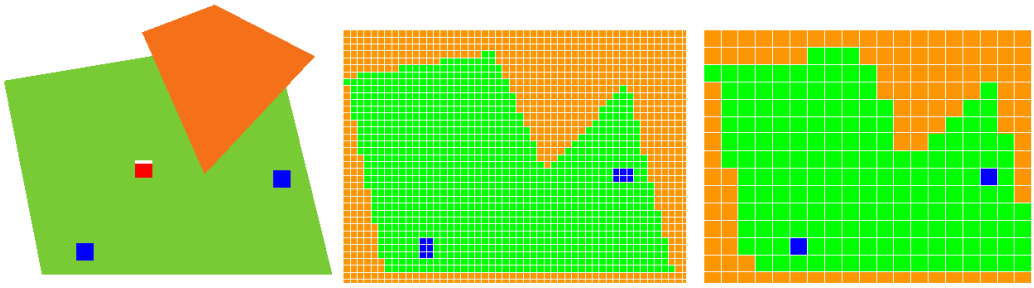
směrů se vydat (viz obrázek 9). Algoritmus skončí ve chvíli, kdy budou všechna náležitá políčka zpracována.

3.2.1 Dekompozice pole – samplování

Samplování je **diskretizační proces**, kde zjednodušíme libovolně škálovatelná vektorová data (elementy pole, překážky, body energetické obsluhy, ...) do podoby jednoduché mřížky. Jednotlivé čtverečky této mřížky nazveme **samply**. Velikost těchto samplů bude určena *jemností* samplování – tento parametr nazveme ***samplovací rozlišení***. Tzn. pokud bude mít samplovací rozlišení např. hodnotu 300, znamená to, že je celé pole rozloženo na čtvercovou mřížku, kde hrana každého samplu má 300 cm.

Pak již nezbývá, než pro každý sample určit jeho hodnotu, tedy *kam patří*. To provedeme tak, že vezmeme bod uprostřed každého ze samplů a podíváme se zpět do vektorových dat, zda tento bod patří do některého z elementů pole, elementů překážky, nebo jde o bod energetické obsluhy, či je úplně mimo (pak je rovněž označen jako překážka).

Příklad samplovaného pole při různém rozlišení ilustruje obrázek 10. O tom, jakým způsobem pole nasamplovat a jak nasamplované pole efektivně uložit, si řekneme více při samotné aplikaci v sekci 3.2.3.



Obrázek 10: Samplování pole pro různá rozlišení – originál / 200 cm / 500 cm

3.2.2 Detekce kolizí

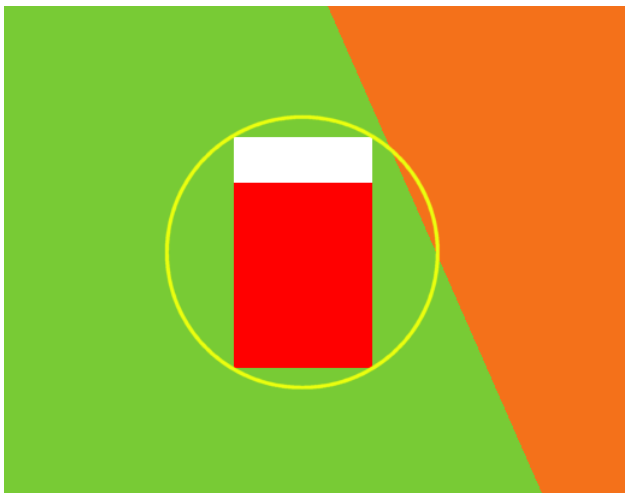
Při dalším zpracovávání se nám bude hodit mechanismus pro detekci kolize robota s překážkou, resp. mechanismus pro **předcházení těmto kolizím**.

Robot se otáčí na místě, kolem svého středu *jako tank*. Pro dosažení maximální úrovně bezpečnosti je třeba zajistit, aby se robot v žádném okamžiku své práce nedostal do postavení, kdy by například otočení jedním směrem znamenalo vyjetí části jeho těla mimo pole či kolizi s překážkou (pozn.: robot nedisponuje *zpátečkou*, takže není možné se z těchto nepříznivých stavů vrátit prostým couvnutím na předchozí stav).

Pro detekci kolizí nějakého objektu s překážkami se obecně používají nějaké **obalové struktury** (typicky obdélník, kružnice, konvexní obal apod.), které se pak testují na průnik s hranami kritických polygonů.

Pro náš případ bude jistě nejvhodnější **obalová kružnice**, se středem v centru robota. Oblast ohraničená touto kružnicí udává prostor všech možných otočení našeho robota na dané pozici, a právě tuto oblast budeme chtít uchránit před kolizemi. Kružnici budeme testovat na průnik s hranami elementů překážek a hranicemi pole (jednoduchý geometrický algoritmus na určení průniku kružnice a úsečky).

Obrázek 11 níže ilustruje situaci, kdy robot samotný sice není v kolizi s překážkou, nicméně jeho obalová kružnice ano – pokud by se chtěl robot v tomto okamžiku otočit o 90° doleva, došlo by ke kolizi jeho zadní části s překážkou.

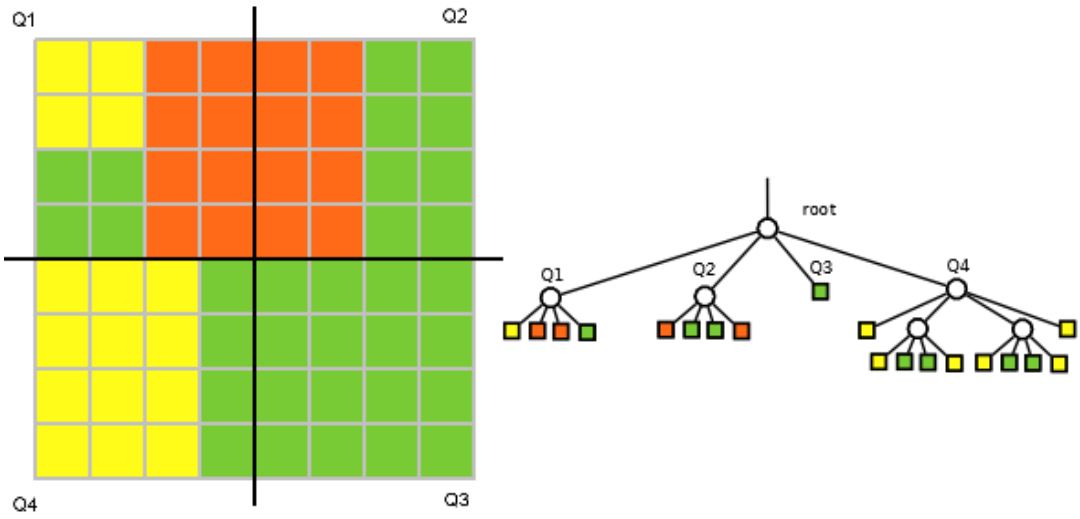


Obrázek 11: Kolize obalové kružnice s překážkou

3.2.3 (Ne)zpracované části pole

Podstatnou otázkou našeho aktuálního přístupu je, jak si pamatovat, které části pole již máme zpracované a které doposud nikoli (a také ty, které nebudeme chtít zpracovávat vůbec). Samplováním si z pole vytvoříme teoreticky velmi rozsáhlou mřížku, kterou budeme chtít **efektivně uložit** v paměti. Pro úsporné kódování dvourozměrných dat se bude jevit velmi výhodně datová struktura zvaná *kvadrantový strom* (angl. *QuadTree* nebo jen *QTree*). Tu si nyní blíže představíme.

Kvadrantový strom [7] představuje stromovou datovou strukturu, jejímž specifikem je, že každý uzel tohoto stromu obsahuje právě čtyři potomky. Nejčastěji se používá pro úspornou reprezentaci nějakých dvourozměrných dat, a to takovým způsobem, že rekurzivně dělí daný prostor do čtyř regionů (kvadrantů).



Obrázek 12: Kvadrantový strom – úsporné kódování pole

Z podstaty tohoto dělení vyplývá, že potřebujeme, aby původní prostor (bodů, pixelů) měl velikost $2^n \times 2^n$. Každý z takto získaných kvadrantů se buď dále dělí na další subkvadranty, nebo jde o list – reprezentující hodnotu pro celý region.

Při reprezentaci nějakých dvourozměrných dat pomocí matice o velikostí $n \times n$ bodů dostaneme prostorovou složitost $\Theta(n^2)$. Kvadrantový strom nám sice dá v nejhorším případě složitost $O(n^2)$, nicméně v praxi se budeme dostávat na hodnotu mnohem nižší, obzvláště při reprezentaci nějakých souvislých ploch, což je přesně náš případ. V nejlepším případě se můžeme dostat až na úroveň $O(1)$ – například obrázek, který má celý jednu barvu, bude reprezentovaný jediným vrcholem stromu (kořenem).

Názornou představu o kvadrantovém stromu dává obrázek 12. Jde vidět, že pokud bychom původní pole reprezentovali bod po bodu, potřebovali bychom 64 záznamů, při kódování s kvadrantovým stromem si však vystačíme s pouhými 19 záznamy.

Časová složitost přístupu k libovolnému bodu uloženém v kvadrantovém stromu odpovídá hloubce stromu, která pro obrázek $2^n \times 2^n$ bodů odpovídá n. Čili přístup k libovolnému prvku kvadrantového stromu bude stát logaritmický čas.

Kvadrantový strom budeme stavět *odspodu*, postupně budeme zjišťovat hodnoty jednotlivých bodů a případně je budeme spojovat do větších celků. Díky tomu nám stačí zjistit hodnotu každého z bodů právě jednou (narozdíl od konstrukce *shora*, dle definice), a strom tak postavíme v čase $\Theta(n^2)$.

Nyní využijeme kvadrantový strom k efektivnímu **uložení nasamplovaného pole** (viz samplování – sekce 3.2.1). V prvé řadě potřebujeme určit vhodné **samplovací rozlišení**. Chceme, aby se část pole odpovídající nějakému samplu při zpracování tohoto samplu skutečně zpracovala. Dané nástroje na spodku robota jsou vždy horizontálně vycentrované, díky čemuž můžeme stanovit, že samplovací rozlišení bude odpovídat šířce daného nástroje, a pokud řekneme, že robot stojí na nějakém samplu, pak to znamená, že jeho centrální bod (střed robota) je uprostřed daného samplu. Nyní, pokud robot bude projíždět pole, pak každý sample, který **zpracuje**, bude skutečně odpovídat zpracovanému úseku na poli. Daný nástroj pochopitelně nemusí mít čtvercový rozměr a nemusí být umís-

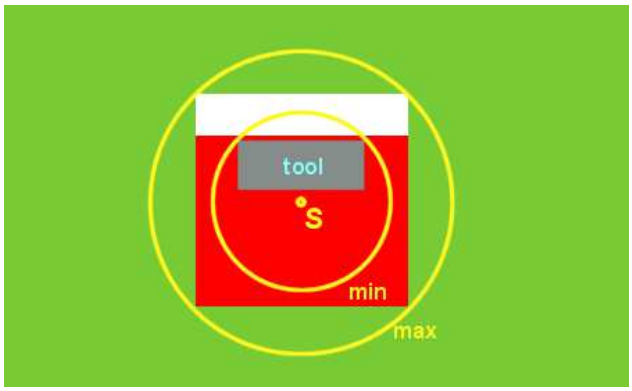
těn vertikálně přesně ve středu robota (o možných problémech více v sekci 3.2.9), ovšem k samotnému zpracování pole dochází právě při přejezdech mezi samply.

Uvedli jsme, že pozice robota v samplu odpovídá pozici jeho středové souřadnice na středu samplu. Aby robot byl v korektní pozici na nějakém samplu ještě před odstartováním své práce, musíme kvůli tomuto **upravit počátek**, odkud započneme samplování pole. Určíme vektor posunu středu robota na střed nejbližšího samplu (v případě že bychom započli samplování pole na pozici [0,0] odpovídající nejlevějšímu hornímu rohu celého pole) a posuneme zmíněný počátek samplování o opačný vektor.

Jak jsme již zmínili výše, v kvadrantovém stromu kódujeme obrázky a plochy, které mají rozměr $2^n \times 2^n$, čili si pro naše pole určíme **nejmenší obalový čtverec**, který tomuto odpovídá.

Nyní máme vše, co potřebujeme, abychom naše nasamplované pole mohli uložit do kvadrantového stromu. Při ukládání pole nyní navíc vstoupí do hry **detekce kolizí**, kterou jsme si představili v sekci 3.2.2. Pokud budeme ukládat nějaký sample, který odpovídá poli (není mimo pole, není to překážka, ani bod energetické obsluhy), vezmeme v úvahu obalovou (kolizní) kružnici pro robota, pokud by na tomto samplu stál, a provedeme detekci kolizí. Pokud zaznamenáme kolizi s překážkou či hranicí pole, daný sample se označí jako překážka a tato oblast se nebude zpracovávat. Obdobně provedeme s body energetické obsluhy.

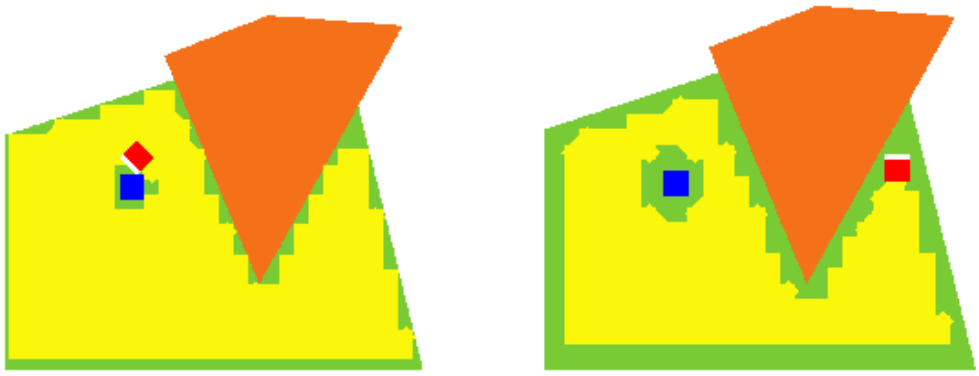
Toto může vést k **menším ztrátám** při nezpracování okolí překážek a hranic pole, ty však budou zanedbatelné (a hlavně s tím nic nenaděláme, pokud robot k daným okrajům prostě nemůže jet).



Obrázek 13: Velikost kolizní kružnice robota při nastavení min. a max. úrovně parametru *collision safety level*

Máme k dispozici možnost ovlivnit, **jak důkladná** detekce kolizí bude. Úplná kolizní kružnice sice zajistí maximální míru bezpečí, ovšem za cenu toho, že větší okrajové regiony v poli budou nezpracované. V Konfiguračním modulu v nastavení parametrů projektu lze určit hodnotu této úrovně (označeno jako *collision safety level*). Pokud zvolíme **maximální úroveň** (implicitně), pak je provedená úplná kontrola s kolizní kružnicí odpovídající obalové kružnici celého robota. Máme tedy zaručeno, že k žádné kolizi nedojde. **Minimální úroveň** tohoto parametru znamená, že kolizní kružnice má stále svůj střed ve středu robota a její poloměr je určen polohou daného nástroje – kružnice musí vždy obalovat daný nástroj. Při libovolném otočení tak bude nástroj stále v dané kolizní kružnici. Situaci ilustruje obrázek 13.

Při minimální úrovni parametru jistě nedojde k zpracování těch částí pole, které jsou mimo hranice pole, jsou bodem energetické obsluhy, nebo jsou překážkami, ovšem některé části konstrukce robota při zpracování teoreticky mohou kolidovat se zmíněnými elementy. Můžeme tedy počítat s menšími ztrátami, ovšem vyšším rizikem kolize. Záleží na volbě uživatele. Vymezením pole v Konfiguračním modulu může myslet striktně oblasti, kde má robot povoleno se pohybovat, nebo jako oblasti, které chceme skutečně zpracovat a drobný *výjezd* robota mimo pole nevadí. Co nastavení tohoto parametru na minimální a maximální úroveň udělá se zpracováváním pole, ilustruje obrázek 14.



Obrázek 14: Zpracované pole při min. a max. úrovni parametru *collision safety level*

Poměr ztrát v nezpracovaných částech pole bude klesat s nižším samplovacím rozlišením (tj. drobnější dekompozicí) a větší velikostí pole. Je třeba vzít v úvahu, že pole na obrázku 14 je pro ilustrační účely netypicky malé a daný robot netypicky veliký (5×5 m).

Při ukládání nasamplovaného pole do kvadrantového stromu se ukládají 3 hodnoty: `Q_BAULK` (překážka či oblast mimo hranice pole), `Q_ENERGY` (bod energetické obsluhy) a `Q_UNPROCESSED` (pole jako takové, zatím nezpracované) – viz obrázek 10. Při zpracovávání se samply s hodnotami `Q_UNPROCESSED` postupně budou měnit na `Q_PROCESSED`.

Celý tento proces je prováděn ve fázi **pre-processingu**, před spuštěním práce robota. Délka tohoto pre-processingu se bude lišit v závislosti na velikosti pole (a hodnotě samplovacího rozlišení), neměla by však přesáhnout hodnotu několika sekund.

3.2.4 Kontrola úbytku energie a komodit

Během provádění algoritmu bude nutné hlídat zbývající množství energie v robotovi – ten musí včas dorazit na některý z **bodů energetické obsluhy** (na obrázcích označen modrou barvou), aby své akumulátory dobil. Program nesmí dopustit, aby se robot ocitl uprostřed pole se zcela vybitou baterií, v takovém případě by šlo o vážné selhání algoritmu. V bodech energetické obsluhy si robot rovněž doplňuje zbývající komodity, vodu a osivo – jednotlivé body energetické obsluhy však mají omezené kapacity těchto komodit.

Zmíněné **hlídání** bude probíhat tak, že se *jednou za čas* robot podívá na svou zbývající úroveň energie, podívá se okolo na body energetické obsluhy a zkalku-

luje, zda má stále dostatek energie na dojetí k nějakému z těchto bodů. Pokud množství zbývající energie klesne pod určitou úroveň – tj. situace, kdy má robot dostatek energie na dojetí pouze k jednomu z bodů energetické obsluhy a i tato potřebná energie **klesne pod kritickou hranici** (s určitou rezervou pro případ neočekávaných překážek a s ohledem na možné zvýšení spotřeby vlivem výškových nerovností, viz dále), robot se vydá k danému bodu energetické obsluhy. Tzn. provede se hledání nejkratší cesty k tomuto bodu a robot se přesně po této cestě vydá.

Pro kalkulaci potřebného množství energie na dojetí k bodu energetické obsluhy můžeme použít Eukleidovskou vzdálenost mezi daným bodem a robotem a určit nejhorší případ spotřeby pro přímou jízdu mezi oběma body. Toto je však pouze dolní odhad a nám by se velmi hodil právě **odhad horní**. K tomu si definujeme tzv. ***zóny*** okolo každého z bodů energetické obsluhy. Půjde o kruhy (resp. mezikruží) s poloměrem násobků R (např. pro R=100: 100 m – 200 m – 300 m atd.). Pro každou tuto zónu si určíme nejdelší vzdálenost (v samplech) od příslušného bodu energetické obsluhy, jakou lze ujet, a spočítáme nejhorší případ spotřeby (užijeme algoritmu vlny). Tím máme zajištěno, že pro libovolný bod z dané zóny budeme znát nejvyšší možnou spotřebu.

