Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

Gráfelméleti módszerek alkalmazása a bioinformatikában

Témavezető:

Dr. habil. Szabó László Ferenc

egyetemi docens

Szerző:

Temesi András

Programtervező informatikus BSc.

Budapest, 2020

Tartalom

1. Bevezetés 4

2. Elméleti háttér 6

2.1 Egyszerű eset 7

2.1.1 A probléma komplexitása 8

2.1.2 Informatikai reprezentáció 9

2.1.3 Alapvető algoritmusok 10

2.1.4 Korábbi heurisztika 11

2.1.5 Javaslatok 13

2.1.6 Precheck és rekurzió 14

2.1.7 Egyszerű eset eredmények 16

2.2 Rendezett általános eset 17

2.2.1 A feladat meghatározása 18

2.2.2 Alapalgoritmusok 19

2.2.3 Precheck általánosítása 21

2.2.4 Rekurzió általánosítása 22

2.2.5 Általános eset eredmények 23

2.3 Rendezetlen általános eset 24

2.3.1 A feladat meghatározása 24

2.3.2 A probléma komplexitása 25

2.3.3 Algoritmusok 26

2.4 Összefoglalás 26

3. Felhasználói dokumentáció 28

3.1 Szükséges környezet 28

3.2 A program használata felhasználói felülettel 29

3.3 Baloldali menügombok 29

3.3.1 „Feladat bemutatása” gomb 29
3.3.2 „Megoldás adott számsorra” gomb 31
3.3.3 „Tesztelési eredmények” gomb 33
3.3.4 „Haladó funkciók” gomb 34
3.3.5 „Súgó és információk” gomb 37
3.4 Backend rész 38
4. Fejlesztői dokumentáció 40
4.1 Célkitűzések 40
4.1.1 Logikai rész elképzelése 40
4.1.2 Felület elképzelése 41
4.2 Megvalósítás 42
4.2.1 Backend 42
4.2.2 Algoritmusok 43
4.2.3 Kapcsolat a backend és frontend közt 48
4.2.4 Frontend 50
4.2.5 DrawWidget 56
4.3 Tesztelés 57
4.3.1 Manuális tesztelés 57
4.3.2 Backend unittesztek 57
4.3.3 Frontend unittesztek 58
5. Összefoglalás 59
5.1 További fejlesztési lehetőségek 59
6. Köszönetnyilvánítás 60
7. Irodalomjegyzék 61

1. Bevezetés

A baktériumsejtek sok replikációs ciklus során spontán mutációkat halmoznak fel, melyek új klónok születését eredményezik. Ennek következtében a fejlődő baktériumpopuláció idővel eltérő klonális összetétellel rendelkezik, amint azt a hosszútávú evolúciós kísérletek is mutatják. A populációban az új klónok haplotípusának, a klónok gyakoriságának, valamint az evolúciós történetének pontos meghatározása hasznos a korrelációs mutációk evolúciós okainak megértéséhez. (Elena & Lenski, 2003)

Az elmúlt évtizedben kiderült, hogy különböző öröklődési minták meghatározására tett kísérletek algoritmikus problémákra vezethetők vissza. Ezen minták szemléltetésére speciális gráfokat, irányított fákat használnak. A témában új megközelítést alkalmaz egy 2019-ben megjelent cikk (Ismail & Tang, 2019), mely egy likelihood-függvényt definiál, ami valószínűségi értékeket rendel a lehetséges gráfokhoz. Sajnos a függvényt maximalizáló optimális fa meghatározására nem ismert hatékony módszer. Az optimális fa meghatározása feltehetően NP-nehéz probléma, ahogy valószínűsíthetően annak eldöntése is, hogy adott adatsorra létezik-e egyáltalán a feltételeket kielégítő fa. A problémával kapcsolatban a naív-keresésen felül (ahol minden fa értékét ellenőrizzük) egy heurisztika született, mely alapvetően közelíteni próbálja az optimális fát (Ismail & Tang, 2019). Munkám során ennek a publikációnak eredményeiből indulok ki, kutatásom célja pedig olyan algoritmus kifejlesztése, amely hatékonyabban állítja elő az öröklési mintát reprezentáló, a likelihood-függvény szerint optimális irányított gráfot.

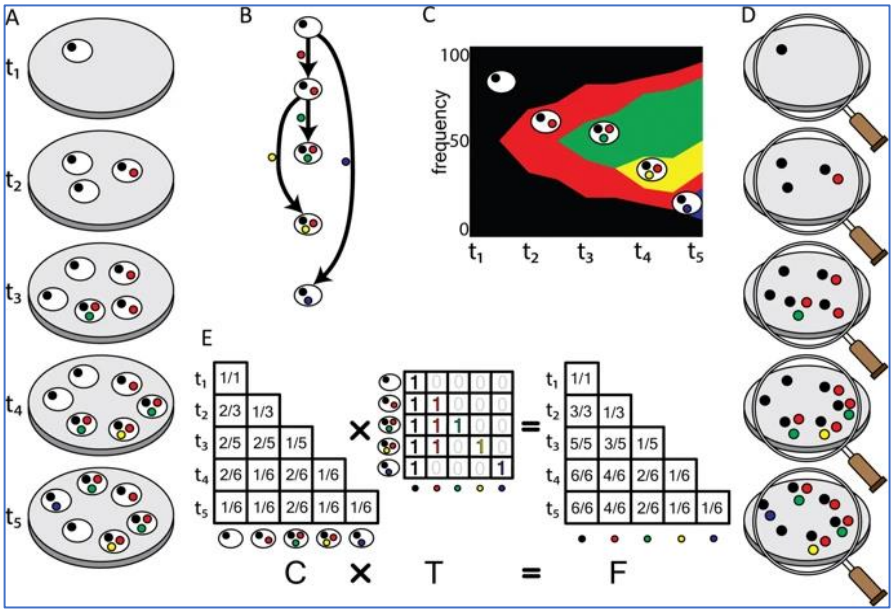
A végső cél a baktériumok valós, azaz tapasztalati adatsoraiból az optimális fa megépítése, mint például a - gyorsan változó klonális összetétele miatt a témában gyakran kutatott - E. Coli baktérium esetén (Rainey & Rainey, 2003) (Barrick, és mtsai., 2009) (Behringer, és mtsai., 2018) Dolgozatomban azonban főleg korábban előállított, szimulált adatsorokkal dolgozom, melyek különböző mértékben egyszerűsítései a valós adatsoroknak. A szimulációs algoritmus a likelihood-függvénnyel az (Ismail & Tang, 2019) publikációban jelent meg, és az adatokat egy sztochasztikus folyamattal készíti el. Néhány szimulált adatsort az (Ismail & Tang, 2019) cikk szerzői bocsátottak

rendelkezésemre. Már meglévő algoritmusokat, és két általam kifejlesztett algoritmust (2.1.6 fejezet) implementálok, a különböző egyszerűsítések szerinti módosításokkal, melyek együttesen, kis elemszámú vagy ideális adatsor esetén a teljes keresett gráfot elő tudják állítani. A korábbi heurisztikához képest fontos előrelépés, hogy mindenképp a legjobb függvényértékű fát találja meg az eljárás. Bár a naív-keresésnél nagyságrendekkel jobb a futási idő, a nagyobb elemszámú, nem ideális esetekben továbbra is túl számításigényes legjobb fa meghatározása.

Az általam bevezetett új heurisztika a gráf egy részét teljes biztonsággal, polinomiális költséggel határozza meg. Valós adatokon tesztelve az látszik, hogy a legjobb fa megtalálásáról valószínűleg le kell mondanunk, azonban a részgráfokat meghatározó algoritmusom, a korábbi heurisztikával kombinálva javíthat a közelítő fa megtalálásának futási idején.

2. Elméleti háttér

A bevezetőben említett, baktériumok esetén megfigyelt evolúciós folyamatot szeretnénk modellezni. Különböző időpillanatokban méréseket végzünk, és feljegyezzük a populáción belül a klónok arányát. Az ilyen megfigyelések mérési adatai mátrixokkal reprezentálhatók. Az alábbi ábra esetén az F-fel jelölt mátrix ismert számunkra. Szeretnénk felállítani a B-vel jelölt fát, ami az evolúciós folyamatot írja le.



1. ábra: klónok és haplotípusok példa (Ismail & Tang, 2019)

A likelihood-függvény azt feltételezi, hogy a mutációk véletlenszerűen történnek. Ez alapján egy új klón észlelésénél annak az esélye, hogy egy korábbi klón az újnak a szülője, a korábbi klón előző mérési időpontban vett, teljes populáción belüli arányával határozható meg. Minden klónnak csak egy szülője lehet, ezáltal ha a legelső klónon kívül mindegyikre ezen arányok alapján választunk szülőt, akkor az arányok szorzata egy jó mérőszám arra vonatkozóan, hogy mennyire valószínű, hogy a mutációk ténylegesen a felállított fa szerint történtek. Az 1. ábra D betűvel jelzi a méréseinket, melyeket az F mátrixban foglalunk össze. A valóságban egy új időpontban történő mérés során több, mint egy új klónt fedezünk fel általánosságban. Az ábrán látható F négyzetes háromszögmátrix egyszerűsített modell, például az E. Coli baktérium esetén vett mérési adatok mátrixa sokkal több oszlopot tartalmaz, mint sort. A valós, mért adatsorok megoldásáról az 2.3 fejezetben írok bővebben. Három lépéssel fogok eljutni odáig,

először tovább egyszerűsítem a modellt, erre az egyszerűsített modellre vizsgálom az algoritmusokat, majd ezeket az algoritmusokat általánosítom.

2.1 Egyszerű eset

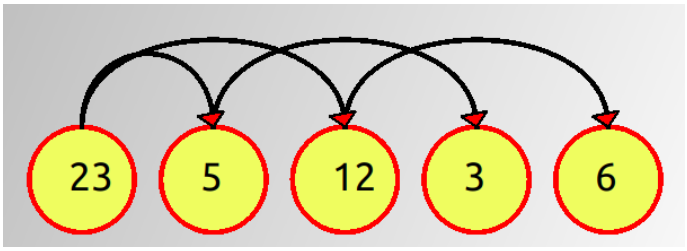
A probléma a legegyszerűbb esetben a következőképp fogalmazható meg:

- Adott n darab, 0-től $n-1$-ig indexelt pozitív szám. Tekintsük ezeket egy gráf csúcsainak. A nullától kezdődő indexelés használatával a későbbiekben egyszerűbbé válik a feladat informatikai implementációja.
- A második számtól kezdve (1-es index), minden csúcshoz pontosan egy kisebb indexűt rendelünk.
- Az így kapott rendezett párok (pl. $[0,1]$, $[0,2]$, $[1,3]$, $[2,4]$ stb.) tekinthetők a gráf irányított éleinek. Nevezzük a startcsúcsot szülőnek, míg a célcsúcsot gyereknek.
- Az így meghatározott gráf egy irányított fa, az első csúcsot leszámítva minden csúcsra pontosan egy bemenő éllel.
- A gráfot $\mathrm{megengedettnek}$ nevezzük, ha minden csúcsra teljesül, hogy az értéke legalább annyi, mint a belőle kimenő élek célcsúcsainak összege.
- Az összes lehetséges $\mathrm{megengedett}$ gráf közül szeretnénk megtalálni azt (több azonos esetén egyet), mely maximalizálja a következő $\mathrm{likelihood}$-függvényt:

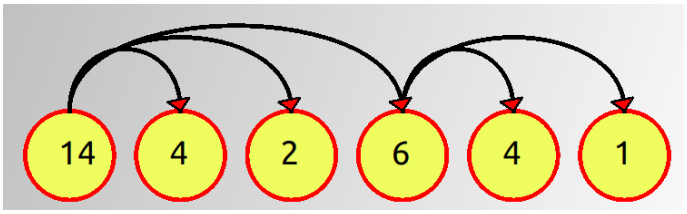
$$\prod_{i=0}^{n-2}\left(num[par[i]]-\sum_{j=0}^{i-1}num[j+1](\rho ar[j]=par[i])\right)$$

A képletben a num tömb jelöli az n darab számot, a $\mathrm{num}[i]$ pedig a számok közül az i indexűt. A par jelöli az élek startcsúcsának indexét. Mivel a 0 indexű csúcsba nincsen bemenő él, ezért a par tömb eggyel kevesebb elemű, mint a num tömb, és az indexek az eggyel nagyobb indexű csúcsra vonatkoznak. Például a fentebb írt négy él a következőképp írható le: $par[0]=0,\ par[1]=0,\ par[2]=1,\ par[3]=2$. Tehát a $par[i]$ jelöli az $(i+1)$. csúcsba bemenő él startcsúcsának az indexét. Másképp fogalmazva: $par[i]$ jelöli az $(i+1)$. csúcs szülőjének indexét. A zárójeles feltétel azt jelzi, hogy a gyerekcsúcs indexénél kisebb indexű csúcsok közül azokat vonjuk ki, melynek ugyanaz a szülője, mint a gyerekcsúcsnak. Azaz röviden megfogalmazva a tényező értéke a szülő értéke mínusz a szülő korábbi gyerekeinek összege. Az így kapott $n-1$ darab tényező szorzata a fához tartozó függvényérték, feltéve, ha az adott gráf megengedett.

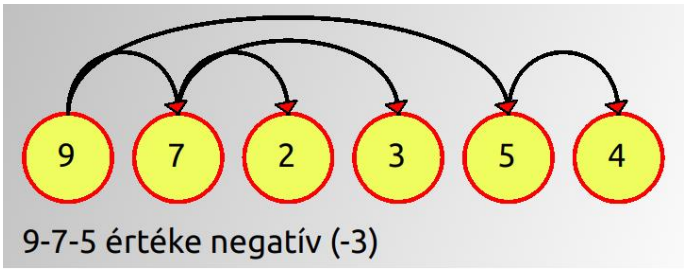
Példák: két megengedett, illetve két nem megengedett gráf



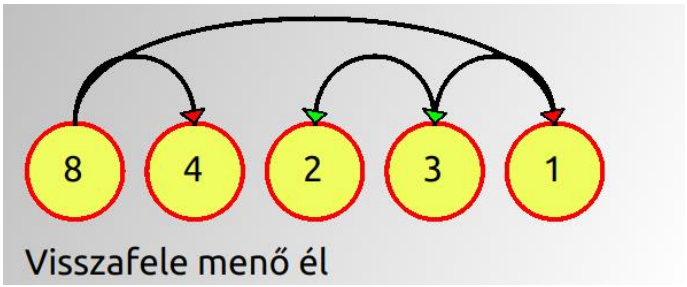
2. ábra: megengedett gráf



3. ábra: megengedett gráf



4. ábra: nem megengedett gráf



5. ábra: nem megengedett gráf

2.1.1 A probléma komplexitása

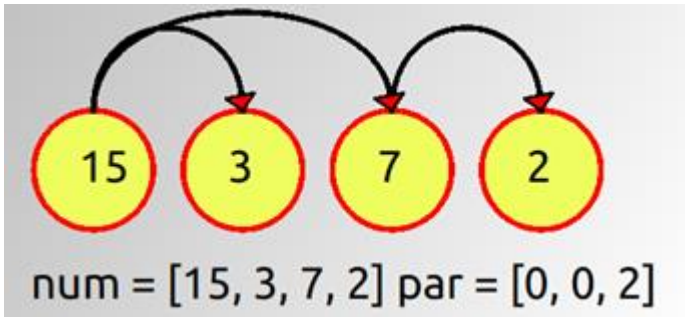
A probléma még ebben, a leginkább leegyszerűsített esetben is nehéz, az összes lehetséges fából $(n-1)!$ darab van. Ez könnyen kiszámolható, hiszen az i. csúcs esetén i lehetőségünk van a szülő kiválasztására, i értéke pedig végigmegy 1-től $(n-1)$-ig. A függvény szerinti optimális fa megtalálása feltehetően NP-nehéz probléma, ahogy azt egy hasonló algoritmikus kérdés esetén már bizonyították (El-Kebir, Oesper, Acheson-Field, & Raphael, 2015). A problémához tartozik egy másik kérdés is, mégpedig az, hogy egy adott input esetén hogyan dönthető el, hogy lehetséges-e megengedett fa építése.

A sejtés erre a kérdésre is az, hogy polinomiális költséggel nem dönthető el, hogy az $(n-1)!$ fából teljesíti-e valamelyik a megengedettségi feltételt.

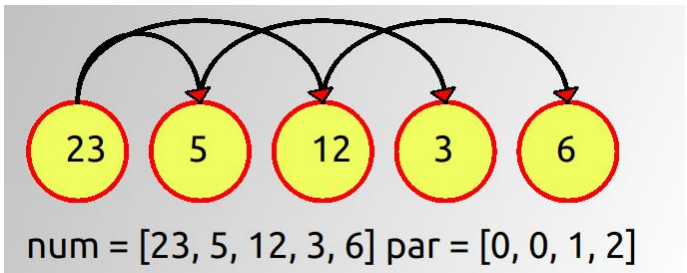
2.1.2 Informatikai reprezentáció

A problémát reprezentálni szeretnénk úgy, hogy programozási szempontokból egyszerűen kezelhető legyen, ehhez a tömbökkel történő megfogalmazást pontosítjuk. A lehetséges megoldásokat, azaz a speciális gráfokat két darab listával ábrázoljuk. Nevezzük az egyiket num-nak, ez a bemeneti számokat tartalmazza (numbers) és ennek hossza n, míg a másik lista neve legyen par, ami n-1 hosszú és az éleket jelöli. A num listát 0-tól $(n-1)$-ig indexeljük, míg a par listát 0-tól $(n-2)$-ig oly módon, hogy $par[i]$ jelöli az $(i+1)$. csúcsba bemenő él startcsúcsának indexét. Azért teszünk így, mert a 0. csúcsba nincs bemenő él.

