УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Бојана Јошић

РАЗВОЈ АПЛИКАЦИЈЕ ЗА АНАЛИЗУ И ВИЗУЕЛИЗАЦИЈУ ПОДАТАКА О ПРОТЕИН-ПРОТЕИН ИНТЕРАКЦИЈАМА

мастер рад

Београд, 2023.

Ментор:

др Јована Ковачевић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Мирјана Маљковић, доцент
Универзитет у Београду, Математички факултет

др Владимир Перовић, виши научни сарадник
Универзитет у Београду, ИНН Винча

Датум одбране: _____

Породици

Наслов мастер рада: Развој апликације за анализу и визуелизацију података о протеин-протеин интеракцијама

Резиме: У овом раду је описан процес интеграције података о протеин-протеин интеракцијама из више база у једну графовску базу хуманих протен-протеин интеракција. Затим је описан систем за управљање графовским базама *Neo4j*, као и упитни језик за ову базу - *Cypher*. Даље је креирана клијент-сервер апликација за визуелизацију протеин-протеин интеракција. Серверска апликација се заснива на *Graphql* технологији и имплементирана је употребом *Spring* радног оквира. Клијентска апликација је креирана употребом *Angular* радног оквира за развој једностраничних апликација. У раду је додатно креирана релациона база у којој се складиште исти подаци, након чега је анализирана ефикасност извршавања *SQL* и *Cypher* упита, односно ефикасност складиштења протеин-протеин интеракција у релационој, те графовској бази.

Кључне речи: протеин-протеин интеракције, *Neo4j*, *GraphQL*, *Cypher*, *Spring*, *Angular*

Садржај

1 Увод **1**

2 Базе протеин-протеин интеракција **3**
2.1 Карактеристике неких јавних база 3
2.2 Подаци о протеин-протеин интеракцијама 4

3 Графовске базе и систем *Neo4j* **8**
3.1 Графовске базе података 8
3.2 Основни концепти *Neo4j* базе 9
3.3 *Neo4j* платформа 11
3.4 Упитни језик *Cypher* 12

4 *Spring* радни оквир **17**
4.1 Управљање пројектом помоћу *Maven* алата 17
4.2 *Spring* радни оквир 19
4.3 *GraphQL* 23
4.4 Мере чворова графа 24

5 Развој апликације **26**
5.1 Преузимање и интеграција података 26
5.2 Развој серверске апликације 31
5.3 Развој клијентске апликације 34

6 Закључак **39**

Библиографија **41**

Глава 1

Увод

Протеини представљају најбитније молекуле за функционисање једног организма. Више од 80% протеина у организму не остварује своју биолошку улогу самостално, већ ступањем у интеракције са другим молекулима [16]. Када два или више протеина ступају у интеракцију, реч је о **протеин-протеин интеракцијама**. Оне су заслужне за одвијање неких од најважнијих ћелијских процеса, као што су ћелијска деоба, метаболички процеси, итд. Анализа интеракција између два протеина је, дакле, кључна за разумевање функционисања једног организма и од великог је значаја у медицини и фармацији.

Подаци о протеин-протеин интеракцијама се чувају у различитим базама од којих су најзначајније ***BioGRID*** [13], ***STRING*** [20], ***IntAct*** [5] и друге. Постоји потреба за интеграцијом података у једну базу, да би се добила унификована репрезентација свих података о протеин-протеин интеракцијама. Међутим, интеграција ових података није тривијалан задатак, јер различите базе користе различите генске или протеинске идентификаторе за означавање протеина, затим користе различите методе за одређивање скорова интеракција (на основу којих се утврђује колико је вероватна та интеракција), интеракције су детектоване различитим експерименталним методама или методама предикције, па две базе могу да складиште различите податке о потпуно истој интеракцији.

У овом раду интегрисани су подаци о хуманим протеин-протеин интеракцијама из база *BioGRID*, *STRING*, *IntAct*, ***Reactome*** [6] и ***HIPPIE*** [17] у једну графовску базу података. За управљање графовском базом су коришћени систем ***Neo4j*** [12] и упитни језик ***Cypher*** [12]. Додатно је креирана

релациона база над истим скупом података и на том примеру је показано да је графовска база ефикаснија за складиштење података о протеин-протеин интеракцијама. Након графовске базе креирана је клијент-сервер апликација за визуелизацију ускладиштених протеин-протеин интеракција. Сервер се повезује на графовску базу и податке прослеђује клијенту, по захтеву. Серверска апликација се заснива на ***GraphQL*** технологији и имплементирана је употребом ***Spring*** радног оквира. Клијентска апликација је креирана у ***Angular*** [1] радном оквиру. Апликација осим визуелизације података о протеин-протеин интеракцијама додатно рачуна неке мере над чворовима: степен чвора, *page rank* и разне варијанте централности.

Глава 2

Базе протеин-протеин интеракција

2.1 Карактеристике неких јавних база

Јавно доступне базе протеин-протеин интеракција се међусобно разликују, како по моделу, тако и по броју интеракција које складиште, по начину прикупљања података и креирања базе, али и по природи самих података. Тачније, у неким базама складиште се искључиво интеракције које су експериментално потврђене, у другим оне које су добијене рачунарским методама предикције, а постоје базе које складиште оба типа.

Према начину прикупљања података базе се класификују као **примарне** или **секундарне** [3]. У примарним базама подаци се издвајају из објављених научних радова у којима су детаљно описани изведени експерименти, као и интеракције које су притом детектоване. За прикупљање података и њихово складиштење у базу задужени су кустоси база, који анализирањем радова извлаче корисне информације о протеин-протеин интеракцијама, као што су: протеини који интерагују, метода којом је интеракција детектована, организми из којих су протеини издвојени, тип интеракције, итд. За разлику од примарних база података, секундарне базе се креирају и допуњују интеграцијом података из других база протеин-протеин интеракција (примарних или секундарних).

Још једна карактеристика неких база јесте везаност за специфични организам. Уколико је база везана за организам, то значи да се у њој складиште искључиво оне интеракције које се детектују у том организму. Пример овакве

базе је *HIPPIE* у којој се складиште само оне интеракције које су из људског организма.

У табели 2.1 су наведене карактеристике неких јавних база протеин-протеин интеракција [3].

Табела 2.1: Карактеристике неких база протеин-протеин интеракција

Назив	Примарна/ Секундарна	Бр. интеракција	Организми	Тип података (експериментални / предиктивни)
IntAct	примарна	1.8 милиона	више (>10)	експериментални
BioGRID	примарна	2.6 милиона	више (>10)	експериментални
STRING	секундарна	20 милијарди	више (>10)	експериментални и предиктивни
HPRD	примарна	40 хиљада	човек	експериментални
HIPPIE	примарна и секундарна	700 хиљада	човек	експериментални
DIP	секундарна	80 хиљада	више (>10)	експериментални
GPS-Prot	секундарна	300 хиљада	човек & ХИВ	експериментални
MINT	примарна	100 хиљада	више (>10)	експериментални

2.2 Подаци о протеин-протеин интеракцијама

У базама се осим **идентификатора интерактора** (протеина који ступају у интеракцију) најчешће чувају и додатне информације о једној интеракцији као што су: **методи детекције интеракције** (неки од многих експерименталних метода или метода предикције), **таксономски идентификатори организама интерактора** (нпр. таксономски идентификатор човека је 9606), **типови интеракције**, **скор**, итд.

Идентификатори протеина

Протеини се складиште у базама протеина као што су ***UniProt*** [4], ***Protein Data Bank*** [24] и друге. Ипак, како гени записују кодове о генерисању протеина, на одређени протеин може да се реферише и преко гена који га кодира. Гени се складиште у базама гена попут ***Entrez Gene*** базе [9]. Протеинске и генске базе дефинишу сопствене идентификаторе за податке које чувају. Један ген може да утиче на стварање више од једног протеина (алтернативно сплајсовање), па један генски идентификатор може да се односи на више протеина. Иако је ово случај, протеини се неретко идентификују генским идентификаторима у базама протеин-протеин интеракција. На пример, у бази *BioGRID* интерактори се примарно записују помоћу *Entrez Gene* генског идентификатора. У овој бази се записују и информације о разним алтернативним генским и протеинским идентификаторима интерактора, па су протеини једнозначно одређени. У базама *IntAct*, *HIPPIE* и *Reactome* примарно се користи *UniProt* идентификатор (уз мноштво алтернативних идентификатора), а у бази *STRING* се користи протеински ***Ensembl Protein*** [25] идентификатор. У табели 2.2 је дат пример једног протеина и неких његових идентификатора:

Табела 2.2: Неки идентификатори протеина *Clarin-2* из људског организма

UniProt ID	*Ensembl Gene ID*	*Gene ID*	*Ensembl Protein ID*
A0PK11	ENSG00000249581.2	645104	ENSP00000424711.2

Скор протеин-протеин интеракције

Одређени методи детекције интеракција нису сасвим поуздани (што се нарочито односи на предиктивне методе), па се свакој протеин-протеин интеракцији додељује **скор**, који представља меру на основу које се утврђује колико је детектована протеин-протеин интеракција заиста вероватна. Постоје различити алгоритми за генерисање скорова. Један од њих је ***MiScore*** [22], који ће у наставку бити детаљно описан. Различите базе уобичајено користе различите алгоритме и упоређивање скорова исте интеракције записане у више таквих база није могуће. Примера ради, *HIPPIE* и *STRING* дефинишу сопствене алгоритме за генерисање скорова, док *IntAct* база користи наведени *MiScore* алгоритам.

MiScore

MiScore при генерисању скора интеракције разматра следеће:

– Како је интеракција откривена (експериментални или предиктивни методи детекције).

– Ког типа је интеракција (за једну интеракцију може бити утврђено више типова).

– Број научних радова у којима је интеракција описана.

Што више ових информација о интеракцији постоји, скор ће очекивано бити већи. При генерисању скора се разматрају и други фактори - нпр. експериментални методи ће коначном скору допринети више него предиктивни. *MiScore* алгоритам подразумевано генерише нормализован скор (S_{MI}) на основу следеће формуле:

$$S_{MI} = \frac{K_p \cdot S_p(n) + K_m \cdot S_m(cv) + K_t \cdot S_t(cv)}{K_p + K_m + K_t},$$

где су K_p, K_m и K_t произвољни **тежински фактори** из интервала $[0, 1]$, а S_p, S_m и S_t **скор публикација**, **скор метода** и **скор типова**, редом. Подешавањем вредности тежинских фактора се регулише колико сваки од скора публикација, метода и типова доприноси резултујућем скору. У наставку ће бити описано како се појединачно генерише сваки од ових скорова.

