Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Bakalářská práce

Vizualizace a editace molekul proteinu

Plzeň, 2009 Zdeněk Janáček

Poděkování

Rád bych chtěl poděkoval vedoucímu své bakalářské práce Ing. Michalovi Zemkovi za odborné vedení, ochotné poskytnutí cenných rad a připomínek při vypracování.

Prohlášení

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

V Plzni dne 14.5.2010

Zdeněk Janáček

Abstract

Visualization and editing of protein molecules

Proteins are natural substances and they are the essence of all living organisms. Proteins can serve various functions. A minor change in a protein structure can lead to a change in its properties and functions. These properties' changes are used by chemists to find a suitable protein with improved properties. Chemists use many programs for studying proteins' structures. One such program is CAVER. My task was to create GUI which is incorporated as a plug-in a panel of the CAVER program. This GUI provides various options for editing proteins and is implemented in Java. It was also necessary to create a bridge between the GUI and the C# library which carries out all computing. The bridge was developed C++/CLI as a native library.

Obsah

1 Úvod....2
2 Teoretická část....4
2.1 Tunely v molekulách proteinu....4
2.2 Funkce výsledného programu....7
2.2.1 Použité aminokyseliny:....8
2.3 Java NetBeans Platform....9
2.3.1 Vytvoření NetBeans Platformy....9
2.4 Seznámení s JNI....10
2.4.1 Popis JNI....10
2.4.1.1 Java Platforma....10
2.4.1.2 Hostitelské prostředí....10
2.4.1.3 Role JNI....11
2.4.2 Vytvoření Nativní knihovny....11
2.4.2.1 Deklarace nativní metody....11
2.4.2.2 Zkompilování Class....12
2.4.2.3 Vytvoření hlavičkového souboru Nativní metody....12
2.4.2.4 Implementace Nativní metody....13
2.4.2.5 Kompilace zdrojového souboru C++ a vytvoření Nativní knihovny....13
2.4.2.6 Spuštění Java aplikace....14
2.4.3 JNI typy a datové struktury....14
2.4.3.1 Java primitivní typy....14
2.4.3.2 Typy Java objektů....14
2.4.3.3 Typy deskriptorů....15
2.4.3.4 Práce s řetězci....16
2.4.4 Používané JNI funkce....17
2.4.4.1 Operace s třídami....17
2.4.4.2 Operace s metodami....19
2.4.4.3 Operace s objekty....19
2.4.4.4 Přístup k atributu objektu....21
3 Realizační část....22
3.1 Kriteria výběru....22
3.2 Struktura vypracování....23
3.3 Popis GUI....24
3.4 Běh a popis aplikace....25
3.4.1 Popis programu....26
3.4.2 Použití metod....26
3.4.3 Metoda Load Protein....26
3.4.4 Metoda Compute Channel....27
3.4.5 Metoda Mutate....27
3.4.6 Metoda SaveMutantProteinToPDB....27
3.4.7 Nativní knihovna sloužící jako můstek....27
3.4.8 Knihovna pro výpočet tunelu a mutací....28
3.5 Problémy s implementací....28
3.5.1 Používání knihoven....28
3.5.2 Debugování knihovny....29
4 Závěr....31
Přehled pojmů a zkratek....32
Reference....33
Přílohy....34

1 Úvod

Proteiny jsou přírodní látky a jsou podstatou všech živých organismů. Proteiny mohou zastávat různé funkce. Drobná změna struktury může vést ke změně vlastností a funkcí. Tyto změny vlastností využívají chemici k nalezení proteinu s lepšími vlastnostmi. Chemici používají mnoho programů ke zkoumání struktury bílkovin, odborně proteinů. Jedním z těchto programu je Caver.

Protein se skládá z několika desítek až tisíců aminokyselin, které jsou spojeny vazbami. Chemici se snaží, za pomoci drobných změn v proteinu, změnit jeho vlastnosti. Místo, které se chemici snaží upravit, nazývají aktivním místem. Změny jsou realizovány dopravením určité molekuly do aktivního místa z povrchu proteinu, nebo opačným směrem. Při této úpravě je důležité zachovat strukturu proteinu, proto se hledají dráhy, které neprotínají žádný atom proteinu a zároveň spojují povrch proteinu a aktivní místo. Tyto dráhy se nazývají tunely.

Program Caver umožňuje kompletní 3D vizualizaci struktury proteinu a tunelu, bližší popis programu Caver můžete nalézt zde [5]. Tato bakalářská práce rozšiřuje možnosti tohoto programu. Toto rozšíření bude přímo implementováno do programu Caver. Chemici budou mít k dispozici nástroje pro úpravu struktury proteinu. Tyto úpravy budou realizovány záměnou některých aminokyselin, které obklopují tunel, za jiné. Chemici budou moci volit, zda chtějí pouze zaměnit některé z aminokyselin beze změny tunelu, nebo jestli se má při záměně aminokyselin tunel rozšířit nebo zúžit.

Mým úkolem bylo sestrojit editor, který bude nabízet požadované možnosti. Editor slouží pro zobrazení proteinu, tunelu v proteinu a zobrazení aminokyselin, které tunel obklopují a jejich strukturní detaily. Dále nabízí možnosti pro změnu tunelu: zúžení, roztáhnutí a záměnu zvolených aminokyselin. Případně uložení těch nejzajímavějších z nových "mutací" proteinu. Jelikož Caver je Java aplikace, musel být tento editor implementován v Javě. Editor musí komunikovat s C# knihovnou, která provádí veškeré výpočty tunelů. Proto bylo nutné vytvořit můstek v C++/CLI, což je nástupce managed C++. V C++/CLI je možné vytvářet jak managed tak unmanaged kód v jednom projektu, což je potřeba pro vytvoření a používání nativní knihovny. Pro stručnost budeme dále používat místo C++/CLI jen C++. Můstek je vytvořen jako nativní knihovna pomocí JNI.

Editor se od původního zadání značně zjednodušil. Zobrazení proteinu, tunelu a aminokyselin je realizováno formou tabulek s číselnými hodnotami, které popisují strukturu proteinu a průběh tunelu. Možnosti pro úpravu tunelu zůstaly stejné. Oproti tomu se ztížila implementace můstku, který musí danou datovou strukturu předat ze C# knihovny do Javy a zpět. Tyto změny funkce a implementace probíhaly dle požadavků supervizora.

2 Teoretická část

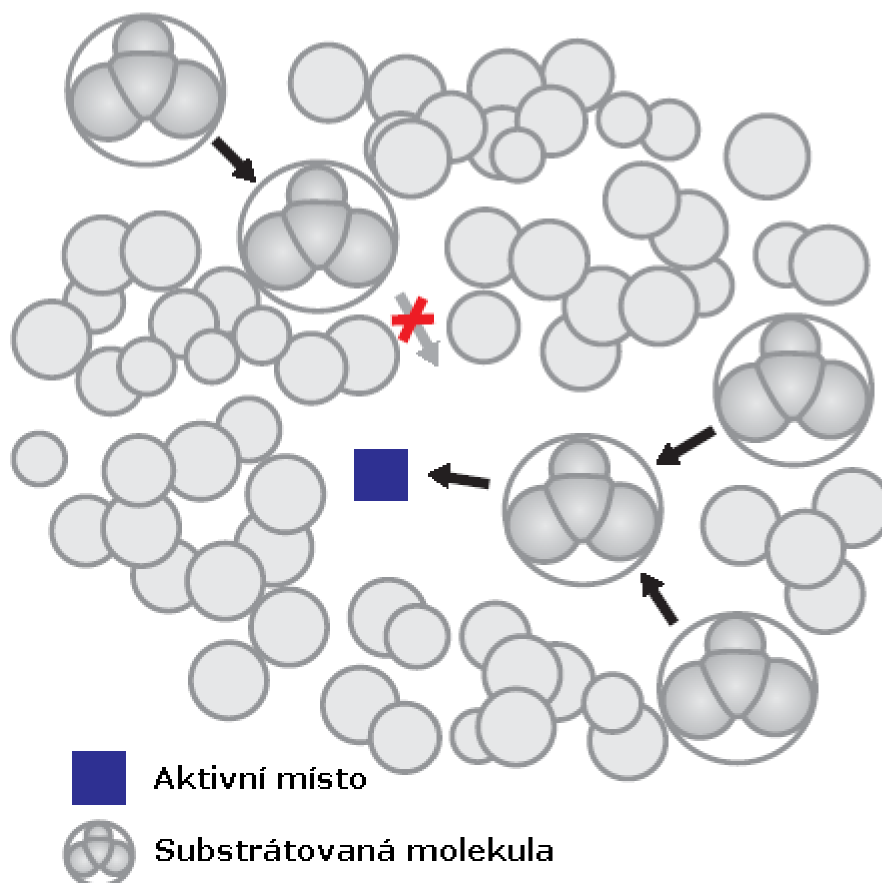
Tato teoretická část obsahuje další čtyři podkapitoly. První z nich se zabývá podrobným popisem postupu pro nalezení ideálního tunelu v proteinu. Druhá kapitola je zaměřena na popis výsledné funkce programu. Třetí část popisuje možnosti Java NetBeans Platform. Poslední kapitola je nejrozsáhlejší, jelikož obsahuje podrobný popis použití JNI, které je klíčové k vytvoření můstku.

2.1 Tunely v molekulách proteinu

Seznámíme se s metodou použitou na zkoumání vlastností proteinu, která je podrobněji popsána zde [1]. Podle dlouhodobé biochemické studie je zřejmé, že chování molekuly proteinu závisí na existenci tunelů, které vedou z vnitřku molekuly na její povrch. Metoda výpočtu tunelů je založena na vypočtení geometrie a využívá Voroinův diagram a jeho duální strukturu - Delaunayovu triangulaci. Výhodou této metody při počítání tunelů je bezesporu vysoká kvalita přesnosti a odpovídající čas výpočtu, který je kratší než při použití starších metod viz [2].

