Analiza in načrtovanje bioloških vezij z ogrodjem GReNMlin ter ostale teme s področja biološkega procesiranja

Uredila prof. dr. Miha Moškon in prof. dr. Miha Mraz

Januar, 2025

Predgovor

Navodila za urejanje:

- Vaše datoteke se nahajajo v direktorijih `Skupina*`, kjer * predstavlja številko vaše skupine - glavna datoteka je textttmain.text.
- Če mape za vašo skupino še, jo ustvarite.
- Slike shranjujte v svoj direktorij.
- Vse labele začnite z znaki `g*:`, kjer * predstavlja številko vaše skupine.
- Pri referenciranju virov uporabite datoteko `references.bib`, ki se nahaja v korenskem direktoriju projekta.
- Pred dodajanjem novih virov v datoteko `references.bib` dobro preverite, če je vir mogoče že vsebovan v datoteki - v tem primeru se sklicujte na obstoječ vnos.

Seminarska dela so osredotočena na načrtovanje in optimizacijo bioloških logičnih gradnikov implementiranimi z gensko regulatornimi omrežji. Pri tem boste izhajali iz knjižnice GReNMlin[1], ki je namenjena delu s tovrstnimi gradniki. V svojih delih se lahko osredotočite na načrtovanje, analizo in optimizacijo novih bioloških logičnih vezij ali pa na razširitve predlagane knjižnice. Izberete si lahko tudi katerakoli druga orodja ali teme s področja biološkega procesiranja.

Ljubljana,
januar, 2025

prof. dr. Miha Moškon
prof. dr. Miha Mraz

[1] https://github.com/mmoskon/GRenMlin/

Zahvala

TODO

prof. dr. Miha Moškon in prof. dr. Miha Mraz, Ljubljana, januar 2025

Kazalo

Poglavje 1
Realizacija MUX in DEMUX z orodjem GReNMlin

Anže Arhar, Kristjan Kostanjšek in Nejc Ločičnik

Povzetek V seminarski nalogi obravnavamo implementacijo in analizo multiplekserjev (MUX) in demultiplekserjev (DEMUX) z uporabo orodja GReNMlin, Pythonovega paketa za modeliranje genskih regulacijskih omrežij. Predstavimo uspešne implementacije osnovnih komponent, kot so 2:1 MUX in 1:2 DEMUX in razširitve na bolj kompleksne različice (npr. 4:1 MUX in 1:4 DEMUX). V seminarski nalogi razvijemo in predstavimo tudi funkcije, ki zgenerirajo poljuben MUX oziroma DEMUX, glede na število vhodnih oziroma izhodnih linij, ki jih uporabnik lahko poda kot parameter funckije. Te implementacije lahko služijo kot dodatek oziroma nadgradnja orodja GReNMlin. Razvite funkcije tudi ovrednotimo in izpostavimo njihove morebitne slabosti. V zadnjem delu seminarske naloge pa predstavimo še način optimizacije izhoda modelov genskih regulacijskih omrežij z uporabo genetskih algoritmov, ki pa jih zaradi njihove dolgotrajnosti in omejene izboljšave rezultatov nismo vključili v samo implementacijo funkcij za generiranje generaliziranih MUX in DEMUX struktur.

1.1 Uvod

V uvodnem delu seminarske naloge bomo najprej opredelili temeljne pojme, ki so ključni za razumevanje vsebine, obravnavane v nalogi. To vključuje osnovne koncepte genskih regulacijskih omrežij (GRN) ter njihovo uporabo pri modeliranju digitalnih logičnih komponent. Poleg tega bomo predstavili orodje GReNMlin, ki

Anže Arhar
UL FRI, Večna pot 113, 1000 Ljubljana, e-mail: aa3178@student.uni-lj.si

Kristjan Kostanjšek
UL FRI, Večna pot 113, 1000 Ljubljana, e-mail: kk7609@student.uni-lj.si

Nejc Ločičnik
UL FRI, Večna pot 113, 1000 Ljubljana, e-mail: nl4952@student.uni-lj.si

omogoča simulacijo in analizo GRN, ter podali pregled osnovnih funkcionalnosti multiplekserjev in demultiplekserjev, na katerih temelji praktični del naloge.

