Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

# Diplomová práce

# Plánování pohybu pro modelování a vizualizaci proteinů

Plzeň, 2017 Ondřej Byrtus

# Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 26.6.2017 _____

Ondřej Byrtus

# Poděkování

Tímto bych rád poděkoval vedoucí práce prof. Dr. Ing. Ivaně Kolingerové a Ing. Jakubovi Szkanderovi za cenné rady, odborné vedení této práce, stejně jako za trpělivost a časté konzultace.

# Abstract

**Motion planning for protein modeling and visualization**
This master thesis deals with the problem of motion planning in the protein models and the algorithms using for their desciption graph representation.

Main goal of the thesis is on the basis of available resources to propose and implement the algorithm of motion planning suitable for the use in protein models. The implemented algorithm is described in detail in the results of its testing on real data are presented.

# Abstrakt

**Plánování pohybu pro modelování a vizualizaci proteinů**

Tato diplomová práce se zabývá problematikou plánování pohybu v prostředí proteinových modelů a algoritmů používajících pro jejich popis grafovou reprezentaci.

Hlavním cílem práce bylo na základě dostupné literatury navrhnout a implementovat algoritmus plánování pohybu vhodný pro využití v modelech proteinů. Práce do hloubky popisuje implementovaný algoritmus a vyhodnocuje výsledky testování tohoto algoritmu na reálných datech.

# Obsah

# 1 Úvod

Jedním z tradičních problémů algoritmizace je hledání cest v různých modelech. Vždyť Dijkstrův algoritmus bývá zařazen do základní výuky po boku abstraktních datových typů a algoritmů řazení.

U algoritmů hledání cesty pracujících nad grafy je vždy k dispozici popis volného a bezpečného prostoru. Ten ovšem obecně nemusí být k dispozici – častým zadáním bývá pouze popis překážek a přechod k popisu volného prostoru nemusí být triviální. Celý problém pak nabývá na náročnosti i pouhým zavedením nenulové velikosti navigovaného agenta.

V modelech, kde by byla deterministická tvorba kvalitního grafového modelu stejně složitá jako hledání cesty samotné, přichází ke slovu algoritmy pravděpodobnostní. U pravděpodobnostních algoritmů hledaní cesty, protože obvykle hledají cesty robotickým agentům, se převážně mluví o plánování pohybu namísto hledání cesty. Plánování pohybu může zahrnovat i další omezení kromě bezkoliznosti cesty.

Pro adresování zmíněných problémů byly vyvinuty pravděpodobnostní algoritmy, jež tato práce popisuje a v případě RRT* i implementuje. Vzhledem k obecnosti popisu těchto algoritmů se nejedná o postupy aplikovatelné bez pečlivé analýzy prostředí a možností jejich implementace. Zde přichází ke slovu tato práce.

## 1.1 Cíl práce

Práce si dává za cíl prozkoumat možnosti řešení navigování složitých sond v prostředí modelů proteinů za pomoci pravděpodobnostních algoritmů, navrhnout takové algoritmy a implementovat aplikaci, jež umožní výpočet a snadnou budoucí modifikaci těchto algoritmů spolu s vizualizací výsledků.

## 1.2 Obsah práce

Práce je dělena do tří hlavních částí. První část práce se věnuje analýze doposud známých řešení, počínajíc od úvodu do problematiky v sekci 2.1 přes popis zástupců grafových a pravděpodobnostních algoritmů, po srovnání jejich předností a nevýhod.

Druhá část, kapitola 3, se věnuje vypracovanému řešení. Počínaje detailním popisem implementovaného algoritmu hledání cesty a možností jeho úprav v sekci 3.1 přes volbu technologie v sekci 3.2 po implementaci aplikace v sekci 3.3.

V poslední z hlavních částí, kapitole 4, je věnován prostor experimentům nad algoritmem ze sekce 3.1. U jednotlivých provedených experimentů jsou

popsány podmínky, metodika, výsledky i následné vyhodnocení výsledků experimentů.

# 2 Teoretická část

Teoretická část se nejprve v sekci 2.1 věnuje uvedení do kontextu plánování cest pro problematiku proteinových modelů. Sekce 2.1 též definuje základní pojmy využívané zejména u popisu algoritmů v dalších sekcích. Samotným modelům využívaným v práci se pak věnuje sekce 2.2.

Po představení modelů a úvodu do problému plánování pohybu následují části popisující studované algoritmy, které byly vybrány po domluvě s vedoucí. Jedná se pravděpodobnostní algoritmy založené na pravděpodobnostní mapě (Probabilistic roadmap) a tzv. rychle prozkoumávajícím náhodném stromu (Rapidly exploring random tree) popsaných v sekcích 2.4 a 2.5. Sekce 2.6 porovnává algoritmy z obou jmenovaných skupin a vysvětluje finální výběr.

## 2.1 Plánování pohybu a hledání cesty

Před dalším postupem je třeba zavést pojmy plánování pohybu (motion planning), plánování cesty (path planning), hledání cesty (path finding) a jejich souvislosti v kontextu této práce. Pro jednoduchost je v následující definicích používáno shodné označení „bod“ jak pro vrchol grafu, tak pro komplexní stav navigovaného robota. Cestou se rozumí posloupnost záchytných bodů. Volným prostorem se rozumí prostor, v němž se nenachází žádná z překážek

Hledáním cesty je možné rozumět proces nalézání bezkolizní cesty pro agenta zadaným prostorem. Obvyklým výstupem procesu hledání cesty je posloupnost záchytných bodů respektive vrcholů grafu spojující bod počáteční a cílový. Typickým vstupem procesu bývá graf.

Příbuzným pojmem je plánování pohybu. Tento pojem je používán zejména v robotice. Na rozdíl od hledání cesty se plánování pohybu nezabývá cestou samotnou, ale jejím následováním. Obdobně jako u hledání cesty se požaduje, aby byl pohyb bezkolizní – modely plánování pohybu mohou obsahovat překážky. Plánování pohybu může navíc zahrnovat omezení kinematická (klouby) stejně jako dynamická (např. setrvačnost). Vstupem procesu může být posloupnost záchytných bodů, výstupem je pak například posloupnost příkazů řídící jednotce robota.

Pojem plánování cesty je neutrální a může obsáhnout kombinaci obou výše zmíněných problémů. Vzhledem k cílům práce je nejvhodnější používat pojmu plánování cesty.

Plánování pohybu nalézá uplatnění například v řízení vesmírných a podvodních robotů, helikoptér, humanoidů anebo ve virtuálním prototypování.

### 2.1.1 Prostor konfigurací C-space

Modely plánování cesty, jak je uvedeno v sekci 2.1, mohou uvažovat nejenom pozici agenta, ale i například orientaci anebo další omezení. Konfiguracemi jsou pak označovány vektory popisující pozici agenta v prostoru těchto modelů. Například u robota v trojrozměrném prostoru by mohla konfigurace být například šestirozměrným vektorem popisujícím jeho transformaci skládající se z vektorů pozice a orientace v prostoru. Názvosloví také používá pojmu „stav“ pro vektory popisující konfigurace agentů v modelech, jež zahrnují dynamická omezení.

Jako označení pro prostory modelů plánování cesty se zavádí pojem „prostor konfigurací“ (C-space) . Modely proteinů využívané v práci setrvačnost neuvažují, proto se v práci mluví převážně o prostorech konfigurací. Navíc z pohledu algoritmizace jsou mnohdy rozdíly mezi prostorem konfigurací a prostorem stavů zanedbatelné.

### 2.1.2 Metriky v kontextu plánování cest

Formálně [14] je metrika $\rho$ na množině $X$ funkcí, jež je definována na kartézském součinu $X \times X$ s hodnotami náležícími $E_1$ reálných čísel a splňující všechny následující podmínky:

1. $\rho(x, y) \geqslant 0$ pro všechna $x, y \in X$ (nezápornost)
2. $\rho(x, y) = 0$ právě tehdy, když $x = y; x, y \in X$ (totožnost)
3. $\rho(x, y) = \rho(y, x)$ pro všechna $x, y \in X$ (symetrie)
4. $\rho(x, z) \leqslant \rho(x, y) + \rho(y, z)$ pro všechna $x, y, z \in X$ (trojúhelníková nerovnost)

V kontextu hledání cesty se obvykle rozumí metrikou vzdálenost, euklidovská norma pozic vrcholů grafu. V kontextu plánování cest se obvykle uvažují i metriky složené. Důvodem skládání několika různých metrik je potřeba reflektovat lišící se významy jednotlivých dimenzí konfigurace. Volba metriky je netriviální záležitostí vzhledem k jejímu vlivu na výkon algoritmů.

Volba metrik se také liší model od modelu. Z tohoto důvodu uvádění konkrétních metrik v popisech algoritmů nebývá obvyklé. V plánování i hledání cesty se často za metriky označují (formálního pohledu nesprávně) i funkce, které nesplňují některé z výše vyjmenovaných podmínek.

### 2.1.3 Algoritmy hledání cesty v grafech

V případě plánování cesty popis volného prostoru často chybí – volný prostor je implicitně popsán překážkami. Toto platí i pro v práci využívané proteinové modely. Výstupem některých pravděpodobnostních algoritmů plánování cesty ovšem nemusí být přímo cesta, výstupem algoritmu může být i graf ve kterém se cesta hledá dodatečně běžnými algoritmy hledání cesty. Příkladem algoritmu generujícího graf je algoritmus PRM uvedený v sekci 2.4.

Z těchto důvodů může být při plánování cesty využito i algoritmů hledání cest v grafu. Jak již název napovídá, tyto algoritmy pracují nad grafem. Vrcholy grafu mohou odpovídat konfiguracím ve volném prostoru, hrany pak spojují vrcholy lokální bezkolizní cestou. Základními algoritmy pracujícími s grafy jsou:

**BFS** (Breadth first search) [2] je prohledáváním do šířky. Algoritmus pomocí fronty rekurentně prochází všechny sousedy zpracovávaného vrcholu. Tento proces probíhá, dokud nejsou zpracovány všechny vrcholy komponenty grafu. Následně je zvolen nový počáteční vrchol v jedné z neprozkoumaných komponent.

**DFS** (Depth first search) [2] je prohledáváním do hloubky. Využívá principu „backtrackingu“ – algoritmus se vrací, neexistuje-li žádný plně neprozkoumaný soused. V případě vyčerpání vrcholů ke zpracování (všichni sousedé počátečního vrcholu jsou plně prozkoumáni) volí nový počáteční vrchol v jedné z neprozkoumaných komponent.

**Dijkstrův algoritmus** [2] nalézá v $\mathcal{O}(n \log n)$ nejkratší cestu mezi dvěma vrcholy. Postup algoritmu je obdobný BFS. Hlavním rozdílem je využití informace o délce nejkratší cesty do vrcholu jako váhy prioritní fronty vrcholů ke zpracování.

**A*** (A star)[3, 6] je algoritmus využívající kombinace heuristického hledání a prohledávání založeného na nejkratší cestě. Algoritmus je uspořádaným prohledáváním (best-first search), prohledávání je řízeno odhadem ceny vrcholů. V každé iteraci je z fronty vybrán k expanzi vrchol s minimální cenou. Cena vrcholů je odhadována dle rovnosti $f(v) = h(v) + g(v)$, kde $h(v)$ je heuristická vzdálenost (např. Eukleidovská) k cílovému vrcholu a $g(v)$ je délka doposud nalezené cesty z vrcholu počátečního do prozkoumávaného vrcholu.

Implementace algoritmu obvykle využívají prioritní frontu. V každé iteraci je z fronty vybrán ke zpracování vrchol minimalizující cenu. Poté jsou do fronty zařazeni sousedi zpracovávaného vrcholu. Sousedům je

odpovídajícím způsobem upravena cena – k úpravě dochází, snižuje-li se cestou přes aktuálně zpracovávaný vrchol.

**LPA*/D* Lite** Algoritmy LPA* (Lifelong Planning A*) a D* Lite (Dynamic A* Lite) rozšiřují A* pro použití v měnícím se grafu. Algoritmus D* Lite [11] byl odvozen z LPA* [12] a sdílí důležité vlastnosti, proto se následující odstavce věnují pouze D* Lite. D* Lite je považován za jednodušší než D*.

Algoritmus D* Lite zavádí prioritu vrcholu jako dvourozměrný vektor $k = [k_1, k_2]$ daný vztahy 2.1,

$$\begin{aligned} k(v) &= [min(g(v), rhs(v)) + h(v), min(g(v), rhs(v))], kde \\ rhs(v) &= min_{v' \in Pred(v)}(g(v') + c(v', v)), \\ rhs(v_{start}) &= 0 \end{aligned} \tag{2.1}$$

kde $Pred(v)$ je množina předchůdců $v \in V$ a $0 < c(s', s) < \infty$ udává cenu přesunu z $v$ do $v'$. Obdobně jako u A* $h(v)$ je heuristikou odhadující vzdálenost vrcholu cílového a $g(v)$ je délka doposud nalezené cesty z počátečního vrcholu. Priority jsou řazeny lexikograficky, tzn. nejprve jsou porovnány složky $k_1$, složky $k_2$ jsou porovnávány pouze v případě rovnosti prvních složek.

První krok algoritmu je obdobný výpočtu nejkratší cesty v algoritmu A*, ovšem využívá priorit $k$. V dalších iteracích jsou v grafu hledány změny. Po nalezení změny je aktualizována cena příslušné hrany a její počáteční vrchol je vložen do prioritní fronty. Při zpracování vrcholu je přepočítána jeho priorita. Zvýší-li se priorita vrcholu, je znovu přidán do fronty. Po aktualizaci ceny jsou do prioritní fronty přidáni sousedé zpracovávaného vrcholu.

## 2.2 Proteinové modely

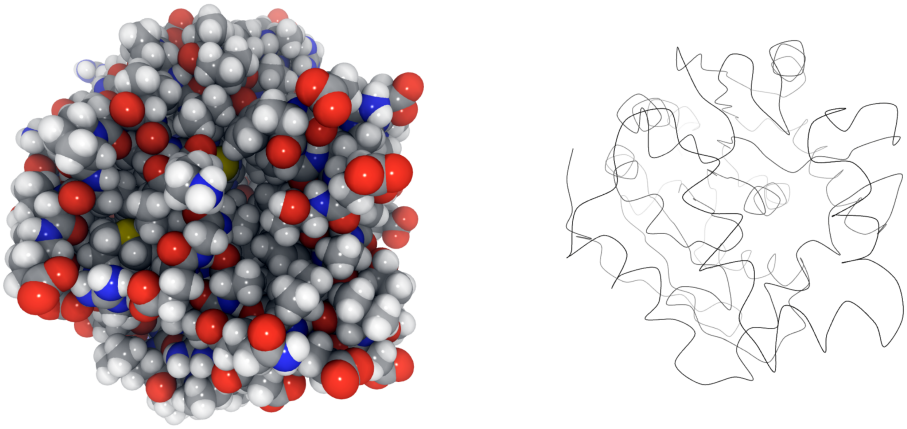
Základními stavebními složkami proteinů [17] jsou proteinogenní aminokyseliny, molekuly sestávající se z několika málo atomů, kterých je v závislosti na přesné definici zhruba dvacet druhů.

Každá aminokyselina obsahuje část umožňující navázat peptidové vazby, a tak vytvořit řetězec aminokyselin. Proteiny se skládají z jednoho anebo více těchto řetězců. Pořadí v řetězci je nazýváno primární strukturou proteinu. Toto pořadí je zakódováno v DNA a řetězy aminokyselin jsou syntetizovány pomocí ribozomů při transkripci a translaci.

Jednotlivé části řetězců mohou být uspořádány například do listů anebo šroubovic, což je označováno za sekundární strukturu proteinů.

Terciární struktura popisuje trojrozměrné uspořádání listů a šroubovic řetězce. Funkce proteinu je určena právě terciální strukturou. K zachycení této struktury spolu s přitažlivými a odpudivými silami mezi atomy se využívá van der Waalsova modelu.

Van der Waalsův model zachycuje terciární strukturu proteinů spolu se silami vzájemně působícími mezi atomy a umožňuje studium funkce proteinů [17]. V tomto modelu jsou atomy představovány sférami. Je-li mezi atomy silná vazba, jejich sféry se překrývají. V ostatních případech by se sféry neměly překrývat z důvodu působení odpudivých sil. Příklad van der Waalsova modelu modelu je zachycen na obrázku 2.1.