Példák a reprezentációra:



6. ábra: két listához tartozó fa



7. ábra: két listához tartozó fa

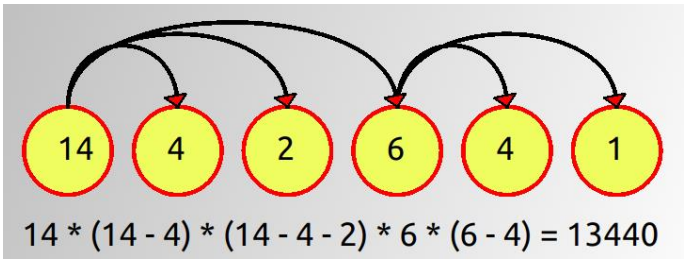
2.1.3 Alapvető algoritmusok

Az ebben a fejezetben szereplő algoritmusok szükségesek ahhoz, hogy az előbbiekben megadott problémával foglalkozzunk. Egyrészt szükségünk van arra, hogy adott fa esetén meghatározzuk a likelihood függvény értékét. Másrészt szükségünk van egy „brute force" megközelítésre, hogy ellenőrizhessük a heurisztikus algoritmusok helyességét, és legyen összehasonlítási alapunk. Először a likelihood-függvény kiszámítását kell implementáljuk, nevezzük ezt az algoritmust count_prob-nak. Az algoritmusnak két bemenete van, a korábban említett számokat tartalmazó num lista és az éleket jelölő par lista. Ez a két bemenet, az általunk definiált módon meghatároz egy gráfot, ami nem biztos, hogy megengedett. Célunk, hogy meghatározzuk a gráf likelihood-függvény szerinti értékét. Az algoritmus a likelihood-függvénynek megfelelően a num lista elemeinek veszi a szülőjét, majd kivonja belőle a szülő korábbi gyerekeit. Minden tényezőre ellenőrzi, hogy sérül-e a megengedettségi feltétel, ha igen, akkor -1-et ad vissza, ha egyszer sem, akkor a kiszámolt valószínűséget és a szülő vektort. Így a függvény kimenete egyszerre határozza meg, hogy megengedett-e a gráf (nem -1), és ha igen, akkor mi a hozzátartozó függvényérték. Futási idő: $O(n^2)$

```
1   algorithm count_prob(num, par):
2       float prob := 1
3       bool valid := True
4       for i from 0 to size(par)-1:
5           x=num[par[i]]
6           for j from par[i] to i-1:
7               if(par[j] = par[i]):
8                   x := x - num[j+1]
9           prob := prob * x
10          valid := (valid and (x - num[i+1]) >= 0)
11      if(not valid):
12          return -1
13      return prob, par
```

8. ábra: psedocode count_prob

Példa: egy fa likelihood-értékének számolása:



9. ábra: számolás

Természetesen az igazi feladat az, hogy a num lista ismeretében, hogyan határozzuk meg az optimális par listát. A legkézenfekvőbb megoldás a naív-algoritmus, mely egy maximumkeresést alkalmaz az összes lehetséges fa függvényértékein. Ez az előbb említett „brute force" megközelítés. Az algoritmus esetén a $0, 1, 2, .. n-2$ számhalmaznak (lehetséges szülők) vesszük az összes $n-1$ hosszú permutációját. Az összes lehetséges permutáción iterálunk végig, és adott permutáció esetén a kapott listát nevezzük per-nek. Ha teljesül, hogy a perben minden csúcshoz tartozó szülő kisebb indexű, mint a gyerek, akkor meghívjuk a count_prob algoritmust, és ha a visszaadott függvényérték jobb, mint az eddig megtalált maximum, ez lesz az új maximum. Ha nem teljesül az előző feltétel, akkor azt a permutációt elvetjük. Futási költség: $O(n!)$

```
1   algorithm naiv(num):
2       x <- -1, []
3       for per in permutations of (0..size(num)-1)
4           bool correct := True
5           for i from to size(per)-1:
6               if per[i] > i:
7                   correct := False
8           if correct:
9               y <- count_prob(num, per)
10              if y > x:
11                  x <- y
12      return x
```

10. ábra: pseudocode naiv

2.1.4 Korábbi heurisztika

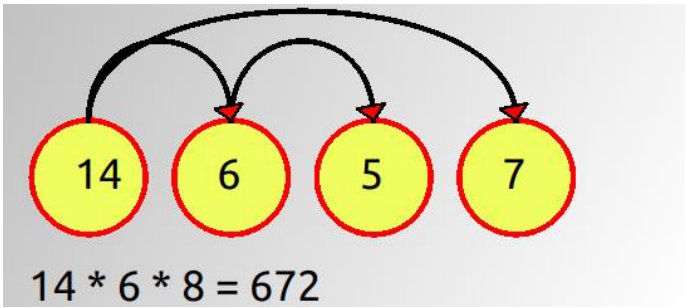
Mivel az említett naiv-algoritmus $(n-1)!$ fa esetén számolja ki a függvényértéket, ezért a műveletigénye $O(n!)$. Nyilvánvaló, hogy nagyobb elemszámú num lista esetén a feladat a futási idő miatt nem lesz hatékonyan megoldható. A témában született korábbi heurisztikát nevezzük mohó-algoritmusnak, amely az optimális fát közelíteni szeretné (Ismail & Tang, 2019).

Az algoritmus, mint általában a mohó algoritmusok, a lokálisan legjobbnak tűnő szülőt választja ki. Azokat a csúcsokat vizsgálja index szerinti sorrendben, melyeknek szülőt szeretnénk választani. A lehetséges szülőkre kiszámolja a likelihood-függvénynek megfelelően elérhető tényezőket, és ezen értékek közül mindig a maximálisat választja, kivéve, ha már a maximális is kisebb lenne, mint a célcsúcs értéke, ezáltal sérülne a megengedettségi feltétel. Ekkor nem talál megoldást. Egyszerűbben megfogalmazva, a kezdeti lista i. tagjának úgy választ szülőt, hogy a $0,1,\ldots,i-1$ indexű lehetséges szülőkre figyelembe veszi a szülő korábbi gyerekeit, és ezáltal a lokálisan elérhető maximális szorzótényezőt tudja választani. Az algoritmus futási költsége polinomiális, így nagyobb elemszámú esetekre is futtatható. Hátránya, hogy a fát csupán közelíti, és nem feltétlenül a legjobb megoldást találja meg. További hátránya, hogy lehetséges, hogy nem talál megengedett megoldást úgy, hogy egyébként létezik az adott adatsorra.

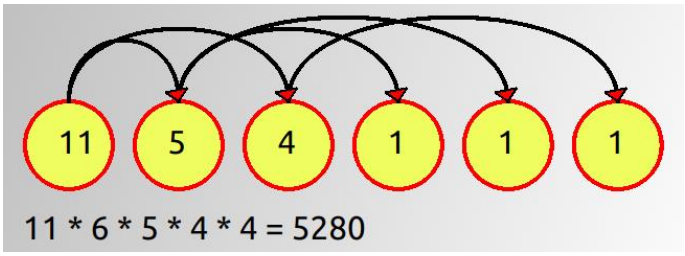
Példák: a mohó algoritmus és hátrányai

```
1   algorithm simple_greedy(num):
2       par <- [-1, -1, ..]
3       for i from 0 to size(num)-1:
4           float best := 0
5           int bestind := -1
6           for j from 0 to i-1:
7               float choice := num[j]
8               for k from j+1 to i-1:
9                   if par[k-1] = j:
10                      choice := choice - num[k]
11              if choice > best:
12                  best := choice
13                  bestind := j
14          if best < num[i]:
15              return -1
16          else:
17              par[i-1] := bestind
18      return count_prob(num, par)
```

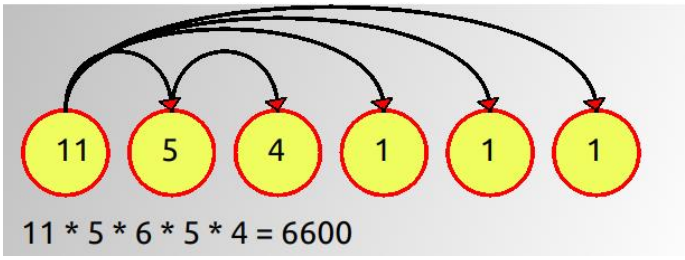
11. ábra: pseudocode mohó



12. ábra: megoldható, de a mohó nem találna megoldást



13. ábra: mohó által talált fa



14. ábra: ténylegesen optimális fa

Az egyik probléma kiküszöbölhető az algoritmus módosításával. A módosítás lényege a visszalépéses keresés, azaz ha olyan helyzet áll fenn, amikor a lokálisan elérhető maximális szülő lehetőséggel is sérülne a megengedettségi feltétel, akkor visszalép, és az előző célcsúcsra a második (avagy az aktuálisnál eggyel rosszabb) lehetőséget választja szülőnek. Ha ennél a választásnál is sérülne a megengedettségi feltétel, akkor még egyet visszalép, egészen addig, amíg lehetséges megengedett választás. Ez a javítás megoldja annak a problémáját, hogy megoldható bemenetre nem kapunk megoldást, azonban - mivel a sejtés az, hogy nem létezik erre polinomiális algoritmus - a futási idő rossz esetben $O(n!)$. A legjobb esetben elérhető az $O(n^2)$.

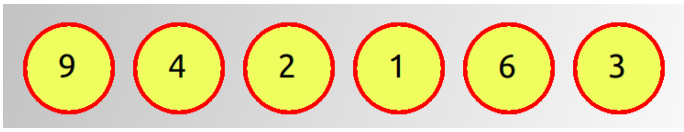
2.1.5 Javaslatok

Az új gondolat, ami alapján szemléltem a problémát, a megengedettségi feltétel betartása. Egy adott csúcs értéke mindig legalább annyi kell legyen, mint a gyerekei értékeinek összege. Ez magában foglalja azt, hogy mikor szülőt választunk az i. csúcsnak, akkor nem csak az indexek értéke szűkíti le a lehetőségeinket, hanem az is, hogy azok a csúcsok, melyek i-nél kisebb indexűek, nagyobb értékűek-e, mint az i. csúcs értéke. Ebből kifolyólag az 1 indexű csúcs szülője csak a 0 indexű lehet, ha pedig azt nézzük, hogy a második legnagyobb csúcsba honnan mehet él, akkor annak az élnek a startcsúcsa csak a maximális értékű csúcs lehet.

Az új megszorítást nézve vannak olyan elemek a par vektorban melyek egyértelműen meghatározhatók. Például a $par[0]$ értéke mindig 0 lesz. (Az 1 indexű csúcs szülője a 0

indexű csúcs) Az ilyen egyértelmű kapcsolatok miatt lesznek olyan input adatok melyekre ránézésre el tudjuk dönteni, hogy a feladat nem megoldható.

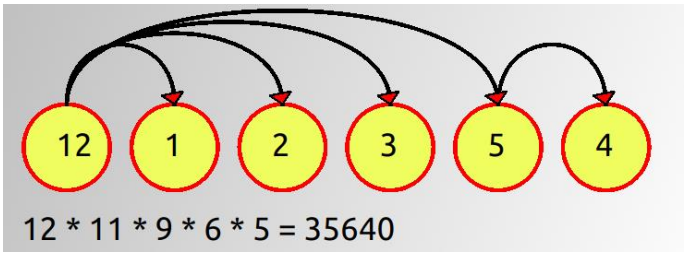
Példa: polinomiális költséggel kiderül, hogy nem megoldható



15. *ábra: nem megoldható*

Fontosabb észrevétel, hogy a döntéseink függvényében változtathatjuk a num listát a futási időben. Ha például a num lista legnagyobb és második legnagyobb száma között behúzzuk az élt, amit biztosan tudunk, akkor a legnagyobb számot csökkenthetjük a második legnagyobb szám értékével. Ezáltal tudjuk, hogy abból a csúcsból már csak a maradék értéknél kisebb értékű csúcsba indíthatunk élt. Természetesen, amikor a likelihood-függvény értékét számoljuk ki, akkor az eredeti num listával kell számoljunk. Az elvégzett módosításokkal lehetséges, hogy a teljes fa meghatározható polinomiális költséggel.

Példa: megoldható, csak így lehet megengedett a fa



16. *ábra: megoldható polinomiálisan*

2.1.6 Precheck és rekurzió

A következőkben a 2.1.5-ös észrevételeim segítségével konstruálok két új algoritmust. Nevezzük az első algoritmust prechecknek, melynek lényege, hogy meghatározza a gráf egy részét, azokat az éleket, melyek egyértelműek a megengedettségi feltétel miatt. Ha valamelyik csúcsba mutató élnek nem az lenne a startcsúcsa, mint amit az algoritmus ad, a többi választástól függetlenül nem lehetne megengedett a teljes fa. Az algoritmus lényege, hogy minden csúcsra megszámolja, hogy hány olyan csúcs van, ami a listában előtte van és nagyobb nála. Ennek költsége $O(n^2)$. Tehát adott gyerekcsúcsra kiválasztja a lehetséges szülőket azzal, hogy a num listában megnézi a kisebb indexűek közül a nagyobb értékűeket. Ha nincs ilyen, akkor az algoritmus befejeződik, és -1-et ad vissza. Ekkor a feladat nem megoldható. Ha pontosan egy van, az azt jelenti, hogy biztosan az

az egy lesz a szülő. Ekkor ezt jelezzük a par vektorban, és a szülő értékét csökkentjük a gyerek értékével. A lista minden elemére (kivéve az elsőt) megcsináljuk ezt a kiválasztást, és elvégezzük a módosításokat. Addig ismételjük az algoritmust, amíg egyszer nem fedezünk fel új élt. Ha csak egyszer végeznénk el a fenti lépéseket, lehetséges, hogy nem találnánk meg olyan csúcsnak a szülőjét, ami egyértelművé vált az előző körös – későbbi - változások után. A nem ismert helyeken a lehetséges szülőket tartalmazó lista hasznos lesz az általánosabb esetekre. Az algoritmus költsége, ha minden körben egy élt fedezünk fel: $O(n^3)$. Ekkor azonban a teljes gráfot meg tudja határozni.

```
algorithm line_pre_check(num, par):
    for i from 0 to len(par)-1:
        if par[i] != -1:
            vec[par[i]] := vec[par[i]]-vec[i+1]
    bool modified := True
    while modified do:
        list parents
        modified := False
        for i from 1 to size(num)-1:
            if par[i-1] != -1:
                parents.append([par[i-1]])
            else:
                int options := 0
                int parent := -1
                list paropt
                for j from 0 to i-1:
                    if num[j] >= num[i]:
                        options : =options + 1
                        parent := j
                        paropt <- j
                parents <- paropt
                if options = 0:
                    return -1
                if options =  1:
                    par[i-1] := parent
                    num[parent] := num[parent] - num[i]
                    modified := True
    return parents
```

17. ábra: pseudocode precheck

Általánosságban a precheck algoritmussal nem találjuk meg a teljes fát. A maradék élek meghatározására egy rekurzív algoritmust implementálok, mely ugyanazon az elven működik, mint az előző algoritmus. A rekurzív algoritmus az eredeti num listát, a részlegesen kitöltött par listát (ismeretlen helyeken -1-gyel) és a par lista szerint módosított num listát kéri input adatnak. Az eredeti num lista szerint elkészíti az indsort

nevű listát, melyben a $0,1..,n-1$ számok vannak a hozzájuk tartozó num[i] érték szerint rendezve. Ebben a sorrendben fogjuk kitölteni a par listát, hiszen, ha a nagy értékű csúcsoknak választunk előbb szülőt, azzal a lehető legjobban csökkentjük a csúcsok értékét, ezzel pedig a legtöbb esetet zárjuk ki futás közben. Az algoritmus az indsort lista sorrendjében keresi a szülőket és ugyanúgy számolja meg a lehetséges szülőket, mint a korábbi precheck algoritmus. Ha a par listában már van az indexhez tartozó szülő, akkor nem módosítunk. Ha még ismeretlen a szülő, akkor a nagyobb számok között keresem a kisebb indexűeket, és ha ilyeneket találok, akkor úgy veszem mindegyikre, mintha az lenne az igazi szülő, és levonom a startcsúcs értékéből a célcsúcs értékét, valamint jelölöm a kapcsolatot a par vektorban. Az eredeti num listával, és a módosított num és par listákkal hívom meg rekurzívan a függvényt az összes lehetséges szülőre - az első szülő nélküli gyerekcsúcs esetén - és a rekurzív függvényhívások maximumát adom vissza. Ezáltal az algoritmus felderíti az összes fát, de futás közben eliminál eseteket a levonások miatt. A rekurzió „mélyén”, amikor teljesen kitöltött par listával hívom meg az algoritmust, akkor a visszaadott érték már természetesen a count_prob algoritmussal számított függvényérték lesz. A rekurzió legrosszabb esetben $O(n!)$ költségű. Legjobb esetben lehetséges az $O(n^2)$ költség.

```
algorithm simple_recursive_alg(origvec, num, par, indsort):
    maxiprob <- -2, []
    for i from 0 to size(indsort)-1):
        if par[indsort[i]-1] = -1:
            maxiprob= <- -1, []
            for j from 0 to i-1:
                if(indsort[j] < indsort[i]) and (num[indsort[j]] >= num[indsort[i]]):
                    num2 <- num
                    par[indsort[i]-1]=indsort[j]
                    num2[indsort[j]]=num2[indsort[j]]-num2[indsort[i]]
                    x <- simple_recursive_alg(origvec, num2, par, indsort)
                    if x > maxiprob:
                        x <- maxiprob
            break
    if maxiprob = -2: #nem -1, tehát teljes a par lista
        maxiprob <- count_prob(origvec, par)
    return maxiprob
```

18. ábra: egyszerű eset rekurzió

2.1.7 Egyszerű eset eredmények

A precheck+rekurzív algoritmus előnye, hogy nem közelíti az optimális fát, hanem mindig azt találja meg (több maximális függvényértékű esetén egyet közülük). A precheck algoritmussal megtalált, részlegesen kitöltött par vektorral hívom meg a rekurzív eljárást, ami ezután az ismeretlen helyeken próbálja meg a lehetséges szülőket.