1.1.1 Gensko regulatorno omrežje

V biološkem procesiranju je modeliranje genskih regulacijskih omrežij (GRN) ključen računalniški pristop za raziskovanje kompleksnih regulacijskih interakcij med geni, transkripcijskimi faktorji in vplivi iz okolja. Ta omrežja uravnavajo izražanje genov, kar omogoča celicam, da se prilagodijo okolju, diferencirajo v specifične celične tipe in ohranjajo homeostazo. Z uporabo računalniških tehnik za modeliranje GRN lahko raziskovalci simulirajo in napovedujejo vedenje bioloških sistemov, analizirajo njihovo robustnost na motnje ter načrtujejo sintetične regulacijske sisteme za biotehnološke ali medicinske aplikacije.

Modeliranje genskega izražanja je v zadnjih desetletjih postalo nepogrešljivo orodje v biološkem raziskovanju. Simulacije omogočajo preverjanje hipotez o izražanju genov, kar zmanjšuje potrebo po dolgotrajnih in dragih eksperimentih. Računalniško podprto modeliranje omogoča raziskovalcem, da preverijo pravilnost hipotez s simulacijami, kar zmanjšuje tveganje za neuspešne eksperimente. Pristop omogoča iterativni razvoj hipotez in eksperimentalnih načrtov, s čimer pospeši napredek pri razumevanju bioloških procesov.

1.1.2 Orodje GReNMlin

GReNMlin (Gene Regulatory Network Modeling) je Pythonov paket, zasnovan za konstruiranje, simulacijo in analizo osnovnih logičnih komponent, kot so multiplekserji, števci in druga digitalna vezja, z uporabo genskih regulacijskih omrežij. Temelji na sistemih diferencialnih enačb, ki opisujejo dinamiko interakcij med geni in njihovimi regulatorji. Te enačbe omogočajo natančno modeliranje procesov, kot so aktivacija, represija in sinteza genov, kar uporabnikom omogoča simulacijo vedenja GRN v času.

Postopek modeliranja logičnih komponent s pomočjo GRN v GReNMlinu je zasnovan na naslednjih korakih:

1. **Definicija omrežja**:Omrežje je definirano s specifikacijo species (vhodne in izhodne vrste), regulatorjev (povezave med vhodnimi vrstami) in produkta (izhodne vrste).
2. **Uporaba oblike SOP**: Na podlagi preučenih primerov smo ugotovili, da je najpreprostejši način za realizacijo osnovnih logičnih komponent preoblikovanje njihovih formul v obliko Sum of Products (SOP). Pri tem vsak regulator ustreza enemu produktu (AND), združevanje več regulatorjev v en produkt pa ustreza njihovi vsoti (OR). Funkcija NOR se doseže z inverzijo tipa regulatorja.

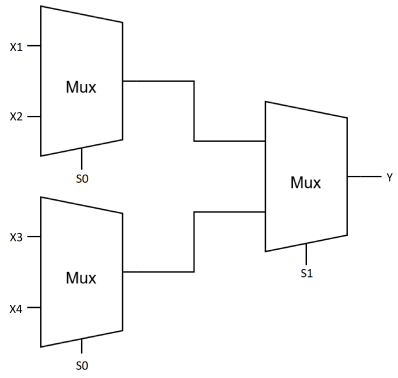
3. **Enostavna izdelava komponent**: Ta pristop omogoča preprost in pregleden način za izdelavo osnovnih logičnih komponent s pomočjo GRN. Poleg tega zagotavlja čiste signale že privzeto, brez potrebe po optimizaciji parametrov, kot sta kd ali n.

1.1.3 Multiplekser

Multiplekser je osnovno logično vezje, ki ima pomembno vlogo v računalniškem svetu. Njegova zasnova vključuje 2^n vhodnih signalov, n adresnih signalov (katere poimenujemo tudi kontrolni signali oziroma izbirne linije) in en izhodni signal. Njegovo osnovno delovanje temelji na izbiri enega izmed vhodnih signalov glede na vrednosti adresnih signalov in preslikava izbranega signala na izhod.

