UNIVERZA V LJUBLJANI

Fakulteta za računalništvo in informatiko

Damjan Demšar

INDUKTIVNO LOGIČNO PROGRAMIRANJE S SPECIALIZACIJO HIPOTEZ

DOKTORSKA DISERTACIJA

LJUBLJANA, 2003

UNIVERZA V LJUBLJANI

Fakulteta za računalništvo in informatiko

Damjan Demšar

INDUKTIVNO LOGIČNO PROGRAMIRANJE S SPECIALIZACIJO HIPOTEZ

DOKTORSKA DISERTACIJA

Mentor: prof. dr. Ivan Bratko

LJUBLJANA, 2003

Povzetek

Večina ILP sistemov za induciranje hipotez uporablja prekrivni algoritem in se večinoma razlikujejo le v načinu konstrukcije posamezne klavzule. Vendar so tako konstruirane klavzule lahko le lokalno optimalne in običajno ne sestavljajo globalno optimalne hipoteze. To se še najbolje pokaže pri domenah, pri katerih so v rešitvi posamezne klavzule medsebojno odvisne (recimo rekurzivne domene). Če pa vse klavzule v hipotezi razvijamo vzporedno (drugače rečeno *specializiramo hipoteze*), niso več optimalne klavzule, ampak je optimalna hipoteza (če je res globalno optimalna, je odvisno od preiskovanja). Prednosti se predvsem pokažejo pri konstrukciji rešitev na domenah z medsebojno odvisnimi klavzulami in tudi v velikosti hipotez, ki so pogosto manjše od hipotez, ki jih zgradimo z uporabo prekrivnega algoritma. Te prednosti je pokazal ILP sistem HYPER, ki specializira celotne hipoteze. Čeprav je specializacija celotnih hipotez časovno kompleksnejša od običajnih pristopov h ILP, je bilo ravno tako razvidno, da s primernim vodenjem sistema skozi prostor hipotez najdemo rešitev po pregledu le majhnega delčka celotnega prostora.

Zaradi tega potenciala specializacije hipotez smo v okviru te naloge razvili nov sistem, ki specializira celotne hipoteze (HYPER^2) in je obdržal nekatere lastnosti prejšnjega sistema, ter izboljšal druge lastnosti. Razlike predvsem temeljijo na optimiranosti specializacije hipotez (sistem zaznava in odpravlja nastanke kopij hipotez in klavzul), poleg tega sistem tudi odpravlja nekatera druga podvajanja. Poleg tega smo dodali možnost drugačnega širšega preiskovanja prostora in (kar se je izkazalo kot najbolj donosno) omogočili uporabo informacije o načinu uporabe iskanih predikatov (natančneje o tem ali so argumenti ciljnega predikata vhodni ali izhodni), ki jih osnovni sistem ni uporabljal (čeprav je uporaba teh podatkov razširjena v ILP sistemih).

Spremembe v novem sistemu smo nato temeljito eksperimentalno ovrednotili. Napravili smo več verzij sistema, ki so se razlikovale po vključenosti oziroma izključenosti posameznih sprememb. Te verzije smo nato testirali na dveh domenah programske sinteze in primerjali njihove rezultate (pravilnost, čas, število pregledanih hipotez in število klicev meta-interpreterja) med seboj in z originalnim sistemom ter ugotovili, katere spremembe največ pripomorejo k učinkovisti sistema, katere pripomorejo manj in katere celo poslabšajo učinkovitost sistema.

Ker se je pokazalo, da določene izboljšave (oziroma spremembe, za katere smo predvidevali, da bi lahko izboljšale delovanje sistema) niso prinesle izboljšanja, ampak so celo poslabšale

delovanje v enem ali več vidikih, smo poskušali pomanjkljivosti izboljšati ali nadomestiti z drugačno rešitvijo. Na ta način so nastale zadnje verzije sistema HYPER^2: najprej verzije, ki uporabljajo iskanje s snopom in nato (ko se je videlo, da druge spremembe prinesejo več kot iskanje s snopom) tudi novejše verzije, ki tako kot prve uporabljajo iskanje najprej najboljši. Kot najboljša se je izkazala verzija, ki uporablja iskanje najprej najboljši, zna upoštevati informacije o vhodnem/izhodnem tipu spremenljivk v glavah stavkov hipoteze in si zapomni, katere (negativne) učne primere pokriva posamezna hipoteza.

Nekatere osnovne verzije smo nato na devetih domenah programske sinteze primerjali z drugimi standardnimi ILP sistemi ter analizirali obnašanje vseh sistemov (na osnovi pravilnosti, porabljenega časa in velikosti vrnjenih hipotez). V primerjavi se je pokazalo, da je HYPER^2 manj kot ostali sistemi občutljiv na manjkajoče učne podatke.Še najbolj se mu približa MARKUS, medtem ko je FOIL najbolj občutljiv na manjkajoče podatke. HYPER^2 tudi konsistentno najde majhne hipoteze, kot tudi MARKUS in CHILLIN, medtem ko ALEPH, CPROGOL in FOIL pogosto generirajo prekompleksne hipoteze.

Vendar se je tudi izkazalo, da, predvsem pri domenah z več učnimi podatki, pride do izraza večja kompleksnost preiskovanja celotnih hipotez, in zato HYPER^2 v takih domenah lahko deluje občutno počasneje od ostalih sistemov, med katerimi je običajno najhitrejši FOIL.

V domenah, kjer so iskani predikati nekaj bolj kompleksni (obe domeni z urejanjem seznamov), se kot konstantno dober izkaže predvsem MARKUS, ki najde rešitve v obeh domenah, vendar ne najde pravilne rešitve v dosti preprostejši domeni Next. Domeno InsertionSort poleg MARKUS-a reši še HYPER^2 z iskanjem s snopom (ter občasno tudi druge verzije sistema HYPER^2, ki pa ne rešijo problema v domeni Quicksort). Obe domeni reši tudi FOIL, vendar, kakor kaže, za to potrebuje vse pozitivne učne primere.

Torej je, če je na voljo malo učnih primerov in obstaja sum, da nekateri ključni učni primeri manjkajo, je najbolje uporabiti sistem HYPER^2; če je na voljo več učnih primerov, a kljub temu še ne vsi, je priporočljivo uporabiti MARKUS in poskusiti tudi s HYPER^2; če so pa na voljo vsi učni primeri (ali vsaj velika večina) in je le teh veliko, je priporočljiva uporaba sistema FOIL.

Izkazalo se je, da bi bilo potrebno HYPER^2 še dodatno modificirati. Predvsem bi bilo potrebno sistem pospešiti in mu omogočiti delo z več učnimi primeri, za kar bi bilo potrebno HYPER^2 napisati v kakšnem drugem narečju jezika PROLOG, saj SICStus omogoča delo le z 256MB spomina.

Poleg tega bi bilo potrebno spremeniti način preiskovanja prostora možnih hipotez. HYPER^2 občasno zaide na napačno pot, iz katere se ne zna vrniti. Ker take poti običajno vsebujejo stavke, ki ne pokrivajo nobenega pozitivnega učnega primera, bi lahko take stavke kaznovali, vendar bi bilo potrebno kazen določiti zelo natančno, da ne bi pretirano škodovali dejstvu, da HYPER^2 lahko poišče hipoteze, tudi ko ni pokrit celoten prostor učnih primerov (prav zato ker dovolimo stavke, ki ne pokrivajo pozitivnih učnih primerov).

Abstract

Most ILP systems use covering algorithm to induce hypothesis and only differ in the way they construct clauses. However, clauses constructed one by one can be only locally optimal and usually do not combine into a globally optimal hypothesis. This can be best seen in problem domains, which consist of mutually dependent clauses, such as recursive problem domains. If we construct all clauses in parallel (we specialize whole hypotheses) single clauses may not be optimal, but so constructed hypothesis is optimal (whether it is also globally optimal depends on the search procedure). In this way it is much easier to solve problem domains that demand mutually depended clauses. As a bonus so constructed hypotheses are usually smaller than the hypotheses constructed using covering algorithm. Although the specialization of whole hypotheses is more time consuming than the usual approaches, time consumption can be limited using appropriate guidance through the space of hypotheses (as was shown by the ILP system HYPER).

Because of this potential, we developed a new system, that specializes whole hypotheses (HYPER^2), which retained some properties of its predecessor HYPER and bettered others. The differences are mainly in the optimization of the whole procedure (like detecting and suppressing copies of clauses and hypotheses). We added the possibility of wider searching and added the ability to use the information on the input/output modes of arguments of the searched predicate.

We then thoroughly tested the modifications of the system. After producing several versions, each with different modifications turned off (and using the original procedure), we tested them on two program synthesis domains. The collected data (correctness of produced solutions, time taken, the number of searched hypotheses and the number of calls to the meta-interpreter) was used to compare HYPER and various versions of HYPER^2. We found out which modifications behave as expected and which do not.

Since some of the modifications did not produce the expected improvement (or even deteriorated the performance) we tried to eliminate those weak points. We so produced new versions, first using beam search. Later, when we realized, that some other changes introduced at the same time as beam search produced a better effect, we returned to best-first search using only the other changes (the main one was using the information on the input/output modes of arguments of the searched predicate).

Some of the main versions were then tested on 9 program synthesis domains along with some other standard ILP systems. We analyzed their performance (correctness of the solution, time taken and the size of the produced solution). This analysis showed that HYPER^2 is less sensitive to missing data than the other systems. From the others, the closest is MARKUS, while FOIL is the most sensitive. HYPER^2 also consistently finds small hypotheses, as do MARKUS and CHILLIN, while ALEPH, PROGOL and FOIL often generate too big and too complex hypotheses.

However, the analysis also showed that in more complex domains, with more data, the complex procedure used by HYPER^2 shows its effect and as the result HYPER^2 becomes slower than the other systems (with FOIL being the fastest). In the complex domains, the best performance is shown by MARKUS (which can have troubles with some simpler domains). From the results of analysis we can make the next recommendation: if there is only a small amount of data available, and there exists a suspicion that some key data could be missing, HYPER^2 should be used. If there is more data, but still far from complete coverage, MARKUS should be used, and when all (or almost all) data is available FOIL should be used.

The experiments showed that HYPER^2 should be further modified. First of all it needs to be faster, which needs porting into another Prolog dialect or even another programming language. The searching procedure also needs some additional work, since HYPER^2 occasionally gets into a dead end (starts to search in the wrong direction, sometimes the search returns to the correct path in time, but often it does not). This usually involves clauses which do not cover any available data. However, if we punished such clauses HYPER^2 could lose its ability to construct vital parts of hypothesis when not enough data is available, so that a fine tuned compromise is needed.

Zahvala

Najprej bi se zahvalil mentorju prof. dr. Ivanu Bratku za sprejem v svojo delovno skupino, za usmerjanje pri delu in predvsem za potrpežljivost.

Inštitutskemu mentorju doc. dr. Marku Bohancu in ostalim sodelavcem Odseka za inteligentne sisteme na Institutu Jožef Stefan se zahvaljujem za sprejetje v stimulativno delovno okolje in potrpežljivost, ki so mi jo pokazali.

Ministrstvu za šolstvo, znanost in šport se zahvaljujem za sredstva, ki so mi omogočila delo in usposabljanje na Institutu Jožef Stefan.

Končno se zahvaljujem tudi družini in prijateljem, ki so me spodbujali, ko je bilo potrebno, in me pustili pri miru, ko je bilo to potrebno.

Kazalo

1 Uvod

V tem delu bomo predstavili naše delo na sistemu HYPER^2, ki je izboljšava sistema HYPER [3, 4]. Predstavili bomo tako originalni sistem HYPER kot tudi spremembe uvedene v novi verziji. Prav tako bomo predstavili nekatere spremembe, ki se niso obnesle, kot je bilo pričakovano in poskušali razložiti, zakaj te spremembe ne prinesejo izboljšanja. Vse spremembe smo temeljito eksperimentalno ovrednotili. Pri tem smo primerjali učinek posameznih sprememb med seboj in z originalnim sistemom. Upoštevali smo uspešnost reševanja, potreben čas, število pregledanih hipotez in število klicev meta-interpreterja. Pri teh primerjavah se je izkazalo, da največjo izboljšavo prinese upoštevanje informacije, katere spremenljivke iskanega predikata so vhodne in katere izhodne, ter izboljšave, ki preprečujejo preiskovanje že pregledanih hipotez.

Nekatere verzije, ki se razlikujejo glede na različne izboljšave, smo nato eksperimentalno primerjali s standardnimi sistemi induktivnega logičnega programiranja. Poskusi na domenah programske sinteze so pokazali, da pristopi, ki jih uporabljajo, lahko prinesejo veliko prednost, ko sistemi nimajo na voljo dovolj podatkov. Vendar se prav tako izkaže, da prav pristopi, ki so učinkoviti pri redki množici učnih podatkov, lahko zavedejo sistem iz prave smeri. Najbolj očiten pristop, ki to omogoča, je dovolitev stavkov, ki ne pokrivajo nobenega učnega primera. Prav tu se kaže potreba po nadaljnjem delu in po ustreznem kompromisu.

Glavni dosežki našega dela so: sam sistem HYPER^2, ugotovitve, katere izboljšave delujejo najbolje in zakaj nekatere ne delujejo, primerjava nekaterih ILP sistemov ter ugotovitve njihovih slabosti.

1.1 Organizacija dela

Najprej bomo v uvodu v motivaciji navedli razloge za delo na tej problematiki. Nato bomo na kratko povzeli rezultate. V drugem poglavju bomo na osnovnem nivoju predstavili induktivno logično programiranje, kolikor je potrebno za lažje razumevanje preostanka disertacije. V tretjem poglavju bomo predstavili sistem HYPER in modifikacije, ki smo jih uvedli v sistemu HYPER^2. Nato bomo (četrto poglavje) primerjali učinke modifikacij na dveh domenah programske sinteze in nato še (peto poglavje) primerjali delovanje sistema HYPER^2 z nekaj drugimi ILP sistemi na večih domenah programske sinteze. Pri vsakem primerjanju se bomo osredotočili na učno krivuljo algoritmov, oziroma na občutljivost algoritma na pomanjkljive –

redke podatke ter poskusili pojasniti rezultate. V zaključku bomo povzeli naše ugotovitve in podali priporočila glede uporabe različnih ILP sistemov glede na problemsko domeno.

1.2 Motivacija

Tema sodi v področje strojnega učenja oziroma na podpodročje induktivnega logičnega programiranja (ILP). Večina ILP sistemov kot so ALEPH [33], CPROGOL [21, 22], FOIL [29, 30], in MARKUS [13, 14],uporablja prekrivni algoritem, ki generira hipoteze klavzulo za klavzulo. Algoritem začne s prazno hipotezo, ki ji doda novo klavzulo. Nato se pozitivni primeri, ki jih pokriva klavzula, odstranijo iz učne množice. Postopek generiranja nove klavzule in odstranjevanja primerov se ponovi, dokler niso iz učne množice odstranjeni vsi pozitivni primeri, to je dokler hipoteza ne pokriva vseh pozitivnih primerov.

Poleg prekrivnega algoritma obstajajo tudi drugi načini generiranja končnih rešitev, vendar jih uporabljajo le redki sistemi. CHILLIN [34, 35] uporablja kombinacijo generalizacije in specializacije. TILDE [2] generira kompletne hipoteze hkrati, vendar ima vdelane nekatere omejitve, ki zmanjšajo izrazno moč teh hipotez. Enostaven algoritem, ki ga uporablja HYPER [4], ki generira celotne hipoteze, pokaže, da se z generiranjem celotnih hipotez namesto klavzul lahko izognemo pomanjkljivostim prekrivnega algoritma.