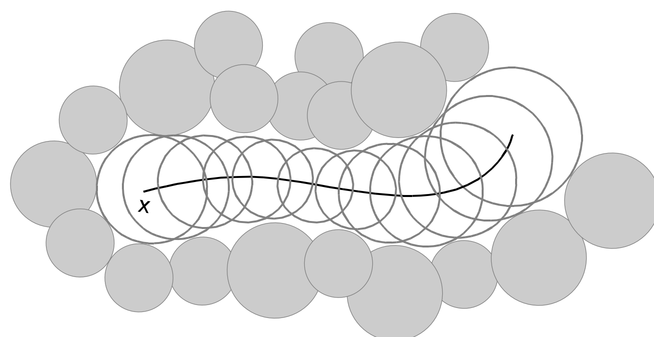
Je zjištěno, že vlastnosti proteinových molekul jsou založeny na reaktivitě proteinu, která je úzce spojena s přítomností drah vedoucích z povrchu proteinu do příslušné "dutiny" uvnitř proteinu. Toto místo je označováno jako aktivní místo. V tomto aktivním místě se odehrávají chemické reakce mezi proteinem a další molekulou. Tato dráha, též nazývaná tunel, je zde nutná proto, aby vybraná molekula mohla dosáhnout aktivního místa bez protnutí proteinových atomů. Obrázek 2.1 znázorňuje substrátovou molekulu a dva odlišné tunely vedoucí do aktivního místa. Dále je vidět z obrázku, že třetí tunel nedosáhl aktivního místa, jelikož substrátová molekula neprošla mezi atomy proteinu.

Nutno zdůraznit, že geometrická existence samotného tunelu není dostačující, aby byl zaručen přístup substrátové molekuly do aktivního místa. Schopnost proteinu reagovat se substrátem je založena na mnoha odlišných fyzikálních a chemických faktorech. Z geometrického pohledu může vypočtený tunel poskytnout informace, které v budoucnu pomohou chemikům, aby se zaměřili na určité části proteinu.



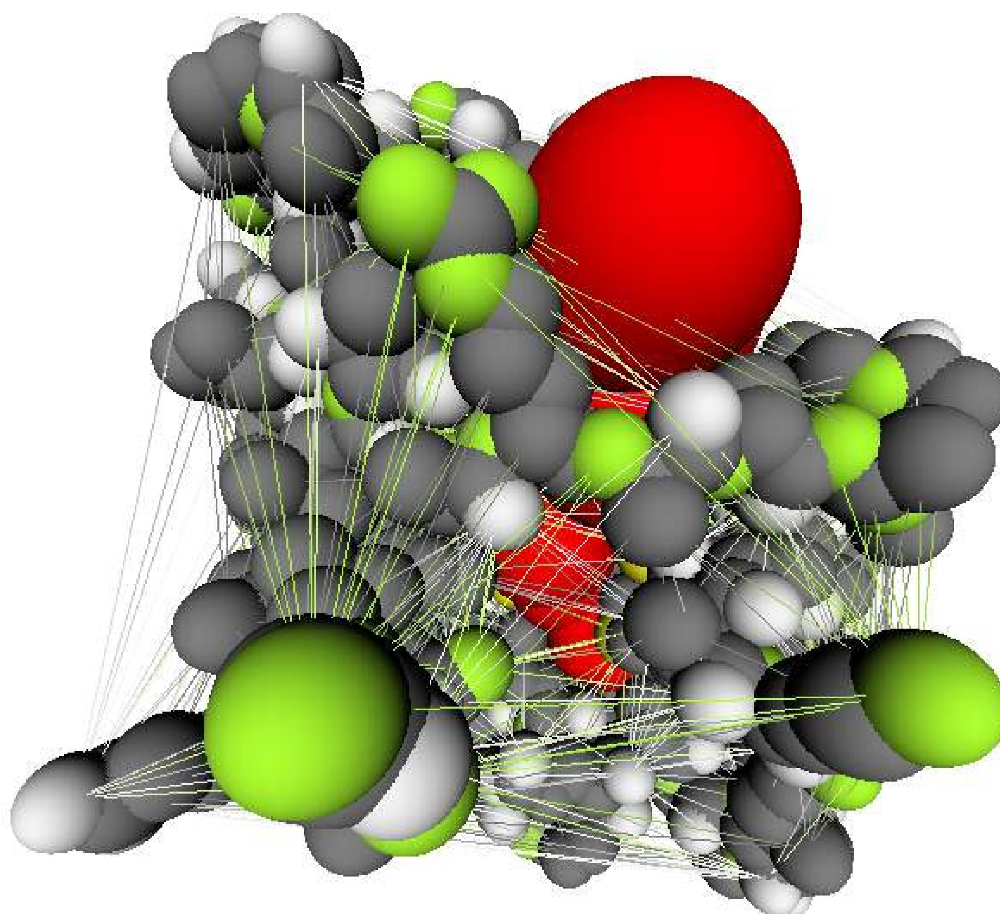
Obrázek 2.1.1 Dva odlišné tunely vedoucí do aktivní zóny.

Geometrie molekuly proteinu je velmi složitá, proto je nutno si tuto molekulu zjednodušit. Molekula proteinu se zjednodušuje na řadu koulí, kdy každá koule představuje jeden atom. Každá z těchto koulí se nalézá na určité pozici v 3D prostoru a má vhodný Van der Waals poloměr. Tunel bude tvořen koulemi, které neprotínají žádný atom proteinu. Středy těchto koulí budou určovat osu tunelu. Na obrázku 2.2 je znázorněn takovýto tunel. Šedivé koule znázorňují atomy proteinu a šedivé kružnice znázorňují koule tunelu. Křížek značí aktivní místo. Celá základní molekula (substrátová) je aproximována koulí. Díky těmto zjednodušením mají námi konstruované tunely vždy kruhový profil (na příčném řezu) a různé tunely vedoucí do téhož aktivního místa pak můžeme porovnávat podle průměru jejich minimálního průřezu. Průřez chápeme jako poloměr jednotlivých kruhových částí tunelu. Minimálním průřezem pak chápeme místo, kde má tunel nejmenší poloměr.



Obrázek 2.1.2 Tunel v proteinu

Pro větší představu o vzhledu proteinu a tunelu je zde obrázek 2.3, na němž můžeme vidět příklad 3D modelu reálného vzhledu nalezeného tunelu. Tunel je zde reprezentován jako řetězec červených koulí. Ostatní koule reprezentují atomy proteinu.



Obrázek 2.1.3 3D zobrazení nalezeného tunelu v proteinu

2.2 Funkce výsledného programu

Pro správný chod programu je nutné jako první do programu načíst protein a poté vypočítat tunel v proteinu, poté je možno tunel i protein upravovat. Níže jsou přesně popsány možnosti programu (tzn. jaké možnosti budou mít chemici k dispozici v uživatelském rozhraní).

- **Načtení proteinu:**

  Načte chemikem zadaný protein z pdb souborů, které obsahují popis struktury proteinu a prostorové souřadnice jednotlivých atomů proteinu. Tyto soubory je možno získat z Protein databank, které jsou k nalezení zde [7]

- **Nalezení tunelu v proteinu:**

  Nalezení tunelu z povrchu proteinu do chemikem zadaného aktivního místa a následné zobrazení popisů koulí tunelu a aminokyselin, které tunel obklopují

- **Náhrada aminokyselin:**

  Pro každou aminokyselinu je možnost vybrat množinu náhradníků pro záměnu a tím vytvořit mutovaný protein z těchto záměn.

- **Úpravy tunelu:**

  Je možné hledat "mutaci" proteinu, která zadaný tunel rozšíří nebo naopak zúží. Chemici mají k dipozici dvě možnosti: rychlý, ale nepřesný výpočet nebo pomalý výpočet s důvěryhodnějšími výsledky. Mutací proteinu chápeme protein, který vznikl z původního proteinu záměnou některých aminokyselin.

- **Možnosti při mutaci:**

  Při mutaci můžeme upravovat tunel (zúžení, rozšíření) nebo jen provést samotnou mutaci a to beze ohledu na tunelu.

- **Uložení:**

  Jakýkoli mutovaný protein můžeme uložit pro pozdější využití.

2.2.1 Použité aminokyseliny:

Jelikož všechny proteiny obsahují pouze 20 základních aminokyselin, program akceptuje pouze tyto. Uživatelské rozhraní bude využívat pouze jejich zkratky. V tabulce 2.1 je uveden jejich seznam. Podrobný popis a strukturu jednotlivých aminokyselin naleznete zde [6]. Program většinou využívá jednopísmenné zkratky pro popis aminokyseliny, jen při zobrazení primární struktury proteinu jsou použity jednopísmenné zkratky. Barvy jednotlivých aminokyselin udávají skupiny, do kterých spadají. Těmito barvami jsou značeny aminokyseliny v primární struktuře.

aminokyselina	třípísmenná zkratka	jednopísmenná zkratka
glycine	Gly	G
alanine	Ala	A
valine	Val	V
leucine	Leu	L
isoleucine	Ile	I
methionine	Met	M
phenylalanine	Phe	F
tryptophan	Trp	W
proline	Pro	P
serine	Ser	S
threonine	Thr	T
cysteine	Cys	C
tyrosine	Tyr	Y
asparagine	Asn	N
glutamine	Gln	Q
aspartic acid	Asp	D
glutamic acid	Glu	E
lysine	Lys	K
arginine	Arg	R
histidine	His	H

Tabulka 2.1: Seznam použitých aminokyselin

2.3 Java NetBeans Platform

Tato kapitola se zabývá popisem a možnostmi využití Java Netbeans Platform. Tato platforma poskytuje spolehlivou a flexibilní aplikační architekturu. Je založena na modulární architektuře, což znamená, že je snadné vytvářet aplikace, které jsou robustní a snadno rozšiřitelné. Tato platforma je použita pro vytvoření GUI.

NetBeans Platform je obecný rámec pro Swing aplikace a poskytuje:

- Podporu pro správu modulů a práci s nimi
- Správu oken
- Správu akcí
- Automatickou podporu pro JNLP
- Nápovědu
- atd.