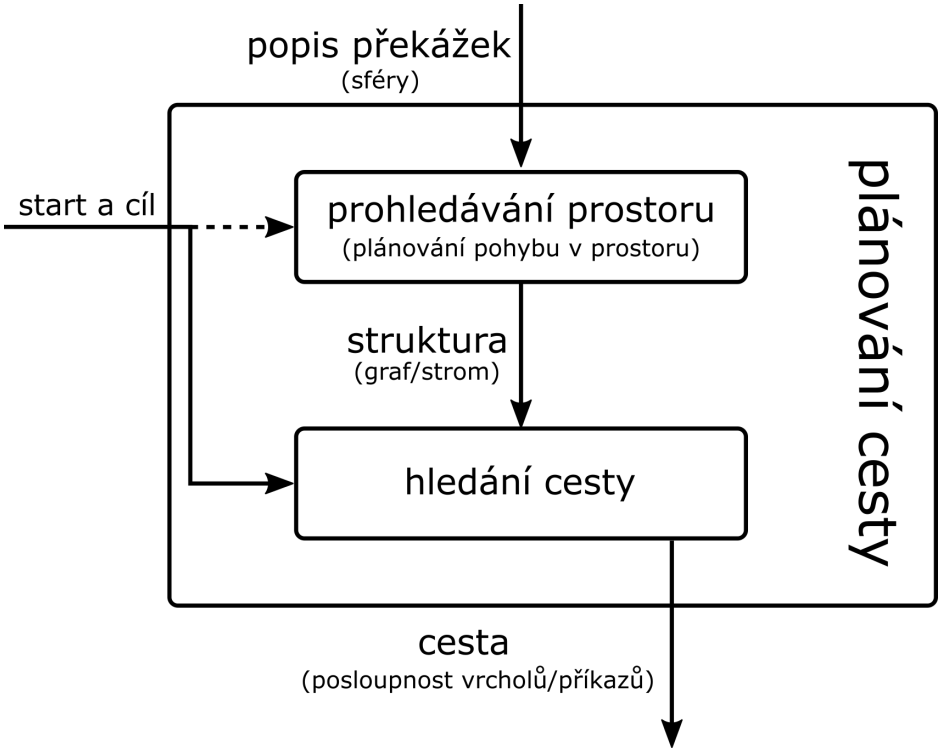


Obrázek 2.1: Ukázka van der Waalsova modelu [17] vykreslený programem CAVER Analyst [13] (vlevo) a řetězu aminokyselin (vpravo)

Poloměry sfér, konstanty van der Waalsova modelu, jsou určeny experimentálně měřením. Polohy sfér jsou určovány pomocí NMR spektroskopie nebo rentgenové krystalografie. Sféry modelu bývají také obvykle obarveny dle chemického prvku, kterému přísluší.

## 2.3 Definice problému

Problém plánování cesty řešený v práci zachycuje obrázek 2.2 – vstupem algoritmů je popis překážek, výstupem pak hledaná cesta. Modely proteinů (popsané v sekci 2.2) se skládají ze sfér (koulí) různých poloměrů. V případě proteinových modelů se obvykle agent nazývá „sonda“.




Obrázek 2.2: Schéma plánování pohybu

Sondy v těchto modelech se obvykle také sestávají ze sfér a jejich konfigurace je popsána pozicí v prostoru a orientací. Zatímco je počáteční konfigurace na vstupu, koncová konfigurace se může nacházet kdekoliv vně proteinu. U koncové konfigurace zásadním způsobem záleží na definici „vně“ proteinu, pro jednoduchost lze ale uvažovat, že její pozice je minimálně v určité vzdálenosti od nejbližší překážky.

V modelu může existovat několikero cest, cílem navrhovaného algoritmu je nalézt všechny unikátní cesty. Unikátní cestou se rozumí cesta vedoucí „tunelem“, jehož podstatnou část nesdílí s jinou nalezenou cestou.

## 2.4 Pravděpodobnostní mapa

První a starší z popisovaných je algoritmus pravděpodobnostní mapy (Probabilistic Roadmap). Algoritmus je dále v práci označován jako PRM. Článek [10] definuje algoritmus v prostoru konfigurací. Jeho popis zahrnuje několik upravitelných částí. To umožňuje velkou volnost jeho nastavení. Jedná se však o netriviální nastavení, které je ale pro výkonnost algoritmu klíčové.

### 2.4.1 Popis algoritmu

Algoritmus PRM [10, 5] staví nad prozkoumanými částmi prostoru graf. Prostor modelu je prozkoumáván náhodným vzorkováním. Postačujícím vstupem algoritmu je množina překážek $P$. Znalost počáteční ani cílové konfigurace není samotným algoritmem vyžadována, nicméně jejich znalost může být využita v některých heuristikách vzorkování. Pro zjednodušení popisu se předpokládá, že obě konfigurace jsou na vstupu k dispozici. Výstupem algoritmu je orientovaný graf mapující prozkoumaný prostor. Algoritmus pracuje s časovou složitostí $\mathcal{O}(n \log n)$ [9]. Tento graf je využíván k nalezení cesty některým z grafových algoritmů. Příklady grafových algoritmů lze nalézt v sekci 2.1.3.

Základní myšlenkou algoritmu je propojování nově vygenerovaných náhodných konfigurací s podmnožinou již vložených vrcholů grafu bezkolizní cestou. Přesná implementace náhodného vzorkování, stejně jako procedury výběru kandidátů k propojení i testování propojitelnosti (*local planner*) konfigurace je ponechána na implementátorovi algoritmu. Možnostmi implementace těchto procedur se věnuje studie [5], jež také tyto procedury detailně srovnává.

Pseudokód 2.1 popisuje variantu algoritmu se známou počáteční i cílovou konfigurací. Vstupem je počet iterací $n$, množina překážek $P$ a konfigurace počáteční $v_{start}$ a koncová $v_{goal}$. Množiny $E$ i $V$ mohou být též na vstupu v případě že je rozšiřována již existující mapa (řádka 3 je pak vynechána). Výstupem algoritmu je graf $G$.

Na řádcích 2 a 3 je množina hran $E$ inicializována na prázdnou a do množiny vrcholů $V$ jsou vloženy vrcholy $v_{start}$ a $v_{goal}$.

V každé iteraci (řádky 5 až 15) je nejprve na základě konfigurace vygenerované procedurou *Sample* vytvořen nový vrchol, jenž je posléze vložen do množiny vrcholů grafu. Následuje výběr „užitečných“ vrcholů grafu procedurou *FindUseful* a jejich zpracování (řádky 8 až 15) vzestupně dle vzdálenosti od nového vrcholu $v_{new}$.

Při zpracování „užitečného“ vrcholu $v$ je ověřována vzájemná propojitelnost s $v_{new}$. V případě úspěšného nalezení spojení je do grafu přidána hrana s odpovídající orientací spojující $v$ a $v_{new}$.

U náhodného vzorkování prostoru (*Sample*) se předpokládá, že jeho výstupem jsou bezkolizní konfigurace. Vygenerované konfigurace jsou přidávány jako vrcholy grafu. Možnostem implementace náhodného vzorkování se detailněji věnuje sekce 2.4.2. Při plánování více než jedné cesty grafem je možné obdobným způsobem do grafu zařadit dodatečné počáteční a cílové konfigurace.

**Algoritmus 2.1** PRM

```
1: procedure BUILD_PRM(n, P, v_start, v_goal)
2:     V ← {v_start, v_goal}
3:     E ← ∅
4:     for i = 1 ... n do
5:         v_new ← Sample (P)
6:         V ← V ∪ v_new
7:         N_v ←FindUseful(V, E, v_new)
8:         for all v ∈ N_v, ascending by distance to v_rand do
9:             if CanConnect(v, v_new)) then
10:                E ← E ∪ {(v, v_new)}
11:            end if
12:            if CanConnect((v_new, v)) then
13:                E ← E ∪ {(v_new, v)}
14:            end if
15:        end for
16:    end for
17:    return G = (V, E)
18: end procedure
```

Jedním z klíčových faktorů úspěšnosti a výkonnosti je procedura vybírající „užitečné“ vrcholy (*FindUseful*) grafu ve smyslu schopnosti nalezení cesty prostorem. Po vytvoření nového vrcholu na základě náhodně vygenerované konfigurace je procedurou *FindUseful* vybrána odpovídající množina „užitečných“ vrcholů grafu. Ke každému prvku této množiny se následně pokusí procedura *CanConnect* najít bezkolizní spojení. Pokud tento proces uspěje, do grafu je přidána hrana spojující daný vrchol a novou konfiguraci. Možnostem implementace výběru „užitečných“ vrcholů grafu se detailněji věnuje sekce 2.4.3.

U procedury *CanConnect* testující propojitelnost je kladen důraz na rychlost. Proceduře je nicméně umožněno se mýlit falešným odmítnutím (tj. chyba II. typu). Ve článku [10] se uvádí jako jedna z možností implementace inkrementální detekce kolizí [18]. Hrany, jež propojují dvě komponenty grafu, vylepšují schopnost grafu nalézt cestu, a proto mohou být algoritmem preferovány. Nicméně hrany spojující vrcholy stejné komponenty umožňují potenciálně nalézt cestu s nižší cenou a mohou být též akceptovány, je-li cílem získat kratší cestu.

### 2.4.2 Vzorkování prostoru

Studie [5] uvádí několik možností vzorkování. Navíc je uvedena i možnost kombinace vícero vzorkování pro různé dimenze prostoru konfigurací. Tyto metody lze dále dělit na přístupy s využitím a bez využití překážek.

**Bez využití znalosti překážek**
Přestože tyto metody vzorkování nevyužívají informací o překážkách ve scéně, výsledky příliš nezaostávají [5] za komplikovanějšími přístupy v následujícím odstavci. Tyto metody též slouží jako podklad pro metody využívající znalost překážek:

- **Uniformní náhodné vzorkování** – složky vektoru konfigurace náhodně generovány z uniformního náhodného rozdělení.
- **Hierarchická mřížka** začíná s hrubou pravidelnou mřížkou. Mřížka je v pozdějších iteracích algoritmu postupně zjemňována. Vzorky v rámci jedné úrovně jsou generovány v náhodném pořadí.
- **Hierarchické náhodné buňky** – kombinace výše zmíněných – hierarchický postup obdobný hierarchické mřížce. Okolí vrcholů mřížky jsou využívána jako středy buněk, z nichž jsou konfigurace vzorkovány.
- **Haltonská množina bodů** – deterministická varianta algoritmu vzorkování využívající Haltonovské posloupnosti (deterministická posloupnost jevící se jako náhodně generovaná) místo pseudonáhodného generátoru čísel k vytvoření konfigurací.
- **Randomizovaná Haltonská množina bodů** – kombinace metody využívající buněk jako okolí ke vzorkování a metody s Haltonovskou posloupností. Haltonovská posloupnost bodů je využita jako střed buněk v nichž probíhá vzorkování.

**S využitím znalosti překážek**
Následující metody se snaží o jemnější vzorkování v blízkosti překážek. Tyto metody vycházejí z předpokladu, že vzorky v blízkosti překážek mají vyšší vliv na schopnost algoritmu nalézt cestu prostorem.

- **Náhodné gaussovské vzorkování** – vzorek je generován lineární interpolací dvou vybraných náhodných vzorků. Právě jeden z vybraných interpolovaných vzorků musí být v kolizi s překážkami. Parametr interpolace má gaussovské náhodné rozdělení a je omezen na interval $< 0, 1 >$.

- **Náhodný posun** – je vygenerována náhodná konfigurace v kolizní pozici. Poté je vybrán náhodný směr ve kterém je konfigurace přesouvána, dokud se nenachází v pozici bezkolizní.

### 2.4.3 Výběr propojovaných uzlů

Neméně důležitou součástí algoritmu je výběr „užitečných“ uzlů, k nimž se *local planner* pokusí nový vzorek připojit. Protože testy připojitelnosti mohou být drahé, velký počet těchto uzlů není žádoucí. Stejně tak příliš nízký počet uzlů může vyústit v selhání připojení vrcholu do grafu a tudíž i nalezení cesty. Je vysloven předpoklad [5], že bližší uzly jsou „užitečnější“, protože se očekává že šance úspěšného spojení velmi vzdálených uzlů je minimální. Volba vhodné metriky $\rho$ této vzdálenosti je též nepřímo ovlivňuje tyto metody.

- **Nejbližších n** – přímočará volba, jež vybírá $n$ nejbližších vrcholů grafu.
- **Nejbližší v komponentě** – k propojení vždy vybrá nejbližší vrchol z každé komponenty grafu.
- **Nejbližích n v komponentě** – kombinuje výše zmíněné možnosti a vybírá z každé komponenty $n$ nejbližších vrcholů k propojení.
- **Viditelnost** – nový vzorek označuje za „užitečný“, jsou-li nalezena spojení tohoto vrcholu s žádnou anebo více než jednou komponentou grafu.

### 2.4.4 sPRM

Oproti předchozím sekcím, jež se věnovaly různým heuristikám vylepšujícím výsledky algoritmu PRM, je sPRM [9] naopak zjednodušenou verzí. Spíše než pro praktické využití je proto využíván pro analýzu navazujících algoritmů. Ačkoliv je časová složitost tohoto algoritmu horší ($\mathcal{O}(n^2)$ oproti $\mathcal{O}(n \log n)$), narozdíl od předchozích metod s $n \to \infty$ nalézá cestu asymptoticky optimální [8].

Postup algoritmu zachycený pseudokódem Algoritmus 2.2 se liší od PRM v odstranění procedury *FindUseful* (za „užitečné“ jsou považovány všechny vrcholy). Odlišný je i první krok algoritmu (řádky 2 až 6), ve kterém je kromě vložení počáteční a cílové konfigurace též vygenerováno a vloženo do množiny vrcholů $V$ všech $n$ vrcholů. Algoritmus postupně zpracovává všechny vrcholy $V$ (řádky 7 až 13) způsobem obdobným jako v algoritmu PRM (pseudokód 2.1, řádky 9 až 14), kdy je v případě úspěšného nalezení propojení dvojice vrcholů přidána do množiny hran $E$ odpovídající hrana.

**Algoritmus 2.2** sPRM

```
1: procedure BUILD_sPRM(n, P, v_start, v_goal)
2:     V ← {v_start, v_goal}
3:     E ← ∅
4:     for i = 1 ... n do
5:         V ← V ∪ Sample(P)
6:     end for
7:     for all v ∈ V do
8:         for all u ∈ V\{v} do
9:             if CanConnect(v, v_new)) then
10:                E ← E ∪ {(u, v_new)}
11:            end if
12:            if CanConnect((v_new, v)) then
13:                E ← E ∪ {(v_new, u)}
14:            end if
15:        end for
16:    end for
17:    return G = (V, E)
18: end procedure
```

### 2.4.5 PRM*

Článek [8] představuje další variantu algoritmu, PRM* (PRM star, viz Algoritmus 2.3). Jedná se o algoritmus vycházející z sPRM s jediným rozdílem, že potenciální vrcholy k propojení jsou vybírány z okolí $r$ (řádka 9) daného vztahy 2.2:

$$\begin{aligned} r &= \gamma_{PRM}(\log n/n)^{1/d}, kde \\ \gamma_{PRM} > \gamma^*_{PRM} &= 2(1+1/d)^{1/d}(\mu(C_{free})/\zeta_d) \end{aligned} \tag{2.2}$$

kde $d$ je počet dimenzí prostoru konfigurací $C$, $n$ počet vrcholů grafu, $C_{free}$ prostorem bez překážek, $\mu(C_{free})$ je Lebesgueova míra (objem) prostoru bez překážek a $\zeta_d$ je objem jednotkové koule v $d$-dimenzionálním Eukleidovském prostoru.

Cílem tohoto omezení je snížit počet pokusů o propojení na průměrně $\log n$ vrcholů, což vede na $\mathcal{O}(n \log n)$ časovou složitost. Stejně jako sPRM, tento algoritmus nalézá cestu asymptoticky optimální [8].

**Algoritmus 2.3** PRM*

```
1: procedure BUILD_PRM*(k, P, v_start, v_goal)
2:     V ← {v_start, v_goal}
3:     E ← ∅
4:     r ← γ_PRM(log K/K)^{1/d}
5:     for i = 1 ... K do
6:         v ← V ∪ Sample(P)
7:     end for
8:     for all v ∈ V do
9:         U ← u ∈ V\{v}, ρ(u, v) ⩽ r
10:        for all u ∈ U do
11:            if CanConnect(v, v_new) then
12:                E ← E ∪ {(u, v_new)}
13:            end if
14:            if CanConnect(v_new, v) then
15:                E ← E ∪ {(v_new, u)}
16:            end if
17:        end for
18:    end for
19:    return G = (V, E)
20: end procedure
```

## 2.5 Rychle prozkoumávající náhodný strom

Druhý a novější z popisovaných pravděpodobnostních algoritmů se nazývá rychle prozkoumávající náhodný strom (Rapidly-exploring Random Tree). Algoritmus je dále v práci označován jako RRT. Článek [15] definuje algoritmus v prostoru stavů (namísto prostoru konfigurací), čímž zdůrazňuje, že byl navržen pro využití v modelech s řadou komplexních fyzikálních omezení. Tento důraz je v definici umocněn využitím vstupů řízení agenta jako ohodnocení hran stromu.