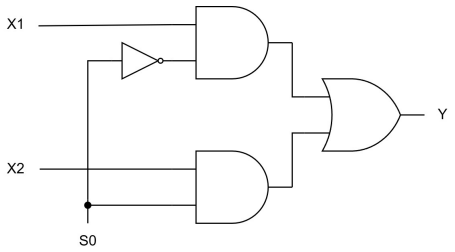
Uporaba multiplekserjev je široka in obsega različna področja. V digitalnih sistemih se multiplekserji uporabljajo za usmerjanje signalov, optimizacijo vodil in zmanjšanje števila potrebnih povezav. V procesorskih enotah se pogosto uporabljajo za izbiro med različnimi viri podatkov ali ukazov, v komunikacijskih sistemih pa omogočajo združevanje več signalov na eno prenosno pot, kar povečuje učinkovitost prenosa.

Hkrati multiplekserji predstavljajo poln funkcijski nabor. To pomeni, da z uporabo multiplekserjev lahko implementiramo poljubno logično funkcijo. Prav tako pa lahko z združevanjem več osnovnih (npr. 2:1) multiplekserjev ustvarimo večje in zmogljivejše multiplekserje, kot so 4:1, 8:1 in še večji. Slika (1.1) prikazuje primer kombiniranja osnovnih multiplekserjev, za izgradnjo kompleksnejšega multiplekserja.



Slika 1.1 Kaskadno združevanje multiplekserjev

Na sliki (1.2) je prikazana najbolj osnovna logična predstavitev multiplekserja, izvedena z uporabo NOT, AND in OR logičnih vrat, po kateri smo se tudi zgledovali pri realizaciji multiplekserja z orodjem GReNMlin.



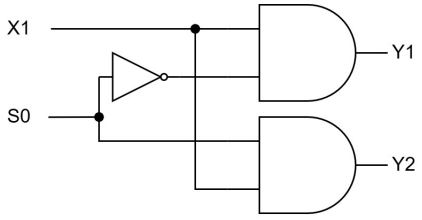
Slika 1.2 Osnovna realizacija multipleksorja (AND, OR, NOT).

1.1.4 Demultiplekser

Demultiplekser (DEMUX) je logično vezje, katerega delovanje je obratno od multiplekserja, vendar temelji na podobnem principu uporabe adresnih signalov. Medtem ko multiplekser omogoča izbiro enega izmed več vhodnih signalov in ga usmeri na izhod, demultiplekser prejme en vhodni signal in ga preusmeri na enega izmed izhodnih signalov. Zasnovan je torej z enim vhodnim signalom, n adresnimi signali in 2^n izhodnimi signali.

Demultiplekserji so nepogrešljivi v digitalnih sistemih, kjer je treba razporediti podatke ali signale iz enega vira na več ciljev. Uporabljajo se v komunikacijskih sistemih za distribucijo signalov v več kanalov, pri dekodiranju naslovov v pomnilniških enotah in v procesorjih za upravljanje z ukazi.

Slika (1.3) prikazuje osnovno logično predstavitev demultiplekserja, katero izvedemo z uporabo NOT in AND logičnih vrat. Tudi ta shema je služila za zgled pri implementaciji multiplekserja z orodjem GReNMlin.

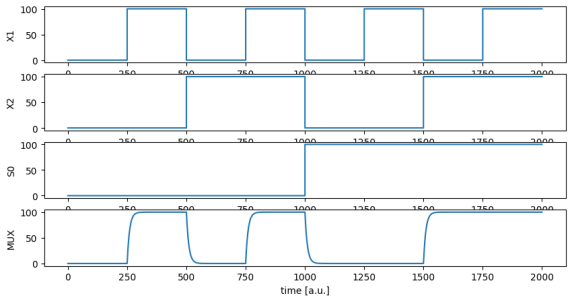


Slika 1.3 Osnovna realizacija demultipleksorja (AND, OR, NOT).