Az algoritmus megnézi az összes esetet, de olyan „irányba” nem próbálkozik, ami sértené a megengedettségi feltételt. A továbbiakban hivatkozott xls, és sims fájlok elérhetők itt: https://people.inf.elte.hu/b4zppx/tdk.html

Ezen felül 100 db egyszerű, szimulált, 8 klónos adatsoron tesztelve, a talált fa függvényértéke minden esetben pontosan megegyezett a naív-algoritmus által talált fa függvényértékével (azaz az optimális fával). A teljes teszt nagyjából 0.31 másodperc alatt futott le, összehasonlítva a naív esetén tapasztalt 70 másodperccel (results1.xls, simple8.sims).

Precheck+recursion				Naiv		
Function Value	Parent Vector	Running Time		Function Value	Parent Vector	Running Time
0.00802958285	[0, 0, 2, 0, 0, 0, 0]	0.00166893005		0.00802958285	[0, 0, 2, 0, 0, 0, 0]	0.727602005
0.0041500105	[0, 0, 0, 3, 3, 5, 2]	0.00124406815		0.0041500105	[0, 0, 0, 3, 3, 5, 2]	0.72075510025
0.00251786306	[0, 1, 0, 0, 0, 2, 5]	0.00454998016		0.00251786306	[0, 1, 0, 0, 0, 2, 5]	0.70829701424
0.00488097103	[0, 0, 1, 0, 1, 1, 1]	0.00389885902		0.00488097103	[0, 0, 1, 0, 1, 1, 1]	0.67992997169
0.00597751226	[0, 0, 2, 3, 3, 2, 6]	0.00129199028		0.00597751226	[0, 0, 2, 3, 3, 2, 6]	0.6840569973
0.00227611893	[0, 1, 0, 1, 4, 0, 0]	0.00563406944		0.00227611893	[0, 1, 0, 1, 4, 0, 0]	0.67373681068

19. ábra: results1.xls részlet

Egy másik, kicsit nagyobb, 100 db-os egyszerű, 10 klónos adatsoron futtatva a korábbi heurisztikát, a visszalépés nélküli mohó algoritmust, azt látjuk, hogy az az eljárás 36 esetben nem talál megengedett megoldást. A maradék 64 esetben a rekurzív algoritmus függvényértékei átlagosan 110%-kal nagyobbak, mint a mohó algoritmus esetén. A precheck által felfedezett élek aránya $231/900$, nagyjából 26%-os (results5.xls, simple.sims).

2.2 Rendezett általános eset

Az eddig vizsgált modell nem vette figyelembe, hogy a valóságban a már létező klónok aránya változik a populáción belül. Ahogy már a 2. fejezet bevezető részében is említettem a bemeneti adatsor konkrét mérési eredmények esetében egy mátrix. Ebben a fejezetben olyan modellt állítunk fel, mely jobban reprezentálja a valós helyzetet. A kiindulási alapul vett (Ismail & Tang, 2019) cikk főként ezen a modellen alapuló szimulációkon tesztelte algoritmusait. Ha az egyszerű esetben úgy tekintünk a bemeneti tömbre, mint egy $1 \times n$-es mátrix, akkor az, hogy az oszlopok a klónok mennyiségét jelentik továbbra is igaz. Azzal az információval bővítjük a modellünket, hogy mikor az i. oszlopban lévő klónt észleljük, addigra a korábbiaknak (kisebb indexű oszlopok) az aránya változott a populáción belül. Tehát nem számolhatjuk a becslésünket a szülőre vonatkozóan az aktuális sorból, hanem az előző sorból kell számoljuk, hiszen az új

mutáció óta változott a korábbi klónok egymáshoz viszonyított aránya. Ez abból a megfigyelésből fakad, hogy a populáció nem csak új klónokkal bővül időről időre, hanem a már meglévők szaporodnak, némelyek jobban, míg mások kevésbe. Ezáltal a mátrix sorai az egymás utáni időpontokban történő mérések, és a sor elemei a klónok aránya az adott időpillanatban.

2.2.1 A feladat meghatározása

Ezek alapján a probléma bonyolultabb változata, még mindig egyszerűsítve a valós helyzetet így adható meg:

- A bemenő adatok egy alsó háromszög mátrixot alkotnak. Eszerint minden sorban eggyel több a nemnulla mezők száma. Legyen a mátrix dimenziója $n \times n$-es.
- Az oszlopok a „csúcsok”, tehát a mátrix oszlopai között szeretnénk a speciális gráfot felépíteni, ahol hasonlóan van $0, 1, .., n-1$ sorszámú oszlop, avagy csúcs.
- A megengedettségi feltétel lényege, hogy akkor megengedett a mátrixhoz tartozó par vektor, ha minden sort nézve a par vektorral, az egyszerű esetben megengedett fát kapnánk. Tehát a feltétel itt is az, hogy a szülő értéke nagyobb, mint a gyerekei értékeinek összege, és ennek minden időpillanatban (sorban) teljesülnie kell.
- A likelihood-függvény továbbra is azon a feltevésen alapul, hogy a mutációk véletlenszerűen történnek és annak az esélye, hogy a klón az új klón szülője, a populáción belüli arányával jellemezhető. Ez alapján az általánosítás az, hogy az új „csúcs” tényezőjét az előző sorból számoljuk, úgy ahogy az egyszerű esetben is tettük.

Lent látható a likelihood-függvény képlete általános esetben, ahol a par vektor az eddigi par vektor oszlopokra való általánosítása, az F a bemeneti mátrix, $F_{i,j}$ a mátrix eleme.

$$\prod_{i=0}^{n-2} \left(F_{i,par[i]} - \sum_{j=0}^{i-1} F_{i,(j+1)} (par[i] = par[j]) \right)$$

Az eddig megadott algoritmusokat általánosítjuk, a bővebb kérdéskörhöz igazítva. A következő általánosítások logikusan lesznek igazítva az összetettebb problémához.

2.2.2 Alapalgoritmusok

A probléma reprezentációja hasonló lesz, a par lista ugyanaz marad, de a régi num lista helyett mostantól egy matrix nevű mátrixunk lesz. Természetesen a likelihood-függvényt kiszámító algoritmussal kell kezdeni, melynek célja ugyanúgy, egy már teljes par listával történő függvény kiértékelés. Nevezzük az eljárást general_count_prob-nak. Az algoritmus először minden sorra ellenőrzi, hogy megengedett-e a megadott fa, ehhez minden sorra meghívja az egyszerű esetben használt count_prob algoritmust. Ha valamelyik sor nem az, akkor -1-et ad vissza. Ezután az egyes tényezőket hasonlóan számolja ki, ahogy tette az egyszerű esetben, azzal a különbséggel, hogy a szülő értékét, és a szülő korábbi gyerekeinek értékét az előző sorból veszi.

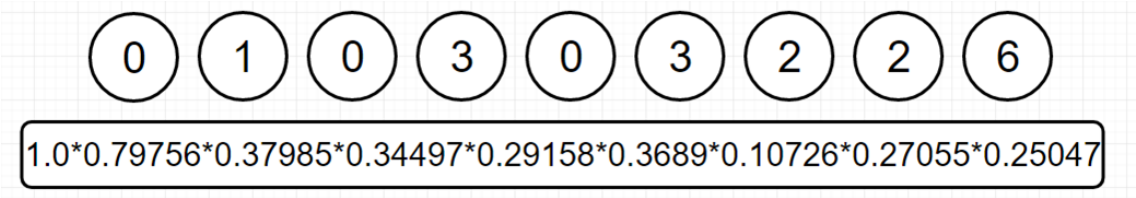
```
1   algorithm general_count_prob_good(matrix, par):
2       for i in rows of matrix:
3           if count_prob( i , par ) = -1:
4               return -1
5       float prob := 1
6       for i from 1 to size(matrix)-1:
7           float actprob = matrix[i-1][par[i-1]]
8           for j from 0 to i-2:
9               if par[j] = par[i-1]:
10                  actprob := actprob - matrix[i-1][j+1]
11          prob := prob * actprob
12      return y, par
```

20. ábra: pseudocode general_count_prob

Példaszámítás:

```
1 1.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
2 1.00000 0.79756 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
3 1.00000 0.62015 0.29453 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
4 1.00000 0.32287 0.20563 0.34497 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
5 1.00000 0.38405 0.12751 0.32437 0.17157 0.00000 0.00000 0.00000 0.00000 0.00000
6 1.00000 0.15137 0.07551 0.45416 0.08526 0.09032 0.00000 0.00000 0.00000 0.00000
7 1.00000 0.40087 0.10726 0.42465 0.09439 0.05639 0.03480 0.00000 0.00000 0.00000
8 1.00000 0.48610 0.29662 0.44782 0.32906 0.06231 0.11424 0.02607 0.00000 0.00000
9 1.00000 0.48065 0.32447 0.31153 0.05669 0.18949 0.25047 0.04280 0.21884 0.00000
10 1.00000 0.34699 0.32684 0.45180 0.22860 0.08554 0.14420 0.10251 0.21277 0.03495
```

21. ábra: példamátrix



22. ábra: par lista és számítás

A naív-algoritmus általánosításának lényege a likelihood függvény kiértékelését végző algoritmus megváltoztatása. Ahogy az egyszerű esetben alkalmazott naív algoritmusnál, úgy itt is, a permutációk elkészítése, helyességének ellenőrzése után számítjuk ki a permutációhoz tartozó likelihood-függvény értéket. A különbség annyi, hogy a számítást a general_count_prob-bal végezzük.

```
1   algorithm naiv_general(matrix):
2       x <- -1, []
3       for per in permutations of (0..(matrix dimension)-1)
4           bool correct := True
5           for i from to size(per)-1:
6               if per[i] > i:
7                   correct := False
8           if correct:
9               y <- general_count_prob(matrix, per)
10              if y > x:
11                  x <- y
12      return x
```

23. ábra: pseudocode general naiv

A többször hivatkozott (Ismail & Tang, 2019) cikkben megjelenő heurisztika, vagyis a mohó-algoritmus alapvetően általános esetre készült. Az általánosítás erre az algoritmusra is a lehető legkézenfekvőbb módon történik, azaz az új oszlopokra az elérhető tényezőket a szülő és a korábbi gyerekek értékeivel számolja ki. A különbség itt is az, hogy mindezt az előző sor alapján csinálja, azaz az i. oszlop esetén az (i-1). sor szerint. Tehát itt is úgy választ szülőt, hogy belekalkulálja a szülő korábbi gyerekeit, és ezáltal a lokálisan elérhető maximális szorzótényezőt tudja választani.

Ugyanúgy, mint az egyszerű esetben, elmondható, hogy az algoritmus futási költsége polinomiális, így nagyobb elemszámú esetekre is futtatható. Hátránya, hogy a fát csupán közelíti, nem feltétlenül a legjobb megoldást találja meg. További hátránya, hogy lehetséges, hogy nem talál megengedett megoldást úgy, hogy egyébként létezik az

adott adatsorra. Ez ismét kiküszöbölhető a visszalépéssel, ami a legrosszabb esetben exponenciális költséget eredményez.

```
algorithm general_greedy(matrix):
    par <- [-1, -1, ..]
    for i from 0 to matrix dimension-1:
        float best := 0
        int bestind := -1
        for j from 0 to i-1:
            float choice := matrix[i][j]
            for k from j+1 to i-1:
                if par[k-1] = j:
                    choice := choice - matrix[i][k]
            if choice > best:
                best := choice
                bestind := j
        if best = 0:
            return -1
        else:
            par[i-1] := bestind
    return general_count_prob(matrix, par)
```

24. ábra: pseudocode mohó általános

2.2.3 Precheck általánosítása

Ennek az algoritmusnak a működésénél látni fogjuk, hogy miért volt ideális az összes lehetséges szülőt visszaadni az egyszerű esetben. A sorokra teljesülnie kell az egyszerű esetben látott megengedettségi feltételnek. Emiatt minden sorra meghívhatjuk a korábbi line_precheck algoritmust, melynek segítségével megkapjuk, hogy az oszlopokra milyen szülő lehetőségeink vannak. Ha ezt megtesszük akkor lesz n darab listánk, melyek listákat tartalmaznak. (lehetséges, hogy egy elemű lista helyett csak egy számot, hiszen akkor egyértelműen felfedezte a line_precheck) Tehát az objektum tekinthető egy listákat tartalmazó mátrixnak. Ezeket az adatokat kell összefésülni az oszlopok szerint. A sorokra való meghívást a matrix_pre_check hajtja végre, az összefésülést pedig a full_pre_check. Azaz halmazként tekintünk a listákra, és minden oszlop esetén a kapott n db listának (amelyek az üres részeken, azaz a

```
algorithm matrix_pre_check(matrix, par):
    list optionalparents
    for i in matrixcopy:
        lineoptions <- line_pre_check(i,par)
        optionalparents <- add lineoptions
    return optionalparents
```

25. ábra: pseudocode matrix_precheck

felsőháromszögben, nem tartalmaznak lényegi információt) vesszük a közös metszetét. Azokon a helyeken, ahol ez a közös metszet egy darab elemet tartalmaz, ott annak az oszlopnak biztosan ismerjük a par vektorban az indexéhez tartozó értéket. Ha van oszlop amire a metszet üres, akkor a feladat nem megoldható. Ezután vesszük ezeket a helyeket, ahol csak egyet tartalmazott a metszet, ezeket jelöljük par vektorban, és újból lefuttatjuk az eddigieket. Itt is addig kell ismételni, amíg egy körben nem történik változás.

```
1   algorithm full_pre_check(matrix, par):
2       parcopy=deepcopy(par)
3       newpar=[]
4       listoflists <- matrix_pre_check(matrix,par)
5       for i from 0 to size(par)-1:
6           intersect <- set of listoflists[0][i]
7           for j in listoflists:
8               intersect <- intersect.intersection(j[i])
9           if(len(intersect)==1):
10              newpar.append(list(intersect)[0])
11          else:
12              newpar.append(-1)
13      if(newpar.count(-1) != parcopy.count(-1)):
14          newpar <- full_pre_check(matrix,newpar)
15      return newpar
```

26. ábra: pseudocode full_precheck

2.2.4 Rekurzió általánosítása

A rekurzió az általános esetben is a precheck után részlegesen felfedezett fával kezdődik. A bemenet annyiban módosul, hogy az eredeti lista helyett, az eredeti mátrixot adjuk át, és a módosított lista helyett pedig a részlegesen megkapott par vektor szerint módosított utolsó sort. Lényegében az utolsó sorra végzünk egy egyszerű esetben látott rekurziót, azonban mikor itt az algoritmus „mélyére” érünk, akkor az általános esetben definiált likelihood-függvény szerint számoljuk ki az eredeti mátrix és par listához tartozó függvényértéket.

```
algorithm general_recursive(origmatrix, vec, par, indsort):
    maxiprob <- -2, []
    for i from 1 to size(indsort)-1:
        if par[indsort[i]-1] = -1:
            maxiprob <- -1,[]
            for j from 0 to i-1:
                if (indsort[j] < indsort[i]) and (vec[indsort[j]] >= vec[indsort[i]]):
                    par2[indsort[i]-1]=indsort[j]
                    vec2[indsort[j]]=vec2[indsort[j]]-vec2[indsort[i]]
                    x <- general_recursive(origmatrix, vec2, par2, indsort, algParam)
                    if x > maxiprob:
                        maxiprob <- x
            break
    if maxiprob = -2:
            maxiprob=general_count_prob_good(origmatrix, par)
    return maxiprob
```

27. ábra: pseudocode general_recursive

2.2.5 Általános eset eredmények

A precheck a rekurzív algoritmussal együtt az általános esetben is az optimális fát találja meg, az általánosítás lényege csupán az eltérő számítás a likelihood-függvény értékére. Emellett 100 darab 10 klónos rendezett általános esetben minden alkalommal pontosan a naív kereséssel egyező értéket adta. A teljes teszt futási ideje nagyjából 4 másodperc volt a precheck+rekurzió esetén, míg 694 másodperc a naív esetben (results2.xls, square.sims). Nagyobb adatsor esetén a precheck algoritmus jól teljesít, 15 klónos rendezett általános esetben, ugyanúgy 100 tesztesettel, az összes ismeretlen $100 * 14 = 1400$ ismeretlen élből 931-et azonosított, azaz az élek nagyjából kétharmadát. A teljes teszt futásideje nagyjából 0.28 másodperc volt (results6.xls, general_15o.sims).