Prekrivni algoritem je običajno požrešen, saj pri vsaki ponovitvi doda klavzulo, ki je bila optimalna po nekem kriteriju. Ta klavzula pa je običajno optimalna le lokalno, glede na trenutno množico preostalih učnih primerov. Algoritem ne zagotavlja globalno optimalne hipoteze. Na drugi strani pa klavzule, ki sestavljajo globalno dobro (optimalno) hipotezo, same po sebi niso nujno lokalno optimalne. Tipični problemi, ki jih ima prekrivni algoritem, so med drugim:

- preveč klavzul v hipotezi
- neučinkovito in problematično obravnavanje rekurzije (sistemi, ki uporabljajo prekrivni algoritem, potrebujejo v učni množici vse primere, v katere se z rekurzijo razvije poljubni primer v učni množici)
- neučinkovito učenje več, medsebojno odvisnih predikatov (podobno kot pri rekurziji algoritem potrebuje v učni množici vse pozitivne primere, v katere se razvije poljuben učni primer)

Zaradi teh problemov se pojavi ideja, da bi generirali celotne hipoteze hkrati. S tem bi odstranili slabosti prekrivnega algoritma. Vendar pa takoj najdemo možen argument proti tej

ideji; če generiramo celotne hipoteze, s tem preiskujemo prostor, ki je večji od že zelo velikega prostora pri iskanju posameznih klavzul. Ta problem je razlog, da so se do sedaj le redki lotili generiranja celotnih hipotez. Midelfart [17] teoretično obravnava specializacijo celotnih hipotez, ampak te ideje ni uporabil v delujočem sistemu. Sistem TILDE [2] generira celotne hipoteze hkrati v obliki odločitvenega drevesa s pogoji v vozlih v logiki prvega reda. Vendar TILDE ne zna reševati rekurzivnih domen. HYPER [4] je sistem, ki sprotno generira celotne hipoteze in nima podobnih omejitev kot TILDE. Algoritem, ki ga uporablja HYPER, bomo uporabili kot osnovo v doktorski disertaciji.

1.3 Rezultati in prispevki

Izvirni prispevki k znanosti so:

1. **razvoj in implementacija sistema HYPER^2**
2. **analiza velikosti preiskanega prostora in kompleksnosti algoritma**
3. **ugotovitve, kateri poskusi izboljšav originalnega sistema HYPER delujejo in kateri ne**. Kot najbolj pomembne izboljšave so se izkazale:

   - **Grajenje gozda stavkov.** Iz vsakega podanega začetnega stavka gradimo drevo naslednikov začetnega stavka. Same hipoteze so sestavljene iz kazalcev na stavke. Na ta način nam ni potrebno večkrat izračunavati naslednikov enega stavka (ki se lahko pojavi v več hipotezah). Najbolj pomembna prednost grajenja gozda stavkov pa je enostavnost preverjanja obstoja kopij stavkov in še lažjega preverjanja kopij hipotez, kar preprečuje ponavljanje procesiranja.

   - **Hipoteze poznajo učne primere, ki jih pokrivajo.** Ker se pri ostrenju pokrivanje primerov lahko samo zmanjša, je treba pri računanju pokritosti učnih primerov s strani naslednikov preveriti samo primere, ki so jih pokrivali starši.

   - **Uporaba informacije o vhodno/izhodnem tipu spremenljivk v glavah stavkov hipoteze.** Če uporabimo tako informacijo, se nekoliko zmanjša prostor možnih hipotez, kar že samo po sebi lahko pospeši delovanje. Poleg tega pa se izkaže, da je lahko poskus ovrednotenja hipoteze, ki s tem znanjem ne bi bila upoštevana (in recimo uporablja drugače izhodno spremenljivko, ki še nima znane vrednosti kot vhod pri klicu predznanja) lahko povzroči upočasnitev delovanja (zaradi možnega "neskončnega" števila odgovorov napačno klicanega predznanja).

Nekaj "izboljšav", ki niso prinesle pričakovanega izboljšanja:

- Stavki poznajo primere, ki jih pokrivajo. Ta osnovna izboljšava, od katere smo pričakovali enega izmed večjih izboljšanj, se ni obnesla po pričakovanjih (zato je bila tudi narejena verzija, ki si zapomni pokritost na nivoju hipotez). Izkazalo se je, da sta glavna problema za to izboljšavo prekrivanje pokritosti – ko več stavkov v hipotezi pokrije isti primer. To izniči prednost, da je potrebno za vsak stavek le enkrat izračunati pokritost. Glavna težava pa je bila dejstvo, da stavki, odvisni od preostanka hipoteze, pokrivajo primere drugače v vsaki hipotezi (tega smo se sicer zavedali, ampak nismo pričakovali tako velikega učinka).
- Izboljšava, ki upošteva, da je nadmnožica kompletne hipoteze (pogosto) tudi kompletna, ni prinesla pričakovanega izboljšanja. Problem je predvsem veliko število kompletnih hipotez, ki so kandidati za podmnožico hipoteze, ki jo trenutno preverjamo. Dodatne težave prinese dejstvo, da v okviru meta-interpreterja vrstni red stavkov lahko vpliva na to, ali hipoteza pokriva določen primer ali ne.

4. **ugotovitve primerjalnega testiranja z drugimi ILP sistemi:**
   - $HYPER^2$ se običajno bolje obnese na zelo redkih učnih množicah in je pogosto sposoben generirati pravilne rešitve iz zelo majhnega števila učnih podatkov
   - $HYPER^2$ je zelo občutljiv na kompleksnost domene
   - FOIL je najbolj občutljiv na manjkajoče podatke, a je običajno najhitrejši
   - ALEPH in PROGOL potrebujeta večje število učnih primerov določene oblike
   - CHILLIN potrebuje lepo porazdeljene pozitivne in negativne učne primere
   - MARKUS je običajno dober kompromis, a mu lahko enostavne domene povzročajo težave

2 Induktivno logično programiranje

V tem razdelku bomo podali enostaven opis induktivnega logičnega programiranja (opis je povzet po [12]) in nekaterih osnovnih konceptov, ki jih uporabljamo v nalogi. Poznavanje teh konceptov poenostavi razumevanje preostanka te naloge.

2.1 Sintaksa

Induktivno logično programiranje (ILP) [12, 20] je panoga strojnega učenja, ki se ukvarja z učenjem logičnih programov iz učnih primerov in predznanja. Za to uporablja jezik logike prvega reda. Le ta je sestavljen iz spremenljivk, predikatnih simbolov in funkcijskih simbolov (kamor spadajo tudi konstante). Spremenljivka in funkcijski simbol, ki mu sledi v oklepajih našteta n-terica izrazov, sta izraza. Predikatne in funkcijske simbole običajno predstavljamo z nizom (črk, številk in nekaterih simbolov), ki se začne z malo začetnico, medtem ko so spremenljivke običajno predstavljene z nizom z veliko začetnico. Konstanta je funkcijski simbol z mestnostjo 0 (torej mu v oklepajih sledi 0-terica izrazov). Predikatni simbol, ki mu v oklepajih sledi n-terica izrazov, se imenuje atom (atomska formula).

Dobro oblikovana formula je ali atom ali pa je ene od naslednjih oblik: F, (F), $\overline{F}$, $F \vee G$, $F \wedge G$, $F \leftarrow G$, $F \leftrightarrow G$, $\forall X{:}F$ in $\exists X{:}F$. Kjer sta F in G dobro oblikovani formuli, $\overline{F}$ je negacija F, $\vee$ predstavlja logično disjunkcijo (ali), $\wedge$ predstavlja logično konjunkcijo (in), $\leftarrow$ predstavlja logično implikacijo (če G potem tudi F), $\leftrightarrow$ pa ekvivalenco (F natanko takrat ko G). $\forall$ je univerzalni kvantifikator (za vsak X velja F), $\exists$ pa eksistenčni kvantifikator (obstaja X tako, da velja F). V formulah $\forall X{:}F$ in $\exists X{:}F$ so vse pojavitve spremenljivke X vezane. Stavek ali zaprta formula je dobro oblikovana formula, v kateri so vse pojavitve vsake spremenljivke vezane.

Posebna oblika za formule prvega reda je klavzalna oblika. Klavzula je disjunkcija literalov (pozitivni literal je atom, negativni literal pa negacija atoma), pred katero je predpona univerzalnih kvantifikatorjev (eden za vsako spremenljivko v disjunkciji). Drugače rečeno, klavzula je formula oblike $\forall X_1 \forall X_2 \ldots \forall X_s (L_1 \vee L_2 \vee \ldots L_m)$, kjer so L_i literali in $X_1, X_2, \ldots X_s$ so vse spremenljivke, ki se pojavijo v $L_1 \vee L_2 \vee \ldots L_m$.

Klavzulo lahko predstavimo tudi s končno (lahko tudi prazno) množico literalov. Množica $\{A_1, A_2, \ldots, A_a, \overline{B_1}, \overline{B_2}, \ldots, \overline{B_b}\}$, kjer so A_i in B_i atomi, predstavlja klavzulo

$\left(A_1 \vee A_2 \vee \ldots \vee A_a \vee \overline{B_1} \vee \overline{B_2} \vee \ldots \vee \overline{B_b}\right)$, ki jo prav tako lahko predstavimo v obliki $A_1 \vee A_2 \vee \ldots \vee A_a \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_b$. Najbolj običajno pa je predstavljena v obliki $A_1, A_2, \ldots, A_a \leftarrow B_1, B_2, \ldots, B_b$, kjer $A_1, A_2,...,A_a$ imenujemo glava in $B_1, B_2,...,B_b$ telo klavzule, vejice v glavi predstavljajo logično disjunkcijo, vejice v telesu pa logično konjunkcijo. Množico klavzul imenujemo klavzalna teorija in predstavlja konjunkcijo njenih klavzul.

Klavzula je Hornova klavzula, če vsebuje največ en pozitivni literal, in je definitna klavzula, če vsebuje natanko en pozitivni literal. Množico definitnih klavzul imenujemo definitni program. Dejstvo je definitna klavzula s praznim telesom. Cilj je Hornova klavzula brez pozitivnih literalov.

Programska klavzula je klavzula oblike $A \leftarrow L_1, L_2, \ldots, L_m$, kjer je A atom in vsak L_i je pozitivni ali negativni literal. Negativni literal v telesu programske klavzule je napisan v obliki *not B*, kjer je B atom. Normalni program (ali logični program) je množica programskih klavzul. Definicija predikata je množica programskih klavzul z istim predikatnim simbolom in mestnostjo v glavah stavkov.

Množico spremenljivk v izrazu, atomu ali klavzuli F označimo kot *vars(F)*. Zamenjava (substitution) $\theta = \{V_1 / t_1, \ldots, V_n / t_n\}$ je določitev izrazov t_i spremenljivkam V_i. Uporaba zamenjave θ na izrazu, atomu ali klavzuli F proizvede instanciran izraz, atom ali klavzulo $F\theta$, kjer so vse pojavitve spremenljivke V_i zamenjane z izrazom t_i. Izraz, atom ali klavzulo imenujemo utemeljeno (grounded), kadar v njej ne nastopa nobena spremenljivka.

Klavzula ali klavzalna teorija je prosta funkcij, če vsebuje samo spremenljivke kot izraze, to je če ne vsebuje nobenega funkcijskega simbola (torej tudi nobene konstante). Na primer, *hcer(X,Y)* ← *zenska(X), mati(Y,X)* je prosta funkcij, medtem ko *even(s(s(X))* ← *even(X)* ni prosta funkcij. Datalog klavzula (ali program) je definitna klavzula (program), ki ne vsebuje nobenega funkcijskega simbola z nenično mestnostjo. To pomeni, da se samo spremenljivke in konstante lahko uporabijo kot argumenti predikata. Velikost izraza, atoma, klavzule ali klavzalne teorije T, je število simbolov, ki se pojavi v T, to je število vseh pojavitev predikatnih simbolov, funkcijskih simbolov in spremenljivk v T.

2.2 Naloga ILP

Običajna naloga ILP je učenje logičnih definicij relacij [29], kjer dobi na razpolago primere, ki pripadajo ali ne pripadajo iskani relaciji. Iz učnih podatkov nato ILP sistem inducira logični

program (oziroma definicijo predikata), ki definira iskano relacijo glede na ostale relacije, ki so bile podane kot predznanje.

Natančneje, podane so n-terice, ki pripadajo iskani relaciji *p* (pozitivni primeri), n-terice, ki ne pripadajo iskani relaciji (negativni primeri), podano predznanje (v obliki relacij ali predikatov) in jezik hipotez, ki določa sintaktične omejitve za definicijo predikata *p*. Naloga je najti definicijo predikata *p*, ki je konsistentna (ne pokriva nobenega negativnega primera) in kompletna (pokriva vse pozitivne primere).

Če to napišemo bolj formalno: pri dani množici primerov $E = P \cup N$, kjer je P množica pozitivnih učnih primerov in N množica negativnih učnih primerov, ter podanem predznanju B, je potrebno najti hipotezo H, tako da velja $\forall e \in P: B \wedge H \models e$ (H je kompletna, H skupaj z predznanjem pokriva primer e) in $\forall e \in N: B \wedge H \not\models e$ (H je konsistentna) [10], kjer $\models$ predstavlja logično implikacijo.

Za generiranje rešitev ILP sistemi običajno uporabijo prekrivni algoritem. Ta v glavni zanki sestavlja množico klavzul. Začenši s prazno množico, generira klavzulo, ki razloži nekaj pozitivnih učnih primerov, nato doda to klavzulo v množico (hipotezo), odstrani pozitivne učne primere, ki jih je dodana klavzula razložila, in ponavlja, dokler niso razloženi vsi pozitivni učni primeri (oziroma dokler hipoteza ni kompletna).

V notranji zanki se generirajo klavzule s pomočjo hevrističnega preiskovanja prostora možnih klavzul (kot ga oblikuje uporabljen specializacijski oziroma ostrilni ali generalizacijski oiroma. posplošitveni operator). Običajno začne s splošno klavzulo (ki ne vsebuje pogojev v telesu), nato dodaja literale v telo, dokler klavzula ne razloži (pokriva) samo pozitivne učne primere (je konsistentna). Preiskovanje je lahko od spodaj omejeno s pomočjo tako imenovanih spodnjih klavzul, ki se naredijo s pomočjo najmanj splošne posplošitve (least general generalization) ali pa z obrnjenim zaključkom (inverse resolution /entailment).

2.3 Struktura prostora klavzul

Da lahko sistematično pregledujemo prostor programskih klavzul, je koristno v sam prostor vpeljati nekakšno strukturo oziroma ureditev. Taka ureditev recimo temelji na *θ-zajetju* (*θ-subsumption*).

Najprej se je potrebno spomniti, da klavzule lahko zapišemo tudi v obliki množice. Tako lahko *hcer(X,Y)←stars(Y,X)* zapišemo tudi kot $\{hcer(X,Y), \overline{stars(Y,X)}\}$.

Recimo da sta c in c' programski klavzuli. Klavzula c θ-zajame klavzulo c', če obstaja taka zamenjava θ, da velja $c\theta \subseteq c'$. Drugače rečeno, če obstaja taka zamenjava spremenljivk v c z izrazi (treba je spomniti, da je spremenljivka tudi izraz), da se vsak izmed literalov (po spremembi) pojavi v c'.

θ-zajetje uvede tudi pojem splošnosti. Klavzula c je vsaj toliko splošna kot c' ($c \leq c'$), če c θ-zajame c'. Klavzula c je bolj splošna kot c' ($c < c'$), če velja $c \leq c'$, $c' \leq c$ pa ne velja. V tem primeru rečemo, da je c' specializacija (ali ostritev) od c, c pa je generalizacija (posplošenje) od c'.

Za *θ-zajetje* velja:

- Če c θ-zajame c', potem c logično povzroči (implicira) c', oziroma $c \models c'$, obratno pa ne velja vedno.
- Relacija $\leq$ uvede mrežo nad množico skrčenih (reduced) klavzul [25]. Skrčene klavzule so najmanjši predstavniki ekvivalenčnih razredov klavzul, ki jih določa θ-zajetje. Uvedba mreže pomeni, da imata poljubni skrčeni klavzuli tako najnižjo zgornjo mejo (least upper bound) kot tudi najvišjo spodnjo mejo (greatest lower bound).

Druga lastnost *θ-zajetja* privede do definicije najmanj splošnega posplošenja (least general generalization lgg) [27] dveh klavzul c in c', ki se označi tudi kot *lgg(c,c')*. To je najnižja zgornja meja klavzul c in c', kot jo določa mreža θ-zajetja.

2.4 Preiskovanje prostora klavzul

Večina ILP sistemov preiskuje prostor programskih klavzul od zgoraj navzdol, od splošnih k specifičnim, s pomočjo ostrilnega operatorja [32] na osnovi θ-zajetja. Le ta glede na jezik hipotez $\boldsymbol{L}$ preslika klavzulo c v množico klavzul *p(c)*, množico naslednikov klavzule *c,* ki ji lahko rečemo tudi starš ali starševska klavzula (če obdelujemo celotne hipoteze, lahko uporabimo enake izraze). Bolj formalno lahko množico naslednikov opišemo kot $p(c) \subseteq \{ c' \mid c' \in \boldsymbol{L}, c < c' \}$.