Poté v závislosti na vzdálenosti robota od bodu energetické obsluhy určíme příslušnost do některé zóny a máme nejhorší možný případ spotřeby pro dojetí k danému bodu energetické obsluhy. Robot si tedy při provádění hlavního algoritmu v daný intervalech kontroluje tento **horní odhad spotřeby** pro dojetí ke všem bodům energetické obsluhy a rozhoduje se, zda již není čas dojet se dobít. Ke všemu je navíc zohledněna možnost, že robot při cestě k bodu energetické obsluhy **narazí na neočekávanou překážku** a také možné **zvýšení spotřeby vlivem výškových nerovností** v poli. Jakou rezervu by si měl robot nechávat vzhledem k oběma těmto aspektům určují dané parametry (*úroveň obezřetnosti vůči neočekávaným překážkám* a *úroveň výškových změn*) nastavitelné v parametrech projektu (v Konfiguračním modulu).

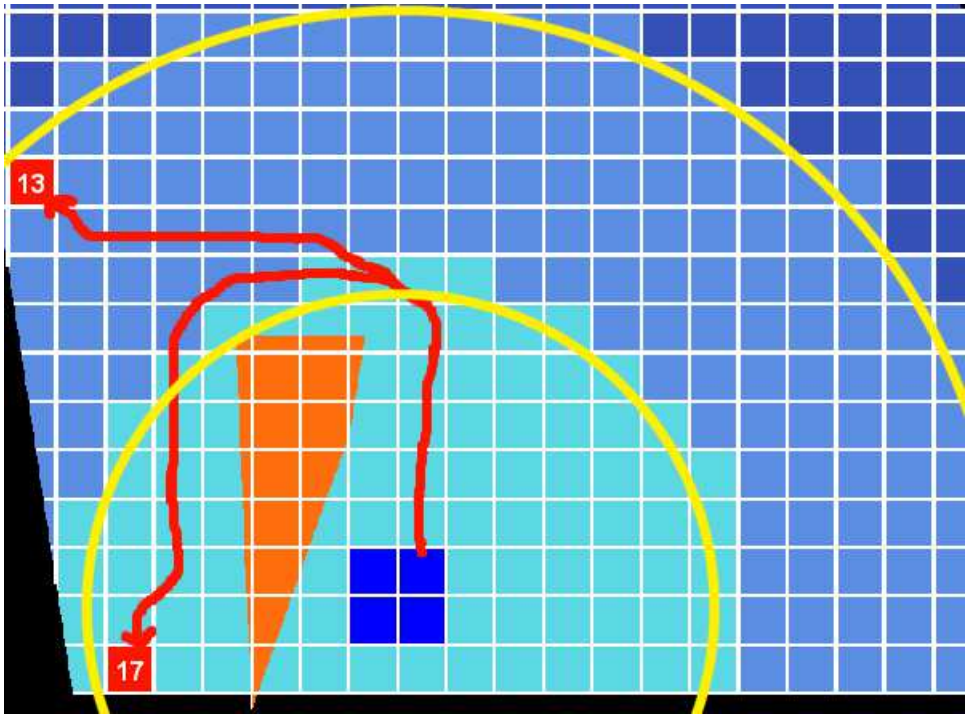
Jak může fungování zón pro body energetické obsluhy vypadat v praxi, ilustruje obrázek 15. První zóna má zde nejvzdálenější bod 17 samplů daleko, v druhé vzdálenější zóně je to paradoxně pouze 13 samplů. Tím máme tedy zaručeno, že pokud se robot bude nacházet v první zóně daného bodu energetické obsluhy, budeme potřebovat nejvýše energii na zdolání 17 kroků.

Stinnou stránkou celého **předzpracování** jednotlivých zón bodů energetické obsluhy je to, že v závislosti na velikosti pole a počtu bodů energetické obsluhy může trvat určité množství času (jednotky až desítky sekund).

Hlídání množství **ostatních zbývajících komodit** (voda, osivo) je v tomto přístupu implementováno triviálně – pokud je spuštěn program setí a robotu dojde osivo, najde si nejbližší (ve vztahu ke spotřebě energie) bod energetické obsluhy, který stále obsahuje nějaké osivo, spustí opět trackování a po této cestě se k danému bodu vydá.

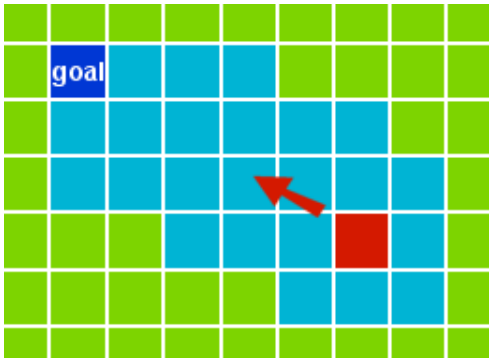
Trackování (myšleno nalezení nejkratší cesty robota k danému bodu energetické obsluhy) probíhá ve všech případech stejně – jde v zásadě o standardní prohledávání do šířky, vylepšené o heuristiku – tzv. *A* prohledávání* [6].

Popišme si blíže fungování **A* prohledávání**. Máme k dispozici prioritní frontu Q, která na počátku obsahuje jediný bod v_{init}. Dokud není Q prázdná, je vždy vybrán bod v – pokud jde o cílový bod, máme hotovo. Jinak jsou k danému



Obrázek 15: Odstupňované zóny bodu energetické obsluhy a nejvzdálenější body pro první dvě zóny

bodu v vygenerovány jeho doposud nenavštívené sousední body a jsou vloženy do Q. Pro každý z těchto bodů máme definovanou ***cenu*** $c(v)$, která bude sloužit pro prioritizaci ve frontě Q. Cena pro bod v bude definována $c(v) = l(v) + h(v)$, přičemž $l(v)$ značí vzdálenost bodu v od počátku v_{init} a $h(v)$ značí hodnotu heuristické funkce pro bod v. **Heuristická funkce** pro bod v je definována jako dolní odhad vzdálenosti bodu v od cílového bodu. Typicky (a bude tomu tak i v našem případě) tento odhad odpovídá přímé vzdálenosti daného bodu v od cílového bodu v Eukleidovské metrice. Prioritizace v Q funguje samozřejmě tak, že je hledána minimální hodnota $c(v)$ a takový bod v je pak vybrán pro další zpracování. Algoritmus selže, pokud zůstane Q prázdná (tzn. neexistuje cesta z původního bodu k cílovému).



Obrázek 16: Ukázka postupu k cílovému bodu při A* prohledávání

Proč je pro nás A* prohledávání **výhodné**? Poměrně často bude nejkratší cesta z aktuálního bodu na bod energetické obsluhy odpovídat *přímé* cestě. A ty-

to body na přímé cestě budou při prohledávání brány přednostně, tj. v mnoha případech dojde k **významnému urychlení** celého prohledávání, ve srovnání s *klasickým* prohledáváním do šířky (viz obrázek 16: k nalezení trasy k cílovému bodu potřeboval A* algoritmus prohledat pouze 25 políček, standardní algoritmus vlny by potřeboval políček 93). I pokud nepůjde o přímou cestu, algoritmus ve většině testů podá podstatně lepší výkonnost (tj. prohledá méně bodů) než jednoduchý algoritmus vlny. Ať tak či onak, A* algoritmus nám vždy zajišťuje **optimalitu řešení**, tj. jistě půjde o nejkratší cestu k cílovému bodu. Časová a prostorová složitost algoritmu je stejná jako u algoritmu vlny.

3.2.5 Neočekávané překážky v poli

Mechanismus **zotavení** při detekci neočekávané překážky jsme do značné míry popsali již v sekci 2.3.2 (komunikace pracovních vláken). Zbývá jen objasnit, co to znamená, že vlákno `NavCoreThread` po oznámení neočekávané překážky od robota *upraví svůj vnitřní stav*, než se opět pustí do plánování.

Vlákno má totiž **dva typy proměnných**, říkejme jim *reálné* a *virtuální*. Reálné proměnné popisují aktuální konfiguraci robota i pole, zatímco virtuální proměnné popisují nějaký budoucí, naplánovaný (zatím pouze *virtuální*) stav. Na základě virtuálních proměnných se plánují akce robota. Jakmile naplánujeme nějakou sérií akcí, robotovi bude typicky relativně dlouho trvat, než všechny tyto akce reálně provede. Až jakmile nějakou akci skutečně vykoná, pošle potvrzení o jejím vykonání `NavCoreThreadu` a ten aktualizuje reálné proměnné.

Tento mechanismus využijeme při detekci neočekávané překážky – jakmile robot takovou detekci oznámí, hodnoty virtuálních proměnných pozbývají smyslu, neboť bude třeba kalkulovat nový plán jízdy. Tzn. virtuální proměnné se nastaví na hodnotu reálných proměnných a začíná se nanovo.

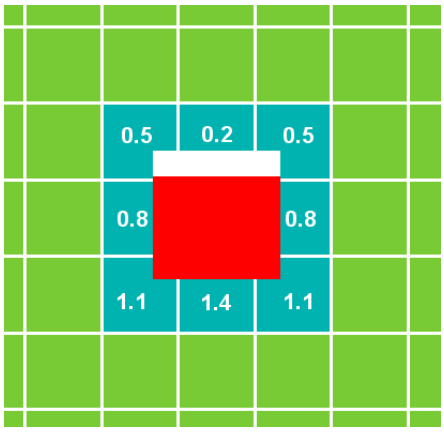
Týká se to i **hodnot v kvadrantovém stromě** – ten máme, nehledě na virtuální a reálné proměnné, pouze jeden a budeme v něm rozlišovat dva typy zpracovaných samplů: `Q_VIRTUAL_PROCESSED` (virtuálně zpracovaný) a `Q_PROCESSED` (reálně zpracovaný). Jakmile budeme potřebovat zneplatnit dříve naplánované akce (vlivem detekce neočekávané překážky), bude nutné všechny hodnoty `Q_VIRTUAL_PROCESSED` v kvadrantovém stromě nastavit zpět na `Q_UNPROCESSED` (nezpracované samply). Procházení celé datové struktury a hledání takových políček by jistě trvalo dlouho, proto robot, jakmile je ve stavu čekání na zotavení, přeposílá všechny naplánované příkazy zpět `NavCoreThreadu` a ten tak bude vědět přesně, kde kvadrantový strom opravit. Následně se samozřejmě do kvadrantového stromu reflektuje nově detekovaná překážka.

Uživatel má možnost nastavit v Konfiguračním modulu tzv. *úroveň obezřetnosti vůči neočekávaným překážkám*. Minimální hodnota tohoto parametru znamená to, že si je uživatel jistý, že v poli nejsou neočekávané překážky, zatímco maximální úroveň odpovídá značnému riziku detekce neočekávaných překážek – robot si tedy nechává **větší rezervu** pro dojezd k bodu energetické obsluhy.

3.2.6 Kritéria rozhodování

Jsme ve fázi, kdy je vše připravené na to, aby se robot mohl pustit do práce. V každém kroku se v závislosti na aktuální pozici robota vyberou **všechny mož-**

né samply (tahy) okolo něj, na které by se mohl robot přemístit (viz obrázek 17). Každý z těchto možných kroků **ohodnotíme *ztrátovou funkcí***, kterou si níže definujeme, a vybereme krok s nejnižším ohodnocením. Pro daný krok vygenerujeme sérii příkazů pro robota – typicky případné vypnutí/zapnutí nástroje, správné otočení a pohyb vpřed. Tato sekvence se pošle na příjmový buffer robota pro zpracování.



Obrázek 17: Robot a potenciální tahy z aktuální konfigurace (hodnoty ztrátové funkce pro tahy pouze ilustrační)

Ztrátová funkce bude určovat naší strategii pohybu po poli. Bude definována jako (pro každý tah u z množiny možných tahů U):

$$l(u) = \texttt{consum}(u) + K * \texttt{IsIsolated}(u) + L * \texttt{NearestUnprocessed}(u) + M * \texttt{IsBaulked}(u),$$

kde `consum`(u) značí množství spotřeby pro provedení daného tahu z aktuální konfigurace robota, `IsIsolated`(u) a `IsBaulked`(u) bude vracet hodnoty z $0, 1$ a `NearestUnprocessed`(u) bude vracet hodnoty z $0, ..., \infty$. K, L, M jsou konstanty, pro které platí $0 << K << L << M < \infty$.

Funkce `IsBaulked`(u) je nejjednodušší, vrátí 1 pokud daný sample bod v kvadrantovém stromu má hodnotu `Q_BAULK`, tj. jde o překážku nebo oblast mimo hranice pole.

Funkce `NearestUnprocessed`(u) vrací vzdálenost daného sample bodu od nejbližšího nezpracovaného samplu. Pokud sám tah u je nezpracované políčko, vrátí se 0, pokud v celém poli již není žádné nezpracované políčko, vrátí se speciální hodnota ∞.

`IsIsolated`(u) přímo definuje naši ***nalepovací* strategii** – vrací 1, pokud sousední sample body jsou taktéž nezpracovaná políčka (tedy je *izolované*), a vrací 0, pokud některé ze sousedních polí je již zpracované. Čili právě tento parametr určuje ve svém důsledku nejvíce plánovanou trasu robota. Naším záměrem je, aby robot zpracovával **souvislé úseky** pole, nikoli nějaké izolované plochy. Pomocí této funkce lze blíže definovat lehce odlišné přístupy (na to se podíváme blíže v sekci 3.2.8).

Samotná implementace v programu **není doslovnou interpretací** výše definované ztrátové funkce. Tahy, které jsou překážkami (resp. mimo oblast pole), jsou vyřazeny okamžitě a není dopočítávána celá ztrátová funkce. Podobně, pokud existuje alespoň jeden tah, který reprezentuje nezpracované políčko, jsou ostatní

tahy (které nejsou nezpracované) vyřazeny z dalšího hodnocení. Ve skutečnosti se nebude pro každý potenciální tah určovat hodnota `NearestUnprocessed(`u`)`, ale pouze v případě, že okolo robota neexistuje žádné nezpracované políčko, se spustí prohledávání do šířky (algoritmu vlny) a nalezne se nejbližší nezpracovaný bod, ke kterému se robot vydá.

3.2.7 Souhrn algoritmu R4F-A

Diagram na obrázku 18 ilustruje **proces zpracovávání pole** algoritmem R4F-A. Před započnutím první iterace proběhne pre-processing zahrnující přípravu zón pro body energetické obsluhy (viz sekce 3.2.4) a kvadrantového stromu (viz sekce 3.2.3).

V každé iteraci algoritmu se zkontroluje, zda robot neoznámil detekci neočekávané překážky – v takovém případě dojde k zotavení (přenastavení proměnných a následnému plánování nové trasy – viz sekce 3.2.5).

Následně se určí potenciální tahy z aktuální konfigurace a dle kritérií (sekce 3.2.6) se vybere optimální tah. Pokud žádný ze sousedních bodů není nezpracovaným bodem, vyhledá se pomocí algoritmu vlny nejbližší takový bod. Nenajde-li se žádný, algoritmus skončí. Pokud se nezpracovaný bod najde, určí se sekvence příkazů nutných pro přemístění na tento bod. Stejná sekvence je nutná i při přemístění na sousední bod, pokud se našel alespoň jeden potenciální tah.

Daná sekvence se následně prověří vzhledem ke spotřebě energie a ostatních komodit (viz sekce 3.2.4). Pokud provedením dané sekvence příkazů neklesne úroveň energie či komodit pod kritickou úroveň, sekvence se pošle robotovi. V opačném případě se najde bod energetické obsluhy a robot se k němu vydá. Po každém částečném zpracování pole se aktualizují příslušné samply v kvadrantovém stromě.

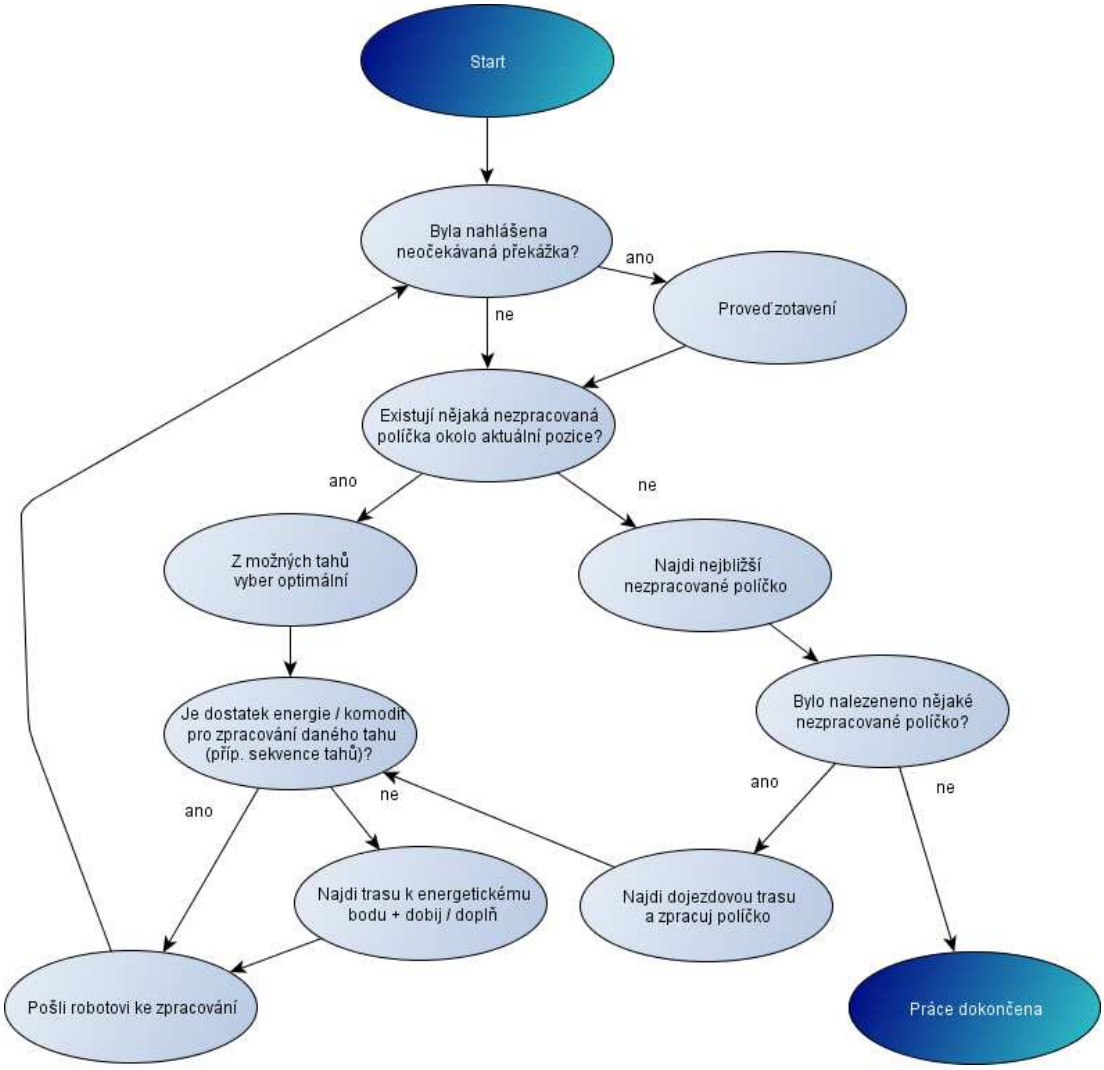
Nástroj pro aktuální program robota je puštěn po celou dobu, k jeho vypnutí dochází pouze při otáčení robota (neboť tato operace může trvat relativně dlouhou dobu), při jízdě za vzdáleným nezpracovaným políčkem nebo jízdě za bodem energetické obsluhy.