Základní verze algoritmu je oproti PRM v několika ohledech jednodušší – namísto grafu generuje strom, ve kterém je následné nalezení cesty triviální. Dále se namísto $n$ nejbližších vrcholů hledá jediný kandidát k propojení.

### 2.5.1 Popis algoritmu

Algoritmus RRT [15, 16] staví nad prozkoumanými částmi prostoru strom. Obdobně jako PRM je prostor modelu prozkoumáván náhodným vzorkováním. Vstupem algoritmu je množina překážek $P$ a počáteční konfigurace

$v_{start}$. Znalost cílové konfigurace $v_{goal}$ není vyžadována, nicméně může být využita v některých heuristikách vzorkování a ukončení. V základní verzi je vstupem též počet vrcholů $n$ ke vložení. Výstupem algoritmu je strom mapující prozkoumaný prostor.

Zařazením existujícího stromu $\tau$ anebo koncové konfigurace $v_{goal}$ mezi vstupy lze dosáhnout dodatečného rozšíření stromu (vynecháním řádky 2), respektive stavbu stromu využitelného k hledání cesty. Při stavbě stromu pro hledání cesty ukončovací podmínkou může být například vytvoření vzorku blízkého koncové konfiguraci $\rho(v_{new}, v_{goal}) < \epsilon$ nebo přímo úspěšné vložení $v_{goal}$ do stromu.

Základní myšlenkou algoritmu je propojování nových náhodně vygenerovaných konfigurací s nejbližším vrcholem grafu bezkolizní cestou. Kromě testu propojitelnosti by lokální plánovač měl mít na starost i určení vstupů řízení agenta. Obdobně jako u PRM se přesná implementace náhodného vzorkování i lokálního plánovače neuvádí a je ponechána na implementátorovi algoritmu.

Pseudokód algoritmu 2.4 popisuje variantu s lokálním plánovačem generujícím vstupy řízení agenta a s ukončením po vložení $n$ vzorků v prostoru konfigurací. Vstupem je počet vzorků $n$, množina překážek $P$ a počáteční konfigurace $v_{start}$. Posledním vstupem je okolí $r$. Tato základní verze algoritmu pouze demonstruje stavbu stromu mapujícího prostor, který není podkladem pro hledání cesty.

**Algoritmus 2.4** RRT

```
1:  procedure BUILD_RRT(v_start, P, n)
2:      τ.init(v_start)
3:      for i = 1 ... n do
4:          v_rand ← Sample(P)
5:          v_near ← argmin_{u∈τ.V}(ρ(u, v_rand))
6:          v_new ← SteerTo(v_rand, v_near)
7:          τ.addChild(v_near, v_new)
8:      end for
9:      return τ
10: end procedure
```

Na řádku 2 strom je inicializován, $v_{start}$ se stává kořenem stromu. Hlavní smyčka (řádky 3 až 8) v každé z $n$ iterací nejprve náhodně generuje procedurou *Sample* dočasný vrchol $v_{rand}$. Na řádcích 5 a 6 je nalezen vrchol $v_{near}$ stromu $\tau$ nejbližší od dočasného vrcholu $v_{rand}$ dle metriky $\rho$. Procedura *SteerTo* (řádek 6) poté na základě jejích vstupů $v_{rand}$ a $v_{near}$ vytváří nový vrchol $v_{new}$, jenž je propojitelný bezkolizní cestou s vrcholem $v_{near}$. Nově vy-

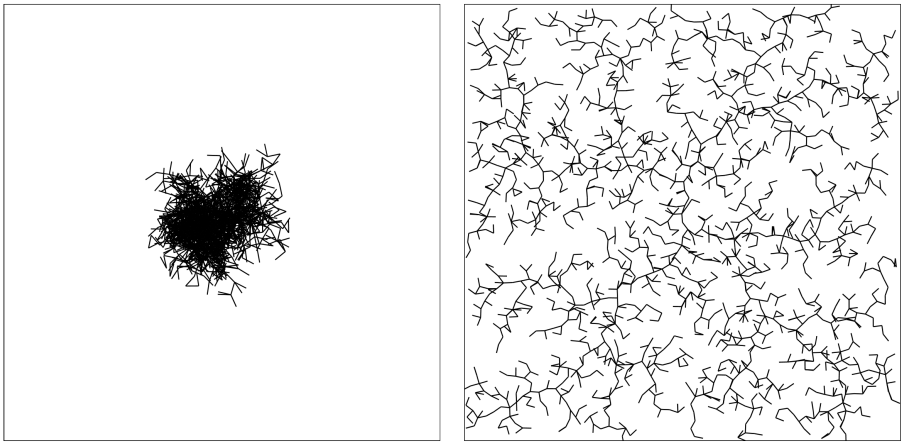
generovaný vrchol $v_{new}$ je posléze vložen (řádka 7) do stromu jako syn vrcholu $v_{near}$.

Stejně jako v algoritmu PRM se u procedury náhodného vzorkování prostoru (*Sample*) předpokládá, že jejím výstupem jsou konfigurace bezkolizní. Této procedury lze využít k řízení růstu stromu. Příkladem takového využití může být směřování k cílové konfiguraci nebo zaměření na část prostoru.

Procedura vytváří nové vrcholy *SteerTo* na základě nejbližšího vrcholu stromu a vrcholu náhodně generovaného. Tato procedura je obdobou procedury *CanConnect* algoritmu PRM (viz sekce 2.4.1). Na rozdíl od *CanConnect* procedura *SteerTo* vytváří konfiguraci novou. Cílem procedury *SteerTo* je vytvořit tuto konfiguraci co nejpodobnější parametru $v_{near}$. U *SteerTo* se nepředpokládá selhání, protože existuje triviální řešení $v_{new} \approx v_{near}$.

### 2.5.2 Vlastnosti algoritmu

Pro uvedení vlastností algoritmů je nejprve třeba definovat pojem Voroného diagramů. Voroného diagramy [1] jsou dělením prostoru na oblasti, buňky, jež dělí prostor na sféry vlivu. Nechť je daný prostor $M$ a množina $S \subset M$ středů buněk $s$. Buňka diagramu $b_s$ se sestává ze všech bodů $x \in M$, na něž má střed $s$ nejsilnější vliv ze všech možných středů buněk v $S$.



Obrázek 2.3: Naivní náhodný strom (vlevo) a rapidly exploring random tree (vpravo) [16]

Oproti naivnímu náhodnému stromu se RRT vyznačuje sklonem k rozšiřování nejméně prozkoumaných částí prostoru [15], viz obrázek 2.3. To je způsobeno faktem, že okrajové buňky Voroného diagramu jsou největší, a tudíž je pravděpodobnost rozšíření stromu z okrajové buňky vyšší. Náhodné

vzorkování má tedy vyšší šanci rozšířit „periferie“ stromu. Upřednostňování neprozkoumaných částí prostoru je velmi žádoucí vlastnost. Tato vlastnost směřuje expanzi stromu k neprozkoumaným částem prostoru.

Experimenty provedené v článku [15] zmiňují faktor 1,3 až 2,0 násobku délky nalezené cesty k délce cesty optimální. Dále se uvádí nezávislost výsledků a výkonu na volbě počáteční konfigurace $v_{start}$. Algoritmus RRT, podobně jako PRM, umožňuje využít binární detekce kolizí. Mezi další vlastnosti [16] patří:

**Distribuce vrcholů se blíží distribuci vzorkování.** Při expanzi stromu se postupně vyplňuje prostor, proto s $n \to \infty$ se pravděpodobnost $p$, že se nový vzorek nachází do vzdálenosti dosažitelné v $\Delta t$, blíží jedné. V tomto případě je nový vzorek přímo přidán do stromu a odpovídá tedy rozdělení vzorkování.

**Pravděpodobnostně kompletní.** Pravděpodobnost, že RRT bude obsahovat $x_{goal}$ se blíží jedné, když počet iterací $n \to \infty$.

**Spojitý v každé iteraci.** Oproti PRM ve stromě v každé iteraci algoritmu existuje cesta k počáteční konfiguraci (kořenu stromu).

### 2.5.3 RRT*

Tento algoritmus vychází z RRT. Obdobně jako PRM* s $n \to \infty$ asymptoticky nalézá cestu optimální. Algoritmus pracuje s $\mathcal{O}(n \log n)$ časovou složitostí [8].

Analogicky s PRM* je při vkládání nového vrcholu $v_{new}$ do stromu zvažováno několik kandidátů z okolí $r = min\{\gamma_{PRM}(\log n/n)^{1/d}, \eta\}$ (viz vztahy 2.2 v sekci 2.4.5). Nicméně oproti PRM* struktura zůstává stromem – $v_{new}$ je spojen s maximálně jedním z kandidátů.

Z kandidátů na připojení je vybrán takový, jenž minimalizuje cenu cesty (z kořene stromu, vrcholu $v_{start}$) do nového uzlu $c(v_{new})$. Po připojení nového vrcholu jsou v okolí upraveny údaje vrcholů o rodičovství, dojde-li jejich úpravou ke zlepšení ceny cesty z daného vrcholu.

Vstupem algoritmu je počáteční konfigurace $v_{start}$, množina překážek $P$, poloměr okolí kandidátů ke spojení $r$ a počet iterací $n$. Výstupem je strom $\tau$ mapující prostor modelu. Strom $\tau$ je optimální vzhledem k ceně cesty z jeho vrcholů do kořene stromu.

Pseudokód algoritmu na řádkách 2 až 6 postupuje identickým způsobem jako u RRT. Nejprve je kořen stromu inicializován počáteční konfigurací, procedura *SteerTo* vytváří novou konfiguraci $v_{new}$ z náhodného vzorku $v_{rand}$ procedury *Sample*. Hlavní smyčka běží $n$ iterací.

První změny přichází na řádcích 7 až 15. Nejprve je nalezena množina kandidátů k připojení $V_{near}$ v okolí $r$. Poté je z této množiny vybrán vrchol $v_{min}$, přes který je cena cesty (existuje-li) z $v_{new}$ do kořene nejkratší. Nový vrchol $v_{new}$ je posléze přidán do stromu jako syn vrcholu $v_{min}$ (řádek 16).

Řádky 17 až 21 se zabývají úpravou rodičovství. Všem vrcholům v okolí $V_{near}$ je změněn rodič na $v_{new}$, existuje-li cesta přes vrchol $v_{new}$ snižující cenu daného vrcholu.

**Algoritmus 2.5** RRT*

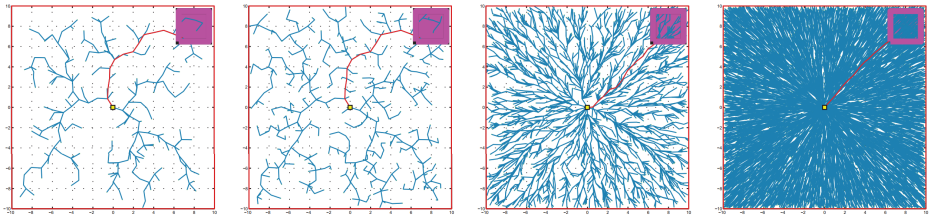
```
1:  procedure BUILD_RRT(v_start, P, r, n)
2:      τ.init(v_start)
3:      for i = 1 ... n do
4:          v_rand ← Sample(P)
5:          v_near ← argmin_{u∈τ.V}(ρ(u, v_rand))
6:          v_new ← SteerTo(v_rand, v_near)
7:          V_near ← u ∈ τ.V, ρ(u, v_new) < r
8:          v_min ← v_nearest
9:          c_min ← c(v_near) + ρ(v_near, v_new)
10:         for all u ∈ V_near do
11:             if (c(u) + ρ(u, v_new)) < c_min ∧ CanConnect(u, v_new) then
12:                 v_min ← u
13:                 c_min ← c(u) + ρ(u, v_new))
14:             end if
15:         end for
16:         τ.addChild(v_min, v_new)
17:         for all u ∈ V_near do
18:             if CanConnect(v_new, u) ∧ c(v_new) + ρ(v_new, u) < c(u) then
19:                 ChangeParent(u, c_new)
20:             end if
21:         end for
22:     end for
23:     return τ
24: end procedure
```

Procedury *Sample* a *SteerTo* jsou identické s procedurami RRT stejného jména (viz sekce 2.5.1). Procedura *CanConnect* pak odpovídá stejnojmenné proceduře algoritmu PRM (viz 2.4.1). U procedury *ChangeParent* je důležité zajistit konzistenci stromu – nejenom, že je potřeba nastavit vrcholu nového rodiče, ale i odpovídajícím způsobem upravit informace o synech a cenách

vrcholů. Cena $c(v)$ vrcholu $v$ je rekurentně definována vztahy 2.3,

$$\begin{aligned} c(v_{root}) &= 0, \\ c(v) &= c(v') + \rho(v', v) \end{aligned} \tag{2.3}$$

kde $v_{root}$ je kořenem stromu (odpovídajícím počáteční konfiguraci) a $v'$ je rodičem vrcholu $v$.



Obrázek 2.4: Nalezená cesta (purpurová) a strom algoritmu RRT* s rostoucím počtem uzlů [8]

Jedná se o tzv. *anytime* typ algoritmu – nejprve je (rychle) nalezena cesta, jejíž cena je pak s rostoucím počtem vrcholů postupně snižována, viz obrázek 2.4.

### 2.5.4 Další varianty RRT

Alternativa RRT*, kdy výběr kandidátů uvažuje $k$ nejbližších uzlů stromu namísto všech vrcholů v okolí $r$, se nazývá k-nearest RRT*.

Další variantou RRT je Informed RRT*[4] stavící na RRT* s tím rozdílem, že po prvotním nalezení cesty je vzorkování nových vrcholů zaměřeno na elipsoid, v němž se cesta nachází. Cílem této změny je urychlit algoritmus zamítáním vzorků, u kterých je pravděpodobnost vylepšení nalezené cesty minimální.

Varianta algoritmu, T-RRT [7] (Transition based RRT), pak rozšiřuje původní algoritmu o využití cenové mapy s cílem vylepšit kvalitu výsledné cesty.

## 2.6 Srovnání PRM a RRT

Článek [8] věnující se analýze složitosti algoritmů nabízí srovnání algoritmů po stránce složitosti a schopnosti nalézt asymptoticky cestu optimální, tyto poznatky jsou shrnuty v tabulce 2.1.

Tabulka 2.1: Porovnání složitostí popsaných algoritmů [8]

| algoritmus | asymptoticky optimální | časová složitost | paměťová složitost |
|---|---|---|---|
| PRM | ne | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n)$ |
| sPRM | ano | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ |
| PRM* | ano | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n \log n)$ |
| RRT | ne | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n)$ |
| RRT* | ano | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n \log n)$ |

Jak je patrné z tabulky 2.1, všechny algoritmy, kromě sPRM mají stejnou časovou složitost. Nalezení asymptoticky optimálních cest pak nepřekvapivě zvyšuje složitost paměťovou. Hlavním rozdílem mezi PRM skupinou algoritmů a RRT jsou tedy jejich vlastnosti – zatímco PRM předpokládá mnohanásobné dotazování nad vytvořeným grafem, RRT je navrženo k odpovězení jediného dotazu. RRT ovšem nabízí *anytime* vlastnosti spolu s garancí existence cesty do kořene (počáteční konfigurace).

# 3 Řešení

Tato kapitola se věnuje řešení cíle této práce – úkolem bylo navrhnout a implementovat algoritmus plánování cesty sond v proteinových modelech. Tyto sondy jsou složeny z několika sfér.

Sekce 3.1 se zabývá popisem implementovaných variant algoritmu RRT*. Tyto varianty byly vystavěné na základě teoretické části práce (kapitola 2), zejména pak na částech věnujících se RRT*, srovnání algoritmů a na sekci 2.3, která definuje řešený problém.

V sekci 3.2 se analyzují rizika implementace projektu, na nichž jsou pak založena kritéria hodnocení a určuje užší výběr technologií. Dále se pak posuzuje vhodnost použití vybraných technologií pro implementaci projektu vzhledem k určeným kritériím. V neposlední řadě se na konci sekce zdůvodňuje výběr použité technologie.