NetBeans Platform poskytuje všechny tyto nástroje v jednom balíku. Popis všech funkcí platformy naleznete zde [9]. Nemusíme ručně psát kódy k základním prvkům Swing aplikace. Automaticky vygenerovaný kód k těmto prvkům lze následně upravovat. V projektu lze vytvářet další Java třídy, které komunikují s modulem NetBeans Platform. Pro vytvoření Java struktur je čerpáno z [8].

2.3.1 Vytvoření NetBeans Platformy

Hlavní výhoda NetBeans Platform je jeho modulární architektura. Druhá výhoda je, že NetBeans Platform se spoléhá na Swing UI toolkit, což je oficiální nástroj pro vytváření uživatelského rozhraní v Javě.

Vytvoření projektu NetBeans Platform provází několik kroků. Jako první vytvoříme hlavní projekt (New Project). V kategoriích projektu zvolíme NetBeans Modules a vybereme projekt: Netbeans Platform Application. Poté do tohoto projektu přidáme nový modul (New Module Project). Tím nám vznikne nový modul a jeho struktura je zobrazena v okně projektů. Na konec do tohoto modulu přidáme nový soubor (New File). V kategorii Other - Module Development zvolíme Window Component. Nové okno se přidá ke zdrojové struktuře našeho modelu, spolu s několika pomocnými XML soubory. Většinou se toto okno otvírá přes soubor "TextTopComponent.java". Zde už je přístupná paleta se Swing komponentami.

2.4 Seznámení s JNI

V této kapitole se zmíním o všech nutných znalostech pro práci s JNI. Toto rozhraní je využito při tvorbě můstku mezi C# a Javou. Tento můstek je implementován jako nativní knihovna. Můstek je vytvořen v C++/CLI. Rozsáhlejší popis funkcí a dovedností JNI můžeme nalézt zde [3] a [4].

2.4.1 Popis JNI

JNI je prvek platformy Java. Aplikace, které používají JNI, mohou volat nativní knihovny napsané v jazycích C, C++,atd. Abychom dokázali popsat roli JNI, je nutné si nejprve popsat základní pojmy jako je Java Platform a hostitelské prostředí.

2.4.1.1 Java Platforma

Platforma Java je programovací prostředí, které se skládá z JVM (což je modul, který zpracovává tzv. mezikód (Java bytecode)) a JAPI (což je rozhraní, které určuje jakým způsobem se volají procedury nebo funkce dané knihovny nebo jiného programu ze zdrojového kódu programu). Java aplikace jsou psány v programovacím jazyce Java. Java kompilátor převede zdrojový kód do Java bytecode (jinými slovy strojový kód, který je srozumitelný pro JVM) a JRE převede bytecode do nativního strojového kódu, který může být spuštěn na jakémkoli virtuálním stroji JVM. JRE je prostředí pro spuštění aplikací (a vývojových nástrojů) a obsahuje JVM a JAPI. JAPI se skládá z předdefinovaných tříd. Při jakémkoliv zavedení platformy Java je zaručena podpora programovacího jazyka Java, JVM a JAPI. Java Platformy jsou běžně nasazeny na vrcholu hostitelského prostředí.

2.4.1.2 Hostitelské prostředí

Java Platformy jsou běžně nasazeny na vrcholu hostitelské prostředí. Java Platform nabízí sadu funkcí, které aplikace mohou spolehlivě využít nezávisle na hostitelském prostředí. Hostitelské prostředí představuje hostitelský operační systém, soubor původních knihoven a soubor instrukcí procesoru. Nativní aplikace jsou napsány v nativních programovacích jazycích (C, C++, atd.), sestaveny do hostitelských specifických binárních kódů a spojeny s nativními knihovnami. Nativní aplikace a nativní knihovny jsou většinou závislé na konkrétním hostitelském prostředí. Aplikace jsou většinou postavené pro jeden operační systém

2.4.1.3 Role JNI

Jestliže je Java platforma nasazena na vrcholu hostitelského prostředí, je nutné, aby Java aplikace úzce spolupracovala s nativními kódy napsanými v jiném jazyce. JNI je silný prvek, který umožňuje využít Java platformy a ještě využít kód napsaný v jiném jazyce jako část implementace JVM. JNI je obousměrné rozhraní,které umožňuje Java aplikaci vyvolat nativní kód a naopak.

JNI je stavěno na situace, kdy kombinuje Java aplikace s nativním kódem. Jako obousměrné rozhraní podporuje dva typy nativního kódu: nativní knihovny a nativní aplikace. Můžete použít JNI k napsání nativních metod, které umožňují Java aplikaci zavolat funkce implementované v nativní knihovně.

2.4.2 Vytvoření Nativní knihovny

Nativní knihovna se vytváří postupně podle zavedeného postupu. Tento postup můžeme vidět v tabulce 2.2. V prvním sloupci vidíme operace, které musíme provést v tomto pořadí. V druhém sloupci vidíme výstup dané operace. Tedy formát souboru, který daná operace vytvoří. Posledním krokem je spuštění java-aplikace, což může proběhnou až po sestavení spustitelného souboru *.exe nebo spustit aplikaci přímo ve vývojovém prostředí.

prováděné operace	**výstup operace**
napsáním Java kódu	*.java
zkompilováním	*.class
vygenerování hlavičkového souboru	*.h
implementace nativní metody	*.c
kompilace nativního kódu a vytvoření knihovny	*.dll
spuštění Java aplikace	

Tabulka 2.2: Postup pro vytvoření nativní knihovny

2.4.2.1 Deklarace nativní metody

Před vytvořením nativní knihovny, která v našem případě implementuje můstek mezi Javou a C#, je nutné nejprve vytvořit v java-programu deklaraci nativních metod, které bude výsledná aplikace používat. Výsledný java-program poté bude volat C++ nativní knihovnu skrze tyto nativní metody. Tyto metody se deklarují pomocí direktivy "native". V kódu 2.1 vidíme jednoduchý příklad deklarace a volání nativní metody v Javě.

```
class název_třídy {
      public static native int Load();
      public static void main(String[] args) {
            Load();
            }
}
```

Kód 2.1: Voláni nativní metody

2.4.2.2 Zkompilování Class

Pro kompilaci použijeme javac, což je primární Java překladač a je součástí JDK. JDK je soubor základních nástrojů pro vývoj aplikací pro Javu. Překladač přijímá zdrojový kód, odpovídající specifikaci jazyka Java a vytváří bytecode odpovídající specifikaci JVM.

Použití: javac název_třídy.java

Výstup: název_třídy.class

2.4.2.3 Vytvoření hlavičkového souboru Nativní metody

Pro vytvoření použijeme javah, což je aplikace, která dovoluje vytvoření hlavičkového souboru (v programovacím jazyce C/C++) z přeloženého kódu Javy (tedy .class). Hlavičku je možné vytvořit ručně, ale použitím javah zabráníme zbytečným chybám. Zde si popíšeme jeho použití.

Použití: javah -jni název_třídy

Jestliže je třída částí balíku, musíme zadat i název balíku, ve kterém je třída obsažena

Použití: javah -jni název_balíku.název_třídy

Výstup: název_třídy.h

Formát vygenerované hlavičky můžeme vidět v kódu 2.2. První argument je pointr, který odkazuje na JNI rozhraní a je typu JNIEnv. Druhý argument se liší v závislosti na tom, jestli je metoda static nebo non-static (tedy nestatická). Jestliže je nativní metoda nestatická, je to odkaz na java-objekt, z něhož byla nativní metoda volaná. Jestliže je nativní metoda statická, je to odkaz na Java třídu.

V kódu 2.3 jednoduchý příklad. Název metody je "LoadProtein", je umístěna v balíku "org" ve třídě "Call". Metoda má návratovou hodnotu objekt a má dva parametry.

```
JNIEXPORT  návratový_typ_metody  JNICALL
Název metody (JNIEnv *, jobject/jclass);
```

Kód 2.2: Hlavička nativní metody

```
JNIEXPORT jobject JNICALL Java_org_Call_LoadProtein
(JNIEnv *env, jclass cls, jstring path, jobject res){
…
}
```

Kód 2.3: Ukázka hlavičky metody LoadProtein

2.4.2.4 Implementace Nativní metody

Vytvořený hlavičkový soubor nám pomůže napsat C implementaci pro nativní metody. Funkci, kterou napíšeme, musí být vytvořena z vygenerované hlavičky. Soubor musí vždy obsahovat:

#include <jni.h> - hlavičkový soubor, který obsahuje informace o nativním kódu a je nutný k volání JNI funkcí.

#include ”název_třídy.h” - hlavičkový soubor, který je generován javah

V kódu 2.4 vidíme příklad vzhledu metody public static native int Load(). Metoda je bez parametru a jako návratovou hodnotu vrací celé číslo (int)

```
JNIEXPORT  int JNICALL  Load (JNIEnv *env, jobject obj){
                tělo metody
                return 0;
                }
```

Kód 2.4: Příklad metody Load

2.4.2.5 Kompilace zdrojového souboru C++ a vytvoření Nativní knihovny

Různé operační systémy podporují různé způsoby, jak vytvořit nativní knihovnu. Pro operační systémy Win32 je nejjednodušším způsobem založit v odpovídajícím vývojovém prostředí dll projekt, který rovnou při sestavení vytvoří knihovnu.

V ostatních případech použijeme příkaz:

cl -Ic:\java\include -Ic:\java\include\win32 -MD -LD název_třídy.c -název_knihovny.dll

Pro vytvoření knihovny na operačním systému Solaris použijeme podobný příkaz:

cc -G -I/java/iclude -I/java/include/solaris název_třídy.c -o název_knihovny.so

2.4.2.6 Spuštění Java aplikace

V tomto okamžiku máme dvě složky připravené ke spuštění programu. Class soubor (*.class), který volá nativní metodu a nativní knihovnu (Library.dll), která provádí nativní metodu. Jelikož třída má metodu main, můžeme spustit program na Solaris nebo Win32 příkazem: java název_třídy

Je důležité nastavit správnou cestu k nativní knihovně, jinak Java knihovnu nenačte.