Diagram je zjednodušený v tom, že i po dokončení plánování program stále přijímá potvrzení od robota a v případě detekce neočekávané překážky dojde k zotavení a znovuspuštění plánovací fáze. Navíc jde pouze o vykonání jednoho z nastavených programů robota, např. setí – pokud je nastaveno navíc zavlažování, provede se re-inicializace kvadrantového stromu a algoritmus se spustí nanovo. Při doplňování vody se případné zbývající osivo vysype v aktuálním bodu energetické obsluhy.

Navíc, program může z mnoha důvodů selhat, např. pokud na všech energetických bodech dojde osivo/voda, pokud je nějaký energetický bod nedosažitelný apod.

3.2.8 Jiné varianty strategií

O směřování trasy robota se nejvíce zasazuje funkce `IsIsolated(`u`)`, popsaná v sekci 3.2.6, která pro možný tah u určuje, zda je *izolovaný*, tzn. žádné ze sousedních políček (samplů) není zpracované. Lépe ohodnoceny tím pádem budou ty potenciální tahy, které izolované nejsou, čili mají za souseda již zpracované



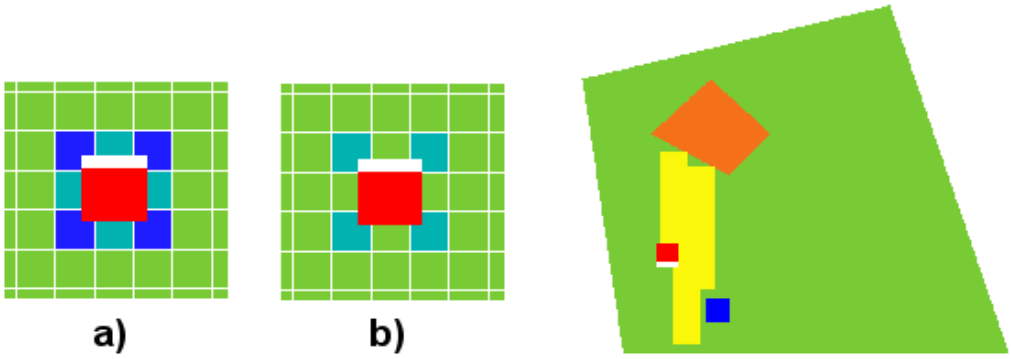
Obrázek 18: Diagram znázorňující proces plánování algoritmu R4F-A

políčko. Robot se tedy snaží zpracované pruhy na sebe ***nalepovat***, zpracovávat pokud možno souvislé úseky.

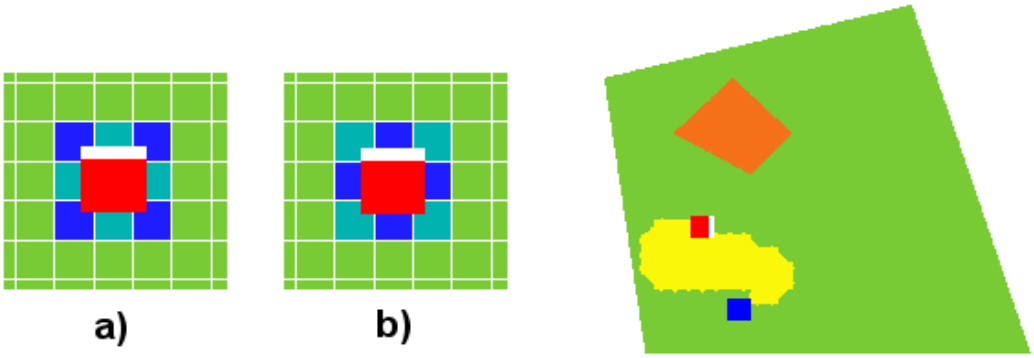
Implementovaná verze (obr. 19) bere v úvahu pouze tahy v horizontálním a vertikálním směru a zkoumá jejich izolovanost (obr. 19 a) – purpurová políčka jsou potenciální tahy a přísl. sousední modrá políčka pak zkoumaní sousedé), příčné tahy označuje vždy za izolované (obr. 19 b)). Výsledkem je vcelku rozumné souvislé zpracovávání pole.

Alternativní varianta (obr. 20) by mohla brát v úvahu i příčné tahy a jako sousedy porovnávat horizontální a vertikální směr (obr. 20 b) – legenda stejná jako v předchozím případě). Výsledkem je sice souvislé, ale dosti chaotické, zpracovávání pole, které ve všech testech dosáhlo větší spotřeby energie i delší doby zpracovávání (příliš mnoho otáčení).

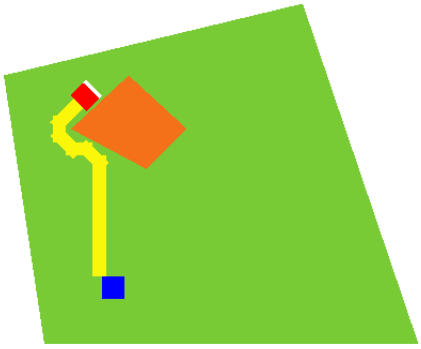
Další varianta implementované strategie by mohla být taková, kdy bychom sousední samply považovali za zpracované i tehdy, pokud by šlo o okraje překážek nebo hranice pole (obr. 21). Taková strategie se však vůbec nejeví jako výhodná, vede opět k dosti chaotickému, a navíc ne vždy souvislému zpracovávání pole, neboť robot se bude často snažit kopírovat okraje překážek či hranic pole.



Obrázek 19: Implementovaná strategie *nalepování* zpracované cesty



Obrázek 20: Alternativní strategie 1



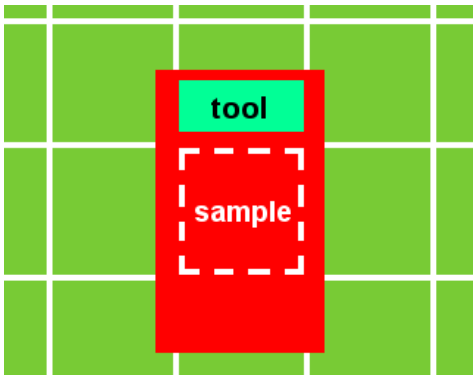
Obrázek 21: Alternativní strategie 2

3.2.9 Zhodnocení algoritmu R4F-A

Algoritmus je velmi **jednoduchý**. Jak již bylo zmíněno, je navržen tak, aby zpracovával co **nejsouvislejší úseky** pole *nalepováním* nově zpracovávaných linek na již zpracované oblasti pole. Ne vždy však svou roli bude vykonávat dobře, podíváme se na **potenciální problémy** při specifických nastaveních projektu.

Jak bylo několikrát zmíněno, poloha robota je vždy dána pozicí středu robota přesně na středu nějakého samplu. V ideálním případě by nástroj robota byl čtvercový a byl umístěn přesně ve středu robota – to by znamenalo, že jakmile se robot nachází na nějakém políčku, automaticky by toto políčko zpracovával

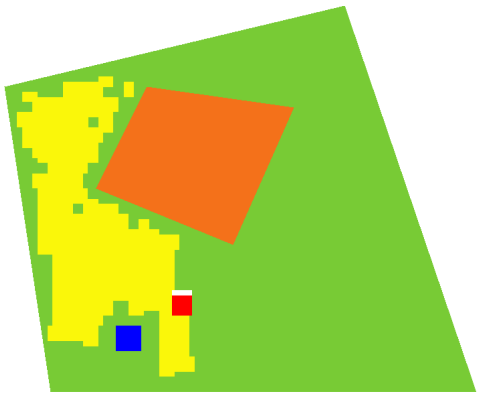
(jelikož rozměr nástroje bude odpovídat rozměru samplu). Takové štěstí však mít často nebudeme, nástroj bude typicky obdélníkový a bude horizontálně posunutý – může dojít i k tomu, že daný nástroj vůbec nebude zpracovávat aktuální sample (viz obrázek 22).



Obrázek 22: Nástroj robota zcela mimo aktuální sample

To může vést k nepěkným **nezpracovaným fragmentům**, jak ilustruje obrázek 23 (je však třeba znovu brát v úvahu, že rozměr robota na obrázku je pro ilustrativní účely nereálně velký, kvůli čemuž se zdají vůči poli větší i způsobené fragmenty). Pokud však nebude tento horizontální posun nástroje od středu robota velký, je aktuální přístup pro zpracování pole použitelný. Proto bylo několikrát výše zmíněno, že robot zpracovává pole přejezdem mezi samply, samotná přítomnost robota na nějakém samplu totiž automaticky neznačí zpracování tohoto samplu.

K minimalizaci problému je třeba **maximalizovat rovné jízdy** robota, tzn. pokud možno minimalizovat frekvenci otáčení. V následujícím přístupu (R4F-B) budeme zpracovávat pole pouze rovnými tahy a problém se tak zcela vyřeší.

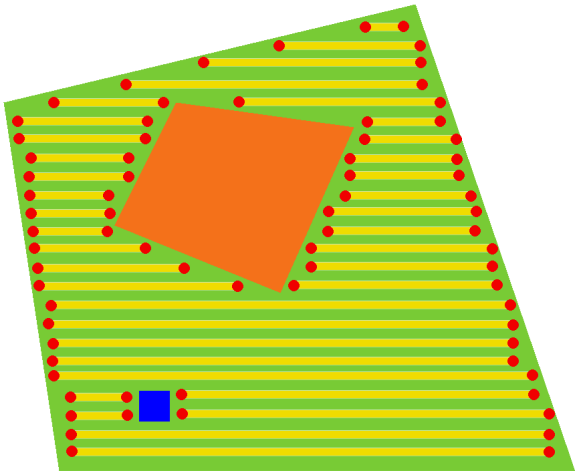


Obrázek 23: Nezpracované fragmenty v poli (extrémní případ)

Další požadavek pro zpracování pole může být jistá **uniformnost zpracovávání**. Algoritmus tak, jak je navržený, ve většině případů **nebude zpracovávat** pole zcela **uniformně** a někdy může degenerovat i do poměrně chaotického obalování zpracovaného úseku tvořením krátkých linek a častým otáčením (což je operace energeticky i časově náročná). Přesně těmto věcem se budeme snažit vyhnout v následujícím přístupu.

3.3 Řádkový přístup (algoritmus R4F-B)

Idea následujícího přístupu je taková, že si celé pole předem rozložíme na **linky** (*řádky*), které následně zpracujeme (viz obrázek 24). Motivace pro tento přístup je taková, že jakmile dokončíme jednu z plánovaných linek, je velká pravděpodobnost, že **hned vedle** bude začínat linka nová – a na tu se vydáme a zpracujeme ji.



Obrázek 24: Řádkování – rozklad pole na zpracovávané linky (řádky)

Robot tak typicky bude zpracovávat plochu pole „cik-cak“ způsobem a výsledkem bude **pěkně uniformně** (všechny pruhy jedním směrem) **zpracované pole**, se snahou o **minimalizaci otáčení robota**, nulovým rizikem degenerace algoritmu do nějakého chaotického obalování zpracovaného úseku a pouze minimem nezpracovaných fragmentů pole u okrajů překážek a hranic pole (jednoduše ty části, kam se robot nedostane). Celé pole budeme chtít opět diskretizovat a využijeme hned několik věcí z přístupu předchozího.

3.3.1 Samplování, kvadrantový strom, řádkování

Celé pole **zdiskretizujeme**, tj. využijeme samplování (sekce 3.2.1) a pro uložení opět použijeme úspornou datovou strukturu kvadrantového stromu (sekce 3.2.3) společně s detekcí kolizí (sekce 3.2.2).

Kvadrantový strom již nebudeme využívat pro reprezentaci nezpracovaných částí pole. Na to nám bude sloužit databáze **nezpracovaných linek**, kterou vytvoříme pomocí procesu, který nazveme ***řádkování*** (obr. 24). Jednoduše se bude postupovat řádek po řádku v nasamplovaném poli a tvořit linky všude tam, kde nejsou překážky, oblasti mimo hranice pole nebo body energetické obsluhy.

Každou z linek budeme reprezentovat počátečním a koncovým bodem (samplem), ty nazveme ***kritickými body***. Každý z těchto bodů si vyznačíme v ploše, tj. v kvadrantovém stromě, pro snadné vyhledání následujícího kritického bodu po zpracování aktuální linky. Za **nový kritický bod** se při zpracovávání pole vezme nejbližší takový bod vůči aktuální konfiguraci (algoritmus vlny na samplovaném poli) a zpracuje se příslušející linka.

Jakmile je linka zpracována, je z odebrána ze seznamu nezpracovaných linek (a jsou odebrány pozice kritických body v kvadrantovém stromě). Díky tomu můžeme vědět okamžitě, zda v poli ještě existují nějaké nezpracované plochy, nebo zda je již **kompletně zpracované**.

Zbývá uvést, že linky dlouhé jeden sample, tj. takové, kde počáteční a koncový kritický bod splývají, při řádkování ignorujeme a nezařazujeme je do databáze nezpracovaných linek. Pole jako takové je zpracováváno přejezdy mezi samply, a pokud máme sample jediný, nemá smysl tuto oblast vůbec zpracovávat.

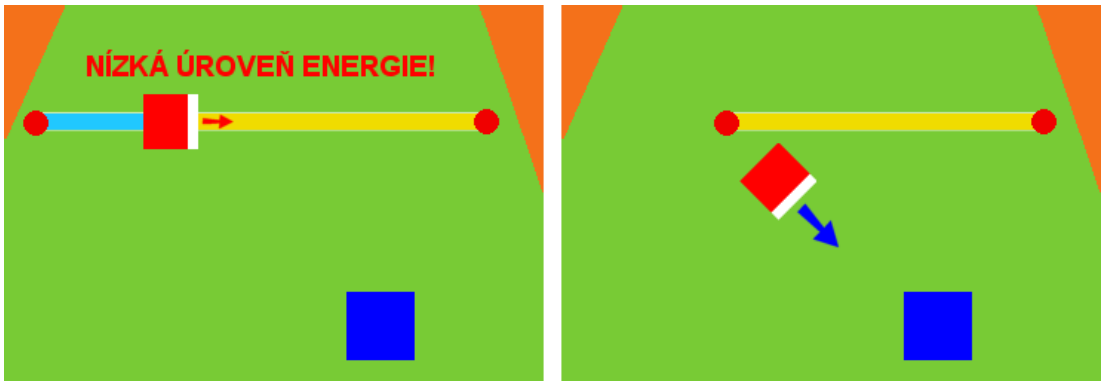
Pokud jde o **uložení** databáze **nezpracovaných linek**, tu bude v zásadě tvořit pouze seznam dvojic bodů (poč. a konc. kritický bod linky), které pro rychlé vyhledávání dle klíče (pozice kritického bodu) umístíme do **asociativního pole**, přičemž každá linka tu bude dvakrát (každý kritický bod je zároveň počátečním i koncovým bodem nějaké linky).

3.3.2 Kontrola úbytku energie a komodit

Kontrola zbývající úrovně energie robota a komodit probíhá v zásadě stejně jako v předcházejícím algoritmu (sekce 3.2.4), včetně předzpracování bodů energetické obsluhy. Průběžně se kontroluje zbývající úroveň akumulátoru a množství komodity a klesne-li jedno z nich pod **kritickou úroveň**, robot se vydá k bodu energetické obsluhy.

Pokud toto nastane během zpracovávání nějaké linky, dojde k **rozdělení této linky** na dvě a zahození té poloviny linky, kterou již robot stihl zpracovat. Situaci ilustruje obrázek 25. Na samplu, kde robot přerušil svoji činnost, se vyznačí **nový kritický bod** a zpracování dané linky je doděláno *někdy v budoucnu*.

Pokaždé, než robot zamíří k nějakému kritickému bodu, se zkontroluje, zda má dostatek energie pro dojetí k tomuto bodu a alespoň nějakému zpracování příslušné linky – aby nejel k danému bodu zbytečně.



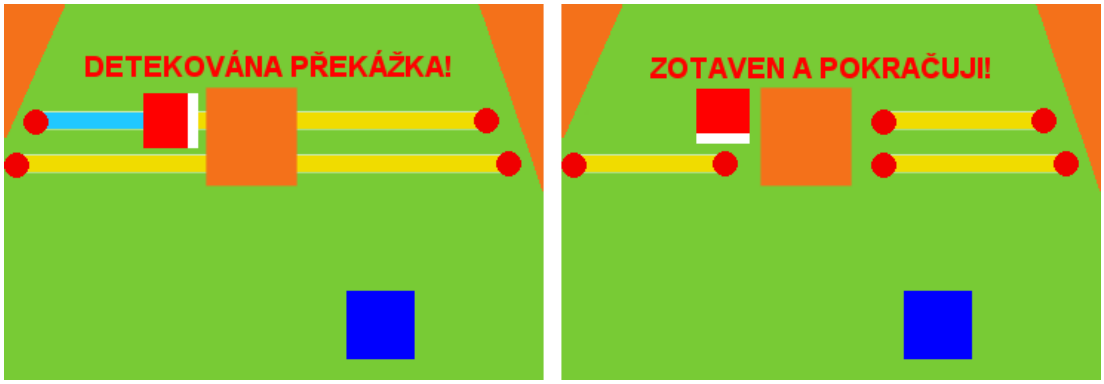
Obrázek 25: Rozdělení zpracovávané linky při detekci nízké úrovně energie

3.3.3 Neočekávané překážky v poli

I řešení neočekávaných překážek je podobné tomu v předcházejícím přístupu (sekce 3.2.5). Mechanismus **zotavování** funguje totožně, opět máme dva typy proměnných, ovšem v kvadrantovém stromě budeme obnovovat pouze pozice kritických bodů.

Důležité je reflektovat detekci neočekávané překážky vzhledem k vybudovanému **řádkování**. Aktuálně zpracovávaná linka se, jako v sekci 3.3.2, rozdělí na dvě a již zpracovaná část se zahodí. Viz obrázek 26. Ovšem je nutno upravit i zbývající nezpracované linky zasahující do nově detekované překážky spolu s nejbližším okolím.

Jakmile se robot zotaví, opět hledá nejbližší kritický bod a vydá se zpracovávat příslušející linku.