Скор публикација се рачуна помоћу формуле:

$$S_p = log_{b+1}(n + 1),$$

где је b број научних радова за који би се генерисао максимални скор публикација (подразумевано је $b = 7$), а n је број научних радова у којима је заправо описана интеракција.

Скор метода се рачуна помоћу формуле:

$$S_m(cv_i) = log_{b+1}(a + 1)$$

$$a = \Sigma(scv_i \cdot n_i)$$

$$b = a + \Sigma(Max(Gscv_i)),$$

при чему scv_i представља вредност из интервала $[0, 1]$ која се додељује методу детекције cv_i (ову вредност за сваки метод дефинише *MI ontology*) [8], n_i представља колико је пута cv_i наведен као метод детекције интеракције, а $Gscv_i$ је скуп вредности метода детекције, које припадају истој категорији (имају заједничког родитеља у *MI ontology* стаблу).

Скор типова се рачуна помоћу формуле:

$$S_t(cv_i) = log_{b+1}(a + 1)$$

$$a = \Sigma(scv_i \cdot n_i)$$

$$b = a + \Sigma(Max(Gscv_i)),$$

а у овом случају се $cv, scv, Gscv$ односе на типове интеракције и њихове вредности.

Глава 3

Графовске базе и систем *Neo4j*

3.1 Графовске базе података

Релационе базе података су дуго времена најкоришћеније базе података. У њима се могу складиштити било који подаци, али ипак - не сви ефикасно. У релационим базама се подаци моделирају помоћу табела и колона, што није најпогоднија репрезентација за све податке, на пример за оне који нису структуирани, али ни за оне који су структуирани хијерархијски или графовски.

Из тог разлога се за „незгодне" податке све чешће употребљавају ***NoSQL*** базе које, као што сам назив наглашава, одступају од класичног релационог модела. **Графовске базе** су један тип *NoSQL* база, у којој се подаци моделирају помоћу компоненти графова - чворова, грана и својстава.

Употреба графовских база је у порасту када је реч о складиштењу и анализи података код којих су везе између ентитета једнако битне као и сами ентитети. На пример, друштвене мреже *Facebook*, *LinkedIn*, итд. се ослањају на графовске базе у анализирању повезаности корисника и одређених садржаја. Неки примери система за управљање графовским базама су: *ArangoDB*, *NebulaGraph*, *RedisGraph*, *Amazon Neptune*, ***Neo4j***,...

Поређење релационих и графовских база

Релационе базе користе унапред дефинисану схему за складиштење података. Схемом се прецизно описују елементи базе - табеле, колоне и везе између табела. Одређена колона или вектор колона једне табеле се дефинише као

примарни кључ. Вредност примарног кључа је јединствена за сваки ред табеле. Такође, веза између две табеле се остварује помоћу *страног кључа*, који у једној табели представља колону (или вектор колона) којом се реферише на примарни кључ друге табеле.

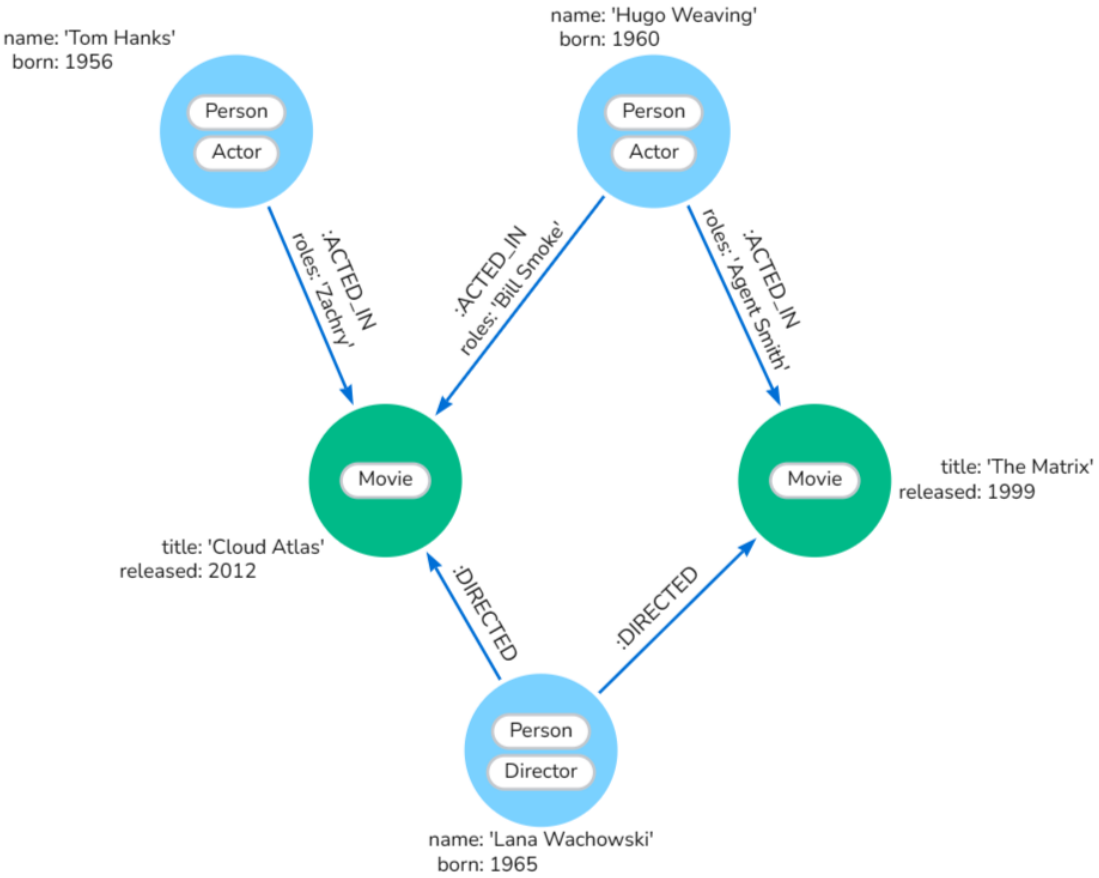
Пре складиштења у релациону базу, податке је најпре потребно прилагодити тако да одговарају дефинисаној схеми. Уколико схема није добро дефинисана или уколико податке није могуће добро прилагодити, складиштење у базу и читање података из базе неће бити једноставно и ефикасно.

Као што је већ напоменуто, графовске базе су бољи избор од релационих за складиштење оних података код којих је у фокусу њихова међусобна повезаност. Та повезаност би се у релационој бази моделовала помоћу страних кључева табела, што отежава управљање базом. Такође, за дохватање оваквих података би било неопходно писање сложених упита за спајање табела (помоћу операције `JOIN`). У графовским базама се ентитети представљају као чворови, а релације између њих као гране између чворова, што је природни модел за те податке. И чворови и гране могу имати својства, којима се додатно описују.

За разне системе за управљање графовским базама постоје разни упитни језици. Један од њих је *Cypher*, који се примарно користи у систему *Neo4j*. Ови језици омогућавају писање једноставних и интуитивних упита за манипулисање и читање података. Ипак, непостојање стандардизованог упитног језика за графовске базе представља недостатак у поређењу са релационим базама које користе *SQL* упитни језик.

3.2 Основни концепти *Neo4j* базе

Neo4j представља графовску базу у којој су подаци до нивоа складиштења представљени као графови. Дакле, граф овде није само апстракција која се гради над другом технологијом (што важи за многе друге графовске базе). Овај дизајн омогућава брз и ефикасан обилазак графа - података. Предност *Neo4j* базе је и у ефикасном хоризонталном скалирању, као и у једноставном модификовању чворова и релација, па је данас *Neo4j* најзаступљенија графовска база података. У наставку ће над базом филмова, која је доступна као пример по преузимању *Neo4j*-а, бити детаљно описане компоненте: чворови, релације и њихова својства (слика 3.1).



Слика 3.1: Графички приказ неких чворова и релација базе филмова [12]

Чворови

Сви ентитети скупа података се моделују чворовима графа. Чворовима се додељују **лабеле**, на основу којих се групишу у различите скупове. Додатно, помоћу лабела се чвору могу доделити информације о ограничењима (енгл. *constraints*) или индексу. У примеру базе филмова постоје следеће лабеле чворова: `Person`, `Movie`, `Director` и `Actor`. Даље, чворови могу имати **својства** у виду *кључ-вредност* структуре, којом се дефинишу атрибути једног ентитета. У примеру базе филмова, својства једног од чворова са лабелом `Movie` су `title: 'The Matrix'` и `released: 1999`. Чворови који имају исту лабелу могу имати различит број својстава. На пример, могуће је да постоји и трећи чвор у бази са лабелом `Movie`, који има својства `title: 'Get Out'`, `released: 2017` и додатно `Oscars: 1`.

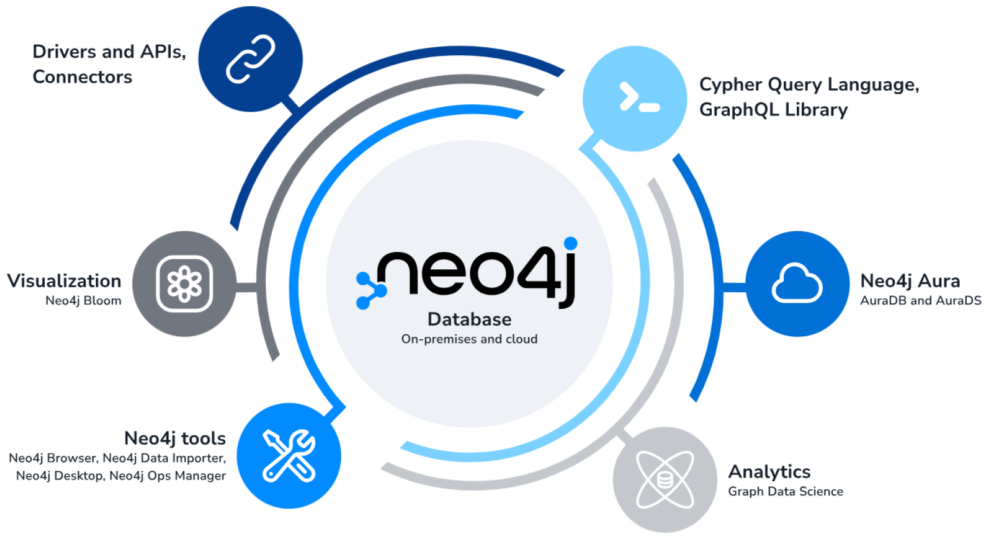
Релације

Релације представљају усмерене гране између чворова. Релација се дефинише почетним и циљним чвором и типом. Као и чворови, релације додатно могу имати својства. Такође, могуће је да једна релација има исти чвор за почетни и циљни. На слици 3.1 су приказане релације чији су типови: `DIRECTED` и `ACTED_IN` (која се карактерише својством са кључем `roles`).

