Usporedba rješenja problema N kraljica u SWISH-u

Simić, Ivan

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Simić

Usporedba rješenja problema N kraljica u SWISH-u

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Simić

Matični broj: 0016152602

Studij: Informacijski i poslovni sustavi

Usporedba rješenja problema N kraljica u SWISH-u

ZAVRŠNI RAD

Mentorica:

Vlatka Sekovanić, mag. educ. inf.

Varaždin, srpanj 2024.

Ivan Simić

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U ovom radu opisani su temelji logičkog programiranja, vrste Prologa, te mogućnosti sinergije SWI-Prologa i Pythona. Isto tako, opisan je i problem N kraljica jer se on koristi u prikazu Prologove funkcionalnosti. Unutar SWI-Prologove online platforme SWISH izrađena su programska rješenja koja opisuju temeljnu sintaksu SWI-Prologa kao i moguća rješenja promatranog problema N kraljica. Programska rješenja u kontekstu problema N kraljica međusobno su uspoređivana na više razina – na razini SWI-Prologa, zatim na razini Pythona, te na kraju na razini spoja SWI-Prologa i Pythona. Na temelju navedenih usporedbi uočene su brojne prednosti i nedostaci pojedinih programskih rješenja kao što su kompleksnost, funkcionalnost, kapaciteti i tako dalje. Posebno se izdvaja kombinirano programsko rješenje SWI-Prologa i Pythona u kojemu se može vidjeti kako se, dva suštinski različita programska jezika, međusobno nadopunjuju pri čemu se kompleksnost programa može svesti na jednostavniju razinu.

Ključne riječi: swi-prolog; swish; problem n kraljica; python; logičko programiranje

Sadržaj

1. Uvod

Cilj ovog rada je prikaz mogućnosti logičkog programskog jezika SWI-Prologa, uključujući njegovu povijest, strukturu, način rada i glavne karakteristike. Radi se o deklarativnom programskom jeziku koji je puno drugačiji od danas najpopularnijih jezika na tržištu, zbog čega i zahtjeva ekstenzivan opis. Za ovu svrhu bit će opisan i primijenjen problemski zadatak N kraljica, koji se već koristi za prikaz funkcionalnosti ostalih programskih jezika, uključujući i SWI-Prolog.

Kako bi se problem N kraljica mogao pravilno koristiti u izradi rješenja, proučit će se njegova logika i struktura, te primjerom pokazati kako se on može riješiti matematički. Nakon toga opisati će se SWI-Prolog rješenje izrađeno unutar SWISH platforme, te pozitivni i negativni aspekti tog rješenja. Usporedbom izrađenog rješenja sa ostalim pronađenim rješenjima prikazati će se različiti načini programiranja unutar SWI-Prolog jezika, odnosno kako se isti problem može riješiti na više načina.

Na kraju će se sa odabranim problemom prikazati razlike između SWI-Prologa i jednog od najpopularnijih proceduralnih programskih jezika, Pythona, kako bi se dublje dočarale razlike između njih. Također, pokazat će se kako se ta dva različita programska jezika mogu spojiti, što omogućava korištenje njihovih najboljih karakteristika zajedno. Moći će se vidjeti kako je SWI-Prolog u stanju unaprijediti postojeće programe svojom sintaksom.

2. Prolog

Kako je i bilo navedeno, u ovom radu glavnu ulogu ima programski jezik Prolog i njegovo rješavanje problema N kraljica. Ovo poglavlje koristit će se kao uvod u Prolog i njegove vrste, te uvod u specifičnu implementaciju Prologa koja će se koristiti u rješavanju odabranog problema. No, za bolji uvod i prikaz Prologa, prvo će se govoriti o samoj vrsti jezika u koju on spada, a to su logički programski jezici.

2.1. Logičko programiranje

Logički programski jezici vrsta su deklarativnih jezika koji su napisani u obliku formalne logike. U ovim jezicima programi se pišu sa logičkim iskazima koji se zovu predikati, kako bi se stvorile upute za izvođenje programa. Najviše se primjenjuju u korist učenja programiranja, razvijanja umjetne inteligencije, za strojno učenje i upravljanje bazama podataka [1, 2].

Prvi programski jezik ove vrste bio je Prolog kojeg su 1972. razvili informatičari Alain Colmerauer i Phillipe Roussel u projektu kojem je cilj bio stvaranje programskog jezika koji za programiranje koristi prirodni jezik i logiku. Zbog ovog razloga neki su ljudi za sam koncept logičkog programiranja koristili naziv Prolog, unatoč tome što oni nisu sinonimi [3]. Danas zato postoji više različitih logičkih jezika, pa ne dolazi tako često do ove zabune. Prolog je ostao najkorišteniji logički programski jezik, a najpoznatiji uz njega su Datalog i Answer Set Programming.

Ovi jezici svrstani su pod deklarativne jezike zbog načina kako se u njima programi izvode. Naime, u ostalim programskim jezicima navode se naredbe jedna po jedna. Potrebno je za svaku naredbu detaljno opisati njenu funkciju i interakciju sa ostalim naredbama kako bi program mogao pravilno funkcionirati. Kod logičkih, tj. deklarativnih jezika programer opisuje željene ciljeve i što je programu dostupno na putu do tog cilja. Nigdje se u programu ne navodi kako se mora doći do tog cilja, već sam program treba koristiti dostupne informacije da do njega dođe [1, 2]. Dakle, programer u program unosi bazu znanja, korisnik programu zadaje nekakav upit, a program treba taj upit izvršiti sam sa njemu dostupnom bazom znanja.

Logički programski jezici poznati su po tome što ih je lakše stvarati i uređivati. U njima se može navoditi puno različitih pravila i ograničenja koja se mogu lako i promijeniti ako se zahtjevi brzo izmjenjuju. Također, logički se programi, zbog svoje strukture, lako mogu ponovo upotrebljavati u drugim programima slične namjene [1, 2]. Iz ovoga isto slijedi da se logički programi lakše međusobno povezuju. Nažalost, logički jezici danas nisu u širokoj upotrebi zbog sljedećih razloga: programi se sporo izvode, programi nisu i ne mogu biti previše

kompleksni, te neki jezici nisu u stanju povezati svoje implementacije sa ostalim programskim jezicima različitih vrsta [4]. Zato se logički jezici većinom koriste samo u onim specifičnim slučajevima gdje će oni najbolje funkcionirati.

Za razliku od jezika koji su bazirani na Turingovom stroju, logički jezici koriste se logikom prvog reda koja je ograničena Hornovim klauzulama (eng. *Horn Clause*) kako bi izvršavali Turing funkcije. Hornova klauzula je logička formula koja se sastoji od implikacije i najviše jednog pozitivnog literala. Prema [5] ako uzmemo da je $\{A, \neg B_i, \dots, \neg B_n\}$ Hornova klauzula, tada je njen oblik sljedeći:

$$A \leftarrow B_1, \dots, B_n = programska\ klauzula$$

$$A \leftarrow true = činjenica$$

$$false \leftarrow B_n = ciljna\ klauzula$$

Ova formula prikazana je kao deklarativna funkcija u kojoj se za dokazivanje glave A prvo treba dokazati tijelo B_n. Ovako su zapravo složeni svi logički programi; programske klauzule predstavljaju pravila programa, a zajedno sa činjenicama stvaraju skup koji se zove baza znanja. Taj skup predstavlja jedan logički program. Takav program odgovara na upite koji su u njega uneseni u obliku ciljnih klauzula [5].

2.2. Vrste Prologa

Prolog je deklarativni logički jezik baziran na logici prvog reda. Radi se o jeziku koji funkcionira na temelju algoritma dubinskog pretraživanja (eng. *depth-first search algorithm*), što znači da će se prvo izvršavati klauzule upisane na vrhu programa, a u samim klauzulama literali se čitaju s lijeva na desno [5]. Riječ Prolog akronim je za „*PROgramming in LOGic*“.

Većina već napomenutih svojstva logičkih programskih jezika vrijede i za Prolog, pošto je on bio prvi takve vrste. Glavna razlika je u načinu na koji se sintaksa upisuje. Kako je već bilo navedeno, programi se pišu u obliku Hornovih klauzula, ali na način da se implikacija ← zamjeni sa simbolom :−, a umjesto konjunkcije ∧ piše se zarez [5]. Kako bi se bolje dočarali dijelovi sintakse i sama funkcionalnost, napravit će se kratki program na primjeru koji se često koristi u Prologu, obiteljsko stablo:

```
roditeljOd(zoran,ivan). %činjenica
roditeljOd(milos,zoran). %činjenica
otac(X,Y):-roditeljOd(X,Y). %pravilo
djed(X,Y):-roditeljOd(X,Z),roditeljOd(Z,Y). %pravilo
```

U ovom primjeru može se vidjeti mali program koji je u stanju odrediti koji član obitelji je nekome otac, a koji je djed. Baza znanja ovog programa sastoji se od četiri klauzule, odnosno dvije činjenice i dva pravila. Činjenice se koriste za zapisivanje podataka, a pravila služe kao operacije koje koriste te podatke. Ime svake klauzule naziva se predikat. Ovaj program sastoji se od tri predikata. Ime predikata zapisuje se sa brojem argumenata koji on sadrži ili prima. Dakle, u ovom programu nalaze se sljedeći predikati: *roditeljOd/2*, *otac/2* i *djed/2*.

Važno je napomenuti da je u pisanju sintakse unutar Prologa bitno paziti na velika i mala slova. Objekti pisani velikim početnim slovom zovu se varijable. Tim objektima se vrijednost sprema samo u njihovim klauzulama. Zato se objekt X može ponavljati u više različitih klauzula. Objekt X u predikatu *otac/2* i *djed/2* ne predstavlja istu vrijednost. Varijable se najčešće koriste u pisanju argumenata vezanih uz neko pravilo [6]. Za razliku od toga objekti pisani malim početnim slovom, nazvani atomi, se uvijek referenciraju na mjesto gdje je taj objekt prvi put napisan. Zato se atomi najčešće koriste u pisanju predikata i argumenata vezanih za neku činjenicu [6]. Simbol % predstavlja komentar u programu. On ne utječe na njegovo izvođenje, te može biti pisan u bilo kakvom obliku.

Ovakva Prolog sintaksa bazirana je na ISO standardu stvorenom za Prolog 1995. godine. Međutim, nisu sve implementacije Prologa bazirane na ovom standardu [7]. Razlog tome je što su se već od prvog Prologa 1972. godine razvijale različite implementacije. Prva generalno prihvaćena verzija bio je DEC-10 Prolog iz 1975. godine koji je svoj naziv dobio po DEC-10 računalu na kojem se jedino i mogao kompajlirati. Taj Prolog bio je prvi na istoj razini korisnosti i brzine kao i ostali programski jezici tada. Sintaksa ovog Prologa dobila je naziv Edinburgh sintaksa, te su se po njoj razvijale daljnje implementacije i sam ISO standard [8].

Sljedeći bitan dodatak bio je novi način kompajliranja sa Warren Abstract Machine-om (WAM). Ovaj kompajler, stvoren 1983. godine, uvelike je smanjio količinu potrebne sintakse za izvršavanje Prolog programa. Također je omogućio Prologu da bude više prijenosan jer nije bilo potrebno koristiti DEC-10 računalo. Prva programska implementacija WAM sustava bila je za Prolog sustav Quintus. Ovaj sustav se danas još uvijek koristi kao standard u izradi Prologa zbog njegove pristupačnosti [8]. Drugi dio ISO standarda izrađen je 2000. godine, ali zbog toga koliko je sintaksa i implementacija u njemu bila različita od većine Prologa, koji su i dalje bili bazirani na Quintus sustavu, taj standard je većina ljudi ignorirala tijekom razvijanja vlastitih Prologa [7].

Danas se različite verzije Prologa još uvijek razvijaju. Trenutno na tržištu ima puno različitih verzija. Razlikuju se po dostupnosti, mogućnostima i sintaksi koju koriste. Prema [8] i [9] neke od najkorištenijih implementacija Prologa su sljedeće:

- **B-Prolog**: koristi ISO sintaksu. Brzo izvršava programe. Omogućava kvalitetno programiranje sa ograničenjima i implementaciju slučajeva. Sučelje sa jezicima C, Java.
- **Ciao**: koristi ISO sintaksu i ostale standardne sintakse. Jako razvijeno sučelje koje prikazuje izvođenje programa i greške u kodiranju. Omogućava korisniku da implementira vlastite ekstenzije u program. Podržava module i paralelno programiranje. Sučelje sa jezicima C, Java, Python, JS.
- **ECLiPSe**: omogućava korištenje više sintaksi u isto vrijeme. Razvijen za programiranje sa ograničenjima. Podržava module i paralelno programiranje. Sučelje sa jezicima C, Java, Python, PHP.
- **GNU Prolog**: koristi ISO sintaksu. Fokus na brze i lagane programe. Jednostavno izgrađen i lagan za korištenje. Sučelje sa jezicima C, Java, PHP.
- **SICStus Prolog**: koristi ISO i Quintus sintaksu. Za komercijalnu upotrebu. Kvalitetno programiranje sa ograničenjima. Podržava module. Sučelje sa jezicima C, Java, .NET, Tcl/Tk.
- **SWI-Prolog**: koristi ISO i Edinburgh sintaksu. Podržava module i paralelno programiranje. Sučelje sa jezicima C, C++, Java. Više o ovoj implementaciji u sljedećem poglavlju.
- **YAP**: kompatibilan sa ISO, Edinburgh, Quintus i SICStus sintaksom. Fokus na integraciju sa ostalim Prolog implementacijama i programskim jezicima. Za izvođenje velikih programa i uvođenje baza podataka. Podržava module. Sučelje sa jezicima C, Python, R.