Obrázek 26: Zotavení a úprava řádkování po detekci neočekávané překážky

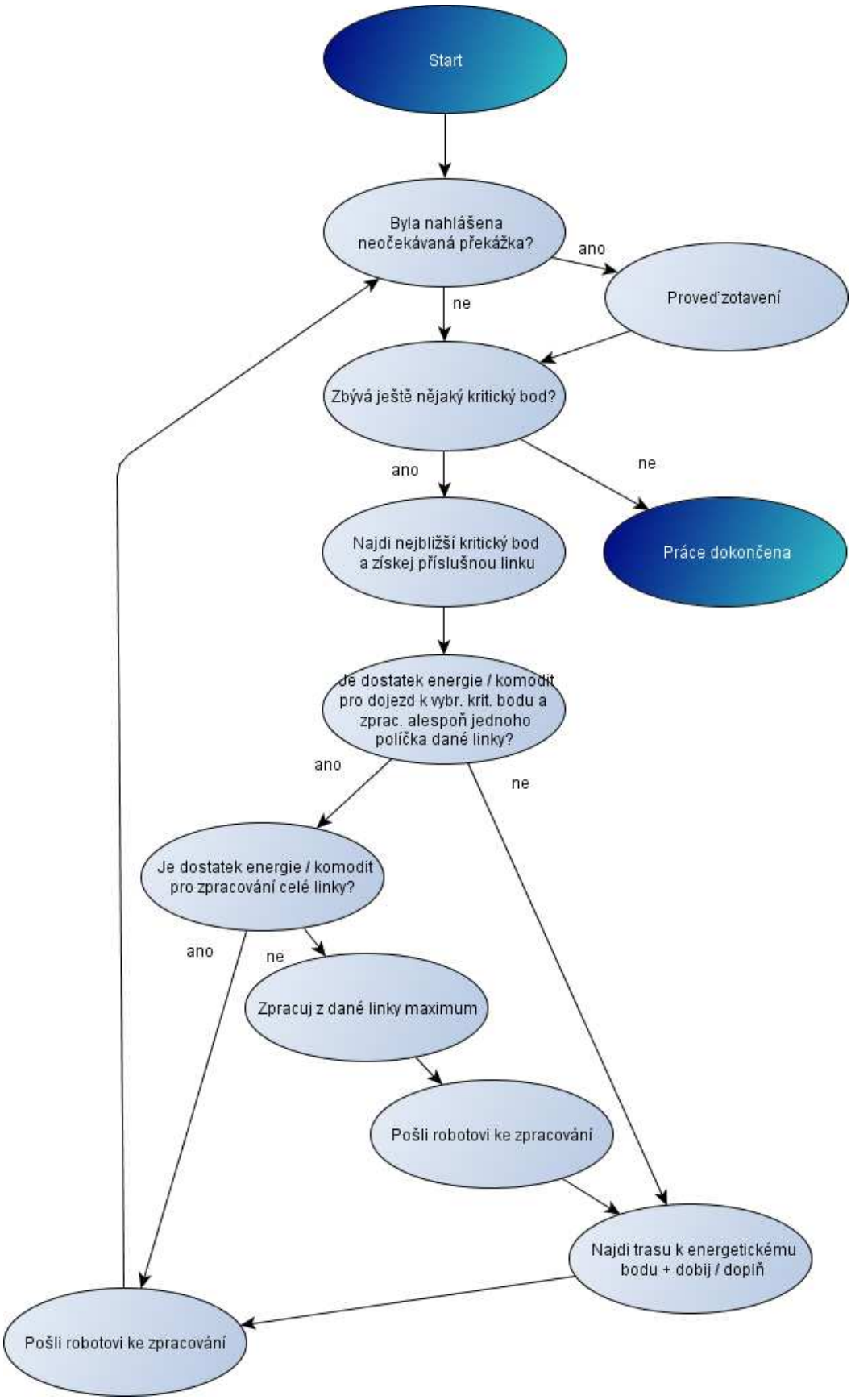
3.3.4 Souhrn algoritmu R4F-B

Diagram na obrázku 27 opět ilustruje **proces zpracovávání pole**, tentokrát algoritmem R4F-B. Před započnutím první iterace proběhne pre-processing zahrnující přípravu zón pro body energetické obsluhy (viz sekce 3.3.2), kvadrantového stromu a databázi kritických bodů (resp. linek; viz sekce 3.3.1).

V každé iteraci algoritmu se zkontroluje, zda robot neoznámil detekci neočekávané překážky – v takovém případě dojde k zotavení (přenastavení proměnných a následnému plánování nové trasy – viz sekce 3.3.3).

Určí se nový kritický bod (nejbližší možný – algoritmus vlny) a příslušná linka pro zpracování. Určí se sekvence příkazů nutných pro dojetí na vybraný kritický bod i pro zpracování příslušející linky a zkontroluje se, zda je k dispozici dostatek energie či dané komodity. Pokud robot nemá dostatek energie ani k dojetí k danému kritickému bodu, vydá se rovnou k bodu energetické obsluhy, jinak zpracuje z dané linky „co se dá“, a v případě potřeby se včas vydá směrem k bodu energetické obsluhy. Po každé akci se aktualizuje kvadrantový strom (mazání pozic bývalých kritický bodů, příp. přidávání nových krit. bodů) a databáze řádků (smazání zpracovaných řádků či částí řádků). Algoritmus skončí ve chvíli, kdy v databázi již nebudou žádné nezpracované linky nebo budou z pozice robota nedosažitelné.

Opět, diagram je lehce zjednodušený v tom, že i po dokončení plánování program stále přijímá potvrzení od robota a v případě detekce neočekávané překážky dojde k zotavení a znovuspuštění plánovací fáze. Navíc diagram znázorňuje průběh vykonávání pouze jednoho z nastavených programů robota, např. setí – pokud je nastaveno navíc zavlažování, provede se re-inicializace kvadrantového stromu a databáze kritických bodů a algoritmus se spustí nanovo. Při doplňování vody se případné zbývající osivo vysype v aktuálním bodu energetické obsluhy.



Obrázek 27: Diagram znázorňující proces plánování algoritmu R4F-B

Program může z mnoha důvodů selhat, např. pokud na všech energetických bodech dojde osivo/voda, pokud je nějaký energetický bod nedosažitelný apod.

3.3.5 Zhodnocení algoritmu R4F-B

Výsledkem práce algoritmu je pěkně, **uniformně zpracované** pole, kde všechny zpracované pruhy budou vést jedním směrem. Toto může být v praxi důležité, protože například chaoticky rozorané pole by mohlo být dále obtížně zpracovatelné.

Lze dosáhnout i toho, aby mezi zpracovávanými linkami v poli byly **mezery**, například určené pro pohonný aparát robota, stačí nastavit rozměry nástrojů na celou šířku robota (i když ve skutečnosti tak široké nebudou).

Zavedení řádkování vyřešilo hned několik problémů prvního přístupu. **Zamezilo se** vytváření **nezpracovaných fragmentů** – v tuto chvíli nám pozice nástrojů na spodku robota nezpůsobuje žádné problémy, neboť pole zpracováváme pouze rovnými linkami. Získali jsme zmíněnou uniformnost zpracování a mělo by dojít i ke **snížení spotřeby**, neboť jsme do značné míry snížili frekvenci zatáčení.

Výhodou je také to, že program při plánování díky evidenci nezpracovaných linek okamžitě vidí, zda již má zpracované celé pole, nebo zda ještě nějaké nezpracované linky existují (typicky není třeba dlouhého prohledávání prostoru).

Prostorem pro další ladění a rozšiřování algoritmu je **výběr nového kritického bodu**. Místo výběru nejbližšího by se například mohla zavést nějaká heuristika, která by brala ohled na další postup při zpracovávání pole. Dále by se mohlo ověřit, zda pro dané pole není výhodnější učinit „sloupečkování“, tj. rozložení pole na linky provést v jiném (vertikálním) směru. Nevýhodou druhého přístupu je, že stále **nebere v potaz úbytek energie a komodit**, a na to se zaměříme v třetím přístupu.

3.4 Zónový přístup (algoritmus R4F-C)

Následující přístup opět založíme na tom předchozím, tentokrát však budeme chtít **zohlednit úbytek komodit** (tj. osiva a vody) v nádrži robota i v samotných bodech energetické obsluhy. Dané pole rozložíme na tzv. **zóny**, kde každá z nich bude příslušet k právě jednomu bodu energetické obsluhy. Robot by měl v ideálním případě obsloužit každou z těchto zón osivem/vodou z příslušejícího bodu energetické obsluhy.

Motivací je víceméně **rovnoměrný úbytek komodit** a zamezení situacím, kdy robot bude muset kvůli doplnění osiva jet potenciálně velkou vzdálenost na jiný než nejbližší bod energetické obsluhy. Místo toho bude zpracovávat postupně jednotlivé zóny s tím, že k validnímu bodu energetické obsluhy to nikdy **nebude** mít **příliš daleko**. Efektivita algoritmu však bude pochopitelně silně záviset na rozmístění jednotlivých bodů energetické obsluhy i množství komodit v nich obsažených.

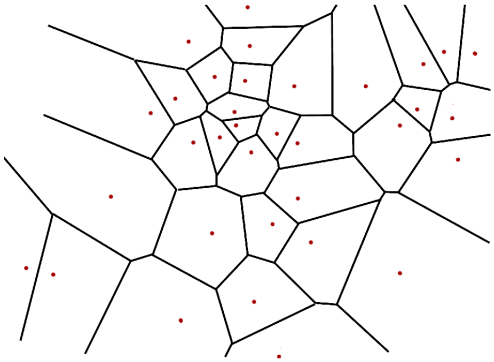
Využijeme opět dekompozice pole na řádky a zachováme si tak všechny výhody předcházejícího přístupu. Kontrola úbytku energie a zotavení při detekci neočekávaných překážek bude fungovat analogicky s předchozími přístupy.

3.4.1 Dekompozice pole na zóny

K rozložení pole na zóny využijeme tzv. ***Voroného diagramy*** [2] (také *Voroného dekompozice* či *Voroného teselace*; angl. *Voronoi diagram*). Jde o speciální typ dekompozice (metrického) prostoru určený vzdálenostmi od nějaké množiny objektů. Tyto objekty se často nazývají ***generátory*** či *seedy* a ke každému z těchto objektů přísluší tzv. ***Voroného buňka*** (či *Voroného region*), což je množina všech bodů prostoru, jejichž vzdálenost od daného generátoru není větší než vzdálenost od kteréhokoli jiného generátoru.

Neboli, formálně řečeno, nechť X je metrický prostor, d Eukleidovská metrika na tomto prostoru, K konečná množina indexů a $(g_k)_{k \in K}$ uspořádaná $|K|$-tice generátorů (bodů v prostoru). Pak Voroného buňka R_k je definována jako $R_k = \{x \in X | d(x, g_k) \leq d(x, g_j), \forall j \neq k\}$. Voroného diagram pak není nic jiného než uspořádaná $|K|$-tice Voroného buněk $(R_k)_{k \in K}$. Voroného generátory přitom obecně nemusí být jednotlivé body, ale nějaké neprázdné podmnožiny daného prostoru. Příklad Voroného diagramu ilustruje obrázek 28.

V našem případě za Voroného generátory **použijeme pozice bodů energetické obsluhy** rozmístěné v poli. K vygenerování Voroného diagramu pak využijeme známého algoritmu Stevena Fortuna, resp. jeho volně šiřitelnou C++ implementaci Shana O'Sullivana (odkazy v sekci 2.1). Tento algoritmus však vygeneruje pouze seznam vnitřních hran diagramu (volitelně zaříznutý na okrajích, jinak by tyto hrany šly do *nekonečna*). Pro další použití bude **nutné získat** přímo jednotlivé **Voroného regiony**.



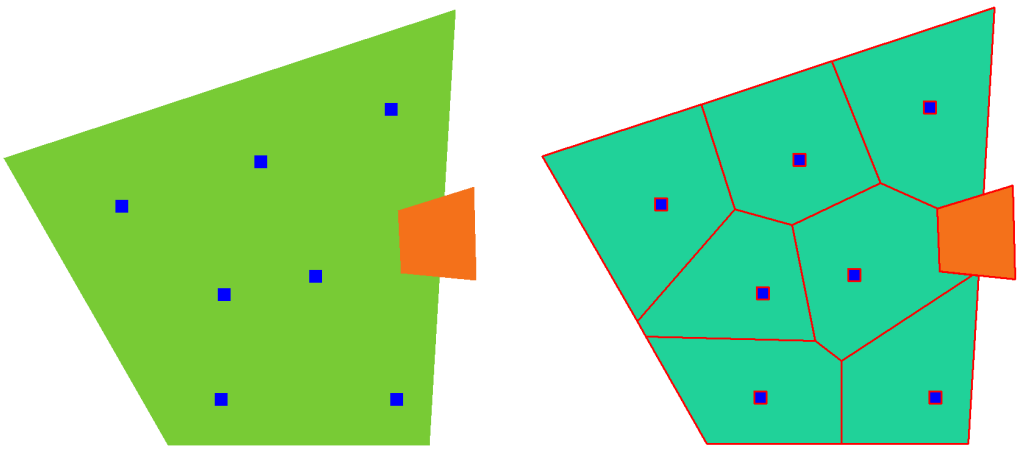
Obrázek 28: Voroného diagram s bodovými generátory

V prvé řadě najdeme obalový čtverec pro celé pole, spustíme O'Sullivanův generátor a doplníme okrajové hrany. Nyní ze seznamů hran potřebujeme dostat jednotlivé polygony. Užijeme hrubou sílu a postupně projdeme všechny vnitřní hrany (ty, které dostaneme od O'Sullivanova generátoru) – u těch víme, že každá z těchto hran odděluje právě dva Voroného regiony. Pro každou z těchto hran najdeme bod ze středu této hrany (jelikož takový bod bude na rozhraní právě dvou Voroného regionů, narozdíl od okrajových bodů hrany, které mohou příslušet třem regionům) a určíme dva Voroného generátory, které mají k tomuto bodu nejkratší Eukleidovskou vzdálenost. Danou hranu asociujeme s danými dvěma generátory a tedy s příslušnými Voroného regiony (uložíme např. do nějakého seznamu). Zbývá to samé provést s okrajovými hranami pole, zde však každá z hran bude příslušet právě jednomu generátoru. Máme tedy seznam hran pro každý Voroného region a jednoduše již sestavíme polygony jakožto seznamy bodů.

Zbývá dané polygony zastřihnout dle skutečného tvaru našeho pole a vzít v úvahu překážky v poli. Vezmeme tedy všechny elementy pole získané modelováním v Konfiguračním modulu (sekce 2.2), provedeme jejich sjednocení a dáme do průniku se získanými Voroného regiony. Stejně tak sjednotíme překážkové elementy a dané oblasti odečteme od Voroného regionů. Výsledkem nám je **seznam polygonů** (Voroného regiony) určující dekompozici pole, příslušející daným bodům energetické obsluhy (použitým jako Voroného generátory). Tuto dekompozici využijeme vzápětí v sekci 3.4.2.

Fortunův algoritmus generuje Voroného diagram v čase $O(n \log(n))$, nicméně celková složitost bude odpovídat $O(n^2)$. Právě tak rychle totiž bude pracovat námi navrhnutý post-processing. Typicky však budeme pracovat s malými diagramy a časová složitost nebude hrát velkou roli, proto nám pro post-processing stačila hrubá síla.

Příklad dekompozice vymodelovaného pole pomocí Voroného diagramu udává obrázek 29. Jednotlivé zóny okolo bodů energetické obsluhy samozřejmě nemusí být stejně velké a jejich rozmístění při modelování bude hrát podstatnou roli při posuzování efektivity algoritmu.



Obrázek 29: Vymodelované pole a příslušný Voroného diagram

3.4.2 Řádkování se zónami

Jako v předcházejícím přístupu opět provedeme diskretizaci pole pomocí samplování (viz sekce 3.2.1), samply následně uložíme do datové struktury kvadrantového stromu (sekce 3.2.3), přitom užijeme i detekci kolizí (sekce 3.2.2).

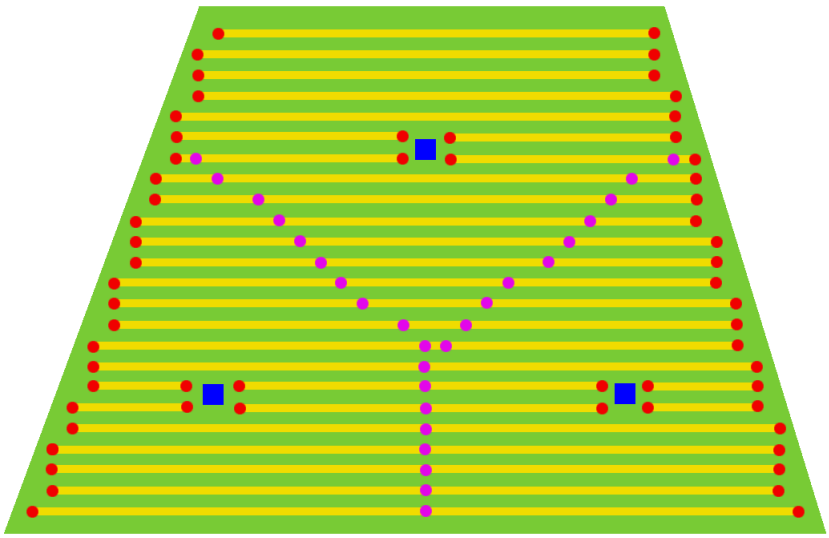
Následně dekomponujeme pole pomocí tzv. ***řádkování*** (viz sekce 3.3.1). Rozdíl bude v tom, že nyní jednotlivé linky navíc rozdělíme podle **příslušnosti do zón** na základě vytvořené dekompozice pole pomocí **Voroného diagramu** v sekci 3.4.1. Znovu budeme stavět linky řádek po řádku v nasamplovaném poli, ovšem každý sample otestujeme na příslušnost do polygonů (Voroného regionů).

Vyjma kritických bodů budeme v kvadrantovém stromě rozlišovat tzv. ***dvojité kritické body*** – to jsou body, kde končí linka jedné zóny a začíná linka jiné zóny. Toto rozlišení dvou typů kritických bodů nám pomůže při následném zpracovávání pole. Indikuje nám, že jakmile skončíme zpracovávání linky na kritickém

bodu, nemůžeme ho automaticky smazat, neboť zde začíná linka nová, příslušející jiné zóně. Příklad vytvořeného řádkování na základě Voroného dekompozice ilustruje obrázek 30 – červeně označené jsou kritické body, fialově pak dvojité kritické body.

Zpracovávání probíhá podobně jako u předcházejícího algoritmu, nicméně zpracovávají se **postupně jednotlivé zóny**. Tzn. jakmile se zpracuje linka příslušející do zóny A, hledá se kritický bod další linky ze zóny A. Jakmile dojde komodita (tj. osivo/voda), jezdí se robot doplnit výhradně do bodu energetické obsluhy příslušné zóny. Po zpracování dané zóny se začne zpracovávat další zóna. Detailnějšímu popisu zpracovávání pole, včetně řešení problematických situací se věnují další sekce níže.

Databázi nezpracovaných linek uložíme pro rychlé vyhledávání do **asociativního pole**, kde klíčem bude číslo zóny a hodnotou další asociativní pole, tj. databáze jednotlivých linek pro danou zónu (dvojice bodů tvořená počátečním a koncovým kritickým bodem). V některých případech sice budeme znát pouze polohu kritického bodu a budeme potřebovat určit index příslušné zóny a daná reprezentace bude spíše nevýhodná, častěji tomu však bude naopak.



Obrázek 30: Řádkování pole se zónami

3.4.3 Kontrola úbytku energie a komodit

Kontrola zbývající **úrovně komodit** (tj. osiva/voda) probíhá stejně jako u předchozího přístupu. Zpracovávají se jednotlivé linky aktuální zóny a jakmile dojde v nádrži daná komodita, aktuálně zpracovávaná linka se rozdělí, zpracovaná část zahodí a nezpracovaná část je odložena na budoucí zpracování, zatímco robot míří k bodu energetické obsluhy příslušné zóny pro doplnění komodity.

Jakmile je **zpracována** jedna zóna, přechází se na sousední. Problém nastává v momentě, kdy v bodu energetické obsluhy **dojde komodita** pro právě zpracovávanou zónu (přičemž stále zbývají linky ke zpracování). Nezpracované linky z aktuální zóny jsou v tu chvíli označeny speciálním indexem -1 vyhrazeným pro tzv. ***odložené linky***. Robot se vydá zpracovávat další zónu, a pokud mu po

zpracování dané zóny naopak **zůstane** určitá **rezerva komodity**, ať už v nádrži robota, tak i v aktuálním bodu energetické obsluhy, jsou tyto komodity okamžitě využity pro zpracování zmíněných dříve odložených linek.

Pokud po zpracování všech zón stále zůstávají nějaké linky s indexem -1, prohledají se všechny body energetické obsluhy, zda v nějakém nezbyla daná komodita a dojde k případnému dokončení zpracování těchto linek.