3.3 *Neo4j* платформа

Neo4j, Inc је компанија која је развила *Neo4j* графовску базу. Постоје два издања базе: *Community Edition*, које је могуће бесплатно преузети и *Enterprise Edition*. Базу је могуће користити и у облаку кроз сервис *Neo4j AuraDB*. Поред базе, компанија развија разне апликације и алате који олакшавају рад са графовским подацима (слика 3.2), од којих су неки:

- *Neo4j Browser* - алат за извршавање *Cypher* упита и визуелизацију резултата.
- *Neo4j Desktop* - окружење за рад са *Neo4j* базама и креирање апликација.
- *Neo4j Bloom* - алат за визуелизацију графовских података.



Слика 3.2: *Neo4j* платформа [12]

У овом раду ће се користити *Neo4j Community Edition 5.4.0* и алат *Neo4j Browser*, који се додатно добија при преузимању.

3.4 Упитни језик *Cypher*

Упитни језик *Cypher* инспирисан је *SQL* језиком за релационе базе, тако да се приликом писања упита фокус ставља на податке које је потребно прочитати, а не на описивање начина читања података. Овај језик има врло једноставну и интуитивну синтаксу, која имитира визуелну репрезентацију компоненти. Наиме, чвор се записује као `(node)` (заграде имитирају кругове којима се представљају чворови), а једна релација представља се стрелицама `-->` или `<--` између два чвора. У наставку ће бити детаљно описана синтакса овог упитног језика.

Репрезентација чворова

Чвор се представља као `(node)`, при чему `node` означава лабелу, променљиву или се изоставља. У случају изостављања, запис `()` се односи на било који чвор у бази. Навођењем назива лабеле у заградама, представљају се сви чворови који припадају групи дефинисаној лабелом. Нпр. `(:Actor)` се односи на све чворове који имају лабелу `Actor` (пре назива лабеле се наводе две тачке). У запису `(a)`, `a` је променљива, помоћу које се касније у упиту може референсати на чворове. `(a)` се такође односи на све чворове, а могућа је и синтакса која укључује лабелу: `(a:Actor)`.

Репрезентација релација

Релације се представљају стрелицама `-->` или `<--` између два чвора (стрелицом `-->` уколико је лево записан почетни чвор, а десно циљни, а у обрнутом случају стрелицом `<--`). Остале информације, као што су тип релације или својства, наводе се у угластим заградама између стрелица. Иако релације обавезно имају смер, у упитима је дозвољено користити и синтаксу у којој се он не назначава: `--`. Ово пружа одређени ниво флексибилности, јер се у том случају занемарује смер релације приликом претраге, односно обиласка графа.

Као што је случај са чворовима, релацији је могуће доделити променљиву. Испред типа релације се наводи двотачка, што није случај са идентификаторима променљивих. У наставку су примери неких релација:

- `(actor:Actor)-[:ACTED_IN]->(movie:MOVIE)`
- `(director:Director)-[d:DIRECTED]->(movie:Movie)`
- `()<--(p:Person)`
- `(movie:Movie)-[rel]-(director:Director)`

Својства чворова и релација

Својства се записују у витичастим заградама у дефиницији чвора или релације. На пример:

- `(:Person {name: 'Tom Hanks', born: 1956})`
- `-[:ACTED_IN {roles: 'Agent Smith'}]->`

Свако својство је карактеристично за један од следећих доступних типова: `BOOLEAN`, `DATE`, `DURATION`, `FLOAT`, `INTEGER`, `LIST`, `LOCAL DATETIME`, `LOCAL TIME`, `POINT`, `STRING`, `ZONED DATETIME` и `ZONED TIME`.

Упити

У овом поглављу биће описане поједине клаузуле у *Cypher* језику, као и начини креирања упита.

`MATCH` клаузула

`MATCH` клаузулом се врши претрага чворова, релација, својстава, лабела или шаблона (подграфа дефинисаног одређеним чворовима и релацијама). `MATCH` одговара `SELECT` клаузули у `SQL` упитном језику. Да би се пронађене вредности вратиле као резултат упита, након `MATCH` клаузуле се наводи `RETURN`.

`RETURN` клаузула

Овом клаузулом се наглашава повратна вредност упита, помоћу претходно дефинисаних променљивих. На пример:

- ```
  MATCH (p:Person)
  RETURN p
  ```

  Овај упит проналази и враћа све чворове са лабелом `Person`. У `RETURN` клаузули се помоћу променљиве `p` реферише на све пронађене чворове.

- ```
  MATCH (tom:Person {name:'Tom Hanks'})
  RETURN tom
  ```

  Упит проналази и враћа све чворове који имају лабелу `Person` и својство `name: 'Tom Hanks'`.

- ```
  MATCH (person:Person)
  WHERE person.name='Tom Hanks'
  RETURN person
  ```

  Клаузулом `WHERE` се испитују вредности својстава[1]. Овај упит је у сваком смислу еквивалентан претходном.

- ```
  MATCH (:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie:Movie)
  RETURN movie.title
  ```

  Овим упитом се враћају вредности својства `title`, односно називи свих филмова у бази у којима глуми `Tom Hanks`.

Додавање података у базу

`CREATE` клаузулом се чворови, релације или шаблони креирају и додају у базу. Наредним упитом се креира и додаје нови чвор:

- ```
  CREATE (:Movie {title: 'The Green Mile', released: 1999})
  ```

[1]Након кључне речи `WHERE`, може се навести `NOT`, а сврха таквог упита би била испитивање неједнакости. Такође се може испитати и да ли вредност својства припада одређеном опсегу (употребом неких од оператора $<, >, <=$ и $>=$), да ли одређено својство постоји (употребом `exists` метода над својством), итд.
  ```

Клаузула `CREATE` се често комбинује са другим клаузулама, нпр. са `RETURN` или са `MATCH`:

- ```
  MATCH (p:Person {name: 'Tom Hanks', born:1956})
  MATCH (m:Movie {title: 'The Green Mile', released: 1999})
  CREATE (p)-[:ACTED_IN {roles: 'Paul Edgecomb'}]->(m)
  ```

Други начин за додавање података је употребом клаузуле `IMPORT`. Подаци се на овај начин могу учитати из *csv* датотеке, из релационе базе, итд.

Остале клаузуле

- `SET` - Ова клаузула се користи за постављање или мењање вредности својстава чворова и релација.
- `DELETE` - Употребљава се за брисање чворова и релација. Брисање чвора који има релације узрокује грешку. Стога је пре брисања чвора најпре потребно обрисати његове релације. Клаузулом `DETACH DELETE` се поступак брисања чвора скраћује, односно није потребно прво покренути упит за брисање његових релација, већ се то у овом случају имплицитно одвија.
- `REMOVE` - Употребљава се за брисање својстава назначених чворова или релација.
- `MERGE` - Овом клаузулом се најпре испитује да ли чвор или релација постоји у бази. Искључиво у случају када то не важи, назначени чвор или релација се креира и додаје у базу. Дакле, употребом `MERGE` клаузуле се елиминише појава редундантности.

Агрегационе функције

Cypher упитни језик дефинише разне агрегационе функције, нпр. за рачунање минималне или максималне вредности, сума, просека, итд. Агрегација се рачуна за све повратне вредности или за сваку групу засебно (групе се формирају на основу поља у `RETURN` клаузули која нису агрегациона).

Једна од агрегационих функција је `count`, која се може употребити на два начина. Први начин подразумева синтаксу `count(n)`, којом се пребројава
  ```

појављивање `n` (`n` је чвор, релација или својство), не укључујући `null` вредности. Други начин јесте позивом `count(*)`, помоћу којег се пребројавају и `null` вредности. Дакле, за пребројавање чворова који имају одређено својство потребно је покренути следећи упит:

- ```
  MATCH (n:Node)
  RETURN count(n.property)
  ```

А сви чворови могу се пребројати на следећи начин:

- ```
  MATCH (n:Node)
  RETURN count(*)
  ```

Још неке агрегационе функције су: `size`, `collect`, `avg`, `max`, `min`, итд.

Глава 4

Spring радни оквир

У овој глави биће описане неке елементарне функционалности *Spring радног оквира*. Употребом ***Spring for GraphQL*** [18] и ***Spring Data Neo4j*** [18] пројеката ће касније бити имплементирана серверска апликација која се повезује на графовску базу и обрађује *GraphQL* захтеве клијентске апликације. Једно поглавље ће бити посвећено и *GraphQL* технологији, како би се разумели принципи *Spring for GraphQL* пројекта. Поред описивања самих карактеристика, биће наведене разлике између *REST* и *GraphQL* технологија, двеју најзаступљенијих технологија за развој *API* сервера. Прво поглавље је посвећено *Maven* алату, помоћу којег ће се управљати пројектом серверске апликације.

4.1 Управљање пројектом помоћу *Maven* алата

У управљању великим софтверским пројектом се јавља много изазова. На пример, захтевно је преузети и подесити све екстерне библиотеке од којих пројекат зависи. Такође, превођење би био обиман и исцрпљујућ посао, с обзиром на велики број датотека које чине код. Из овога следи и проблем у организацији свих фајлова, ради поједностављивања проналажења функционалности у коду и, уопштено, поједностављивања управљања пројектом.