Ennél is nagyobb adatsor esetén a precheck algoritmus a 20 klónos rendezett általános esetben, ugyanúgy 100 tesztesettel, a kérdéses $100 * 19 = 1900$ ismeretlen élből 1263-at azonosított, azaz itt is az élek nagyjából kétharmadát. A teljes teszt futásideje 0.68 másodperc volt (results7.xls, general_20o.sims).

Parent Vector	Running Time
[0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]	0.023049116
[0, 1, 1, 1, 3, 2, 4, 0, 6, 2, 6, 0, 8, -1, -1, -1, -1, -1, -1]	0.011543036
[0, 1, 2, 0, 1, 5, 0, 0, 4, 6, 10, 7, 10, 13, 10, -1, -1, -1, -1]	0.007432938
[0, 1, 0, 3, 2, 1, 4, 1, 4, 9, 7, 10, 3, 8, -1, 14, 9, -1, -1]	0.00582099
[0, 1, 0, 3, 1, 5, 6, 2, 4, 8, 6, 0, -1, -1, -1, -1, -1, -1, -1]	0.00493598
[0, 0, 2, 2, 0, 3, 1, 4, 5, 7, 10, 2, 3, 4, 9, 14, -1, -1, -1]	0.007513046
[0, 1, 0, 0, 2, 4, 2, 2, 8, 2, 5, 5, 11, -1, -1, -1, -1, -1, -1]	0.006566048

28. ábra: results7

A rekurzív algoritmus futási ideje sajnos a klónok számának növelésével drasztikusan nő, az előbb említett 15 klónos adatsoron csak 90 bemenetet tud legfeljebb 15 másodperc alatt megoldani. A cél, a mérési adatsorokból az optimális fa építése, számításigényt tekintve még nehezebb (2.3 fejezet), így ez alapján feltételezhető, hogy az optimális fa megtalálásáról le kell mondanunk. Az, hogy közelítésre kell hagyatkoznunk -mivel a likelihood-függvény is becslés- lehet, hogy nem igazi probléma. A precheck algoritmus viszont remekül kombinálható lenne a korábbi heurisztikával.

2.3 Rendezetlen általános eset

Ebben a fejezetben a rendezetlen általános esettel foglalkozom. Ha méréseket végzünk egy baktériumpopuláción, akkor a feladat bonyolultabb a rendezett általános esethez képest. Az új információ, amit felhasználunk az az, hogy egy új időpontban több új klónt is felfedezhetünk, hiszen méréseink nem feltétlenül elég gyakoriak. Tehát a mátrix több oszloppal rendelkezik, mint sorral, hiszen kisebb a mérési időpontok száma, mint a klónok száma. Azok között a klónok között, melyeket egy időpontban fedezünk fel, nem ismert a sorrend. Tehát nem tudjuk, hogy arra van-e esély, hogy az egyik a másiknak a szülője, vagy fordítva. Ezért ideális esetben az összes lehetséges rendezést vizsgálnunk kellene, mindegyik esetén felállítani az optimális fát, és ezek közül venni a legnagyobbat. Az, hogy a klónok időben hogyan jelenhettek meg egymáshoz viszonyítva, csak részben adott. Ha az egyiket az $i.$, míg a másikat az $(i+1)$. időpontban fedeztük fel, akkor kettejük között a sorrend egyértelmű. Ha viszont van négy klón, amiket az $i.$ időpontban észleltünk először, akkor ez $4! = 24$ lehetséges sorrendben történhetett. Emiatt az új feladat kitűzésénél gondolnunk kell arra, hogy nem csak a bemeneti mátrixot vizsgáljuk.

2.3.1 A feladat meghatározása

A fentiek alapján a tényleges probléma, a valódi mérési adatok megoldása esetén így írható fel:

- A bemenő adatok egy mátrixot alkotnak. Eszerint minden sorban legalább eggyel több a nemnulla mezők száma. A nulla értékű mezők a sorok végén találhatók. Legyen a mátrix dimenziója $n \times m$-es.

- Az oszlopok a „csúcsok”, tehát a mátrix oszlopai között szeretnénk a speciális gráfot felépíteni, ahol hasonlóan van $0, 1, .., m-1$ sorszámú oszlop, avagy csúcs.
- A megengedettségi feltétel lényege, hogy akkor megengedett a mátrixhoz tartozó par vektor, ha minden sort nézve a par vektorral, az egyszerű esetben megengedett fát kapnánk. Tehát a feltétel itt is az, hogy a szülő értéke nagyobb, mint a gyerekei értékeinek összege, és ennek minden időpillanatban (sorban) teljesülnie kell.
- A likelihood-függvény továbbra is azon a feltevésen alapul, hogy a mutációk véletlenszerűen történnek és a klónnak az esélye, hogy az új klón szülője, a populáción belüli arányával jellemezhető. Ez alapján az általánosítás az, hogy az új csúcsok közül az elsőnek a tényezőjét az előző sorból számoljuk, úgy ahogy az egyszerű esetben is tettük. Ha több, mint egy új csúcs van egy sorban, akkor a másodiktól kezdve az aktuális sorból számítjuk a tényezőt, így modellezhetjük azt, hogy egy új csúcs egy másik új csúcs szülője.
- Az összes lehetséges sorrendben vizsgáljuk a mátrixot a fentiek szerint. Az első pillanatban egy klón van, ezután a további pillanatokban k_2, k_3 … k_n csúcs jelenik meg. Tudjuk, hogy $1 + k_2 + .. + k_n = m$ azaz a soronkénti új csúcsok száma megadja az összes csúcsot, és az összes lehetséges sorrend száma: $k_2! \cdot k_3! \cdot \cdots k_n!$. Ennyiféle mátrix esetén vett optimális fát kell összehasonlítanunk.

2.3.2 A probléma komplexitása

A probléma jelentősen bonyolultabb, mint a rendezett általános eset esetén, a lehetséges mátrixok számának nagyságrendje: $O(n!)$. A kiindulási alapul vett (Ismail & Tang, 2019) cikk két algoritmust implementál a mátrixok meghatározására, az egyik egy „brute-force” jellegű, míg a másik mohó-heurisztika. A „brute-force” megközelítés természetesen az összes lehetséges mátrixot felépíti, és azokra kell aztán az optimális fát meghatározni. Itt valóban exponenciális költségű a lehetséges rendezések előállítása. A mohó-heurisztika lényege, hogy a lehetséges permutációkat lokálisan választjuk ki. Egy időpontban az újonnan megjelenő csúcsok között az összes permutációra felépíti a fát a rendezett esetben látott mohó algoritmussal, és az így kapott szülőket választja a teljes fára vonatkozóan. Ez helyenként kizárja a korábbi időpontban megjelenő klónokat. A

permutációk keresésének költsége jó esetben $\sum_{i=2}^{n} k_i!$. A visszalépés miatt azonban legrosszabb esetben lehetséges a $\prod_{i=2}^{n} k_i!$ költség is.

2.3.3 Algoritmusok

A rendezett általános esetben definiált algoritmusaink továbbfejlesztése csupán a bemeneti mátrix átalakításával történik. A meghatározott rendezések esetén, az egy időpontban, azaz egy sorban felfedezett új klónok (oszlopok) részt kell módosítanunk. Ha k nemnulla oszlop jelenik meg az i. időpontban, akkor az $(1.4.2)$-ben említett algoritmusok adhatnak köztük valamifajta permutációt. Egy ilyen permutáció segítségével alakítjuk a mátrixunkat négyzetes mátrixra. Az i. sorban csupán a permutáció szerinti első új oszlop értékét írjuk, a további oszlopokba nulla értékeket helyezünk. Ezután beszúrunk egy $(i+1)$. sort, a sor értéke az i. sor értékével egyezik meg minden elemre, kivéve a permutáció szerinti második oszlopot, melynek bemeneti i. soros értékét írjuk az $(i+1)$. sorba. Ezután ezt folytatva, az említett k új oszlop esetén összesen $k-1$ sort szúrunk be az i. sor után és alsóháromszög alakú mátrixot csinálunk a sorrészből, majd ezt elvégezve az összes blokkra az egész mátrixból. A kialakult négyzetes mátrixra meghívhatók a korábban definiált algoritmusaink.

Példa: rendezetlen mátrix, és rendezetté alakított mátrix

```
1.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.53850 0.51360 0.45410 0.47870 0.62280 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.43750 0.30560 0.48840 0.26190 0.34430 0.21880 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.58260 0.37500 0.63950 0.26390 0.31200 0.46430 0.07080 0.08860 0.20530 0.12290 0.20000
```

29. ábra: rendezetlen

```
1.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.53850 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.53850 0.51360 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.53850 0.51360 0.45410 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.53850 0.51360 0.45410 0.47870 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.53850 0.51360 0.45410 0.47870 0.62280 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.43750 0.30560 0.48840 0.26190 0.34430 0.21880 0.00000 0.00000 0.00000 0.00000 0.00000
1.00000 0.58260 0.37500 0.63950 0.26390 0.31200 0.46430 0.07080 0.00000 0.00000 0.00000 0.00000
1.00000 0.58260 0.37500 0.63950 0.26390 0.31200 0.46430 0.07080 0.08860 0.00000 0.00000 0.00000
1.00000 0.58260 0.37500 0.63950 0.26390 0.31200 0.46430 0.07080 0.08860 0.20530 0.00000 0.00000
1.00000 0.58260 0.37500 0.63950 0.26390 0.31200 0.46430 0.07080 0.08860 0.20530 0.12290 0.00000
1.00000 0.58260 0.37500 0.63950 0.26390 0.31200 0.46430 0.07080 0.08860 0.20530 0.12290 0.20000
```

30. ábra: rendezett, sorok és oszlopok száma egyenlő

2.4 Összefoglalás

A valós adatok alapján az optimális fa megtalálásáról le kell mondanunk, hiszen már az összes lehetséges permutáció előállítása exponenciális költségű. Mivel maga a likelihood-függvény is becslés volt, ez nem feltétlenül probléma. A rekurzió (egyszerű

vagy rendezett általános bemenetre) ideális adatok esetén rövid idő alatt is megtalálja a teljes fát, így tesztelések során a heurisztikus algoritmusokat nagyobb bementre is össze tudjuk hasonlítani az optimális fával, olyanokra, ahol a naív keresés már túl hosszú futási idővel rendelkezett. A precheck algoritmus mohó-algoritmussal való kombinációja pedig javíthatna a mohó algoritmus átlagos futási idején. Továbblépési lehetőség tehát, hogy mohó-heurisztikával állítjuk elő a lehetséges permutációkat, ezekre a permutációkra alkalmazzuk a rendezett általános esetes precheck algoritmust, és ezután a rekurzív algoritmus helyett a mohó keresést futtatjuk.

3. Felhasználói dokumentáció

Az előző fejezetben ismertetett algoritmusokhoz egy egységes felhasználói felületet hoztam létre a könnyebb kezelhetőség érdekében. Az alábbiakban írom le, hogyan érhetjük el a különböző funkciókat, és mit jelentenek a program által megjelenített üzenetek a futtatás közben.

3.1 Szükséges környezet

A program logikai részét Python nyelven írtam. A fejlesztés során a 2.7.17-es verziójú Python interpretert használtam, azonban verzió specifikus nyelvi elemek nincsenek a forráskódban, leszámítva néhány szintaktikai eltérést (pl. print zárójelezés). A programhoz készült grafikus felhasználói felületet C++ nyelven valósítottam meg Qt keretrendszerrel.

A szoftvert bioinformatikai számításokhoz használhatják a terület specialistái. Segítségével különböző algoritmusokat tudunk összehasonlítani egy adott problémakörben. Az egyszerű esetben elérhető opció, a szemléletes gráf kirajzolás segíti az algoritmusok eredményének megértését, míg a részletes Excel kimutatással az algoritmusok jobban összevethetők, a hatékonyságuk elemezhető.

A telepítés során a tömörített mappát kell kibontani, mely tartalmazza a Python programot, a Qt projekt fájljait, néhány szimulált adatsort bemenetként, valamint néhány eredmény fájlt. Ezek mellett a fejlesztéshez tartozik egy tesztelő program a Python, azaz backend részre, és egy másik a C++, azaz a frontend részre. A felhasználói felülettel való használatához a *Qt Creator*-on belül a megfelelő projektet (*thesisfront.pro*) kell megnyitni és futtatni. Az algoritmusokat tartalmazó Python program önmagában is használható, ekkor az alkalmazás nyújtotta különböző futtatási lehetőségek parancssori argumentumok megadásával érhetők el. Az argumentumokat feldolgozó függvények egyszerű módosításával változtatható, hogy az eredményeket hova szeretnénk írni. Erről bővebben a fejlesztői dokumentációban írok.

A fejlesztést Ubuntu 19.10-es operációs rendszer alatt végeztem, ebből kifolyólag az alkalmazást ezen az operációs rendszeren használtam. A program más operációs rendszeren is működik, ha teljesülnek a szükséges feltételek. Mivel a program backend

része egy Python program, ezért a kompatibilis verziójú (2.7.17.) Python értelmező telepítve kell legyen, használatához megfelelő jogosultsággal kell rendelkeznünk, és a program build könyvtárából relatív útvonal nélkül meghívható kell legyen (pl. Windows operációs rendszeren környezeti változóként). A számítógép, amin teszteltem a programot, 2GB RAM-mal rendelkezik, azonban mivel komoly számításigényű algoritmusokról van szó, ez a minimálisan ajánlott erőforrás, ami szükséges a használathoz. A program alapvetően desktop alkalmazásnak készült, a megjelenítési felület minimális mérete 600x400 képpont. Az ablak mérete természetesen ennél nagyobbra skálázható, a felület megfelelően átméreteződik.

3.2 A program használata felhasználói felülettel

A QtCreatorban történő indítás után megjelenik az alkalmazás kezdőoldala. A bal oldalon öt darab sárga gomb jeleníti meg a választható opciókat. A gombok megszokott módon, bal oldali egérkattintással működnek. A baloldal a menü, ami az alkalmazás használata során végig látható és elérhető. Az interakciók hatására az ablak jobb oldala változik, a kiválasztott menüpontnak megfelelően. A kezdőoldal az ötödik menüpont, amiről a „Súgó és információk” részben olvashatunk bővebben.

3.3 Baloldali menügombok

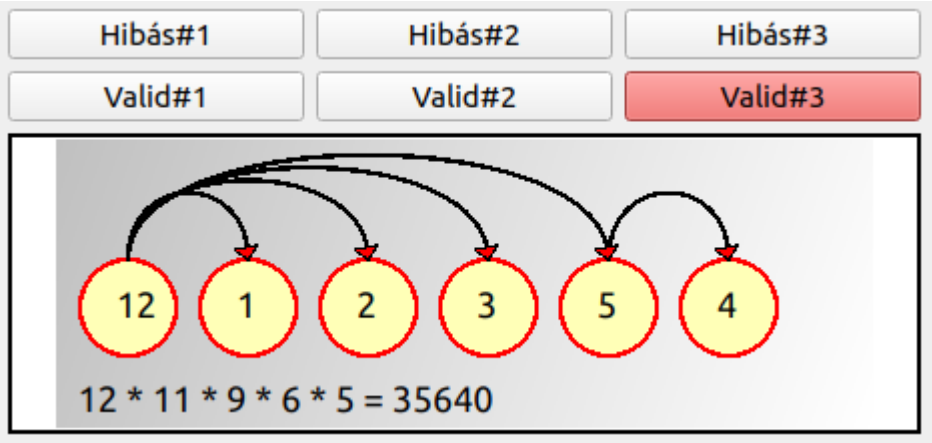
A következőkben menügombok szerint csoportosítva mutatom be az alkalmazást.

3.3.1 „Feladat bemutatása” gomb

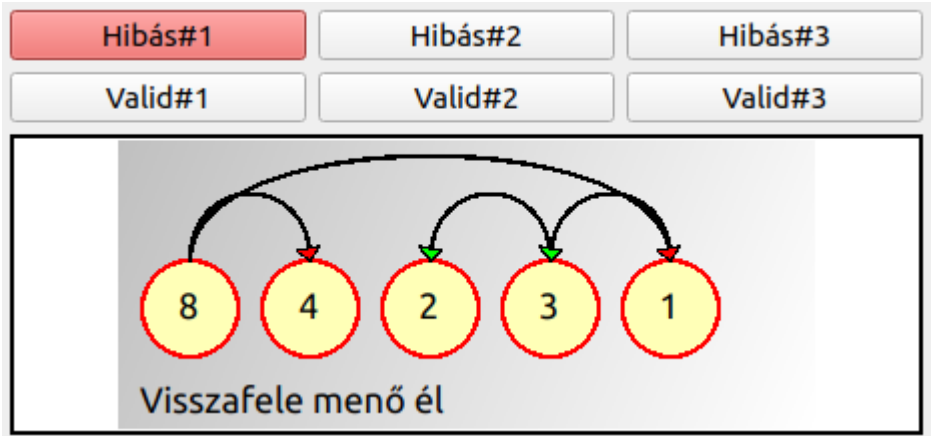
A legfelső gomb a főmenüben, kattintásra a megoldandó problémáról olvashatunk többet. Megtudhatjuk, hogy az egyszerű esetben milyen feltételeket kell teljesíteni a gráfoknak, mit jelent néhány alapfogalom, és hogy melyik gráfot szeretnénk megtalálni. Ezek után hat darab előre elkészített példából választhatunk, ezek szemléltetik az előző leírást. A példákat gombokkal érhetjük el.