Samozřejmě zmíněné situace výrazně **snižují efektivitu** aktuálního přístupu. V optimálním případě by měly být body energetické obsluhy v poli rozmístěny tak, aby příslušné zóny zaujímaly přibližně stejnou plochu a v každém z bodů byl dostatek komodit pro zpracování příslušné zóny.

Pokud jde o kontrolu zbývající **úrovně energie**, využijeme stejného pre-processingu jako v předcházejících přístupech a zajistíme si horní odhady spotřeby pro dojezd k jednotlivým bodům energetické obsluhy. Obecně se předpokládá, že komodita (osiva/voda) při zpracovávání pole **dojde rychleji** než energie v akumulátoru. Proto každé doplnění komodit je spojeno s dobitím akumulátoru, aby se předešlo zbytečnému cestování k bodu energetické obsluhy. Nicméně úroveň energie je i tak po celou dobu monitorována a v případě klesnutí pod kritickou úroveň se robot vydá k bodu energetické obsluhy.

Odlišné bude zpracovávání pole **při rytí**. Robot stále zpracovává pole po zónách, nicméně není zde žádná potřeba doplňovat komodity, pouze si hlídá úroveň energie a včas se vydá k bodu energetické obsluhy pro dobití akumulátoru.

3.4.4 Detekce neočekávaných překážek

Detekce neočekávaných překážek a následný mechanismus zotavování funguje **stejně** jako v **předchozím přístupu** (viz sekce 3.3.3). Pouze úprava řádkování je o něco složitější vzhledem k přítomnosti dvojitých kritických bodů, zmíněných výše – u těch je třeba mít na paměti, že zasahují do dvou linek (tj. jsou koncovým bodem jedné linky a počátečním jiné linky), přičemž každá z nich bude patřit do jiné zóny.

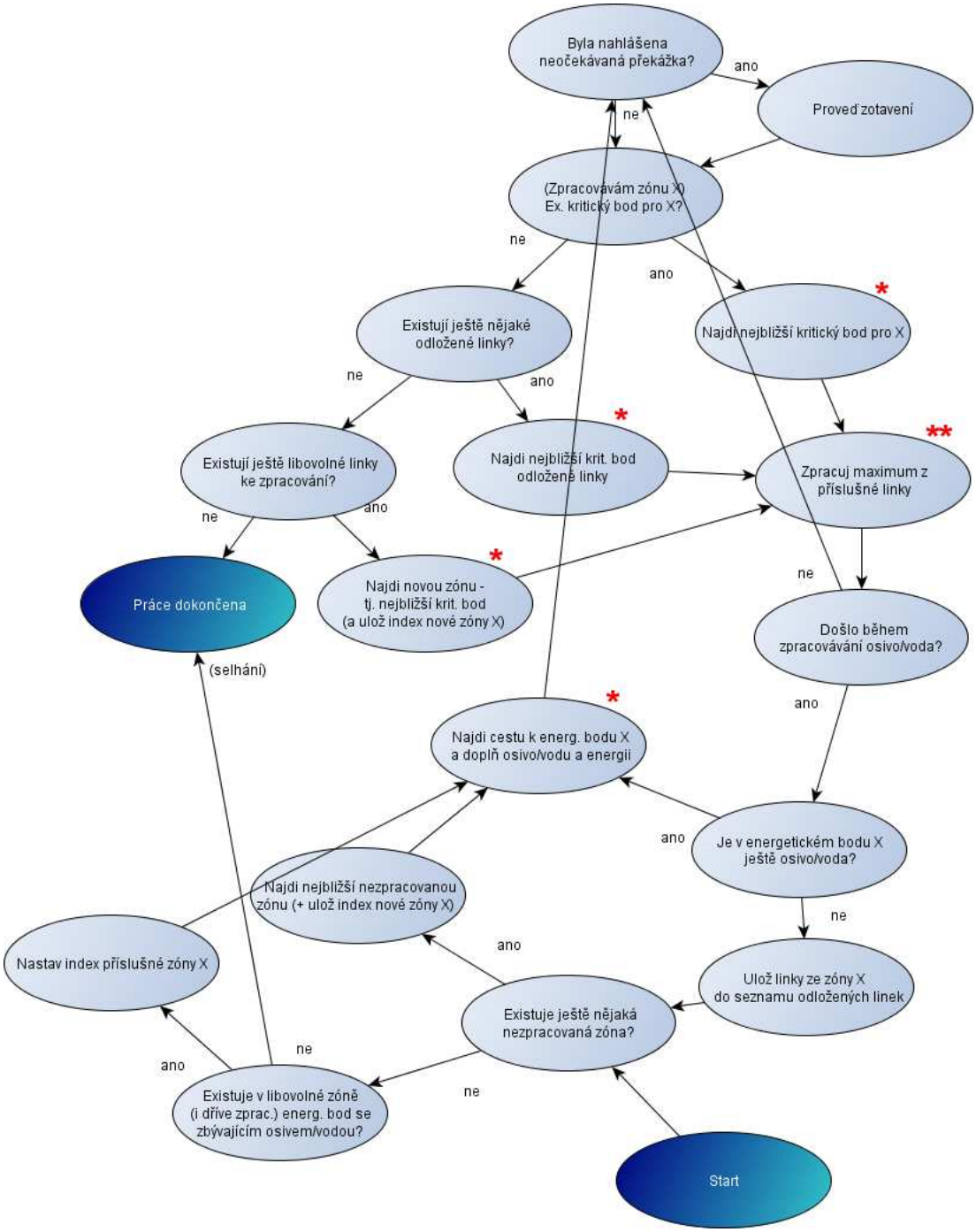
3.4.5 Souhrn algoritmu R4F-C

Diagram na obrázku 31 ilustruje **proces zpracovávání pole** algoritmem R4F-C. Před započnutím první iterace proběhne pre-processing zahrnující přípravu zón pro body energetické obsluhy (viz sekce 3.4.3), kvadrantového stromu, Voroného dekompozici pole na zóny (viz sekce 3.4.1) a vytvoření databáze kritických bodů (viz sekce 3.4.2).

Nejprve se určí počáteční zóna zpracovávání, a pokud jde o program setí či zavlažování, dojede si robot naplnit nádrž danou komoditou.

V každé iteraci algoritmu se zkontroluje, zda robot neoznámil detekci neočekávané překážky – v takovém případě dojde k zotavení (přenastavení proměnných a následnému plánování nové trasy – viz sekce 3.4.4).

Pro aktuálně zpracovávanou zónu se vždy určí nový nejbližší kritický bod a zpracuje se příslušná linka. Pokud není v nádrži dostatek komodit (osiva/vody) pro zpracování dané linky, zpracuje se z ní maximum a robot se následně vydá k bodu energetické obsluhy pro doplnění komodit. Jakmile je zpracována jedna zóna, přechází se na sousední.



Obrázek 31: Diagram znázorňující proces plánování algoritmu R4F-C

Pokud robot vyčerpal veškeré množství komodity v zásobníku bodu energetické obsluhy aktuální zóny a stále máme nezpracované linky pro aktuální zónu, pak se tyto nezpracované linky uloží do seznamu *odložených* linek, tj. linky s indexem -1, nepříslušející k žádnému bodu energetické obsluhy, a robot se je pokusí zpracovat později.

Zůstane-li po zpracování určité zóny nějaká zbývající rezerva dané komodity (v nádrži robota nebo zásobníku bodu energetické obsluhy), využijí se tyto komodity pro zpracování odložených linek (těch s indexem -1) v rámci aktuální zóny.

Pokud po zpracování všech zón stále zůstávají nějaké linky s indexem -1, prohledají se všechny body energetické obsluhy, zda v nějakém nezbyla daná komodita a dojde k případnému dokončení zpracování těchto linek.

Po každé akci se aktualizuje kvadrantový strom (mazání pozic bývalých kritický bodů, příp. přidávání nových krit. bodů) a databáze kritických bodů, tj. nezpracovaných linek (smazání zpracovaných linek či částí linek, přesouvání linek pod „zónu“ -1). Algoritmus skončí ve chvíli, kdy v databázi již nebudou žádné nezpracované linky, budou z pozice robota nedosažitelné, či pokud v poli není dostatek osiva pro zpracování.

Diagram je zjednodušený v několika ohledech. Stavy označené jednoduchou hvězdičkou (*) jsou takové, kdy se před provedením daného úkonu zkontroluje, zda má robot dostatek energie pro provedení dané akce. Pokud ne, najde se příslušný bod energetické obsluhy a robot se dobije. Nicméně, předpokládá se, že komodity budou docházet častěji než energie v akumulátoru a robot se automaticky doplňuje při každém doplnění komodit v bodu energetické obsluhy, takže by v zásadě k uplatnění dané hvězdičky v diagramu nemělo docházet. Dvojitá hvězdička (**) v diagramu pak značí kontrolu úbytku energie, nikoli před provedením daného úkonu, ale v jeho průběhu (zpracuje max. linky a pak se vydá směrem k bodu energetické obsluhy).

Po dokončení plánování program stále přijímá potvrzení od robota a v případě detekce neočekávané překážky dojde k zotavení a znovuspuštění plánovací fáze. Navíc diagram znázorňuje průběh vykonávání pouze jednoho z nastavených programů robota, např. setí – pokud je nastaveno navíc zavlažování, provede se re-inicializace kvadrantového stromu a databáze kritických bodů a algoritmus se spustí nanovo. Při doplňování vody se případné zbývající osivo vysype v aktuálním bodu energetické obsluhy.

V případě zpracovávání pole při programu rytí budou v diagramu stavy týkající se komodit redundantní. Robot bude nadále zpracovávat pole po zónách, nicméně k bodům energetické obsluhy bude jezdit pouze pro dobití akumulátorů.

Pro zjednodušení chybí v diagramu, narozdíl od předchozích přístupů, stav *pošli robotovi ke zpracování*, ale je zřejmé, na kterých místech k tomuto dojde.

3.4.6 Zhodnocení algoritmu R4F-C

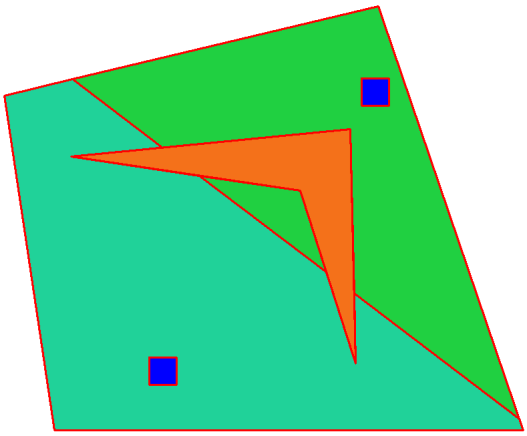
Oproti předcházejícím přístupům, bylo hlavní motivací **zohlednit spotřebu komodit** (osiva/vody). Komodity z každého bodu energetické obsluhy přímo **přidělíme pro jednotlivé regiony** vzešlé z Voroného dekompozice. V optimálním případě (tj. pokud je v jednotlivých bodech dostatek komodit pro zpracovaní příslušných oblastí – dáno kapacitou těchto bodů i jejich rozmístěním v poli) by toto rozdělení pole na zóny mělo mít **příznivý efekt na spotřebu**. Zamezili jsme situaci, kdy by robot byl nucen dojet pro danou komoditu na druhý konec pole, protože všechny bližší body energetické obsluhy by již měly prázdné zásobníky. Máme tedy **větší kontrolu nad spotřebou** komodit.

Myšlenka s *řádkováním* je převzata z druhého algoritmu a zachovává výhody s tím spojené, tj. **uniformní zpracování pole** a **minimalizaci otáčení robota** (a tím pádem nižší spotřeby, neboť otáčení je energ. náročnější operace).

Jak bylo zmíněno, efektivita algoritmu do značné míry **závisí na rozumném rozmístění** bodů energetické obsluhy. Pokud komodity v energetických bodech

často nebudou stačit na zpracování příslušných zón, robot se bude muset později vracet a spotřebovávat další energii. Samozřejmě, že stejně tak mají vliv velikost samotného pole a rychlost spotřeby dané komodity robotem.

Problémem pro aktuální přístup mohou být do značné míry **překážky v poli**. Původní Voroného dekompozice pole s nimi totiž nepočítá a větší překážka může způsobit rozdělení některých zón na více celků – s tím, že některé tyto části nemusí být úplně nejvhodnější zpracovávat v rámci dané zóny. Jak to může vypadat, ilustruje obrázek 32.



Obrázek 32: Překážka v poli (oranžový element) rozdělující jednu ze zón

Dále je třeba říct, že pro program **rytí** bude aktuální přístup spíše **nevhodný**. Při tomto programu žádné komodity nedoplňujeme a dekompozice na zóny ztrácí význam. Robot se pouze v případě potřeby dopraví k nejbližšímu bodu energetické obsluhy pro dobití akumulátoru. Předpokládá se, že minimálně pro program rytí by měl být výhodnější předcházející algoritmus.

Možností pro budoucí **rozšíření** algoritmu je po Voroného dekompozici určit pro jednotlivé regiony, zda by nebylo lepší rozložit pole na linky v odlišném, tj. vertikálním, směru. Jde nám pochopitelně o další maximalizaci rovných linek a minimalizaci otáčení, pro další zlepšování spotřeby robota.

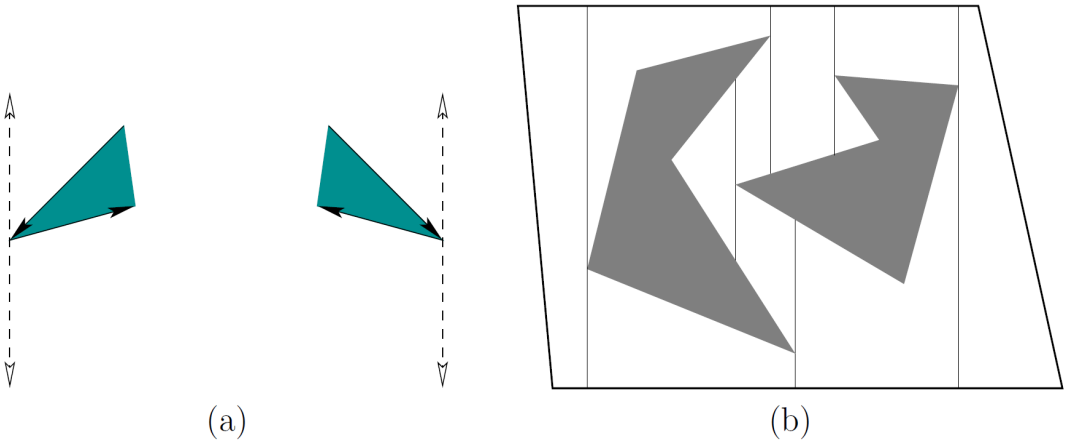
3.5 Alternativní přístupy

Krátce si představíme dva přístupy užívané při řešení typu problému podobného tomu našemu. Jde o tzv. ***plánování pokrytí*** (angl. *coverage planning*), kde je úkolem robota (nebo jakkoli danou entitu nazveme) projet danou oblast tak, že na své cestě navštíví každou oblast vymezené plochy. Aplikací je celá řada, od automatického vysavače místností po lakování automobilů. Nalezení optimální cesty je NP-těžký problém [4] a na řadu tak přicházejí aproximační metody či heuristiky. Ani jeden z následujících přístupů však nebude vhodný pro řešení našeho konkrétního problému a vzápětí si řekneme proč.

3.5.1 Přístup „tažného vola“

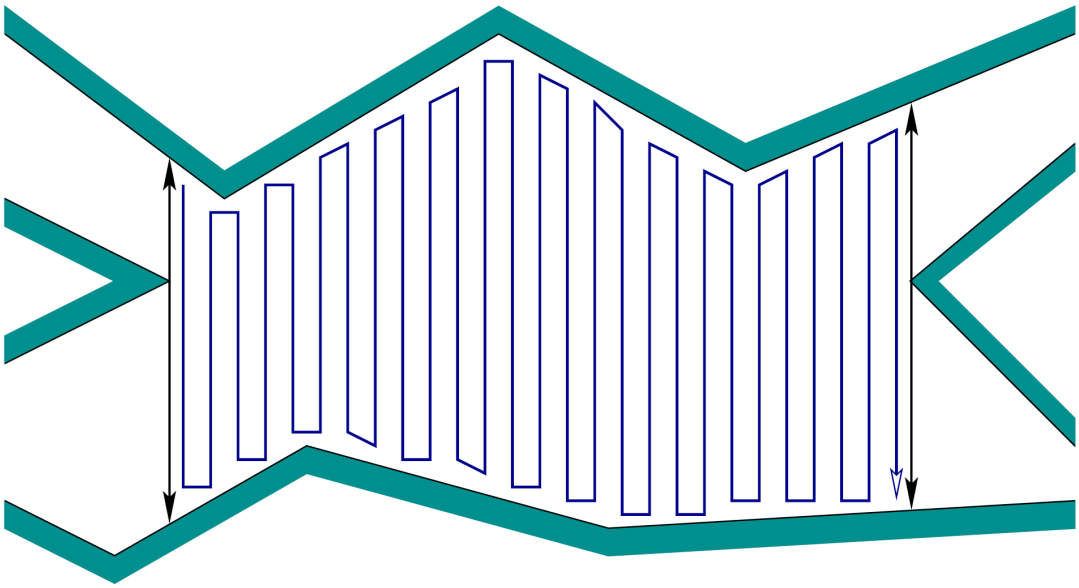
Jedním z přístupů je pro danou oblast užít tzv. ***bustrofedónovou dekompozici*** (angl. *Boustrophedon decomposition*; kde *bustrofedón* je řecký výraz pro „obracení

volů“, tj. způsob orby pole za pomocí tažného dobytka).



Obrázek 33: Bustrofedónová dekompozice pole (šedivé elementy značí překážky)

Pomocí *zametání roviny* přímkou vytvoříme dekompozici jako máme na obrázku 33 b) (převzato z [4]). Při zametání se přitom rozlišují dvě situace, při které se vytvoří sečná úsečka, ty jsou znázorněny na obrázku 33 a) (převzato z [4]). Získané oblasti (buňky) nyní budeme jednu po druhé zpracovávat – zbývá zjistit v jakém pořadí. Vytvoří se graf sousednosti, kde vrcholy představují jednotlivé buňky. Na tomto grafu se pak řeší standardní problém obchodního cestujícího. Jednotlivé buňky se přitom zpracovávají **stylem „tažného vola“**, tj. pohyb ilustrovaný na obrázku 34 (převzato z [4]). Cílem je pochopitelně minimalizovat počet otáčení jakožto nákladné operace.



Obrázek 34: Zpracovávání pole stylem „tažného vola“

Daný přístup bohužel **není příliš vhodný** pro náš konkrétní problém. Celá idea projet pole optimálně (v zásadě *jedním tahem*) je nám k ničemu, neboť přímo ve specifikaci máme uvedenu jako základní premisu fakt, že robot nebude mít dostatek energie pro zpracování celého pole, nehledě na doplňování komodit a potenciální detekci neočekávaných překážek.

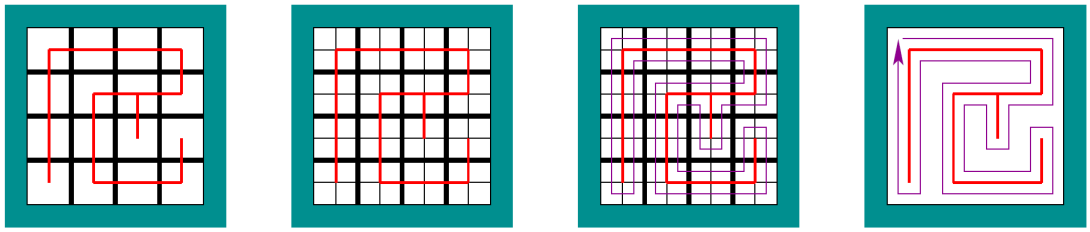
Ilustrovaného „cik-cak“ pohybu však ve většině případů dosáhneme námi navrženými algoritmy R4F-B (řádkový přístup) a R4F-C (zónový přístup).