1.2 Metode

1.2.1 Realizacija multiplekserja

2:1 MUX smo z orodjem GReNMlin implementirali po sliki (1.2). Implementacija osnovnega 2:1 multiplekserja je od samega začetka delovala brez težav, kar je razvidno tudi na sliki (1.4). Ta uspešna zasnova je služila kot osnova za razvoj bolj kompleksnih multiplekserjev, kot sta 4:1 in 8:1 MUX, pri katerih prav tako nismo naleteli na nikakršne težave. Na podlagi teh rezultatov se kaže velik potencial za razvoj funkcije posplošenega multiplekserja. Takšna funkcija bi omogočala generiranje multiplekserjev poljubne kompleksnosti glede na število vhodnih linij, ki jih določi uporabnik. Posplošeni multiplekser bi lahko predstavljal dodatek oziroma izboljšavo obstoječega orodja GReNMlin.



Slika 1.4 Rezultati simulacije 2:1 multipleksorja.

1.2.1.1 Posplošen MUX

Posplošen MUX smo razvili po izrazu:

$$Y = (X_1 \cdot \overline{S}_1 \cdot \overline{S}_2 \cdot \ldots \cdot \overline{S}_{\log_2 N}) + (X_2 \cdot S_1 \cdot \overline{S}_2 \cdot \ldots \cdot \overline{S}_{\log_2 N}) + \ldots + (X_N \cdot S_1 \cdot S_2 \cdot \ldots \cdot S_{\log_2 N})$$

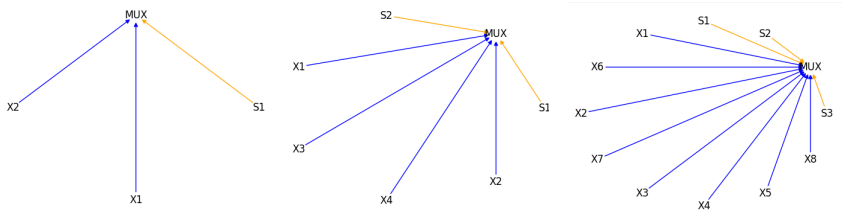
Sopdnji izsek programske kode prikazuje logiko delovanja funkcije, ki zgenerira poljuben MUX, definiran s parametrom N, ki predstavlja število vhodnih linij. Na začetku najprej preverimo, da je uporabnik za parameter N vnesel potenco števila 2, ki je večja od 1 (torej 2, 4, 8, 16, ...), nato definiramo ustrezno število vhodnih in izhodnih vrst (angl. species), na koncu pa generiramo željeni MUX po zgoraj definiranem izrazu.

```
# Check if N is power of 2
if (N < 2) or not (N and (N & (N - 1)) == 0):
    raise ValueError("N must be a power of 2
```

```
        and larger than 1 (e.g., 2, 4, 8, 16, ...)")

num_select_lines = int(np.log2(N))

# Initialize GRN
my_grn = grn.grn()

# Add input species for data lines X1, X2...
for i in range(1, N + 1):
    my_grn.add_input_species(f"X{i}")

# Add input species for select lines S1, S2...
for i in range(1, num_select_lines + 1):
    my_grn.add_input_species(f"S{i}")

# Add output species for MUX
my_grn.add_species("MUX", 0.1)

# MUX logic
for i in range(1, N + 1):
    # Determine binary representation
    binary_select =
        f"{i - 1:0{num_select_lines}b}"[::-1]

    # Define regulators for the current gene
    regulators = [{"name": f"X{i}", "type": 1,
        "Kd": 5, "n": 2}]
    for j, bit in enumerate(binary_select):
        regulators.append({
            "name": f"S{j + 1}",
            "type": 1 if bit == "1" else -1,
            "Kd":   5,
            "n":    3
        })

    # Define products for the current gene
    products = [{"name": "MUX"}]

    # Add the gene to the GRN
    my_grn.add_gene(10, regulators, products)
```

Testiranje funkcije za posplošen multiplekser je pokazalo, da funkcija deluje pravilno in skladno s pričakovanji. Omrežja, ki jih pridobimo z izvajanjem funkcije za različne vrednosti parametra N so prikazane na sliki 1.5.

Edina težava, na katero smo naleteli, je strmo naraščanje časa simulacije s povečevanjem parametra N (tabela 1.1). Medtem ko je simulacija 2:1 in 4:1 multi-



Slika 1.5 Omrežja 2:1, 4:1 in 8:1 multipleksorja.