31. ábra: Első menüpont

A hat példából három esetén nem teljesül valamelyik fentebb írt feltétel. Ezeken a gombokon a „Hibás” felirat szerepel. Kattintásra az alsó grafikus mezőben megjelenik egy az elkészített gráfokból. Mivel nem teljesíti a feltételeket, a sérülő feltételt egy rövid szöveg jelzi a kirajzolt gráf alatt. A három különböző példa mind különböző feltételt sért meg. A gráf jobban megvizsgálható azzal, hogy lehet közelíteni, valamint mozgatni a látóteret. Ez akkor lesz igazán hasznos, mikor nagyobb gráfokat jelenítünk meg. A másik három gombon a „Helyes” felirat szerepel. Ezen gombok kattintása egyenként, egy-egy feltételnek megfelelő gráfot rajzol ki, valamint a gráf alatt a likelihood-függvény szerinti számítást írja fel. A végeredmény mellett feltünteti a szorzótényezőket is. A megjelenítés hasonló, a csúcsokat és éleket kirajzoló látóteret tetszőlegesen mozgathatjuk itt is.



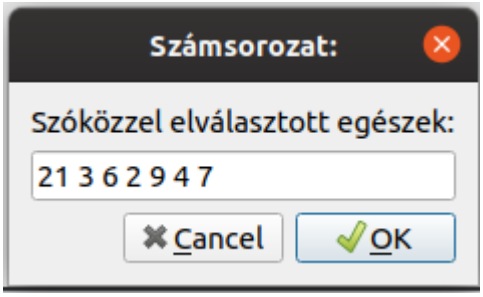
32. ábra: Megengedett gráf



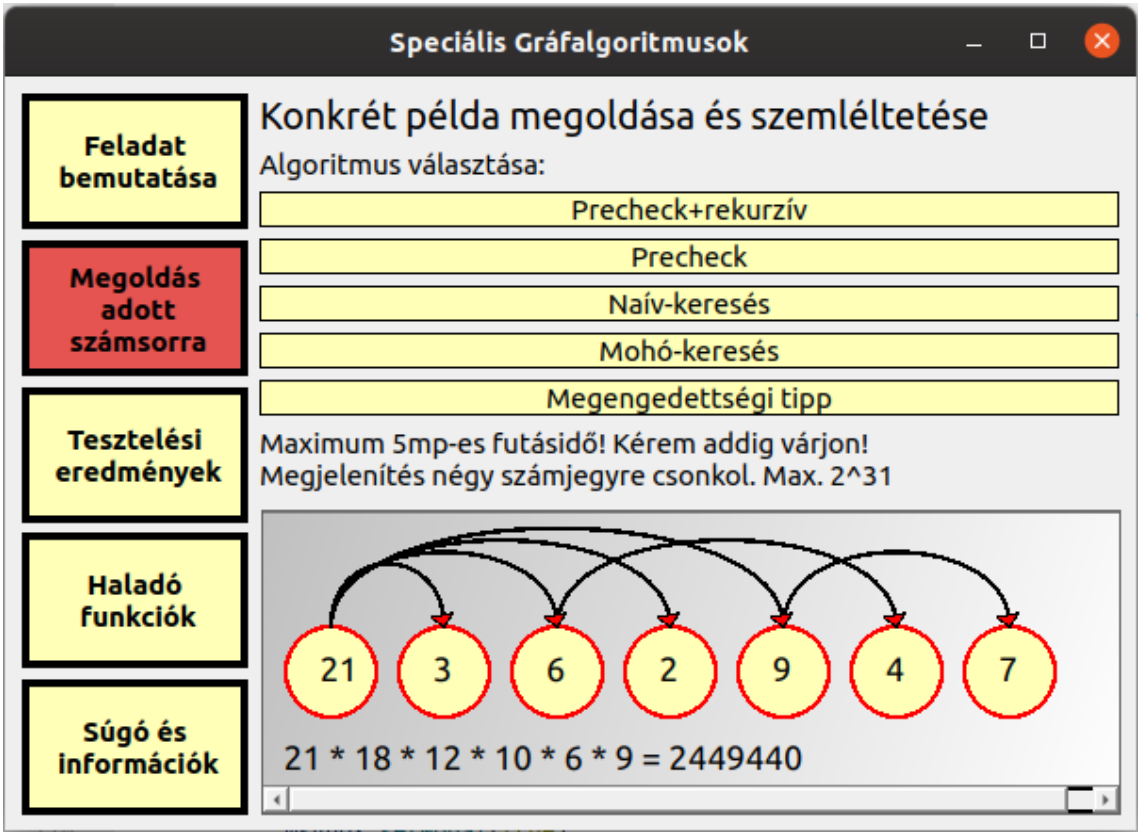
33. ábra: Nem megengedett gráf

3.3.2 „Megoldás adott számsorra” gomb

A második gomb a baloldali menüben, kattintásra kipróbálhatjuk az algoritmusokat az egyszerű esetben. Öt különböző gomb közül választhatunk, melyek különböző algoritmusokat jelölnek. Ha kiválasztjuk az egyik algoritmust, akkor a jobb alsó sarokban új párbeszédablak ugrik fel, így vihetjük be az adatokat. Bemenetként szóközzel elválasztott számokat kell megadni. A szövegmező kitöltése után az „OK” gombbal tudjuk indítani az algoritmust, míg a „Cancel”-lel vagy az ablak bezárásával vissza tudunk lépni az algoritmus választáshoz. Ha az elválasztókaraktereken kívül nem csak számokat írunk be, akkor erre a „Csak egész számokat adjon meg” feliratú felugró ablak emlékeztet minket. Elválasztásnak tetszőlegesen adható meg több szóköz karakter is, a bemeneti sorból csak a számokat veszi figyelembe az algoritmus.

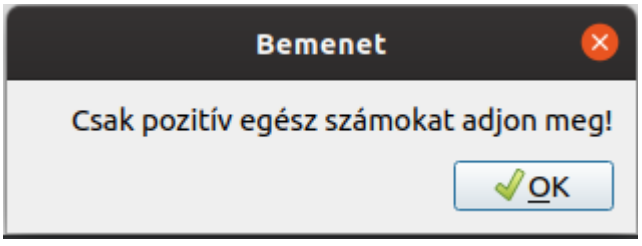


34. ábra: Számok megadása

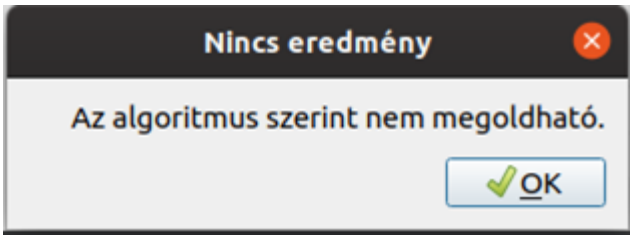


35. ábra: Második menüpont és a bemeneti adatok megoldása

Az öt különböző gomb feliratai a különböző bioinformatikai algoritmusokat jelölik. A „megengedettségi tipp” gomb használata esetén a kimenet arra vonatkozik, hogy az adott számsorra lehet-e a feltételeket kielégítő fát építeni. A kimenet ebben az esetben két értéket vehet fel, vagy megoldható az algoritmus szerint, vagy sem. A másik négy gomb használata esetén több dolog történhet. Ha az adott algoritmus nem talált megoldást, akkor ezt „Az algoritmus szerint nem megoldható” felirat jelzi. Ha talál megoldást akkor az elkészült gráfot kell megjelenítse az alkalmazás alul, a jobb oldalon. A „precheck” algoritmus esetében lehetséges, hogy a gráf csupán részleges kitöltésű. Azokban az esetekben amikor teljes fát kapunk, akkor a gráf alatt mindig megjelenik a hozzá tartozó számolás is. A megjelenítés ugyanúgy történik, mint az előző menüpontnál. *Lehetséges felugró ablakok:*



36. ábra: Bemenet hibaüzenet



37. ábra: Nem talált megoldást párbeszédablak

Az összes algoritmus esetén előfordulhat, hogy a program nem fut le 5 másodperc alatt. Ilyekor a „Nem futott le 5 mp alatt” szöveg jelenik meg felugró ablakban. Ahogy a szöveg is tanácsolja, ilyenkor célszerű a futtatást a „haladó funkciók” menüpont alatt végezni, ahol nincs időkorlát a futásra vonatkozóan.

38. ábra: Átlépte az időkorlátot hibaüzenet

3.3.3 „Tesztelési eredmények” gomb

A középső gomb a főmenüben. Kattintásra a jobb oldalon egy szöveg jelenik meg, ami azt magyarázza el, hogy mit kell vizsgálnunk, amikor az algoritmus helyességéről szeretnénk megbizonyosodni. A különböző gombok az elmondott adatokhoz való hozzáférést segítik. A négy gombbal más és más szempontok szerinti táblázatokat tölthetünk le.

Az első gomb a „results1.xls” felirattal kattintásra megmutatja, hogy valóban működik a precheck és rekurzió algoritmus, és hogy a 100 egyszerű tesztesetre a függvényértékek mindig teljesen egyezőek voltak az optimális fával. A második „results2.xls” feliratú gomb egy olyan táblázatot nyit meg, ami az általános esetben ellenőrzi az algoritmus helyességét. A harmadik gomb, az algoritmusomat a mohóval hasonlítja össze a függvényértékek szemszögéből. A negyedik pedig azt kívánja megmutatni, hogy a precheck algoritmus polinomiális költséggel hány élt tud meghatározni az általános esetben.

39. ábra: Harmadik menüpont

3.3.4 „Haladó funkciók” gomb

Ezen menüpont alatt időkorlát nélkül futtathatjuk az algoritmusokat. Például a tesztelési eredmények oldalon látott táblázatok is elkészíthetők a programmal a haladó funkciók menüponton belül. Kattintásra feljönnek a lehetőségek, hogy milyen beállításokkal szeretnénk futtatni a programot. Ebben az esetben a beolvasás fájlból történik, és az eredményeket fájlba írjuk ki, az eredményeknél látott sémához hasonlóan. A bemeneti fájlban szereplő adatok lehetnek szimulált vagy valós adatok, valamint eltérő típusúak is. Néhány szimulált példaadatsor található a „build” könyvtárban elhelyezett „testcases” mappán belül.

A beállítások első része a megoldandó feladat típusáról szól. Lehetséges az egyszerű esetet választani, amikor a második menüponthoz hasonló feladatokat old meg a program. A bemeneti fájlban az adatok, vagyis a számsorok, egy „#” -et tartalmazó sorral kell legyenek elválasztva. Ha az általános gombot választjuk, akkor az általános esetre oldja meg a program a feladatokat, ebben az esetben a használandó függvényt is kiválaszthatjuk. Alapvetően az „Alapértelmezett függvény” használata az általános, eltérés csupán a rendezetlen általános esetben van, amikor különbözőek az opciók.

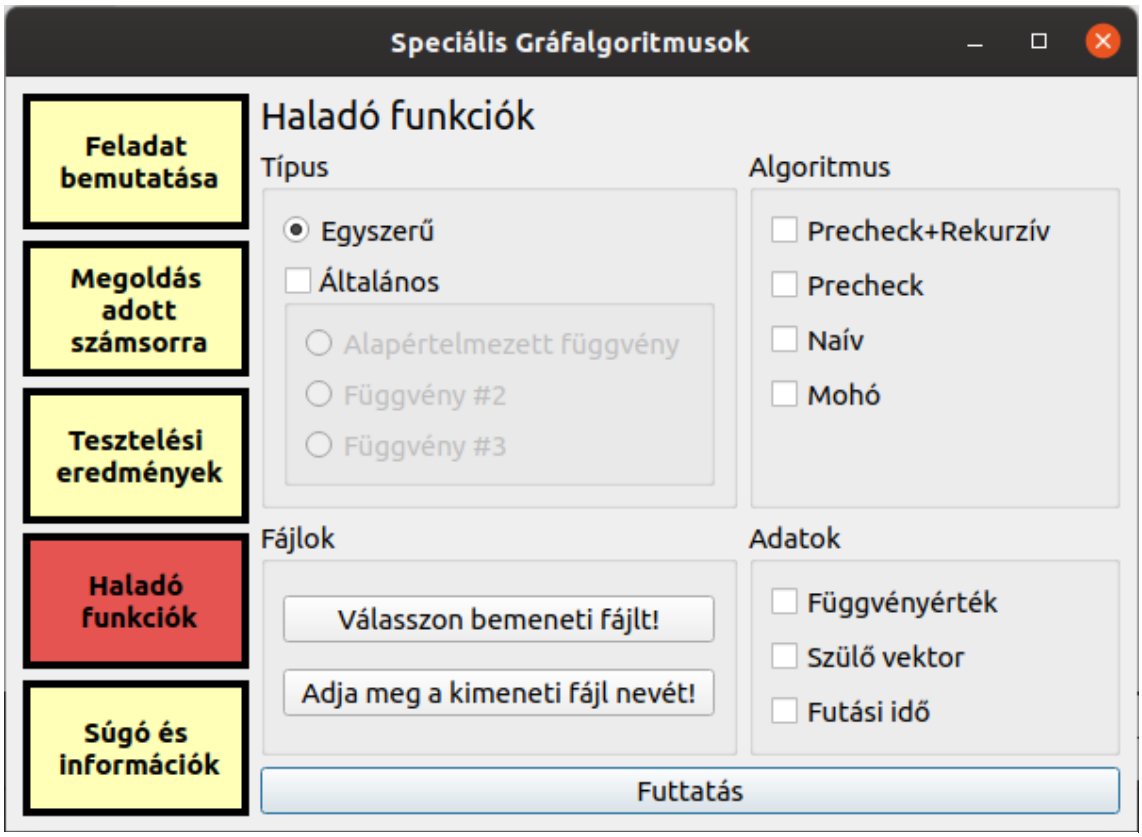
Általános esetet választva mátrixokat vár a bemeneti fájlban az algoritmus, a szeparátor itt is egy olyan sor, amiben szerepel a „#” karakter. Az általános esetes példák lehetnek vegyesen rendezettek vagy rendezetlenek is. A bemeneti adatoknál az első -viszonyítási alapnak vett- oszlopot nem kell jelölni a fájlban. A fájl „sims” kiterjesztéssel kell rendelkezzen, az ilyen típusú fájlokat tudjuk megnyitni a fájl kiválasztása résznél.

Példa bemeneti fájlra:

```
#
0.68000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.34667 0.36169 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.17204 0.44311 0.01576 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.03201 0.67994 0.26602 0.11693 0.00000 0.00000 0.00000 0.00000 0.00000
0.12960 0.42074 0.21116 0.10226 0.05938 0.00000 0.00000 0.00000 0.00000
0.09199 0.47742 0.15545 0.23532 0.12782 0.08669 0.00000 0.00000 0.00000
0.07828 0.54396 0.20522 0.25386 0.23515 0.13640 0.07111 0.00000 0.00000
0.11532 0.42216 0.08490 0.32675 0.21156 0.22778 0.16055 0.07108 0.00000
0.16382 0.57438 0.18284 0.22735 0.11325 0.05815 0.12833 0.01254 0.06847
#
0.87490 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.57419 0.35801 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.36815 0.17533 0.41217 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.59366 0.18856 0.28274 0.12714 0.00000 0.00000 0.00000 0.00000 0.00000
0.48755 0.20078 0.19005 0.18180 0.01965 0.00000 0.00000 0.00000 0.00000
0.40654 0.17537 0.28506 0.21862 0.09758 0.12568 0.00000 0.00000 0.00000
0.17564 0.14621 0.42943 0.19927 0.06731 0.24113 0.08704 0.00000 0.00000
0.05873 0.02612 0.45847 0.30534 0.09178 0.38145 0.19174 0.16195 0.00000
0.17007 0.15495 0.28794 0.14671 0.10611 0.44777 0.16704 0.15894 0.09545
#
0.51795 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.69129 0.38417 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.86756 0.34432 0.20941 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.84741 0.41225 0.14178 0.16655 0.00000 0.00000 0.00000 0.00000 0.00000
0.71821 0.46567 0.17817 0.22656 0.13418 0.00000 0.00000 0.00000 0.00000
0.75922 0.52700 0.00312 0.36497 0.17808 0.19463 0.00000 0.00000 0.00000
0.78139 0.39373 0.14114 0.32748 0.16433 0.09759 0.01388 0.00000 0.00000
0.78100 0.38879 0.12388 0.09997 0.04449 0.09447 0.17329 0.14249 0.00000
0.71921 0.46818 0.07978 0.21899 0.04461 0.16358 0.12168 0.16601 0.08696
```

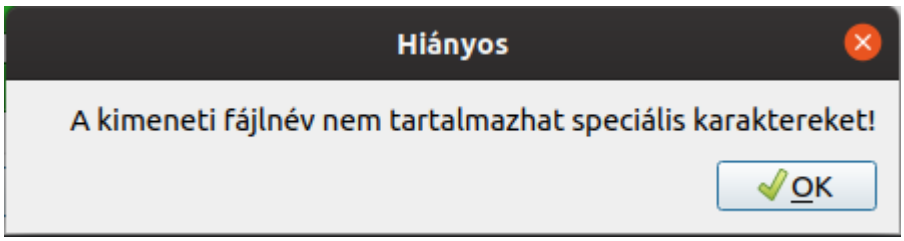
40. ábra: Bemeneti fájl példa (általános eset)

A beállítások második része arról szól, hogy melyik algoritmusokat szeretnénk futtatni. A négy lehetséges beállítás a „Precheck+rekurzív”, „Precheck”, „Naív”, és „Mohó”. Ez a beállítási blokk módosul annak függvényében, hogy egyszerű, vagy általános esetről van-e szó. A naív és a mohó lehetőségek nem érhetők el az általános esetre. Tetszőlegesen kombinálhatók az algoritmusok, természetesen legalább egyet kell választani.