Slijede objašnjenja pojedinih pojmova iz gornje liste. **Programiranje sa ograničenjima** odnosi se na implementaciju predikata koji se izvode samo u određenim uvjetima. Najbolji primjer su operacije nejednakosti gdje se neki dijelovi programa ne izvode ako zadana varijabla nema traženu vrijednost. Također je moguće programiranje sa *true* i *false* varijablama [10]. **Implementacija slučajeva** slična je kao i programiranje sa ograničenjima, samo što se ovdje radi o cijelim dijelovima programa ili o petljama koje se izvode ako je ispunjen uvjet ili ako se izvede nekakva akcija.

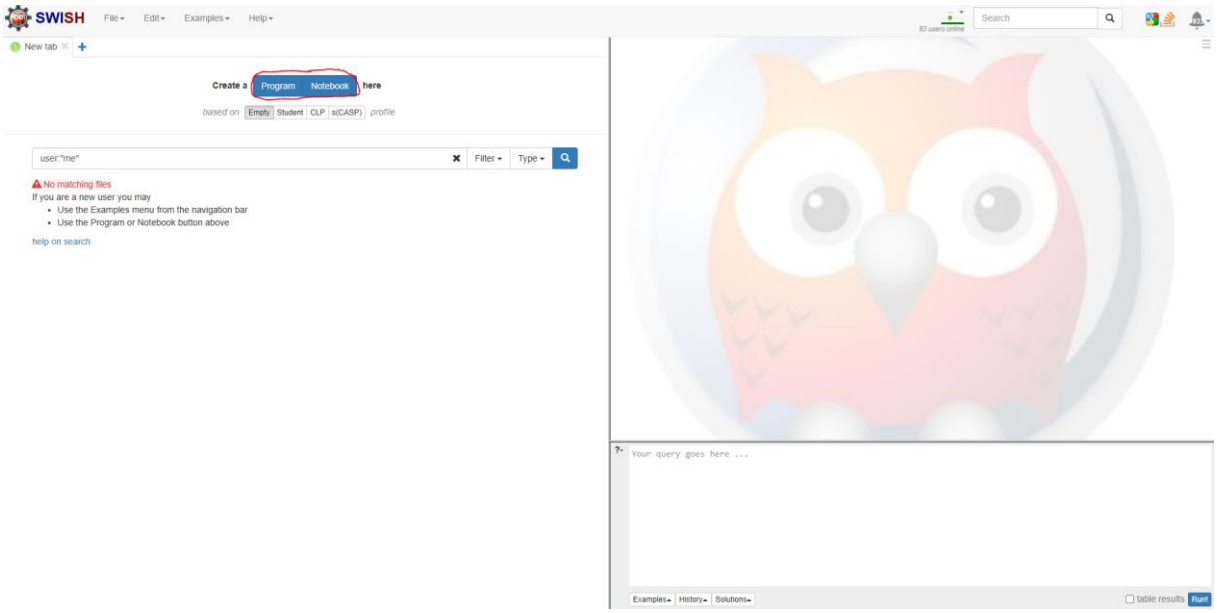
Sučelje sa jezicima omogućava Prolog programu da se spoji sa drugim podržanim jezikom kako bi se riješili problemi koji nisu prilagođeni Prologu. Ovo može biti izvršeno tako da se program izvršava sa više različitih sintaksi ili da jedan jezik poziva drugi sa svojim funkcijama [8]. **Moduli** su skupovi predikata koji se mogu pozivati u programu. Definirani predikati mogu se sačuvati u modulu i kasnije koristiti u drugim programima bez da se oni ponovo definiraju, nego ih se može odmah pozvati i koristiti. Jedan modul može u svojoj implementaciji koristiti druge module [11]. **Paralelno programiranje** dodaje mogućnost

izvođenja više funkcija u isto vrijeme tijekom pokretanja programa. Ovaj proces oduzima puno memorije, pa je potrebno optimizirati cijeli program kako bi se više niti programa moglo izvoditi u isto vrijeme [8].

2.3. SWISH (SWI-Prolog)

SWI-Prolog, poznat i po svojoj online verziji SWISH, jedna je od danas najkorištenijih implementacija jezika Prolog stvorena za generalno korištenje. Glavni cilj ove implementacije je povezanost sa ostalim sustavima. Zato ova implementacija podržava korištenje HTTP protokola i čitanje drugih tipova podataka (npr. RDF, HTML, XML, JSON, YAML). Kompatibilan je sa određenim implementacijama Prologa i omogućava programiranje sa ograničenjima [8].

Pošto je SWISH odabrana platforma za rješavanje problema, u njoj će se detaljno objasniti sintaksa SWI-Prolog implementacije. Torbjörn Lager prvi je izradio tu platformu u počast SWI-Prologu. Trenutnu verziju izradio je Jan Wielemaker koji ju i dalje održava [12]. Platforma je bila izrađena kako bi korisnicima pružala brzo dostupnu implementaciju SWI-Prologa, te tako povećala utjecaj i korištenje Prolog jezika. Danas se najčešće koristi u edukacijske svrhe.



Slika 1: Početni ekran SWISH-a (autorski rad)

Već na početnom ekranu SWISH platforme korisniku je omogućeno korištenje jezika bez registracije ili dodatnih naplata. Korisnik se može prijaviti u sustav kako bi spremio svoje programe na server, ali ih može i preuzeti lokalno na računalo, te ponovo uvesti u kasnijoj

instanci. Na početku programiranja moguće je odabrati već postojeći predložak za programiranje ili započeti sa praznim programom.

Program može biti pisan u obliku klasičnog Prolog programa koji podržava SWI-Prolog sintaksu, a dostupno je programirati i u obliku bilježnice (eng. *Notebook*) gdje se može koristiti više različitih jezika. Sintaksa, odnosno baza znanja programa piše se na lijevoj strani, a desna strana služi za upisivanje upita i ispis njihovih rezultata. Za prikaz programiranja na platformi koristit će se primjer obiteljskog stabla iz prijašnjeg poglavlja.

```
1 %baza znanja
2
3 %činjenice
4 roditeljOd(zoran,ivan).
5 roditeljOd(milos,zoran).
6
7 %pravila
8 otac(X,Y):-roditeljOd(X,Y).
9 djed(X,Y):-roditeljOd(X,Z),roditeljOd(Z,Y).
```

Slika 2: Program obiteljsko stablo (autorski rad)

Sama platforma SWISH označava pojedine dijelove sintakse različitim bojama kako bi korisniku odmah bilo jasno o kakvim vrstama podataka se radi. Komentari pisani sa % simbolom označeni su zelenom bojom, te oni ne utječu na izvođenje programa, nego ih korisnik može koristiti za bolje organiziranje strukture programa. Atomi su označeni tamnožutom bojom, a varijable tamnocrvenom. Predikati su u pravilu crni, ali ako ostanu neiskorišteni u programu, tj. nigdje ih se u programu ne poziva, onda su crveni. Takvi predikati su najčešće oni koje se poziva upitom kako bi se program izveo.

Stvoreni program može se koristiti za provjeravanje odnosa među članovima obitelji ili za pronalaženje podataka. Upit se upisuje pored simbola ?−, a rezultat upita ispisuje se iznad njega. Ovisno o točnosti tvrdnje, rezultat može biti true ili false.

Slika 3: Rezultati upita (autorski rad)

Iz stvorenih upita može se vidjeti kako program pronalazi rješenja. Ako se u upitu predikatu daju atomi kao argumenti, program ispituje točnost te tvrdnje. Iterira se kroz svaki predikat sa istim imenom, u ovom slučaju to je predikat *otac/2*. On je pravilo koje ispituje odnos članova prema dobivenim argumentima na način da pozove predikat *roditeljOd/2*. To pravilo će proći kroz svaki takav predikat, tj. činjenicu i tražiti postoji li u bazi činjenica sa traženim atomima unutar argumenata kakvi su primljeni u upitu. U slučaju da postoje, program vraća *true*. Inače vraća *false*.

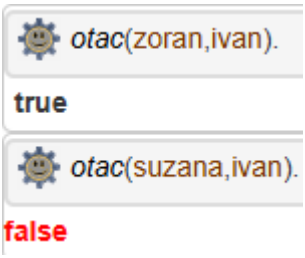
Ako se u upitu predikatu daju atom i varijabla kao argumenti, proces će biti sličan kao i prije, samo što će ovaj put program tražiti gdje se u bazi nalazi predikat sa traženim atomom. U ovom primjeru prvi argument je atom „milos“, a drugi varijabla X. Program će sada prema pravilu *otac/2* tražiti svaki predikat *roditeljOd/2* gdje je prvi argument „milos“. Za svaki takav predikat koji se pronađe program će ispisati njegov drugi argument u obliku varijable X. Ovdje će se u varijablu X pohraniti „zoran“, jer jedina činjenica *roditeljOd/2* kojoj je prvi argument „milos“ kao drugi argument ima atom „zoran“.

Program se također može koristiti za ispisivanje svih članova iz određene kategorije. Za prikaz te funkcionalnosti prvo će se program proširiti. Dodaju se novi članovi i kategorije.

```
%baza znanja

%činjenice
roditeljOd(zoran,ivan).
roditeljOd(zoran,luka).
roditeljOd(zoran,vedran).
roditeljOd(suzana,ivan).
roditeljOd(suzana,luka).
roditeljOd(suzana,vedran).
roditeljOd(milos,zoran).
roditeljOd(kata,zoran).
roditeljOd(ivo,suzana).
roditeljOd(mira,suzana).
spol(zoran,m).
spol(milos,m).
spol(ivo,m).
spol(suzana,z).
spol(kata,z).
spol(mira,z).

%pravila
otac(X,Y):-roditeljOd(X,Y),spol(X,m).
majka(X,Y):-roditeljOd(X,Y),spol(X,z).
djed(X,Y):-roditeljOd(X,Z),roditeljOd(Z,Y),spol(X,m).
baka(X,Y):-roditeljOd(X,Z),roditeljOd(Z,Y),spol(X,z).
roditelji(X):-roditeljOd(Y,X),write(Y),nl,fail.
roditelji(_).
braca(X):-otac(Y,X),otac(Y,Z),\=(X,Z),write(Z),nl,fail.
braca(_).
starci(X):-djed(Y,X),write(Y),tab(2),fail.
starci(X):-nl,baka(Y,X),write(Y),tab(2),fail.
starci(_).
```

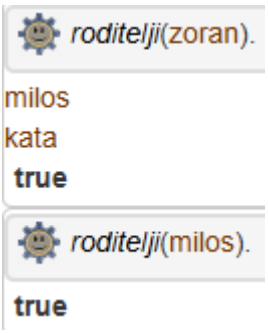
Slika 4: Nadograđeni program obiteljsko stablo (autorski rad)

U ovoj verziji programa dodani su novi članovi obitelji. Kako bi se roditelji razlikovali po spolovima dodana je nova činjenica *spol/2* u kojoj je zapisan član i njegov spol. Taj predikat dodan je u pravila *otac/2*, *majka/2*, *djed/2* i *baka/2* kako bi se u svakom moglo odrediti kojeg spola mora biti roditelj da zadovolji uvjet. Prema pravilu *otac/2*, osoba X nije otac ako nije muškog spola. Suprotno vrijedi i kod pravila *majka/2*, gdje osoba X nije majka ako nije ženskog spola.



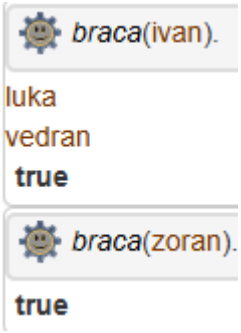
Slika 5: Pravilo otac (autorski rad)

Dodana su nova pravila *roditelji/1*, *braca/1* i *starci/1* kojima je funkcija ista, a to je za određenog člana ispisati njemu povezane članove. Svako od ovih pravila implementirano je na sličan način. Pravilo *roditelji/1* za jednog člana X ispisuje sve njegove roditelje. Predikat *roditelji/1* pronalazi jednog roditelja Y. Predikat *write/1* ispisuje pronađenog roditelja, a predikat *nl/0* stvara novi red za ispis. Predikat *fail/0* sprječava dvostruki ispis; svi pronađeni roditelji odmah će se ispisati jedan za drugim. Zbog tog predikata pravilo *roditelji/1* vratiti će vrijednost *false* nakon izvođenja. Da bi nakon ispisa program vraćao *true*, a ne *false*, uvedeno je pravilo *roditelji(_)*. Taj dodatak nije toliko bitan u ovom primjeru, ali u situaciji kada bi izvođenje ovog predikata utjecalo na daljnji tijek programa njega je potrebno dodati.