Ostrilni operator običajno vrne množico najmanj izostrenih (najbolj splošnih) izostritev klavzule v okviru θ-zajetja. Na klavzuli v ta namen izvaja dve operaciji:

- izvede zamenjavo nad klavzulo
- doda literal telesu klavzule

Prostor programskih klavzul je mreža, ki jo določa θ-zajetje. V tej mreži lahko definiramo ostrilni graf kot usmerjen, aciklični graf, v katerem so vozlišča programske klavzule in povezave so osnovne ostrilne operacije (zamenjava spremenljivke z izrazom in dodajanje literala v telo).

Ostrilni graf se običajno preiskuje hevristično. Uporabljena hevristika pogosto temelji na številu pozitivnih in negativnih učnih primerov, ki jih klavzula pokriva. Ker je sam graf izredno razvejan, je koristno omejiti število naslednikov, ki jih ima vsaka klavzula (ne da bi s tem izvrgli potencialne pravilne rešitve). En izmed najbolj pogosto uporabljenih načinov, kako zmanjšati velikost ostrilnega grafa, je, da ostrilni operator upošteva tipe argumentov posameznih predikatov ter ali so argumenti vhodni ali izhodni. Na ta način lahko pri vsakem dodajanju novega literala uporabimo za vhodne argumente samo spremenljivke ustreznega tipa, pridobimo pa nove spremenljivke tipov izhodnih argumentov. Na ta način se zmanjša število možnih kombinacij argumentov (manj je tako permutacij argumentov, ker je primernih manj spremenljivk, kakor tudi permutacij starih in novih spremenljivk, ker ostrilni operator točno ve, katere so vhodne in katere izhodne spremenljivke). Preiskovanje je lahko od spodaj omejeno s pomočjo tako imenovanih spodnjih klavzul, ki se lahko naredijo s pomočjo relativne najmanj splošne posplošitve (relative least general generalization) [23] dveh ali več primerov ali pa z obrnjenim zaključkom (inverse resolution /entailment) primera [20]. Iskanje v tem primeru omejimo na poti, ki vodijo do teh spodnjih klavzul.

2.5 Relativna najmanj splošna posplošitev

Ideja najmanj splošne posplošitve je osnova previdne posplošitve: če sta dve klavzuli c_1 in c_2 resnični, je zelo verjetno, da je tudi $lgg(c_1, c_2)$ resnična. Najmanj splošna posplošitev klavzul c_1 in c_2, ki jo označimo $lgg(c_1, c_2)$, je najmanjša zgornja meja klavzul c_1 in c_2, kot jo določa mreža θ-zajetja. Da lahko izračunamo *lgg* dveh klavzul, je najprej potrebno definirati *lgg* izrazov in literalov [27].

lgg dveh izrazov izračunamo po naslednjih pravilih, kjer so s, t, s_i, t_i izrazi ter f in g funkcijska simbola:

- $lgg(t, t) = t$
- $lgg(f(s_1, ..., s_n), f(t_1, ..., t_n)) = f(lgg(s_1, t_1), ..., lgg(s_n, t_n))$
- $lgg(f(s_1, ..., s_m), g(t_1, ..., t_n)) = V$, kjer $f \neq g$ in V je spremenljivka, ki predstavlja $lgg(f(s_1, ..., s_m), g(t_1, ..., t_n))$
- $lgg(s, t) = V$, kjer $s \neq t$ in vsaj ena, bodisi s bodisi t, spremenljivka. V tem primeru V predstavlja $lgg(s, t)$.

Na primer $lgg([a, b, c], [a, c, d]) = [a, X, Y]$ in $lgg(f(a,a), f(b,b)) = f(lgg(a,b), lgg(a,b)) = f(V,V)$, kjer V predstavlja $lgg(a, b)$. Pri izračunavanju *lgg* je potrebno paziti, da se uporabi ista spremenljivka za vse pojavitve *lgg* enakega podizraza (v tem primeru $lgg(a, b)$).

lgg dveh atomov $lgg(A_1, A_2)$ se izračuna kot:

- $lgg(p(s_1, ..., s_n), p(t_1, ..., t_n)) = p(lgg(s_1, t_1), ...,lgg(s_n, t_n))$, če imata atoma isti predikatni simbol p
- $lgg(p(s_1, ...,s_m), q(t_1, ..., t_n))$ je nedefinirano če $p \neq q$

lgg dveh literalov se izračuna kot:

- če sta L_1 in L_2 atoma se $lgg(L_1, L_2)$ izračuna po zgornjem predpisu
- če sta L_1 in L_2 negativna literala, $L_1 = \overline{A_1}$ in $L_2 = \overline{A_2}$, potem $lgg(L_1, L_2)$ izračunamo po naslednjem predpisu: $\lg g(L_1, L_2) = \lg g(\overline{A_1}, \overline{A_2}) = \overline{\lg g(A_1, A_2)}$
- če je L_1 pozitivni in L_2 negativni literal ali obratno je, $lgg(L_1, L_2)$ nedefiniran

Glede na to, da je klavzula množica literalov, lahko *lgg* dveh klavzul izračunamo na naslednji način: če je $c_1 = \{L_1,... ,L_n\}$ in $c_2 = \{K_1,... ,K_m\}$ potem je $lgg(c_1, c_2) = \{M_{ij} = lgg(L_i, K_j) \mid L_i \in c_1, K_j \in c_2, lgg(L_i,K_j)$ obstaja$\}$

Relativna najmanj splošna posplošitev (relative least general generalization, rlgg) je osnovana na semantični splošnosti [7, 25] in upošteva tudi predznanje. Definirana je tako: Klavzula c je vsaj tako splošna kot klavzula c' glede na predznanje B, če $B \cup \{c\} \models c'$. Treba je opozoriti, da je semantična splošnost v splošnem neodločljiva in ne tvori mreže nad klavzulami. *rlgg* dveh klavzul c_1 in c_2 je najmanj splošna klavzula, ki je bolj splošna tako od c_1 kot tudi od c_2 glede

na predznanje *B*. *rlgg* uporablja ILP sistem GOLEM [24], ki pa mora zaradi težav s semantično splošnostjo uporabljati predznanje samo v obliki dejstev. Če je *K* konjunkcija vseh dejstev iz *B*, potem lahko *rlgg* dveh osnovnih atomov (pozitivnih primerov) glede na *B* izračunamo kot $rlgg(A_1, A_2) = lgg((A_1 \leftarrow K), (A_2 \leftarrow K))$.

V splošnem je *rlgg* učnih primerov (ki se uporablja za definicijo spodnjih klavzul) lahko sestavljen iz neskončno literalov. Običajno pa velikost narašča eksponentno s številom učnih primerov. Da zmanjšamo njihovo velikost, lahko uporabimo nekatere omejitve, kot je prepoved literalov, ki so nepovezani z glavo klavzule (preko verige spremenljivk). Kljub omejitvam pa ima lahko spodnja klavzula ogromno število literalov, ponavadi več sto.

V tem podpoglavju smo na kratko predstavili nekatere osnovne koncepte induktivnega logičnega programiranja, katerih razumevanje pripomore k razumevanju preostalega dela naloge. Bolj podrobne osnove ILP (razložene s primeri) so v urejenih knjigah po ILP konferencah [11, 20], samih zbornikih konferenc ILP od 1991 dalje in knjigah o ILP [1, 12, 16, 26].

3 Hyper

V tem poglavju bomo najprej opisali originalni algoritem sistema HYPER [3, 4]. Pri tem se bomo med drugim osredotočili na ostrilni operator (angl: refinement operator) hipotez in na meta–interpreter hipotez. Potrebno je poudariti, da je sistem HYPER namenjen učenju na čistih, brezšumnih podatkih. V drugem delu poglavja se bomo osredotočili na modifikacije in izboljšave originalnega algoritma.

Tako originalni algoritem kot tudi izboljšana verzija sta algoritma v prologu in sicer v narečju SICStus. Ker SICStus prolog obstaja za različne platforme (UNIX, LINUX, MS Windows), tudi HYPER in HYPER^2 lahko tečeta na vseh teh platformah. Ker SICStus prolog lahko naslavlja le 256MB pomnilnika, lahko oba algoritma uporabita največ 256MB (HYPER običajno uporabi le del razpoložljivega prostora, medtem ko HYPER^2 uporabi večji del le tega).

Uporaba obeh sistemov lahko traja v odvisnosti od zahtevnosti domene in števila učnih primerov od manj kot sekundo do več ur. Sistema se nadzirata preko dejstev, ki morajo pred zagonom sistema dodana v bazo prologa.

3.1 Originalni algoritem

Poenostavljen algoritem sistema HYPER je predstavljen na Sl. 3.1. Pomembnejši koraki algoritma so podrobneje predstavljeni v nadaljevanju poglavja.

Postavi začetne hipoteze z 1 do največ $\mathbf{C_{MAX}}$ stavki, jih oceni in vstavi v množico hipotez $\boldsymbol{H_{NS}}$
Ponavljaj dokler nisi našel konsistentne hipoteze in $\boldsymbol{H_{NS}}$ ni prazna
    Vzemi trenutno najboljšo hipotezo **H** v $\boldsymbol{H_{NS}}$
    Inicializiraj množico naslednikov hipoteze **H:** $\boldsymbol{H_N}$ ={}
    Ponovi za vsak stavek **S** iz hipoteze **H**
        Specializiraj stavek **S** in dobi množico naslednikov $\boldsymbol{S'}$
        Za vsak stavek $\mathbf{S'} \in \boldsymbol{S'}$ ponovi
            Dobi množico $\boldsymbol{H_{N'}}$ naslednikov **H**, tako da v **H** zamenjaš stavek **S** s stavkom **S'**
            Dodaj množico naslednikov $\boldsymbol{H_{N'}}$ v $\boldsymbol{H_N}$
        Izloči nekompletne hipoteze iz $\boldsymbol{H_N}$ in oceni ostale hipoteze
        Izloči **H** iz $\boldsymbol{H_{NS}}$
        Vstavi hipoteze iz $\boldsymbol{H_N}$ v $\boldsymbol{H_{NS}}$

Sl. 3.1 Osnovni algoritem sistema HYPER

Ta algoritem preiskuje prostor hipotez, katerega velikost lahko izrazimo z:

$$ProstorHipotez = \sum_{i=1}^{C\max} ProstorStavkov^{i},$$

kjer je *Cmax* največje dovoljeno število stavkov v hipotezi. *ProstorStavkov* je prostor stavkov, ki ga lahko izrazimo kot:

$$ProstorStavkov = SC * \sum_{i=0}^{L\max} \prod_{j=1}^{i} \sum_{pred \in Predznanje} ProstorPredikatov(pred, Spremenljivke(Stavek)),$$

kjer je *SC* število od uporabnika definiranih začetnih stavkov. *Lmax* je največja dovoljena dolžina stavka, *Predznanje* je množica od uporabnika definiranih predikatov predznanja, *Spremenljivke(Stavek)* pa je množica spremenljivk, ki nastopajo v stavku do tega predikata. *ProstorPredikatov* je prostor predikata, ki ga lahko ocenimo z:

$$\begin{aligned} ProstorPredikatov(pred, Sprem) = & \prod_{j=1}^{Mestnost(pred)} (st\_sprem(tip\_argumenta(pred, j), Sprem) + \\ & + st\_izrazov(tip\_\arg umenta(pred, j), Sprem) + \\ & + izhod(pred, j)) \end{aligned},$$

kjer je *Mestnost* mestnost predikata, *tip_argumenta(pred,j)* pa je tip argumenta, ki ga predikat zahteva na mestu *j*. *st_sprem* je število spremenljivk, ki jih lahko uporabimo kot argument, in ga lahko izrazimo kot:

$$st\_sprem(T, Sprem) = \left|\{a \mid a \in Sprem \,\&\, tip(a) = T\}\right|,$$

kjer je *type(a)* tip spremenljivke *a*.

st_izrazov je število izrazov, ki jih lahko uporabimo kot argument predikatu in ga lahko izrazimo kot:

$$st\_izrazov(T, Sprem) = \left|\{x \mid tip(x) = T \,\&\, x = izraz(Sprem)\}\right|.$$

Pri tem je *izraz(Sprem)* od uporabnika definiran izraz, ki lahko vsebuje spremenljivke *Sprem*. Treba je poudariti, da je zelo enostavno definirati rekurzivno definicijo izraza, ki dopušča neskončno izrazov.

izhod(pred,j) pa doda možnost nove spremenljivke, če je j-ti argument predikata izhoden:

$$izhod(pred, j) = \begin{cases} \mathbf{1} \text{ če je j - ti argument predikata pred izhoden} \\ \mathbf{0} \text{ sicer} \end{cases}.$$

Če poenostavimo (vse spremenljivke enakega tipa, vsi predikati predznanja z enako mestnostjo, ...) lahko prostor hipotez izrazimo kot:

$$ProstorHipotez = Cmax*SC^{Cmax}*Lmax^{Cmax}*StPred^{Lmax*Cmax}* \\ *(StPred*Mestnost + St\_izrazov + 1)^{Mestnost*Lmax*Cmax}.$$

Prostor hipotez torej eksponentno narašča z mestnostjo predikatov predznanja, največjo dovoljeno dolžino stavkov in številom stavkov v hipotezi.

3.1.1 Preiskovanje

Sistem HYPER [3, 4] v nasprotju z večino ostalih ILP sistemov ne specializira posameznih stavkov, ampak celotne hipoteze. To počne od zgoraj navzdol z generiranjem in preiskovanjem gozda hipotez, dokler ne najde konsistentne hipoteze – to je hipoteze, ki ne pokriva nobenega negativnega primera. Koren vsakega drevesa v gozdu je ena izmed začetnih hipotez, ki so sestavljene iz enega do $\mathbf{C_{MAX}}$ začetnih stavkov, ki jih poda uporabnik sistema. Vozlišča takega drevesa pa so hipoteze **H**, ki imajo za naslednike hipoteze $\mathbf{H_i}$, ki so v nekem smislu najmanj specifične izostritve hipoteze **H**. Vsak naslednik je vsaj enako - če ne bolj - specifičen od predhodne hipoteze. Zato vsak naslednik hipoteze **H** pokriva podmnožico primerov (ni nujno, da pravo podmnožico – pokriva lahko enako množico), ki jih pokriva hipoteza **H**. Pokritost primerov izračuna HYPER glede na vgrajeni meta-interpreter, pri katerem je dokaz pokritosti omejen na dolžino (in ne globino) dokaza.

HYPER začne preiskovanje z množico, ki vsebuje vse možne hipoteze, sestavljene iz enega do $\mathbf{C_{MAX}}$ začetnih stavkov, ki jih definira uporabnik. Običajno so začetni stavki le glave stavkov, vendar lahko uporabnik definira tudi bolj specifične stavke in na ta način usmeri preiskovanje v želeno smer. Za primer vzamemo učenje o sodosti/lihosti dolžine seznamov, to sta predikata even(List) in odd (List). Največje dovoljeno število stavkov v hipotezi $\mathbf{C_{MAX}}$ naj bo nastavljeno na tri od uporabnika pa naj bosta podana začetna stavka:

odd(L).
even(L).

V tem primeru dobimo naslednjo začetno množico hipotez:

odd(L).
even(L).
odd(L). odd(L).

odd(L). even(L).

even(L). odd(L).

even(L). even(L).

odd(L). odd(L). odd(L).

odd(L). odd(L). even(L).

odd(L). even(L). odd(L).

odd(L). even(L). even(L).

even(L). odd(L). odd(L).

even(L). odd(L). even(L).

even(L). even(L). odd(L).

even(L). even(L). even(L).

Kot je razvidno, bi v takem primeru HYPER generiral 14 začetnih hipotez. Vsaka od teh predstavlja koren drevesa v gozdu, ki ga preiskuje HYPER. Vsako od začetnih hipotez se preveri ali je kompletna, to je ali pokriva vse pozitivne primere. Glede na to, da nasledniki teh hipotez lahko pokrivajo največ iste primere (običajno pa le podmnožico), ni mogoče iz hipoteze, ki ne pokriva vseh pozitivnih primerov, generirati hipoteze, ki bi prekrivala vse pozitivne primere. Iz tega razloga lahko izločimo hipoteze, ki niso kompletne. V primeru učenja sodosti/lihosti dolžine seznama bi običajno (odvisno od podanih primerov) izločili vse hipoteze, ki vsebujejo samo stavke za odd ali samo za even, saj ne morejo pokrivati primerov tako za odd kot tudi za even, torej niso kompletne (razen če bi bila množica primerov izrojena in bi vsebovala primere samo enega tipa).