2.4.3 JNI typy a datové struktury

2.4.3.1 Java primitivní typy

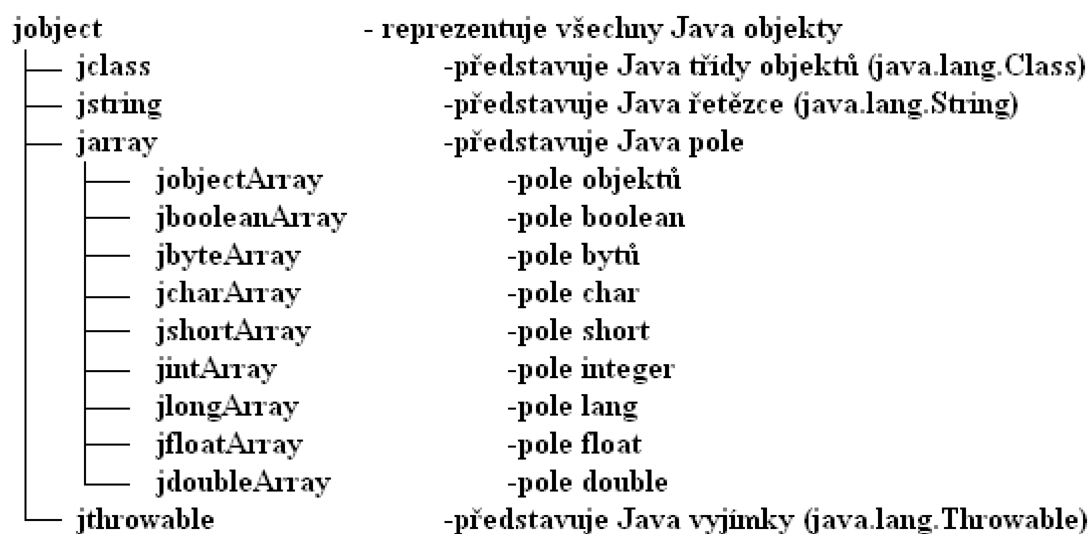
Nativní metoda může přímo používat pouze některé datové typy Javy. Tabulka 2.3 popisuje mapování těchto Java typů na nativní typy a jejich popis.

Java typy	Nativní typy	Popis
boolean	jboolean	8 bits unsigned
byte	jbyte	8 bits signed
char	jchar	16 bits unsigned
short	jshort	16 bits signed
int	jint	32 bits signed
long	jlong	64 bits signed
float	jfloat	32 bits
double	jdouble	64 bits
void	void	n/a

Tabulka 2.3: Primitivní typy Java

2.4.3.2 Typy Java objektů

JNI obsahuje řadu referenčních typů, které odpovídají různým druhům Java objektů. JNI referenční typy jsou organizovány v hierarchii jak znázorňuje obrázek 2.3.



Obrázek 2.4.3.2.1 Hierarchický strom JNI ref. typů

2.4.3.3 Typy deskriptorů

JNI využívá JVM k zastoupení Java proměnných tzv. deskriptorů. Tyto deskriptory popisují proměnné, které volají JNI funkce z C++ do Javy. Dovolují tedy JNI funkci zavolat metodu jakéhokoli typu s jakýmkoli parametrem.

Deskriptor tříd (class):

Deskriptor tříd představuje název třídy nebo rozhraní. To může být odvozeno z plně kvalifikovaného jména třídy nebo jména rozhraní, jak jsou definovány v JLS. Znak "." se nahradí za znak "/". Jako příklad můžeme uvézt deskriptor třídy pro java.lang.String : "java/lang/String".

Deskriptor polí (Fields):

V tabulce 2.4 můžeme vidět osm základních datových typů Javy a jejich deskriptor pro JNI.

Deskriptor referenčního typu vždy začínají "L" a jsou ukončeny ";". Deskriptory polí jsou tvořeny stejně jako deskriptor pole třídy. V tabulce 2.5 jsou vidět některé z

nejpoužívanějších deskriptorů. Deskriptor pole se tvoří pomocí znaku "[", po němž následuje deskriptor datového typu pole (viz. tabulka 2.4).

Popis Pole	Java typ
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double

Tabulka 2.4: Deskriptor polí

Popis Pole	Java typ
"Ljava/lang/String;"	String
"[type"	type[]
"[Ljava/lang/Object;"	Object[]

Tabulka 2.5: Deskriptor polí

Deskriptor metod (Methods):

Deskriptor metod se skládá z několika částí. První je tvořena závorkami, které obsahují všechny parametry. Mezi těmito parametry nejsou žádné mezery ani jiné oddělovače. Druhá část obsahuje návratový typ metody. "V" se používá pro metody bez návratového typu, v Jave označovaného jako "void". Tabulka 2.6 ukazuje některé příklady deskriptorů metod JNI a jejich odpovídající metody:

Popis Metody	Java typ
"()Ljava/lang/String;"	String f();
"(Iljava/lang/Class;)J"	long f(int I, Class c);
"([B)V"	void f(byte[] bytes)

Tabulka 2.6: Příklady deskriptorů metod

2.4.3.4 Práce s řetězci

Datový typ jstring reprezentuje textové řetězce v jazyce JVM a je odlišný od běžného typu řetězce C / C++. Není možné použít jstring jako normální C řetězec. Použití by nejspíše znamenalo pád JVM. Nativní kód metod musí používat vhodné JNI funkce pro převod jstring objektů do C / C++ řetězce. JNI podporuje konverzi do i z Unicode a UTF-8 řetězce.

V kódu 2.5 můžeme vidět funkci Java_Prompt_getLine, která volá JNI funkci GetStringUTFChars, aby přečetla obsah řetězce. Tato funkce je k dispozici prostřednictvím ukazatele JNIEnv rozhraní. To převádí jstring reference, typicky reprezentováno JVM jako sekvence Unicode, do C řetězce reprezentováno v UTF-8 formátu. Funkce níže předpokládá, že textový řetězec bude mít maximálně 127 znaků.

```
JNIEXPORT jstring JNICALL
 Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt){
     char buf[128];
     const jbyte *str;
     str = (*env)->GetStringUTFChars(env, prompt, NULL);
     (*env)->ReleaseStringUTFChars(env, prompt, str);
     return (*env)->NewStringUTF(env, buf);
     }
```

Kód 2.5: Převod jstring objektu

2.4.4 Používané JNI funkce

Pro správné používání JNI je nutné naučit se pracovat s JNI funkcemi. V této kapitole si popíšeme funkci a způsob použití těchto funkcí.

2.4.4.1 Operace s třídami

FindClass:

Funkce FindClass získá odkaz na třídu. Při vytváření Java struktury v nativní knihovně, je tato funkce použita jako první, jelikož všechny další operace probíhají právě nad touto třídou. Metoda vrací odkaz na třídu, nebo NULL, pokud nebyla třída nalezena. V kódu 2.6 můžeme vidět obecný vzhled funkce.

Funkce má dva parametry:

- env – pointer JNI rozhraní
- name – plně kvalifikovaný název třídy (tj. jméno balíku, následována "/" za nímž následuje název třídy). Začíná-li jméno třídy "["(pole s popisem charakteristik), vrací pole objektu třídy.

```
jclass FindClass(JNIEnv *env, const char *name);
```

Kód 2.6: Funkce FindClass

2.4.4.2 Operace s metodami

GetMethodID:

V jazyce Java existuje mnoho druhů metod. JNI podporuje kompletní sadu funkcí, které nám umožňují jejich volání z nativního kódu. V našem případě používáme nejčastěji nestatické metody. Tyto metody musí být použity pouze na konkrétní instanci třídy. Funkce vrací ID metody dané třídy nebo rozhraní. Pokud metoda neexistuje, nebo nebyla nalezena, vrátí metoda NULL. Funkce vykonává hledání metody v dané třídě. V kódu 2.7 může vidět obecný předpis funkce. Funkce má níže popsané parametry:

- env – pointer na JNI rozhraní
- class – třída Java objektu
- name – jméno metody modifikované jako UTF-8 řetězec
- sig – deskriptor metody modifikované jako UTF-8 řetězec

```
jmethod GetMethodID(JNIEnv *env, jclass class,
const char *name, const char *sig);
```

Kód 2.7: Funkce GetMethodID

2.4.4.3 Operace s objekty

NewObject:

JNI funkce NewObject volá konstruktor stanovený methodID. V kódu 2.8 můžeme vidět obecný popis funkce, která vytvoří nový Java objekt. Argumenty třídy nesmí odkazovat na pole tříd. Vrací Java objekt, nebo NULL v případě, že Java objekt nebyl nalezen. Funkce má následující parametry:

- env – pointer na JNI rozhraní
- class – třída Java objektu
- methodID - udává, který konstruktor třídy se má zavolat. Konstruktor získáme použitím funkce GetMethodID(), jako parametr name zadáme: <int> a jako parametr sig: (V).

```
jobject NewObject(JNIEnv *env, jclass class,
jmethodID methodID, ...);
```

Kód 2.8: Funkce NewObject

2.4.4.4 Přístup k atributu objektu

GetFieldID:

Programovací jazyk Java podporuje dva druhy atributů. JNI funkce poskytují nativní kód, který lze použít k získání nestatického atributu v objektech nebo statického atributu v třídách. V našem případě používáme nestatické atributy. Funkce GetFieldID vrací ID atributu pro nestatický atribut třídy. Atribut je upřesněn jménem a popisem (deskriptorem). Nelze použít k získání délky atributu pole. Vrací ID atribut, nebo NULL v případě selhání. Obecný popis funkce je viděn v kódu 2.9. Funkce má tyto parametry:

- env – pointer JNI rozhraní
- class – třída Java objektů
- name – jméno atributu modifikované jako UTF-8 řetězec
- sig - podpis atributu modifikované jako UTF-8 řetězec

```
jfieldID GetFieldID(JNIEnv *env, jclass class,
const char *name, const char *sig);
```