41. ábra: Negyedik menüpont

A harmadik szekció a fájlok, ami két gombot tartalmaz. A „Válasszon bemeneti fájlt!” gomb kattintásával megnyílik új ablakban a fájlkezelő, és a felhasználónak lehetősége van a grafikus felületen kiválasztani a bemeneti fájlt. Sikeres választás esetén a gomb színe zöldre változik. A másik gombon az „Adja meg a kimeneti fájl nevét” felirat szerepel, ennek kattintása esetén egy párbeszédablak nyílik. Itt adhatjuk meg szövegesen a fájl nevét, ahova .xls kiterjesztéssel az eredményeket írjuk a futás végén. Sikeresen megadott fájlnév esetén ez a gomb is zöld színűre változik. Természetesen, hogy a megadott szöveg fájlnévként használható legyen, nem tartalmazhat speciális karaktereket.



42. ábra: Speciális karakterek hibaüzenet

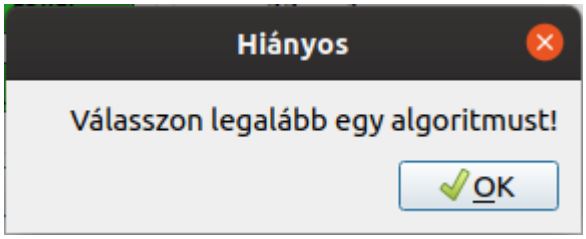
A negyedik blokk a kimenő adatok típusát tartalmazza, amik szerint össze szeretnénk hasonlítani az algoritmusokat. A lehetséges választások a „Függvényérték”, „Szülő

vektor” és a „Futási idő”. Ahogy a második szekció esetén, itt is a checkboxok közül legalább egy választása kötelező.

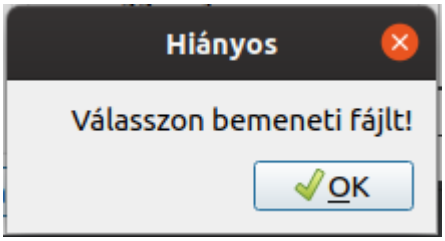
Az alsó, „Futtatás” feliratú gombbal tudjuk elindítani a Python programot. Mivel nagy számításigényű algoritmusokról van szó, ahol bemeneti adatsortól függően nagyon hosszú időt is igénybe vehet a futás, a programot leválasztva indítjuk el. Ez azt jelenti, hogy a felhasználó az alkalmazáson belül nem kap értesítést, a futás befejeződéséről, hanem a fájl létrejöttekor tudja megnézni az eredményeket. Az eredményeket tartalmazó „fájlnév”.xls, ahol a fájlnév a megadott szöveg, a program build mappájában található meg a futás végeztével.

A futtatás gomb kattintása esetén ellenőrzésre kerülnek a kiválasztott opciók, emiatt felugró ablakban a következő hibaüzeneteket kaphatja a felhasználó: „Válasszon bemeneti fájlt”, „Válasszon kimeneti fájlt!”, „Válasszon legalább egy algoritmust”, „Válasszon legalább egy kimeneti értéket”. Ilyen esetekben a megadott utasítást kell követni, és pótolni a hiányzó részt.

Példaüzenetek:



43. ábra: Hibaüzenet 1.

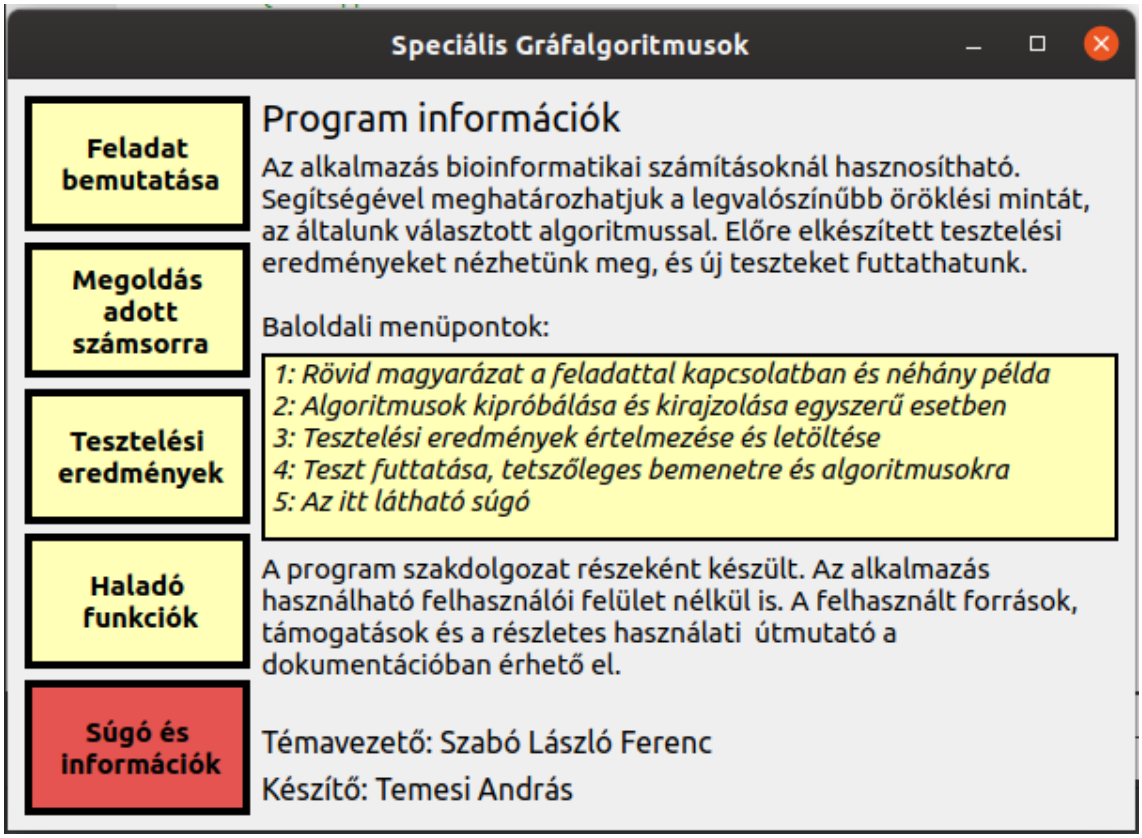


44. ábra: Hibaüzenet 2.

3.3.5 „Súgó és információk” gomb

A gomb kattintásával a jobb oldalon információkat láthatunk a programra vonatkozóan. Ezek közé tartozik a program célja, feladata, a célcsoport, akik számára a program

készült, a program funkcionalitása, a menüpontok leírása, a felhasznált irodalom, a készítő és egyéb információk.



45. ábra: Ötödik menüpont

Általánosságban igaz, hogy a program futása bármikor megszakítható, és bezárható, a jobb felső sarokban a piros „X” kattintásával. Ha át szeretnénk méretezni az ablakot, megtehetjük a szokásos módon, a széleken állítva a méretet. Amikor egy gráf kirajzolásra kerül, akkor a látótéren belül mindig van lehetőségünk görgetni az egérrel, ezáltal közelíteni vagy távolítani, avagy a bal egérgomb lenyomásával bármely irányba mozgatni az alakzatokat.

3.4 Backend rész

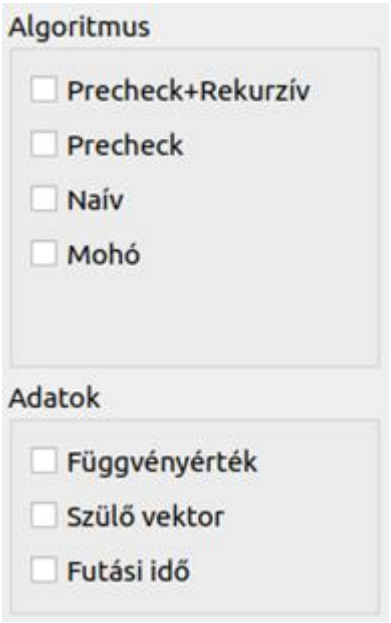
Az algoritmusok.py önmagában is használható, bármilyen felhasználói felület nélkül. Ilyenkor több lehetőségünk is van. Gyakorlott felhasználóként akár a forráskódot is módosíthatjuk, közvetlenül meghívva a számunkra szükséges függvényeket. A módosítás után a parancssorból a „python algoritmusok.py” utasítást kiadva végrehajtásra kerülnek a módosítás szerinti függvényhívások.

Kevesebb tudást igénylő módszer, ha forráskód módosítás nélkül, parancssorból a „python algoritmusok.py” utasítás után megadjuk az argumentumlistát. Erről bővebben a fejlesztői dokumentációban lesz szó, de ilyen esetben a hívás így néz ki:

```
andras@andras-K53E:~/Desktop/thesis/build-thesisfront-Desktop-Debug$
python algoritmusok.py 0 1 0 1 0 0 1 1 1 /home/andras/Desktop/thesis/
build-thesisfront-Desktop-Debug/testcases/square.sims myoutput
andras@andras-K53E:~/Desktop/thesis/build-thesisfront-Desktop-Debug$
python algoritmusok.py 4 66 1 22 18 6 1
```

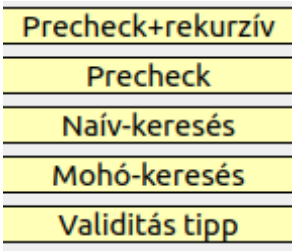
46. ábra: Argumentumlista példahívások

A felső parancs a „Haladó funkciók” esetén történő hívás mintája. A python algoritmusok.py rész után az első szám jelenti, hogy Haladó funkciók szerinti hívásról van szó (0), a második szám a használt függvény (0- egyszerű, 1-általános, 2 és 3-általános alternatívák) a következő 7 darab szám pedig 0/1-gyel jelzi, hogy az adott (felhasználói felületen is látott) funkció be van-e kapcsolva (47. ábra). Az utolsó két argumentum adja meg a bemeneti fájlt és a kimeneti fájl nevét. A példa jelentése: square.sims bemeneti fájlból általános esetek megoldása a precheck algoritmussal, a myoutput nevű kimeneti fájl pedig tartalmazza a függvényértékeket, szülő vektorokat és a futási időt is.



47. ábra: Funkciók sorrendben

Az alsó parancs esetén, a második menüpont szerinti hívás történik felhasználói felület nélkül. Az első szám 1-től 5-ig vehet fel értéket (azaz nem a másik minta szerinti 0-val kezdődik), a maradék számok pedig a bemeneti adatsor. Az algoritmusok sorrendje a második menüpontban látott sorrend (48. ábra). A példa jelentése tehát: mohó keresés a 66 1 22 18 6 1 számsorra, természetesen az egyszerű esetben.



48. ábra: Algoritmusok

4. Fejlesztői dokumentáció

Szakdolgozatomban egy bioinformatikai problémakört vizsgálok, az ahhoz tartozó algoritmusokat implementálom, valamint egy felhasználói felületet készítek, amivel az algoritmusok használata gördülékenyebb, a feladat és a megoldás érthetőbb, az eredmények pedig könnyebben összehasonlíthatók.

4.1 Célkitűzések

Az alkalmazás tervezésénél a fő cél egy olyan program megalkotása volt, mellyel algoritmikus számítások végezhetők. Az egyszerű kezelhetőség érdekében a programhoz grafikus felület készült. Szempontként merült fel, hogy a témában kevésbé jártas felhasználó is megismerhesse a problémakört, ezért a grafikus felületen, a gombok és feliratok mellett maga a megoldás, azaz az adatsort realizáló gráf is szemléletesen kirajzolásra kerül. Mivel a problémának több változata létezik, és több algoritmussal lehet dolgozni, fontos volt, hogy a gyakorlottabb felhasználó minél több esetre alkalmazhassa a programot, és az eredményekről tudjon kimutatást is készíteni. Kutatási szempontok miatt az alkalmazás használható felhasználói felület nélkül is, azaz az algoritmusok implementációja különálló részét képezi a programnak.

4.1.1 Logikai rész elképzelése

Eredeti célkitűzésünk - vagyis, hogy az algoritmusok grafikus felület nélkül is használhatók legyenek - miatt a logika és a felület elkülönítése adott volt. A problémakörben gyakori a script nyelvek, azon belül a Python használata, így az algoritmusok implementálását, vagyis magát a backendet, Pythonban terveztem elkészíteni. Mivel főként eljárásokat implementálok, pozitívum volt, hogy a nyelv támogatja a procedurális programozást. További előny a Python választása esetén, hogy egyszerű megvalósítást biztosít néhány szükséges feladatra, például permutációk készítésére. Így a kód maga sokkal olvashatóbb és átláthatóbb lett. Mivel kisebb alkalmazásról van szó, ezért nem jelentett hátrányt a nyelv gyengén típusossága, a fejlesztés kevésbé volt komplikált, és a jövőre nézve is egyszerűen bővíthető maradt a backend rész új algoritmusokkal vagy esetleges módosításokkal.

4.1.2 Felület elképzelése

A felhasználói felületet Qt keretrendszerben akartam elkészíteni C++ nyelven, ami rendkívül egyszerű fejlesztést biztosít kisebb asztali alkalmazások számára. A két program kommunikációját argumentumokkal terveztem megvalósítani, ezáltal biztosítani, hogy az összes grafikus felületen elérhető funkció – a gráfok kirajzolását leszámítva - használható legyen a Qt keretrendszer elhagyásával, egyszerűen a parancssorból. A felület esetén cél volt egy letisztult kinézet és az egyszerű kezelhetőség. Mivel a program alapvetően egy kutatói célcsoport számára készült, néhány rész, például a bemeneti adatsor formátuma, az eredménytáblázatok részletesebb értelmezése, elvár egy alap tájékozottságot a témában. Természetesen a technikai, és formai követelmények a felhasználó számára megtalálhatók a felhasználói dokumentációban. Az elképzelt felület egy menüalapú megjelenítés volt, állítható ablakmérettel, és interakcióra való lehetőséggel a kirajzoláshoz használt látótér esetén. A program által kínált részletesebb beállítások megvalósítását rádiógombokkal és jelölőnégyzetek használatával, az eredmények megjelenítését a felhasználói felületről közvetlenül megnyitható táblázatokkal terveztem. Az említett kinézethez szükséges eszközöket, és még másokat is, (pl. felugró ablak) a Qt keretrendszer mind biztosította számomra.

A bemeneti adatokat célszerű fájlból beolvasni, ehhez maga a fájl megfelelő formátumú kell legyen. Több bemeneti adat esetén egy „#”-et tartalmazó szeparátorral szükséges elválasztani az adatokat. Alapértelmezetten a felhasználók fájlból fogják az adatokat beolvasni, de egyszerűbben is kipróbálható az algoritmusok működése kézzel megadott bemenet esetén, ekkor a felnyíló ablakban lehet beírni az adatokat (felület nélkül pusztán argumentumlista használatával). Bonyolultabb adatsor esetén a fájlból való beolvasást kell válassza a felhasználó. A terv része volt a backend oldalon elválasztani az algoritmusokat, a beolvasást, és az argumentumok feldolgozását. A beolvasás során a bemeneti adatokat listában terveztem tárolni, a listák elemei lehetnek számok vagy további listák adatsortól függően. A tárolt adatok feldolgozása az algoritmusok hívásával történik, paraméterben megadva az adatokat. Cél volt, hogy végül az adatok beolvasása, tárolása, feldolgozása, és maguk az algoritmushívások is összegezve legyenek egy

eljárásban, így egy fájl feldolgozása egy eljáráshívással történhet, ezáltal könnyen kezelhető marad a program.

4.2 Megvalósítás

A terveknek megfelelően sikerült megvalósítani a programot. Fejlesztés közben egy funkción kellett módosítani, az eredményeket tartalmazó dokumentumok megnyitása nem az elképzeléseknek megfelelően közvetlenül az adott számítógépen, helyi elérésű programmal történik, hanem a fájlok internetes kapcsolattal tölthetőek le. Erre azért volt szükség, mert például Linux rendszer alatt, több, táblázatokat kezelő programmal kompatibilitási problémája volt a Qt rendszernek.