HYPER preiskuje gozd hipotez po principu najprej najboljši. Pri tem so vse (tudi začetne) hipoteze ocenjene s ceno, ki upošteva tako velikost hipoteze kot tudi število pokritih negativnih primerov. Ta cena je preprosta utežena vsota:

$$Cena(H) = C_1 * Velikost(H) + C_2 * PokritiNeg(H)$$

Pri tem je PokritiNeg(H) preprosto število negativnih primerov, ki jih pokriva hipoteza, Velikost(H), pa je zopet utežena vsota:

$$Velikost(H) = C_3 * StLiteralov(H) + C_4 * StSpremenljivk(H)$$

Pri tem je StLiteralov(H) število literalov, ki sestavljajo celotno hipotezo, in StSpremenljivk(H) je število spremenljivk, ki se pojavljajo v hipotezi. Izkazalo se je, da preiskovanje dobro deluje z naslednjimi vrednostmi utežnostnih konstant:

$C_1 = 1$

$C_2 = 10$

$C_3 = 10$

$C_4 = 1$

Torej je cena izračunana z naslednjo formulo:

Cena(H) = 10 * StLiteralov(H) + StSpremenljivk(H)+ 10 * PokritiNeg(H)

Vrednosti parametrov C_i se lahko razložijo na naslednji način: literali povečujejo kompleksnost hipoteze. Vsak pokriti negativni primer prispeva k ceni hipoteze toliko kot en literal. To se lahko razloži kot ceno dodatnih literalov, ki bili potrebni, da se naštejejo izjeme k hipotezi. Število spremenljivk pa odloča med drugače enako ocenjenimi hipotezami.

V vsaki iteraciji preiskovanja se vzame trenutno najboljša hipoteza, nato se generirajo nasledniki le-te. Nasledniki se ocenijo in kompletni nasledniki se vstavijo v urejeno množico hipotez.

3.1.2 Ostrilni operator

Specializacija hipotez v sistemu HYPER se začne z izbiro stavka v hipotezi. Vse možne specializacije dobimo tako, da se iterativno izbere in potem tudi specializira vsak stavek v hipotezi. Nadalje se sestavijo nasledniki hipoteze, tako da se v hipotezi zamenja izbrani stavek z njegovimi nasledniki:

$$\boldsymbol{H_N} = \{ \mathbf{H} - \{\mathbf{S}\} \cup \{\mathbf{S'}\} \mid \mathbf{S} \in \mathbf{H} \wedge \mathbf{S'} \in \boldsymbol{S_N} \}$$

Tu je $\boldsymbol{H_N}$ množica naslednikov hipoteze **H**, **S** je stavek v hipotezi **H** in $\boldsymbol{S_N}$ je množica naslednikov stavka **S**, to je možnih specializacij stavka **S**. Pri tem se izkaže, da lahko pogosto uporabimo hevrističen pristop, ki lahko zmanjša kompleksnost. Če določen stavek v hipotezi sam, brez ostalega dela hipoteze (na nek razširjen način lahko rečemo nerekurzivno), pokrije kak negativni primer, je nujno potrebno specializirati ta stavek. Zato lahko v tem koraku specializiramo samo ta stavek in prestavimo specializacijo preostalih stavkov na kasnejše korake.

Pri specializaciji hipoteze se posamezen stavek specializira na enega izmed naslednjih načinov:

1. Z unifikacijo dveh spremenljivk, npr. A=B.
2. Z zamenjavo spremenljivke z izrazom, npr. L se zamenja z [A|L1].
3. Z dodanim literalom iz predznanja.

Pri tem je treba poudariti naslednje podrobnosti:

1. Vse spremenljivke imajo določen tip. Unificiramo lahko le spremenljivke istega tipa. Ravno tako so pretvorbe spremenljivk v izraze definirane za določene tipe. Pretvorbe definira uporabnik. Tudi klici predznanja zahtevajo in vračajo spremenljivke določenega tipa (kot jih določi uporabnik).
2. Argumenti pri klicih predznanja so lahko definirani kot vhodni ali izhodni. Ko se doda nov literal, se potrebni vhodni argumenti nedeterministično unificirajo z že obstoječimi spremenljivkami ustreznih tipov. Izhodni argumenti pa se dodajo kot nove spremenljivke.

Po razlagi delovanja algoritma (skupaj z ostrilnim operatorjem) ocenimo kompleksnost algoritma:

$$kompleksnost = \sum_{i=1}^{hipoteze} BF(i) * ocenjevanje$$

,

kjer *hipoteze* predstavljajo število izostrenih hipotez, *ocenjevanje* je cena ocenjevanja hipoteze. *BF* je razvejitveni faktor, ki ga lahko izrazimo z

$$BF(i) = \sum_{j=1}^{|hipoteza(i)|} uni(i,j) + izraz(i,j) + pred(i,j)$$

.

uni(i, j) je število unifikacij spremenljivk, ki se lahko izvede v stavku *j*, hipoteze *i*.

$$uni(i,j) = \left|\left\{(a,b) \mid a \in Sprem(i,j) \,\&\, b \in Sprem(i,j) \,\&\, tip(a) = tip(b) \,\&\, a \neq b\right\}\right|,$$

kjer *Sprem(i, j)* predstavlja množico spremenljivk v stavku *j* hipoteze *j*. *tip(a)* pa je tip spremenljivke a.

izraz(i, j) predstavlja število različnih izrazov, s katerimi lahko spremenimo spremenljivko v stavku *j* hipoteze *i*.

$$izraz(i,j) = \left|\left\{(x,a) \mid x \in Izraz \,\&\, a \in Sprem(i,j) \,\&\, tip(x) = tip(a)\right\}\right|,$$

kjer *Izraz* predstavlja množico izrazov, ki jo je definiral uporabnik.

pred(i, j) predstavlja število predikatov, ki jih lahko dodamo stavku *j* v hipotezi *i*:

$$pred(i,j) = \begin{cases} \sum_{p \in Predznanje} \prod_{k=1}^{Mestnost_p^+} \left|\left\{a \mid a \in Sprem(i,j) \,\&\, tip(a) = tip(atribut^+(p,k))\right\}\right| & \text{če } \left|Stavek(i,j)\right| < Lmax \\ \mathbf{0} \text{ sicer} \end{cases},$$

kjer je *Predznanje* množica s strani uporabnika definiranih predikatov predznanja, $Mestnost^+$ je število vhodnih atributov, $tip(atribut^+(p,j))$ pa je tip *j*-tega vhodnega atributa predikata *p*.

ocenjevanje lahko ocenimo kot:

$$ocenjevanje = st\_primerov * povp\_cena\_izrač\_pokrit\_prim,$$

kjer je *st_primerov* število učnih primerov, *povp_cena_izrač_pokrit_prim* pa je povprečna cena za izračun pokritosti enega učnega primera.

Če poenostavimo (predpostavimo, da so vse spremenljivke istega tipa, vsi predikati predznanja imajo enako mestnost ...) lahko kompleksnost izrazimo kot:

$$\begin{aligned} kompleksnost = {} & \left|hipoteze\right| * Cmax * st\_primerov * povp\_cena\_izrač\_pokrit\_prim * \\ & * \left(\left|Sprem\right|^2 + \left|Sprem\right| * \left|Izraz\right| + \left|Predznanje\right| * \left|Sprem\right|^{Mestnost}\right) \end{aligned}.$$

Če še predpostavimo, da je število spremenljivk (v najslabšem primeru) enako produktu največjega dovoljenega števila predikatov in mestnosti teh predikatov, dobimo:

$$\begin{aligned} kompleksnost = {} & \left|hipoteze\right| * Cmax * st\_primerov * povp\_cena\_izrač\_pokrit\_prim * \\ & * (L\max^2 * Mestnost^2 + L\max * Mestnost * \left|Izraz\right| + \\ & + \left|Predznanje\right| * L\max^{Mestnost} * Mestnost^{Mestnost}) \end{aligned}.$$

Kompleksnost je torej več kot eksponentno odvisna od mestnosti predikatov, na drugi strani pa je le linearno odvisna od števila učnih primerov. Vendar to velja le, če se primeri ponavljajo, kajti od števila učnih primerov je pogosto odvisno število izostrenih hipotez, preden pridemo do končne rešitve. Kot se izkaže pri poskusih, se število izostrenih hipotez s povečanjem števila učnih primerov najprej povečuje (ko namesto enostavne, a napačne rešitve, sistem po daljšem iskanju najde pravilno rešitev), ko pa se število učnih primerov še dodatno povečuje, se število izostrenih hipotez zopet zmanjša (ko učni primeri bolje usmerjajo iskanje).

3.1.3 Meta-interpreter

Za določanje pokritosti določene hipoteze HYPER uporablja meta interpreter, ki s pomočjo predznanja, definiranega v prologu, in definicije hipoteze simulira delovanje prologovega interpreterja. Glavna razlika med standardnim prologovim interpreterjem in meta-interpreterjem v sistemu HYPER je, da ima meta-interpreter omejeno dolžino dokaza (največja dolžina dokaza se nastavi s parametrom). Ker pa je meta-interpreter omejen na dolžino dokaza, ne smemo primere, pri katerih bi prešli največjo dovoljeno dolžino, označiti kot nedokazljive. Zavedati se moramo da bi standardni prologov interpreter v takem primeru:

1. Prišel do pozitivnega odgovora ali
2. Zašel v neskončno zanko ali
3. Prišel do negativnega odgovora, se poskusil vračati in najti alternativni odgovor, ki pa bi bil lahko bodisi pozitiven (celo z dokazom krajšim od naše omejitve) bodisi negativen ali, pa bi zašel v neskončno zanko.

Iz tega razloga je potrebno odgovore meta interpreterja, ki so posledica prekoračenja postavljene meje, označiti na poseben način. Zato na klic meta interpreterja sistema HYPER, ki se glasi **prove(Cilj, Hipoteza, Odgovor)** in ki vedno uspe, dobimo odgovor:

- Odgovor = yes, če je Cilj dokazljiv iz Hipoteze in predznanja v največ D korakih (D je v tem primeru največja dovoljena dolžina dokaza).
- Odgovor = maybe, če je bilo iskanje dokaza prekinjeno zaradi presežene največje dovoljene dolžine dokaza.
- Odgovor = no, če so bile vse alternative pri iskanju dokaza izčrpane, še preden smo prišli do dolžine dokaza D. V tem primeru bi bil odgovor enak tudi pri poljubno večjem D.

Treba je pojasniti, kako HYPER reagira na odgovor »maybe«, ko se določa pokritost učnega primera s strani določene hipoteze. HYPER tak odgovor vedno ovrednoti kot najslabšo možnost. Če je učni primer pozitiven, se vzame, da primer ni, pokrit in obratno, če je primer negativen, HYPER reagira, kot da je primer pokrit.

K dolžini dokaza štejejo samo koraki, ki vsebujejo klice stavkov, definiranih v trenutni hipotezi. Klici predznanja, ki je definirano v Prologu, se predajo standardnemu Prologovemu interpreterju in se ne štejejo v dolžino dokaza. Pri predznanju mora tako uporabnik poskrbeti, da je le-to ustrezno definirano in ne privede do neskončne zanke. V primeru, da se mora meta-interpreter pri iskanju dokaza vračati in iskati alternativni dokaz, se koraki, po katerih se je vrnil, ne štejejo v skupno dolžino dokaza. Ko mora meta interpreter dokazati več ciljev, mora biti vsota dolžin dokazov vsakega izmed ciljev manjša od največje dovoljene dolžine dokaza.

3.1.4 Slabosti sistema HYPER

Največja slabost sistema HYPER je ponavljanje procesiranja. Ker običajno isti stavek nastopa v več kandidatnih hipotezah, HYPER pogosto več kot enkrat izračuna naslednike stavka kot del hipoteze. Če k temu dodamo še dejstvo, da lahko do istega stavka pridemo po več poteh ostrenja, posledično lahko obstaja več identičnih hipotez. HYPER pa ne premore nobenega sistema za odpravljanje dvojnikov. Vse to povečuje potrebno procesiranje in že tako velik prostor možnih hipotez na ta način navidezno postane še večji. Poleg tega pa dvojniki tudi nepotrebno zasedajo že tako omejen pomnilnik (SICStus Prolog je omejen na 256 MB).

HYPER poleg tega ne izkoristi informacije, ki je uporabniku pogosto na voljo in jo v nekaterih primerih tudi poda sistemu. Gre za informacijo o tem ali so spremenljivke v glavi iskanega predikata (njegovi parametri) vhodne ali izhodne. Glede na to, da s tem opišemo uporabo iskanega predikata, je razumljivo, da to informacijo uporabnik ima. Še več – kadar je tak predikat definiran kot rekurziven, mora ta podatek uporabnik sistemu navesti, da ga sistem zna klicati. Izkaže pa se, da, če te podatke sistem zna uporabljati, lahko nekatere hipoteze zavrne kot neprimerne in s tem zmanjša velikost preiskanega prostora.

Dodatno dejstvo, ki bi ga HYPER lahko enostavno izkoristil, je, da lahko nasledniki (stavki ali hipoteze) pokrivajo največ primere, ki jih pokriva starševski stavek.

V sistemu so še nekatere manjše slabosti, ki pa so povezane le z implementacijo in nekoliko omejujejo delovanje sistema.

3.2 Izboljšave in modifikacije originalnega algoritma

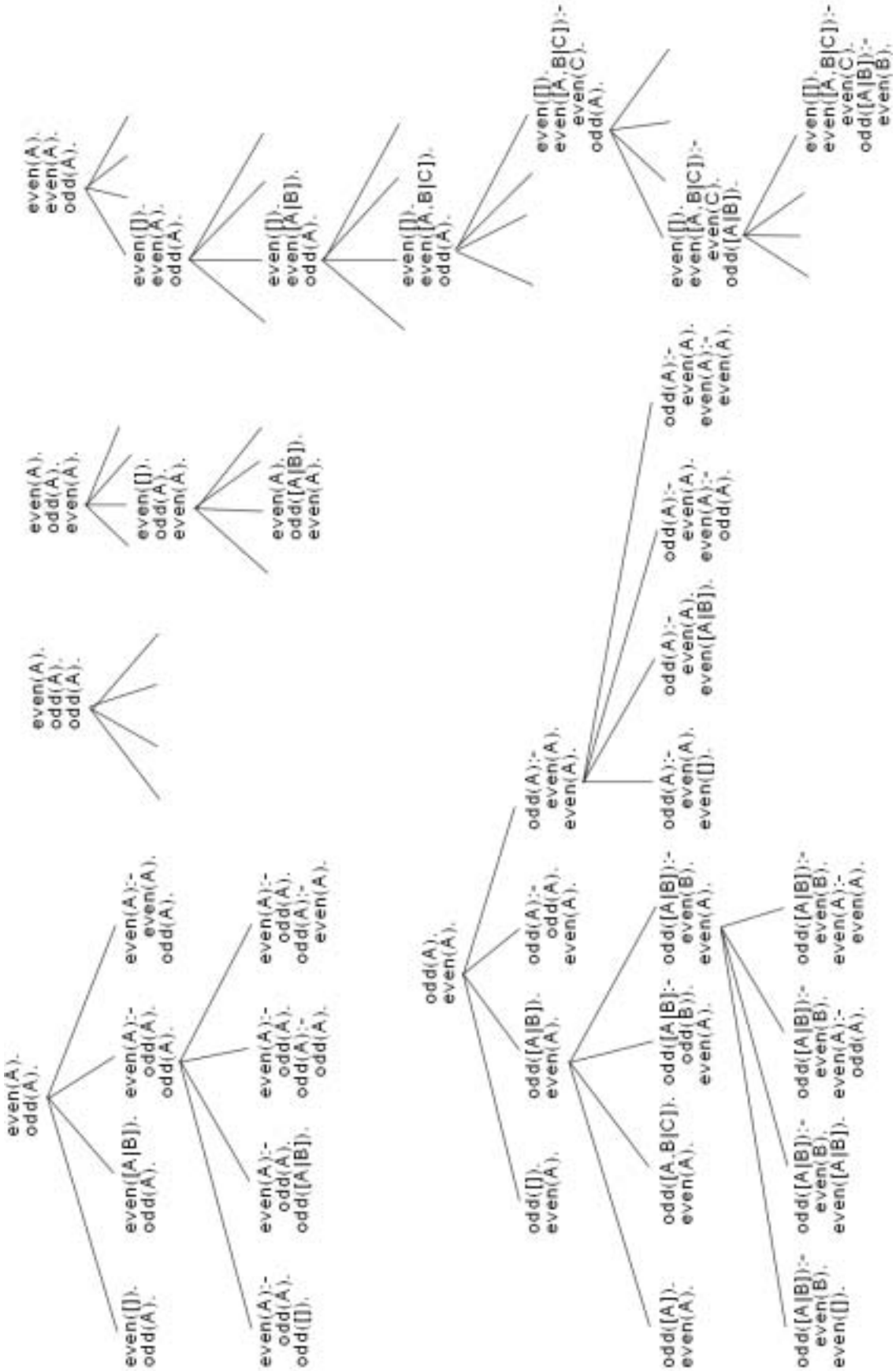
Tako kot originalni algoritem je tudi HYPER^2 pisan v prologu (v narečju SICStus). Algoritem je dolg približno 1000 vrstic kode. Algoritem se usmerja preko dejstev, ki so v bazi prologa. Algoritem (zaradi omejitev SICStus prologa) uporabi do 256MB pomnilnika (v odvisnosti od domene), čas izvajanja pa je običajno od dela sekunde do več ur.