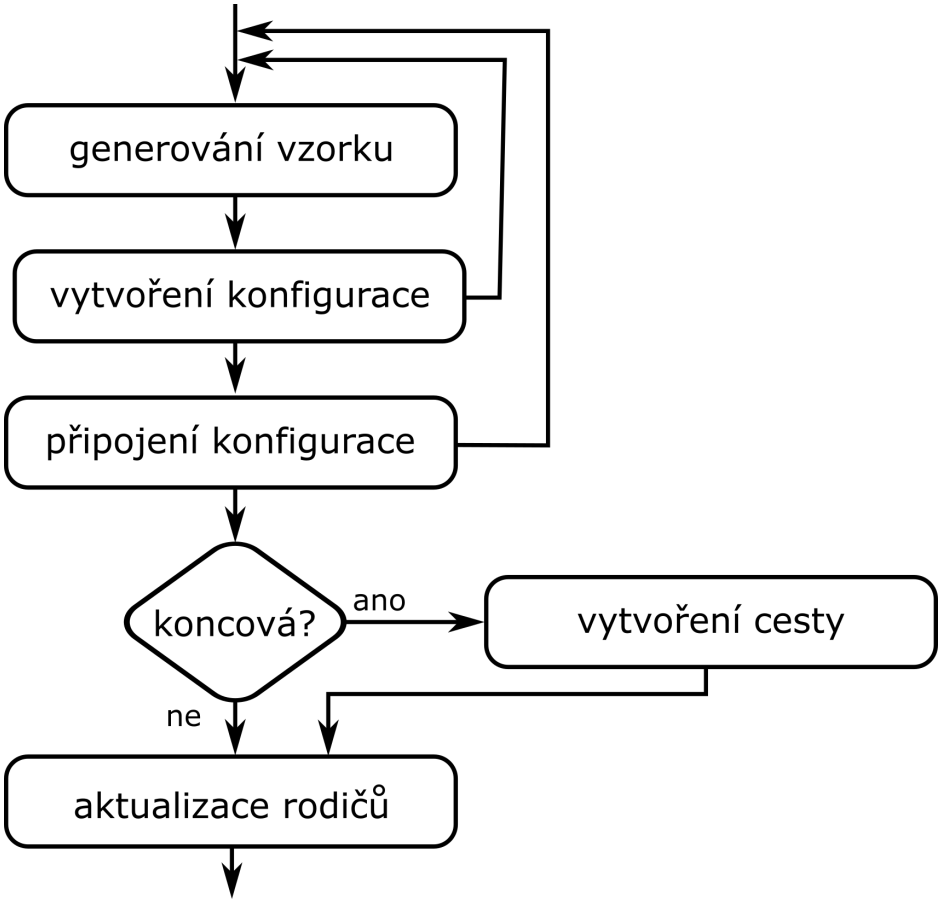
Sekce 3.3 se zabývá popisem řešení ze softwarového hlediska. Zejména se pak věnuje architektuře, použitým technologiím a navrženým rozhraním. Prostor je zde věnován i dokumentaci, testování a kvalitě kódu.

## 3.1 Algoritmus

Na základě sekce 2.6 a po domluvě s vedoucí práce byl vybrán k implementaci algoritmus založený na RRT*. Vhledem k pevně dané počáteční konfiguraci nebylo třeba uvažovat znovuvyužití struktury na další dotazy, což je hlavní výhodou algoritmů založených na PRM. Lze též očekávat že udržování stromu bude jednodušší. Popis algoritmu lze nalézt v sekci 2.5.3.

Zjednodušený přehled jedné iterace implementovaného algoritmu vytvoření nového vrcholu lze shrnout do vývojového digramu na obrázku 3.1. Z diagramu lze též vyčíst drobné odchýlení od verze popsané v literatuře – při generování vzorku není vyžadována bezkolizní konfigurace. Vygenerovaná konfigurace se v následující fázi často mění, proto by byl přínos složitějšího algoritmu generování vzorků minimální.

V algoritmu též byla zjednodušena procedura *SteerTo*, od níž původní algoritmus vyžaduje neomylné vytvoření vždy připojitelné konfigurace. Tyto nároky se ukázaly jako příliš omezující. Takové nároky by navíc neúměrně zvyšovaly složitost této fáze. Z diagramu lze tedy vyčíst, že fáze vytváření konfigurace může selhat (např. u řešení $v_{new} \approx v_{near}$ dochází k odmítnutí) a připojení nové konfigurace se nemusí zdařit.




Obrázek 3.1: Vývojový diagram vytvoření nového vrcholu stromu

Vzhledem k tomu, že cílem algoritmu je nalézt všechny existující cesty v modelu z počátečního vrcholu do cílové oblasti mimo protein, zastavovací podmínka není definována. Volba počtu kroků (příp. délka přiděleného výpočetního času), po kterých je výpočet zastaven, je ponechána na uživateli.

Následující sekce popisují jednotlivé fáze algoritmu dle vývojového diagramu 3.1 a zabývají se možností konfigurace, upravitelnosti a implementovaných variant. Sekce 3.1.1 se zabývá problematikou vzorkování prostoru modelu. Sekce 3.1.2 popisuje fázi generování správných konfigurací, dále sekce 3.1.3 se zbývá připojitelností výstupů předchozí fáze do stromu. Sekce 3.1.4 se věnuje jak identifikaci koncových vrcholů, tak i vytváření cest. Sekce 3.1.5 probírá problematiku úprav stromu po připojení nové konfigurace, zejména aktualizaci rodičů blízkých vrcholů.

### 3.1.1 Generování vzorků

Fáze generování vzorků odpovídá proceduře *Sample* referenčního algoritmu. Této verzi je však umožněno generovat konfigurace kolizní.

Při hledání cesty v modelech proteinů se problematika vzorkování zjednodušuje hustým a rovnoměrným rozložením překážek v modelu. Navíc, cílem algoritmu bylo nalezení všech existujících cest – lze tedy předpokládat, že vzorkování nebude využito pro směřování růstu stromu.

Ze zmíněných důvodů bylo implementováno pouze AABB (osově orientovaný ohraničující box) rovnoměrné náhodné vzorkování. S přihlédnutím k identifikaci koncových konfigurací v sekci 3.1.4 lze uvažovat i jiném, než minimálním ohraničujícím objemu. Generování $n$-dimenzionálních vzorků v této implementaci probíhá po složkách, pseudokód generování je zachycen v algoritmu 3.1. Procedura *PRNG* je generátorem pseudonáhodných čísel s rovnoměrným náhodným rozdělením.

**Algoritmus 3.1** Generování AABB vzorků

```
1: procedure GENERUJ_VZOREK(v_min, v_max, mult, n)
2:     min ← v_min − (v_max − v_min) * mult
3:     max ← v_max + (v_max − v_min) * mult
4:     v ← 0
5:     for i = 0 ... n do
6:         v[i] ← PRNG(min[i], max[i])
7:     end for
8:     return v
9: end procedure
```

Ve výchozí implementaci lze volitelně využít násobitele minimálního objemu i počáteční semínko pseudonáhodného generátoru. Fáze generování vzorků je plně nahraditelná implementacemi odpovídajících rozhraní (viz příloha B).

### 3.1.2 Vytvoření konfigurace

Fáze vytváření zhruba odpovídá proceduře *SteerTo* referenčního algoritmu, nicméně test připojitelnosti je odložen do následující fáze. Za správný výstup se považuje konfigurace bezkolizní, žádné další požadavky na konfiguraci kladeny nejsou. V souladu s referenčním algoritmem se však předpokládá maximální podobnost s vstupní konfigurací $v_{rand}$.

Vstupem procedury Algoritmu 3.2 je nejbližší vrchol stromu $v_{near}$ a náhodná konfigurace $v_{rand}$ vytvořená na základě výstupu předchozí fáze algo-

ritmu spolu s množinami překážek $P$ a uzávěrů $U$ (více v sekci 3.1.4). Konfigurace $v_{rand}$ je předávána odkazem. Původní algoritmus [15] též využívá omezení maximální vzdálenosti připojované konfigurace od vrcholu stromu omezením $\Delta t$. Tato konstanta je k dispozici implementacím této fáze spolu s doporučeným počtem pokusů o vytvoření bezkolizní konfigurace.

**Generování konfigurací půlením vzdáleností**
První dvě varianty této fáze využívají půlení vzdálenosti při každém neúspěšném pokusu – v případě vytvoření konfigurace, jež je v kolizi, se $v_{new}$ přesouvá na poloviční vzdálenost ve směru $v_{near}$. Přesněji řečeno, půlí se parametr $t$ lineární interpolace (procedura $Mix$). Ve výchozí implementaci jsou lineárně interpolovány pozice center konfigurací $v_{near}$ a $v_{rand}$, zatímco v alternativní implementaci (Algoritmus 3.2) se interpolují všechny dimenze konfigurace (tzn. včetně orientace).

**Algoritmus 3.2** Základní generování konfigurací

```
1:  procedure GENERUJ_KONFIGURACI(v_near, v_rand, P, U)
2:      t ← Min(1, max_dist/|v_near − v_rand|)
3:      if |v_near − v_rand| > max_dist then
4:          v_rand ← Mix(v_near, v_rand, t)
5:      end if
6:      attempts ← 0
7:      while v_new.Collides(P) or v_new.Collides(U) do
8:          attempts ← attempts + 1
9:          if attempts > max_attempts then
10:             return false
11:         end if
12:         t ← t/2
13:         v_new ← Mix(v_near, v_new, t)
14:     end while
15:     return true
16: end procedure
```

**Generování konfigurací na kolmé kružnici**
Další realizovaný algoritmus (Algoritmus 3.3) této fáze vytváří $n$ konfigurací rovnoměrně rozmístěných na kružnici kolmé ke směru vektoru $dir_o$ (vztah 3.1)

$$dir_o = v_{near}.Center - o.Center \tag{3.1}$$

kde $v_{near}.Center$ je pozice konfigurace nejbližšího vrcholu stromu a $o.Center$ je pozice středu překážky kolidující s novou konfigurací $v_{new}$. Tato kružnice má minimální poloměr zajišťující bezkoliznost s hlavní sférou sondy (tzn. se středem v pozici konfigurace). Obdobně jako u výchozí varianty, nachází-li se nová konfigurace příliš daleko od nejbližší, je nová konfigurace nejprve přesunuta lineární interpolací na maximální přípustnou vzdálenost (řádky 2 až 5).

Je-li konfigurace v kolizi, je na základě pozice a poloměru překážky s níž koliduje vypočítán poloměr $s$ a bázové vektory roviny kružnice $bv_1$ a $bv_2$ (řádky 8 až 13). Poté je vzorek přesouván po této kružnici s krokem $\frac{2\pi}{max\_attempts}$ až do vyčerpání $max\_attempts$ pokusů (řádky 15 až 21).

**Algoritmus 3.3** Generování konfigurací na kolmé kružnici

```
1:  procedure GENERUJ_KONFIGURACI(v_near, v_new, P, U)
2:      t ← Min(1, max_dist/|v_near − v_new|)
3:      if |v_near − v_new| > max_dist then
4:          v_new ← Mix(v_near, v_new, t)
5:      end if
6:      obstacle ← v_new.FirstColliding(P)
7:      if Exists(obstacle) then
8:          b ← cos(φ)(obstacle.Radius + v_new.Radius)
9:          a ← sin(φ)(obstacle.Radius + v_new.Radius)
10:         dir ← (v_near.Center − v_new.Center).Normalized
11:         center ← obstacle.Center + b * dir
12:         bv_1 ← [center.z, 0, −center.x].Normalized
13:         bv_2 ← Cross(dir, bv1).Normalized
14:         i ← 0
15:         while i < 2π do
16:             v_new.Center ← center + bv_1 * a  * cos(i) + bv_2 * b * sin(i)
17:             if !v_new.Collides(P)  & !v_new.Collides(U)  then
18:                 return true
19:             end if
20:             i ← i  + 2π/max_attempts
21:         end while
22:         return false
23:     end if
24:     return !v_new.Collides(U)
25: end procedure
```

Procedura *FirstColliding* vrací první překážku, s níž konfigurace koliduje,

nebo hodnotu prázdnou, je-li konfigurace bezkolizní. Vlastnost *Normalized* vrací normalizovaný vektor, zatímco procedura *Cross* vrací vektorový součin parametrů.

Výstupem této fáze řešení je informace o úspěšnosti vytvoření konfigurace – její vytvoření z daného vzorku může selhat vyčerpáním doporučeného počtu pokusů o vytvoření. V případě selhání této fáze se proces vytváření nového vrcholu vrací na začátek a generuje se nový vzorek. Procedura *Collides* testuje, zda je konfigurace kolizní. Samotný výpočet kolizí sondy v používaném modelu je triviální, neboť se navigovaná sonda i překážky skládají ze sfér.

### 3.1.3 Připojení konfigurace

Připojení konfigurace zhruba odpovídá řádkám 10 až 15 v algoritmu 2.5, ovšem nepředpokládá spojitelnost $v_{near}$ a nejbližšího vrcholu stromu $v_{near}$. Důvodem této změny je v předchozích sekcích uvedené zjednodušení fáze vytváření konfigurací. Algoritmus procedury *CanConnect* předpokládá využití diskrétního výpočtu kolizí, nicméně spojitá detekce kolizí zakázána není. Implementace této procedury využívají několika iterací algoritmu diskrétní detekce kolize z fáze vytváření konfigurace.

Vstupem této fáze je bezkolizní konfigurace k připojení $v_{new}$ a konfigurace nejbližšího vrcholu stromu $v_{near}$. Dále je předána množina překážek $P$, uzávěrů $u$ a poloměr blízkého okolí $r_{near}$. Výsledkem fáze je vložený (řádky 13 až 17) nový vrchol stromu $\tau$ anebo odmítnutí, není-li $v_{new}$ ke stromu připojitelný.

Konfigurovatelné jsou konstanty $r_{near}$ a implementace metody *CanConnect* ověřující existenci bezkolizní spojitelnost přechodu stavů. Jako výchozí je použita diskrétní kolize s volitelným počtem kroků.

**Algoritmus 3.4** Základní generování konfigurací

```
1: procedure PRIPOJ_KONFIGURACI(v_near, v_new, P, U, r_near)
2:     x_near ← Near(v_new, r_near)
3:     v_min ← NIL
4:     cost_min ← MAX_VALUE
5:     for all x ∈ x_near do
6:         if CanConnect(x, v_new, P, U) then
7:             if x.Cost + x.CostTo(v_new < cost_min) then
8:                 cost_min ← x.Cost + x.CostTo(v_new
9:                 v_min ← x
10:            end if
11:        end if
12:    end for
13:    if v_min! = NIL then
14:        v_new ← cost_min
15:        τ.V ← τ.V ∪ v_new
16:        τ.E ← τ.E ∪ {(v_min, v_new)}
17:    end if
18:    return v_min! = NIL
19: end procedure
```

### 3.1.4 Vytvoření cesty

Tato fáze (Algoritmus 3.5) ověřuje, zda nově přidaný vrchol lze označit za koncový. Fáze zahrnuje i heuristiku uzavírání tunelu, jímž případná nová cesta vede, s cílem zabránit nežádoucí expanzi stromu do volného prostoru mimo model.

Jako koncová konfigurace je označena (řádka 5) ta, jejíž pozice je minimálně v určité vzdálenosti (konstanta) od nejbližší překážky. Po takovém označení je její vrchol přidán do seznamu cest $C$ jako počátek nové cesty (řádka 9). Vrcholy-předkové jsou dále rekurentně příslušně označeny jako vrcholy cesty (řádky 11 až 13). Strom též uchovává informaci o nejkratší nalezené cestě, jež je v tomto okamžiku případně aktualizována (řádky 14 až 17). Uzavírání je uskutečněno vložením nové umělé zábrany (řádek 8), koule s poloměrem rovným násobku vzdálenosti nejbližší překážky a středem na pozici nové konfigurace do seznamu uzávěrů $U$.

Vstupem této fáze je nově vložený vrchol $v_{new}$, množiny překážek $P$ a uzávěrů $U$ spolu s množinou cest $C$. Algoritmu též využívá konstanty minimální vzdálenosti pro označení vrcholu za koncový *path_threshold*. Výstupem této fáze jsou patřičně aktualizované množiny $U$ a $C$.

**Algoritmus 3.5** Vytvoření cesty

```
1: procedure IsGoal(v_new, P)
2:     return Min(|v_new.Center − p|, p ∈ P) > path_threshold
3: end procedure
4: procedure VYTVOR_CESTU(v_new, P, U, C)
5:     if !IsGoal(v_new, P) then
6:         return
7:     end if
8:     U ← {(v_new.Center, Min(|v_new.Center − p|, p ∈ P))}
9:     C ← v_new
10:    v ← v_new
11:    while v! = NIL do
12:        v.OnPath ← true
13:    end while
14:    if c_best > v_new.Cost then
15:        c_best ← v_new.Cost
16:        v_best ← v_new
17:    end if
18: end procedure
```

Jedinou možností konfigurace této fáze nastavení konstanty *path_threshold*. Samotná metoda *IsGoal* nahraditelná není.