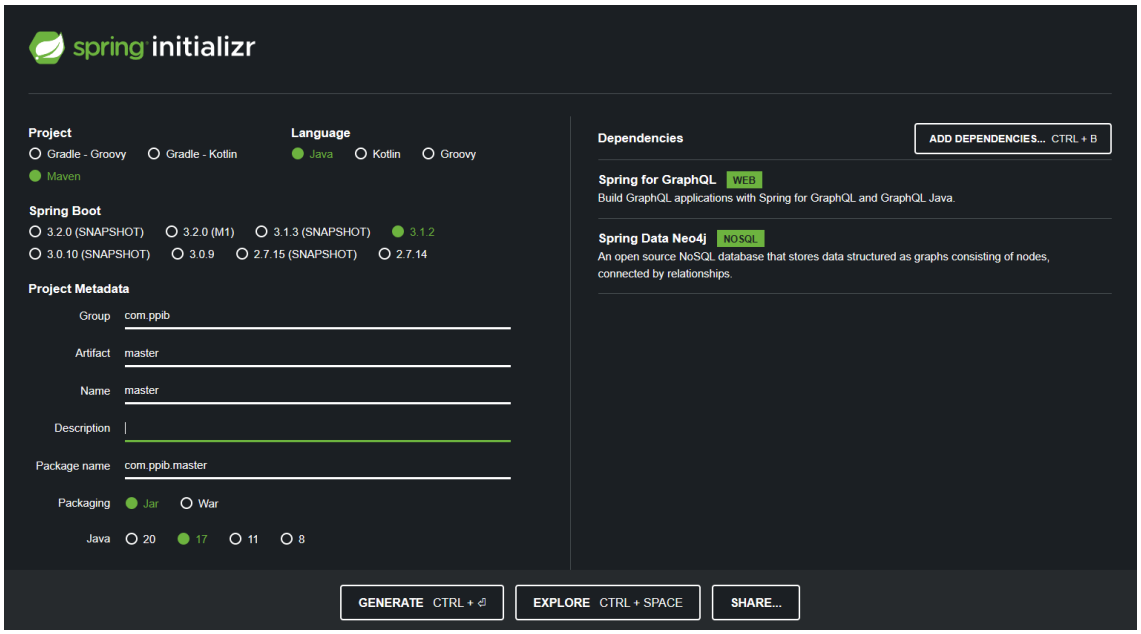
Ови процеси се могу олакшати употребом неког од алата за управљање пројектима. Један од њих је ***Maven*** [10], који је карактеристичан за Јава пројекте и који ће бити употребљен у овом раду. *Maven* аутоматизује про-

цес покретања програма и разрешавања зависности од екстерних библиотека. *Maven* разрешава и проблем организације кода - код се организује у складу са конвенцијом *Standard Directory Layout*, чији је шематски приказ дат на слици 4.1.

```
project-root
   ├── src
   │   ├── main
   │   │   ├── java              (Изворни код)
   │   │   └── resources         (Остали ресурси, подешавања,...)
   │   └── test
   │       └── java              (Тестирање изворног кода)
   ├── target                    (Излазни директоријум за резултате компилације,...)
   ├── pom.xml                   (Конфигурација за Maven)
   └── other project files       (README, license,...)
```

Слика 4.1: Шематски приказ организације *Maven* пројекта

Датотека ***pom.xml*** представља срж *Maven* пројекта. У овом раду је та датотека, али и цео шаблон пројекта аутоматски генерисан помоћу алата ***Spring Initializr*** (слика 4.2).



Слика 4.2: *Spring Initializr* опције за пројекат серверске апликације

У *pom.xml*-у се дефинишу неке од карактеристика пројекта (назив пројекта, верзија Јаве, верзија пројекта, родитељски пројекат...), али и екстерне библиотеке од којих пројекат зависи. Зависности се набрајају унутар ознака

`<dependency>`. За коришћење *Spring for GraphQL* и *Spring Data Neo4j* пројеката се у *pom.xml* фајлу дефинишу следеће зависности:

1. `spring-boot-starter-data-neo4j`
2. `spring-boot-starter-graphql`
3. `spring-boot-starter-test`
4. `spring-webflux`
5. `spring-graphql-test`
6. `spring-boot-starter-web`

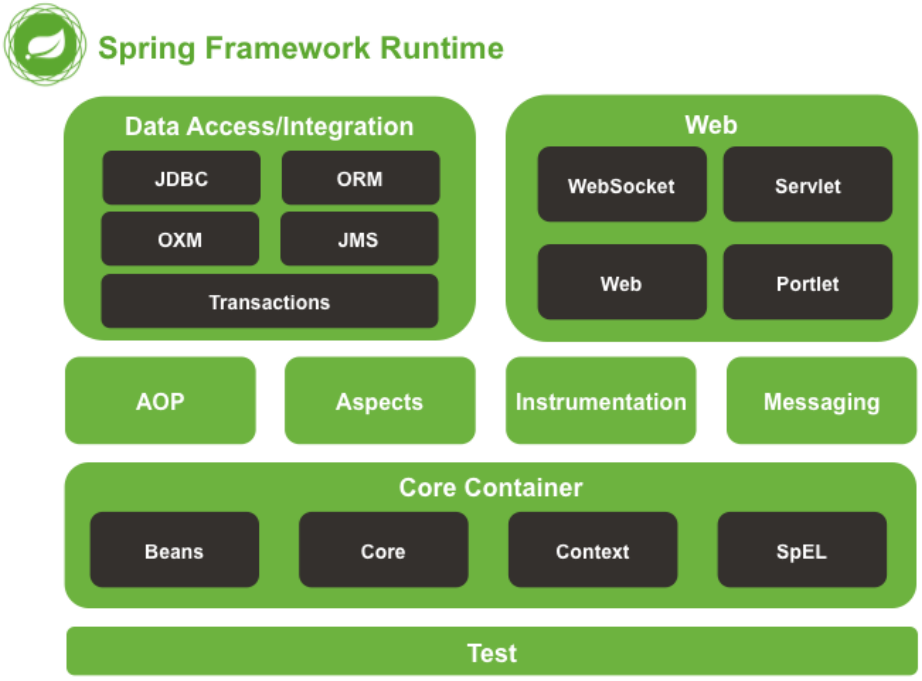
Пројекат серверске апликације наслеђује родитељски пројекат `spring-boot-starter-parent`, што заправо значи да ће од њега преузети конфигурације и зависности. Овај приступ конфигурисања је уобичајен за *Spring Boot* апликације.

Дакле, помоћу *Maven* алата је значајно олакшан процес управљања пројектом. Такође је једноставно изградити и покренути пројекат. Приликом изградње се најпре аутоматски анализира *pom.xml* датотека и проналазе карактеристике пројекта. Затим се преузимају све екстерне библиотеке из ***Maven* репозиторијума** [11] (уколико раније нису већ преузете) и додају у путању пројекта, након чега се компилира изворни код и покреће програм.

4.2 *Spring* радни оквир

Spring радни оквир (енгл. *Spring Framework*) је креиран 2002. године са циљем да се омогући удобност у процесу развоја Јава апликација. *Spring* омогућава писање чистог кода, који је, према томе, једноставно одржавати и тестирати. Временом је *Spring* постао много више и данас се говори о многобројним ***Spring пројектима***. У основи свих њих се налази сами *Spring* радни оквир. Неки од пројеката су: *Spring Boot*, *Spring Cloud*, *Spring Authorization Server*, *Spring for GraphQL*, *Spring Data Neo4j*, који је део фамилије пројеката *Spring Data*, итд.

Spring радни оквир се састоји од различитих модула, који су приказани на слици 4.3. У наставку ће бити описане неке карактеристике *Spring* контејнер (енгл. *Core Container*) модула, који имплементира најважније *Spring* функционалности.



Слика 4.3: *Spring* радни оквир

Spring контејнер

Spring контејнер је само језгро *Spring* радног оквира. Управља креирањем и животним циклусом објеката, разрешавањем њихове међусобне зависности употребом принципа **уметања зависности** (енгл. *dependency injection*), итд. Објекти који живе у *Spring* контејнеру се називају **зрна** (енгл. *beans*) . Управљање животним циклусима објеката се одвија преко интерфејса `org.springframework.context.ApplicationContext`, за који постоје 2 типа имплементације. Прва имплементација се заснива на *xml* конфигурацији - односно, зрна се дефинишу у *xml* датотеци (у оквиру ознаке `<bean>`), а друга на употреби *Spring* анотација.

У наредном примеру је приказано дефинисање зрна у *beans.xml* датотеци, које представља инстанцу класе `Protein` чији је *UniProt* идентификатор H3BN02. Наведене су и вредности осталих идентификатора - што су заправо вредности одговарајућих атрибута класе *Protein*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 https://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean id="H3BN02" class="Protein">
   <property name="uniprotid" value ="H3BN02"> </property>
   <property name="ensemblid" value ="ENSP00000454623"> </property>
   <property name="geneid" value ="-"> </property>
 </bean>
</beans>
```

Овај објекат може да се дохвати у `main` функцији на следећи начин:

```
public static void main(String []args) {
  ApplicationContext context =
    new ClassPathXmlApplicationContext("beans.xml");

  Protein protein = (Protein) context.getBean("H3BN02");
}
```

Други начин дефинисања зрна је употребом анотација. Анотацијом `@Component` изнад дефиниције класе се *Spring*-у наглашава да су објекти класе зрна. На пример:

```
@Component
public class Protein {
  String uniprotid;
  String ensemblid;
  String geneid;

  //...
}
```

Додатно је потребно креирати класу која је обавезно означена анотацијама `@Configuration` и `@ComponentScan`, која се прослеђује конструктору контекста, уместо *xml* датотеке:

```
@Configuration
```

```
@ComponentScan
public class Konf {
}
```

Дакле, креирање контекста сада изгледа овако, па *Spring* аутоматски проналази класе које су означене анотацијом `@Component` и њихове објекте карактерише као зрна:

```
public static void main(String []args) {
  ApplicationContext context =
   new AnnotationConfigApplicationContext(Konf.class);