Kód 2.9: Funkce GetFieldID

Get<type>Field a Set<type>Field:

Tyto funkce slouží pro přístup k hodnotám jednotlivých atributů. Tabulka 2.7 popisuje Set/Get<type>Field rutinní názvy a typy hodnot. Set slouží pro nastavení hodnoty a Get pro předání hodnoty. Při použití těchto funkcí musíme použít jeden z rutinních názvů z tabulky a nahradit jej za odpovídající nativní typ z tabulky. Obecný popis funkce můžeme vidět v kódu 2.10. Funkce mají tyto parametry:

- env – pointer JNI rozhraní
- obj – Java objekt (nesmí být NULL)
- fieldID – platný ID atribut, tento atribut získáme voláním metody GetFieldID

Set / Get<type>Field Routine Name	Native Type
Set / GetObjectField()	jobject
Set / GetBooleanField()	jboolean
Set / GetByteField()	jbyte
Set / GetCharField()	jchar
Set / GetShortField()	jshort
Set / GetIntField()	jint
Set / GetLongField()	jlong
Set / GetFloatField()	jfloat
Set / GetDoubleField()	jdouble

Tabulka 2.7: Popis rutinních názvů a jejich typy hodnot

```
Set/Get<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID);
```

Kód 2.10: Funkce Set/Get Field

3 Realizační část

Popis realizační části se skládá z pěti podkapitol. První z kapitol popisuje kritéria pro výběr daných programovacích jazyků. Druhá kapitola znázorňuje strukturu vypracování. Třetí kapitola popisuje možnosti uživatelského rozhraní. Čtvrtá kapitola popisuje běh programu a základní realizační části. Poslední kapitola je zaměřena na všechny problémy vzniklé při implementaci.

3.1 Kriteria výběru

Jak bylo zmíněno v úvodu, měl jsem za úkol vytvořit GUI a můstek mezi C# knihovnou a Javou. Po přezkoumání možných řešení jsem za pomoci svého vedoucího zvolil vytvořit můstek v jazyku managed C++/CLI. C++/CLI je nadstavba C++ pro práci s .NET (což je prostředí potřebné pro běh aplikací). Umožňuje vytvářet tzv. mixed assembly obsahující jak managed tak unmamaged kód. To mi dovoluje vytvořit nativní knihovnu, přes kterou mohu volat z Javy metody C# knihovny a předávat nazpět data pomocí JNI. GUI jsem vytvořil v Java Netbeans Platform. Vyžití této platformy bylo nezbytné, jelikož GUI má být později implementováno do java-programu Caver.

3.2 Struktura vypracování

V této části jsou popsány všechny datové struktury výsledného projektu. Tuto strukturu můžeme vidět na obrázku 3.1. Celý model má tyto tři části:

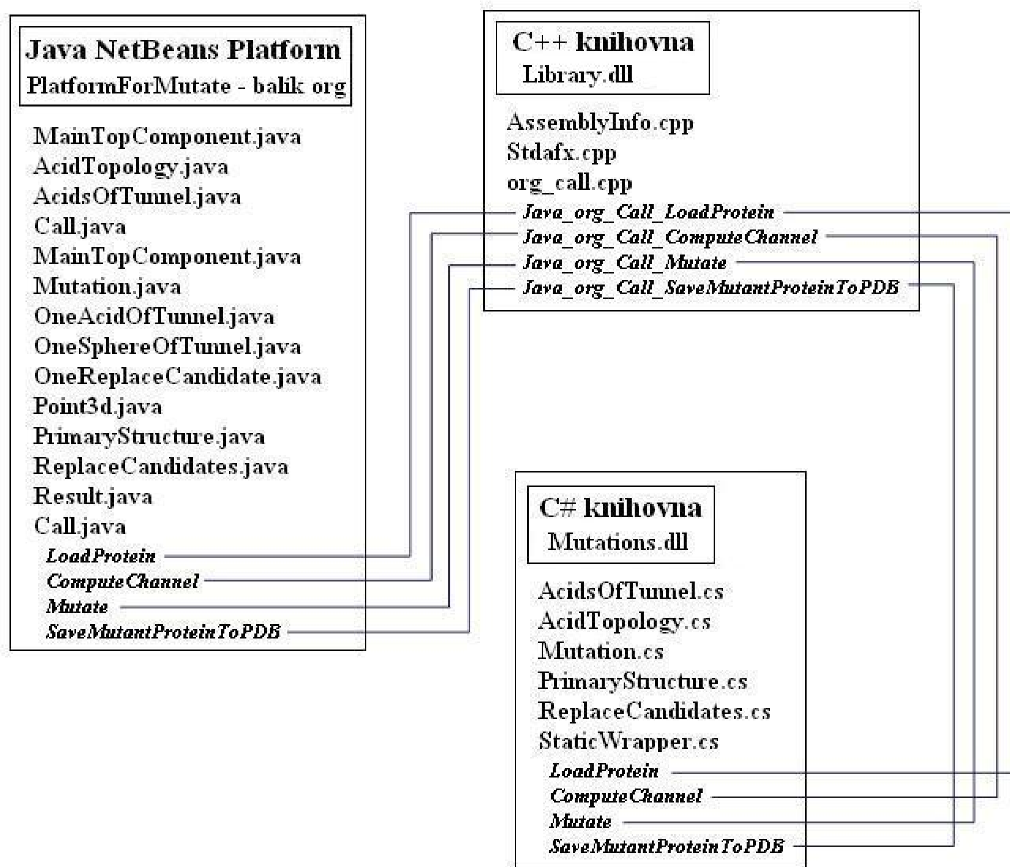
- java-projekt
- C++ nativní knihovna Library.dll
- C# knihovna Mutations.dll

Mezi nejdůležitější třídu v java-projektu patří MainTopComponent.java, která obsahuje popis GUI. Další důležitá třída je Call.java. Tato třída obsahuje popis čtyř nativních metod, které se použijí při volání C++ nativní knihovny. Dále tato třída obsahuje statickou část, kde se načítá C++ nativní knihovna a C# knihovna Mutations.dll. Ostatní třídy popisují datovou strukturu proteinu, tunelu a mutace proteinu.

C++ knihovna obsahuje tři C++ soubory:

- AssemblyInfo.cpp obsahuje hlavní informace o sestavení programu..
- Stdafx.cpp je zdrojový soubor, který je standardně připojen.
- org_Call.cpp obsahuje popis čtyř nativních metod, které jsou volány z java-projektu a metodu ResolveEventHeader, která načítá všechny potřebné knihovny.

C# knihovna mimo jiné obsahuje pět hlavních tříd, které obsahují popis struktury proteinu, tunelu a mutací proteinu spolu s metodami pro výpočet tunelů a mutací proteinu. Dále obsahuje třídu StaticWrapper.cs, která obsahuje čtyři metody, které jsou volány z C++ knihovny.



Obrázek 3.2.1 Model vypracování

3.3 Popis GUI

GUI bylo vytvořeno v Java NetBeans Platform. Obsahuje nabídku možností pro chemiky a bloky pro zobrazení výsledků. Jako první si chemik vybere protein, který chce načíst. Proteiny lze získat z databáze proteinů, kterou naleznete zde [7]. Načtený protein se zobrazí jako primární struktura proteinu, použity jsou jednopísmenné zkratky (po načtení tunelu se aminokyseliny, obklopující tunel, zvýrazní). Poté je nutno zadat souřadnice aktivního místa, souřadnice se zadávají formou tří reálných čísel, které přesně definují jeho polohu. Následně bude vypočítán tunel do zadaného aktivního místa.

Po načtení tunelu si může chemik prohlížet tunel, který je popsán a zobrazen pomocí jednotlivých koulí tunelu a aminokyseliny, které tento tunel obklopují. Tyto informace jsou zobrazeny v tabulkách. Ačkoli se zdají být tabulky nepřehledné, pro chemiky jsou číselné údaje mnohem přehlednější a účelnější než složité grafy. Pro zápis jednotlivých aminokyselin, ať v tabulce s koulemi tunelu nebo v tabulce s aminokyselinami, které tunel obklopují, je použita zkratka. Tato zkratka se skládá z číselného indexu aminokyseliny a jednopísmenné zkratky (viz. tabulka 2.1).

První tabulka obsahuje aminokyseliny, které obklopují tunel. Každá aminokyselina je v jedné řádce tabulky a obsahuje počet a poloměr koulí tunelu, kterých se daná aminokyselina dotýká. Druhá tabulka popisuje jednotlivé koule tunelu. Opět je každá koule na jedné řádce tabulky. Každá řádka obsahuje poloměr dané koule a jména (zkratky) aminokyselin, které kouli obklopují. Chemik může koule tunelu řadit podle jejich poloměrů nebo jejich pořadí v tunelu. Díky tomu snadno zjistí, kde je tunel nejužší a kde nejširší, což dává chemikovi možnost snadno se zaměřit na kritickou část tunelu. Poté se může podívat jaké aminokyselin obklopují tuto část tunelu (kouli tunelu) a které aminokyseliny obklopují tunel nejvíce.. Díky tomu může chemik rozhodnout, které z aminokyselin mají významný vliv na vzhled tunelu.

Aminokyseliny, které si chemik myslí, že je rozumné nahradit, může pohodlně zvolit v další tabulce. Pro každou může vybrat seznam aminokyselin, za které se mají nahradit. Tyto aminokyseliny budeme dále nazývat kandidáty. Seznam obsahuje 20 základních aminokyselin. (seznam aminokyselin nalezneme v tabulce 2.1). Dále je možnost pro každého kandidáta ze seznamu zvolit tzv. index pravděpodobnosti. V této tabulce je pro seznam aminokyselin použita jednopísmenná zkratka. Program pro kombinaci kandidáta vypočte novu mutaci proteinu a nový tunel v daném proteinu.