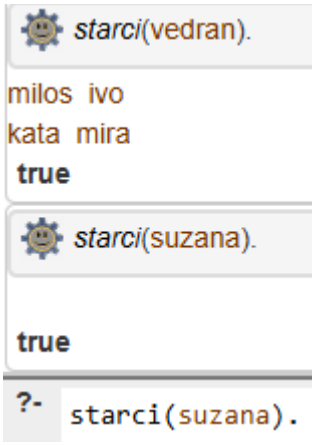
Slika 6: Pravilo roditelji (autorski rad)

Dalje slijedi predikat *braca/1* koji za određenog člana X pronalazi svakog njegovog brata. Pošto u zapisanoj obitelji ne postoji nijedna sestra, nije potrebno provjeravati spol. Prvo se sa predikatom *otac/2* pronalazi otac člana X. Zatim se za pronađenog oca Y traže njegovi sinovi. Naredba \=*(X,Z)* omogućava da se član X ne ispisuje u rezultatu upita. Pronađeni sin se zapiše, te se stavlja novi red za bolji prikaz rezultata.



Slika 7: Pravilo braca (autorski rad)

Zadnje je pravilo *starci/1* koje prikazuje bake i djedove od člana X. Pošto je potrebno pretraživanje dva različita predikata, *djed/2* i *baka/2*, pravilo je zapisano u dva reda. Prvi red pronalazi sve djedove i koristi predikat *tab/1* kako bi ih ispisao sa razmakom. Drugi se izvodi na isti način, no ovaj prvo stavlja ispis u novi red, te pronalazi sve bake.



Slika 8: Pravilo starci (autorski rad)

Iz odrađenog primjera može se vidjeti da u SWISH-u, odnosno SWI-Prologu postoje neki ugrađeni predikati. To su predikati kojima se radnja ne treba postavljati unutar programa, već su oni definirani unutar samog jezika. U ovom primjeru to su *write/1*, *nl/0*, *tab/1* i *fail/0*. Ostali bitniji ugrađeni predikati su sljedeći: *asserta/1* stavlja novu prvu klauzulu u navedeni predikat, *assertz/1* stavlja novu zadnju klauzulu u navedeni predikat, *retract/1* briše klauzule iz baze, *not/1* je predikat koji uspije samo ako njemu zadani argument ne uspijeva (negacija) [6].

3. Problem N kraljica

Sada je, nakon uvoda u programski alat koji će se koristiti za rješavanje problema, moguće opisati sami logički problem koji će se rješavati. Problem N kraljica (eng. *N queens problem*) je problem u kojem se na šahovskoj ploči, dimenzija NxN, treba postaviti N broj kraljica, ali na način da se nijedna kraljica ne napada.

Prema samim pravilima šaha, kraljica je vjerojatno i najjača figurica jer ima mogućnost micati se i napadati u bilo kojem smjeru (horizontalno, vertikalno, okomito) i to za proizvoljan broj polja na ploči. Dakle, u ovom problemu potrebno je sve kraljice postaviti na šahovsku ploču tako da nijedna od njih ne dijeli red, stupac i/ili dijagonalu. Zato problem nije moguće riješiti za $N = 2$ ili $N = 3$ jer na pločama takvih dimenzija nije moguće postaviti kraljice bez da se napadaju.

Kada je prvi put bio osmišljen problem se rješavao na običnoj šahovskoj ploči koja se u pravilu sastoji od osam redaka i stupaca. Dakle, u početku je bio problem osam kraljica. Takvog ga je 1848. osmislio Max Bezzel, šahovski kompozitor (eng. *chess composer*) iz Njemačke. Problem je u ovom obliku bio riješen za samo dvije godine. Franz Nauck je 1850. riješio problem osam kraljica i pronašao svih 92 mogućih rješenja problema, no tada nije uspio dokazati da su to stvarno sva moguća rješenja [13].

Nauck je 1850. godine također i proširio problem na N kraljica, kako je i danas problem poznat [14]. Trenutno, ovaj problem se koristi kao vježba u programiranju, specifično u polju rekurzije i traženja unazad (eng. *backtracking*). Rješavanje problema ne mora biti gotovo kada se jedno rješenje pronađe, već se zadatak može proširiti na pronalaženje svih mogućih rješenja za poseban N. Upravo zato se ovaj problem koristi kako bi se na različite načine i u različitim programskim jezicima mogao prikazati rad rekurzije [15].

Razlog zašto se neke vježbe svode na pronalaženje svih rješenja umjesto jednog je zato što je pronaći jedno rješenje za problem trivijalno [16]. Postoje matematičke formule kojima se lako, bez korištenja računala, može doći do rješenja. Glavni parametri u računanju rješenja za problem su sljedeći:

- Je li odabrani N paran ili neparan?
- Može li se N dijeliti sa brojem 6?

Korištenjem ovih parametara i formula iz [16] moguće je svaki N problem riješiti sa sljedećom shemom: uzmimo da par (i, j) predstavlja jedno polje na šahovskoj ploči, gdje i predstavlja redak, a j stupac tog polja. Sada možemo, prema odabranom broju N, iskoristiti jedan od tri moguća slučaja ili formule:

A. Odabrani N je paran i nije oblika $6k + 2$. Kraljice se na ploču postavljaju na sljedeće elemente:

$$(j, 2j)$$

$$\left(\frac{N}{2} + j, 2j - 1\right)$$

$$j = 1, 2, \ldots, \frac{N}{2}$$

B. Odabrani N je paran i nije oblika $6k$. Kraljice se na ploču postavljaju na sljedeće elemente:

$$(j, 1 + \left[2(j-1) + \frac{N}{2} - 1 \; mod \; N\right])$$

$$(N + 1 - j, N - \left[2(j-1) + \frac{N}{2} - 1 \; mod \; N\right])$$

$$j = 1, 2, \ldots, \frac{N}{2}$$

C. Odabrani N je neparan. Kraljice na ploču postaviti tako da se problem riješi za $N - 1$ koristeći potreban slučaj (A ili B), te nakon toga u rješenje dodati kraljicu na element (N, N).

Jedan slučaj dokazat će se uz primjer sa problemom gdje je $N = 5$. Prvo se gleda kakav je N. U ovom slučaju on je neparan, dakle potrebno je koristiti slučaj C. Sada se treba pronaći rješenje za $N - 1$, što je 4. Traži se slučaj kojem on pripada. Za slučaj A potrebno je da N nije sljedećeg oblika:

$$6k + 2 = 4$$

$$6k = 2$$

$$k = \frac{2}{6}$$

Dobiveni k nije prirodan broj, dakle N nije zadanog oblika. Može se koristiti slučaj A. Sada se može početi sa postavljanjem kraljica na polje. Pozicije se mogu dobiti na sljedeći način:

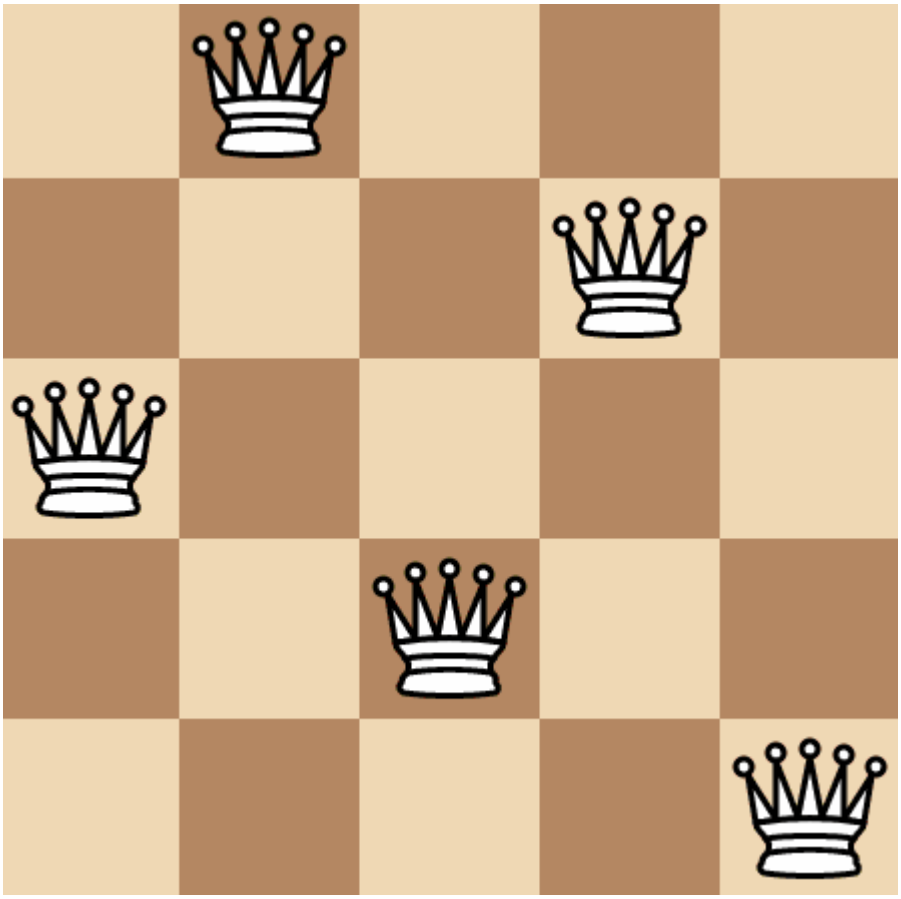
$$j = 1;\ (1, 2 * 1)$$

$$j = 2;\ (2, 2 * 2)$$

$$j = 1;\ \left(\frac{4}{2} + 1, 2 * 1 - 1\right)$$

$$j = 2;\ \left(\frac{4}{2} + 2, 2 * 2 - 1\right)$$

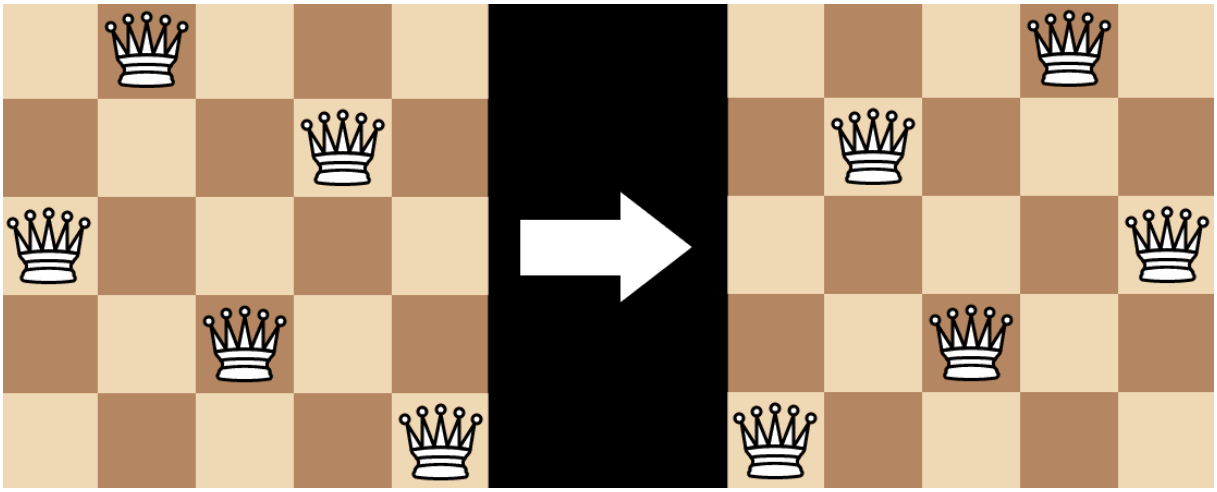
Dodavanjem zadnje kraljice na polje (N, N) možemo dobiti sljedeći poredak: $(1,2), (2,4), (3,1), (4,3), (5,5)$. Rješenje na ploči tada izgleda ovako:



Slika 9: Rješenje primjera (autorski rad)