### 3.1.5 Aktualizace rodičů

Aktualizace rodičů je založena na řádkách 20 až 25 v Algoritmu 2.5. Reálná implementace uchovává váhy vrcholů v mezipaměti, což vede k problému aktualizace těchto údajů při změnách ve stromě. Tento problém adresuje tato fáze.

Změna rodiče (Algoritmus 3.6) na nově přidanou konfiguraci $v_{new}$ je provedena, snižuje-li cenu vrcholu $v$ z okolí poloměru $r_{near}$. Poloměr okolí $r_{near}$ je stejný jako v sekci připojení konfigurace 3.1.3). Váhy jsou upraveny všem potomkům vrcholu, jemuž byl změněn rodič.

**Algoritmus 3.6** Změna rodiče

```
1: procedure ZMEN_RODICE(v, v_new, new_cost)
2:     v.Parent.Children ← v.Parent.Children\{v}
3:     v.Parent ← v_new
4:     diff ← v.Cost − new_cost
5:     for all d ∈ v.Descendants do
6:         d.Cost ← d.Cost − diff
7:     end for
8:     v_new.OnPath ← v.OnPath||v_new.OnPath
9: end procedure
```

Fáze algoritmu aktualizace rodičů nenabízí žádné možnosti úprav nebo konstanty k definování kromě konstanty $r_{near}$, jež sdílí s fází připojení konfigurace.

## 3.2 Analýza dostupných technologií

Jedním z klíčových aspektů úspěšného projektu je vhodná volba technologie. Tato sekce se proto snaží tuto problematiku analyzovat. Hlavními kritérii pro výběr technologie k hodnocení byla její podpora, vlastnosti a zkušenosti autora s touto technologií. Širší výběr technologií se snažil obsáhnout pouze ty nejdůležitější zástupce.

### 3.2.1 Rizika

Mezi identifikovaná rizika volby technologie patří hlavně nedostatečná úroveň abstrakce vedoucí k nedosažení zvoleného rozsahu práce v přiděleném čase. Mezi další rizika patří také nedostatečná zkušenost autora s technologií, jež by mohla vést ke zpomalení vývoje, případě snížit kvalitou kódu. S nižší kvalitou kódu také souvisí obtížnější předání, případně i horší rozšiřitelnost práce. V neposlední řadě je nutné též zmínit modularitu, jejíž nedostatek může vést k neefektivnímu využití času spojeného s implementaci funkcí, jež jsou pro jiné technologie dostupné v základu nebo po integrace middleware/pluginů.

Rizikem by také mohla být nízká obecná obeznámenost s technologií mezi skupinou lidí, které bude práce potencionálně předána k dalšímu vývoji. Důležitost tohoto rizika ovšem záleží až na výsledcích práce.

### 3.2.2 Kritéria hodnocení

Součástí práce je i návrh vylepšení algoritmu jenž bude hodnocen v prostorech s vysokým počtem dimenzí. Proto musí být kladen velký důraz na

možnosti vizualizace modelu, výsledků i průběhu výpočtu algoritmu. To vede k požadavku na intuitivní tvorbu robustního uživatelského rozhraní.

Do kritérií je také nutné zařadit modularitu související s flexibilitou výsledného řešení. Mezi kritéria nebyly zařazeny požadavky na výkon technologie, a to z důvodů obtížného objektivního hodnocení před vypracováním prototypu. Z tohoto důvodu je výkon hodnocen pouze ústně, a to pouze v extrémních případech.

Váhy přiřazované jednotlivým kritériím jsou na škále od nízké přes střední po vysokou váhu. Kritéria a jejich váhy shrnuje tabulka 3.1.

Tabulka 3.1: Kritéria hodnocení technologií

| Kritérium | Váha |
|---|---|
| Abstrakce grafického API | vysoká |
| Snadnost tvorby GUI | střední |
| Modularita | střední |
| Zkušenost autora | střední |
| Úroveň dokumentace | nízká |

### 3.2.3 Vybrané technologie

Prvními volbami je využití jazyků C++ nebo Java spolu s OpenGL API a s příslušnými knihovnami řešící jednotlivé požadované funkce. Dalšími volbami bylo využití již hotových frameworků pro zmiňované technologie. Jako zástupci byli vybráni Ogre3D a LibDGX. Poslední skupinou jsou komplexní frameworky, do této skupiny byl zařazen WPF (zástupce moderního GUI frameworku) spolu s Unreal Engine 4 a Unity 3D (zástupci pokročilých *engine*).

Zařazení řešení pomocí technologií nižších úrovní bylo z důvodu obavy nedostatečné možnosti konfigurace komplexnějších řešení. Roli v tomto rozhodnutí hrála i poměrně vysoká zkušenost autora s tímto typem řešení a existující kódová základna (*codebase*) příbuzných projektů na katedře.

### 3.2.4 Hodnocení

Za výchozí hodnocení je považováno střední, v případě že nelze mít zásadní námitky k řešení daného kritéria technologií. Nicméně technologie se středním hodnocením také nenabízí žádné zvláštní výhody oproti konkurenci. Následující odstavce zdůvodňují udělená hodnocení shrnutá v tabulce 3.2.

Důvodem udělení nízkého hodnocení snadnosti tvorby GUI a abstrakce prvním dvěma řešením (C++ A Java s middleware) je nízkoúrovňový přístup

k hardware přes OpenGL API. Možným protiargumentem je existence specializovaných GUI knihoven (např. Qt), nicméně jejich integrace do OpenGL scény se v praxi často ukazuje jako problematická (správa kontextů, nedostatečná podpora nových verzí OpenGL atd.).

V případě modularity mluví jasně ve prospěch Javy existence nástrojů typu Maven, ač pro C++ existují podobné nástroje – např. projekt Conan, jejich podpora není ani zdaleka taková. Manuální integrace a údržba C++ knihoven je bezesporu obtížná.

Zatímco zkušenosti s oběma technologiemi jsou na dobré úrovni, řešení s využitím Javy s největší pravděpodobností nabídne lepší dokumentaci díky rozšířenému využívání automatizovaných nástrojů a široké komunitě.

Zatímco technologie LibGDX je ucelenější, Ogre3D je vysoce modulární agregací middleware. Obě technologie nabízí slušnou úroveň abstrakce grafického API i pohodlí tvorby GUI. V obou kritériích mírně vede Ogre3D díky vysoce modulární architektuře – na výběr je i ze skoro tuctu GUI knihoven.

Jak bylo zmíněno, zatímco Ogre3D nabízí vysokou modularitu, LibGDX ničím v tomto ohledu nepřekvapuje. Ogre3D dostává přes vysokou úroveň dokumentace pouze hodnocení střední, protože je kvůli zmíněné modularitě často třeba využívat dokumentace třetích stran, které této úrovně nedosahují. Ač je pokrytí dokumentace u LibGDX skoro stoprocentní díky automaticky generované dokumentaci, samotný obsah vygenerované dokumentace je místy obtížně srozumitelný či nedostatečný (absence popisu) a proto dostává hodnocení pouze nízké.

Přes vysokou abstrakci grafického API v případě Unreal 4 a Unity3D, bylo uděleno hodnocení pohodlí tvorby GUI pouze střední kvůli relativní nemotornosti jejich GUI API. Ani jedna z těchto technologií nebyla navržena pro tvorbu bohatého uživatelského rozhraní. V posledním zmiňovaném kritériu ovšem mírně vede Unity3D díky vysoké modularitě a možnosti nahrazení celého GUI systému za systém třetí strany. Důvodem udělení nízkého hodnocení Unreal 4 je obvykle obtížnější integrace a celkově nižší podpora rozšíření. Dokumentace obou technologií je bezchybná, nicméně Unity3D si vytváří drobný náskok díky aktivnější komunitě. Za zmínku také stojí, že Unity3D obsahuje i JavaScript backend, což může usnadnit orientaci v Unity3D vyššímu počtu lidí.

Poslední hodnocenou technologií je WPF. Zástupci moderního GUI frameworku nelze neudělit jiné než vysoké hodnocení pohodlí tvorby GUI. Abstrakci grafického API je ovšem obtížné hodnotit – zatímco je abstrakce v rámci frameworku nejvyšší z hodnocených řešení, toto API je v 3D aplikacích obvykle nahrazeno adaptérem Direct3D nebo OpenGL API.

Přestože WPF nabízí správu balíčků (*package management*) NuGet, fle-

xibilitu WPF sráží závislost na platformě Windows[1]. Navíc tento framework nebyl navržen pro použití v 3D aplikacích, a proto je nabídka modulů v této oblasti velmi omezená. Dokumentace WPF je výborná, podpora komunity široká.

Po výkonnostní stránce lze zařadit produkty do tří kategorií, Java a C++ s middleware a WPF zastupují kategorii technologií neoptimalizovaných pro 3D, zatímco ostatní řešení nabízejí již v základu slušnou optimalizaci vykreslování 3D scén.

Tabulka 3.2: Hodnocení technologií

| technologie | Snadnost tvorby GUI | Abstrakce grafického API | Modularita | Zkušenost autora | Úroveň dokumentace |
|---|---|---|---|---|---|
| Java s middleware | nízká | nízká | vysoká | střední | vysoká |
| C++ s middleware | nízká | nízká | nízká | střední | střední |
| LibDGX | střední | střední | střední | nízká | nízká |
| Ogre3D | střední | střední | vysoká | nízká | střední |
| Unreal 4 | střední | vysoká | nízká | nízká | střední |
| Unity3D | střední | vysoká | vysoká | střední | vysoká |
| WPF | vysoká | nízká | střední | vysoká | vysoká |

### 3.2.5 Závěr

První čtyři řešení nejsou doporučena z důvodů buď nízké abstrakce, nedostatečných zkušeností anebo neuspokojivé dokumentace.

Mezi zvažovatelná řešení lze počítat Unreal 4 a WPF. Nicméně Unreal4 naráží na nižší modularitu a omezené zkušenosti autora s technologií, zatímco WPF naráží na problémy s výkonem (známé díky zkušenostem s podobným projektem).

Jako nejvhodnější řešení se jeví Unity3D díky minimalizaci rizik a vysoké abstrakci i modularitě slibující rychlý vývoj aplikace.

## 3.3 Implementace

Přestože v zárodku projektu první 2D experimenty s algoritmy probíhaly na implementaci v jazyce C++, záhy se údržba všech závislostí ukázala příliš

[1] WPF lze v omezené míře využít spolu s frameworkem Xamarin jež je multiplatformní

časově náročná. Navíc vývojový cyklus značně zpomalovaly požadavky na nové funkce, protože typicky ke každé nové funkci bylo nejprve třeba navrhnout a implementovat nízkoúrovňové řešení v OpenGL nebo integrovat novou nativní knihovnu do projektu.

Uvedené problémy vedly k vypracování analýzy dostupných technologií, která na základě nabytých zkušeností kladla vysoký důraz na míru abstrakce. Tato analýza je popsána v sekci 3.2. Na základě této analýzy byla s přechodem do 3D vypracována implementace v prostředí Unity 5.6.0f3 se skriptovacím backendem pro jazyk C#. To je verze vypálená na CD a verze, na níž byly provedeny experimenty v kapitole 4.

Následující sekce pak popisují toto řešení – sekce 3.3.1 představuje prostředí Unity, sekce 3.3.2 slouží jako přehled řešení, sekce 3.3.3 se věnuje vstupním a výstupním formátům, sekce 3.3.4 pak specifikům uživatelského rozhraní. Sekce 3.3.5 se zaobírá dokumentací, dokumentovaností, správností a zajištění kvality kódu. Konečně sekce 3.3.6 hledí do budoucnosti projektu a nabízí možné cesty vylepšení a rozšíření.

### 3.3.1 Prostředí Unity a ECS

Takto sekce si dává za úkol představit prostředí Unity a celkový pohled na architekturu aplikace. Samotný framework Unity je příkladem návrhového vzoru ECS[1] (Entity Component System), z nejhrubšího pohledu tedy aplikace implementuje tento návrhový vzor. Protože se jedná o jeden z méně známých vzorů v odvětví vývoje softwaru, následující odstavce tento návrhového vzor krátce představují.

V ECS [19] objekty jsou reprezentovány entitami skládajícími se z komponent. Komponenty reprezentují vlastnosti entit v simulaci, například fyzické (tuhé těleso) nebo grafické znázornění (geometrický model s materiálem). Systémy jsou moduly zodpovídající za aspekty simulace (např. fyzikální engine, modul renderování). Pro zařazení do smyčky systému musí entita vlastnit potřebné komponenty. Příklad entity je zachycen v příloze na obrázku D.1.

Výhodami ECS je vysoká úroveň oddělení zodpovědností a jednoduché dynamické sestavování objektů při běhu programu. Využívá se skladby namísto dědičnosti, což vede k mnohem vyšší flexibilitě, znovupoužitelnosti a rozšiřitelnosti než u OOP (objektově orientované programování) vzorů.

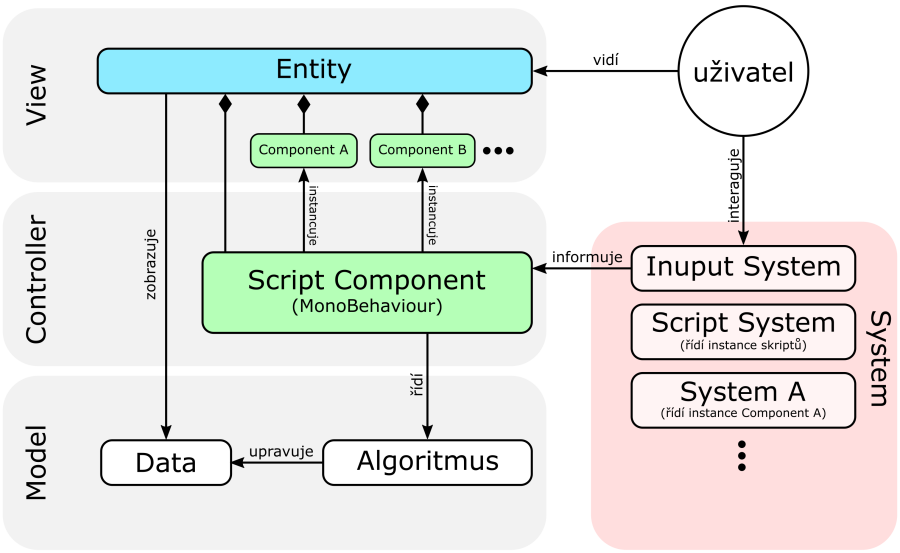
Z názvosloví použitého v Unity stejně jako ze způsobu použití technologie je zřejmé, že se jedná o návrhový vzor ECS. Bohužel oficiální doku-

[1] představen na GDC 2002 Scottem Bilasem, přednášku lze nalézt na `http://gamedevs.org/uploads/data-driven-game-object-system.pdf`

mentace neposkytuje dostatek podkladů k detailnějšímu popisu architektury frameworku samotného.

### 3.3.2 Architektura

Vzhledem k tomu, že jedním ze systémů ECS je i systém spravující skripty, je potřeba v rámci systému skriptů také organizovat kód. Zde přichází ke slovu návrhový vzor MVC (*Model-view-controller*) a objektově orientované programování. Vzhledem k rozšíření pravděpodobně není třeba OOP ani vzor MVC blíže představovat. Kombinace stylu frameworku a aplikce vytváří architekturu zachycenou na obrázku 3.2.




Obrázek 3.2: Diagram zachycující makro-architekturu aplikace

Aplikace samotná spolu se skripty volně implementuje návrhový vzor MVC. Data modelu aplikace se skládají z doposud vytvořeného stromu a modelu proteinu. Oba zdroje dat jež jsou centrem výpočtu algoritmu nejsou perzistentně uloženy v databázi. Aplikace nicméně podporuje serializaci nastavení i modelů do souborů. Serializace slouží jako datové rozhraní s existujícími aplikacemi, bližší popis lze nalézt v sekci 3.3.3. Těžko tedy zde mluvit o modelu ve smyslu, jaký lze například nalézt u typických webových aplikací.

Rolí controllerů v aplikaci je generování komponent a sestavování entit z dat modelu, nad kterým pracuje algoritmus. Všechny controllery jsou potomky tříd MonoBehaviour nebo ScriptableObject. Komponenty řízené skriptovacím systémem Unity musí být potomky jedné z těchto tříd.

View vrstva je pak realizována odpovídajícími komponentami (například MeshRenderer nebo LineRenderer). Tyto komponenty jsou součástí frameworku. Za součást view vrstvy lze pokládat i GPU (*shader*) programy jež jsou přiřazené materiálům. Materiálem se rozumí popis vykreslovaní povrchu objektů.

Mimo MVC vrstvu stojí rozšíření prostředí a úpravy procesů v editoru spolu s tzv. assety a šablony. V názvosloví Unity je asset balíkem skládajícím se ze zapouzdřeného kódu, sítí anebo materiálů. Šablonou (*prefab*) se v názvosloví Unity rozumí vzor, podle něhož lze vytvářet určitý typ entity (např. atom proteinu nebo sonda). Příklad takové šablony je zachycen v příloze na obrázku D.1.