Nové mutace i tunely lze libovolně prohlížet. Pro výpočet tunelu má chemik možnost volit mezi rychlým nebo pomalejším výpočtem. Rozdíl v těchto výpočtech je důvěryhodnost výsledku. K tomu může chemik zvolit, zda by chtěl, při záměně některých aminokyselin, daný tunel rozšířit či zúžit nebo jen provést záměnu. Nově vypočtené tunely si může chemik libovolně prohlížet. Všechny vypočítané mutace může chemik uložit pro pozdější zkoumání.

3.4 Běh a popis aplikace

Tato kapitola popisuje jednotlivé části programu a obecný běh programu. Popisuje hlavně způsob použití C++ můstku, způsob volání a funkci čtyř hlavních nativních metod, které zajišťují přenášení dat mezi java-programem a C# knihovnou Mutations.dll.

3.4.1 Popis programu

Program se spouští jako Java NetBeans Platform formou GUI. Program bude později začleněn do programu Caver. Program využívá C++ nativní knihovnu Library.dll a C# knihovnu Mutations.dll. Knihovna Mutations.dll obstarává veškeré výpočty, které následně GUI zobrazuje. Jako můstek pro předávání dat slouží Library.dll, tato knihovna obstarává veškerý přenos dat mezi Mutations.dll a GUI. Můstek je volám ze zvláštní třídy (Call.java), která obsahuje nativní metody pro volání Library.dll. Metody se spouštějí přes určená tlačítka v GUI. K dispozici jsou čtyři metody:

- Načtení proteinu – LoadProtein
- Vypočítání tunelu – Compute Tunnel
- Úprava tunelu a vytvoření „mutace proteinu“ - Mutate
- Uložení dané „mutace proteinu“ - SaveMutantProteinToPDB

Tyto čtyři metody obstarají všechny požadované funkce. Veškeré vstupy funkcí jsou volitelné přímo v GUI. Výstupy funkcí jsou zobrazeny do bloků v GUI, většinou pomocí tabulek.

3.4.2 Použití metod

GUI získává data ze C# knihovny skrze C++ nativní knihovnu. Název C# knihovny je Mutations.dll. GUI má k dispozici čtyři metody, jejich popis a funkce jsou níže. Při každém volaní jedné z metod je volaná C++ nativní knihovna. Nativní knihovna využívá C# knihovnu, která obsahuje odpovídající metody. C++ knihovna předává strukturu mezi C# a Javou. Každá metoda předává (mimo jiné) jeden argument o tom, zda operace proběhla bez chyby či nikoliv.

3.4.3 Metoda Load Protein

Ve chvíli spuštění je program zcela bez dat. Proto je třeba první načíst požadovaný protein zadáním úplné cesty k souboru proteinu. Po potvrzení cesty k proteinu se zavolá metoda LoadProtein. Metoda vrátí popis struktury proteinu. Celá tato struktura se v GUI zobrazí jako jeden dlouhý pás aminokyselin. V našem případě stačí znát 20 základních aminokyselin.

3.4.4 Metoda Compute Channel

Ovšem GUI slouží hlavně pro práci s tunely, proto je ještě třeba zadat parametry pro hledaný tunel, který chceme vypočítat. Zadání je realizováno pomocí souřadnic x, y, z, jež jsou reálná čísla. Tyto souřadnice udávají polohu aktivního místa v proteinu. Metoda nalezne tunel z povrchu proteinu do tohoto místa. Při potvrzení zadaných souřadnic se zavolá metoda Compute Channel. Metoda vrací nalezený tunel (který je reprezentován množinou koulí) a dále informaci o tom, které aminokyseliny tento tunel obklopují. To vše je zobrazeno pomocí několika tabulek. První tabulka obsahuje aminokyseliny, které obklopují tunel. O každé aminokyselině víme, kolik atomů aminokyseliny opravdu obklopuje tunel. Druhá tabulka popisuje jednotlivé koule tunelu. U každé koule je uveden její poloměr, její pořadí v tunelu a aminokyseliny, které tuto kouli obklopují. Dále je k dispozici tabulka s detaily jednotlivých aminokyselin.

3.4.5 Metoda Mutate

Jak už jsem zmínil v úvodu, málokdy se stane, že vypočtený tunel je ideální. Pro možnosti změny parametrů tunelu slouží dvě nabídky. První z nich umožňuje zadání zvolených kandidátů. Druhá nabídka slouží k zadání typu akce (zúžení, rozšíření, pouze záměna aminokyselin) a k módu mutace (rychlý přibližný výpočet, pomalý, přesnější výpočet s reálnějšími výsledky). Po zadání se zavolá metoda Mutate. Metoda vrací

množinu různých "mutací" proteinu (Mutací proteinu chápeme protein, který vznikl z původního proteinu záměnou některých aminokyselin.) spolu s popisem tunelů v jednotlivých mutacích. Biochemik pak může vybrat mutaci, která je z jeho pohledu nejlepší.

3.4.6 Metoda SaveMutantProteinToPDB

Tato metoda slouží pro uložení zvolených mutací proteinu a nemá žádnou návratovou hodnotu.

3.4.7 Nativní knihovna sloužící jako můstek

Jméno knihovny je Library.dll. Knihovna slouží jako datový můstek mezi C# knihovnou a GUI napsaným v jazyku Java. Knihovna obsahuje čtyři metody, které přistupují k objektům v Javě nebo k Mutations.dll knihovně pomocí JNI metod. V C++ kódu dokážeme za pomoci JNI metod předávat stejné datové struktury jako jsou v Javě nebo v C#. Díky tomu můžeme předávat data ze C# do Javy a zpět. Při předávání dat ze C# do Javy, vytvoří C++ knihovna vytvoří datovou strukturu shodnou se strukturou v Javě, kterou chceme naplnit (předat data). Knihovna pak jednoduše přistupuje k objektům v Mutations.dll a jimi plní vytvořenou Java datovou strukturu. Naplněnou datovou strukturu pak předá Javě pomocí JNI metod, například návratovou hodnotou metody. Při předávání dat opačným směrem je postup analogický.

3.4.8 Knihovna pro výpočet tunelu a mutací

V knihovně Mutations.dll probíhá načtení proteinu a výpočty tunelů a mutací proteinu. Knihovna je načítána v C++. Datové struktury, které jsou používány v GUI, byly vytvořeny na základě struktury, implementovaných v knihovně Mutations.dll. V java-programu jsou implementovány pouze struktury, které jsou potřebné pro operace v GUI. C# knihovna Mutations.dll je používá pomocí čtyř výše zmíněných metod.

3.5 Problémy s implementací

Jelikož jsem se s JNI dosud nikdy nesetkal a jelikož není mnoho zdrojů, kde lze čerpat spolehlivé informace, tak jsem se v průběhu vytváření nativní knihovny setkal s několika problémy:

3.5.1 Používání knihoven

Pro správné využívání nativní knihovny je nutné tuto knihovnu načít do java projektu. Nativní knihovna se načte za pomocí jednoduchého příkazu: `Systém.load(úplná cesta ke knihovně);`

Nutno dodat, že při každé změně nativní knihovny je nutno knihovnu opětovně přeložit, aby java-program používal správnou verzi knihovny. Pro správnou funkci nativní knihovny je třeba, aby nativní metoda implementovala událost AppDomain.AssemblyResolve. V této události je nutné konkrétně načíst knihovny, které se programu nepodařilo automaticky načíst. Bližší popis této události nalezneme zde [10]. Knihovny, které chceme načítat, musí být ve stejném adresáři jako je nativní knihovna, vzniklá překladem tohoto projektu. Dále musí obsahovat statický konstruktor, v našem případě: MyInitialization. Tento konstruktor obsahuje delegáta, který obsluhuje událost AppDomain.AssemblyResolve. Tento konstruktor musí byt až za metodami, které jsou v tomto konstruktoru volány.

C# knihovna Mutations.dll vyžaduje pro svůj běh adresář "data", který obsahuje popis aminokyselin. Dále C# knihovna Mutations.dll využívá některé složky DirectX (což je sada knihoven umožňující JAPI přímé ovládání hardwaru) a mohou nastat problémy při jejich používáni v jiném vývojovém prostředí než je VisualStudio 08. Při implementaci jsem raději přešel z vývojového prostředí VisualStudio verze 10 na verzi 08, jelikož problémy s využíváním DirectX zde nebyly. Dalším důvodem pro přechod na verzi 08 byla absence IntelliSense ve verzi 10 pro C++/CLI projekty. IntelliSense je jakýsi našeptávač, který poskytuje rychlý a pohodlný přístup k popisu funkcí (formou seznamu) a také poskytuje seznamy parametrů pro tyto funkce.

3.5.2 Debugování knihovny

Kompilátory, které sestavují program v Javě a v C++/CLI hlídají pouze syntaktické chyby. Kompilátor tedy nezajistí, že daný program funguje jak má.

Při vytváření programu jsem často narazil na proměnou, která měla jinou hodnotu, než jsem požadoval, nebo jsem přiřazoval jiné konstruktory jiným metodám atd. Tyto chyby jsou jen těžko odhalitelné. Pro nalezení těchto chyb bylo tedy nutné použít debugování. Jinými slovy krokování programu postupně po jednotlivých příkazech.

Debugování jsem používal jak ve vývojovém prostředí NetBeans, tak ve Visual Studiu. Debugování v NetBeans není nijak složité, není potřeba dělat žádné zvláštní nastavení. Je ovšem důležité zkontrolovat, že aplikace využívá aktuální verzi knihovny. Debugování ve Visual studiu vyžaduje jisté nastavení. Pokud chceme debugovat knihovnu, je nutné nastavit cestu k aplikaci, kterou naše knihovna používá. Ve vlastnostech C++/CLI projektu ve složce Debugging je nutné vyplnit položky: Command a Command Arguments. Při prvním pokusu o debugování nativní knihovny, jsem tyto pole vyplnil pro projekt Java NetBeans Platform. Standardně se při kompilaci java-projektu vytvoří jar soubor, který se poté spouští. Níže vidíme nastavení položek pro debugování ve Visual Studiu pro jar soubor.