3.5.2 Pokrývání kostrou grafu

Zajímavou aproximační metodou je tzv. ***pokrývání kostrou grafu***. Zpracovávanou oblast dekomponujeme na čtvercovou mřížku (v podstatě obdoba samplování v sekci 3.2.1), kde hrana jednotlivých čtverců má velikost dvojnásobnou než je nástroj robota. Z mřížky si vytvoříme graf umístěním vrcholů do středu každého čtverce a hran spojujících každé dva z těchto vrcholů, jestliže příslušné čtverce jsou sousedy (v horiz. a vert. směru).

Na získaném grafu najdeme **kostru grafu** – což je souvislý podgraf, který obsahuje každý z vrcholů původního grafu a neobsahuje kružnici [5]. Takových koster lze najít mnoho na základě různých preferencí uživatele.

Cestu pro robota vytvoříme tak, že bude **opisovat danou kostru grafu** a robot se po projetí celého pole vrátí do svého počátečního bodu. Celý proces ilustruje obrázek 35 (převzatý z [4]).



Obrázek 35: Pokrývání kostrou grafu

Stejně jako v předcházející metodě dostaneme tzv. úplné pokrytí, tzn. že robot se dostal na každou část pole. Existují ještě heuristické metody, které však nezajišťují žádnou záruku kompletního zpracování oblasti. Jde o metodu aproximační, neboť určité části pole zůstanou nezpracované vlivem zavedené dekompozice oblasti na čtvercovou mřížku.

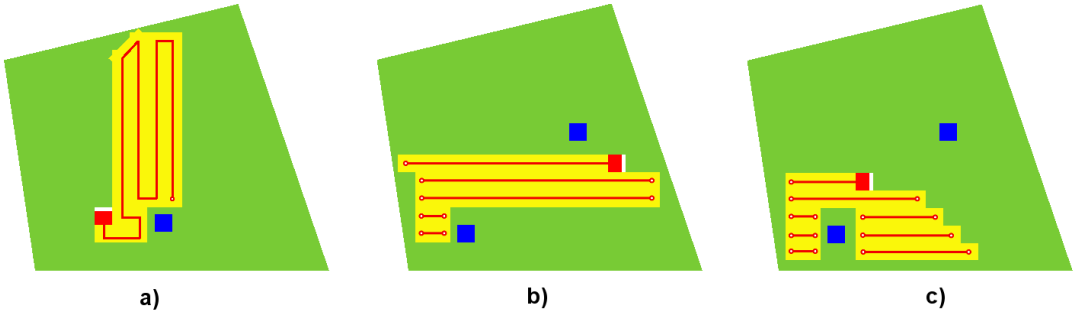
Ani tato metoda však pro řešení našeho problému **není vhodná**, ze stejných důvodů popsaných u předchozí metody, ačkoli neočekávané překážky v poli by řešitelné byly – existuje totiž on-line verze algoritmu, která buduje kostru grafu během zpracovávání pole robotem (algoritmus popsaný v práci [3]). Navíc by se zde opět projevil nedostatek popsaný u algoritmu R4F-A (rozhodovací přístup) v sekci 3.2.9 (tvorba fragmentů).

3.6 Analýza – porovnání navržených algoritmů

Na závěr třetí kapitoly si porovnáme fungování a výkonnost námi navržených algoritmů (rozhodovací, řádkový a zónový přístup) na základě různých typů vstupních polí. Ty se budou lišit velikostí, tvarem, počtem překážek, rozmístěním bodů energetické obsluhy apod.

3.6.1 Způsob zpracovávání pole

Obrázek 36 ilustruje průběh zpracovávání pole jednotlivými algoritmy. Vidíme, že rozhodovací algoritmus (obr. 36 a)) bude jen **obtížně kontrolovatelný** a v zásadě přesně nevíme, jakým způsobem bude výsledné pole zpracováno. U řádkového a zónového algoritmu (obr. 36 b), c)) naopak již na počátku víme, jak bude výsledné pole vypadat, a navíc máme zajištěnou **uniformnost výsledného zpracování**, což může být v praxi důležité, neboť např. chaoticky rozorané pole může být dále špatně zpracovatelné při setí apod.



Obrázek 36: Zpracovávání pole jednotlivými algoritmy

3.6.2 Testy výkonnosti

Pro účely testů byla vymodelována sada polí menších a středních velikostí, s různým počtem a velikostí překážek. V tuto chvíli nebudeme brát v úvahu možný výskyt neočekávaných překážek v poli – předpoklad je takový, že by se měly vyskytovat jen výjimečně a neměly by disponovat přehnanou velikostí, tudíž nebudou mít pro běh jednotlivých algoritmů zásadní dopad. Úspěšnost algoritmů budeme posuzovat dle množství využité energie pro zpracování daného pole a dále dle množství času nutného pro zpracování daného pole.

Stanovena byla jednotná velikost robota i velikost a poloha jednotlivých nástrojů robota – ty budou vždy na středu robota, abychom bez obav mohli využít rozhodovacího algoritmu (daným problémem se zabývala sekce 3.2.9).

Výsledky testů shrnují následující řádky. Referencí bude vždy první ze zmíněných algoritmů, tedy rozhodovací přístup.

Test 1

Předpokládejme, že robot je schopen zpracovat pole bez dobíjení, na jedno naplnění nádrže příslušnou komoditou. Čili výsledek nebude záviset na probíhajícím programu robota (rytí/setí/zavlažování) a k zhodnocení daného tak použijeme pouze operaci rytí.

Řádkový algoritmus se ukázal být vždy o **1 až 3 % energeticky efektivnější** než rozhodovací algoritmus a vždy alespoň o 1 % efektivnější než zónový přístup. Potvrdilo se, že zónovému přístupu nesvědčí překážky, ty totiž štěpí dané zóny a znepříjemňují robotu zpracovávání pole. Přesto byl zónový přístup ve většině případů efektivnější než rozhodovací přístup. Řádkový přístup vyhrává i po rychlostní stránce, kde dokáže rozhodovací algoritmus porazit **až o hodnotu 10 %**. Zónový přístup je naopak nejpomalejší.

Test 2, test 3

Typicky bude komodita (tj. osivo/voda) docházet rychleji než energie v akumulátoru robota a algoritmy jsou implementovány tak, že s doplňováním komodity se automaticky doplní i energie. Nebude tedy příliš často docházet k tomu, že by se robot vydal k bodu energetické obsluhy čistě kvůli nedostatku energie – až na případ programu rytí, kterému tak bude věnován i druhý test (tentokráte s pravidelným dobíjením).

Analogický je třetí plánovaný test, kdy budeme osévat (příp. zavlažovat) s tím, že neřešíme množství komodit v bodech energetické obsluhy – tzn. že klidně celé pole může být oseto osivem z jediného bodu energetické obsluhy.

Testy potvrdily podobné výsledky pro druhý i třetí test, kdy řádkový algoritmus je konzistentně **o 2 % energeticky efektivnější** než referenční rozhodovací algoritmus, zatímco zónový algoritmus je naopak často méně efektivní než rozhodovací algoritmus. Časově je na tom nejlépe taktéž řádkový algoritmus, údaje kolísají od **0 až 5 %** ve prospěch tohoto algoritmu. Zónový algoritmus i zde selhává a prohrává s referenčním algoritmem.

Test 4, test 5

Nakonec otestujeme situaci, kdy je v bodech energetické obsluhy dostatek komodity „tak akorát“ na zpracování celého pole. Efektivita zpracování samozřejmě závisí na rozmístění jednotlivých bodů energetické obsluhy. Ve čtvrtém testu rozmístíme zmíněné body libovolně, v pátém se budeme snažit rozmístit je optimálně – tj. tak, aby zóny příslušející jednotlivým bodů energetické obsluhy byly pokud možno stejně velké.

Dle předpokladů špatné rozmístění bodů energetické obsluhy mělo za následek zhoršení výkonnosti zj. u zónového algoritmu. Nicméně, i při optimálním rozestavění dosáhl zónový přístup **jen výjimečně lepších výsledků než řádkový algoritmus**. Referenční rozhodovací algoritmus zde prohrává na celé čáře, v průměru je o 3 % méně efektivní než řádkový algoritmus a o 1 % méně efektivní než zónový algoritmus. Pokud jde o celkový čas zpracovávání pole, koresponduje zde s energetickou efektivitou.

Souhrn

Jasným **vítězem testů se stal řádkový algoritmus** a je tak nejlepším kandidátem pro další vývoj (potenciální možnosti rozšíření uvádíme v zhodnocení daného přístupu v sekci 3.3.5). V testech stabilně porážel zbylé dva algoritmy o několik procentních bodů ve smyslu energetické efektivity.

Naopak **zónový algoritmus nenaplnil očekávání**. Ukázalo se, že dokáže být výhodnější pouze za dosti specifických podmínek a to ještě ne ve všech případech. Je to dáno tím, že vynucené postupné zpracovávání pole po zónách vyžaduje příliš velkou energetickou „režii“, která ve většině případů převáží výhody plynoucí z charakteru tohoto zpracovávání (tj. robot má jistotu, že nemusí jezdit daleko pro doplnění komodity).

Rozhodovací přístup je použitelný, nicméně přetrvávají již zmíněné **nevýhody** (neuniformně zpracované pole, možné fragmentování při specifické poloze nástroje).

4. R4Farmer – uživatelská a programová dokumentace

4.1 Uživatelská dokumentace

4.1.1 Program R4Farmer

Program R4Farmer je součástí projektu R4Farmer. Jde o konfigurační a simulační nástroj pro systém autonomního robotického zemědělce. Detailnější informace o projektu viz předchozí kapitoly.

4.1.2 Instalace programu

Na CD s programem se na nachází jednoduchý instalační program (pro platformu MS Windows), který uživatele provede celým procesem instalace aplikace R4Farmer. Pro spuštění instalace otevřete průzkumníka a spusťte soubor *setup.exe* umístěný v kořenovém adresáři daného CD. Program se následně spustí souborem *r4farmer.exe* umístěném v adresáři zadaném při instalaci, případně vytvořeným zástupcem na ploše.

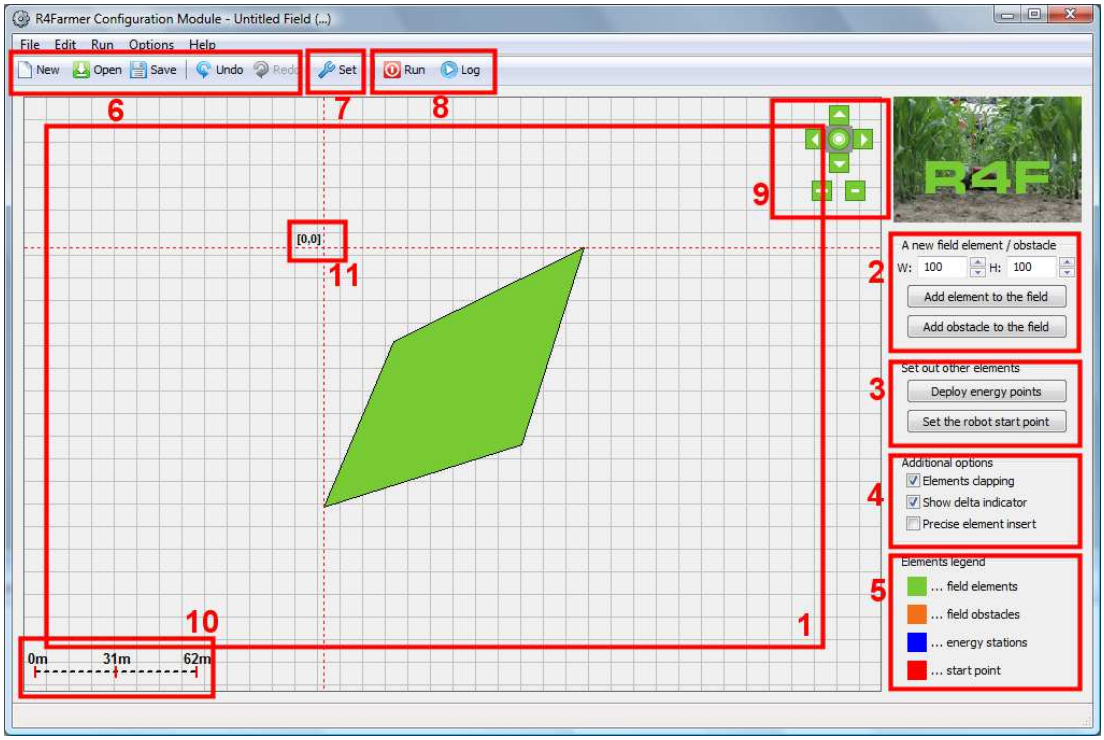
4.1.3 Konfigurační modul

Po spuštění programu se uživatel ocitne v Konfiguračním modulu aplikace. Ta slouží pro modelování polí určených ke zpracování robotem a nastavení všech nutných parametrů programu robota. Pro začátek si popíšeme **ovládací prvky prostředí** daného modulu (budeme odkazovat na očíslování jednotlivých prvků na obrázku 37). K bližšímu použití jednotlivých prvků se dostaneme vzápětí.

Největší plochu zaujímá samozřejmě modelovací plocha (1) umožňující editovat jednotlivé prvky pole. Pravý panel pak nabízí možnosti pro přidání elementů pole a elementů překážek zvolených rozměrů (2), dále rozestavění bodů energetické obsluhy a startovního bodu robota (3), nastavení vlastností aktuálního modelování (4) a legendu (5) znázorňující význam barev jednotlivých elementů pole na modelovací ploše.

Horní panel (*toolbar*) nabízí možnost odstartovat modelování nového pole, nahrání a uložení nějakého pole z/do souboru (koncovka „.r4f“) a undo-redo volby pro případné navrácení nějaké akce (6), dále tlačítko *Set* (7) dává k dispozici formulář pro nastavení všech důležitých parametrů programu robota, tlačítko *Run* (8) poskytuje volby pro samotné spuštění práce robota a konečně tlačítko *Log* (8) umožňuje přehrávat dříve zaznamenané logy, tj. záznamy o práci robota na poli (soubory s koncovkou „.r4log“).

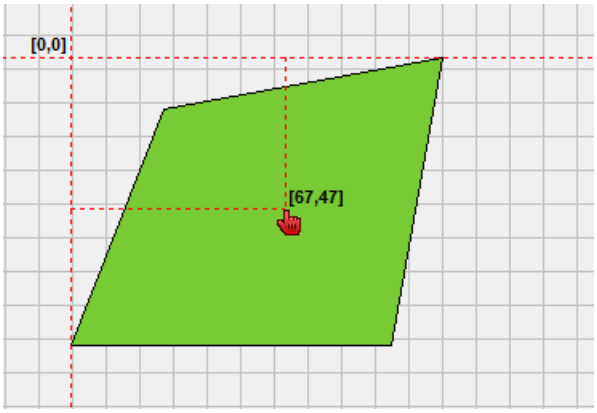
Modelovací plocha nabízí jednoduchou navigaci (9) ve formě posouvání obrazu či zoomování (to je možné i kolečkem myši). Aktuální přiblížení ukazuje měřítko v levém dolním rohu modelovací plochy (10). K lepší orientaci na plánku pole slouží ukazatel tzv. virtuálního počátku souřadnic (11) – půjde vždy o minimální levý horní roh aktuálního pole.



Obrázek 37: Konfigurační modul programu R4Farmer

4.1.4 Konfigurační modul – modelování pole

Pokud chceme začít modelovat pole, stačí v pravém panelu navolit počáteční rozměry daného obdélníku (č. 2 na obr. 37) a **přidat příslušný prvek** na modelovací plochu. Obdobně přidáme novou překážku do pole. Elementy pole mají světle zelenou barvu, zatímco překážky v poli jsou oranžové.

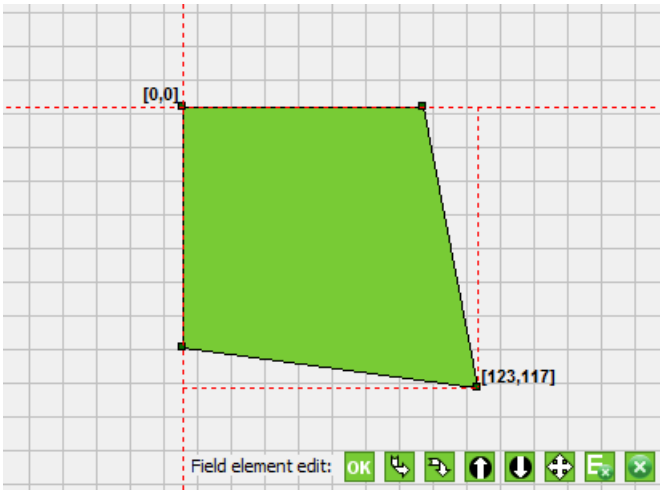


Obrázek 38: Aktivní režim přidávání nového startovního bodu

Do vytvořeného pole lze následně **přidávat body energetické obsluhy** a startovní bod robota. Po kliknutí na příslušné tlačítko v pravém panelu (č. 3 na obr. 37) se spustí režim přidávání daného elementu, v pravém dolním rohu modelovací plochy se objeví další možnosti právě aktivního režimu (užitečné především při následném editování prvků), změní se barva kurzoru dle režimu a ukazuje se aktuální poloha vůči zmíněnému virtuálnímu počátku souřadnic (v metrech) – viz obrázek 38.

Modrou barvu má kurzor při přidávání či editaci bodů energetické obsluhy, červenou při manipulaci se startovním bodem robota. Ukazování aktuální polohy kurzoru (při přidávání prvků) či elementů (při editaci prvků) vůči počátku lze vypnout odškrtnutím příslušné volby v pravém panelu (č. 4 na obr. 37). Daný režim přidávání/editování lze zrušit kliknutím na tlačítko *OK* v dolním panelu.

Tvar přidaných elementů pole a překážek v poli **lze dále měnit**. Stačí kliknout na daný element, čímž element získá fokus a uživatel je v editovacím režimu pro daný prvek (viz obrázek 39). Pak již stačí za daný element táhnout kurzorem a libovolně ho přesouvat či zatáhnout za jeden ze čtyř rohů a pozměnit původní obdélník do libovolného tvaru. Další možnosti editace v dolním panelu jsou rotace, přesunutí vpřed či vzad (neboť elementy se mohou překrývat), přesný posun libovolným směrem, smazání všech bodů energetické obsluhy asociovaných s tímto elementem či úplné smazání elementu.



Obrázek 39: Editování elementu pole

Možnosti editace jsou analogické i pro ostatní prvky (body energetické obsluhy a startovní bod) – lze je libovolně přemisťovat, či smazat.