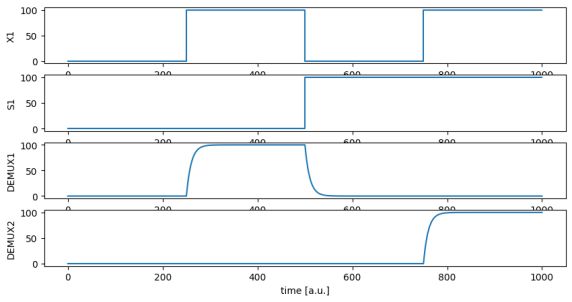
plekserja trajala manj kot sekundo, je simulacija 8:1 multiplekserja zahtevala že 30 sekund. Izvedba 16:1 multiplekserja pa je trajala tako dolgo, da časa nismo uspeli izmeriti. Po sedmih urah delovanja funkcije še vedno ni bilo videti, da bi se simulacija kmalu zaključila, zato smo jo ročno prekinili. Funkcijo za posplošen multiplekser bi bilo torej smiselno uporabljati le za osnovnejše multiplekserje (2:1, 4:1 in 8:1 MUX).

N	Čas za izvedbo [s]
2	0.12
4	0.43
8	33.86

Tabela 1.1 Čas izvedbe glede na velikost N multipleksorja

1.2.2 Realizacija demultipleksorja

2:1 DEMUX smo implementirali po sliki (1.3). Rezultati testiranja so prikazani na sliki (1.6).



Slika 1.6 Rezultati 1:2 demultipleksorja.

Tudi pri implementaciji demultiplekserjev nismo naleteli na nobene težave. Brez problemov smo razvili tudi kompleksnejše (4:1, 8:1, ...) demultiplekserje. Tudi to-

krat smo opazili potencial, za razvoj funkcije za generiranje posplošenega demultiplekserja, ki bi se poleg posplošenega multiplekserja lahko vključila kot izboljšava orodja GReNMlin.

1.2.2.1 Posplošen DEMUX

Posplošen DEMUX smo razvili po sledečih izrazih:

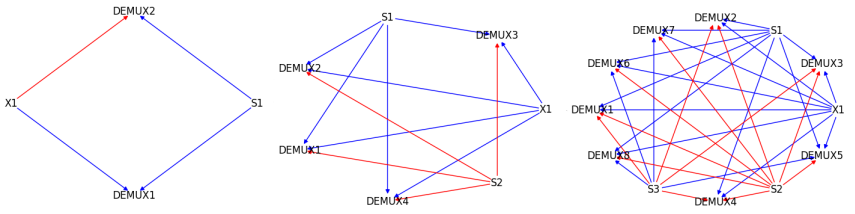
$$Y_1 = X_1 \cdot \overline{S}_1 \cdot \overline{S}_2 \cdot \ldots \cdot \overline{S}_{\log_2 N}$$
$$Y_2 = X_1 \cdot S_1 \cdot \overline{S}_2 \cdot \ldots \cdot \overline{S}_{\log_2 N}$$
$$\vdots$$
$$Y_N = X_1 \cdot S_1 \cdot S_2 \cdot \ldots \cdot S_{\log_2 N}$$

Spodnji izsek programske kode, prikazuje našo implementacijo funkcije. Funkcija deluje na zelo podobnem principu, kot funkcija za posplošen MUX, torej najprej preverimo ali je uporabnik vnesel ustrezen parameter N, nato definiramo vhodne in izhodne vrste, sledi pa DEMUX logika, ki po zgoraj definiranem logičnem izrazu ustrezno nastavi regulatorje.

```
# Check if N is power of 2
if (N < 2) or not (N and (N & (N - 1)) == 0):
    raise ValueError("N must be a power of 2
        and larger than 1 (e.g., 2, 4, 8, 16, ...)")

num_select_lines = int(np.log2(N))

# Initialize the GRN
my_grn = grn.grn()

# Add input species X1 and S1, S2, ..., Slog2(N)
my_grn.add_input_species("X1")
for i in range(1, num_select_lines + 1):
    my_grn.add_input_species(f"S{i}")

# Add output species for each Y_i
for i in range(1, N + 1):
    my_grn.add_species(f"DEMUX{i}", 0.1)

# DEMUX logic
for i in range(1, N + 1):
    # Determine binary representation
    binary_select =
        f"{i - 1:0{num_select_lines}b}"[::-1]
```

```
# Define regulators
regulators = [{'name': 'X1', 'type': 1,
    'Kd': 5, 'n': 2}]
for j, bit in enumerate(binary_select):
    regulators.append({
        "name": f"S{j + 1}",
        "type": 1 if bit == "1" else -1,
        "Kd": 5,
        "n": 3
    })

# Output gene for DEMUX_i
products = [{'name': f"DEMUX{i}"}]

# Add the gene to the GRN
my_grn.add_gene(10, regulators, products)
```