Command: C:\Program Files\Java\jdk1.6.0_17\bin\java.exe

Command Arguments: -jar "C:\Documents and Settings\....\dist\Main.jar"

Toto nastavení se ukázalo být v tomto případě nefunkční. Poté následovalo ještě několik dalších pokusů s jiným nastavení, ale ty vždy vedly k pádu debugování nebo chybné funkci debugování. Proto jsem byl nucen vytvořit v NetBeans standardní projekt: Java Application. Pro tento projekt bylo nastavení plně funkční a bez problémů.

4 Závěr

Proteiny jsou přírodní látky a jsou podstatou všech živých organismů. Proteiny mohou zastávat různé funkce. Drobná změna struktury může vést ke změně vlastností a funkcí. Tyto změny vlastností využívají chemici k nalezení proteinu s lepšími vlastnostmi. Každý nástroj, který pomáhá chemikům ulehčovat a prohlubovat znalosti v poznávání struktury a vlastnosti proteinu, je velice užitečný. Tato bakalářská práce přináší více možností pro program Caver, čímž pomáhá chemikům v jejich výzkumné práci.

Program Caver již umí vypočítat tunel v proteinu a zobrazit jej v 3D vizualizaci. Ovšem postrádá prostředky pro záměnu aminokyselin, které tunel obklopují. Tyto záměny mohou mít značný vliv na vzhled tunelu. Tunel se těmito změnami může rozšířit, nebo zúžit, což může vést k lepšímu přístupu do aktivního místa.

Mojí prací bylo vytvořit GUI, které se zabuduje jako zásuvný panel do programu Caver. GUI jsem vytvořil v Java NeatBeans Platform, což bylo požadavkem při vypracování. S tím souviselo vytvoření můstku mezi tímto GUI a C# knihovnou, která má na starosti veškeré výpočtové operace. Můstek je implementován v C++/CLI a je vytvořen jako nativní knihovna pomocí JNI.

Toto GUI umožňuje načíst vybraný protein a číselně zobrazit koule tunelu a aminokyseliny, které tento tunel obklopují. Hlavní přínos tohoto programu je však možnost vybrat pro každou aminokyselinu, která obklopuje tunel, výčet kandidátů, za které se daná aminokyselina nahradí. Což znamená, že pro každou aminokyselinu obklopující tunel můžeme vybrat náhradníky z 20 základních aminokyselin. Po vybraní těchto kandidátu aplikace vypočítá pro každou změněnou aminokyselinu novou mutaci proteinu a nově vzniklý tunel v každé této mutaci. Tyto mutace lze libovolně prohlížet. Je zde také možnost uložit zvolené mutace. Tyto možnosti dávají chemikům velice užitečný nástroj pro práci s tunely.

Přehled pojmů a zkratek

JNI	Java Native Interface
GUI	Graphical User Interface
JVM	Java Virtual Machine
JAPI	Java Application Programing Interface
JDK	Java Development Kit
3D	Trojrozměrný prostor
JLS	JavaTM Language Specification
cpp	C plus plus
CLI	Common Language Infrastructure
pdb	formát souboru Palm DOC
XML	eXtensible Markup Language
dll	Dynamic Link Library

Reference

[1] P. Medek , P. Beneš a J. Sochor. Computation of tunnels in protein molecules using Delaunay triangulation. Journal of WSCG 15, 1-3, 107-114. 2007

[2] M. Petřek, M. Otyepka, P. Benáš, P. košinová, J. Koča a J. Damborský. CAVER: a new tool to explore routes from protein clefts, pockets and cavities. BMC Bioinformatics 7, 316-324. 2006

[3] Sheng Liang. Java(TM) Native Interface: Programmer's Guide and Specification. Addison-Wesley Profesional. 1999

[4] Jackwind Li Fuojie. Professional Java Native Interfaces with SWT/JFace. Wrox. 2005

[5] Computer Graphics Group, Human Computer Interaction Laboratory & Protein Engineering Group, Loschmidt Laboratories. Masaryk University. CAVER. [online]. URL:<http://loschmidt.chemi.muni.cz/caver/>

[6] Department of Biology. Davidson College.Davidson. List of Amino Acids and Their Abbreviations [online]. [cit. 3.4.2010]. URL:<http://www.bio.davidson.edu/Biology/aatable.html >

[7] Research Collaboratory for Structural Bioinformatics. RCSB PDB Protein Data Bank. [online]. [cit. 3.4.2010]. URL:<http://www.pdb.org>

[8] Brůha Lubomír. Java Hotová řešení. Computer Press. 2003

[9] Adam Myatt. Pro NetBeans 6 Rich Client Platform Edit. Apress. 2008

[10] Microsoft Corporation. .NET Framework Class Library. [online]. [cit. 7.4.2010]. URL:<http://msdn.microsoft.com/en-us/library/ /system.appdomain.assemblyresolve(v=VS.71).aspx>

Přílohy

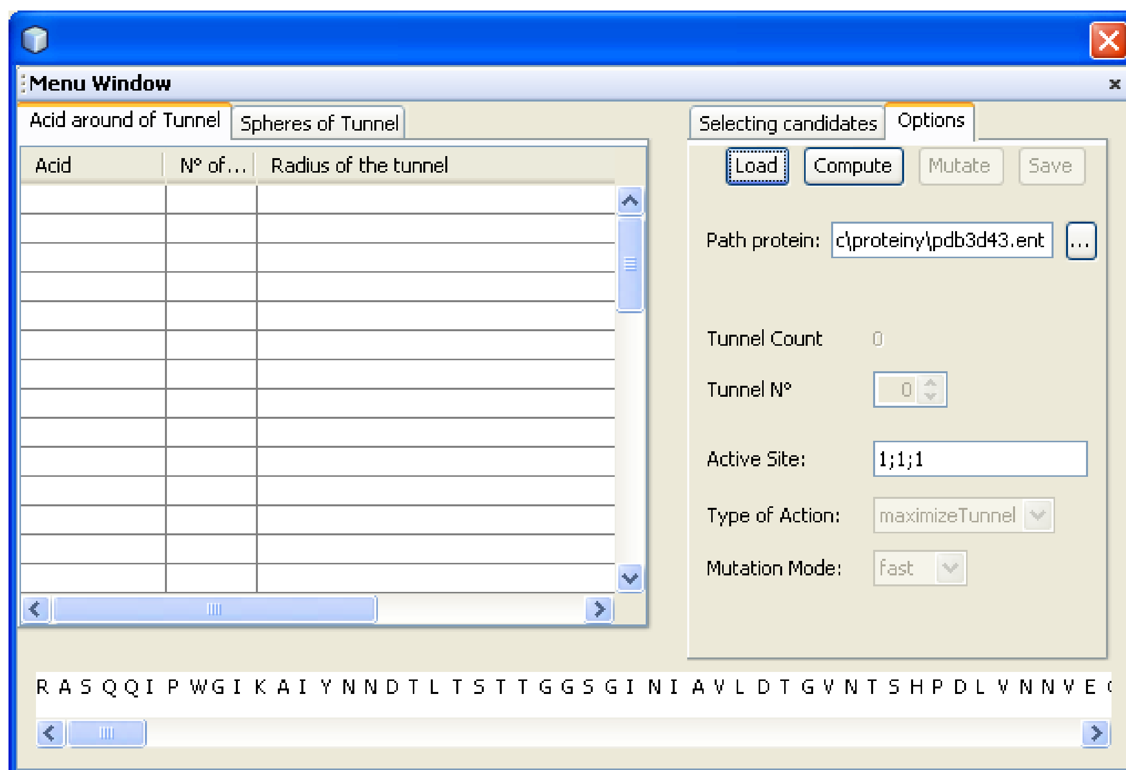
Uživatelský manuál

Hlavním účelem aplikace je zobrazit vybraný protein, informace o vypočítaném tunelu, vybrané mutace proteinu a uložit zvolené mutace. Na obrázku 4.1 vidíme GUI pro tyto operace. GUI obsahuje tři hlavní panely. První panel nalezneme v levé části, kde jsou umístěny tabulky s aminokyselinami,které obklopují tunel (Acid around of tunnel) a koulemi tunelu (Spheres of Tunnel). Druhý panel nalezneme v pravé části, který obsahuje seznam kandidátů, které chceme zaměnit (Selecting candidates) a nabídku s operacemi, které můžeme provádět (Options). Polední panel nalezneme v dolní části GUI. Panel slouží pro zobrazení primární struktury proteinu.

Obrázek 4.1 znázorňuje GUI při jeho prvním otevření. V tuto chvíli máme jedinou možnost a to načtení proteinu. Ostatní operace není možné provádět bez načteného proteinu. Daný protein se volí v poli Path protein, kde je nutno vybrat cestu k pdb souboru proteinu. Načtení proteinu se provede přes tlačítko Load.