3.2.1 Grajenje gozda stavkov

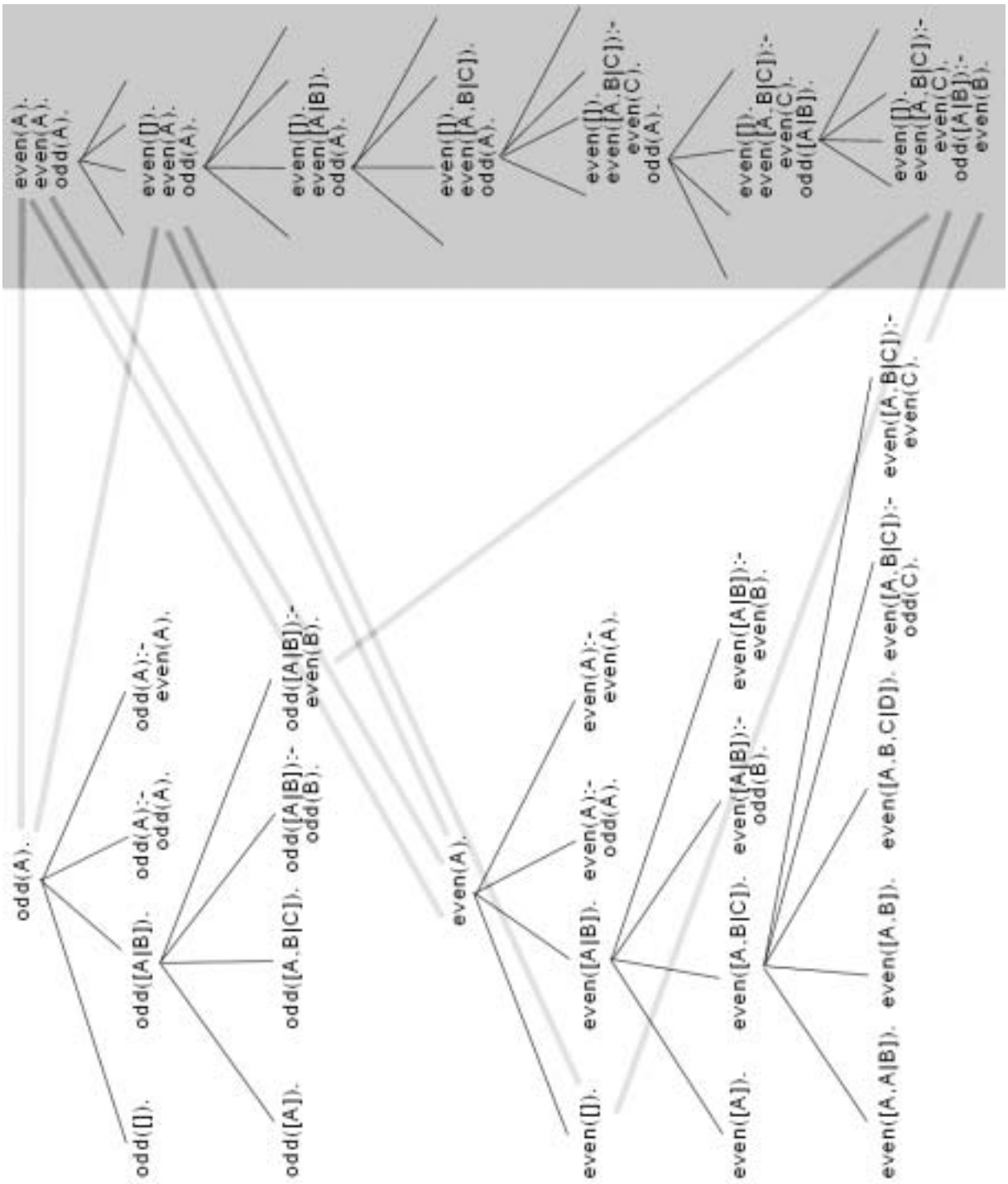
V nasprotju z originalno verzijo sistema HYPER, ki gradi gozd hipotez (Sl. 3.2), izboljšani algoritem HYPER^2 gradi gozd stavkov (Sl. 3.3). Same hipoteze pa so sestavljene iz kazalcev na stavke. Posledično je pri izboljšanem algoritmu en stavek lahko del več hipotez. Tako odpravimo nekatero ponovno procesiranje in ponavljanje informacij v pomnilniku.

Gradnja gozda stavkov in dejstvo, da isti stavek lahko nastopa v več hipotezah, nam prinese naslednje prednosti:

- **Stavki poznajo svoje naslednike.** Ker isti stavek nastopa v več hipotezah, nam stavkov, ki so bili v okviru katere druge hipoteze že specializirani, ni potrebno ponovno specializirati. V takem primeru samo uporabimo naslednike, ki so se izračunali ob prvi specializacije stavka, vključno s primeri, ki jih pokrivajo, njihovo ceno in podobno.
- **Preverjanje podvojenosti stavkov.** Ko specializiramo stavek in generiramo njegove naslednike, se lahko zgodi, da po več različnih vejah pridemo do istega stavka. Zato vsakič preverimo, ali naslednik že obstaja (kot potomec katerega drugega stavka). V tem primeru namesto nove kopije uporabimo original. Torej v resnici ne gradimo specializacijskega drevesa stavkov, ampak gradimo graf (Sl. 3.4). Posledično je že pri tej generaciji hipotez kompleksnost manjša, še večje prednosti pa prinesejo naslednje generacije, ko je potrebno namesto več različnih stavkov specializirati le enega (posledično je potrebno specializirati manj naslednikov v kasnejših generacijah)**.**



Sl. 3.2 Gozd hipotez, kot ga generira originalni algoritem sistema HYPER. (V drevesih, ki so nakazana, so izpisane samo hipoteze, ki so bile specializirane.)



Sl. 3.3 Gozd stavkov kot ga gradi izboljšani algoritem HYPER. No osenčenem delu je viden simuliran razvoj hipoteze. Hipoteze so realno predstavljene le kot kazalci na stavke (vidni kot sive črte)

Samo preverjanje uporabi predstavitev samih stavkov v sistemu. Le-ti so predstavljeni kot dejstva v prologu. Zaradi tega je enostavno (s pomočjo prologa) najti stavke, ki se lahko unificirajo z našim novim stavkom. Stavke, ki se lahko unificirajo, nato še (s pomočjo zamenjave vrednosti spremenljivk s konstantami) preverimo, ali so identični novemu stavku. Poraba časa pri preverjanju podvojenosti stavkov je v najslabšem

primeru sorazmerna s številom prej generiranih stavkov, vendar je običajno potrebno preveriti le nekaj stavkov. Zahtevnost iskanja kandidatov pa je odvisna od učinkovitosti implementacije interpreterja SICStus prologa, ki jo je težko oceniti.

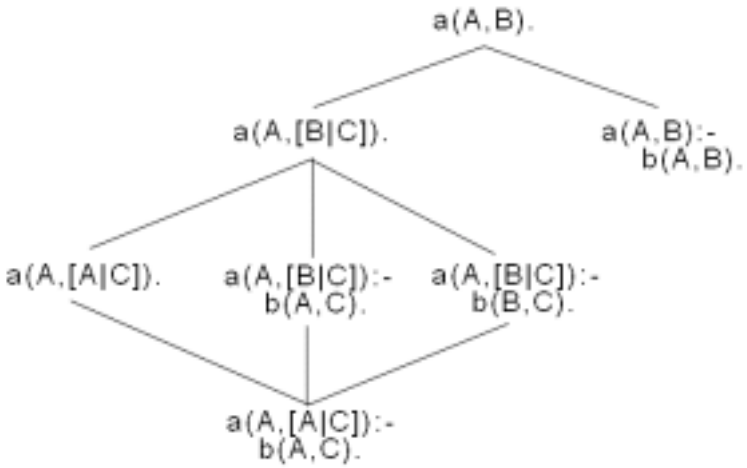
- **Enostavno preverjanje podvojenosti hipotez.** Glede na to, da so hipoteze predstavljene kot skupek kazalcev (v obliki seznama številčnih oznak stavkov) na stavke v specializacijskem grafu, in so tako enostavna struktura, je enostavno preverjati, če smo na nov način generirali dvojnik kake hipoteze (pogledamo, ali v vrsti za specializacijo ali na seznamu že specializiranih hipotez obstaja enaka hipoteza), ker preverimo le, ali določeno dejstvo že obstaja, je učinkovitost odvisna od učinkovitosti implementacije interpreterja in jo je težko oceniti. Ker nimamo dvojnikov hipotez, nam ni potrebno v naslednjih generacijah večkrat specializirati enakih hipotez. Pri tem je treba omeniti, da HYPER hipoteze, ki vsebujejo iste stavke v različnem vrstnem redu, smatra kot različne. Zaradi omejitve dolžine dokaza pokritosti primerov se lahko zgodi, da hipoteza A primer »mogoče« pokriva, ker preseže omejitev dolžine dokaza, preden se meta-interpreter začne vračati, hipoteza B, ki je samo permutirana hipoteza A, pa tak primer pokrije. (Zagotovo ga vsaj mogoče pokrije).

Gradnja gozda stavkov in odprava podvojenega procesiranja sicer ne spremenita enačbe kompleksnosti, ki velja za HYPER, vendar zmanjša število izostrenih hipotez.Izkazalo se je, da HYPER generira in oceni v povprečju 98% (minimalno 92%, maksimalno 99%) originalnih hipotez (ostale so kopije), na domeni member in v povprečju 75% (minimalno 25%, maksimalno 99%) na domeni path. HYPER^2 generira v povprečju 99%.(min 92%, max 100%) na domeni member ter 95% (min 75%, max 100%) originalnih hipotez na domeni path, vendar pa se kopije zaznajo in se ne razvijajo naprej. Sistem HYPER izostri 100% originalnih hipotez na domeni member in v povprečju 72% (min 21%, max 100%). Torej s preprečevanjem podvojenega procesiranja hipotez zmanjšamo procesiranje tudi do 75% (pri bolj kompleksnih domenah lahko tudi več).

3.2.2 Stavki ali hipoteze poznajo primere, ki jih pokrivajo

Glede na to, da HYPER^2 gradi gozd stavkov, je bilo skoraj samoumevno, da se pokritost primerov s strani obdelovane hipoteze računa na nivoju stavkov ter se tudi zapomni kot lastnost stavka. To je sicer možno za vse stavke, vendar je pokritost primerov s strani rekurzivnih stavkov (rekurzivnih v razširjenem pomenu - stavkov, ki kličejo sebe in/ali druge

dele hipoteze) odvisna od preostanka hipoteze in bi si bilo potrebno v tem primeru zapomniti pokritost za kombinacijo stavek-hipoteza, kar pa bi prineslo prevelike zahteve po omejeni količini pomnilnika (zato smo v verziji, kjer si za vsak stavek zapomnimo, katere primere pokriva, prisiljeni, da pri računanju pokritosti rekurzivnih stavkov upoštevamo vse učne primere). Pri tem se pokritost primerov s strani (nerekurzivnega) stavka računa, kot da je ta stavek edini v hipotezi. Preverimo tudi, katere primere pokriva taka hipoteza. Te primere si zapomnimo kot primere, ki jih stavek pokriva. Hipoteza pa pokriva primere, ki jih pokriva vsaj en izmed stavkov hipoteze (pri rekurzivnih hipotezah je potrebno pokritost kompletno preračunati po vsakem ostrilnem koraku). Poznavanje pokritosti učnih primerov prinese več prednosti.



Sl. 3.4 Ostrilni graf prepreči podvajanje stavkov, ki bi nastalo v ostrilnem drevesu

Prva je, da se pri računanju pokritosti naslednikov stavkov lahko osredotočimo le na primere, ki jih je pokrival starševski stavek (povsod pri nerekurzivnih stavkih oziroma v okviru starševske hipoteze pri rekurzivnih stavkih). Pri rekurzivnih hipotezah je potrebno ponovno izračunati pokrivanje rekurzivnih stavkov, tudi če se v okviru hipoteze specializira kateri drug stavek. Takrat velja, da je pokritost tega stavka v okviru specializirane hipoteze podmnožica pokritosti istega stavka v okviru starševske hipoteze. V obeh primerih pa velja, da so primeri, ki jih pokrivajo nasledniki stavka, podmnožica primerov, ki jih pokriva originalni stavek.

Druga prednost, od katere je bilo pričakovano največje izboljšanje, je dejstvo, da je, če stavek nastopa v več hipotezah, potrebno pokritost izračunati le enkrat in ne za vsako hipotezo, v kateri se ta stavek pojavi.

Tretja prednost pa se pojavi pri preverjanju kompletnosti hipotez. Ker poznamo pokrivanje posameznih stavkov, je enostavno preveriti, ali je unija množic prekrivanja pozitivnih primerov posameznih stavkov (v okviru hipoteze pri rekurzivnih stavkih) v hipotezi enaka množici vseh pozitivnih primerov.

Vendar se je izkazalo, da tak pristop prinaša tudi dodatne slabosti. Ker se izračuna pokritost za vsak stavek posebej (čeprav za vsak stavek samo enkrat, ne glede na to, v kolikih hipotezah nastopa), se pogosto zgodi, da je določen primer pokrit s strani več stavkov v hipotezi in tako prevečkrat izračunamo pokritost tega primera. Najprej smo mislili, da prednost, ko pokritost določenega stavka računano samo enkrat, odtehta to pokrivanje več stavkov v okviru ene hipoteze, vendar se je izkazalo, da se pogosto stavki premalokrat uporabijo, da bi to prineslo pričakovano prednost. Poleg tega pa smo že povedali, da tak pristop ni ravno najboljši za rekurzivne hipoteze. Zato smo uvedli tudi verzijo, ki si za vsako hipotezo - in ne stavek – zapomni, katere primere pokriva.

V verziji, kjer si stavki ne zapomnijo primerov, ki jih pokrivajo (zaradi slabosti, ki jih tak pristop lahko prinese), smo vsaki hipotezi dodali informacijo o primerih, ki jih pokriva. Pravzaprav si zapomnimo samo, katere negativne primere pokriva hipoteza, kajti vsaka hipoteza mora pokrivati vse pozitivne primere (pokritost pozitivnih primerov preverimo, če za hipotezo ne vemo ali je kompletna). Poleg informacije o tem, katere negativne primere pokriva hipoteza, si zapomnimo tudi, kateri stavek v hipotezi pokriva posamezni negativni primer (oziroma s klicem katerega stavka smo primer pokrili – ne glede na to, ali je stavek klical katere druge dele hipoteze). S tem občutno olajšamo izbiro stavka za nadaljnje ostrenje. Ko hipoteze poznajo primere, ki jih pokrivajo lahko izkoristimo dejstvo, da vsaka hipoteza naslednica trenutne hipoteze pokriva največ primere, ki jih pokriva trenutna hipoteza. Posledično lahko pri računanju pokritosti s strani hipotez naslednic zmanjšamo število klicev meta-interpreterja in tudi čas izvajanja. To izboljšavo lahko v nasprotju z metodo, kjer si stavki zapomnijo primere, ki jih pokrivajo, uporabimo tudi na rekurzivnih hipotezah.