   //...
}
```

Уметање зависности

Када један објекат зависи од другог, обично није добра пракса да се објекат од којег зависи креира унутар конструктора или неког метода. Боља идеја је да се тај објекат креира на неком другом месту у коду, а потом *умеш̄не* у објекат којем је неопходан. На овој идеји се заснива принцип уметања зависности. Једна од најважнијих карактеристика *Spring* контејнера је подржавање овог принципа у разрешавању зависности међу објектима. Уметање зависности се имплементира аутоматски, експлицитном *xml* конфигурацијом или експлицитном *Java* конфигурацијом.

Аутоматско уметање зависности је најједноставнији приступ и заснива се на употреби анотације `@Autowired` изнад конструктора класе (уколико се уметање зависности дешава у конструктору класе), изнад `set` метода или изнад дефиниције одговарајућег атрибута класе.

Spring Boot

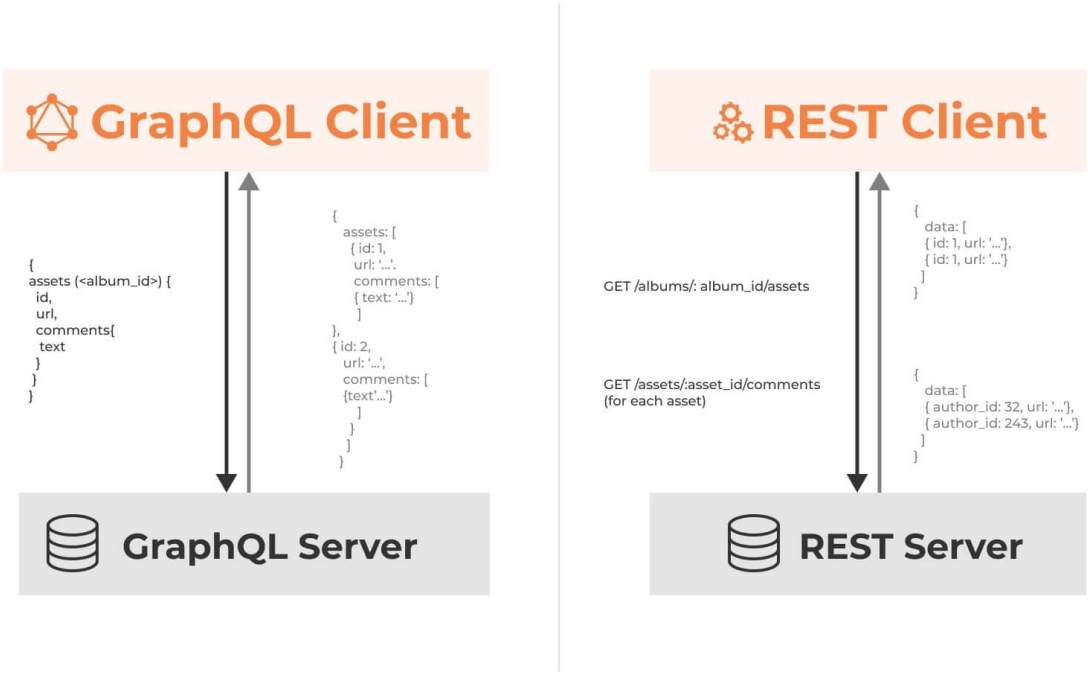
Из свега претходног је приметно да писање кода помоћу *Spring* радног оквира подразумева доста конфигурација, како у *xml* датотекама, тако и Јава конфигурација. Ово заправо представља сасвим нови изазов, који у неким случајевима може да поквари утисак о једноставности писања кода употребом *Spring*-а. Пројекат *Spring Boot* је настао са идејом да се отклоне ови недостаци, тј. да се додатно поједностави писање, конфигурисање и покретање *Spring* апликација.

Spring Boot обезбеђује високу аутоматизацију процеса конфигурације, скуп почетних зависности, алате који олакшавају детекцију и управљање изменама у коду и још много тога.

4.3 *GraphQL*

GraphQL технологија је развијена 2012. године од стране компаније *Facebook*. Ова технологија има много предности у односу на *REST* технологију и све је заступљенија у имплементацијама *API* сервера.

Основна разлика између двеју наведених технологија је у начину на који клијент захтева податке од сервера. *REST* дефинише јединствени *URI* (енгл. *Uniform Resource Identifier*) за сваки ресурс. Клијент онда помоћу *URI*-а дохвата ресурсе. Тачније, клијент у једном позиву од сервера може захтевати искључиво један ресурс. Насупрот овоме, клијент може у једном позиву захтевати произвољан број ресурса од *GraphQL* сервера. Сервер има прецизно дефинисану **схему** на основу које клијент дефинише захтев у виду једне ***GraphQL* операције** (слика 4.4).



Слика 4.4: Слање *REST* и *GraphQL* захтева

Закључак је да је *GraphQL* технологија ефикаснија од *REST* технологије. Ипак, имплементација *GraphQL* сервера је доста сложенија.

GraphQL схема

Сваки *GraphQL* сервер се карактерише схемом у којој се описују типови, односно ресурси који могу да се дохвате, као и начин на који се дохватају. У схеми се дефинишу и поља типова. На основу схеме се пишу *GraphQL* операције, којима се од сервера захтевају ресурси или којима се манипулише подацима. Операције за дохватање ресурса се називају ***GraphlQL* упити**. На пример, део схеме сервера у овом раду, која је дефинисана у фајлу *schema.graphql*, изгледа овако:

```
type Query {
  proteinById(uniprotid: String!): Protein
}
type Protein {
  id: ID!
  uniprotid: String!
  ensembl_ids: String!
  gene_ids: String!
  interacting_proteins1: [IncomingInteraction!]
  interacting_proteins2: [OutgoingInteraction!]
}
type IncomingInteraction {
  ...
```

4.4 Мере чворова графа

У овом поглављу биће описане мере које ће се рачунати над једним чвором, односно протеином. Рачунање ових мера ће у апликацији бити једноставно имплементирано употребом библиотеке ***JGraphT*** [7].

Степен чвора

У теорији графова, степен чвора `deg(v)` се дефинише као број грана које садрже чвор `v`. Дакле, степен једног чвора у овом раду представљаће број интеракција у којима учествује тај протеин (чвор).

Page rank

PageRank је алгоритам који је развила компанија *Google* и представља меру битности веб странице у мрежи. Овај алгоритам је представљао кључну компоненту која је у то време допринела ефикасности *Google* веб претраживача. Дакле, *page rank* чвора је мера битности тог чвора у графу и она се одређује на основу броја грана које садрже чвор, али и на основу *page rank*-ова његових суседа. Из овога следи да ако чвор има велики број битних суседа, велика је вероватноћа да ће имати велики *page rank*.

Варијанте централности чвора

Мером централности се дефинише значај чвора за одржавање повезаности графа. Постоје различите варијанте ове мере. На пример, централност по степену је заправо степен чвора. Такође, по дефиницији је јасно да је и *page rank* још једна варијанта централности. У овом раду ће за сваки чвор бити могуће израчунати **централност по блискости** и **релациону централност**.

Централност по блискости (у ознаци C_c) се рачуна помоћу формуле:

$$C_c(p_i) = \frac{N-1}{\sum_{k=1}^{N} d(p_i, p_k)},$$

где је N број чворова у графу, а $d(p_i, p_k)$ представља најкраће растојање између посматраног чвора и неког од преосталих у графу, са којим је повезан. Ова мера се интерпретира као просечна удаљеност чвора од свих осталих чворова у графу са којима је повезан.

Релациона централност (у ознаци C_b) се рачуна помоћу формуле:

$$C_b(p_i) = \sum_{j=1}^{N} \sum_{k=1}^{j-1} \frac{g_{jk(p_i)}}{g_{jk}},$$

где g_{jk} представља укупан број најкраћих путева који повезују чворове p_j и p_k, а $g_{jk}(p_i)$ представља број таквих путева који укључују чвор p_i.

Глава 5

Развој апликације

У овој глави ће бити описан поступак развоја апликације за визуелизацију и анализу података о протеин-протеин интеракцијама. Пре тога ће бити описан поступак интегрисања података о протеин-протеин интеракцијама у графовску базу. Креирана је и релациона база, која складишти исте податке, са циљем поређења перформанси ове две базе. Резултати поређења су такође наведени у овој глави.

5.1 Преузимање и интеграција података

У раду је најпре потребно преузети податке о протеин-протеин интеракцијама и интегрисати их у нову базу. Из база *IntAct*, *HIPPIE*, *BioGRID* и *Reactome* је могуће преузети податке записане у стандардизованом ***MITAB*** [14] формату за запис молекулских интеракција. *MITAB* формат дефинише записивање интеракције у једном реду текстуалне датотеке, при чему су све информације о њој раздвојене табулатором. Подаци из базе *STRING* се не могу преузети у овом формату, већ се преузима текстуална датотека у којој постоји знатно мање информација о протеин-протеин интеракцијама. Наиме, у једном реду су исписани искључиво *Ensembl Protein* идентификатори интерактора и различити скорови једне интеракције које генерише алгоритам ове базе. Информације о научним радовима, ауторима радова, алтернативним идентификаторима интерактора, организмима, типовима интеракције, итд. које уобичајено постоје у *MITAB* датотеци, нису наведене у овом случају. Податке из база је могуће преузети за специфични организам. У овом раду је фокус на хуманим протеин-протеин интеракцијама, односно на оним

интеракцијама код којих су оба интерактора из људског организма.

У табели 5.1 је приказан број интеракција који се добија преузимањем података из сваке од база.

Табела 5.1: Број хуманих протеин-протеин интеракција у различитим базама

IntAct	*STRING*	*BioGRID*	*Reactome*	*HIPPIE*	Укупно
1 194 448	11 938 498	1 137 537	104 095	783 182	15 157 760

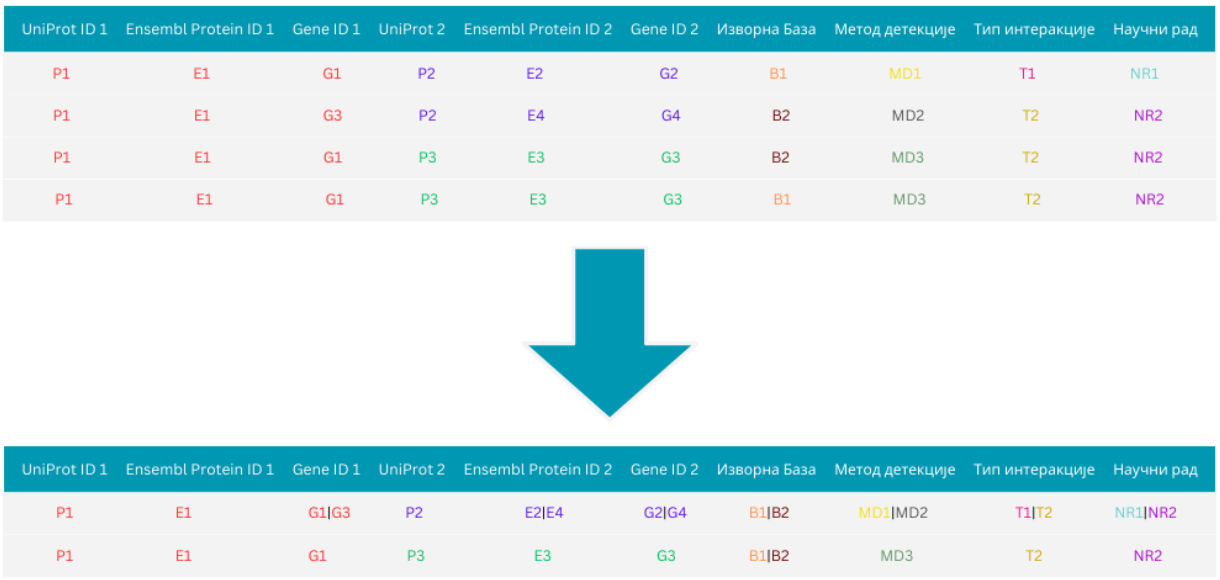
Добијени скуп интеракција је потребно додатно филтрирати, јер су из неких база (*BioGRID* и *IntAct*) преузете и оне интеракције у којој је само један интерактор из људског организма, а не оба.

Интеграција података у релациону базу

У интегрисаној бази ће се за сваку интеракцију чувати информације о *UniProt*, *Ensembl Protein*, *Entrez Gene* идентификаторима оба интерактора, при чему ће пар *UniProt* идентификатора бити примарни кључ релационе базе. Такође ће се чувати и информације о скору интеракције и о називима база из којих је та интеракција екстрахована (нпр. ниска *„biogrid/string"* означаваће да је податак о интеракцији узет из база *BioGRID* и *STRING*).

Неопходно је да за сваки протеин постоји информација о *UniProt* идентификатору (пошто ће то чинити примарни кључ базе). Као што је раније напоменуто, ово није случај са свим интерацторима преузетих интеракција. Подсећања ради, у бази *STRING* се користи *Ensembl Protein* идентификатор и за преузете интеракције не постоји информација о алтернативним идентификаторима интерактора. Алат помоћу којег је могуће мапирање било којих типова идентификатора у *UniProt* идентификатор (и обрнуто) је ***UniProt Mapper*** [21]. Помоћу овог алата мапирани су *Ensembl Protein* и *Entrez Gene* идентификатори скоро свих протеина за које није постојала информација о *UniProt* идентификатору. Ипак, неколицину идентификатора није било могуће пресликати. То су идентификатори протеина који нису ускладиштени у бази *UniProt*, али јесу у базама *Ensembl* или *Entrez Gene* (из тог разлога за њих није дефинисан *UniProt* идентификатор). Уколико за неки интерактор важи ово, интеракција је избачена из скупа података. Такође, мапирање није увек једнозначно. Мали број интеракција у којима учествује барем један протеин који нема једнозначно дефинисан *UniProt* идентификатор је такође избачен из скупа података.

Након мапирања, спојене су интеракције из више база које су исте - односно које имају исте интеракторе. При спајању се креира ниска о називима база из којих је интеракција екстрахована. Такође се надовезују *Ensembl Protein* идентификатори, *Entrez Gene* идентификатори, методи детекције, типови интеракција и научни радови (што је потребно урадити због генерисања скора). Шематски приказ овог корака је приказан на слици 5.1.



Слика 5.1: Спајање интеракција

Након што се изврше претходни кораци, у скупу података остаје 6 185 310 јединствених интеракција, за које се потом генерише скор употребом *MiScore* алата. Чак и за оне интеракције које су екстраховане из *IntAct* базе је потребно изнова генерисати скор, јер постоји могућност да су приликом спајања додате нове информације о интеракцији и да су у *IntAct* бази употребљени другачији тежински фактори, што утиче на резултујући скор.

За интеракције које постоје само у бази *STRING* постоји недостатак информација о типовима те интеракције и о броју научних радова. Број научних радова се поставља на 1 уколико је интеракција добијена неким експерименталним методом (када је *experimental score* или *database score* интеракције већи од 0) а у супротном на 0 [22]. Резултат недостатака информација јесте да ће оне интеракције које су екстраховане искључиво из базе *STRING* имати знатно мањи скор од осталих интеракција.

Скуп интеракција се потом учитава из *csv* датотеке у једну табелу релационе базе ***PostgreSQL*** [15] и притом се користи алат ***pgAdmin 4***.

Интеграција података у графовску базу

Подаци ће бити интегрисани у графовску базу употребом описаног система *Neo4j*. Команда `neo4j-admin database import` је употребљена у овом раду за учитавање података у базу. Подаци, односно чворови и релације које је потребно учитати се записују у прецизно дефинисаним *csv* датотекама [12] и путање до фајлова је потребно проследити као аргументе команде. У овом раду, у датотеци ***protein.csv*** су записани чворови, односно протеини. Првих неколико редова ове датотеке изгледа овако:

```
uniprotid:ID,ensemblid,geneid,:LABEL
1A01_HUMAN,"-","3105",Protein
Q7Z614,"ENSP00000332062|ENSP00000332062.4","124460",Protein
O60282,"ENSP00000393379|ENSP00000393379.1","3800",Protein
O60291-2,"-","-",Protein
E9PL57,"ENSP00000431482|ENSP00000431482.1","-",Protein
```

У заглављу датотеке се наводе називи својстава чворова: `uniprotid`, `ensemblid` и `geneid`, при чему се својство `uniprotid` сматра идентификатором, што значи да његова вредност мора бити јединствена за сваки чвор. Поред својстава се опционо записује и једна или више лабела чвора (што је у заглављу дефинисано са `:LABEL`).

Релације, односно интеракције су записане у датотеци ***interactions.csv***:

```
:START_ID,databases,score,:END_ID,:TYPE
P26572,"string","0.014594848",Q9Y5U8,INTERACT
Q9Y5Y6,"biogrid|hippie","0.4103663",P26572,INTERACT
Q9NWN3,"biogrid|hippie|intact","0.67245513",P60329,INTERACT
P60510,"biogrid|hippie|intact|string","0.91090393",P78318,
   INTERACT
P60709,"reactome","0.5073432",P60953,INTERACT
```

У заглављу датотеке са релацијама је потребно навести `:START_ID`, `:END_ID` И `:TYPE`, али се додатно записују и називи својстава - у овом случају то су `databases` и `score`.

Покретањем команде:

`neo4j-admin database import full --nodes=,,protein.csv'' --relationships=,,interactions.csv''` се креира и попуњава графовска база протеин-протеин интеракција. Подразумевани назив базе је *neo4j*.

Поређење релационе и графовске базе протеин-протеин интеракција

У овом одељку ће бити описане неке разлике у начину складиштења интеракција у графовској и релационој бази, али и разлике у временима извршавања неких упита.

Графовски модел је природнији за податке о интеракцијама од релационог. Сви протеини и њихови атрибути се складиште само једном, као чворови графа (има их 32 643). У релационој бази, у више различитих редова може бити записан исти протеин. Нпр. протеин чији је *UniProt* идентификатор *Q9Y6B6* учествује у 1059 интеракција, па су сви његови атрубити записани у толико редова, што свакако представља мање ефикасан приступ.

Из овога следи да се *Cypher* упити извршавају брже од *SQL* упита. *Cypher* упит за излиставање *UniProt* идентификатора свих протеина у бази се извршава за **3** ***ms***:

- `MATCH (protein) RETURN protein.uniprotid`

Док се аналогни *SQL* упит извршава чак **1** ***min***:

- ```
  SELECT ,,UNIPROTID_first'', ,,ENSEMBLID_first'', ,,GENEID_first''
  FROM ,,INTERACTIONS'' UNION
  SELECT ,,UNIPROTID_second'', ,,ENSEMBLID_second'', ,,GENEID_second''
  FROM ,,INTERACTIONS''
  ```

Даље, следећи упит проналази и враћа све информације о интеракцијама у којима учествује протеин *Q9Y6B6* за **3 ms**:

- `MATCH (p1 uniprotid: 'Q9Y6B6')-[int]-(p2) RETURN p1, int, p2`

Аналогни упит у *SQL* ће се извршавати дупло спорије:
  ```

- `SELECT * FROM ,,INTERACTIONS'' WHERE ,,UNIPROTID_first'' = 'Q9Y6B6' OR ,,UNIPROTID_second'' = 'Q9Y6B6'`

Такође, када се упореде синтаксе ових упита, јасно је да *SQL* делује доста неугледно и неинтуитивно, насупрот *Cypher* језику.

5.2 Развој серверске апликације

Серверска апликација се повезује на графовску базу употребом *Spring Data Neo4j*. У том пројекту је описана синтакса за дефинисање домена података из базе, што апликацији даје могућност дохватања података. Сервер обрађује *GraphQl* захтеве клијента, што је омогућено употребом функционалности *Spring for GraphQL* пројекта.

Spring Data Neo4j

Spring Data Neo4j је део *Spring Data* породице пројеката, којој припадају и *Spring Data REST*, *Spring Data JDBC*, *Spring Data MongoDB* и још многи други пројекти. *Spring Data Neo4j* се директно ослања на *Neo4j Java Driver* (који се још назива и *Bolt*), а инстанцу драјвера ће обезбедити *Spring Boot*.

Помоћу *Spring Data Neo4j* се дефинишу доменске класе. У овом раду класа `Protein` моделује чворове из базе са лабелом `Protein`, што се означава анотацијом `@Node` и њеним атрибутом `primaryLabel`:

```
1  @Node(primaryLabel="Protein")
2  public class Protein {
3    @Id
4    private final String uniprotid;
5    @Property("ensemblid")
6    private String ensembl_ids;
7    @Property("geneid")
8    private String gene_ids;
9
10   @Relationship(type = "INTERACT",
11      direction = Direction.INCOMING)
12   private List<IncomingInteraction> interacting_proteins1;
13
14   @Relationship(type = "INTERACT",
15      direction = Direction.OUTGOING)
```

```
16   private List<OutgoingInteraction> interacting_proteins2;
17
18   //... (Konstruktor, GET metodi)
19 }
```

Анотација `@Property` над пољем класе означава својство чвора (анотацији се прослеђује назив својства у бази), а `@Relationship` релацију. Анотација `@Relationship` има атрибуте `type` (означава тип релације) и `direction` (означава смер релације). Ова анотација се употребљава над листом чији су елементи инстанце класе која описује релацију и њена својства. На пример, класа `IncomingInteraction` изгледа овако:

```
1  @RelationshipProperties
2  public class IncomingInteraction {
3    @Id @GeneratedValue
4    private Long id;
5    private final String score;
6    private final String databases;
7
8    @TargetNode
9    private final Protein interactor;
10
11   //... (Konstruktor, GET metodi)
12 }
```

У овој класи се наводе сва својства релације, али и објекат који представља почетни чвор интеракције (поље означено анотацијом `@TargetNode`). За релације које имају смер `@Direction.OUTGOING`, анотацијом `@TargetNode` се дефинише цильни чвор. Класа `OutgoingInteraction` изгледа исто као класа `IncomingInteraction`, а у њој, када се узме у обзир смер релације, поље `interactor` означава цильни чвор.

У класама које моделују чворове и релације се једно поље означава анотацијом `@Id` и то поље представља идентификатор објекта. Уколико се поље додатно означи са анотацијом `@GeneratedValue`, тада се његова вредност аутоматски генерише (односно, његова вредност се не прослеђује експлицитно конструктору класе).

Након дефинисања домена је потребно креирати **репозиторијум**, који представља апстракцију складиштења података. Репозиторијум се имплементира као интерфејс који наслеђује: `Neo4jRepository<Protein, String>`. Ова апстракција омогућава програмеру једноставно извршавање упита. На-

име, потребно је само позвати неки од многобројних метода доступних из `Neo4jRepository` интерфејса.

```
@GraphQlRepository
public interface ProteinRepository extends
          Neo4jRepository<Protein, String> {
}
```

Овако дефинисан репозиторијум се користи за дохватање чворова који се могу пресликати у класу `Protein` (а то су чворови са лабелом `Protein`, по дефиницији те класе). `String` означава тип идентификатора те класе (поље `uniprotid`) Анотација `@GraphQlRepository` ће бити објашњена у наредном поглављу.