V aplikaci se využívá programování vůči rozhraní a proudové zpracování dat pomocí rozhraní IEnumerable<T>[1]. Paralelizmus algoritmu je vláknový – hlavním důvodem této volby byla snaha o maximální izolaci od smyčky frameworku, jež by mohla zásadně ovlivňovat výsledky experimentů. Ostatní (pseudo)paralelizmus je implementovaný pomocí koprogramů Unity[2]. Koprogramy běží pod hlavním vláknem aplikace.

### 3.3.3 Vstupní a výstupní formáty

Vstupem program je seznam překážek, atomů proteinu. Výchozím formátem je jednoduchý textový formát, kde každé řádce odpovídá jedna sféra s trojrozměrnou pozicí středu (souřadnice se předpokládají v pořadí x, y, z) následovaná poloměrem. Příklad modelu takto serializovaného lze nalézt na přiloženém CD.

Vstupně-výstupním formátem je nastavení algoritmu, jedná se o serializovaný objekt ve formátu JSON.

Výstupním formátem je CSV, do kterého je možné seberealizovat jakoukoli tabulku zobrazenou v GUI. Více informací o této funkci lze nalézt v sekci 3.3.4).

### 3.3.4 Rozšíření GUI

Jak bylo řečeno v sekci 3.2, jednou z nevýhod volby Unity je slabší GUI systém. Z toho důvodu byla implementována následující rozšíření, jejich popis je zahrnut pro případné pokračování projektu.

[1] MSDN dokumentaci IEnumerable lze nalézt na `https://msdn.microsoft.com/en-us/library/system.collections.ienumerable(v=vs.110).aspx`

[2] dokumentace a popis API `https://docs.unity3d.com/Manual/Coroutines.html`

**Tabulka** – určena pro přehledné zobrazení výstupů. Jako vstup přijímá styly (šablony) záhlaví, řádek a jejich buněk, spolu se seznamy zobrazovaných dat. Styl záhlaví zahrnutý v projektu dále rozšiřuje funkčnost o uložení do souboru ve formátu CSV.

**ToolTipService** – služba umožňující zobrazit text na pozici myši. Reference na službu je spravována implementací vzoru *Service Locator*.

**Vizuální vodítka,** jejichž přítomnost je důležitá pro orientaci ve scéně. V projektu jsou zahrnuty dva materiály plnící tuto funkci – pozadí s barevným gradientem[1] pro orientaci vůči horizontu a mřížka[2] zobrazená v jednotkách modelu pro odhad vzdáleností. Dále je též zahrnut kvádr obepínající extrémy pro ukotvení modelu v prostoru.

### 3.3.5 Dokumentace, správnost a kvalita kódu

Všechny veřejné metody, proměnné a vlastnosti jsou komentovány standardními dokumentačními XML notacemi. Těchto notací lze automaticky generovat programovou dokumentaci.

Na projektem byla spuštěna analýza metrik kódu *maintainablity index*. Tato analýza se skládá z cyklomatické složitosti, hloubky dědičnosti, provázanosti a řádek kódu. Výsledná hodnota, více než 80 bodů, výrazně přesahuje doporučené minimální skóre 20 bodů.

Protože jádrem projektu je implementovaný algoritmus, testování správnosti je soustředěno právě na tuto část. Správnost algoritmu je předmětem experimentu 4.2. V aplikaci je mimo zmíněný experiment také interaktivní test kolizí se zobrazenou sondou, ověřující kolize sondy fyzikálním engine Unity.

### 3.3.6 Možnosti rozšíření

Jedním z možných rozšíření práce by byla implementace klienta pro dávkové zpracování, jenž by umožnil automatizovat experimenty. Toto rozšíření by umožnilo podstatným způsobem zvýšit objem experimentů. Složitost toho rozšíření lze odhadnou jako středně obtížnou (Unity dovoluje spuštění v konzolovém módu), nicméně aktuální prioritou je zdokonalování algoritmu.

[1] založené na `https://github.com/keijiro/UnitySkyboxShaders/blob/master/Assets/Skybox%20Shaders/Horizontal%20Skybox.shader`

[2] založené na `https://forum.unity3d.com/threads/wireframe-grid-shader.60071/`

Dalším vylepšením je možnost překrytí ukončovací podmínky. Toto rozšíření není složité a lze ho implementovat zavedením jediného rozhraní. Nicméně do této chvíle nebyl zaznamenán požadavek na tuto funkčnost.

Návrh programových rozhraní a jejich požadavků by bylo s vysokou pravděpodobností možné zjednodušit - návrh tohoto API je poměrně „mladý“ a vyšší počet implementací spolu s dalším testováním by pravděpodobně ukázal možné ladění rozhraní.

Posledním uvedeným rozšířením bylo mohlo být zavedení barev, případně materiálů, do formátu modelu proteinů s cílem dosáhnout vizuální reprezentaci podobnější obrázku 2.1.

# 4 Experimenty

Tato kapitola popisuje experimenty provedené na různých verzích algoritmu. Vzhledem k potenciálně vysokému počtu možných kombinací různých variant algoritmu, testovány byly pouze vybrané verze. Vybrané verze a jejich implementace jsou blíže popsané v sekci 3.1, byly voleny tak, aby vždy v nějakém ohledu posouvaly algoritmus kupředu.

První z experimentů je věnován ověření správnosti (bezkoliznosti) nalezené cesty. Přestože tak není explicitně uvedeno, i v následujících experimentech byla správnost nalezených cest ověřována stejnou metodou. Následující experiment se snaží odhalit míru vlivu náhody (počátečního semínka). Další experiment se věnuje algoritmu zahrnujícímu orientaci sondy. Čtvrtý a poslední experiment rozšiřuje zapojení orientace jejím zahrnutím do metriky používané v algoritmu.

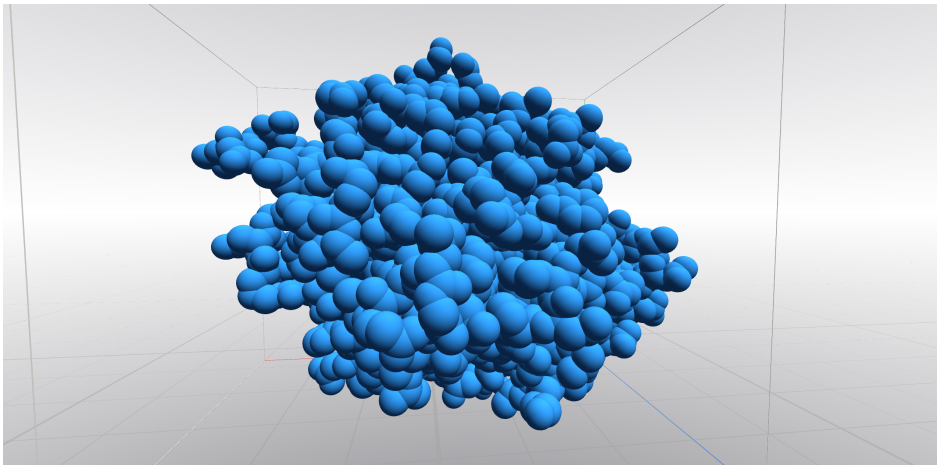
## 4.1 Podmínky experimentů

Není-li uvedeno jinak, všechny dále uvedené experimenty byly spuštěny v prostředí popsaném v této kapitole. Překlad proběhl v Unity verze 5.6.0f3 s rozšířením o podporu C# 6.0, program byl spuštěn pod operačním systémem Windows 10 s procesorem Intel i5. Během výpočtu experimentů neprobíhala žádná jiná uživatelská interakce s PC. Detaily prostředí experimentu jsou uvedeny v tabulce 4.1.

Tabulka 4.1: Prostředí spuštění experimentů

| Prostředí | Unity 5.6.0f3, 64-bit |
|---|---|
| OS | Windows 10 Education, 64-bit |
| CPU | Intel i5-4670K @3.40GHz |
| RAM | 16 GB @1600MHz |
| Pevný disk | Samsung SSD 850 EVO 500GB |

Experimenty se týkají problému, jak byl definován v sekci 2.3. Počáteční konfigurace byla vstupem, stejně jako model proteinu (obrázek 4.1). Navigovaným agentem byla sonda skládající se ze dvou sfér jednotkového poloměru. Cílem je jakákoli konfigurace ležící vně modelu. Práh detekce konfigurace vně proteinu byl nastaven na 5 jednotek modelu.



Obrázek 4.1: Model proteinu s 2358 atomy, na němž probíhaly experimenty

## 4.2 Experiment 1 - ověření správnosti implementace

Cílem tohoto experimentu bylo ověřit správnost implementace. Vzhledem k absenci referenčních výsledků je správnost ověřována testem bezkoliznosti nalezené cesty.

Jak je uvedeno v sekci 3.1.3, algoritmus využívá diskrétního výpočtu kolizí, z čehož vyplývá konečná přesnost vyhodnocení bezkoliznosti cesty mezi dvěma stavy. Z tohoto důvodu je připouštěn průnik (jenž není považován za chybný) při lineární interpolaci mezi stavy, s krokem parametru $t = 0,5$.

### 4.2.1 Metodika

Výpočet byl zastaven několik sekund po nalezení cesty se zhruba dvaceti tisíci testovanými vzorky. Propojitelnost konfigurací (viz sekce 3.1.3 a 2.5.3) byla testována kromě obou konfigurací i pro konfiguraci v jedné polovině vzdálenosti, tzn. lineární interpolace obou koncových konfigurací pro parametr $t = 0,5$. Bezkoliznost nalezené cesty byla ověřena metodou hrubé síly, testující každou překážku s každou částí sondy krokem parametru lineární interpolace mezi stavy nastaveným na hodnotu 0,1. Pro každou konfiguraci byl též zaznamenán maximální průnik s překážkou dle rovnice 4.1:

$$p(s, o) = (s.Radius + o.Radius) - |s.Center - o.Center| \qquad (4.1)$$

kde $s$ je konfigurace sondy a $o$ je překážka. Kladná hodnota maximálního průniku značí, že konfigurace je kolizní. Záporná hodnota pak značí hodnotu „rezervy“ v daném bodě.

### 4.2.2 Výsledky

Z předpokladů a nastavení metody ověření vyplývá, že pro hodnotu parametru $t = 0.5$ musí být hodnota maximálního průniku menší nebo rovna nule. Výsledky uvedené v tabulce 4.2 potvrzují splnění tohoto požadavku. Vzhledem k diskrétní detekci kolizí může docházet ke kolizím mezi konfiguracemi při spojitém pohybu sondy. V tabulce 4.2 takové kolizi odpovídá (pro daný krok metody hrubé síly) hodnota maximálního průniku přes všechny testované konfigurace. Vzhledem k nastavené konečné přesnosti detekce tento typ kolizí nelze považovat za chybný.

Řádku maximálního průniku v tabulce 4.2 pak lze vyložit jako negaci rezervy nalezené cesty. Rezervou se rozumí vzdálenost od nejbližší překážky (použitím detekce kolizí s daným krokem metody hrubé síly). Řádky minima a průměru pak jako maximální a průměrnou rezervu cesty.

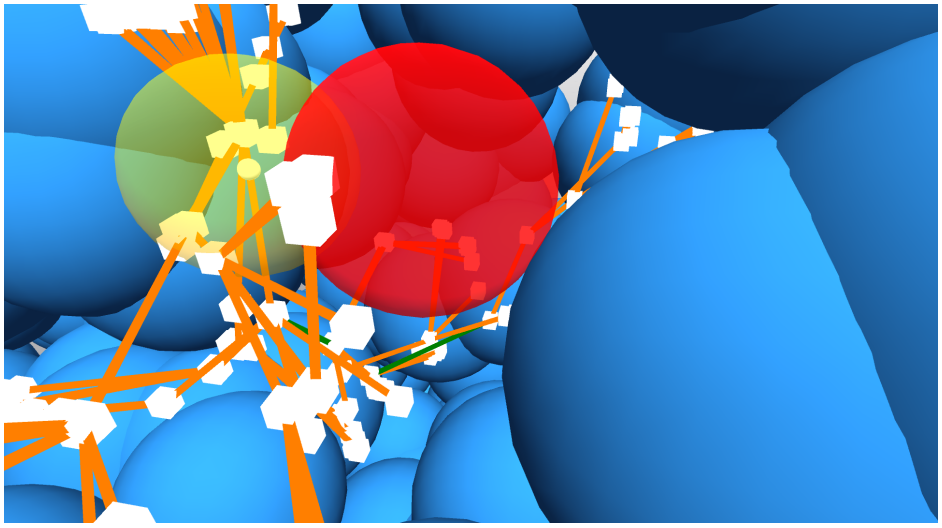
Buňka maximálního průniku přes všechny testované konfigurace udává hodnotu řádově menší než byl poloměr částí navigované sondy. Tento průnik je zachycen na obrázku 4.2.

Tabulka 4.2: Výsledky experimentu 1, maximální průnik konfigurací

| | průnik ve vrcholech cesty [%] | průnik s parametrem $t = 0.5$ [%] | všechny testované konfigurace [%] |
|---|---|---|---|
| minimum | -223,4 | -158,2 | -223,4 |
| průměr | -45,2 | -36,5 | -38,9 |
| maximum | -0,5 | -3,0 | 4,1 |
| odchylka | 543 | 40,1 | 41,6 |

### 4.2.3 Vyhodnocení

Výsledky uvedené v tabulce 4.2 potvrzují správnost nalezené cesty, jak byla definovaná. Experiment lze vyhodnotit úspěšný, ale v závislosti na požadavcích reálného nasazení lze uvažovat o zvýšení rozlišení diskrétní detekce kolizí. Vzhledem k době výpočtu tohoto experimentu (několik vteřin) se zdá podmínka ukončení výpočtu dalších experimentů na dvě minuty výpočetního času jako dostačující.



Obrázek 4.2: Maximální průnik sondy a překážek na nalezené cestě. Modré sféry jsou atomy modelu, bílé krychle vrcholy stromu. Sféra sondy v bezkolizní pozici je značena žlutě, v kolizní pak červeně. Cesta je značena zeleně.

## 4.3 Experiment 2 - vliv náhody

Tento experiment měl za úkol odhadnout závislost výkonu algoritmu na volbě počátečního semínka (*seed*) generátoru pseudonáhodných čísel.

Velikost uzávěrů byla určena jako dvojnásobek vzdálenosti nejbližšího středu překážky od pozice cílové konfigurace. Maximální vzdálenost byla rovna průměru navigované sondy a maximální počet pokusů o připojení vygenerovaného náhodného vzorku $v_{rand}$ byl nastaven na 5. Orientace sondy byla generována náhodně bez interpolace. Při vygenerování vzdáleného vzorku byl tento vzorek nahrazen jiným na maximální vzdálenosti ve směru původního vzorku. V případě, že nová konfigurace byla kolizní, byl vzorek přesunut na polovinu vzdálenosti od nejbližšího vrcholu v příslušném směru. Tento postup půlení vzdálenosti byl opakován až pětkrát za sebou. Podrobnější popis tohoto algoritmu lze nalézt v sekci 3.1.2.

### 4.3.1 Metodika

Doba výpočtu algoritmu v tomto experimentu byla omezena na 2 minuty reálného času, počet unikátních cest byl určen vizuální kontrolou, pro každé opakování bylo voleno jiné počáteční semínko.

### 4.3.2 Výsledky

V tabulce 4.3 jsou uvedena „hrubá“ data výsledků experimentu, první sloupec tabulky udává celkový počet náhodných vzorků, druhý pak celkový počet vrcholů stromu, třetí celkový počet změn rodiče (tzn. kolikrát se podařilo snížit cenu vrcholu), čtvrtý udává celkový počet nalezených cest (v závorce je uveden počet unikátních cest ve smyslu využití tunelů v modelu), pátý počet vrcholů nejkratší nalezené cesty, šestý pak počet vzorků potřebných k nalezení první z cest, sedmý délku nejkratší nalezené cesty a konečně poslední sloupec udává světlost (poloměr nejmenší koule volného prostoru ve vrcholech cesty) nejkratší nalezené cesty.