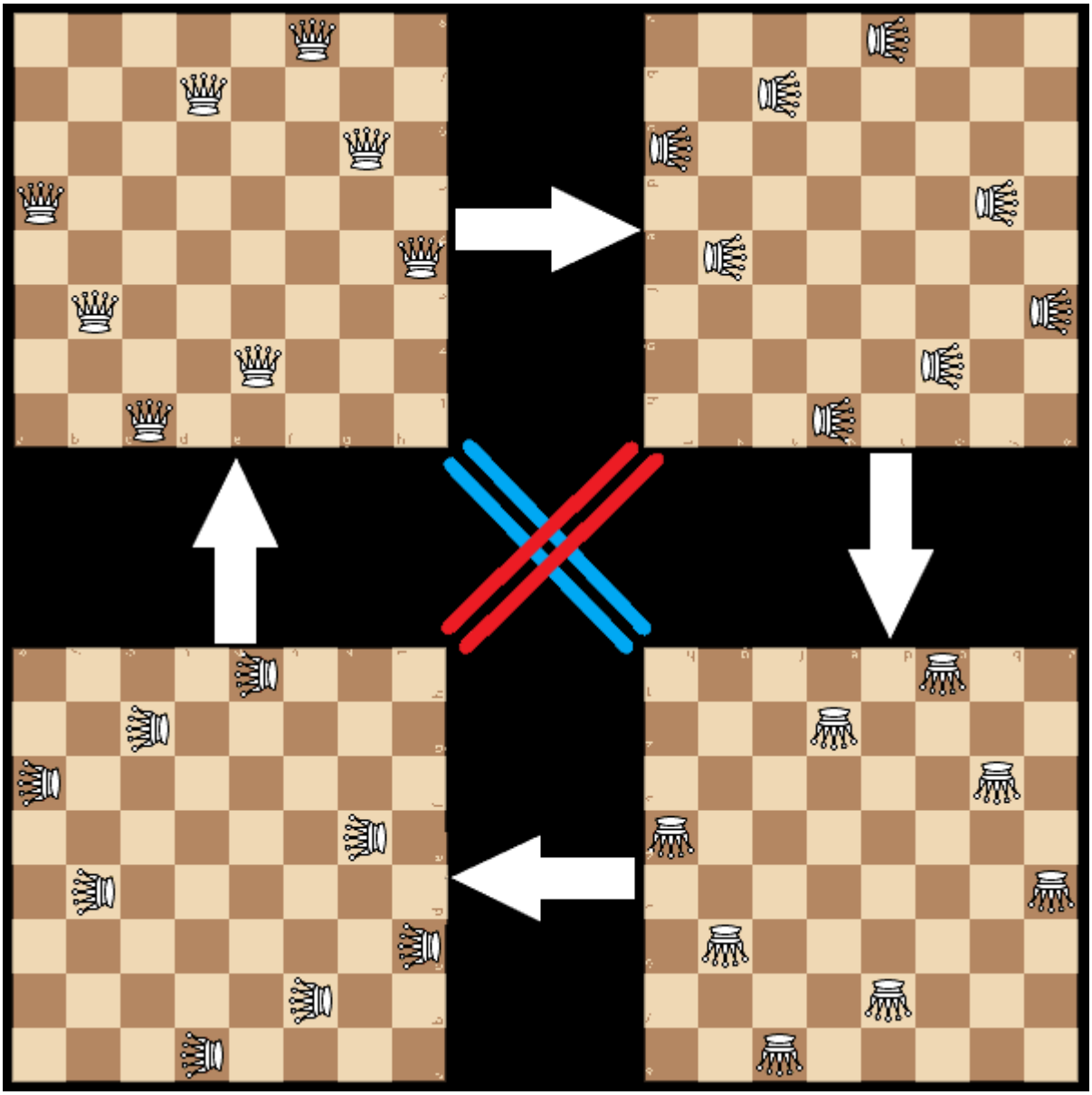
Za kraj ovog djela još valja napomenuti kako se broje rješenja za problem N kraljica. Uzmimo za primjer originalni problem osam kraljica. Za njega se zna da sveukupno ima 92 mogućih rješenja, kako je i Nauck rekao 1850. No, potrebno je naglasiti da nisu sva rješenja ista. Naime, za svako rješenje postoje moguće varijacija.

Varijacija na rješenje dobije se tako da se ono rotira za 90 stupnjeva. Dakle, varijacije postoje za rotacije od 90, 180 i 270 stupnjeva. Svaka od tih varijacija također se može i reflektirati kako bi se dobile sve varijacije. Samo rješenje bez varijacije naziva se fundamentalno rješenje [17].



Slika 10: Refleksija rješenja (autorski rad)

No, zbog onih rješenja koja su ista kada se rotiraju, ne može se zaključiti da svako rješenje postoji u osam varijacija. Takva simetrična rješenja imaju manje varijacija. Ako rješenje zadržava istu poziciju kraljica za svaku rotaciju, onda ono ima dvije varijacije zbog svoje refleksije. Ako je rješenju različita pozicija kraljica samo u prvoj rotaciji, onda ono ima četiri varijacije zbog refleksije sebe i prve rotacije [17]. Zato problem osam kraljica za svojih 12 originalnih ili fundamentalnih rješenja ima sveukupno 92 moguća rješenja, a ne $12 * 8 = 96$ ukupnih rješenja jer jedno rješenje ima varijaciju samo u svojoj prvoj rotaciji.



Slika 11: Različito rješenje samo u prvoj rotaciji (autorski rad)

4. Rješavanje problema

Nakon uvoda u programski jezik SWI-Prolog i problem N kraljica moguće je započeti sa izradom programa koji će taj problem rješavati. Problem N kraljica koristi se kao klasičan primjer za mogućnosti i programiranje u Prologu. Na SWISH stranici dostupno je pregledati već izrađene kodove za rješavanje tog problema kako bi se novim korisnicima dočaralo što je sve moguće izraditi sa SWI-Prologom [18].

4.1. SWI-Prolog rješenja

Izrađeni kod izrađen je po uzoru na rješenje Markusa Triske koji održava web stranicu „*The Power of Prolog*“ [19]. Ona je trenutno jedan od najvećih repozitorija informacija i lekcija vezanih za SWI-Prolog.

```
:- use_rendering(chess).

n_queens(N, Q):-
    length(Q, N),
    queens(Q, N),
    safe(Q).

queens([], _).
queens([Q|Qs], N):-
    queens(Qs, N),
    between(1, N, Q).

safe([]).
safe([Q|Qs]):-
    safe(Qs),
    place(Q, Qs, 1).

place(_, [], _).
place(Q0, [Q1|Qs], D0):-
    Q0 =\= Q1,
    Q0 - Q1 =\= D0,
    Q0 - Q1 =\= -D0,
    D1 is D0 + 1,
    place(Q0, Qs, D1).
```

Slika 12: Programsko rješenje problema N kraljica prema Simić Ivanu (autorski rad)

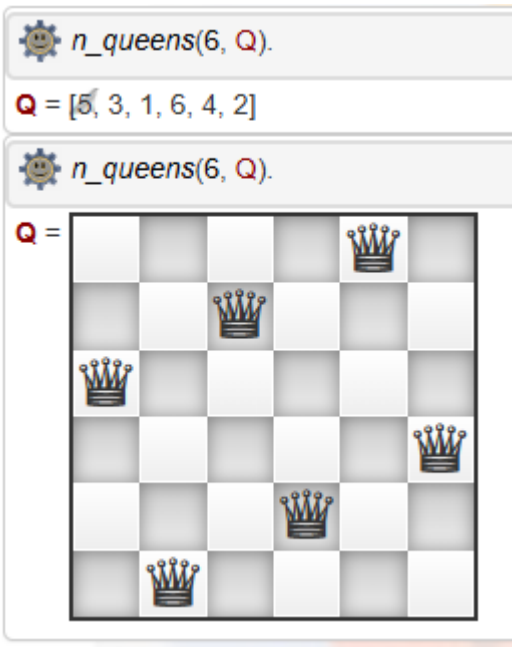
Za bolje shvaćanje tijeka koda, svaki će se predikat posebno objasniti:

- **n_queens/2**: predikat s kojim kod započinje i s kojim se poziva njegovo izvođenje. Dva argumenta koje prima su broj kraljica (N) i pozicija kraljice (Q). Unatoč tome što se na šahovskoj ploči pozicija figurice gleda kao kombinacija retka i stupca, u problemu N kraljica niti jedna figurica ne smije dijeliti niti redak niti stupac. Dakle, kao poziciju možemo uzimati samo stupac figurice, pošto dvije kraljice nikad neće imati isti redak. Naredba **length/2** je ugrađeni predikat kojim određujemo količinu elementa. Broj elemenata Q biti će isti kao i broj upisan u varijabli N. Predikati **queens/2** i **safe/1** se pozivaju kako bi se kod mogao nastaviti izvoditi.
- **queens/2**: služi za stvaranje svih mogućih lista N kraljica. Pošto ovaj predikat kao argument prima listu, potrebno je prije njega napisati praznu verziju predikata kojem su argumenti prazna lista i prazna varijabla. Ovako kod zna da ne treba izvoditi predikat kada se dođe do kraja liste. Ovaj predikat isto prima argumente Q i N, ali pošto mora prolaziti kroz sve kraljice u listi dodaje se argument Qs. Tako se dobije argument [Q|Qs] u kojem Q predstavlja trenutnu kraljicu u listi, a Qs sljedeću kraljicu u listi. Argument Qs koristit će se u rekurzivnom pozivu predikata **queens/2** kojim se predikat ponovo izvodit, ali ovaj put sa sljedećom kraljicom Qs. Rekurzivni pozivi predikata su uvijek podcrtani u kodu. Ugrađeni predikat **between/3** daje argumentu Q vrijednost koja se nalazi između jedan i N. Ovako svaka kraljica dobije svoj broj stupca.
- **safe/1**: služi za rekurzivno pozivanje svake kraljice i prebacuje program na sljedeće pravilo. Prvo rekurzivno poziva samog sebe sa **safe/1**, ali ovaj put sa sljedećom kraljicom Qs. Nakon toga poziva sljedeći predikat **place/3**. Ovako se može provjeriti svaka kraljica u listi.
- **place/3**: ovaj predikat traži poziciju za kraljicu. Njegovi argumenti su trenutna kraljica Q, lista u kojoj su sačuvana sljedeća kraljica Q1 i ona nakon nje Qs i broj D0 koji služi za provjeru dijagonale. Prvo predikat provjerava da trenutna i sljedeća kraljica nisu u istom stupcu. Nakon toga provjerava da nisu u istoj dijagonali. D0 se povećava za provjeru u sljedećem ponavljanju. Na kraju se predikat rekurzivno poziva, ali ovaj put sa sljedećom kraljicom i povećanim D1.

Ovaj program postavlja prvu kraljicu na prvo slobodno polje. Zatim drugu kraljicu postavlja na prvo polje u drugom retku gdje neće napadati prvu kraljicu. Ostale kraljice se postavljaju na isti način. U slučaju da se jedna od kraljica ne može u svom retku postaviti bez da ne napada ostale kraljice, program se mora vraćati unazad i premještati već postavljene kraljice kako bi pronašao polje za novu. Program će se ovako izvoditi sve dok svih N kraljica nije postavljeno.

Program se poziva sa upitom *n_queens(N, Q)*. Rezultat upita je lista brojeva koji predstavljaju broj stupca u kojem se nalazi kraljica u svojem retku. Takva lista može se ispisati u obliku šahovske ploče uz pomoć renderiranja. Renderirati ploču u SWI-Prologu nije zahtjevan posao. Sve što je potrebno je uvesti modul na početku koda. Ovdje je to naredba *use_rendering(chess)*. Ta naredba ne unosi nikakve nove predikate u kod. Samo je potrebno prilagoditi ispis. Jedino kako ta naredba može prikazati ploču jest ako se ispis nalazi u obliku liste koja sadrži N brojeva.

Na sljedećoj slici moguće je vidjeti originalan upit bez renderiranja i verziju sa renderiranjem u kojoj se rješenje upita ispiše u obliku šahovske ploče:



Slika 13: Ispis programskog rješenja problema N kraljica prema Simić Ivanu (autorski rad)

Najveći problem ovog koda je vrijeme potrebno da se on izvede. Za svaki N manji od osam izvođenje koda ne predstavlja problem. Pronalazak ostalih rješenja isto tako ne troši puno vremena. Ali već za $N = 8$ programu treba više od dvije sekunde da pronađe jedno rješenje. Razlog je već prije naveden. Kod se mora stalno vraćati unazad i premještati već postavljene kraljice kako bi našao mjesta za nove. Nije u stanju unaprijed odrediti gdje je najbolje kraljicu postaviti, bez da se ona kasnije treba previše premještati.

Također, tijekom generiranja listi kraljica moguće je stvoriti listu gdje se više brojeva ponavlja. Za takve liste se odmah zna da nisu rješenje, pošto dva ista broja znači da dvije kraljice dijele stupac. Program nije u stanju takve liste ignorirati ili uopće ne stvoriti. Kroz njih se isto iterira zbog čega se potroši puno vremena.

Ovo bi se moglo poboljšati tako da se unaprijed spriječi ponavljanje polja za koje se već zna da drže jednu kraljicu u sebi. Također, može se dodati neki način predviđanja u program, odnosno da se u trenutku postavljanja jedne kraljice detektiraju sva polja na ploči koja više nisu dostupna i da se ona u daljnjem izvođenja koda ignoriraju.