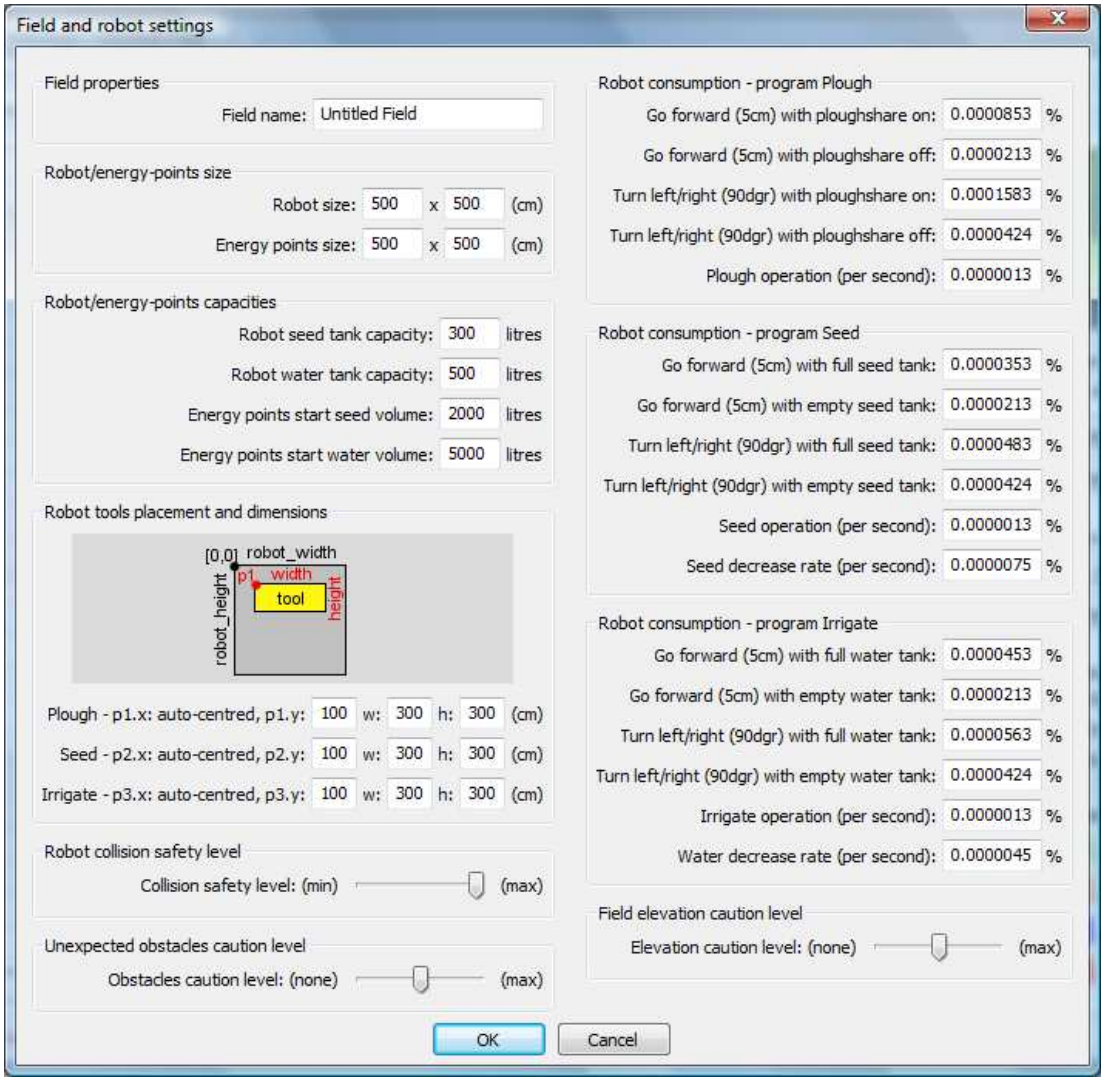
Elementy pole a překážky lze napojovat vedle sebe nebo i přes sebe. Pokud k sobě přiblížíme dva elementy s vodorovnou či svislou hranou, dojde k efektu „doklapnutí“, který umožňuje přesné napojování daných elementů. Tuto funkci lze vypnout v pravém panelu (č. 4 na obr. 37).

Libovolné prvky lze přitom přidávat přesně vůči aktuálnímu počátku souřadnic – stačí opět zaškrtnout příslušnou volbu v pravém panelu (č. 4 na obr. 37). Při přidávání libovolného prvku pak vždy vyskočí formulář, kde uživatel zadá souřadnice nového prvku.

Všechny operace lze libovolně vracet pomocí undo-redo funkcí. Ty jsou uživateli k dispozici v horním panelu (č. 6 na obr. 37).

4.1.5 Konfigurační modul – nastavování parametrů

Po kliknutí na *Set* v horním panelu (č. 7 na obr. 37) dostane uživatel k dispozici formulář pro **specifikaci detailů plánovaného programu robota** (viz obrázek 40). Může nastavit rozměry robota i bodů energetické obsluhy, kapacitu nádrže robota pro osivo a vodu a stejně tak startovní množství osiva a vody



Obrázek 40: Nastavování parametrů programu robota

v bodech energetické obsluhy. Následuje možná specifikace nástrojů robota, tzn. jejich velikost a přesné umístění na spodku robota.

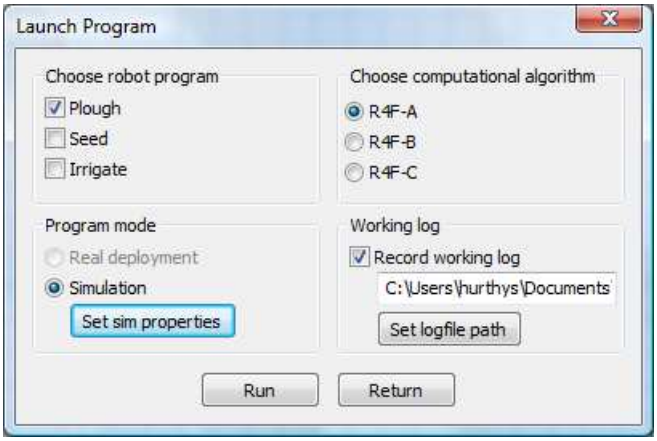
Dále má uživatel k dispozici nastavení spotřeby jednotlivých operací robota i rychlost úbytku osiva/vody. Nakonec obsahuje formulář tři táhla po specifikaci míry obezřetnosti vůči kolizím, vůči neočekávaným překážkám a výškovým nerovnostem.

Všechny hodnoty jsou po stisku tlačítka *OK* kontrolovány a v případě nevalidních hodnot je uživatel upozorněn a požádán o nápravu.

4.1.6 Konfigurační modul – spuštění programu robota

Pro spuštění programu stačí stisknout tlačítko *Run* na horním panelu (č. 8 na obr. 37). Aplikace v tu chvíli zkontroluje, zda vymodelované pole splňuje základní požadavky – tzn. je kompaktní (nejsou dovoleny izolované elementy pole), je nastaven startovní bod robota a alespoň jeden bod energetické obsluhy.

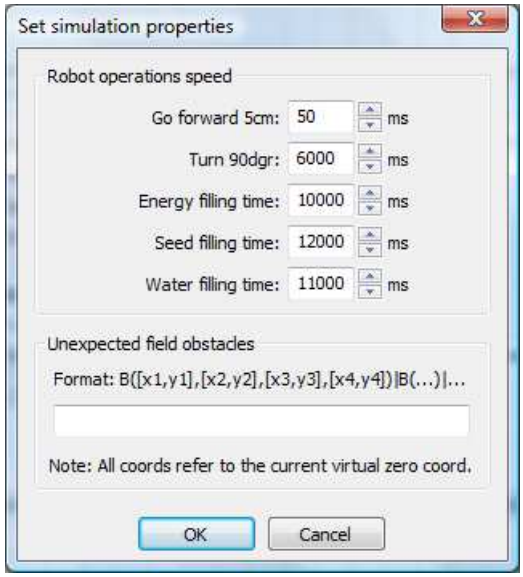
Pokud je pole validní, zobrazí se uživateli spouštěcí formulář, kde dojde k dalšímu **upřesnění právě spouštěného programu robota** (viz. obrázek 41). Uži-



Obrázek 41: Nastavování parametrů při spouštění programu

vatel si zvolí, jaké operace chce na poli provést, zvolí jeden z možných algoritmů, zda chce ukládat log činnosti a případně nastaví **vlastnosti simulace** kliknutím na tlačítko *Set sim properties*.

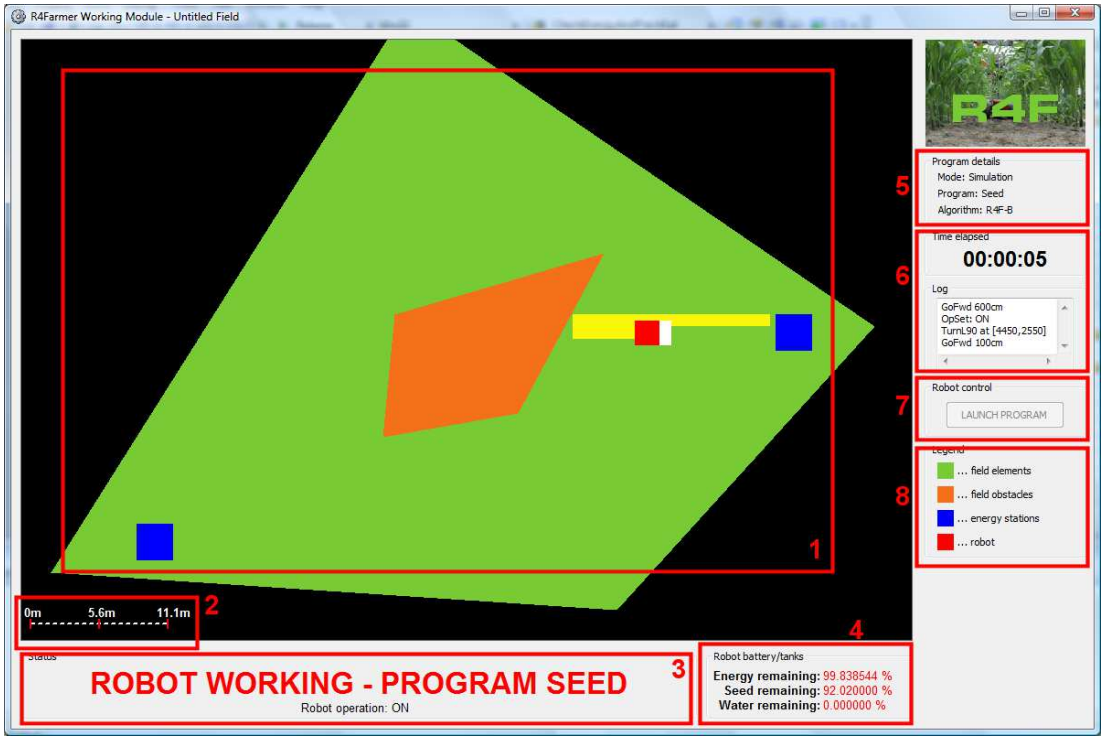
Uživatel v takovém případě dostane další formulář (viz obrázek 42), kde může nastavit rychlost jednotlivých operací robota, případně může určit souřadnice neočekávaných překážek v poli. Je třeba mít na paměti, že dané souřadnice platí vzhledem k aktuální pozici virtuálního počátku souřadnic zmíněného výše.



Obrázek 42: Nastavování vlastností simulace

4.1.7 Pracovní modul

Při spuštění programu robota se zavře Konfigurační modul a aplikace přejde do tzv. Pracovního modulu. Ten slouží pro **účely vizualizace** práce robota, **účely simulace** a **plánování** samotné práce robota na poli. Opět si krátce představíme uživatelské rozhraní (budeme odkazovat na očíslování jednotlivých prvků na obr. 43).



Obrázek 43: Pracovní modul programu R4Farmer

Rozhraní Pracovního modulu se skládá z pracovní plochy (1), která poskytuje opět základní navigaci – uživatel si může posunout pohled táhnutím myši či přiblížit pohled kolečkem myši. Aktuální zvětšení indikuje měřítko (2) v levém dolním rohu pracovní plochy. Dolní panel ukazuje aktuální stav programu (3) včetně toho, zda je aktuální nástroj robota (rycí/secí/zavlažovací) aktivní, či nikoli. Stejně tak má uživatel k dispozici informace o aktuálním množství zbývající energie robota a daných komodit v nádržích robota (4). Pravý panel nabízí jednak informace o probíhajícím programu robota (5), uplynulý čas a log poslední činnosti (6), legendu objasňující význam barev jednotlivých prvků (8) a tlačítko *Launch Program* pro samotné odstartování práce robota (7).

4.1.8 Pracovní modul – práce robota na poli

K odstartování práce robota tedy dojde stisknutím tlačítka **Launch Program**. V té chvíli navigační jádro začne s **plánováním** dalších kroků robota, posílá příkazy robotovi a aktuální situace je reflektována na jednotlivých prvcích uživatelského rozhraní Pracovního modulu. Uživatel si může v průběhu práce libovolně přibližovat a posouvat pole s pracujícím robotem. Celý proces se dá jednoduše přerušit zavřením okna Pracovního modulu.

Jakmile dojde k **úspěšnému dokončení** programu robota, aplikace vydá zprávu o celkovém uplynulém času a množství spotřebované energie a spotřebovaných komodit (osiva/vody). Program robota však může i z mnoha důvodů **selhat**. V takovém případě aplikace vydá zprávu o příčině selhání. Možné příčiny selhání jsou: absolutní vybití akumulátoru robota, příliš vysoká spotřeba operací (některé oblasti pole jsou příliš daleko a nelze je s aktuálním nastavením zpracovat), nedosažitelný žádný bod energetické obsluhy s požadovanou komoditou,

nebo nedosažitelný žádný bod energetické obsluhy.

Během zpracovávání pole může dojít k **detekci neočekávaných překážek** v poli. V takovém případě se nově detekovaná překážka zobrazí na pracovní ploše. Robot následně přeplánuje budoucí kroky a pokračuje v činnosti.

Pole je úspěšně zpracované v momentě, kdy všechny dosažitelné úseky pole jsou zpracované – pokud by nějaká neočekávaná překážka izolovala část pole, pak tato část zpracována nebude. Robot nesmí za žádných okolností opustit hranice pole.

Pokud uživatel před spuštěním programu zvolil možnost zaznamenávání **logu činnosti** robota, může si následně daný log (soubor s koncovkou „.r4log“) zrychleně přehrát v tzv. přehrávači logu, což je pouze jiný režim Pracovního modulu, a lze ho spustit z Konfiguračního modulu (č. 8 na obr. 37).

4.2 Programová dokumentace

4.2.1 Programovací jazyk, použité knihovny

Aplikace je implementována za použití programovacího jazyka C++ a multiplatformních freeware open-source knihoven **wxWidgets** ve verzi 2.8.11. Příslušný balík lze nalézt na webových stránkách[1] projektu nebo na přiloženém CD k této práci. Podrobný návod, jak zprovoznit knihovny wxWidgets pod různými prostředími a platformami, lze rovněž nalézt na stránkách projektu[2].

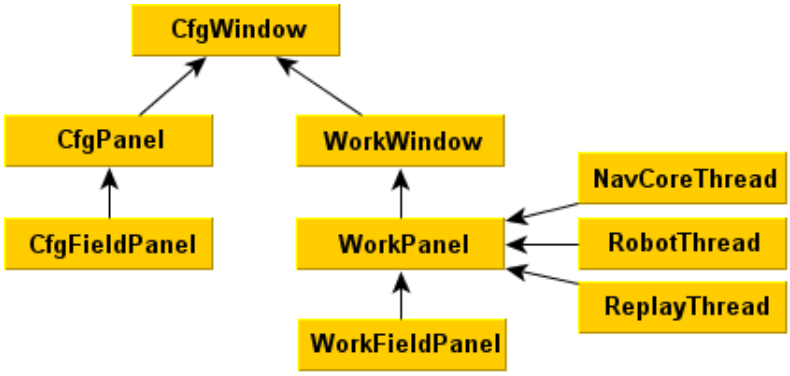
Pro složitější operace nad polygony je dále použita externí freeware knihovna *Clipper* autora Anguse Johnsona[3] a pro generování Voroného digramu volně šiřitelná C++ implementace Shana O'Sullivana[4] původního algoritmu navrženého Stevenem Fortunem[5].

Primární platforma je MS Windows, přičemž zdrojové kódy jsou psány v prostředí MS Visual Studia 2008. Základní portabilita pro unixové systémy sice byla zajištěna – program je zkompilovatelný a spustitelný – nicméně aplikace nebyla dosud řádně odladěna.

4.2.2 Celkový koncept programu

Diagram na obrázku 44 ukazuje závislost jednotlivých vrstev aplikace. Třídy s prefixem *Cfg* reprezentují Konfigurační modul a třídy s prefixem *Work* přísluší Pracovnímu modulu. Třídy `NavCoreThread`, `RobotThread` a `ReplayThread` patří pod Pracovní modul, ale nemají příslušný prefix, neboť jde o samostatná vlákna pracující izolovaně od hlavního programu a GUI. Pokud v následujícím textu budou zmíněny nějaké třídy s prefixem *wx*, jde o třídy knihoven wxWidgets, jejichž on-line dokumentaci lze nalézt na oficiálních stránkách projektu.

[1] `http://www.wxwidgets.org/`
[2] `http://wiki.wxwidgets.org/Guides_&_Tutorials`
[3] `http://www.angusj.com/delphi/clipper.php`
[4] `http://www.skynet.ie/~sos/mapviewer/voronoi.php`
[5] `http://ect.bell-labs.com/who/sjf/`



Obrázek 44: Diagram závislostí klíčových tříd (vrstev) aplikace

4.2.3 Konfigurační modul

Třída `CfgWindow` zastřešuje Konfigurační modul a spouští Pracovní modul. Rozšiřuje při tom třídu `wxFrame`, která vytváří samotné okno aplikace. `CfgWindow` definuje a obsluhuje všechna tlačítka v horním panelu (toolbaru) i všechny položky menu a zajišťuje překreslování všech těchto prvků. Rovněž spouští formulář pro nastavení parametrů projektu (třída `SettingsDialog`) a formulář pro nastavení parametrů programu robota před samotným spuštěním (třída `LaunchDialog` – z té se pak ještě spouští další formulář pro nastavení vlastností simulace, tj. třída `SimPropertiesDialog`).

Všechna tato data pak, společně s daty týkající se vymodelovaného pole, umožňuje uložit do souboru a také zpětně nahrát ze souboru. Vektorová data jednotlivých elementů pole tato třída nemá k dispozici, ale v případě potřeby je získá od panelu `CfgFieldPanel`, nebo naopak může této komponentě poslat nová data. Soubory s vymodelovanými poli jsou uloženy v relativně čitelné formě a lze je editovat i ručně. Koncovka těchto souborů je „.r4f“.

Třída `CfgPanel` reprezentuje boční panel Konfiguračního modulu. Komunikuje s `CfgFieldPanelem` (modelovací plochou) především tak, že mu posílá, jaké elementy a kam na modelovací plochu přidat. Zaškrtnutím příslušných voleb na tomto panelu rovněž indikuje `CfgFieldPanelu`, zda je povoleno zobrazování tzv. virtual delty (aktuální posun vůči virtuálnímu počátku souřadnic) a zda je povoleno „doklapávání“ elementů pole. `CfgPanel` rovněž spouští formulář pro přesné přidávání prvků do pole na určenou pozici, pokud je tato volba zaškrtnuta v pravém panelu (jde o třídu `PreciseAddDialog`).

`CfgFieldPanel` reprezentuje samotnou modelovací plochu. Má v sobě uložena veškerá vektorová data právě modelovaného pole. Tato data jsou následně transformována na základě aktuálního posunutí a přiblížení a vykreslována na danou plochu handlerem `OnPaint()`. Navigace na ploše je zajišťována zmiňovanými parametry posunutí a přiblížení, ty lze měnit buď příslušnými tlačítky v pravém horním rohu modelovací plochy, zoomovat pak lze i kolečkem myši. U jednotlivých akcí na ploše je pak nutné zpětně transformovat dané souřadnice do původní reprezentace.