На крају се у датотеци ***application.properties*** конфигуришу параметри за повезивање на базу: `spring.neo4j.uri=neo4j://localhost:7687` и информације о корисничком имену и лозинци, уколико је за приступ бази неопходна аутентификација.

Важно је напоменути да *Spring Data Neo4j* пројекат није креиран да даје добре перформансе приликом рада са великим базама [19]. *Spring Data Neo4j* дефинише много слојева апстракције над ***Neo4j API***, што нарушава ефикасност, али пружа могућност брзог и једноставног повезивања на базу и удобност при дохватању и раду са подацима из (мање) базе. У овом раду је апликација тестирана на рачунару са `8GB` меморије и исправно је радила када је у *Neo4j* бази било садржано стотину хиљада интеракција.

Spring for GraphQL

Овај *Spring* пројекат пружа функционалности за развој *GraphQL* сервера. Имплементиран је тако да буде једноставан за интеграцију са неким од *Spring Data* пројеката. Потребно је додати анотацију `@GraphQLRepository` изнад класе која представља *Neo4j* репозиторијум и тиме се *Spring for GraphQL*-у наглашава да ће репозиторијум бити употребљен за *GraphQL* упите. Контролер је класа која опслужује захтеве клијента и означена је анотацијом `@Controller`. Овај објекат зависи од инстанце класе `ProteinRepository`:

```
@Controller
public class InteractionsController {
  private final ProteinRepository repository;

```

```
5   @Autowired
6   public InteractionsController(ProteinRepository
7       repository) {
8     this.repository = repository;
9   }
10
11  @QueryMapping
12  public Protein proteinById(@Argument(name = "uniprotid")
13      String uniprotid) {
14    Optional<Protein> opt = repository.findById(uniprotid);
15    return opt.isEmpty() ? null : opt.get();
16  }
17
18  //...
19 }
```

У репозиторијуму постоји више дефинисаних метода. Метод `findById` проналази и враћа податак са прослеђеним идентификатором. Анотацијом `QueryMapping` се метод дефинише као хендлер (енгл. *handler*) и назив метода ће бити мапиран у поље типа `Query` које има исти назив, па се метод може позвати у *GraphQL* упиту на следећи начин:

```
1 {
2   proteinById (uniprotid : "Q8IYB7") {
3     gene_ids
4   }
5 }
```

5.3 Развој клијентске апликације

У овом раду ће се користити радни оквир *Angular* за креирање клијентске апликације. Клијентска апликација дохвата податке од *GraphQL* сервера. Те податке потом визуелизује употребом *JavaScript* библиотеке за генерисање различитих визуелизација - ***vis.js*** [23]. Додатно, за сваки чвор графа се исписују вредности мера - степен чвора, *page rank*, централност по блискости и релациона централност чвора.

CORS

CORS *(cross-origin resource sharing)* је сигурносни механизам имплементиран у веб прегледачима којим се дефинишу дозволе за комуникацију и размену ресурса између две апликације различитих домена. У овом раду, клијент од сервера захтева ресурсе (информације о интеракцијама) и они су покренути на различитим доменима - клијент на: `http://localhost:4200`, а сервер на: `http://localhost:8080`. Како је *CORS* подразумевано онемогућен, да би клијент могао да захтева ресурсе од сервера, у серверској апликацији, односно у датотеци *application.properties* је неопходно додати следеће две линије:

```
spring.graphql.cors.allow-credentials=true
spring.graphql.cors.allowed-origins=http://localhost:4200
```

Након чега захтеви клијента неће бити блокирани.

Визуелизација података о протеин-протеин интеракцијама

Апликација креирана у радном оквиру *Angular* се састоји од модула, сервиса и компоненти. Сваки модул представља једну логичку целину апликације и обично има своју руту. У сервисима се имплементирају позадинске функционалности (нпр. комуникација различитих компоненти), а компоненте дефинишу изглед и функционисање елемената апликације (*HTML*, *CSS*,...). У овом раду су креиране две компоненте: `interactions` и `graph-visualization`.

У компоненти `interactions` је дефинисана форма за уношење идентификатора протеина чије интеракције ће се касније визуелизовати. Када се упише идентификатор протеина, у компоненти се од сервера захтевају подаци о одређеним интеракцијама и протеинима (сви идентификатори и мере), слањем *GraphQL* упита на `http://localhost:8080/graphql`, као `POST` или `GET` захтева. За ово је употребљена библиотека ***apollo-angular*** [2].

У другој компоненти `graph-visualization` су имплементирани сви елементи за визуелизацију података и испис информација. Визуелизација је имплементирана коришћењем библиотеке *vis.js*. Ово је популарна библиотека којом се могу генерисати интерактивне визуелизације података. Различитим интеракцијама са чворовима и релацијама је могуће покренути одређени до-

гађај (енгл. *event*). Прецизније, у овом раду се кликом на неки елемент исписују информације о њему - уколико је елемент чвор, исписаће се информације о идентификаторима протеина и мерама - *page rank*, степен, централност по блискости и релациона централност чвора; у случају релације исписаће се скор интеракције и називи база из којих су информације о интеракцији екстраховане. Догађај који се покреће двоструким кликом на један чвор подразумева визуелизацију свих суседа тога чвора.

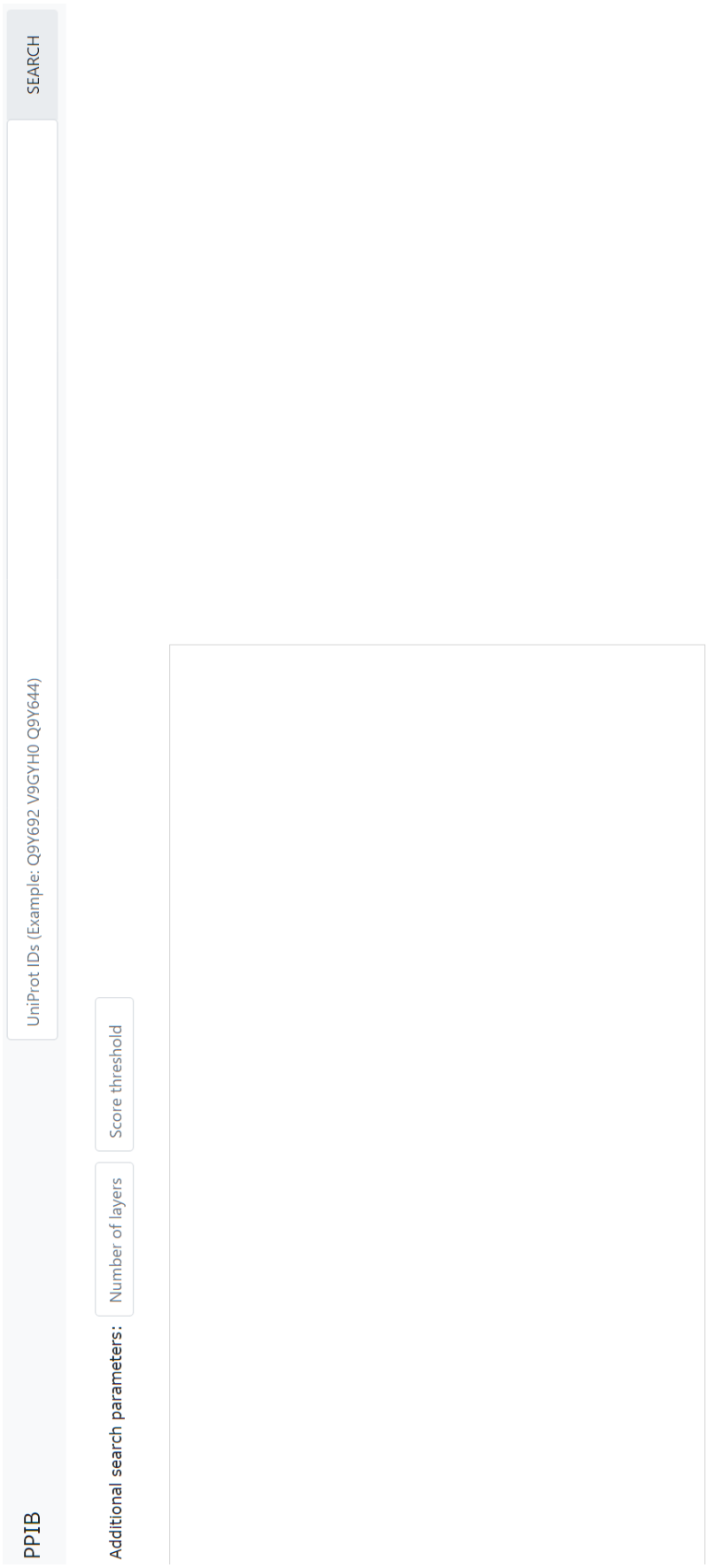
Подаци које је компонента `interactions` преузела од сервера се морају проследити компоненти `graph-visualization`, што је имплементирано кроз различите сервисе: `DataPassingService`, `BetwenneessScoreService`, `PagerankService`, `ProteinDegreeService`, `ClosenessScoreService`.

На слици 5.2 је приказан почетни изглед креиране клијентске апликације.