U sljedećim poglavljima analizirat će se drugi načini implementacije problema N kraljica u SWI-Prologu, te će se usporediti sa već izrađenim programskim rješenjem.

4.1.1. Prva usporedba programskih rješenja (prema Ron Danielson)

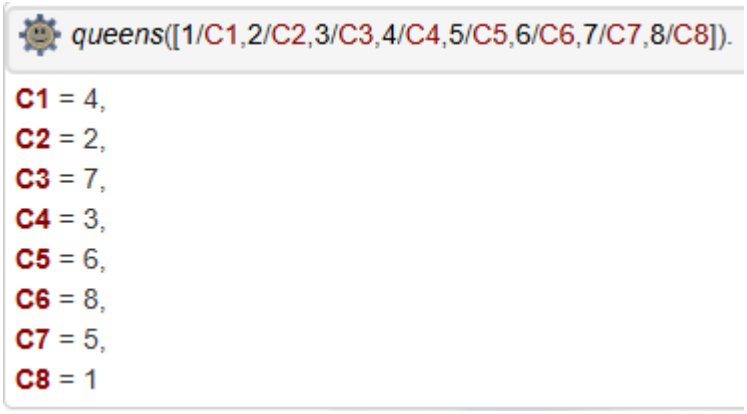
Ovo je rješenje Ron Danielsona, profesora sa Santa Clara sveučilišta u Americi. Danielson je u svome rješenju naveo Ivana Bratka, autora knjige „*Prolog Programming for Artificial Intelligence*“, kao inspiraciju [20].

```
1  queens([]).
2  queens([Row/Col | Rest]):-
3      queens(Rest),
4      member(Col, [1,2,3,4,5,6,7,8]),
5      safe(Row/Col, Rest).
6
7  safe(_, []).
8  safe(Row/Col, [Row1/Col1 | Rest]):-
9      Col =\= Col1,
10     Col1 - Col =\= Row1 - Row,
11     Col1 - Col =\= Row - Row1,
12     safe(Row/Col, Rest).
13
14 member(X, [X | _]).
15 member(X, [_ | Tail]):-
16     member(X,Tail).
```

Slika 14: Programsko rješenje prema Ron Danielson (autorski rad)

U ovom rješenju koriste se i redak i stupac u pronalasku polja za kraljicu. Zbog toga se već početni predikat **queens/1** implementira sa listom. Uz to što se taj predikat rekurzivno poziva za pronalazak sljedećeg elementa, također treba koristiti i pomoćni predikat **member/2** kako bi se svaki redak i stupac pravilno izmjenjivao.

Predikat **safe/2** funkcionira slično kao i u napravljenom rješenju, samo što ovdje nema potrebe za varijablom D0, pošto se ovdje za svako polje navodi i redak i stupac, pa je moguće provjeravati dijagonalu bez dodatnih varijabli.



Slika 15: Ispis programskog rješenja prema Ron Danielson (autorski rad)

Ovo rješenje brže generira upite za razliku od napravljenog rješenja zbog načina na koji se kod kreće kroz listu redaka i stupaca. Međutim, problem u ovom rješenju je izvođenje upita. Kao što je vidljivo na gornjoj slici, zadani upit je velik jer je potrebno upisati svaki redak i stupac posebno. Također, zbog načina na koji se rješenje ispisuje nije moguće renderirati šahovsku ploču za bolji prikaz rješenja.

Još jedan problem je potrebno mijenjanje koda ako se želi pronaći rješenje za drugi N. Korisnik prije izvođenja upita treba u prvom pozivanju predikata *member/2*, unutar početnog predikata *queens/1*, promijeniti listu brojeva na N koji želi ispitati.

4.1.2. Druga usporedba programskih rješenja (prema Richard A. O'Keefe)

Sljedeće je rješenje od Richard A. O'Keefea, profesora sa Royal Melbourne Institute of Technology sveučilišta u Australiji. Rješenje je preuzeto sa kolekcije primjera na SWISH web stranici, a samo rješenje je izvedeno iz O'Keefeove knjige „*The Craft of Prolog*“ [21].

```
:- use_rendering(chess).

queens(N, Queens) :-
    length(Queens, N),
    board(Queens, Board, 0, N, _, _),
    queens(Board, 0, Queens).

board([], [], N, N, _, _).
board([_|Queens], [Col-Vars|Board], Col0, N, [_|VR], VC) :-
    Col is Col0+1,
    functor(Vars, f, N),
    constraints(N, Vars, VR, VC),
    board(Queens, Board, Col, N, VR, [_|VC]).

constraints(0, _, _, _) :- !.
constraints(N, Row, [R|Rs], [C|Cs]) :-
    arg(N, Row, R-C),
    M is N-1,
    constraints(M, Row, Rs, Cs).

queens([], _, []).
queens([C|Cs], Row0, [Col|Solution]) :-
    Row is Row0+1,
    select(Col-Vars, [C|Cs], Board),
    arg(Row, Vars, Row-Row),
    queens(Board, Row, Solution).
```

Slika 16: Programsko rješenje prema Richard O'Keefe (autorski rad)

Ovo rješenje je dosta kompleksnije od ostalih navedenih rješenja. Početni predikat **queens/2** funkcionira kao i u napravljenom rješenju; određuje se količina Q elementa, generira se ploča, te se postavljaju kraljice. Glavna razlika je u generiranju ploče.

Predikat **board/6** služi za stvaranje ploče na način da se svakom stupcu na ploči odmah zadaju ograničenja. Za svaki stupac se poziva poseban predikat **constraints/4** koji njemu određuje ograničenja. U **queens/3** predikatu se prolazi kroz svaki redak na ploči. Program je odmah u stanju odrediti koji stupac je najbolji za iskoristiti zbog stvorenih ograničenja.

Ovo je izrazito kvalitetno složena implementacija temeljena na programiranju sa ograničenjima. Odmah je očito da je ova implementacija brža od napravljene, pošto program unaprijed određuje shemu za postavljanje kraljica, te tako može efikasnije postaviti kraljice sa što manjom potrebom za vraćanjem unazad. Jedini nedostatak ovakve implementacije je njena kompleksnost. Program koristi puno različitih varijabli i ugrađenih predikata, pa je teško pratiti izvođenje programa.

4.1.3.Treća usporedba programskih rješenja (prema Markus Triska)

Zadnje je rješenje već bilo spomenuto na početku poglavlja. Radi se o rješenju Markusa Triske koje je služilo kao inspiracija za napravljeno rješenje.

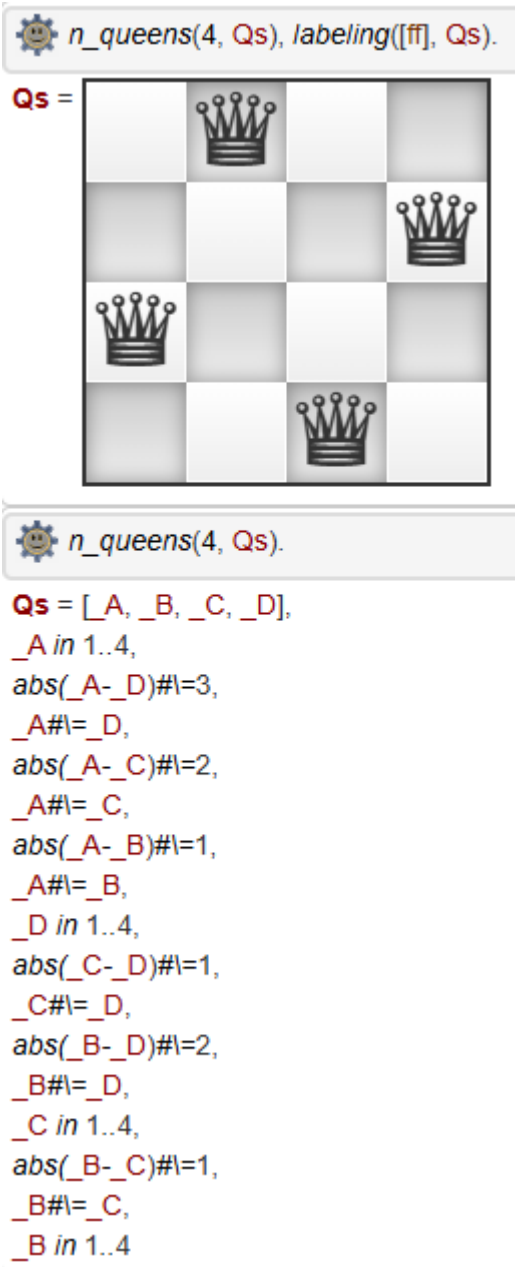
```
1  :- use_rendering(chess).
2  :- use_module(library(clpfd)).
3
4  n_queens(N, Qs) :-
5      length(Qs, N),
6      Qs ins 1..N,
7      safe_queens(Qs).
8
9  safe_queens([]).
10 safe_queens([Q|Qs]) :-
11     safe_queens(Qs, Q, 1),
12     safe_queens(Qs).
13
14 safe_queens([], _, _).
15 safe_queens([Q|Qs], Q0, D0) :-
16     Q0 #\= Q,
17     abs(Q0 - Q) #\= D0,
18     D1 #= D0 + 1,
19     safe_queens(Qs, Q0, D1).
```

Slika 17: Programsko rješenje prema Markus Triska (autorski rad)

Početni predikat **n_queens/2** služi istu funkciju kao i u napravljenom rješenju, samo što ovdje nije potrebno pozivati dodatni predikat za generiranje Qs elemenata. Ovo rješenje koristi *clpfd* modul koji programu omogućava korištenje više aritmetičkih predikata i funkcija.

Jedan takav je ugrađeni predikat *ins/2* kojim se zadaju vrijednosti elemenata Qs. Njegovo izvođenje slično je kao i u napravljenom rješenju, gdje se koristi ugrađeni predikat *between/3*, samo što ovdje nije potrebno stvarati novi predikat koji će ga rekurzivno pozivati jer *ins/2* može odmah zadati varijablu za svaki Qs, dok se izvođenje *between/3* predikata mora ponavljati sve dok svaki Q ne dobije svoju varijablu. Isto tako *ins/2* listi daje samo unikatne vrijednosti, pa se nikad neće stvoriti lista gdje se neki brojevi ponavljaju.

Predikati **safe_queens/1** i **safe_queens/3** imaju istu funkciju kao i *safe/1* i *place/3* u napravljenom rješenju. Jedina je razlika u korištenim operatorima i pojednostavljenoj operaciji provjeravanja dijagonale uz korištenje apsolutne vrijednosti. Operatori =\= i #\= imaju istu funkciju, jedino što operator #\= postoji samo unutar modula *clpfd*.



Slika 18: Ispisi programskog rješenja prema Markus Triska (autorski rad)

Ovo je najoptimiziranije od svih prikazanih rješenja ovdje. Najbrže pronalazi rezultate, te je u stanju izvesti i programe gdje je $N > 100$. Može se smatrati da ovaj program nije povoljan za učenje novih korisnika zbog korištenja dodatnih modula, ali se isto tako može reći da se time novim korisnicima prikazuje kako korištenje modula može poboljšati izvođenje programa.

Jedini potencijalan problem, ako se on uopće tako može smatrati, je to što korištenjem *ins/2* predikata ispis nije prikazan u pravilnom obliku. Zato je potrebno dodati labeliranje u upit. Tako varijable dobivaju pravilan oblik, te je program sposoban prikazati šahovsku ploču. Labeliranje je isto uključeno u *clpfd* modulu. Samo je potrebno na upit dodati *labeling([ff], Qs)*.

4.2. Python rješenja

Pošto se već u prijašnjim poglavljima govorilo o razlikama između Prologa, odnosno logičkih programskih jezika i ostalih, proceduralnih programskih jezika, odlučeno je da se neke od tih razlika prikaže usporedbom dva suštinski različita programa koji rješavaju isti problem. Ovdje će se na problemu N kraljica uspoređivati SWI-Prolog (deklarativni programski jezik) i Python (proceduralni programski jezik).

Python je programski jezik s fokusom na objektno orijentirano programiranje. Razvio ga je nizozemski programer Guido van Rossum u 1991. godini. Zbog svoje jednostavnosti i efektivnosti danas je jedan od najkorištenijih programskih jezika. Najčešće se koristi za statistiku i razvijanje web aplikacija. Korisnicima omogućava korištenje već uvedenih modula i preuzimanje dodatnih biblioteka, koje su razvili ostali korisnici, kako bi obogatili svoj program [22, 23].

U Pythonu je moguće programirati sa više različitih tipova podataka. Uz standardne numeričke varijable moguće je koristiti i *stringove*, liste, rječnike i *bool* varijable (*true*/*false*). Implementiranje slučajeva sa *if-else-while* strukturama omogućava jednostavno upravljanje tokom izvršavanja programa. Cijeli program može se podijeliti na više blokova koda sa uvlačenjem iskaza [24].