4.2.1 Backend

A konkrét megvalósítás során először arról kellett dönteni, hogyan szeretnénk tárolni a különböző adatokat, melyeket használ a program. A bemenetnél kétfajta adathalmaz lehetséges, az egyik számsorozatok egymásutánja, míg a másik mátrixok sorozata. Kérdés volt továbbá a belső reprezentációja a speciális gráfoknak, melyeket használ az algoritmus. A megoldandó adatsorok tárolására az egyszerű esetben kétdimenziós listát (azaz listákat tartalmazó listát) míg az általános esetben háromdimenziós listát (azaz „mátrixokat” tartalmazó listát) használtam. Ez az adatsorokat figyelembe véve a legkézenfekvőbb megoldás volt, a külső lista tartalmazza a teljes bemenetet, ennek elemei az egyes adatsorok, melyek esettől függően lehetnek egy- vagy kétdimenziósak. Kétdimenziós esetben ismét listákat tartalmazó listát használtam. Nevezzük ezt a két tárolási formát a továbbiakban is kétdimenziós és háromdimenziós listának. A speciális gráfok számára új, egy listát és egy számot tartalmazó osztályt (to_return) definiáltam. Ennek segítségével könnyen megadható egy adott bementre talált fa és a hozzá tartozó függvényérték.

Az algoritmusokat az elképzeléseknek megfelelően pythonban implementáltam, az eljárások részletes logikai hátterét az *2.1, 2.2, 2.3* sorszámú fejezetekben mutatom be. A backend jelentős része az algoritmusok definiálása volt, melyeket a megadott argumentumok függvényében hívok meg. Ezeket az algoritmusokat az alábbiakban foglalom össze táblázatos formában.

4.2.2 Algoritmusok

- A függvények kiértékelését végző eljárások:

count_prob, general_count_prob,

general_count_prob_opt2, general_count_prob_opt3

Ezek az eljárások mind egy kész fa likelihood-függvény szerinti értékét hívatottak kiszámítani. Ahogy a nevük is jelzi, az egyszerű és általános esetre vonatkoznak, az utolsó kettő esetén apró módosítással a rendezetlen általános esetre. Összefoglaló táblázat:

Függvény neve	Bemenő adat	Kimenő adat	Tevékenység
count_prob	két darab lista: a számok és a szülők	*to_return* objektum	Az eljárás ellenőrzi, hogy a két lista meghatároz-e egy speciális gráfot, ha igen és ez megengedett, akkor a gráf függvényértékét és a szülőlistát tartalmazó *to_return* objektumot adja vissza, egyébként -1-et
general_count_prob_official	egy kétdimeziós lista, egy lista: a számok mátrix, és a szülők	*to_return* objektum	Az eljárás a bemeneti speciális gráfra kiszámolja az általánosított likelihood-függvény szerinti értékét, ezt és a szülőlistát tartalmazó *to_return* objektumot adja vissza
general_count_prob_opt2	egy kétdimeziós lista, egy lista: a számok mátrix, és a szülők	*to_return* objektum	Mint a *general_count_prob_official*, de rendezetlen általános esetben csak az új csúcs esetén számol az aktuális sorból

general_count_prob_opt3	egy kétdimeziós lista, egy lista: a számok mátrix, és a szülők	*to_return* objektum	Mint a *general_count_prob_official*, de az általános esetben mindig az aktuális sorból számolja ki a likelihood-függvénynek az értékét.

- Beolvasó eljárások: *simple_readin, general_readin*

Ezek az eljárások a fáljból való beolvasás esetén a bemenetet dolgozzák fel és tárolják. Összefoglaló táblázat:

Függvény neve	Bemenő adat	Kimenő adat	Tevékenység
simple_readin	string: a fájlnév	kétdimenziós lista: a feldolgozott bemenet	Megnyitja a paraméterként kapott fájlt és soronként beolvassa az adatokat. Listaként tárolja az adatokat, melynek elemi listák, a listaelemeket *float* típusúvá alakítja és ezt adja vissza
general_readin	string: a fájlnév	háromdimenziós lista: a feldolgozott bemenet	Megnyitja a paraméterként kapott fájlt és soronként beolvassa az adatokat. A szeparátort tartalmazó sorok közt épített kétdimenziós listák a listaelemek. A belső lista elemeit *float* típusúvá alakítja és a mátrixokat tartalmazó listát adja vissza

- Az egyszerű esetben a speciális gráfot kereső algoritmusok: *check_all_poss, line_pre_check, simple_recursive_alg, simple_greedy, guess_solvable*

Ezek az eljárások az egyszerű esetben a már tárolt adatsorra szeretnék meghatározni az optimális fát. Összefoglaló táblázat:

Függvény neve	Bemenő adat	Kimenő adat	Tevékenység
check_all_poss	lista: számok	*to_return* objektum	A számsorra veszi az összes lehetséges szülőlistát, és a *count_prob* segítségével maximumkeresést végez. Ha egyik sem megengedett akkor -1-et ad vissza, egyébként az optimális fa szülőlistáját, és függvényértékét
line_pre_check	két lista: számok és a részlegesen kitöltött szülőlista	kétdimenziós lista: szülőlista lehetőségekkel	A 2.1.6 fejezetben említett módon jelöli a biztosan meghatározható éleket, és a részlegesen kitöltött szülőlistát adja vissza, kivéve, ha nem megoldható, akkor -1-et.
simple_recursive_alg	négy lista: számok, részlegesen kitöltött szülőlista, szülőlista alapján módosított számlista, számok indexei rendezve	*to_return* objektum	A 2.1.6 fejezetben említett módon csökkenő sorrendben keresi a számokra a szülőket, és az összes lehetőség szerint meghívja a módosított listákkal rekurzívan a függvényt. Az optimális fát jelölő szülőlistát és a hozzátartozó függvényértéket adja vissza *to_return* objektumként

simple_greedy	lista: számok	*to_return* objektum	A bemeneti lista esetén közelíti az optimális fát, minden választásnál a lokálisan legnagyobb értékűt választja. Nem feltétlenül az optimális fát reprezentáló *to_return* objektumot adja vissza, akkor is adhat -1-et ha lenne megengedett fa
guess_solvable	lista: számok	logikai: létezik-e megengedett fa	A bemeneti lista esetén polinomiális költséggel adja meg, hogy létezik-e megengedett fa. Lehetséges, hogy létezik, de hamis értéket ad vissza

- A rendezetlen általános esetben a gráfot megfelelő formára hozó algoritmusok: *general_newclones, general_permutations_upd*

Ezáltal tudjuk a későbbiekben a rendezett és rendezetlen esetet ugyanazokkal az eljárásokkal kezelni. Összefoglaló táblázat:

Függvény neve	Bemenő adat	Kimenő adat	Tevékenység
general_newclones	kétdimenziós lista: számok mátrixa	lista: soronkénti új nemnulla oszlopok	A bemeneti mátrixra meghatározza, hogy melyik sorban hány olyan oszlop van, ami az előző sorban 0 volt, az aktuálisban pedig nem. Ezeknek a soronkénti számát listaként adja vissza.

general_permutations_upd	lista: soronkénti új nemnulla oszlopok	lista: a lehetséges permutációk	A bemeneti lista minden eleme esetén veszi az elem értékénél kisebb számok permutációit, és ezeknek adja vissza az összes kombinációját.

Az általános esetben a speciális gráfot kereső algoritmusok: *matrix_pre_check, full_pre_check, general_recursive, general_solver_really, general_precheck*

Ezek az eljárások az általánosításai az egyszerű esetben látottaknak. Összefoglaló táblázat:

Függvény neve	Bemenő adat	Kimenő adat	Tevékenység
matrix_pre_check	kétdimenziós lista, lista: számok mátrixa, részlegesen kitöltött szülőlista	háromdimenziós lista: szülőlehetőségek	A bemeneti mátrix minden sorára meghívjuk a *line_pre_check* algoritmust, ezáltal minden mátrixelem esetén megkapjuk a szülőlehetőségeket. Ezt a listákat tartalmazó mátrixot adja vissza.
full_pre_check	kétdimenziós lista, lista: számok mátrixa, részlegesen kitöltött szülőlista	lista: szülőlista	A *matrix_pre_check* algoritmust alkalmazzuk, mindig az új szülőlistával. A háromdimenziós lista eredményeit összegezzük egy listában, addig amíg felfedezünk új elemet. Ezt a végső listát adjuk vissza.

general_recursive	négy lista, egy int: számok, részlegesen kitöltött szülőlista, szülőlista alapján módosított számlista, számok indexei rendezve, használandó függvény	*to_return* objektum	A *full_pre_check* eljárással keressük az elérhető részlegesen kitöltött szülőlistát, majd a *simple_recursive_alg* eljárást alkalmazzuk a mátrix utolsó sorára, és a paraméter által jelölt - egy általános esetben használt – függvény szerint határozzuk meg az optimális fát, melyet egy *to_return* objektumként adunk vissza.
general_solver_really	string, négy szám: a fájlnév, az opciók (függvényérték, szülőlista, futási idő, használandó függvény	-	A bemeneti fájl nyitja meg és olvassa be a *general_readin* függvénnyel, az adatok négyzetes mátrixra alakítja a *general_newclones* és *general_permutations* függvénnyel, és a *general_recursive* függvénnyel keresi az optimális fát. Az eredményeket a megadott fájlba írja ki.

4.2.3 Kapcsolat a backend és frontend közt

- Argumentumokat kezelő eljárások: *check_args, get_numlist, get_advanced_args*

Összefoglaló táblázat:

Függvény neve	Bemenő adat	Kimenő adat	Tevékenység
check_args	-	logikai	Visszaadja, hogy a program argumentumai számok-e
get_numlist	-	lista	A *check_args* eljárással ellenőrzi, hogy számok-e az argumentumok, majd *int*eket tartalmazó egyszerű esetben használt listaként tárolja azokat. Hibaüzenettel kilép, ha a számok nem felelnek meg a követelményeknek.
get_advanced_args	-	lista	Mint a *check_args*, de az általános esetre vonatkozó argumentumlistával

- Az argumentumokat feldolgozó függvények: *advanced_solver, task_one_precheck_recursive, task_two_precheck, task_three_naiv, task_four_greedy, task_five_guess*

A Qt felület által, vagy kézzel megadott argumentumokat értelmezik, aszerint hívják meg a szükséges eljárásokat, és rögzítik fájlba az eredményeket. Összefoglaló táblázat:

Függvény neve	Bemenő adat	Kimenő adat	Tevékenység
advanced_solver	-	-	Az advanced_args eljárással kapott lista szerint, a megadott (egyszerű vagy általános) esetben, a megadott algoritmusokat hívja meg mindezt úgy, hogy a kért eredményeket kapjuk. (futási idő / függvényérték / szülőlista) A meghívott eljárások által módosított fájlt menti a szükséges néven.

task_one_precheck_recursive	-	-	A *get_numlist* eljárással kapott listára hívja meg a *line_pre_check* és a *simple recursive_alg* eljárásokat. Az eredményt az output.txt-be írja.
task_two_precheck	-	-	A *get_numlist* eljárással kapott listára hívja meg a *line_pre_check* eljárást. Az eredményt az output.txt-be írja.
task_three_naiv	-	-	A *get_numlist* eljárással kapott listára hívja meg a *check_all_poss* eljárást. Az eredményt az output.txt-be írja.
task_four_greedy	-	-	A *get_numlist* eljárással kapott listára hívja meg a *simple_greedy* eljárást. Az eredményt az output.txt-be írja.
task_five_guess	-	-	A *get_numlist* eljárással kapott listára hívja meg a *guess_solvable* eljárást. Az eredményt az output.txt-be írja.

A program végrehajtandó része pedig a megfelelő argumentumokat feldolgozó függvényt hívja meg az első argumentum alapján, nem megfelelő formátum esetén hibaüzenettel kilép.

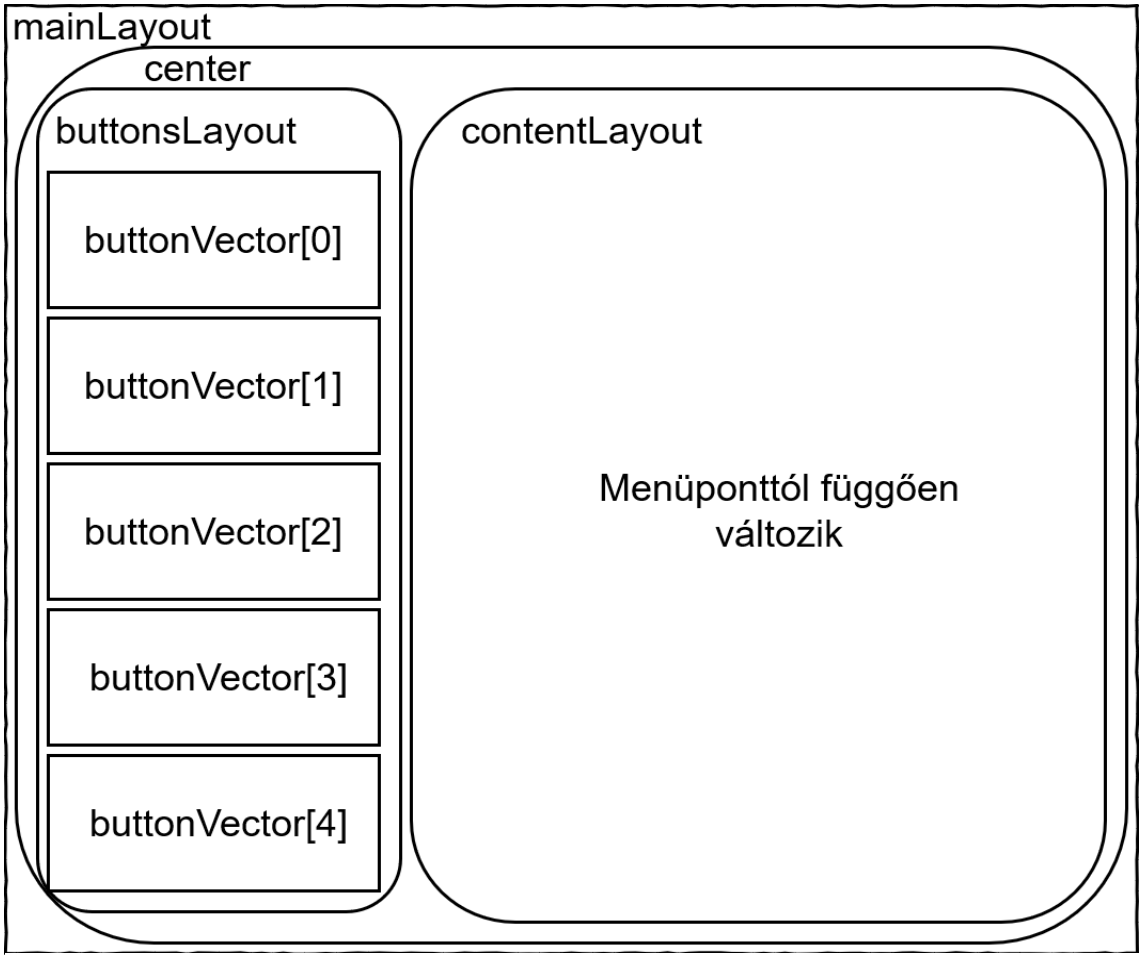
4.2.4 Frontend

A megvalósítás során a legfőbb kérdés az volt, hogy mivel végezzük a gráfok kirajzolását. Szerencsére a Qt keretrendszeren belül létezik az *QGraphicsView* osztály, mellyel lehetséges kétdimenziós ábrákat rajzolni, és egyszerű metódusokat biztosít az események kezelésére. A másik általam készített osztályt, a *DrawWidget*et, a *QGraphicsView*ból származtatom.

A mainwindow.h és a mainwindow.cpp fájlok tartalmazzák a *MainWindow* osztályt. A *MainWindow* egy *QMainWindowból* származtatott osztály, melynek fő feladata az alkalmazás felületének megjelenítése, a gombok és szövegek létrehozása, és ezeknek a

felhasználói interakció révén való módosítása. A felület segítségével tudunk továbbá fájlt kiválasztani, fájlnevet megadni, teszteredményeket megnyitni. Az objektumok a felhasználó aktivitásának megfelelően vannak létrehozva dinamikusan, a már nem szükséges objektumok futási időben kerülnek törlésre. Mindig csak az aktuális, a felhasználó számára látható *widget*ek léteznek. A mainwindow.h tartalmazza a tagfüggvények deklarációját, míg a mainwindow.cpp a tagfüggvények definícióját.

Az osztály adattagjai nagyrészt pointerek. A program indításakor a main.cpp-ben létrehozzuk a *mainWindow* objektumot. Ekkor meghívódik a *mainWindow* konstruktora, és létrehozzuk az ablak alapvető felosztását. Ennek során a *mainWindow* közepére egy center nevű *widget* kerül, melyre a *mainLayout* nevű, vízszintes elrendezést biztosító *QHBoxLayout* típusú objektumot rakjuk. A *mainLayout*hoz adjuk hozzá a *buttonsLayout* és *contentLayout QVBoxLayout* típusú objektumokat, ezzel kialakítva a két oszlopból álló kinézetet. A *buttonsLayout*ba *QPushbutton*okat helyezünk, melyeket szintén dinamikusan hozunk létre, a *QPushbutton* típusú pointereket tartalmazó *buttonVector* adattag segítségével.