Tudi tokrat je funkcija delovala skladno s pričakovanji. Omrežja, ki jih pridobimo z izvajanjem funkcije za različne vrednosti parametra N, so prikazane na sliki 1.7.



Slika 1.7 Omrežja 1:2, 1:4 in 1:8 demultipleksorja.

Poleg tega pa je funkcija delovala hitreje, kot funkcija za posplošen MUX. Čas izvedbe je še vedno naraščal s parametrom N, vendar tokrat manj strmo. Čas za izvedbo funkcije, glede na velikost parametra N je podan v spodnji tabeli (1.2).

N	Čas za izvedbo [s]
2	0.11
4	0.14
8	0.36
16	2.12
32	16.22

Tabela 1.2 Čas izvedbe glede na velikost N demultipleksorja

1.2.3 Optimizacija parametrov

Za izboljšavo naših implementacij smo uporabili genetske algoritme. Ti so nam omogočili optimizacijo parametrov K_d in n regulatornih elementov.

Simulacije agentov v algoritmu smo ocenili s cenilno funkcijo

$$f(s) = \frac{1}{\sum_{i=1}^{N} |s_i - o_i| + \varepsilon}$$

, kjer N predstavlja število diskretnih korakov simulacije, $\varepsilon = 10^{-6}$. Ta na diskretnih korakih izračuna absolutno razliko med optimalno (o) in simulirano (s) vrednostjo.

Z našo implementacijo optimizacije z genetskimi algoritmi smo dosegli le malo boljše rezultate kot tiste, ki smo jih dobili z ročno optimizacijo parametrov. Predvidevamo, da bi za boljše rezultate morali dokaj povečati število generacij in velikost populacije. Po začetnih poskusih povečanja kateregakoli parametra na primeru 2:1 MUX, se čas optimizacije povečuje v rangu več ur. Sklepno smo se odločili, da naši rezultati ustrezajo zadanim ciljem in dodatna optimizacija parametrov ne prinese dovolj dobrih izboljšav glede na vložen čas.

1.3 Zaključek

V seminarski nalogi smo v programskem jeziku Python s pomočjo orodja GRenMlin implementirali enostavne strukture multiplekserjev in demultiplekserjev. Zaradi uspešnega delovanja teh osnovnih struktur smo se odločili razviti funkcije, ki omogočajo samodejno generiranje želenih MUX in DEMUX struktur glede na parametre, ki jih določi uporabnik. Medtem ko funkcija za generiranje DEMUX struktur uspešno zgenerira tudi kompleksne demultiplekserje (npr. 1:32 DEMUX) v sprejemljivem času, smo pri multiplekserjih naleteli na težave. Simulacija za bolj kompleksne multiplekserje, kot je 16:1 MUX, se je izkazala za preveč časovno zahtevno, kar omejuje praktično uporabnost.

Poleg tega smo predstavili postopek uporabe genetskih algoritmov za optimizacijo izhodov multiplekserjev in demultiplekserjev. Čeprav ti algoritmi ponujajo zanimivo možnost izboljšav, jih zaradi njihove dolgotrajne izvedbe in relativno majhnega vpliva na kakovost rezultatov nismo vključili v implementacijo.

V prihodnjem razvoju bi lahko raziskali metode za hitrejše generiranje kompleksnejših multiplekserjev in učinkovitejšo optimizacijo parametrov, da bi dosegli bolj optimalne rezultate v realnem času.

1.4 Doprinosi avtorjev

Anže Arhar	Optimizacija parametrov
Kristjan Kostanjšek	Posplošitev MUX in DEMUX
Nejc Ločičnik	Osnovni 2-8:1 MUX in 1:2-4 DEMUX

Literatura