Obrázek 4.1 GUI pří prvním otevření

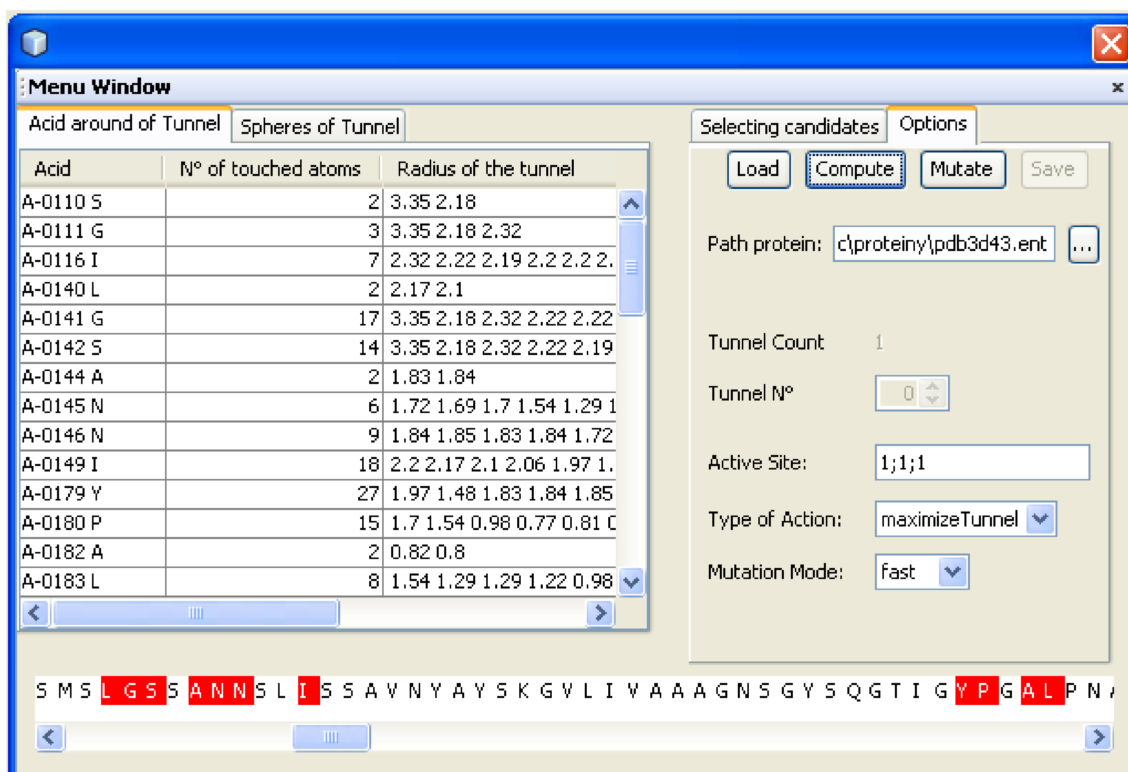
Na obrázku 4.2 můžeme vidět, co se stane po načtení proteinu. V třetím panelu se zobrazila primární struktura proteinu (formou jednopísmenných zkratek) a zpřístupnila se nabídka pro vypočítání tunelu. Souřadnice aktivního místa můžeme vyplnit v poli Active Site. Jak je vidět na příkladu, souřadnice se zadávají jako tři reálná čísla, oddělená středníkem. Výpočet tunelu proběhne po potvrzení tlačítka Compute.



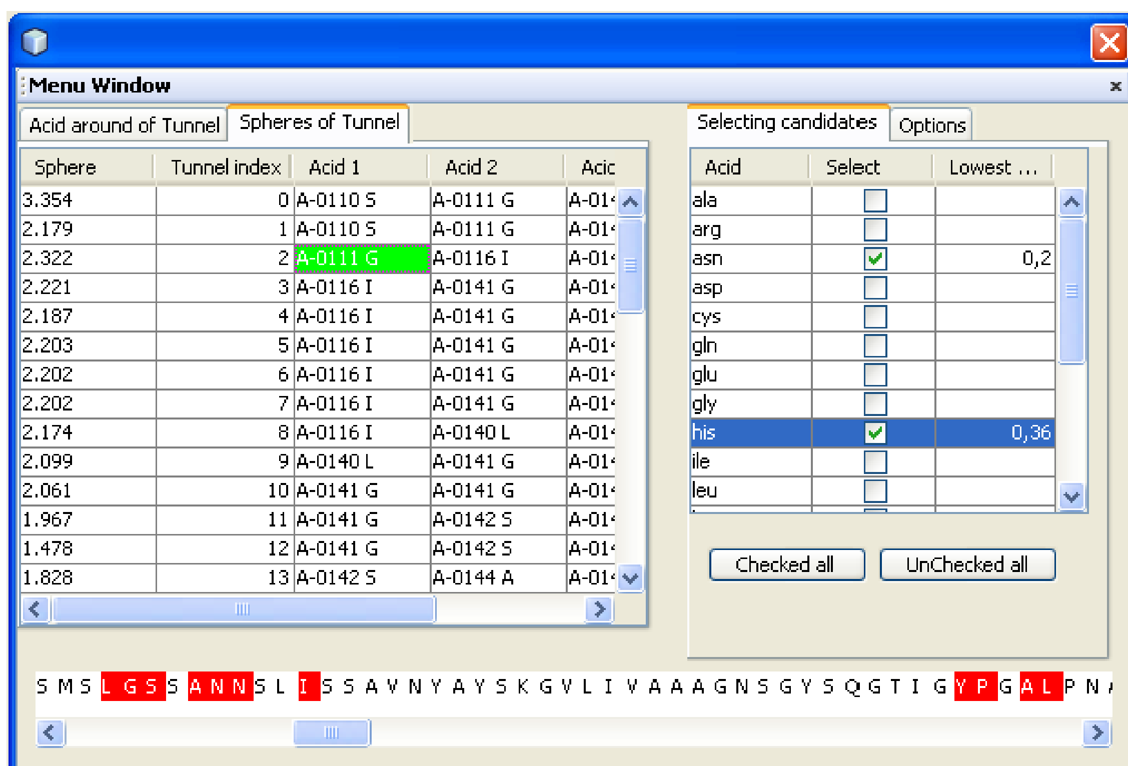
Obrázek 4.2 GUI po načtení proteinu

Obrázky 4.3 a 4.4 znázorňují GUI po vypočítání tunelu. Na obrázku 4.3 vidíme v prvním panelu tabulku s aminokyselinami, které obklopují tunel. V tabulce jsou informace o názvu aminokyseliny, počtu koulí tunelu, se kterými se dotýká a poloměry těchto koulí tunelu. Dále můžeme vidět v druhém panelu, že máme možnost mutovat protein (tedy vybrat kandidáty pro určité aminokyseliny). V nabídce máme nyní možnost zvolit typ akce (rozšíření, zúžení nebo pouze záměna některých aminokyselin). V třetím panelu se červenou barvou zvýraznily aminokyseliny, které obklopují tunel.

Na obrázku 4.4 vidíme v prvním panelu tabulku koulí tunelu. V každé řádce je uvedena jedna koule tunelu. O každé kouli víme její poloměr, pořadí v tunelu a aminokyseliny, kterých se dotýká. V druhém panelu vidíme tabulku pro zvolení kandidáta pro každou aminokyselinu. Pro ulehčení lze označit (nebo odznačit) všechny najednou. Mutaci provedeme tlačítkem Mutate.

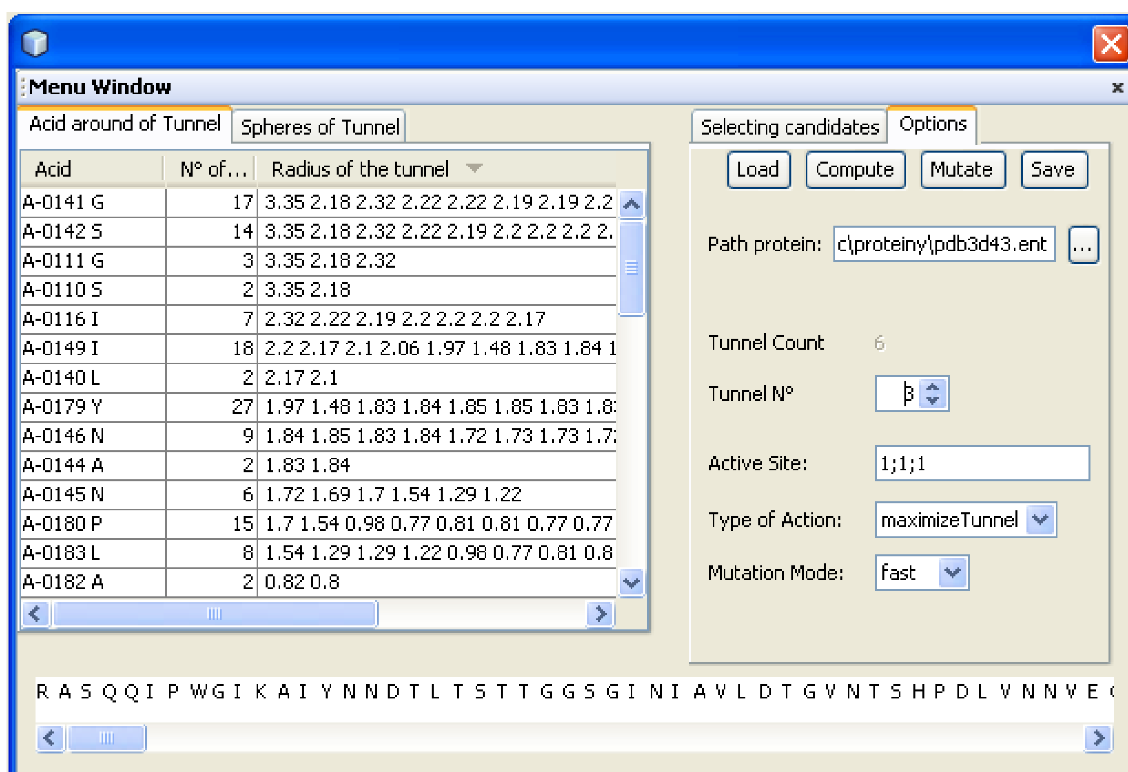


Obrázek 4.3 GUI po vypočítání tunelu, první zobrazení panelů



Obrázek 4.4 GUI po vypočítání tunelu, druhé zobrazení panelů

Na obrázku 4.5 můžeme vidět GUI po vypočítání všech mutací proteinu. Opět můžeme v prvním panelu prohlížet aminokyseliny, které obklopují tunel a koule tunelu jednotlivých mutací. Mezi těmito mutacemi přepínáme v druhém panelu v záložce Options. Zde vidíme počet nových tunelů: Tunnel Count a také pole Tunnel N°, které přepíná mezi jednotlivými mutacemi. Dále se nám zpřístupnila možnost danou mutaci uložit tlačítkem Save. Při potvrzení se uloží mutace, která je zobrazena. Výše zmíněný postup lze opakovat stále dokola.



Obrázek 4.5 GUI po vypočítaní všech mutací