Tabulka 4.3: Výsledky experimentu 2: „hrubá“ data

| počet vzorků | počet vrcholů | počet změn rodiče | počet cest | počet vrcholů cesty | počet vzorků k nalezení | délka cesty | světlost cesty |
|---|---|---|---|---|---|---|---|
| 88302 | 298 | 41 | 1 (1) | 15 (15) | 1615 | 20.95 | 2.74 |
| 86818 | 371 | 34 | 3 (1) | 23 (9) | 52839 | 21.50 | 2.61 |
| 90326 | 307 | 55 | 1 (1) | 13 (13) | 62705 | 19.12 | 2.73 |
| 83994 | 298 | 64 | 1 (1) | 14 (14) | 1068 | 19.24 | 2.70 |
| 83249 | 250 | 68 | 1 (1) | 19 (19) | 1923 | 21.384 | 2.58 |
| 75191 | 303 | 43 | 2 (1) | 17 (6) | 14176 | 19.03 | 2.73 |
| 79437 | 371 | 43 | 3 (1) | 18 (10) | 1534 | 19.89 | 2.70 |
| 83261 | 310 | 66 | 2 (1) | 15 (6) | 1450 | 20.22 | 2.71 |

V tabulkách 4.4 a 4.5 jsou uvedeny vypočtené ukazatele z „hrubých“ dat. V tabulce 4.4 první sloupec udává průměrný počet vzorků potřebných na přidání vrcholu stromu, druhý sloupec udává průměrný počet změn rodiče na vrchol stromu, třetí sloupec uvádí průměrnou vzdálenost mezi vrcholy nejkratší nalezené cesty. Poslední sloupec udává poměr unikátních (nesdílených s jinou cestou) vrcholů nejkratší cesty ku celkovému počtu vrcholů této cesty. V tabulce 4.5 jsou uvedeny statistické ukazatele hodnot z tabulky 4.3, první a druhý sloupec je průměr, resp. směrodatná odchylka naměřených hodnot.

Tabulka 4.4: Výsledky experimentu 2: vypočtené ukazazele

| vzorků na vrchol | změn rodiče na vrchol | prům. vzdálenost vrcholů na cestě | unikátních vrcholů cesty [%] |
|---|---|---|---|
| 296.32 | 0.14 | 1.61 | 100 |
| 234.01 | 0.09 | 1.25 | 39 |
| 294.22 | 0.18 | 1.56 | 100 |
| 281.86 | 0.21 | 1.31 | 100 |
| 333.00 | 0.27 | 1.31 | 100 |
| 248.16 | 0.14 | 1.38 | 35 |
| 214.12 | 0.12 | 1.38 | 56 |
| 268.58 | 0.21 | 1.44 | 40 |

Tabulka 4.5: Výsledky experimentu 2: statisitkcé ukazatele

| | medián | průměr | směrodatná odchylka |
|---|---|---|---|
| počet vzorků | 83627.5 | 83822.25 | 4865.97 |
| počet vzorků k nalezení | 1769 | 17163.75 | 25578.96 |
| počet vrcholů stromu | 305 | 313.50 | 40.16 |
| počet změn rodiče | 49 | 51.75 | 13.16 |
| počet cest | 1.50 | 1.75 | 0.89 |
| počet unikátních cest | 1.00 | 1.00 | 0.00 |
| délka nejkratší | 23.73 | 23.26 | 3.18 |
| počet vrcholů (nejkratší) | 16.00 | 16.75 | 3.24 |
| počet unikátních vrcholů | 11.50 | 11.50 | 4.57 |
| poměr unikátních vrcholů [%] | 77.78 | 71.25 | 31.29 |
| světlost cesty | 2.71 | 2.69 | 0.06 |

### 4.3.3 Vyhodnocení

V žádném z provedených experimentů se nezdařilo najít více než jednu unikátní cestu v přiděleném čase (u některých semínek však byla nalezena později). Vzhledem k vysokému počtu náhodných vzorků potřebných na rozšíření stromu se lze domnívat, že je potřeba vylepšit úspěšnost algoritmu vzorkování.

Rozdíly a směrodatné odchylky sledovaných ukazatelů se ukázaly jako podstatné. Z toho lze vyvozovat jistou závislost na volbě počátečního semínka - v další experimentech by bylo vhodné provést stejný nebo vyšší po-

čet opakování pro zajištění kvality naměřených dat. Z experimentu vyplunulo následující:

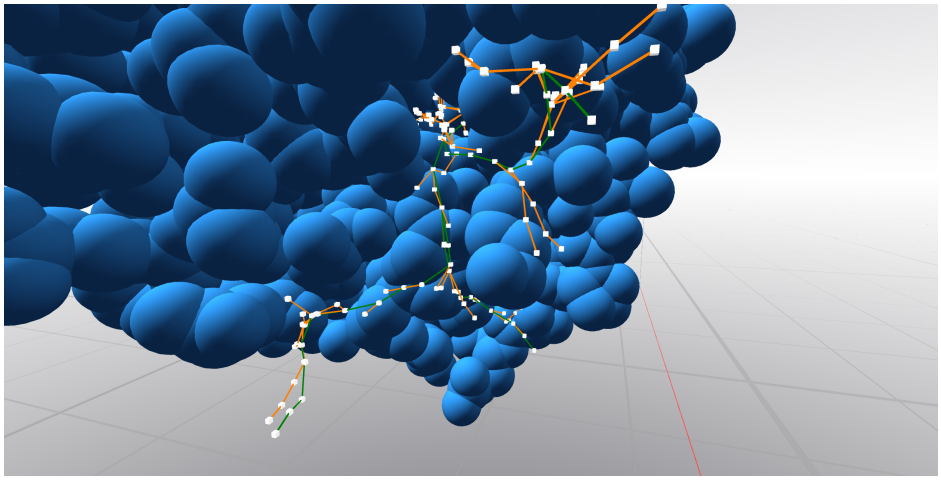
1. *Náhodné vygenerování vhodného vzorku je omezením výkonnosti algoritmu.* Vzhledem k řádově jinému počtu vzorků potřebných k nalezení první cesty širším z hrdel a celkovému počtu se lze domnívat, že důvodem nenalezení cesty užším z hrdel je nízká pravděpodobnost vygenerování vhodných vzorků, což experimentálně potvrzuje tuto vlastnost algoritmu uvedenou v sekci 2.5.

2. *Velký počet neúspěšných vzorků nemá vliv na délku nebo světlost nalezené cesty.* Vzhledem k pozorování uvedenému výše by bylo vhodné uvést, že z experimentů se nezdá, že by problém nalezení vhodného vzorku měl vliv na světlost nebo délku cesty.

3. *Snížení maximálního počtu půlení vzdálenosti vede k eliminaci příliš podobných vrcholů.* Oproti experimentu 1 byl maximální počet půlení vzdálenosti ve *SteerTo* nastaven na 5, místo 10. Tím se, alespoň po vizuální stránce, zdá problém s množstvím vzájemně podobných vrcholů vyřešen.

4. *Na nalezené cestě je příliš zbytečných změn orientace.* Vizuální kontrola odhalila zmiňovaný nedostatek – nalezená cesta působí nepřirozeně z důvodu příliš častých změn orientace sondy. V další verzi algoritmu by proto bylo vhodné proto tento problém řešit např. zavedením penalizace rotace.

5. *Nalezené cesty se liší až v posledních segmentech.* Ani v jednom z experimentů se nepodařilo nalézt více než jedinou unikátní cestu a nalezené cesty se liší až v posledních segmentech, jak je patrné z obrázku 4.3. To značí nedokonalost uzavírání nalezených koridorů (viz sekce 3.1.4). Nicméně lze konstatovat, že vzájemně se podobající cesty nepůsobí problémy v algoritmu. V tomto ohledu se otevírá prostor pro vylepšení uzavírání koridorů, případně rozšíření o dodatečné filtrování příliš podobných se cest.



Obrázek 4.3: Tři cesty(zeleně) se společným počátečním úsekem

## 4.4 Experiment 3 - interpolování orientace

Tento experiment měl za účel vyhodnotit rozšíření o interpolaci orientace při vytváření nových konfigurací. Ostatní nastavení bylo shodné s nastavením experimentu 2.

### 4.4.1 Metodika

Stejně jako v experimentu 2, doba výpočtu algoritmu byla omezena na 2 minuty reálného času, počet unikátních cest byl určen vizuální kontrolou a pro každé opakování bylo voleno stejné počáteční semínko jako v experimentu 2.

### 4.4.2 Výsledky

V tabulce 4.6 jsou uvedena „hrubá“ data, obdobně jako v experimentu 2 první sloupec tabulky udává celkový počet náhodných vzorků, druhý pak celkový počet vrcholů stromu, třetí celkový počet změn rodiče (tzn. kolikrát se podařilo snížit cenu vrcholu), čtvrtý udává celkový počet cest nalezených cest (v závorce je uveden počet unikátních cest ve smyslu využití tunelů v modelu), pátý počet vrcholů nejkratší nalezené cesty, šestý pak počet vzorků potřebných k nalezení první z cest, sedmý délku nejkratší nalezené cesty a konečně poslední sloupec udává světlost (poloměr nejmenší koule volného prostoru ve vrcholech cesty) nejkratší nalezené cesty.

Tabulka 4.6: Výsledky experimentu 3: „hrubá“ data

| počet vzorků | počet vrcholů | počet změn rodiče | počet cest | počet vrcholů cesty | počet vzorků k nalezení | délka cesty | světlost cesty |
|---|---|---|---|---|---|---|---|
| 86703 | 555 | 168 | 2 (1) | 18 (7) | 80754 | 26.3 | 2.80 |
| 76690 | 395 | 69 | 2 (1) | 15 (5) | 2929 | 21.98 | 2.73 |
| 81350 | 455 | 146 | 1 (1) | 18 (18) | 850 | 22.99 | 2.70 |
| 72967 | 497 | 187 | 2 (1) | 16 (7) | 1606 | 21.89 | 2.74 |
| 84511 | 422 | 195 | 1 (1) | 15 (15) | 1514 | 20.75 | 2.71 |
| 60937 | 655 | 141 | 7 (1) | 17 (5) | 1497 | 25.05 | 2.82 |
| 82410 | 476 | 146 | 2 (1) | 15 (7) | 12238 | 20.16 | 2.77 |
| 69988 | 473 | 97 | 3 (1) | 14 (7) | 846 | 21.47 | 2.82 |

V tabulkách 4.7 a 4.8 jsou uvedeny vypočtené ukazatele ze surových dat. V tabulce 4.7 první sloupec udává průměrný počet vzorků potřebných na přidání vrcholu stromu, druhý sloupec udává průměrný počet změn rodiče na vrchol stromu, třetí sloupec uvádí průměrnou vzdálenost mezi vrcholy nejkratší nalezené cesty. Poslední sloupec udává poměr unikátních (nesdílených s jinou cestou) vrcholů nejkratší cesty ku celkovému počtu vrcholů této cesty. V tabulce 4.8 jsou uvedeny statistické ukazatele hodnot z tabulky 4.6, první a druhý sloupec je průměr, resp. směrodatná odchylka naměřených hodnot.

Tabulka 4.7: Výsledky experimentu 3: vypočtené ukazazele

| počet vzorků na vrchol | počet změn rodiče na vrchol | průměrná vzdálenost vrcholů na cestě | počet unikátních vrcholů cesty [%] |
|---|---|---|---|
| 156.22 | 0.30 | 1.46 | 39 |
| 194.15 | 0.17 | 1.47 | 33 |
| 178.79 | 0.32 | 1.28 | 100 |
| 146.81 | 0.38 | 1.37 | 44 |
| 200.26 | 0.46 | 1.38 | 100 |
| 93.03 | 0.22 | 1.48 | 29 |
| 173.13 | 0.31 | 1.34 | 47 |
| 147.97 | 0.21 | 1.53 | 50 |

Tabulka 4.8: Výsledky experimentu 3: sumarizované ukazatele

| | medián | průměr | směrodatná odchylka |
|---|---|---|---|
| počet vzorků | 79020 | 76945 | 8638 |
| počet vzorků k nalezení | 1560 | 12779 | 27727 |
| počet vrcholů stromu | 474.5 | 491.00 | 81.81 |
| počet změn rodiče | 146.00 | 143.63 | 42.89 |
| počet cest | 2.00 | 2.50 | 1.93 |
| počet unikátních cest | 1.00 | 1.00 | 0.00 |
| délka nejkratší cesty | 21.94 | 22.58 | 2.13 |
| počet vrcholů (nejkratší) | 15.50 | 16.00 | 1.51 |
| počet unikátních vrcholů | 7.00 | 8.88 | 4.85 |
| poměr unikátních vrcholů [%] | 45.21 | 55.26 | 28.42 |
| světlost cesty | 2.76 | 2.76 | 0.05 |

V tabulce 4.9 je uveden rozdíl sumarizovaných statistik experimentu 3 a 2. V prvé řadě si nelze nepovšimnout zpravidla vyšších směrodatných odchylek experimentu 3 u většiny sledovaných statistik. Dále pak je důležité vyzdvihnout, že ani počet unikátních cest ani světlost se markantně nezměnila.

Verze algoritmu s interpolací orientací průměrně vygenerovala za přidělený čas méně vzorků, což lze vysvětlit vyšší výpočetní náročností. Jak nižší počet vzorků potřebných k nalezení cesty, tak vyšší počet vrcholů stromu i změn rodiče mohou značit zvýšenou schopnost algoritmu vytvořit přijatelnou konfiguraci. Nižší počet vrcholů a kratší nalezené cesty tuto domněnku pouze podporují.

Nižší podíl počet unikátních vrcholů na cestě lze opět vysvětlit zvýšenou schopností algoritmu vytvořit přijatelnou konfiguraci, ovšem u této statistiky též hraje roli větší podobnost orientací bližších konfigurací – při hledání cesty měl algoritmus tendenci pokračovat po povrchu proteinu. Tato vlastnost byla též pozorována u šestého vzorku a vysvětluje enormní počet cest, strom měl šanci opustit případný uzávěr ještě před jeho vytvořením při nalezení cesty.

Tabulka 4.9: Porovnání experimentu 2 a 3

| | medián | průměr | směrodatná odchylka |
|---|---|---|---|
| počet vzorků | -4607.50 | -6877.75 | 3771.81 |
| počet vzorků k nalezení cesty | -209.00 | -4384.50 | 2148.36 |
| počet vrcholů stromu | 169.50 | 177.50 | 41.65 |
| počet změn rodiče | 97.00 | 91.88 | 29.73 |
| počet cest | 0.50 | 0.75 | 1.04 |
| počet unikátních cest | 0.00 | 0.00 | 0.00 |
| délka nejkratší cesty | -1.79 | -0.68 | -1.05 |
| počet vrcholů nejkratší cesty | -0.50 | -0.75 | -1.73 |
| počet unikátních vrcholů cesty | -4.50 | -2.63 | 0.29 |
| poměr unikátních vrcholů cesty [%] | -32.57 | -15.99 | -2.87 |
| světlost cesty | 0.05 | 0.07 | -0.01 |

### 4.4.3 Vyhodnocení

Z výsledků experimentu lze usuzovat, že změna v algoritmu zvýšila jeho schopnost nalézt cestu. Tato změna ovšem měla vliv na vznik nové tendence algoritmu postupovat po povrchu proteinu což snížilo efektivitu uzávěrů.

## 4.5 Experiment 4 - metrika

Cílem tohoto experimentu bylo ověřit možnost zapojení orientace do metriky (vztah 4.2) používané v algoritmu

$$\rho(a,b) = |a.Center - b.Center| + \frac{AngleBetween(a.Dir, b.Dir)}{180.0} \qquad (4.2)$$

kde procedura *AngleBetween* vypočítává úhel ve stupních z intervalu $\langle 0; 180)$ mezi orientacemi vzorků $a$ a $b$. Ostatní nastavení, kromě použité metriky, byla shodná s experimenty 2 a 3.

### 4.5.1 Metodika

Stejně jako v experimentu 2 a 3 byla doba výpočtu algoritmu omezena na 2 minuty reálného času. Též počet unikátních cest byl určen vizuální kontrolou a pro každé opakování bylo voleno stejné počáteční semínko jako v experimentu 2.

### 4.5.2 Výsledky

V tabulce 4.10 jsou uvedena „hrubá“ data, sloupce odpovídají sloupcům tabulek v experimentech 2 a 3 (viz experiment 2 v sekci 4.3). Jediným rozdílem je poslední sloupec jenž uvádí celkovou cenu cesty dle metriky, který nahradil sloupec světlosti, jenž se ukázal jako málo přínosný.