S tem, ko si sistem zapomni, katere primere pokriva posamezni stavek oziroma hipoteza, lahko zamenjamo v enačbi kompleksnosti *st_primerov*, ki predstavlja število primerov, s povprečnim številom primerov, ki ga pokriva posamezni stavek ali hipoteza. Le to se z razvijanjem počasi manjša do 1/Cmax pozitivnih učnih primerov, pri verzijah kjer je pokritost vezana na stavke (vendar to velja le za nerekurzivne domene), oziroma le do števila pozitivnih učnih primerov pri hipotezah. Bolj natančno je pohitritev nemogoče določiti, ker je odvisna, ne le od domene, ampak tudi od posameznih učnih problemov. Pravzaprav se, ko se pokritost

računa glede na stavke, lahko zgodi, da sploh ni potrebno računati nobenih pokritosti primerov (ko je bil stavek že prej generiran v okviru druge hipoteze) in smo že izračunali katere primere pokriva.

3.2.3 Izračunavanje kompletnosti hipotez

HYPER^2 si zapomni hipoteze, ki so bile dokazano kompletne. Ko se preverja kompletnost nove hipoteze, se najprej preveri, ali nova hipoteza kot podmnožico vsebuje kakšno drugo kompletno hipotezo. V tem primeru se tudi nova hipoteza označi kot kompletna. Za hipoteze v čistem prologu je to tudi vedno res. Vendar je možno pri dodajanju negacij, rezov (ne enega ne drugega HYPER ne uporablja), da nadmnožica kompletne hipoteze ni kompletna. Podoben efekt lahko v sistemu HYPER nastane zaradi omejene dolžine dokaza pri računanju pokritosti. Kot je bilo že rečeno pri razlagi enostavnega preverjanja podvojenosti hipotez, lahko vrstni red stavkov vpliva na pokritost primerov s strani hipoteze. Ker pa so vse hipoteze, ki so poznano kompletne, nekonsistentne (konsistentne hipoteze namreč še nismo našli), bo potrebno vsaj enega izmed stavkov v okviru kompletne podmnožice še specializirati. Po specializaciji nova hipoteza ne bo vsebovala več kompletne podmnožice in bo potrebno kompletnost dokazati na običajen način.

Če hipoteza vsebuje kot podmnožico hipotezo, ki je že sama kompletna, nam ni potrebno preverjati ali hipoteza pokriva pozitivne učne primere. Pri varianti, ko računamo pokritost glede na stavke, nam to ne prinese veliko izboljšanja, odpade samo preverjanje ali je unija po stavkih vseh množic pokritij pozitivnih učnih primerov enaka množici vseh pozitivnih učnih primerov. Ko računamo pokritost glede na hipoteze, lahko (če je hipoteza nadmnožica kompletni hipotezi) spustimo preverjanje ali hipoteza pokriva pozitivne učne primere in preverimo samo, katere negativne učne primere (od tistih, ki jih je pokrivala staševska hipoteza) pokriva naša hipoteza. Ker je stopnja odvisna od domene in od učnih primerov (ter poti, po kateri preiskovanje poteka skozi prostor možnih hipotez) je nemogoče izboljšanje prikazati v poenostavljeni formuli kompleksnosti.

3.2.4 Iskanje s snopom

Med izboljšavami algoritma HYPER je bila preizkušena tudi varianta, ki uporablja iskanje s snopom namesto iskanja najprej najboljši. Uporabljeno je bilo preiskovanje s snopom po algoritmu opisanem v [15]. Poenostavljen osnovni algoritem se vidi na Sl. 3.5. Obstajajo različne verzije iskanja s snopom. V [18] je opisana verzija, ki novo množico hipotez sestavi

samo iz naslednikov in odvrže stare hipoteze, ustavi se pa, ko je množica naslednikov prazna. Te verzije nismo izbrali, ker nam vključitev starih hipotez zagotavlja dodaten ustavitveni pogoj – če so vse nove hipoteze slabše od najslabše od starih hipotez, se ustavimo. S tem prisilimo preiskovanje prostora hipotez, da se premika proti boljšim hipotezam. Širina snopa zagotavlja, da so med hipotezami, ki jih bomo specializirali, tudi malo slabše ocenjene hipoteze, če so le vsaj enako dobre kot najslabše hipoteze v prejšnjem snopu. Na ta način hipoteze ne zavržemo, če se glede na oceno ne izboljša občutno (ali pa celo malo poslabša), kljub temu pa zagotavljamo izboljšavo (oziroma vsaj ohranitev) povprečne ocene hipotez v snopu. Vendar je treba povedati, da iskanje s snopom ni izčrpno iskanje in lahko spregleda pravilne hipoteze, ki jih je zavrgel, ker so naslednice preslabo ocenjenih hipotez. Širina snopa je nastavljiva s parametrom.

Postavi začetne hipoteze z 1 do največ $\mathbf{C_{MAX}}$ stavki, jih oceni in vstavi v množico hipotez $\boldsymbol{H_{NS}}$
Dokler nisi našel konsistentne hipoteze in $\boldsymbol{H_{NS}}$ ni enaka $\boldsymbol{H_{NS}}$,
    $\boldsymbol{H_{NS}}$, = $\boldsymbol{H_{NS}}$
    Inicializiraj množico naslednikov hipotez $\boldsymbol{H_{Next}}$ ={}
    Za vsako hipotezo **H**, **H** ∈ $\boldsymbol{H_{NS}}$
        Inicializiraj množico naslednikov hipoteze **H** $\boldsymbol{H_N}$ ={}
        Izberi stavek **S** iz hipoteze **H**
            Specializiraj stavek **S** in dobi naslednike $\boldsymbol{S'}$
            Dobi množico naslednikov **H** $\boldsymbol{H_{N'}}$, tako da v **H** zamenjaš stavek **S** s stavkom **S'**∈$\boldsymbol{S'}$
            $\boldsymbol{H_N}$:= $\boldsymbol{H_{N'}} \cup \boldsymbol{H_N}$
            Ponovi izbiro stavka
        Izloči nekompletne hipoteze iz $\boldsymbol{H_N}$, ter oceni ostale hipoteze
        $\boldsymbol{H_{Next}}$:= $\boldsymbol{H_N} \cup \boldsymbol{H_{Next}}$
        Vzemi drugo hipotezo iz $\boldsymbol{H_{NS}}$
    $\boldsymbol{H_{NS}}$ = $\boldsymbol{H_{NS}} \cup \boldsymbol{H_{Next}}$
    Uredi $\boldsymbol{H_{NS}}$
    Če je potrebno skrajšaj $\boldsymbol{H_{NS}}$ na največjo dovoljeno velikost snopa
    Ponovi

Sl. 3.5 Iskanje s snopom v okviru sistema HYPER

3.2.5 Podatek o izhodnih spremenljivkah

Pri analiziranju delovanja originalnega algoritma HYPER je bil opažen problem pri rekurzivnih klicih. Izkaže se, da bi se lahko podoben problem pojavil tudi pri uporabi

nerekurzivnih hipotez, ki jih je vrnil HYPER. Originalni algoritem predpostavlja, da imajo vsi argumenti v glavi stavkov hipoteze poznane vrednosti. Ko obdelujemo primere, je to seveda res, vendar če je dovoljena rekurzija in so pri tem določeni izhodni argumenti, le-ti pri rekurzivnem klicu nimajo vedno poznane vrednosti, kar pomeni, da imamo argumente, za katere algoritem predpostavlja, da imajo poznane vrednosti in jih lahko uporablja kot vhodne argumente pri klicih predznanja. Podobno lahko velja tudi pri nerekurzivnih hipotezah, vendar le ko se le-te uporabijo na novih podatkih – ko uporabimo hipotezo oziroma vrnjen predikat in želimo dobiti rezultat pri klicu predikata, uporabimo neinicializirane argumente, za katere pa je HYPER pri gradnji predpostavil, da so inicializirani. V takih primerih lahko pride do neskončne zanke ob klicu predznanja, ker je bilo to uporabljeno na napačen način. Pri uporabi izboljšanega algoritma tako pri definiciji začetnih stavkov v hipotezah definiramo tudi izhodne spremenljivke. Le-te se označijo tako, da nimajo poznanih vrednosti. Pri specializaciji je nato prepovedano uporabiti spremenljivko brez poznane vrednosti kot vhodni argument pri klicu predznanja (ali rekurzivnem klicu). Dodatna sprememba, ki je posledica informacije o nepoznanih vrednostih v stavkih hipotez, je, da pri specializaciji s klicem predznanja lahko spremenljivke, ki nimajo znane vrednosti, uporabimo kot izhodne argumente klica. Originalni algoritem za izhodne argumente klica vedno uporabi nove spremenljivke. Pri tej izboljšavi uporabimo ali nove spremenljivke ali stare, ki še nimajo poznane vrednosti in jim s tem določimo vrednost.

Poglejmo, kako se ostrijo stavki (v okviru hipoteze), če to informacijo uporabljamo, in kako, če jo ne. Najprej si poglejmo poenostavljeno kodo za unifikacijo spremenljivk:

```
%refine(ClauseNo,Body,Vars,NewClauseNo)
   %ClauseNo=oznaka stavka,
   %Body=telo stavka,
   %Vars=seznam spremenljivk,
   %NewClauseNo=oznaka novega stavka

refine(ClauseNo,Body,Vars,NewClauseNo):-        %originalni predikat
   append(Vars1,[V|Vars2],Vars),                %izberi spremenljivko
   member(V,Vars2),                             %jo izenačimo z drugo spremenljivko
   append(Vars1,Vars2,NewVars),                 %sestavimo nov seznam spremenljivk
   remove_duplicates(Body,Body),
   add1(clause,NewClauseNo),
   assertz(clause(NewClauseNo,Body,NewVars)).   % vstavimo nov stavek v bazo


%refine(ClauseNo,Body,InVars/OutVars,NewClauseNo)
   %ClauseNo=oznaka stavka,
   %Body=telo stavka,
   %InVars=seznam inicializiranih spremenljivk
   %OutVars=seznam neinicializiranih spremenljivk
   %NewClauseNo=oznaka novega stavka
```

```
refine(ClauseNo,Body,InVars/OutVars,NewClauseNo):-     %spremenjeni predikat
   (
   append(Vars1,[V|Vars2],InVars),                     %izenači dve inicializirani spremenljivki
   member(V,Vars2),
   append(Vars1,Vars2,NewInVars),
   NewOutVars=OutVars
   ;
   append(Vars1,[V|Vars2],OutVars),                    %izenači dve neinicializirani spremenljivki
   member(V,Vars2),
   append(Vars1,Vars2,NewOutVars),
   NewInVars=InVars
   ;
   append(Vars1,[V|Vars2],OutVars),                    %izenači inicializirano in neinicializirano
   member(V,InVars),                                   %spremenljivko
   append(Vars1,Vars2,NewOutVars),
   NewInVars=InVars
   ),
   remove_duplicates(Body,Body),
   add1(clause,NewClauseNo),
   assertz(clause(NewClauseNo,Body,NewInVars/NewOutVars)).
```

Pri prvem stavku procedure refined, ko ne upoštevamo informacije, enostavno izberemo spremenljivko in jo (v okviru klica member predikata) unificiramo z drugo spremenljivko. Glede na to, da spremenljivke s seboj nosijo informacijo o tipu, se lahko unificirajo samo spremenljivke istega tipa. (Glede na to, da bi sedaj imeli dve kopiji iste spremenljivke, eno zbrišemo iz seznama). V drugem predikatu pa razlikujemo med vhodnimi (oziroma takimi s poznano vrednostjo) in izhodnimi spremenljivkami (oziroma takimi, kjer še ne poznamo vrednosti). V tem primeru se lahko unificirata dve s poznano vrednostjo (lahko zbrišemo poljubno kopijo spremenljivk s seznama), dve z nepoznano vrednostjo (lahko zbrišemo poljubno kopijo spremenljivk s seznama), ali pa ena s poznano in ena z nepoznano vrednostjo (ki ji s tem določimo vrednost, zato zbrišemo kopijo, ki je na seznamu spremenljivk z nepoznano vrednostjo).

Sprememba spremenljivke v izraz je bolj enostavna:

```
%refine(ClauseNo,Body,Vars,NewClauseNo)
   %ClauseNo=oznaka stavka,
   %Body=telo stavka,
   %Vars=seznam spremenljivk,
   %NewClauseNo=oznaka novega stavka
```

```
refine(ClauseNo,Body,Vars,NewClauseNo):-          %originalni predikat
   append(Front,[Var:Type|Back],Vars),            %izberi spremenljivko
   append(Front,Back,Vars1),                      %zgradi seznam preostalih spremenljivk
   term(Type,Var,TVars),                          %spremeni spremenljivko v izraz
   remove_duplicates(Body,Body),                  %prepovedani so dvojni enaki klici
   append(Vars1,TVars,NewVars),                   %zgradi nov seznam spremenljivk
   add1(clause,NewClauseNo),
   assertz(clause(NewClauseNo,Body,NewVars)).     %dodaj nov stavke v bazo


%refine(ClauseNo,Body,InVars/OutVars,NewClauseNo)
   %ClauseNo=oznaka stavka,
   %Body=telo stavka,
   %InVars=seznam inicializiranih spremenljivk
   %OutVars=seznam neinicializiranih spremenljivk
   %NewClauseNo=oznaka novega stavka

refine(ClauseNo,Body,InVars/OutVars,NewClauseNo):-   %spremenjeni predikat
   (
        append(Front,[Var:Type|Back],InVars),        %izberi inicializirano spremenljivko
        append(Front,Back,NewInVars1),
        term(Type,Var,TVars),                        %spremeni jo v izraz
        append(NewInVars1,TVars,NewInVars),
        NewOutVars=OutVars
   ;
        append(Front,[Var:Type|Back],OutVars),       %izberi neinicializirano spremenljivko
        append(Front,Back,NewOutVars1),
        term(Type,Var,TVars),                        %spremeni jo v seznam
        append(NewOutVars1,TVars,NewOutVars),
        NewInVars=InVars
   ),
   remove_duplicates(Body,Body),
   add1(clause,NewClauseNo),
   assertz(clause(NewClauseNo,Body,NewInVars/NewOutVars)).
```

Če ne upoštevamo te informacije, samo izberemo spremenljivko, ki jo lahko pretvorimo v izraz (pretvorba je definirana s predikatom term(Tip, Spremenljivka_pri_klicu oziroma izraz_pri_definiciji, Nove_spremenljivke)). Staro spremenljivko zbrišemo s seznama spremenljivk in dodamo nove (če so). Če upoštevamo informacijo, lahko delamo enako, samo s to spremembo, da imajo vse nove spremenljivke poznano vrednost, če jo je imela prvotna spremenljivka oziroma nepoznano, če je bila taka prvotna spremenljivka.

Dodajanje literala:

```
%refine(ClauseNo,Body,Vars,NewClauseNo)
   %ClauseNo=oznaka stavka,
   %Body=telo stavka,
   %Vars=seznam spremenljivk,
   %NewClauseNo=oznaka novega stavka
```

```
refine(ClauseNo,Body,Vars,Size,norec,NewClauseNo):-
   backliteral(Lit,InArg,RestArg),                    %izberi predikat predznanja
   append(Body,[Lit],NewBody),                        %dodaj ga na konec telesa
   connect_inputs(Vars,InArg),                        %poveži vhodne argumente s
                                                      %spremenljivkami
   remove_duplicates(NewBody,NewBody),                %prepovej podvojene klice
   append(Vars,RestArg,NewVars),                      %dodaj izhodne argumente (nove
                                                      %spremenljivke) seznamu spremenljivk
   add1(clause,NewClauseNo),
   assertz(clause(NewClauseNo,NewBody,NewVars)).      %dodaj nov stavek v bazo


%refine(ClauseNo,Body,InVars/OutVars,NewClauseNo)
   %ClauseNo=oznaka stavka,
   %Body=telo stavka,
   %InVars=seznam inicializiranih spremenljivk
   %OutVars=seznam neinicializiranih spremenljivk
   %NewClauseNo=oznaka novega stavka

refine(ClauseNo,Body,InVars/OutVars,Size,norec,NewClauseNo):-   %špremenjeni predikat
   backliteral(Lit,InArg,RestArg),                    %izberi predikat predznanja
   append(Body,[Lit],NewBody),
   connect_inputs(InVars,InArg),
   connect_outputs(OutVars,RestArg,NewOutVars),       %poveži izhodne argumente z
                                                      %neinicializiranimi spremenljivkami
                                                      %ali pa uporabi nove spremenljivke
   append(InVars,RestArg,NewInVars),
   remove_duplicates(NewBody,NewBody),
   add1(clause,NewClauseNo),
   assertz(clause(NewClauseNo,NewBody,NewInVars/NewOutVars)).
```

V prvem primeru si izberemo literal, ki ga bomo dodali, poiščemo spremenljivke primernih tipov za vhod in dodamo nove spremenljivke, ki so izhod dodanega literala. Prav izbira spremenljivk, ki so vhod literalu, lahko prinese probleme, če le-te še nimajo poznane vrednosti, ko bi jo morale imeti. Na tak način že preprost predikat, kot je member, vrne neskončno rešitev. Če pa vemo, ali imajo spremenljivke poznano vrednost ali ne, vhodne argumente enostavno izberemo med spremenljivkami, katerih vrednost je poznana. Prav tako lahko sedaj dodamo izhodne argumente literala med spremenljivke s poznano vrednostjo ali jih celo unificiramo s spremenljivkami z nepoznano vrednostjo (le-tem na ta način določimo vrednost) oziroma naredimo poljubno kombinacijo. Na ta način dajemo malo prednosti stavkom, ki imajo več spremenljivk s poznanimi vrednostmi.