Glavna prednost Pythona prema SWI-Prologu je upravljanje varijablama. Lakše ih je upisivati u Pythonu i moguće je koristiti više različitih vrsta varijabli. Također, u Pythonu ima puno više ugrađenih funkcija za upravljanje varijablama, te je lakše uređivati oblik inputa i outputa programa. Prednost SWI-Prologa prema Pythonu je to što se u njemu funkcionalnosti programa ne trebaju detaljno raspisivati. Python nije deklarativan; njegove se funkcije moraju programirati korak po korak kako bi ih program mogao pravilno izvršiti. SWI-Prolog programi sami pronalaze rješenje iz baze znanja prema dobivenom upitu [25].

4.2.1.Usporedba programskih rješenja (prema Divyanshu Mehta)

Kako bi se oba programska jezika pravilno usporedila, pronađeno je jednostavno Python rješenje za problem N kraljica. Radi se o rješenju programera Divyanshu Mehta sa *GeeksforGeeks* platforme [26]. Struktura rješenja je smanjena, ali je cijela funkcionalnost ista kao i u originalnom prikazu.

```
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print (board[i][j],end=' ')
        print()

def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col):
    if col >= N:
        return True

    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQUtil(board, col + 1) == True:
                return True
            board[i][col] = 0

    return False

def solveNQ():
    board = [ [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0] ]

    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False

    printSolution(board)
    return True

solveNQ()
```

Slika 19: Programsko rješenje prema Divyanshu Mehta (autorski rad)

Slijedi opis svih definiranih funkcija u programu:

- **printSolution**: služi za ispis konačnog rješenja. Iterira se kroz svaki redak i stupac i ispisuje se varijabla zapisana u svakom pojedinom polju. Ako se na polju nalazi kraljica, zapisana varijabla je 1. Ako je polje prazno, zapisana varijabla je 0.
- **isSafe**: funkcija koja provjerava da postavljena kraljica ne napada nijednu drugu na ploči. Iterira se kroz svaki stupac u trenutnom retku, te se provjeravaju obje dijagonale.
- **solveNQUtil**: petlja kojom se postavljaju kraljice. Ako su sve kraljice postavljene, izlazi se is petlje. Ako nisu, provjerava se da li će na trenutnom polju kraljica biti napadnuta. Ako ne, ona se postavlja na trenutno polje i gleda se sljedeći stupac. U slučaju da se sa trenutnom konfiguracijom nova kraljica ne može postaviti, briše se prijašnja i provjerava za novi redak. Funkcija se rekurzivno poziva sve dok se ne postave sve kraljice, ili dok se nova kraljica ne može nigdje postaviti.
- **solveNQ**: glavna funkcija koja se poziva na početku izvođenja programa. Generira se ploča u obliku liste. Ako funkcija nije u stanju pronaći rješenje, program mora to naznačiti. Ako se rješenje pronađe, ono se i ispiše.

Prikazano rješenje je jednostavna implementacija problema N kraljica u Pythonu. Rješenje ne koristi nikakve dodatne module ili biblioteke. Koriste se samo elementi koji su uvijek dostupni u Python programu. U usporedbi sa izrađenim SWI-Prolog programom, rješenje se puno brže pronalazi, ali ova jednostavna implementacija je daleko kompliciranija. Svaki dio programa (generiranje ploče, ispis rješenja, rekurzivno postavljanje kraljica) morao se detaljno definirati, za razliku od SWI-Prolog programa gdje se oni ili nisu posebno implementirali, ili su bili implementirani u par redaka.

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
True
```

Slika 20: Ispis programskog rješenja prema Divyanshu Mehta (autorski rad)

Najsličnije su si funkcije koje provjeravaju da li se kraljice napadaju. U oba programa se pojedinačno provjeravaju redak, stupac i dijagonale, ali u ovom se programu za svaku tu provjeru morala izraditi posebna petlja. Uz to što je program kompliciraniji, on nema sve mogućnosti koje SWI-Prolog program ima. Ako se želi drugi N provjeriti, potrebno je promijeniti oblik generirane ploče u samom programu. Program također nije u stanju pronaći svako

moguće rješenje, već pronalazi prvo moguće i njega ispisuje. Dakle, program je brži ali kompliciraniji i ima manje funkcionalnosti.

4.2.2.Proširenje funkcionalnosti

Kako bi se kodovi uspoređivali na istim razinama proširen je preuzeti Python kod. U kod je uvedeno renderiranja ploče uz pomoć Pygame biblioteke, te je dodana mogućnost pronalaska više rješenja za određeni N. Uz to je i olakšan način odabira varijable N. Pygame implementacija uvedena je po uzoru na kod programera Keith Gallia [27].

```
import pygame

N = int(input("N = "))
solutions = []
index = 0

def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col):
    if col >= N:
        solutions.append([row[:] for row in board])

    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            solveNQUtil(board, col + 1)
            board[i][col] = 0

def solveNQ():
    board = [[0] * N for _ in range(N)]
    solveNQUtil(board, 0)
```

Slika 21: Izvorni dio programskog rješenja prema Simić Ivanu (autorski rad)

Na početku programa uvodi se Pygame biblioteka. Ona nije dio standardne Python biblioteke, pa ju je prije izvođenja programa potrebno preuzeti sa naredbom *pip install pygame*. Varijabla N nije zadana, već ju se može upisati kada se program izvodi. Dodane su pomoćne varijable *solutions* i *index* kojima će se spremati sva pronađena rješenja. Funkcija **printSolution** je uklonjena jer se ovdje rješenje ispisuje u Pygame prozoru. U funkciji **isSafe** nije došlo do promjena.

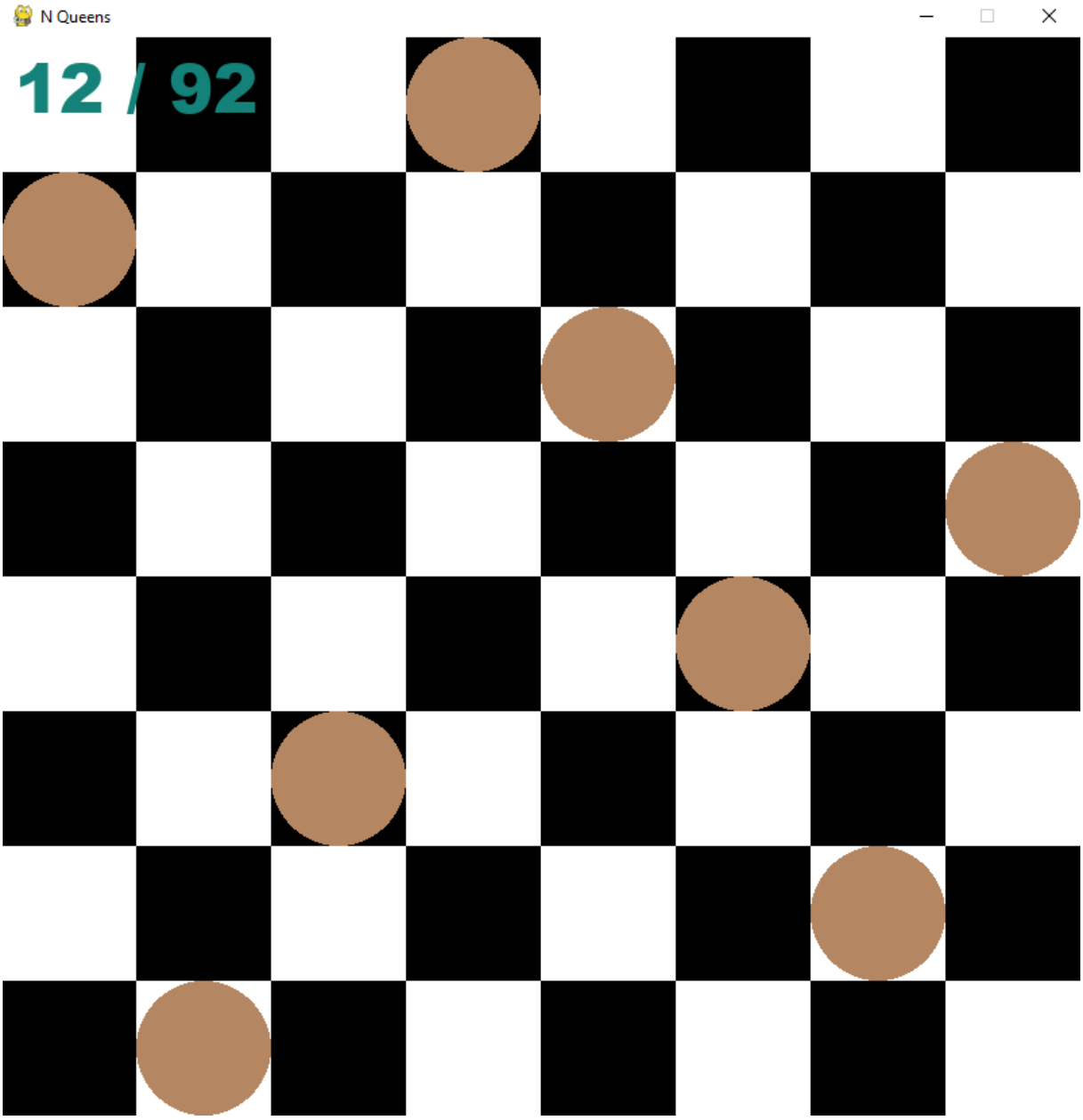
U funkciji **solveNQUtil** promijenjen je uvjet za završetak. Prije je funkcija završavala kada se zadnja kraljica postavila. Ovdje se, nakon zadnje kraljice, sprema konfiguracija polja [pom]. Nakon što se rješenje spremi, petlja se nastavlja. Tako će program nastavljati izvoditi petlju, sve dok se ne pokuša pronaći rješenje za svaku moguću konfiguraciju. Iz funkcije su uklonjene sve *bool* varijable, jer se pronalazak rješenja provjerava izvan nje.

Kod početne funkcije **solveNQ** promijenjen je način generiranja ploče. Prije se ona ručno generirala. Sada je napravljeno da se N puta jedan redak ploče popuni varijablom 0, te da se stvori još takvih redaka N puta [28]. Tako se dobije ploča NxN dimenzija. Ostali dio koda fokusira se na korištenje Pygame biblioteke.

```
pygame.init()
SIZE = 800
SQUARE = SIZE // N
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
QUEEN = (180, 135, 98)
TEXT = (20, 130, 120)
screen = pygame.display.set_mode((SIZE, SIZE))
pygame.display.set_caption("N Queens")
font = pygame.font.SysFont("arialblack", 50)

def drawBoard(board):
    for row in range(N):
        for col in range(N):
            color = WHITE if (row + col) % 2 == 0 else BLACK
            pygame.draw.rect(screen, color, (col * SQUARE, row * SQUARE, SQUARE, SQUARE))
            if board[row][col] == 1:
                pygame.draw.circle(screen, QUEEN, (col * SQUARE + SQUARE // 2, row * SQUARE + SQUARE // 2), SQUARE // 2)
    label = font.render(f"{index + 1} / {len(solutions)}", True, TEXT)
    screen.blit(label, (10, 0))
    pygame.display.update()

solveNQ()

if solutions:
    running = True
    drawBoard(solutions[index])
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN:
                index += 1
                if index >= len(solutions):
                    running = False
                else:
                    drawBoard(solutions[index])

else:
    print("Solution does not exist")