49. ábra: A mainWindow felosztása

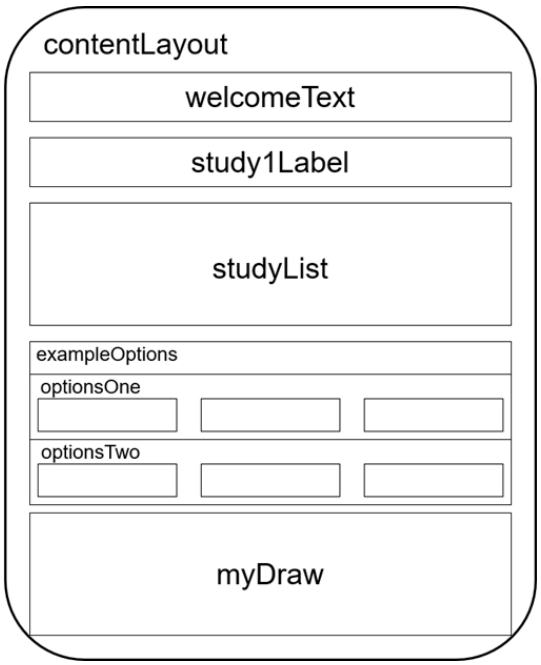
A megjelenített menügombokat kapcsoljuk össze különböző slotokkal, így biztosítva a különböző menüpontokra vonatkozó eltérő *contentLayout* tartalmat. Táblázat a menügombok kattintásával meghívott függvényekre:

Gomb	Slot
buttonVector[0]	*clickedOne()*
buttonVector[1]	*clickedTwo()*
buttonVector[2]	*clickedThree()*
buttonVector[3]	*clickedFour()*
buttonVector[4]	*clickedFive()*

Az alábbiakban a fenti slotok által létrehozott objektumokat táblázattal mutatom be, míg az elrendezést -ahol összetettebb- *contentLayout*ra vonatkozó ábrákkal szemléltetem, mivel csupán az változik, a főmenü gombjai nem.

- *clickedOne()*

Adattag típusa	Adattag neve
*QLabel**	*welcomeText*
*QLabel**	*study1Label*
*QListWidget**	*studyList*
*QVBoxLayout**	*exampleOptions*
*QHBoxLayout**	*optionsOne*
*QHBoxLayout**	*optionsTwo*
*DrawWidget**	*myDraw*
QVector<QPushb.>*	*optionButtons*



50. *ábra: Felosztás az egyes menüpont esetén*

Az optionButtons elemei az exampleGraph slothoz kapcsoltak.

- *clickedTwo()* *clickedThree()*

Adattag típusa	Adattag neve
*QLabel**	*welcomeText*
*QLabel**	*study1Label*
*QVBoxLayout**	*buttons2Layout*
QVector<QPushb.>*	*algoButtons*
*QLabel**	*study2Label*
*DrawWidget**	*myDraw*

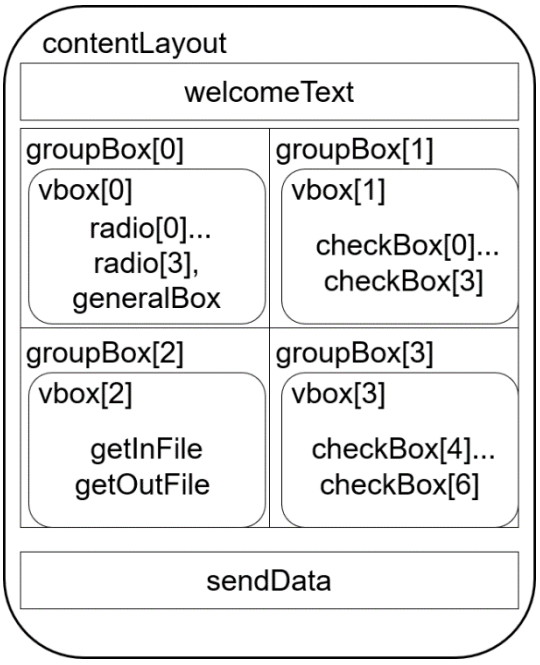
Adattag típusa	Adattag neve
*QLabel**	*welcomeText*
*QLabel**	*study1Label*
*QLabel**	*study2Label*
*QLabel**	*study3Label*
*QLabel**	*study4Label*
*QLabel**	*study5Label*
*QLabel**	*study6Label*
QVector<QPushB>*	*testButtons*

Az *algoButtons* gombjai a *clickedAlgo*, a *testButtons* gombjai a *downloadFile* slothoz kapcsoltak.

- *clickedFour()* *clickedFive()*

Adattag típusa	Adattag neve
*QLabel**	*welcomeText*
*QGridLayout**	*myGrid*
QVector<QGroupBox>*	*groupBox*
QVector<QRadioButton>*	*radio*
QVector<QCheckBox>*	*checkBox*
QVector<QVBoxLayout>*	*vbox*
*QPushbutton**	*sendData*
*QPushbutton**	*getInFile*
*QPushbutton**	*getOutFile*
*QGroupBox**	*generalBox*

Adattag típusa	Adattag neve
*QLabel**	*welcomeText*
*QLabel**	*study1Label*
*QLabel**	*study2Label*
*QLabel**	*study3Label*
*QLabel**	*study4Label*
*QListWidget**	*studyList*



51. ábra: Felosztás a négyes menüpont esetén

Összefoglalva a gombokhoz kötött slotokat:

Gomb	Slot
optionButtons[i]	*exampleGraph()*
algoButtons[i]	*clickedAlgo()*
testButtons[i]	*downloadFile()*
sendData	*runPython()*
generalBox	*displayClicked()*
getInFile	*getFilePath()*
getOutFile	*FileOutName()*

A slotok tevékenysége:

Függvény	Tevékenység
exampleGraph()	A myDraw mutató segítségével létrehozza a mainWindowhoz tartozó DrawWidgetet, és a gombtól függően egy beépített példagráfot rajzol ki.

clickedAlgo()	Párbeszédablakot nyit meg, ahol bemeneti számokat kér. A gombtól függően a megfelelő argumentumokkal hívja meg a számsorra az algoritmusok.py-t, aminek eredményét az output.txt-ből olvassa ki. A myDraw mutató segítségével létrehozza a mainWindowhoz tartozó DrawWidgetet, és a gombtól függően egy beépített példagráfot rajzol ki.
downloadFile()	A teszttáblázatot tölthetjük le a megfelelő URL-ről.
runPython()	A beállításoktól függően a megfelelő argumentumokkal hívja meg az algoritmusok.py programot, hibaüzenettel jelez, ha hiányzik valamilyen beállítás.
displayClicked()	A generalBox állapotától függően tesz elérhetővé két másik checkboxot.
getFilePath()	Párbeszédablakot nyit meg, ahol kattintással választhatunk ki fájl.
FileOutName()	Párbeszédablakot nyit, tárolja a megadott fájlnevet.

Az algortimusok.py program meghívásához használt argumentumok a *clickedAlgo()* esetén:

Argumentum sorszáma	Argumentum értéke	Mi alapján? Kattintott gomb
1	1	*algoButtons[0]*
	2	*algoButtons[1]*
	3	*algoButtons[2]*
	4	*algoButtons[3]*
	5	*algoButtons[4]*
2	-	Felhasználó által megadott számok

Tehát az első argumentum jelentése: melyik algoritmust alkalmazzuk az adatsorra.

Az algortimusok.py program meghívásához használt argumentumok a *runPython()* esetén:

Argumentum sorszáma	Argumentum értéke	Mi alapján?
1	0	Több beállítást akarunk megadni
2	0	Egyszerű eset (*radio*[0])
	1	Általános eset (*radio*[1])
	2	Általános eset második függvényopció (*radio*[2])
	3	Általános eset harmadik függvényopció (*radio*[3])
3	0/1	Az adott beállítás (*checkBox*[0])
4	0/1	Az adott beállítás (*checkBox*[1])
5	0/1	Az adott beállítás (*checkBox*[2])
6	0/1	Az adott beállítás (*checkBox*[3])
7	0/1	Az adott beállítás (*checkBox*[4])
9	0/1	Az adott beállítás (*checkBox*[5])
10	0/1	Az adott beállítás (*checkBox*[6])
11	szöveg	Bemeneti fájl elérési útvonala (*inPath*)
12	szöveg	Kimeneti fájl neve (*outFile*)

4.2.5 DrawWidget

Mint már korábban említettem, a *DrawWidget* osztályt a *QGraphicsView*ból származtatom. A *mainWindow* osztály tartalmaz egy *myDraw* nevű, *DrawWidget*re mutató pointert. Mivel egyszerre csak egy gráfot rajzolunk ki, ezen keresztül hozzuk létre

és semmisítjük meg az osztályt. Az osztály adattagja a *myScene*, mely a *QGraphicsScene* osztály pointere, erre fogunk rajzolni. A *DrawWidget* legtöbb metódusa a látótér mozgatására és az események (görgetés vagy kattintás) kezelésére vonatkozik.

A fő metódus, mely a tényleges kirajzolást végzi a *drawGraph* függvény. A paraméterek a számok és szülők, amelyek meghatározzák a kirajzolandó gráfot. Opcionális paraméter egy szöveg, amelyet illusztrációkhoz használhatunk, a számolás helyett feliratnak. A gráf csúcsait a *QGraphicsScene* beépített *addEllipse* metódusával hozzuk létre, a körök átmérői 40 képpontosak, egymástól 50 képpontra vízszintesen. A csúcsokba a számok értékét *QGraphicsSimpleTextItem*mel tüntetjük fel. Az éleket az *addPath* metódussal hozzuk létre, ahol a csúcsok közé, azok távolságának függvényében rajzolunk Bezier-görbét. Az élek végére a nyilakat *addPolygon* metódussal illesztjük (háromszöget), megfelelő szögben. A számolást segítő feliratot a *calculations* függvénnyel helyezzük a gráf alá.

4.3 Tesztelés

A Python kódrészre és a C++ kódrészre is készültek unittesztek. A backend bonyolultabb algoritmusait a tesztelés során a „brute-force” megközelítéssel kapott eredményekhez hasonlítottam. Emellett természetesen a fejlesztés során folyamatosan végeztem manuális tesztelést, ahol erre lehetőségem volt.

4.3.1 Manuális tesztelés

A felhasználói felület összes lehetséges funkcióját kipróbáltam. Szisztematikusan haladtam végig az egyes menüpontokon, és a helyileg választható interakciókat egyesével próbáltam ki. Az alkalmazás az elvárt működést nyújtotta.

4.3.2 Backend unittesztek

Ahogy az alkalmazásban is el vannak választva a különböző részek, úgy itt is teszteltem a beolvasásra használt függvényeket, és az adatokat feldolgozó függvényeket a különböző esetekre. A teszteléshez a *unittest2* Python könyvtárat használtam fel. A könyvtár által biztosított *assertEqual* és *assertIs* metódusok segítségével ismert fájlokból való beolvasás után ellenőriztem a programban tárolt adatok helyességét. A dimenziók, és a véletlenszerűen választott konkrét értékek is megfelelőek voltak. Teszteltem a

függvények visszatérési értékeként használt *to_return* osztályt, hogy helyesen hozzuk-e létre, és hogy megfelelően kérjük-e le az adatait. Ellenőriztem a tárolt adatok típusát, a helyes konvertálás érdekében. Néhány adatsorra, melyre ismertem az optimális fát, összehasonlítottam azt különböző algoritmusok esetén kapott eredményekkel. Nyilvánvalóan a nagyobb bemenetekre ezt nem ismerhetjük, ezért a unittesztekkel a naív algortimus helyes implementációjáról akartam megbizonyosodni, hogy kicsit nagyobb adatsorokra azzal hasonlíthassam össze a saját algoritmusomat, ezzel bizonyítva helyességét. Az ilyesfajta tesztelésről bővebben az „Egyszerű eset eredmények", „Általános eset eredmények" fejezetekben írok. A futtatott tesztek eredményei a results* nevű excel dokumentumokban találhatók.

```
andras@andras-K53E:~/Desktop/thesis/build-thesisfront-Desktop-Debug$ python tests.py
..........
----------------------------------------------------------------------
Ran 11 tests in 0.088s

OK
```

52. ábra: Python tesztelés

4.3.3 Frontend unittesztek

A felhasználói felületet a Qt keretrendszer *QtTest* könyvtárával teszteltem. Az osztály neve *unittest* lett, amit a *QObjectből* származtattam. A beépített *QCompare* és *QVerify* makrókat használtam, a *unittest* osztályt friend osztályként deklaráltam a *mainWindow* és *DrawWidget* osztályokon belül, hogy a privát metódusaikat is el tudjam érni. Különböző felhasználói tevékenységeket szimuláltam a *QTest mouseClick* metódusával, és ellenőriztem, hogy a szükséges objektumok léteznek-e. A folyamat közben ellenőriztem az ablak méretét, és hogy törlődnek-e a kérdéses objektumok. A *DrawWidget* osztály *calculations* metódusát ellenőriztem, ismert fákra megfelelően hozza létre a számolást magyarázó feliratot.

```
********* Start testing of unittest *********
Config: Using QtTest library 5.12.4, Qt 5.12.4 (x86_64-little_endian-lp64 shared (dynamic) release build; by GCC 9.2.1 20191008)
QDEBUG : unittest::initTestCase() Called before everything else.
PASS   : unittest::initTestCase()
PASS   : unittest::test_case1()
PASS   : unittest::mainWindowSize()
PASS   : unittest::mainWindowFirst()
PASS   : unittest::mainWindowSecond()
PASS   : unittest::mainWindowThird()
PASS   : unittest::mainWindowFourt()
PASS   : unittest::mainWindowFifth()
PASS   : unittest::simulateUse()
PASS   : unittest::testDrawWidget()
QDEBUG : unittest::cleanupTestCase() Called after everything else.
PASS   : unittest::cleanupTestCase()
Totals: 11 passed, 0 failed, 0 skipped, 0 blacklisted, 5843ms
********* Finished testing of unittest *********
```

53. ábra: Qt tesztelés

5. Összefoglalás

A bioinformatikai kutatások igazán fontos területet jelentenek ma, többek között a dolgozatomban érintett terület, a baktériumpopulációban megjelenő mutációk vizsgálata és ennek összekapcsolása a gráfokkal. A feladat megfelelően modellezhető egy tavalyi publikációban (Ismail & Tang, 2019) megjelent modellel. Kutatásom magában foglalta a modellre létező algoritmusok implementálását, új algoritmusok konstruálását, és ezek összehasonlítását. Az eredmények alapján az egyik algoritmusom gyakorlati szempontból is jól használható a keresett gráfok részeinek visszafejtésére. Az algoritmusokhoz egy felhasználói felületet implementálok, mellyel az alkalmazás könnyen kezelhetővé válik, és az eredmények szemléletesen kirajzolásra kerülnek.

5.1 További fejlesztési lehetőségek

A program backend részének továbbfejlesztése kutatási feladat. A már implementált heurisztikus algoritmusok összekapcsolása csökkenthetne a program futási idején. Az eredmények javítására ötlet szintjén felmerült a nagyobb költségű algoritmusok csupán lokális szintű alkalmazása.

A frontend rész továbbfejleszthető lenne több opcióval, mint például az egyszerű esetben, a felhasználó által megadható maximális futási idő. Az alkalmazás szemléltethetné szebben a megoldást reprezentáló gráfot, és kisebb bemenetű esetekben további funkció lehetne az algoritmusok lépéseinek grafikus felületen való lejátszása.

6. Köszönetnyilvánítás

A témával az Indiana University, „Global Talent Attraction” programja jóvoltából ismerkedtem meg. Köszönöm Dr. Haixu Tang és Wazim Ismail közreműködését, és a szimulált teszteseteket, melyeket rendelkezésemre bocsátottak. Köszönöm témavezetőm, Dr. Szabó László Ferenc felügyeletét, és az algoritmusok szemináriumot, ahol mélyrehatóbban foglalkozhattam a problémával. Az EFOP-3.6.3-VEKOP-16-2017-00002. számú projektben elvégzett szakmai feladat az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg.

7. Irodalomjegyzék

Barrick, J., Yu, D., Yoon, S., Jeong, H., Oh, T., Schneider, D., . . . Kim, J. (2009). Genome evolution and adaptation in a long-term experiment with escherichia coli. *Nature*, 461(7268): 1243-1247.

Behringer, M., Choi, B., Miller, S., Doak, T., Karty, J., Guo, W., & Lynch, M. (2018). Escherichia coli cultures maintain stable subpopulation structure during long-term evolution. *Proc. Nat. Acad. Sci.*, 115(20): E4642-4650.

Elena, S., & Lenski, R. (2003). Microbial genetics: evolution experiments with microorganisms: the dynamics and genetic bases of adaptation. *Nature Reviews Genetics*, 4(6): 457-469.

El-Kebir, M., Oesper, L., Acheson-Field, H., & Raphael, B. (2015). Reconstruction of clonal trees and tumor composition from multi-sample sequencing data. *Bioinformatics*, 31(12): 62-70.

Ismail, W., & Tang, H. (2019). Clonal reconstruction from time course genomic sequencing data. *BMC Genomics*, 20: 1002 https://doi.org/10.1186/s12864-019-6328-3.

Rainey, P., & Rainey, K. (2003). Evolution of cooperation and conflict in experimental bacterial populations. *Nature*, 425(6953): 72-76.