Če uporabimo podatke o izhodnih spremenljivkah, se lahko (v odvisnosti od domene) nekoliko zmanjša preiskovani prostor hipotez (brez, da bi se spremenila poenostavljena enačba o velikosti preiskovanega prostora). Prav tako se nekoliko spremeni kompleksnost algoritma. Spremeni se člen, ki določa število novih predikatov, ki jih lahko dodamo stavku. Nova formula postane:

$$pred(i,j) = \begin{cases} \sum_{p \in Predznanje} * \prod_{k=1}^{Mestnost_p^+} \left| \{a \mid a \in Sprem^+(i,j) \,\&\, tip(a) = tip(atribut^+(p,k))\} \right| * \\ \qquad * \prod_{k=1}^{Mestnost_p^-} \left(\left| \{a \mid a \in Sprem^-(i,j) \,\&\, tip(a) = tip(atribut^-(p,k))\} \right| + 1 \right) & \text{če } |Stavek(i,j)| < Lmax \\ 0 \text{ sicer} \end{cases}$$

kjer je $Sprem^+$ množica inicializiranih spremenljivk, $Sprem^-$ množica neinicializiranih spremenljivk in $Mestnost^+$ število vhodnih argumentov predikata p, ter $Mestnost^-$ število izhodnih argumentov predikata p. Sicer se zmanjša število sprejemljivih spremenljivk za vhodne argumente, vendar se poveča število izbir za izhodne argumente (do sedaj neinicializirane spremenljivke in še možna nova spremenljivka). Torej se kompleksnost poveča, vendar se na ta način iskanje po prostoru bolje usmerja, tako da se običajno preišče manj hipotez, kar pa zmanjša kompleksnost. Kljub temu pa poenostavljena enačba za kompleksnost ostaja enaka.

3.2.6 Druge manjše modifikacije

Poleg ostalih večjih modifikacij se pri HYPER^2 pojavijo tudi nekatere manjše spremembe, ki samo preprečijo ponavljanje nekaterih manjših izračunov. Tako stavki poznajo svojo velikost; velikost naslednikov se izračuna iz velikosti stavka, ki smo ga ostrili. Drug tak podatek je informacija o rekurzivnosti stavka, ki je potrebna pri verziji, kjer si stavki zapomnijo, katere primere pokrivajo, saj je pri tej verziji potrebno rekurzivne stavke drugače obravnavati.

4 Primerjalno testiranje novih mehanizmov in izboljšav

Primerjalno testiranje novih mehanizmov in izboljšav smo izvedli z meritvami:

- uspešnosti (ali je rešitev pravilna)
- časa reševanja
- števila izostrenih hipotez
- števila klicev meta-interpreterja

Pri tem smo upoštevali, da je rešitev pravilna, če pokrije vse pozitivne učne primere v domeni in ne pokrije nobenega negativnega primera, ne glede na obliko vrnjene hipoteze (preverjanje je bilo opravljeno s pomočjo enostavnega programa). Klic meta-interpreterja je bil štet vsakič, ko smo z njegovo pomočjo oskušali ugotoviti, ali hipoteza (oziroma klavzula) pokriva določen (pozitivni ali negativni) učni primer.

Poskuse smo izvajali na dveh domenah:

- Domena member, ki je opisovala iskanje elementov v seznamu. Seznam je bil definiran kot vhodni argument, element kot izhodni argument. Seznami so dolžine pet, brez ponavljajočih se elementov. Domena ima 1.305 pozitivnih in 325 negativnih primerov. Učni primeri so bili generirani s pomočjo predikata na Sl. 4.1. Učni primeri so bili izbrani naključno, brez ponavljanja. Poskusi so bili narejeni v devetih serijah opisanih v tab. 4.1.

```
member(A,[A|_]).
member(A,[_|B]):-member(A,B).
```

Sl. 4.1 Definicija predikata member, uporabljana za generiranje učnih primerov.

Pri tem smo vsem verzijam sistema podali naslednje definicije in parametre:

- Začetni stavki:

  start_clause([member(X,L)] / [X:item, L:list]).

  start_clause([member(X,L)] / [L:list],[X:item]).

  Prvi stavek je namenjen verzijam, ki ne uporabljajo informacije o tem, ali so spremenljivke v glavi iskanega predikata vhodne ali izhodne (pomen:

backliteral([začetni stavek]/[deklaracije spremenljivk]), drugi pa je namenjen verzijam, ki to informacijo uporabljajo (pomen: backliteral([začetni stavek]/[deklaracije vhodnih spremenljivk], [deklaracije izhodnih spremenljivk]).

- Definicije in deklaracije predznanja:

  backliteral(member(X,L), [L:list], [X:item]).

  Drugače rečeno, hipoteze lahko uporabljajo rekurzivne klice.

- Pretvorbe spremenljivk v izraze:

  term(list, [X|L], [X:item, L:list]).

  term(list, [], []).

  Spremenljivke tipa list se lahko pretvorijo v seznam s spremenljivko tipa item v glavi in spremenljivko tipa list kot repom, prav tako pa se lahko pretvorijo v prazen seznam.

- Največje število stavkov v hipotezi je bilo nastavljeno na tri.

- Največje število sprecializiranih hipotez je bilo nastavljeno na 700.

Oznaka serije	Št. poskusov v seriji	Št. pozitivnih primerov	Št. negativnih primerov
1 (002 007)	100	3 (0,2%)	3 (0,7%)
2 (005 01)	100	7 (0,5%)	4 (1%)
3 (01 01)	100	13 (1%)	4 (1%)
4 (05 05)	75	66 (5%)	17 (5%)
5 (10 10)	50	131 (10%)	33 (10%)
6 (20 20)	30	261 (20%)	65 (20%)
7 (50 50)	20	653 (50%)	163 (50%)
8 (70 70)	10	914 (70%)	228 (70%)
9 (100 100)	1	1305 (100%)	325 (100%)

tab. 4.1 Opis posameznih serij domene member

- Domena path (učni primeri so bili generirani s pomočjo predikata na Sl. 4.2), ki je opisovala iskanje poti po preprostem usmerjenem grafu (Sl. 4.3). Graf ima pet elementov in pet usmerjenih povezav. Domena ima 14 pozitivnih primerov in 5.136 negativnih primerov. Poskusi so bili narejeni v devetih serijah opisanih v tab. 4.2. Učni primeri so bili izbrani naključno, brez ponavljanja. Poskusi so bili narejeni v 9 serijah opisanih v tab. 4.1.

```
path(A,B[A,B]).
path(A,B[A,C|D]):-link(A,C), path(C,B,D).
```

Sl. 4.2 Definicija predikata path, uporabljana za generiranje učnih primerov.

Pri tem smo vsem verzijam sistema podali naslednje definicije in parametre:

- Začetni stavki:

  start_clause([path(X,X1,L)] / [X:item, X1:item, L:list]).

  start_clause([path(X,X1,L)] / [X:item, X1:item],[L:list]).

  Prvi stavek je namenjen verzijam, ki ne uporabljajo informacije o tem, ali so spremenljivke v glavi iskanega predikata vhodne ali izhodne (pomen: backliteral([začetni stavek]/[deklaracije spremenljivk]), drugi pa je namenjen verzijam, ki to informacijo uporabljajo (pomen: backliteral([začetni stavek]/[deklaracije vhodnih spremenljivk], [deklaracije izhodnih spremenljivk]).

- Definicije in deklaracije predznanja:

  backliteral(path(X,X1,L), [X:item,X1:item], [L:list]).

  backliteral(link(X,X1),[X:item],[X1:item]).

  Drugače rečeno, hipoteze lahko uporabljajo rekurzivne klice in klice predikata link, ki je definiran kot:

  link(a,b).

  link(a,c).

  link(b,c).

  link(b,d).

  link(d,e).

- Pretvorbe spremenljivk v izraze:

  term(list, [X|L], [X:item, L:list]).

  term(list, [], []).

Spremenljivke tipa list se lahko pretvorijo v seznam s spremenljivko tipa item v glavi in spremenljivko tipa list kot repom, prav tako pa se lahko pretvorijo v prazen seznam.

- Največje število stavkov v hipotezi je bilo nastavljeno na tri.
- Največje število sprecializiranih hipotez je bilo nastavljeno na 700.

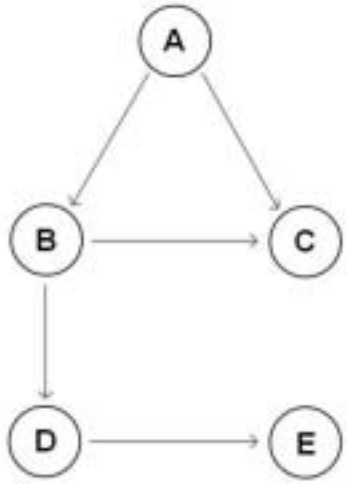
Oznaka serije	Št. poskusov v seriji	Št. pozitivnih primerov	Š. negativnih primerov
1 (15 0005)	100	2 (15%)	3 (0,05%)
2 (15 001)	100	2 (15%)	5 (0,1%)
3 (15 01)	100	2 (15%)	51 (1%)
4 (15 05)	100	2 (15%)	257 (5%)
5 (15 15)	100	2 (15%)	770 (15%)
6 (20 20)	100	3 (20%)	1027 (20%)
7 (50 50)	50	7 (50%)	2568 (50%)
8 (70 70)	20	10 (70%)	3595 (70%)
9 (100 100)	1	14 (100%)	5136 (100%)

tab. 4.2 Opis posameznih serij domene path

Uporabljali smo naslednje verzije sistema HYPER^2 in HYPER:

- hyper0 – originalna verzija sistema, z dodano omejitvijo da se iskanje ustavi po specializiranih 700 hipotezah (kot je nastavljeno s parametrom)
- hyper01 – HYPER^2, ki uporablja enak vhod kot HYPER, torej ne uporablja podatkov o vhodnih/izhodnih spremenljivkah v glavi stavkov iskane hipoteze, omejen na maksimalno 700 specializiranih hipotez
- hyper02 – kot hyper01; si ne zapomni cene stavka in jo je potrebno vsakič znova izračunati
- hyper03 – kot hyper01; si ne zapomni kompletnih hipotez, zato ne more uporabiti načela, da je nadmnožica kompletne hipoteze tudi kompletna hipoteza
- hyper04 – kot hyper01; ne preverja, če hipoteza že obstaja, posledično lahko hipotezo večkrat izostri
- hyper05 – kot hyper01; ne preverja, če nov stavek že obstaja, posledično lahko obstaja več enakih stavkov – torej tudi več enakih hipotez

- hyper06 – kot hyper01; stavki si ne zapomnijo svojih naslednikov, nasledniki se izračunajo vsakič, ko ostrimo stavek. Nasledniki se nato lahko najdejo pri preverjanju kopij stavkov, zato kopije stavkov oziroma hipotez ne obstajajo
- hyper07 – stavki si ne zapomnijo, katere primere pokrivajo, zato je pri vsaki hipotezi vedno na novo potrebno preveriti, katere primere pokriva



Sl. 4.3 Usmerjen graf uporabljen za domeno path

- hyper08 – združene so omejitve hyper03 in hyper07
- hyper09 – združene so omejitve hyper03, hyper06 in hyper07
- hyper10 – združene so omejitve hyper03, hyper05, hyper06 in hyper07
- hyper11 – kot hyper01; stavek ne ve ali je rekurziven. Po potrebi je treba vedno znova preverjati
- hyper12 – kot hyper01; uporablja preiskovanje s snopom (snop širine 50)
    - hyper12A uporablja snop širine 100
    - hyper12_A05 uporablja snop širine 5
    - hyper12_A10 uporablja snop širine 10
    - hyper12_A15 uporablja snop širine 15
    - hyper12_A20 uporablja snop širine 20
    - hyper12_A30 uporablja snop širine 30
    - hyper12_A40 uporablja snop širine 40

- hyper13 – uporablja iskanje s snopom in podatke o vhodnih/izhodnih spremenljivkah v glavah stavkov. Posledično lahko izloči nekatere stavke kot neprimerne. Uporablja snop širine 50
    - hyper13A uporablja snop širine 100
    - hyper13_A05 uporablja snop širine 5
    - hyper13_A10 uporablja snop širine 10
    - hyper13_A15 uporablja snop širine 15
    - hyper13_A20 uporablja snop širine 20
    - hyper13_A30 uporablja snop širine 30
    - hyper13_A40 uporablja snop širine 40
- hyper14 – uporablja iskanje najprej najboljši (kot hyper01) in podatke o vhodnih/izhodnih spremenljivkah v glavah stavkov. Posledično lahko izloči nekatere stavke kot neprimerne
- hyper15 – kot hyper14; s tem, da si pokrite primere zapomnimo v povezavi s hipotezami in ne v povezavi s stavki

V vseh poskusih so bile vse verzije sistema omejene na specializacijo 700 hipotez. Če v 700 hipotezah sistem ni našel primerne hipoteze, se je izvajanje ustavilo. Treba je pojasniti, da so bile verzije od hyper12 dalje narejene naknadno zaradi nekaterih pomanjkljivosti, ki so bile opažene pri osnovni verziji (hyper01) ravno pri testiranju sprememb. Zaradi tega ni bila za osnovno verzijo vzeta verzija, ki spada tako med najbolj uspešne kot tudi najhitrejše (hyper15). Testiranje verzij hyper12, hyper13, hyper14 in hyper15 je bilo opravljeno po prvotnem testiranju sprememb. Obe testiranji sta potekali na istem računalniku. Pri drugem testiranju (dodatnih verzij) se je vedno pognala tudi verzija hyper01. Če je prišlo do časovnih sprememb pri testih verzije hyper01, se je razmerje med časi uporabilo za izračunanje časov, ki bi jih nove verzije dosegle, če bi bile testirane istočasno kot ostale verzije. Ko je do sprememb prišlo, so bile običajno manjše od 5%.

Vse različne verzije se lahko glede na obnašanje pri iskanju razdelijo v več skupin:

- Originalni HYPER (hyper0).
- Skupina A, ki vsebuje verzije, ki so enako uspešne kot osnovna verzija sistema HYPER[2] (tukaj označena kot hyper01). V to skupino spadajo verzije hyper01,

hyper02, hyper03, hyper06, hyper07, hyper08, hyper09 in hyper11. To skupino lahko nadalje razdelimo na podskupino, ki preiskuje prostor enako kot hyper01. To so hyper01, hyper02, hyper03, hyper06 in hyper11, ostali (vsi vsebujejo omejitve verzije hyper07) občasno (a redko) preiščejo prostor malo drugače. Ker se primeri, ki jih hipoteza pokriva, izračunavajo sproti in se ne seštevajo iz primerov, ki jih pokrivajo stavki, se zgodi, da odpade kakšna hipoteza, ki jo je hyper01 sprejel kot možno.

- Skupina B, ki preišče več hipotez kot hyper01 in zato pogosteje naleti na omejitev 700 hipotez (hyper04, hyper05 in hyper10).
- Skupina C, ki uporablja iskanje s snopom (hyper12 in hyper13).
- Skupina D, ki zna uporabljati podatke o vhodnih in izhodnih spremenljivkah v glavah stavkov (hyper14 in hyper 15).