pygame.quit()
```

Slika 22: Pygame dio programskog rješenja prema Simić Ivanu (autorski rad)

Za početak potrebno je inicijalizirati prikaz Pygame prozora. Zadaje se njegova veličina, veličina svakog polja, boje koje će se koristiti u prikazu polja, naziv prozora, te oblik fonta. Ploča se crta sa **drawBoard** funkcijom. Iterira se kroz svaki redak i stupac na polju, te se naizmjenično crtaju kvadratići bijele i crne boje. Ako se dođe do polja gdje je vrijednost 1, odnosno u njemu se nalazi kraljica, nacrtat će se smeđi krug. Na vrhu ploče označeno je koje se rješenje trenutno prikazuje u prozoru.



Slika 23: Ispis Pygame prozora (autorski rad)

Pygame prozor se otvara ako je pronađeno rješenje. U protivnom se ispisuje poruka greške. Ako se prozor otvori, crta se ploča i počinje glavna petlja. Unutar petlje su zadani događaji kojima se upravlja Pygame prozorom. Prozor se ugasi ako se pritisne X gumb. Ako se na ploču pritisne mišem, prozor će pokazati sljedeće rješenje. Ako se dođe do zadnjeg rješenja, pritiskom na ploču se prozor zatvara.

Ovime je stvoren Python program koji sadrži iste funkcionalnosti koje su mu, u usporedbi sa izrađenim SWISH rješenjem, nedostajale. Program je u stanju primiti željenu N varijablu pri pokretanju, bez da ga je prije toga potrebno uređivati. Također je u stanju prikazati sva pronađena rješenja za pojedini N.

Nažalost, kodiranje ovakvog rješenja u Pythonu i dalje je zahtjevnije i kompliciranije nego u SWI-Prologu. Kako bi se renderirala šahovska ploča, potrebno je bilo preuzeti i implementirati posebnu biblioteku. Također je bilo potrebno urediti dijelove koda kako bi se rješenje pravilno prikazalo. Unutar SWI-Prologa nije bilo potrebno urediti izvorni kod, niti preuzimati posebne biblioteke. Samo je bilo potrebno na početku uvesti modul za renderiranje, te osigurati da se rješenje prikaže u obliku liste.

4.3. Sinergija SWI-Prologa i Pythona

Sada je, nakon što se prikazalo kako se problem može riješiti u SWI-Prologu i u Pythonu, vrijeme da se pokaže kako se ta dva jezika mogu spojiti. Točnije, kako se može unutar Pythona koristiti SWI-Prolog klauzulama. Radi se o Pyswip biblioteci kojom se unutar Pythona može stvoriti SWI-Prolog baza znanja, te da se prema toj bazi šalju upiti preko Pythona [29]. Baza znanja može se implementirati direktno u Python kodu ili preko vanjske datoteke koja sadržava SWI-Prolog bazu.

Da bi se počelo programirati sa Pyswip bibliotekom, prvo ju je potrebno preuzeti sa *pip install pyswip* naredbom jer ona, kao i Pygame biblioteka, nije sadržana unutar standardne Python biblioteke. Nakon toga potrebno je preuzeti SWI-Prolog na računalo kako bi se program mogao pravilno kompajlirati. Instalacija SWI-Prologa i Pythona moraju biti za isti sistem, inače program neće moći funkcionirati.

4.3.1.Implementacija

Za kreiranje programa korišteno je izrađeno SWI-Prolog rješenje iz 4.1. poglavlja. U tom istom poglavlju naveden je i glavni nedostatak tog rješenja, a to je količina vremena potrebnog za stvaranje ploče. Naime, taj program na početku generira liste svih mogućih kombinacija N brojeva od 1 do N. To generiranje je sporo i ima veliku grešku; program nije u stanju zabraniti generiranje istih brojeva unutar jedne liste. Ovaj nedostatak će se zamijeniti Pythonom, koji istu zadaću može odraditi puno brže.

Analizom pronađenog Python rješenja iz poglavlja 4.2.1. zaključeno je da je proces rekurzije unutar njega podosta kompliciran. Za razliku od toga, SWI-Prolog rješenje rekurziju obavlja uz pomoć dva mala predikata. Zato će se rekurzija u rješenju obavljati Prologom. Ovom kombinacijom stvoreno je dolje prikazano rješenje.

```
from itertools import permutations
from pyswip import Prolog
prolog = Prolog()

prolog.retractall("safe(_)")
prolog.retractall("place(_, _, _)")

prolog.assertz("safe([])")
prolog.assertz("safe([Q|Qs]) :- safe(Qs), place(Q, Qs, 1)")
prolog.assertz("place(_, [], _)")
prolog.assertz("place(Q0, [Q1|Qs], D0) :- Q0 =\\= Q1, Q0 - Q1 =\\= D0, Q0 - Q1 =\\= -D0, D1 is D0 + 1, place(Q0, Qs, D1)")

N = int(input("N ="))
solutions = []
index = 0

print("Searching...")
print()

perms = [list(perm) for perm in list(permutations(range(1, N + 1)))]
for perm in perms:
    if list(prolog.query(f"safe({perm})")):
        solutions.append(perm)

print("Found solutions:", len(solutions))

for solution in solutions:
    print(f"Q{index + 1} = {solution}")
    print()
    index += 1
    if index < len(solutions):
        if input(f"[Enter] for Q{index + 1} | (n) for Stop:") == 'n':
            break
```

Slika 24: Kombinirano programsko rješenje prema Simić Ivanu (autorski rad)

Za početak se inicijaliziraju sve potrebne biblioteke, te se učitavaju Prolog naredbe. Svaki Prolog predikat može se uvesti u program uz pomoć *assertz* naredbe. No, ovdje je potrebno na početku svim predikatima resetirati vrijednosti sa *retractall* naredbom, jer se inače vrijednosti predikata čuvaju iz prošlog izvođenja programa. Varijable N, *solutions* i *index* imaju istu funkciju kao i u 4.2.2. poglavlju.

Slijedi generiranje svih mogućih kombinacija brojeva unutar liste. Operacija *permutations* upravo tome i služi. Ona će za zadanu sintaksu stvoriti listu svih mogućih kombinacija te sintakse [30]. Ovdje zadana sintaksa označava sve brojene od 1 do N. Svaku generiranu permutaciju šalje se u *safe/1* predikat, kojemu je funkcija ista kao i u 4.1. poglavlju. Ako predikat odredi da se radi o permutaciji koja sadrži rješenje problema, ona se dodaje u listu rješenja.

Ispiše se količina pronađenih rješenja, te se prvo rješenje ispisuje uz pomoć definiranih funkcija. Nakon što se ispiše prvo rješenje, korisnik može odabrati pregled sljedećeg rješenja ili izlazak iz programa. Ako se dođe do zadnjeg rješenja korisniku se ne šalje nikakav upit, već se program automatski ugasi. Ovdje se rješenje ispisuje u obliku liste stupaca gdje se kraljice nalaze. To se može proširiti na prikaz šahovske ploče uz pomoć Pygame biblioteke, ili

ispisivanjem ploče pomoću polja nula i jedinica kako je i napravljeno u izvornom Python rješenju.

```
N = 5
Searching...

Found solutions: 10
Q1 = [1, 3, 5, 2, 4]

[Enter] for Q2 | (n) for Stop:
Q2 = [1, 4, 2, 5, 3]