Elementy pole i překážkové elementy jsou reprezentovány totožně (polygon o čtyřech bodech), odlišeny jsou pouze jednobitovým příznakem. Jednotlivé body energetické obsluhy jsou asociovány vždy právě s jedním elementem pole a jejich poloha je relativní vzhledem k danému elementu. To samé platí pro startovní bod

robota. Pro přidávání všech prvků má `CfgFieldPanel` připraven interface (pro komunikaci s `CfgPanel` a `CfgWindow`).

Akce na ploše jsou prováděny výhradně pomocí myši, čili většinu editování zajišťují handlery pro obsluhu myši a návazné funkce. Program se může nacházet v několika různých módech – buď jde o přidávání nějakého prvku, nebo editaci nějakého prvku. Uživatel toto zjistí díky zobrazení editovacího/přidávacího panelu v pravém dolním rohu pracovní plochy, v kódu je toto indikováno hodnotou proměnné `edit_mode`. Do editovacího módu nějakého prvku se uživatel dostane získáním jeho fokusu (kliknutím na příslušný element). Pokud právě neprobíhá editace nějakého prvku, je hodnota proměnné `focus` na -1.

Každá akce na modelovací ploše je zaznamenána pro potřeby implementace funkcí undo/redo. Jednotlivé akce mají své identifikátory a doplňkové parametry a jsou skladovány (inverze k těmto operacím) v *undo zásobníku* a *redo zásobníku*. Pokud uživatel klikne v toolbaru na tlačítko undo, vytáhne se daný záznam akce ze zásobníku a pošle se interpretační funkci, která rozhodne, jaká akce a jakým způsobem se má provést. Jednotlivé záznamy se mezi undo/redo zásobníky dle potřeby přehazují (zde opět nutné nejprve získat inverzní operaci k dané operaci), nebo mažou.

Pokud je v pravém panelu povolena funkce „doklapávání“ elementů, jsou vodorovné a svislé hrany elementů pole a překážek při přesouvání kontrolovány, a pokud se přiblíží na dostatečně malou vzdálenost, „doklapnou“ k sobě.

`CfgFieldPanel` používá tzv. virtuální počátek souřadnic pro lepší orientaci při modelování pole. Ten je definován jako minimální horní levý roh aktuálního pole a v případě potřeby se jeho poloha aktualizuje. Souřadnice všech elementů jsou pak relativní vzhledem k tomuto počátku souřadnic – alespoň takový je pohled uživatele. Ve skutečnosti jsou souřadnice všech prvků pole úplně jiné, ale všechny akce na modelovací ploše, přidávání nových prvků do pole apod. se dělají právě vzhledem k tomuto bodu. Pokud je navíc v pravém panelu zapnuta volba zobrazování posunu vůči virtuálnímu počátku souřadnic (tzv. *virtual delta*), pak má uživatel přehled o přesné poloze každého z prvků (jakmile ho začne editovat) i o poloze nově přidávaného prvku (aktuální souřadnice se zobrazuje hned vedle kurzoru myši). Pokaždé dochází k jednoduchému přepočítání aktuální polohy bodu na ploše vůči zmíněnému virtuálnímu počátku souřadnic.

4.2.4 Pracovní modul

V momentě, kdy uživatel zvolí spuštění programu z Konfiguračního modulu, skryje se dané okno modulu a otevře se nové okno pro Pracovní modul. Toto okno bude instancí třídy `WorkWindow`, která rozšiřuje třídu `wxFrame`. `WorkWindow` při vytváření obdrží informace předávané od Konfiguračního modulu – jde o údaje o vymodelovaném poli, nastavení projektu a nastavení detailů programu robota (uloženo ve strukturách `ProjFieldData`, `ProjSettings` a `LaunchSettings`). Ty se předají dál třídě `WorkPanel`, předtím však dojde k několika úpravám. Souřadnice objektů jsou převedeny z metrů na centimetry pro zvětšení přesnosti budoucích operací a objekty pole jsou posunuty tak, že počátek souřadnic je v levém horním rohu celého pole. Dále jsou případně z dat vyloučeny takové body energetické obsluhy, které nejsou validní (tj. nacházejí se například za překážkou).

Data jsou postoupena třídě `WorkPanel`, která reprezentuje hlavní panel Pra-

covního modulu. `WorkPanel` bude přijímat zprávy (události) od plánovacího jádra a robota, příp. přehrávacího jádra (pracovní vlákna `NavCoreThread`, `RobotThread` a `ReplayThread`) a tyto zprávy bude obsluhovat. Typicky půjde o aktualizaci údajů o právě probíhající práci na pracovní plochu a ostatní ovládací prvky uživatelského rozhraní. `WorkPanel` aktualizuje informace o aktuálním stavu robota, údaje o zbývajícím množství energie a komodit, odpočítává čas od započnutí operace a zobrazuje log poslední činnosti v postranním panelu. Pro aktualizaci informací přímo na pracovní ploše dává `WorkPanel` příkazy příslušnému panelu (`WorkFieldPanel`) přes poskytnuté rozhraní. Pokud uživatel zvolil zaznamenávat log probíhající práce robota, ukládají se veškeré přijaté akce rovněž do souboru (koncovka „.r4log“). Tento soubor (log) je následně možné přehrát v přehrávači logu (viz dále).

Pro samotné odstartování robota slouží tlačítko *Launch Program* v postranním panelu. To dá pokyn k inicializaci a spuštění pracovních vláken `RobotThread` a `NavCoreThread` a započne odpočítávání uplynulého času celé operace.

Pokud robot úspěšně zpracuje celé pole, či dojde k nějakému selhání, třída `WorkPanel` obdrží příslušnou událost a zobrazí krátkou dialogovou zprávu shrnující údaje o proběhlém programu, případně informace o příčinách selhání programu. V případě přerušení programu (tj. prosté zavření okna Pracovního modulu) se nejprve korektně ukončí všechna pracovní vlákna a teprve pak se zavře Pracovní modul a aplikace se vrátí do Konfiguračního modulu.

Třída `WorkFieldPanel`, jak již bylo zmíněno, reprezentuje samotnou pracovní plochu modulu. Navigace je řešena podobně jako na modelovací ploše v Konfiguračním modulu, dochází k překreslování daných dat reprezentující prvky pole transformovaných dle aktuálního posunutí a přiblížení. K překreslování dochází v pravidelných intervalech za užití časovače. Situace na poli se mění prostřednictvím vybudovaného rozhraní, kde `WorkPanel` užívá metody této třídy pro aktualizaci nové konfigurace robota na poli. `WorkFieldPanel` si navíc pamatuje všechny dříve zpracované úseky pole a to tak, že si ukládá příslušné plochy (linky) vznikající při pohybu robota se zapnutým (např. rycím) nástrojem. Všechny tyto linky jsou pak periodicky vykreslovány spolu s dalšími prvky pole.

4.2.5 Pracovní modul – vlákno `RobotThread`

Pracovní vlákno `RobotThread` reprezentuje robota, jde o potomka třídy `wxThread`. Zpracovává příkazy od pracovního vlákna `NavCoreThread` (plánovací jádro), přičemž vzájemná komunikace probíhá skrze dva buffery (příjmový a odchozí) implementované jako fronty se zajištěnou synchronizací (instance třídy `RobotCmdQueue`).

Příkazy přicházejí ve formátu struktury `RobotCmdP`, což je dvojice identifikátor příkazu (výčtový typ `RobotCmd`) a pomocný index (pro upřesnění příkazu, např. jakou vzdálenost má robot ujet). Ve standardním režimu `RobotThread` zkontroluje každý příchozí příkaz, a pokud je validní, pokusí se ho provést a pošle potvrzení zpátky vláknu `NavCoreThread`. (není-li validní, pošle se zpět ihned). Validní příkazy jsou určeny množinou příkazů, které umí robot zpracovat.

Může se však stát, že se validní příkaz nepovede splnit, protože došlo k detekci neočekávané překážky v poli. Daná situace se signalizuje vláknu `NavCoreThread`, pošlou se mu detaily o nově detekované překážce a čeká se na zotavení. V režimu „čekám na zotavení“ robot nezpracovává další příkazy z bufferu, ale posílá je

hned zpátky plánovacímu jádru `NavCoreThread`. K zotavení robota a navrácení do standardního režimu fungování dojde ve chvíli, kdy robot obdrží příkaz o zotavení – v tu dobu již další příkazy na příchozím bufferu budou validní (v tom smyslu, že budou reflektovat nový stav věcí na poli).

Detekce neočekávaných překážek pobíhá v emulované verzi robota tak, že si robot pamatuje svoji pozici v poli a má informace o poloze neočekávaných překážek. Jakmile by vykonání nějakého příkazu způsobilo kolizi s nějakou překážkou, oznámí se detekce této překážky a získají se souřadnice této překážky průnikem detekční kružnice okolo robota s daným polygonem překážky.

Po každém úspěšně provedeném kroku se pošle zpráva (událost) grafickému rozhraní (tj. třídě `WorkPanel`), která následně aktualizuje situaci na poli. Daná událost má přiložený pointer na vytvořený objekt struktury `RobotPasser`, která obsahuje všechny údaje. Příjemce události má následně povinnost tento objekt dealokovat. Stejným způsobem se případně pošlou údaje o detekci neočekávané překážky v poli. Pokud by došlo ke klesnutí úrovně energie robota pod kritickou hranici, tedy faktickému vybití akumulátoru a tedy selhání plánovacího jádra, daná událost se rovněž oznámí `WorkPanelu` a program robota je zastaven.

4.2.6 Pracovní modul – vlákno `NavCoreThread`

Pracovní vlákno `NavCoreThread` reprezentuje plánovací jádro programu robota, opět jde o potomka třídy `wxThread`. Hlavní smyčka tohoto vlákna pracuje ve dvou fázích. V první fázi přijme z odchozího bufferu robota všechna potvrzení o provedení příkazů a aktualizuje údaje o současné konfiguraci robota na poli (k tomu užívá tzv. *reálné proměnné*). V druhé fázi pak probíhá samotné plánování dalších kroků robota (budoucí stav je reprezentován pomocí tzv. *virtuálních proměnných*) a posílání těchto plánů (resp. příkazů) na přijmový buffer robota. Plánování dalších kroků programu robota zajišťují algoritmy popsané výše v kapitole 3.

Jakmile robot v první fázi zaznamená hlášení o detekci neočekávané překážky v poli, přeruší se plánovací fáze, zpracují se všechny příkazy vrácené od `RobotThreadu` a virtuální proměnné se nastaví na hodnoty reálných proměnných. Jakmile je tedy vlákno vypořádáno s danou situací, pošle na přijmový buffer robota zprávu o úspěšném zotavení, spustí se znovu plánovací fáze a program pokračuje. Plánovací fáze plánuje dopředu do volitelné hloubky (omezení konstantou `ROBOT_MAX_INCOMING_QUEUE` – jakmile je překročen počet příkazů na příchozím bufferu robota, plánovací fáze se pozastaví). Čím více dopředu se však plánuje, tím více se pak první fáze musí při zotavování vracet. Všechny nevalidní příkazy vrácené robotem po detekci neočekávané překážky se totiž použijí na navrácení některých proměnných do korektního stavu (používá se například pro zotavení v kvadrantovém stromu, ten totiž nepatří k reálným ani virtuálním proměnným a všechny akce jsou prováděny pouze v jedné struktuře). Pokud plánovací fáze dokončí plánování celého programu robota, zastaví se tato fáze a vlákno počká na dokončení dané práce robotem. Tedy stále dochází k přijímání potvrzování příkazů od robota, a pokud by došlo k detekci neočekávané překážky, opět dojde k zotavení a znovuspuštění plánovací fáze, neboť dříve naplánované kroky již nebudou validní.

Jakmile robot splní všechny naplánované kroky, pole je zpracováno, a pošle

se zakončující zpráva (událost) grafickému rozhraní (tj. třídě `WorkPanel`).

4.2.7 Pracovní modul – vlákno `ReplayThread`

V případě spuštění Pracovního modulu v režimu přehrávače logu se použije místo pracovních vláken `NavCoreThread` a `RobotThread` vlákno jediné, a sice `ReplayThread` (znovu potomek třídy `wxThread`). Toto vlákno neprovádí žádné kalkulace, pouze čte obsah dříve zaznamenaného logu práce robota na poli. S třídou `WorkPanel` však komunikuje stejně jako vlákno `RobotThread`, tedy prostřednictvím zasílání událostí, a `WorkPanel` tyto události obsluhuje stejně jako v případě `RobotThreadu`. Rychlost zasílání těchto událostí je dána hodnotou proměnné `repl_speed`, kterou uživatel může ovlivnit přímo za běhu klikem na příslušné tlačítko na pracovní ploše modulu. Po dočtení celého logu se zašle `WorkPanelu` zakončující zpráva (událost) spolu se shrnujícími detaily právě proběhlého programu. Tyto detaily nemusí být k dispozici, pokud byl původní program robota (zaznamenávaný v aktuálním logu) předčasně ukončen. `WorkPanel` samozřejmě o této skutečnosti následně informuje uživatele.

Závěr

Představili jsme si projekt **R4Farmer** – systém **autonomního robotického zemědělce**, který by v budoucnu mohl při zpracovávání pole nahradit klasický traktor. Jeho předpokládané nasazení je na **polích Afriky** a dle toho bude uzpůsoben (konstrukce odolná vůči extrémním teplotám, využití solární energie, ...). Ke svému fungování bude využívat výhradně **elektřinu** a bude se proto dobíjet i doplňovat komodity na tzv. **bodech energetické obsluhy** rozmístěných v poli. Robot bude rovněž schopen vyrovnat se s neočekávanými překážkami v poli a pole samotné bude zpracovávat co nejefektivněji vůči spotřebě energie.

Tato práce měla za úkol vytvořit dostatečně kvalitní prostředí pro modelování polí, nastavování všech parametrů plánovaného programu robota, dát k dispozici uživateli možnost sledovat průběh práce robota na poli a navrhnout konkrétní algoritmy zajišťující program robota na poli. Samozřejmostí je pak vytvoření emulačního prostředí pro další ladění a vývoj softwaru robota, bez rizika hospodářských škod.

Tohoto jsme dosáhli v podobě dvou logicky oddělených modulů (*Konfiguračního* a *Pracovního*). **Konfigurační modul** dává uživateli pohodlné nástroje pro vytvoření pole libovolného tvaru s libovolným množstvím překážek a rozmístění bodů energetické obsluhy. Umožňuje rovněž detailní nastavení parametrů robota i prvků pole – jde například o rozměry robota, kapacitu nádrže pro užívané komodity, nastavení předpokládané spotřeby jednotlivých operací robota, úroveň obezřetnosti vůči neočekávaným překážkám apod. Dále si uživatel navolí požadovaný program robota (rytí, setí či zavlažování), zvolí jeden z algoritmů a případně může průběh práce ukládat do logu pro pozdější analýzu. V případě simulace má uživatel taktéž možnost nastavit rychlost provádění jednotlivých operací robota i možnost specifikovat neočekávané překážky v poli.

Pracovní modul následně nabízí **vizualizaci** průběhu práce robota, včetně možnosti užití **emulovaného** robota. Nedílnou součástí modulu jsou plánovací algoritmy, které generují konkrétní program robota na poli. Komunikace navigačního jádra s (emulovaným) robotem je vybudována tak, aby nebyl problém systém nasadit na skutečného robota.

Právě zmíněné **plánovací algoritmy** tvořily další významnou část této práce. Prozkoumali jsme doposud užívané metody řešení obdobných problémů pokrývání oblastí a zjistili jsme, že žádná z nich není vlivem daných omezení našeho systému použitelná. Proto jsme se pokusili implementovat vlastní přístupy. Ze třech navržených metod se jedna z nich ukázala jako potenciálně použitelná v praxi. Důraz byl kladen na to, aby v programu robota **nedošlo k energetickému selhání** – tj. robot by se dostal do situace, kdy by zbývající úroveň energie jeho akumulátoru klesla na nulu a on zůstal uvázlý uprostřed pole.

Provedli jsme **analýzu** daných přístupů a porovnání výkonnosti na různých typech vstupních dat. Srovnávací testy ohledně energetické efektivity mají jasného vítěze, a sice řádkový algoritmus (popsaný v podkapitole 3.3). Je třeba říci, že všechny navržené algoritmy mají své rezervy a možnosti pro další optimalizaci. Tyto možnosti jsou uvedeny v závěru u každého z algoritmů.

Zmiňme, které věci se plánují **řešit v následujících fázích projektu**. Jde především o řešení výškových nerovností terénu. 3D navigaci v terénu sice řeší jiná

část softwaru, která není součástí této práce, nicméně plánovací jádro by mělo mít k dispozici přibližné informace o reliéfu terénu, aby mohlo při svých plánech počítat se zvýšenou spotřebou, např. vlivem jízdy do kopce. V současnosti se počítá s tím, že diference mezi reálnou spotřebou a plánovanou teoretickou bude korigována v rámci měření kapacity akumulátorů robota. Robot pošle zpětnou vazbu plánovacímu programu a dochází k případnému přepočítání trasy robota. Nicméně budoucí verze našeho softwaru by měla umožnit specifikovat přibližný reliéf terénu přímo v Konfiguračním modulu při modelování pole a plánovací jádro tak bude moci s tímto dopředu počítat při plánování trasy.

Dalšími možnostmi pro **potenciální rozšíření** jsou terminálové fungování (např. za použití *chytrého* mobilního zařízení) či možnost definovat při modelování více typů překážek v poli (např. specifikovat oblasti, které sice nemají být zpracované, avšak robot přes ně může libovolně přejíždět). Jiné možné rozšíření je vytváření odhadu, jakou plochu lze s daným množstvím osiva/vody zpracovat (na základě množství komodity v bodech energetické obsluhy, rychlosti operací robota a rychlosti spotřeby dané komodity při zpracovávání pole). Program by pak mohl ještě před započnutím práce oznámit, že je v poli například rozmístěno málo bodů energetické obsluhy, a je tudíž třeba umístit další body nebo zvýšit množství komodit v zásobnících jednotlivých bodů.

Reference

[1] Cormen, Thomas H. *Introduction to Algorithms.* Second Edition. MIT Press, 2001. ISBN 0-262-03293-7.

[2] Fortune, Steven. *Voronoi diagrams and Delaunay triangulations.* CRC Press, 1997.

[3] Gabriely, Yoav a Rimon, Elon. *Spanning-Tree Based Coverage of Continuous Areas by a Mobile Robot.* Israel Institute of Technology, 1999.

[4] LaValle, Steven M. *Planning Algorithms.* University of Illinois, 2004.

[5] Matoušek, Jiří a Nešetřil, Jaroslav. *Kapitoly z diskrétní matematiky.* Třetí vydání. Karolinum, 2007. ISBN 978-80-246-1411-3.

[6] Norvig, Peter a Rusell, Stuart J. *Artificial intelligence: A Modern Approach.* Prentice-Hall, 1995. ISBN 0-13-103805-2.

[7] Samet, Hanan. *The Quadtree and Related Hierarchical Data Structures.* University of Maryland, 1984.

[8] Töpfer, Pavel. *Algoritmy a programovací techniky.* Prometheus, 1995. ISBN 80-85849-83-6.

Příloha A – CD

Obsah disku

- Zdrojové kódy aplikace R4Farmer + použité knihovny
- Instalační soubor balíku wxWidgets verze 2.8.12
- Instalační balíček aplikace R4Farmer pro Windows
- Spustitelné verze aplikace R4Farmer pro Windows
- Testovací data – vymodelovaná pole
- Elektronická verze tohoto dokumentu ve formátu PDF