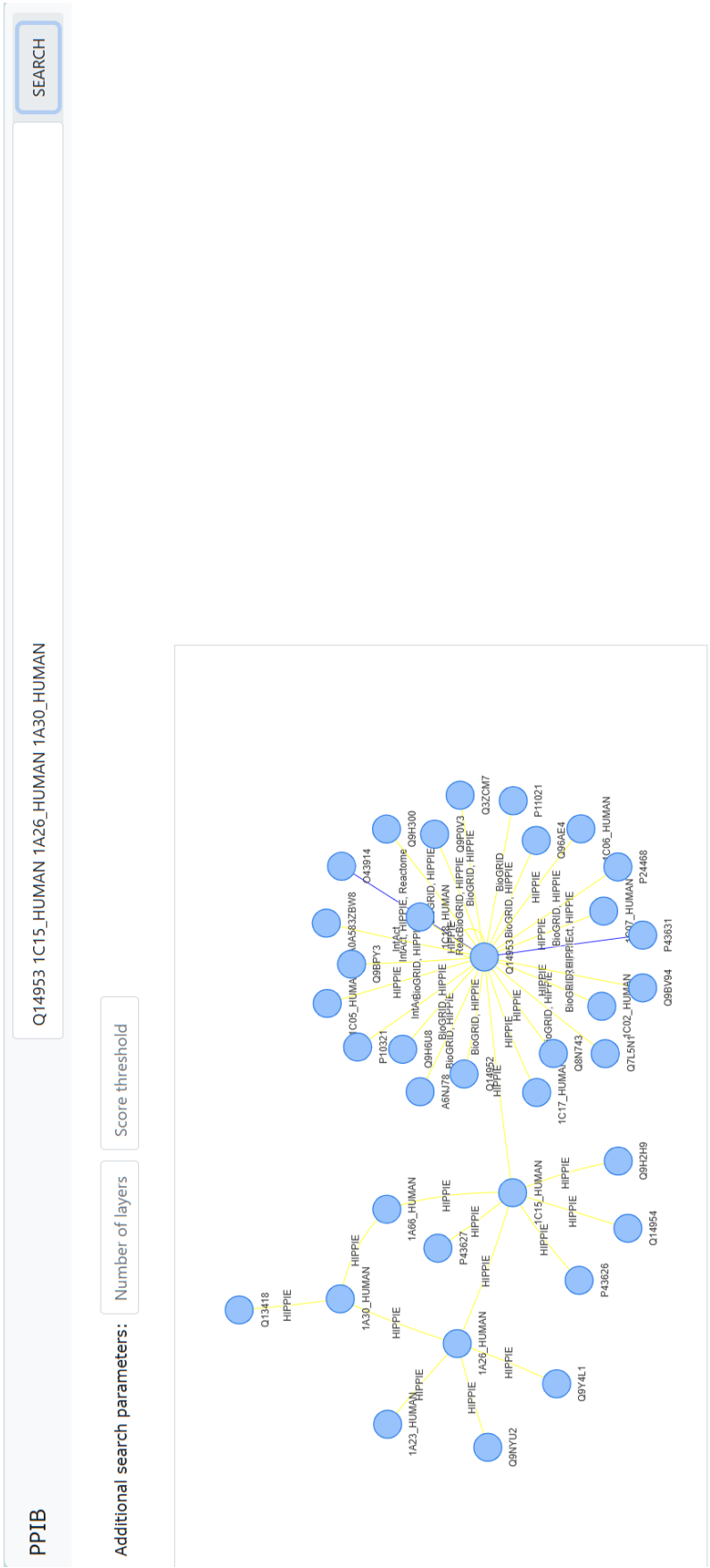
У поље за претрагу се уноси произвољан број *UniProt* идентификатора протеина. Кликом на дугме `SEARCH` се од сервера дохватају све интеракције у којима учествују ти протеини, те се визуелизују (слика 5.3).

Грана је обојена црвеном уколико је скор те интеракције мањи од 0.3, жутом ако је мањи од 0.6 (а већи од 0.3), а у супротном плавом бојом.

Додатни параметри претраге - број слојева (`Number of layers`) и праг скора (`Score threshold`) имају подразумеване вредности 1 и 0 редом. Корисник у одговарајућим пољима може променити њихове вредности. Када је број слојева 1, приказаће се само директни суседи уписаног чвора (или чворова), односно тај протеин и само они протеини са којима он интерагује; уколико је број слојева 2, приказаће се и суседи његових суседа, итд. Другим параметром - прагом скора се обезбеђује да се оне гране које представљају интеракције са скором мањим од прага не приказују. Подешавање овог параметра је корисно, јер дохватање података и њихова визуелизација може бити захтевно, када се узме у обзир чињеница да креирана база садржи преко 6 милиона интеракција.



Слика 5.2: Почетни изглед апликације



Слика 5.3: Визуелизација интеракција

Глава 6

Закључак

У овом раду су интегрисани подаци из јавно доступних база протеин-протеин интеракција у једну графовску базу хуманих протеин-протеин интеракција. Такође је креирана и релациона база у којој су ускладиштени исти подаци. Затим су упоређена времена извршавања упита, али и остале карактеристике ових база и показано је да је графовска база знатно бољи избор за податке о протеин-протеин интеракцијама. Даље је креирана клијент-сервер апликација која користи савремене приступе - нпр. сервер се заснива на релативно новој *GraphQL* технологији, а клијентска апликација је развијена као једностранична апликација употребом *Angular*-а. Подаци о протеин-протеин интеракцијама се визуелизују, али додатно, могуће је извршити неке корисне анализе чворова графа. Односно, могуће је израчунати степен чвора, *page rank* и две варијанте централности чвора: централност по блискости и релациону централност.

Базе протеин-протеин интеракција се редовно ажурирају новим подацима, стога се у будућности и сама графовска база креирана у овом раду може ажурирати додавањем нових интеракција из база *BioGRID*, *STRING*, *HIPPIE*, *Reactome* и *IntAct*. Додатно се могу интегрисати и подаци из неких других база, зарад прикупљања што више доступних протеин-протеин интеракција. Ова база за сада складишти информације о идентификаторима интерактора, скор и назив изворних база интеракција. Унапређење базе би подразумевало и складиштење других корисних информација (нпр. метод детекције интеракције или типови интеракције). И сама апликација може да се унапреди. У ранијем поглављу су изнети детаљи о неефикасности *Spring Data Neo4j*-а при раду са великим базама, каква је база у овом раду. Дакле, потребно је

променити приступ у серверској апликацији и уместо *Spring Data Neo4j* ослонити се на *Neo4j API*. Такође, апликација се може унапредити и додавањем нових анализа. Библиотека *JGraphT* пружа велики број метода за анализу графова, који би се могли искористити.

Библиографија

[1] Angular. `https://angular.io/`.

[2] Angular. `https://www.npmjs.com/package/apollo-angular`.

[3] Akhilesh Kumar Bajpai, Sravanthi Davuluri, Kriti Tiwary, Sithalechumi Narayanan, Sailaja Oguru, Kavyashree Basavaraju, Deena Dayalan, Kavitha Thirumurugan, and Kshitish K. Acharya. Systematic comparison of the protein-protein interaction databases from a user's perspective. *Journal of Biomedical Informatics*, 103:103380, 2020.

[4] The UniProt Consortium. UniProt: the Universal Protein Knowledgebase in 2023. *Nucleic Acids Research*, 51(D1):D523–D531, 11 2022.

[5] Noemi del Toro, Anjali Shrivastava, Eliot Ragueneau, Birgit Meldal, Colin Combe, Elisabet Barrera, Livia Perfetto, Karyn How, Prashansa Ratan, Gautam Shirodkar, Odilia Lu, Bálint Mészáros, Xavier Watkins, Sangya Pundir, Luana Licata, Marta Iannuccelli, Matteo Pellegrini, Maria Jesus Martin, Simona Panni, Margaret Duesbury, Sylvain D Vallet, Juri Rappsilber, Sylvie Ricard-Blum, Gianni Cesareni, Lukasz Salwinski, Sandra Orchard, Pablo Porras, Kalpana Panneerselvam, and Henning Hermjakob. The IntAct database: efficient access to fine-grained molecular interaction data. *Nucleic Acids Research*, 50(D1):D648–D653, 11 2021.

[6] Antonio Fabregat, Konstantinos Sidiropoulos, Guilherme Viteri, Oscar Forner, Pablo Marin-Garcia, Victor Arnau, Peter D'Eustachio, Lincoln Stein, and Henning Hermjakob. Reactome pathway analysis: a high-performance in-memory approach. *BMC Bioinformatics*, 18(1):142, Mar 2017.

[7] Jgrapht. `https://jgrapht.org/`.

[8] Simon Jupp and et al. A new ontology lookup service at embl-ebi. In James Malone and et al., editors, *Proceedings of SWAT4LS International Conference 2015*, 2015.

[9] Donna Maglott, James Ostell, Kim D. Pruitt, and Tatiana Tatusova. Entrez gene: gene-centered information at ncbi. *Nucleic Acids Res*, 33(Database issue):D54–D58, 2005.

[10] Apache maven. `https://maven.apache.org/`.

[11] Maven repository. `https://mvnrepository.com/`.

[12] Neo4j graph database. `https://neo4j.com/`.

[13] Rose Oughtred, Jennifer Rust, Christie Chang, and et al. The BioGRID database: A comprehensive biomedical resource of curated protein, genetic, and chemical interactions. *Protein Science*, 30(1):187–200, 2021.

[14] Livia Perfetto, Marcio L Acencio, Graeme Bradley, and et al. Causaltab: the psi-mitab 2.8 updated format for signalling data representation and dissemination. *Bioinformatics*, 35(19):3779–3785, 2019.

[15] Postgresql. `https://www.postgresql.org/`.

[16] V. S. Rao, K. Srinivas, G. N. Sujini, and G. N. Kumar. Protein-protein interaction detection: methods and analysis. *International Journal of Proteomics*, 2014:147648, 2014.

[17] Martin H. Schaefer, Jean-Fred Fontaine, Arunachalam Vinayagam, Pablo Porras, Erich E. Wanker, and Miguel A. Andrade-Navarro. Hippie: Integrating protein interaction networks with experiment based quality scores. *PLOS ONE*, 7(2):1–8, 02 2012.

[18] Spring. `https://spring.io/`.

[19] Spring Data. Spring data reference documentation: Performance.

[20] Damian Szklarczyk, Rebecca Kirsch, Mikaela Koutrouli, Katerina Nastou, Farrokh Mehryary, Radja Hachilif, Annika L Gable, Tao Fang, Nadezhda T Doncheva, Sampo Pyysalo, Peer Bork, Lars J Jensen, and Christian von Mering. The STRING database in 2023: protein–protein association

networks and functional enrichment analyses for any sequenced genome of interest. *Nucleic Acids Research*, 51(D1):D638–D646, 11 2022.

[21] Uniprot mapper. `https://www.uniprot.org/id-mapping`.

[22] Jose M Villaveces, Rafael C Jiménez, Pablo Porras, and et al. Merging and scoring molecular interactions utilising existing community standards: tools, use-cases and a case study. *Database (Oxford)*, 2015:bau131, 2015.

[23] vis.js. `https://visjs.org/`.

[24] wwPDB consortium. Protein data bank: the single global archive for 3d macromolecular structure data. *Nucleic Acids Res*, 47:D520–D528, 2019.

[25] Andrew D. Yates, Premanand Achuthan, Wasiu Akanni, and et al. Ensembl 2020. *Nucleic Acids Res*, 48(D1):D682–D688, 2020.

Биографија аутора

Бојана Јошић рођена је 16. октобра 1998. у Београду. Гимназију у Младеновцу је завршила као носилац Вукове дипломе. Смер Рачунарство и информатика на студијском програму Математика уписала је 2017. године, а завршила 2021. године. Те године је уписала и мастер студије на истом смеру. Од октобра 2021. године до марта 2023. је била запослена на позицији сарадника у настави на Математичком факултету. Од марта 2023. до данас је демонстратор на Математичком факултету и запослена је у компанији *BGI* као програмер алгоритама (енгл. *algorithm developer*).