[Enter] for Q3 | (n) for Stop: n
```

Slika 25: Ispis kombiniranog programskog rješenja prema Simić Ivanu (autorski rad)

Ovakvo rješenje prikazuje kako je moguće unaprijediti oba jezika njihovim zajedničkim radom. Brže je od izrađenog SWI-Prolog rješenja, te jednostavnije od pronađenog Python rješenja. Kombinacija logičkih i ostalih jezika omogućava stvaranje kompaktnijih programa koji imaju šire mogućnosti programiranja i mogućnost korištenja logike kako bi se brže pronalazila rješenja za problem ili bolje sačuvali podaci unutar jedne baze znanja, te kasnije koristili u daljnjim potrebama programa.

4.4. Zapažanja

SWI-Prolog rješenja mogu biti i jednostavna i kompleksna. **Izrađeno rješenje** ne koristi se nikakvim naprednim metodama i ima malu količinu koda, no zato je sporije od ostalih. **Rješenje listama** brže generira liste, ali je sporije od naprednijih rješenja i nema mogućnost prikazati šahovsku ploču. **Rješenje ograničenjima** žrtvuje jednostavnost za brzinu. Na njemu se vidi kako dodatno kodiranje u SWI-Prologu može optimizirati postojeće programe. **Rješenje modulima** prikazuje da se brzina ne dobiva samo sa većom količinom koda. To se može nadomjestiti modulima koji uvode dodatne funkcije u program.

Dakle, SWI-Prolog rješenja postoje u obliku sa manje kodiranja koji je namijenjen za učenje programiranja i u optimiziranom obliku sa više kodiranja za komercijalne svrhe. Također se, zbog načina na koji SWI-Prolog ispituje upite, automatski mogu pronaći sva rješenja za problem, bez da je to potrebno isprogramirati.

Za razliku od toga, Python rješenja uvijek zahtijevaju puno kodiranja. **Izvorno rješenje** primjer je jednostavnijeg rješenja, no ono nije u stanju izvesti sve funkcionalnosti koje ostali SWI-Prolog programi izvode bez puno napora. **Pygame rješenje** je nadograđeno izvorno rješenje. Ono ima iste funkcionalnosti kao i SWI-Prolog programi, ali je za to bilo potrebno

proširiti postojeće funkcije i uvesti dodatnu biblioteku (Pygame) koja zahtjeva dodatno učenje njene sintakse.

Dakle, Python rješenja uvijek su kompleksnija od SWI-Prolog rješenja. Za pronalazak svih rješenja i njihov ispis potrebno je uvesti dodatne funkcije, pogotovo ako se rješenje želi prikazati kao šahovska ploča, nešto što se u SWI-Prologu izvodi sa jednom naredbom (*use_rendering*). Ali zato su ona i brža. Također je, pošto ima puno više izvora za Python na internetu, puno lakše pronaći razne načine za dodavanje potrebnih funkcionalnosti u program.

U **Pyswip rješenju** vidi se sinergija između SWI-Prologa i Pythona. Jezici sudjeluju na jednostavan način gdje SWI-Prolog pronalazi rješenja, a Python generira liste i ispisuje pronađena rješenja. Rezultat ove sinergije je zlatna sredina ostalih rješenja. Nije jednostavno kao ostala SWI-Prolog rješenja, ali je puno jednostavnije od svih Python rješenja. Nije brzo kao i Python rješenja, ali zato je brže od ostalih SWI-Prolog rješenja.

Rješenje će uvijek biti jednostavnije od drugih Python rješenja, ali njena brzina može biti još bolja. Program bi se mogao ubrzati povećanjem interakcije jezika, tako da oba jezika sudjeluju u procesu rekurzije. Za isti cilj može se i samo optimizirati SWI-Prolog rekurzija. Program možda nije u stanju ispisati rješenje kao i ostali SWI-Prolog programi, ali zato Python omogućava dodavanje više načina ispisa.

Ovdje se interakcija jezika izvela uz pomoć Pyswip biblioteke. Ta biblioteka omogućava jednostavno korištenje SWI-Prolog sintakse unutar Python programa. Sintaksa se može doslovno prepisati iz postojećeg programa. Također je moguće uvesti SWI-Prolog program sa računala, samo ga je potrebno držati na istom mjestu kao i Python program. Ovaj način interakcije nije savršen. Može doći do puno komplikacija tijekom instalacije Pyswip biblioteke, pošto je za nju potrebno na računalo preuzeti SWI-Prolog. Zbog toga može doći do pogrešaka jer Pyswip ne može raditi u svim Python okruženjima. Uz to, potrebno je paziti da su instalirana verzija SWI-Prologa i Python okruženje kompatibilni. Nema puno izvora na internetu za Pyswip, pa se često puno vremena potroši za ovu naizgled jednostavnu instalaciju.

Za kraj je kreirana tablica koja sadržava aspekte svakog prikazanog rješenja. Ovako se mogu na jednom preglednom mjestu vidjeti zaključci za svako rješenje. Tako ih se lakše može međusobno uspoređivati. Za svako rješenje može se vidjeti sljedeće: broj slike na kojoj je ono prikazano, autor, opis, karakteristike i napomene vezane za implementaciju. Kako bi se mogao raspoznati programski jezik korišten za rješenje, redovi su označeni posebnim bojama. SWI-Prolog rješenja označena su narančastom bojom, Python žutom bojom, a njihova kombinacija plavom bojom.

Tablica 1: Prikaz svih rješenja

Slika	Autor	Opis	Karakteristike	Napomene
12	Ivan Simić	Izrađeno rješenje	• Sporo • Jednostavno • Renderiranje	Najjednostavnije rješenje, pogodno za učenje SWI-Prolog programiranja i sintakse.
14	Ron Danielson	Rješenje listama	• Sporo • Jednostavno	Jedino SWI-Prolog rješenje bez renderiranja, koristi se i redak i stupac za polje, potrebno uređivanje koda za ispitivanje drugih N varijabli.
16	Richard O'Keefe	Rješenje ograničenjima	• Brzo • Kompleksno • Renderiranje	Najkompleksnije SWI-Prolog rješenje, teško pratiti tijek koda, za ljude sa postojećim SWI-Prolog predznanjem.
17	Markus Triska	Rješenje modulima	• Brzo • Jednostavno • Renderiranje • Moduli	Najbrže SWI-Prolog rješenje, prikazuje korištenje modula unutar SWI-Prologa, potrebno labeliranje tijekom ispisa.
19	Divyanshu Mehta	Izvorno rješenje	• Brzo • Kompleksno • Python ispis	Koristi se i redak i stupac za polje, potrebno uređivanje koda za ispitivanje drugih N varijabli, ispis uz pomoć Python *print* naredbe.
21 i 22	Ivan Simić	Pygame rješenje	• Brzo • Kompleksno • Pygame ispis	Najkompleksnije rješenje, nadograđeno izvorno rješenje, koristi se i redak i stupac za polje, ispis uz pomoć Pygame biblioteke.
24	Ivan Simić	Pyswip rješenje	• Brzo • Jednostavno	Spoj jednostavnosti SWI-Prologa sa brzinom Pythona, Python generira polja, SWI-Prolog obavlja rekurziju, može se dodati ili Python ili Pygame ispis.

5. Zaključak

Prolog i ostali jezici poput njega nisu u tolikoj širokoj upotrebi. Niti se podučavaju na učilištima, niti su previše traženi na tržištu. Zbog njihove unikatne sintakse i potrebnog znanja logike neki programeri mogu imati poteškoća dok uče kako upravljati njima. Međutim, ti jezici imaju specifične slučajeve korištenja u kojima oni najbolje izvršavaju potrebne zadatke. Jedni od tih slučajeva su tehnologije vezane za umjetnu inteligenciju i obradu prirodnog jezika, dvije znanosti koje u današnje vrijeme postaju sve bitnije i bitnije.

Glavnu strukturu Prologa čine logika i njegova deklarativna priroda. Zbog ta dva aspekta Prolog kao jezik savršen je za upravljanje programima baziranim na umjetnoj inteligenciji [31]. Njegovi programi unutar svoje baze mogu imati pohranjeno puno različitih podataka. Velika količina podataka upravo je i potrebna za učenje umjetne inteligencije. Prolog također omogućava pronalaženje uzoraka unutar tih podataka, a pošto se svaki program izvodi sve dok se ne pronađu svi mogući rezultati, program je u stanju prikazati sve pronađene podatke bez da mu se to treba naglasiti.

Zbog povećane vrijednosti Prologa na trenutnom tržištu, programeri mogu uvelike povećati svoj utjecaj i iskustvo ako ga odluče naučiti. Njegovi već navedeni aspekti, poput pronalaženja svih uzoraka unutar velike baze podataka, omogućili su IBM poduzeću da razviju Watson računalo. To računalo koristilo je Prolog za obradu prirodnog jezika i tako pobijedilo ljudskog protivnika u američkoj televizijskoj igri *Jeopardy!* u 2011. godini [32]. Utjecaj Prologa može se jedino proširiti. Čak i može doći do implementacije potpuno novog jezika ili sustava kome će glavni temelj biti upravo Prolog.

Prolog ne mora nužno postati još jedan jezik koji je potrebno naučiti, već ga se može gledati kao nadogradnju koja olakšava korištenje postojećih programskih jezika, bez da narušava njihovu strukturu. Na Pyswip primjeru može se vidjeti koliko je jednostavno koristiti SWI-Prolog sintaksu u nekom drugom programskom jeziku. Upravo to omogućava Prologu da proširi svoj značaj, jer on je jezik sa specifičnom uporabom. Ostalim jezicima, koji se fokusiraju na opću uporabu i korištenje, Prolog predstavlja jednostavnu i brzu alternativu za probleme za koje oni nisu stvoreni.

Popis literature

[1] Stanford University, "Introduction to Logic Programming". Dostupno: http://logicprogramming.stanford.edu/notes/chapter_01.html [pristupano 11.9.2024.].

[2] J. Novotny, "A Guide to Understanding Logic Programming", *Linode Guides & Tutorials*, 4.4.2023. Dostupno: https://www.linode.com/docs/guides/logic-programming-languages/ [pristupano 11.9.2024.].

[3] R. A. Kowalski, "The Early Years of Logic Programming", *Communications of the ACM*, sve. 31, izd. 1, 1988. Dostupno: https://www.doc.ic.ac.uk/~rak/papers/the%20early%20years.pdf [pristupano 11.9.2024.].

[4] Z. Somogyi, F. Henderson, T. Conway i R. O'Keefe, "Logic Programming for the Real World", *Proceedings of the ILPS*, sve. 95, 1955. Dostupno: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=ecf48edcf27350b9d49caa3b8cc33d0a45e12e80 [pristupano 11.9.2024.].

[5] V. Čačić, P. Paradžik i M. Vuković, "Logičko programiranje", *Hrvatski matematički elektronički časopis*. Dostupno: http://e.math.hr/broj_26/Cacic [pristupano 11.9.2024.].

[6] V. Sekovanić, "Uvod u Prolog" nastavni materijali na predmetu Uvod u modeliranje znanja [Moodle], Sveučilište u Zagrebu, Fakultet organizacije i informatike, Varaždin, 2023.

[7] J. Wielemaker i V. S. Costa, "Portability of Prolog programs: theory and case-studies", 2010. Dostupno: https://www.swi-prolog.org/download/publications/porting.pdf [pristupano 11.9.2024.].

[8] P. Körner i sur., "Fifty Years of Prolog and Beyond", *Theory and Practice of Logic Programming*, sve. 22, izd. 6, 2022. Dostupno: https://www.cambridge.org/core/journals/theory-and-practice-of-logic-programming/article/fifty-years-of-prolog-and-beyond/3A5329B6E3639879301A6D44346FD1DD [pristupano 11.9.2024.].

[9] M. Triska, "Frequently Asked Questions". Dostupno: https://www.metalevel.at/prolog/faq/faq.html [pristupano 11.9.2024.].

[10] SWI-Prolog, "Constraint Logic Programming". Dostupno: https://www.swi-prolog.org/pldoc/man?section=clp [pristupano 11.9.2024.].

[11] SWI-Prolog, "Why Use Modules?". Dostupno: https://www.swi-prolog.org/pldoc/man?section=whymodules [pristupano 11.9.2024.].

[12] SWISH, "About SWISH". Dostupno: https://swish.swi-prolog.org/ [pristupano 11.9.2024.].

[13] C. Bowtell i P. Keevash, "The n-queens problem", 2021. Dostupno: https://arxiv.org/pdf/2109.08083v1 [pristupano 11.9.2024.].

[14] W. R. Ball, "Mathematical Recreations and Essays", *Bulletin des sciences mathématiques*, 2008. Dostupno: https://www.gutenberg.org/files/26839/26839-pdf.pdf [pristupano 11.9.2024.].

[15] O. J. Dahl, E. W. Dijkstra i C. A. R. Hoare, "Structured Programming", *Academic Press Ltd.*, 1982. Dostupno: https://seriouscomputerist.atariverse.com/media/pdf/book/Structured%20Programming.pdf [pristupano 11.9.2024.].

[16] B. Bernhardsson, "Explicit solutions to the N-Queens problem for all N", *ACM SIGART Bulletin*, sve. 2, izd. 2, 1991. Dostupno: https://dl.acm.org/doi/pdf/10.1145/122319.122322 [pristupano 11.9.2024.].

[17] Numberphile, "The 8 Queen Problem - Numberphile", 21.8.2015. Youtube [Video datoteka]. Dostupno: https://www.youtube.com/watch?v=jPcBU0Z2Hj8 [pristupano 11.9.2024.].

[18] SWISH, "Example Programs". Dostupno: https://swish.swi-prolog.org/example/examples.swinb [pristupano 11.9.2024.].

[19] M. Triska, "Solving N-queens with Prolog". Dostupno: https://www.metalevel.at/queens/ [pristupano 11.9.2024.].

[20] R. Danielson, "NQUEENS - Prolog". Dostupno: https://www.cse.scu.edu/~rdaniels/html/courses/Coen171/NQProlog.htm [pristupano 11.9.2024.].

[21] SWISH, "N-Queens (traditional)". Dostupno: https://swish.swi-prolog.org/example/queens.pl [pristupano 11.9.2024.].

[22] Python, "Whetting Your Appetite", *Python documentation*. Dostupno: https://docs.python.org/3/tutorial/appetite.html [pristupano 11.9.2024.].

[23] W3Schools, "Introduction to Python". Dostupno: https://www.w3schools.com/python/python_intro.asp [pristupano 11.9.2024.].

[24] D. Kuhlman, "A Python Book: Beginning Python, Advanced Python, and Python Exercises", 15.12.2013. Dostupno: https://www.davekuhlman.org/python_book_01.pdf [pristupano 11.9.2024.].

[25] M. Kiš, "SWI-Prolog i Python", Završni rad, Sveučilište u Zagrebu, Fakultet organizacije i informatike, Varaždin, 2021. Dostupno: https://urn.nsk.hr/urn:nbn:hr:211:927239 [pristupano 11.9.2024.].

[26] D. Mehta, "Python Program for N Queen Problem | Backtracking-3", *GeeksforGeeks*, 31.5.2022. Dostupno: https://www.geeksforgeeks.org/python-program-for-n-queen-problem-backtracking-3/ [pristupano 11.9.2024.].

[27] K. Galli, "How to Program a Connect 4 AI (implementing the minimax algorithm)", 2.1.2019. Youtube [Video datoteka]. Dostupno: https://www.youtube.com/watch?v=MMLtza3CZFM [pristupano 11.9.2024.].

[28] S. Gallivan, "Solution: N-Queens", *DEV Community*, 22.5.2021. Dostupno: https://dev.to/seanpgallivan/solution-n-queens-5hdb [pristupano 11.9.2024.].

[29] Y. Tekol, "PySwip", *GitHub*. Dostupno: https://github.com/yuce/pyswip [pristupano 11.9.2024.].

[30] GeeksforGeeks, "Python – Itertools.Permutations()", 26.8.2022. Dostupno: https://www.geeksforgeeks.org/python-itertools-permutations/ [pristupano 11.9.2024.].

[31] D. Andre, "What is Prolog?", *All About AI*, 21.8.2024. Dostupno: https://www.allaboutai.com/ai-glossary/prolog/ [pristupano 11.9.2024.].

[32] A. Lally i P. Fodor, "Natural Language Processing With Prolog in the IBM Watson System", *The Association for Logic Programming (ALP) Newsletter*, 2011. Dostupno: https://cse.sc.edu/~mgv/csce330f22/prolog/PrologAndWatson1.pdf [pristupano 11.9.2024.].

Popis slika

Popis tablica

Prilog 1 (Izrađeno rješenje)

```
:- use_rendering(chess).

n_queens(N, Q):-
    length(Q, N),
    queens(Q, N),
    safe(Q).

queens([], _).
queens([Q|Qs], N):-
    queens(Qs, N),
    between(1, N, Q).

safe([]).
safe([Q|Qs]):-
    safe(Qs),
    place(Q, Qs, 1).

place(_, [], _).
place(Q0, [Q1|Qs], D0):-
    Q0 =\= Q1,
    Q0 - Q1 =\= D0,
    Q0 - Q1 =\= -D0,
    D1 is D0 + 1,
    place(Q0, Qs, D1).
```

Prilog 2 (Pygame rješenje)

```
import pygame

N = int(input("N = "))
solutions = []
index = 0

def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solveNQUtil(board, col):
    if col >= N:
        solutions.append([row[:] for row in board])

    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            solveNQUtil(board, col + 1)
            board[i][col] = 0

def solveNQ():
    board = [[0] * N for _ in range(N)]
    solveNQUtil(board, 0)

pygame.init()
SIZE = 800
SQUARE = SIZE // N
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
```

```
QUEEN = (180, 135, 98)
TEXT = (20, 130, 120)
screen = pygame.display.set_mode((SIZE, SIZE))
pygame.display.set_caption("N Queens")
font = pygame.font.SysFont("arialblack", 50)

def drawBoard(board):
    for row in range(N):
        for col in range(N):
            color = WHITE if (row + col) % 2 == 0 else BLACK
            pygame.draw.rect(screen, color, (col * SQUARE, row * SQUARE,
SQUARE, SQUARE))
            if board[row][col] == 1:
                pygame.draw.circle(screen, QUEEN, (col * SQUARE + SQUARE
// 2, row * SQUARE + SQUARE // 2), SQUARE // 2)
    label = font.render(f"{index + 1} / {len(solutions)}", True, TEXT)
    screen.blit(label, (10, 0))
    pygame.display.update()

solveNQ()

if solutions:
    running = True
    drawBoard(solutions[index])
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN:
                index += 1
                if index >= len(solutions):
                    running = False
                else:
                    drawBoard(solutions[index])

else:
    print("Solution does not exist")

pygame.quit()
```

Prilog 3 (Pyswip rješenje)

```
from itertools import permutations
from pyswip import Prolog
prolog = Prolog()

prolog.retractall("safe(_)")
prolog.retractall("place(_, _, _)")

prolog.assertz("safe([])")
prolog.assertz("safe([Q|Qs]) :- safe(Qs), place(Q, Qs, 1)")
prolog.assertz("place(_, [], _)")
prolog.assertz("place(Q0, [Q1|Qs], D0) :- Q0 =\\= Q1, Q0 - Q1 =\\= D0,
Q0 - Q1 =\\= -D0, D1 is D0 + 1, place(Q0, Qs, D1)")

N = int(input("N ="))
solutions = []
index = 0

print("Searching...")
print()

perms = [list(perm) for perm in list(permutations(range(1, N + 1)))]
for perm in perms:
    if list(prolog.query(f"safe({perm})")):
        solutions.append(perm)

print("Found solutions:", len(solutions))

for solution in solutions:
    print(f"Q{index + 1} = {solution}")
    print()
    index += 1
    if index < len(solutions):
        if input(f"[Enter] for Q{index + 1} | (n) for Stop:") == 'n':
            break
```