Tabulka 4.10: Výsledky experimentu 4: „hrubá“ data

| počet vzorků | počet vrcholů | počet změn rodiče | počet cest | počet vrcholů cesty | počet vzorků k nalezení | délka cesty | cena cesty |
|---|---|---|---|---|---|---|---|
| 75520 | 395 | 103 | 1 (1) | 15 (15) | 1522 | 19.79 | 20.5 |
| 65833 | 490 | 121 | 4 (1) | 18 (9) | 2552 | 24.14 | 25.1 |
| 77550 | 442 | 74 | 1 (1) | 16 (16) | 1274 | 21.79 | 22.96 |
| 68712 | 413 | 55 | 1 (1) | 15 (15) | 1006 | 22.23 | 23.05 |
| 64679 | 617 | 71 | 8 (1) | 17 (3) | 32417 | 24.33 | 25.26 |
| 72271 | 644 | 111 | 3 (1) | 15 (3) | 46817 | 21.29 | 22.28 |
| 73823 | 412 | 98 | 1 (1) | 19 (19) | 6149 | 24.77 | 25.62 |
| 78622 | 361 | 84 | 2 (1) | 16 (6) | 34284 | 19.30 | 20.00 |

Obdobně jako v experimentech 2 a 3 jsou v tabulkách 4.11 a 4.12 uvedené vypočtené ukazatele z „hrubých“ dat. V tabulce 4.11 první sloupec udává průměrný počet vzorků potřebných na přidání vrcholu stromu, druhý sloupec udává průměrný počet změn rodiče na vrchol stromu, třetí sloupec uvádí průměrnou vzdálenost mezi vrcholy nejkratší nalezené cesty. V tabulce 4.12 jsou uvedeny statistické ukazatele hodnot z tabulky 4.10, první a druhý sloupec je průměr, resp. směrodatná odchylka naměřených hodnot.

Tabulka 4.11: Výsledeky experimentu 4: vypočtené ukazazele

| počet vzorků na vrchol | počet změn rodiče na vrchol | průměrná vzdálenost vrcholů na cestě | počet unikátních vrcholů cesty [%] |
|---|---|---|---|
| 191.19 | 0.26 | 1.32 | 100 |
| 134.35 | 0.25 | 1.34 | 50 |
| 175.45 | 0.17 | 1.36 | 100 |
| 166.37 | 0.13 | 1.48 | 100 |
| 104.83 | 0.12 | 1.43 | 18 |
| 112.22 | 0.17 | 1.42 | 20 |
| 179.18 | 0.24 | 1.30 | 100 |
| 217.79 | 0.23 | 1.21 | 38 |

Tabulka 4.12: Výsledky experimentu 4: statistické ukazatele

| | medián | průměr | směrodatná odchylka |
|---|---|---|---|
| Počet vzorků | 73047 | 72126 | 5247 |
| Počet vzorků k nalezení | 4350.5 | 15752 | 18830 |
| Počet vrcholů stromu | 427.5 | 471.75 | 104.99 |
| Počet změn rodiče | 91 | 89.63 | 22.40 |
| Počet cest | 1.5 | 2.63 | 2.45 |
| Počet unikátních cest | 1.00 | 1.00 | 0.00 |
| Délka nejkratší cesty | 22.01 | 22.21 | 2.07 |
| Počet vrcholů (nejkratší) | 16 | 16.38 | 1.51 |
| Počet unikátních vrcholů | 12 | 10.75 | 6.30 |
| Poměr unikátních vrcholů [%] | 75 | 65.64 | 38.07 |

Oproti předchozímu experimentu došlo ke zvýšení u směrodatných odchylek počtu vrcholů, unikátních vrcholů a cest, to nicméně neplatí pro směrodatné odchylky ostatních sledovaných veličin. Snížení celkového počtu vzorků spolu s počtem vrcholů stromu si lze vyložit jako zpomalení algoritmu oproti verzi testované v experimentu 3. V datech lze také pozorovat zvýšení počtu unikátních vrcholů cest znamenající větší úspěšnost uzávěrů pro tento algoritmus. Celkově jsou výsledky velmi podobné předchozímu experimentu.

V tabulce 4.13 je uveden rozdíl sumarizovaných statistik tohoto experimentu a experimentu 2. V tomto porovnání výsledků si nelze nepovšimnout podobných trendů jako ve srovnání experimentu 2 a 3 (tabulka 4.9).

Tabulka 4.13: Porovnání experimentu 2 a 4

| | medián | průměr | směrodatná odchylka |
|---|---|---|---|
| počet vzorků | -10580.50 | -11696.00 | 381.52 |
| počet vzorků k nalezení cesty | 2581.50 | -1411.13 | -6748.14 |
| počet vrcholů stromu | 122.50 | 158.25 | 64.83 |
| počet změn rodiče | 42.00 | 37.88 | 9.24 |
| počet cest | 0.00 | 0.88 | 1.56 |
| počet unikátních cest | 0.00 | 0.00 | 0.00 |
| délka nejkratší cesty | -1.72 | -1.06 | -1.11 |
| počet vrcholů nejkratší cesty | 0.00 | -0.38 | -1.73 |
| počet unikátních vrcholů cesty | 0.50 | -0.75 | 1.73 |
| poměr unikátních vrcholů cesty [%] | -2.78 | -5.60 | 6.78 |

### 4.5.3 Vyhodnocení

Tento experiment poskytl podobné výsledky, a tudíž nepřinesl žádné rozhodující informace. Vizuální kontrola nalezených cest ovšem zpravidla ukazuje přirozeněji vypadající pohyb sondy po cestě oproti verzi algoritmu v experimentu 3.

## 4.6 Shrnutí

Celkově se algoritmus osvědčil při řešení problému plánování pohybu v modelech proteinů a nikdy neselhal nalézt cestu v přiděleném čase. Odhaleny však byly problémy s uzavíráním použitých tunelů a potvrzeny obtíže s průchodem úzkými hrdly.

# 5 Závěr

Cílem této práce bylo navrhnout a implementovat algoritmus vhodný pro plánování pohybu v proteinových modelech. Na základě heuristik plánování pohybu dostupných v literatuře byl navržen takový algoritmus a následnými experimenty otestován.

Výsledná aplikace nejenom že implementuje navržený algoritmus, ale je i silným nástrojem pro vizualizaci výsledků a sledování průběhu algoritmu v reálném čase. Hlavním cílem architektury aplikace byla maximální modularita algoritmu, jehož postup lze změnit s minimálními zásahy do kódu – pouhou implementací rozhraní.

Vzhledem k síle vyvinutého nástroje je možná na škodu, že nebylo implementováno ještě více variant algoritmu. Přestože v experimentech proběhly desítky měření se stovkami údajů, vzhledem k roli pravděpodobnosti v algoritmu by mohl být tento objem ještě větší.

Velkým osobním přínosem byla nabytá zkušenost s technologií, stejně jako přehled získaný v problematice pravděpodobnostních algoritmů plánování cesty.

# Reference


[1] Franz Aurenhammer and Rolf Klein. Voronoi diagrams. *Handbook of Computational Geometry*, 5:201–290, 2000.

[2] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[3] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, 96:59–69, 2014.

[4] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 2997–3004. IEEE, 2014.

[5] Roland Geraerts and Mark H Overmars. A comparative study of probabilistic roadmap planners. In *Algorithmic Foundations of Robotics V*, pages 43–57. Springer, 2004.

[6] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[7] Léonard Jaillet, Juan Cortés, and Thierry Siméon. Sampling-based path planning on configuration-space costmaps. *IEEE Transactions on Robotics*, 26(4):635–646, 2010.

[8] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[9] Lydia E Kavraki, Mihail N Kolountzakis, and J-C Latombe. Analysis of probabilistic roadmaps for path planning. *IEEE Transactions on Robotics and Automation*, 14(1):166–171, 1998.

[10] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[11] Sven Koenig and Maxim Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, pages 968–975. IEEE, 2002.

[12] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning a*. *Artificial Intelligence*, 155(1-2):93–146, 2004.

[13] Barbora Kozlíková, Eva Šebestová, Vilém Šustr, Jan Brezovský, Ondřej Strnad, Lukáš Daniel, David Bednář, Antonín Pavelka, Martin Maňák, Martin Bezděka, Petr Beneš, Matúš Kotry, Artur Wiktor Gora, Jiří Damborský, and Jiří Sochor. CAVER Analyst 1.0: Graphic tool for interactive visualization and analysis of tunnels and channels in protein structures. *Bioinformatics*, 30.

[14] Ivan Kramosil and Jiří Michálek. Fuzzy metrics and statistical metric spaces. *Kybernetika*, 11(5):336–344, 1975.

[15] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[16] Steven M LaValle and James J Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[17] Martin Manak. *Application of Computational Geometry to Modeling and Visualization of Proteins*. PhD thesis, University of West Bohemia, 2016.

[18] Madhav K Ponamgi, Dinesh Manocha, and Ming C Lin. Incremental algorithms for collision detection between polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):51–64, 1997.

[19] Dennis Wiebusch and Marc Erich Latoschik. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2015 IEEE 8th Workshop on*, pages 25–32. IEEE, 2015.

# A Struktura jmenných prostorů

Projekt je členěn do jmenných prostorů (obdoba balíků v Javě), odpovídajícím složkám v souborovém systému. Klíčová část implementace se nachází ve jmenném prostoru Scripts pod kořenovým jmenným prostorem je Assets. Členění prostoru Scripts je je zachyceno na obrázku A.1.

**Assests.Scripts** kořenový jmenný prostor

- **Algorithm** implementace algoritmu
  - **Connecting** výchozí implementace a rozhraní testu připojení
  - **Evaluation** testy správnosti algoritmu
  - **Model** odpovídá vrstvě MVC, obshauje potřebná rozhraní a třídy vrcholů stromu, atomů, atd.
  - **Sampling** výchozí implementace a rozhraní vzorkování prostoru
  - **Steering** výchozí implementace a rozhraní tvorby bezkolizních konfigurací
- **Controller** volně odpovídá vrstvě MVC
  - **GUI** controllery používané v GUI
  - **Parsing** rozhraní a výchozí implementace zpracovávající vstupní a výstupní soubory
  - **Variables** konstanty používané v controllerech (masky vrstev, ID kláves, atd.)
- **Extensions** rozšíření RNG, IEnumerable a slovníku používaná v projektu

Obrázek A.1: Struktura projektu

# B Rozhraní

Tato příloha probírá rozhraní a zásady jejich implementace pro zajištění správného fungování rozšíření aplikace. Přehled signatur rozhraní lze nalézt v tabulce B.1.

**IProteinParser** používané k načítání modelu překážek. Vrací datový proud sfér, parametrem je cesta k souboru a to buď relativní anebo absolutní. Neexistuje-li soubor, nebo je ve špatném formátu, lze buď vyvolat výjimku (zachycena a logována frameworkem), nebo vrátit prázdný proud.

**ISampler<T>** používané pro kontrolu fáze vzorkování algoritmu. Členy toho rozhraní jsou vlastnost (property) Seed a metoda Sample bez návratové hodnoty s typovaným parametrem vzorku předávaného referencí.

**ISteering<T>** sloužící k vytvoření nekolizní konfigurace z předaného vzorku z předchozí fáze algoritmu. Generický parametr má omezení an bezparametrický konstruktor, dále pak musí implementovat rozhraní IMeasurable<T>, IMixable<T>, IDeepCloneable<T> a dědit od třídy ATreeNode.

Toto rozhraní má jediného člena - metodu Steer jež akceptuje parametry seznamů překážek, seznamu k logování odmítnutých konfigurací a typované parametry přidávané a nejbližší konfigurace. Implementace rozhraní by měla upravit novou konfiguraci na bezkolizní v omezeném počtu pokusů. Implementacím je povoleno selhat a navrátit hodnotu nepravda.

**INodeConnecting<T>** využívané pro ověření propojitelnosti konfigurací. Generický parametr má omezení na potomky třídy ATreeNode implementující rozhraní IMixable<T>. Členem je metoda CanConnect vracející hodnoty pravda/nepravda, jejíž parametry jsou seznamy překážek a typované konfigurace k propojení. Na činnost implementací nejsou kladeny žádné další požadavky a v závislosti na požadavcích lze uvažovat i implementace triviální (vracející hodnotu pravda) i selhávající nalézt správnou odpověď.

**IDeepCloneable<T>** využívané při tvorbě nových konfigurací, obvykle spolu s rozhraním IMixable<T>. Rozhraní neklade žádné požadavky na generický parametr a obsahuje jedinou bezparametrickou metodu DeepClone vracející typovanou hlubokou kopii objektu.

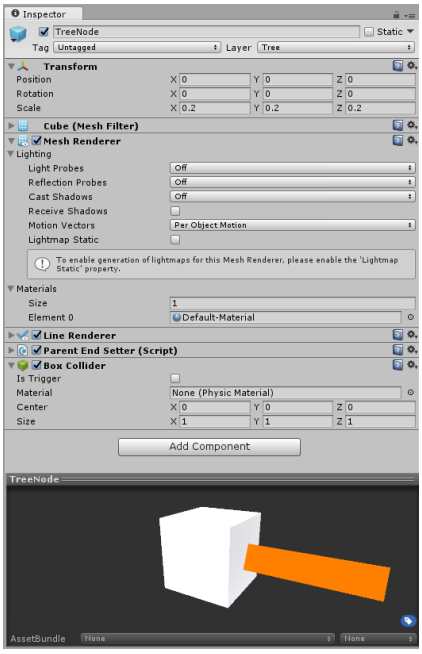
**IMixable<T>** využívané při lineární interpolaci konfigurací. Rozhraní též neklade žádné požadavky na generický parametr a obsahuje jedinou bezparametrickou metodu Mix vracející typovanou lineární interpolaci objektu a typovaného parametru druhého objektu dle parametru v typu s plovoucí desetinnou čárkou.

**IMeasurable<T>** slouží k měření vzdálenosti mezi konfiguracemi. Rozhraní obsahuje vzájemně podobné metody Distance a DistanceSqr vracející skalár vzdálenosti v plovoucí desetinné čárce, typovaným parametrem objekt, ke kterému je vzdálenost měřena. Ač metoda DistanceSqr předpokládá, že implementace bude vracet funkci monotónní vzhledem ke vzdálenosti (např. druhá mocnina), jež je efektivnější vypočítat, za chybné se nepokládá návrat stejné hodnoty jako z metody Distance.

Tabulka B.1: Přehled rozhraní

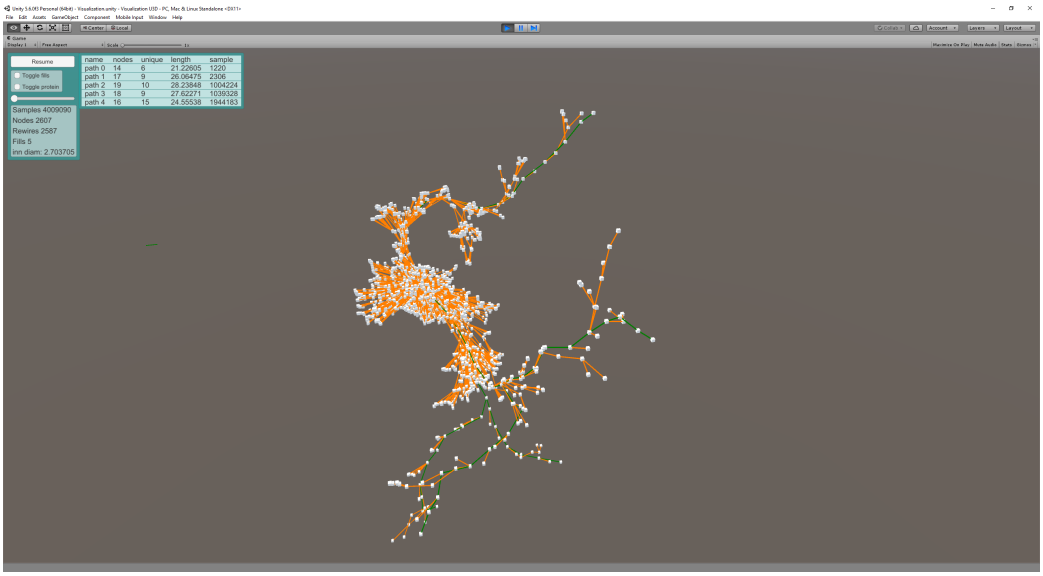
| název | signatury |
|---|---|
| IProteinParser | IEnumerable<Sphere> ParseFile(string path) |
| ISampler<T> | void Sample(ref T node)<br>int Seed { get; set; } |
| ISteering<T> | bool Steer(AlgorithmStrategy<T> strategy, IList<Sphere> obstacles, IList<Sphere> endFills, IList<RejectedNode<T>> rejected, T node, T nearest) |
| INodeConnecting<T> | bool CanConnect(IList<Sphere> obstacles, IList<Sphere> endFills, T a, T b) |
| IDeepCloneable<T> | T DeepClone() |
| IMixable<T> | T Mix(T b, float t) |
| IMeasurable<T> | float DistanceSqr(T other)<br>float Distance(T other) |

# C Ilustrace entity




Obrázek C.1: Ilustrace - entita (a prefab) stromu

# D Ilustrace nalezené cesty




Obrázek D.1: Ilustrace - výstup aplikace (bez vykreslení proteinu) po 4 milionech vzorcích s nalezenou druhou unikátní cestou (nahoře).