4.1 Skupina A

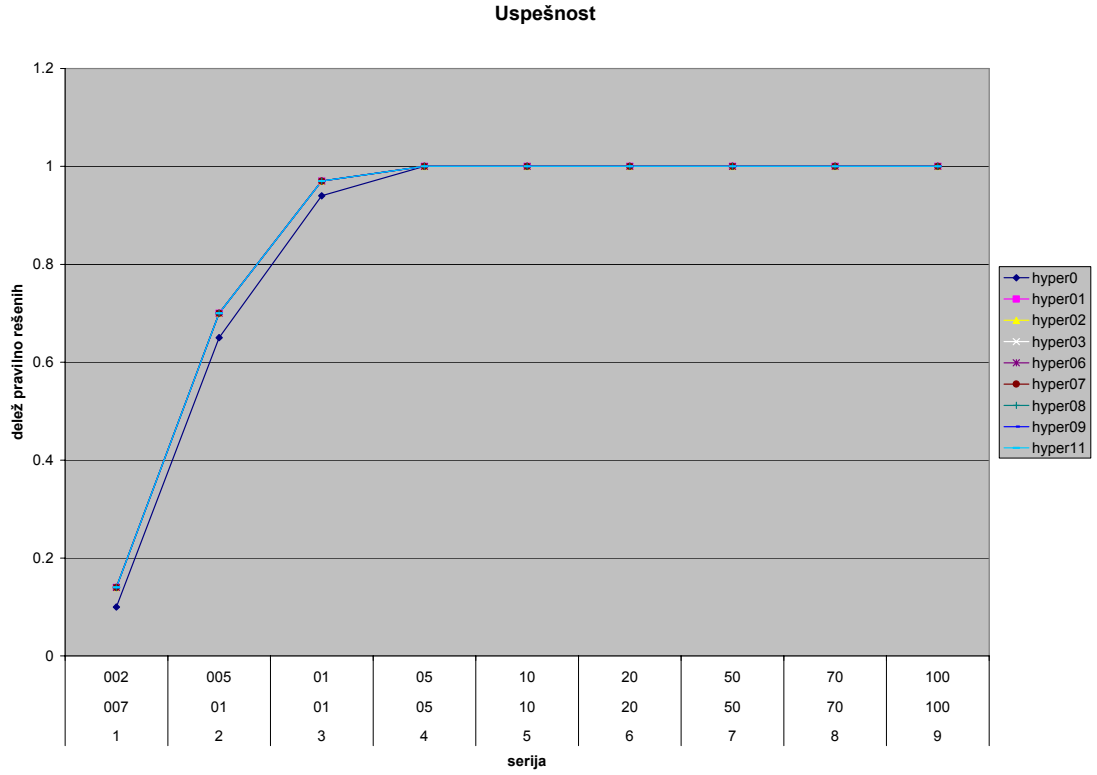
Najprej se osredotočimo na skupino, ki ima enako učinkovitost kot osnovna verzija sistema HYPER[2] (kot se vidi na Sl. 4.4 in Sl. 4.5, kjer izstopa le originalni HYPER). Če uspešnost (delež izvajanj, ki se končajo s pravilno rešitvijo) teh verzij primerjamo z uspešnostjo originalnega sistema HYPER, opazimo, da so na domeni member (Sl. 4.4) te verzije malo boljše (razlika je do 5% vseh izvajanj), medtem ko je razlika na domeni path (Sl. 4.5) že bolj opazna, saj rešijo do 24% več problemov (od 100 v eni seriji), poleg tega začenjajo uspešno reševati probleme z manj učnimi primeri.

Verzije hyper02, hyper03, hyper06 in hyper11 iščejo po prostoru hipotez natanko tako kot osnovna verzija hyper01 (število preiskanih hipotez se lahko vidi na Sl. 4.6 in Sl. 4.7) in ravno tolikokrat pokličejo meta-interpreter, kar se vidi na Sl. 4.8 in Sl. 4.9, saj so modifikacije, ki so izklopljene v teh verzijah, namenjene samo hitrejšemu in lažjemu ocenjevanju novih hipotez. Tak je tudi namen modifikacije, ki jo testiramo z verzijo hyper07 (ter tudi hyper08 in hyper09, kjer so izklopljene kombinacije modifikacij). Pri tej modifikaciji si stavki zapomnijo, katere primere pokrivajo in pokrivanje s strani celotne hipoteze je izračunano kot unija pokrivanja s strani stavkov. Vendar se lahko zgodi, da taka metoda sprejeme v nadaljnjo obdelavo hipoteze, ki bi jih originalni HYPER (v tem testiranju označen kot hyper0) in tudi verzija hyper07, ki pokritost računa sproti, kot originalni HYPER, zavrnil. Lahko se namreč zgodi, da neki pozitivni učni primer prvi stavek v hipotezi mogoče pokriva, sledeči stavek pa ga pokriva. Originalni HYPER in tudi hyper07 tako hipotezo zavrneta, ker

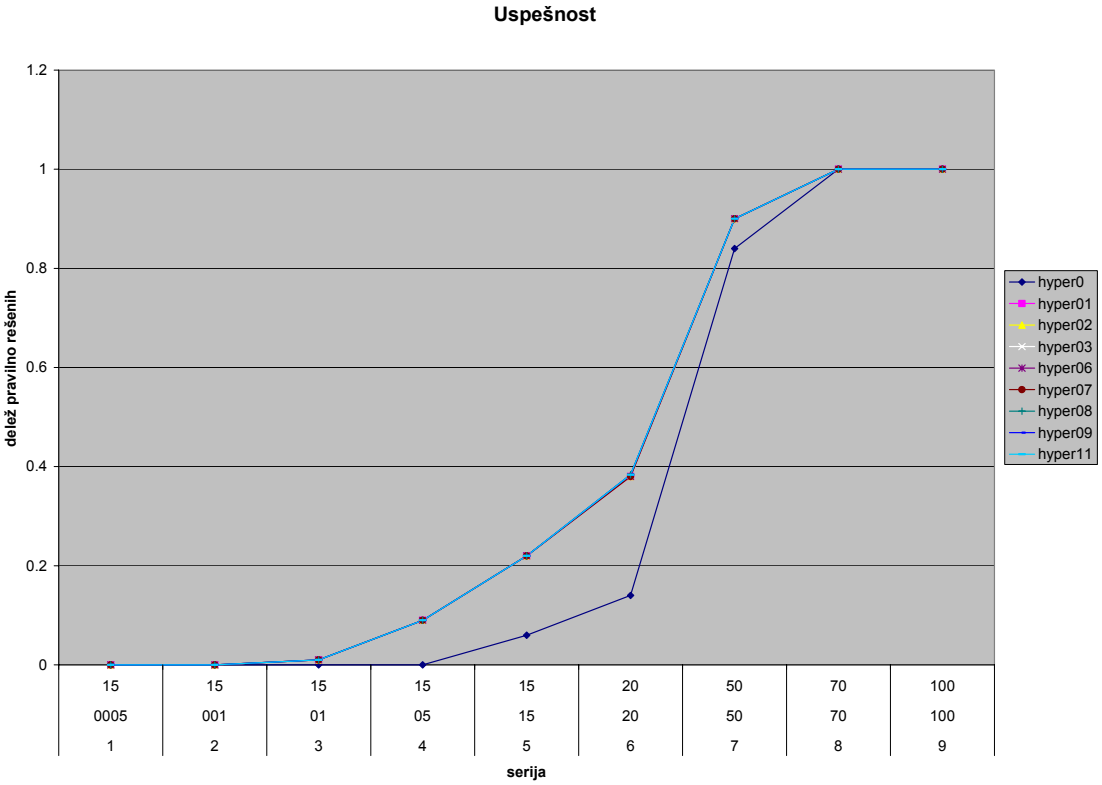
mogoče pokriva ta pozitivni primer (torej ga zaradi strategije najslabši možni primer zanju ne pokriva), medtem ko tako hipotezo osnovna verzija sistema HYPER^2 sprejme, saj misli, da je primer pokrit. Ker pa smo predvidevali, da se taka razlika pojavi le redko (razlika se opazi na Sl. 4.6 in Sl. 4.7), nismo dodali dodatnega mehanizma, ki bi to preverjal.

Kljub temu, da zaradi te razlike HYPER^2 pregleda kakšno hipotezo, ki bi jo originalni HYPER izpustil, pa se večja razlika pokaže pri podvojenih hipotezah in stavkih, ki jih pregleduje originalni HYPER, medtem ko jih HYPER^2 pregleda le enkrat.

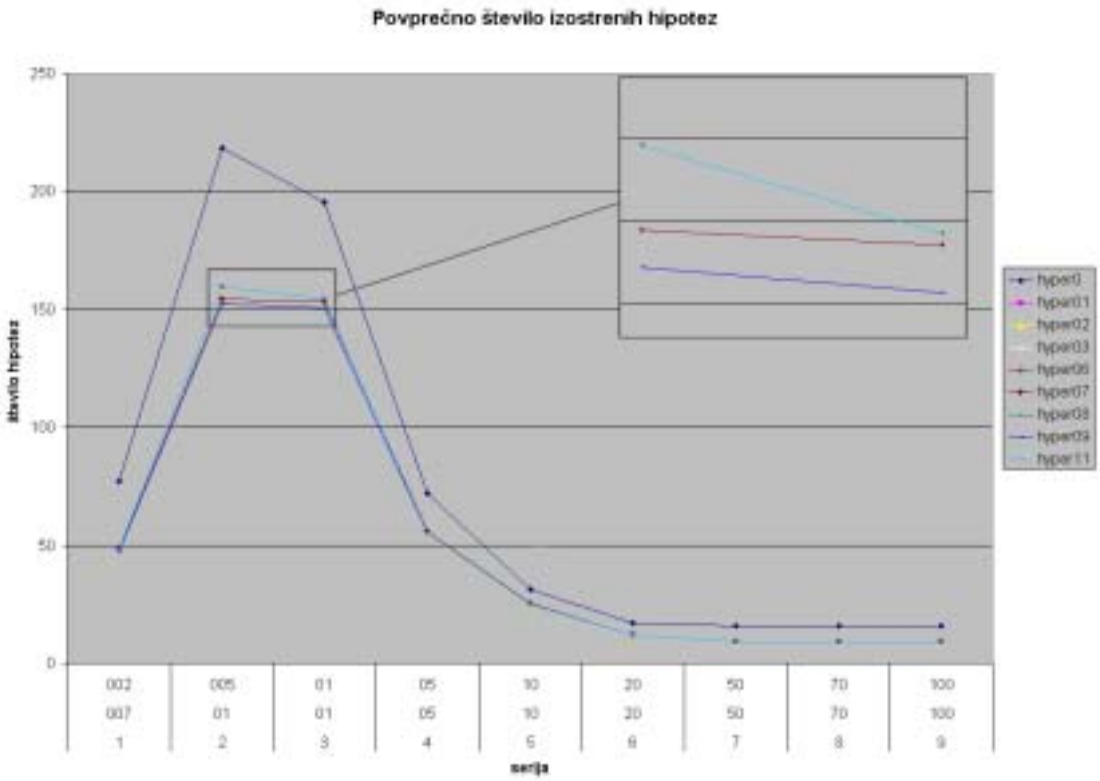
Druga, večja in nekoliko nepričakovana razlika med podskupinami (oziroma predvsem med verzijama hyper01 in hyper07) se pojavi pri številu klicev meta-interpreterja (Sl. 4.8 in Sl. 4.9). Nekoliko nepričakovana, ker je sama razlika bila pričakovana, vendar ne v prednost verzije hyper07, ki glede na meritve občutno redkeje kliče meta-interpreter. Saj če si zapomnimo, katere primere pokrivajo posamezni stavki, ni potrebno računati njihove pokritosti za vse hipoteze, v katerih nastopajo, ampak to naredimo le enkrat. Poleg tega nasledniki določenega stavka lahko pokrivajo največ primere, ki jih pokriva originalni, starševski stavek. To bi v principu moralo zmanjšati število klicev meta-interpreterja. Vendar se izkaže, da lahko v hipotezi več stavkov pokriva en primer, pokritost pa mora biti izračunana za vsak stavek, poleg tega smo testirali na rekurzivnih domenah, za katere velja, da je pokrivanje s strani rekurzivnega stavka odvisno od ostalih stavkov v hipotezi, torej je potrebno na novo izračunati pokritost rekurzivnega stavka, tudi če se le ta ne spremeni, ampak se spremeni kateri koli drug stavek v hipotezi. Dodatni faktor pa še prinese, da v domenah, ki se rešujejo z rekurzivno definicijo veliko večino pozitivnih primerov pokriva rekurzivni stavek (tako da ne pride do velikega vpliva dejstva, da lahko nek rekurzivni stavek pokriva največ primere, ki jih je pokrival njegov predhodnik v starševski hipotezi). Na koncu se izkaže, da prednost prinese tudi dejstvo, da lahko, če računamo pokritost sproti, na nivoju hipotez, nekatere hipoteze zelo hitro zavržemo, ker niso kompletne (v najboljšem primeru ne pokrivajo prvega pozitivnega učnega primera), medtem ko je potrebno pri računanju pokritosti na nivoju stavkov izračunati pokritost za vse stavke (če teh podatkov še nimamo), nato narediti unijo in šele nato lahko ugotovimo, da je hipoteza nekompletna.



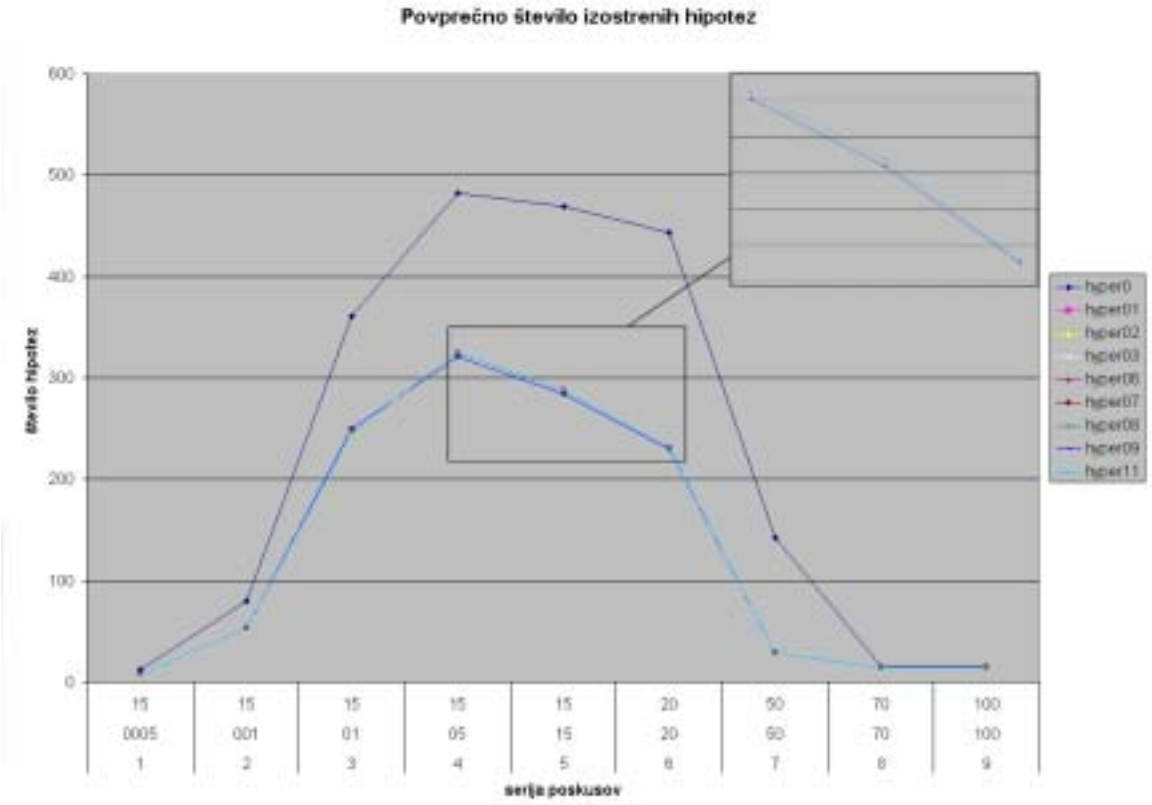
Sl. 4.4 Uspešnost verzij sistema HYPER^2 iz skupine A na domeni member. Vse verzije so enako uspešne, izstopa le originalni HYPER (hyper0)



Sl. 4.5 Uspešnost nekaterih verzij sistema HYPER^2 na domeni path. Vse verzije so enako uspešne, izstopa le originalni HYPER (hyper0)



Sl. 4.6 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 na domeni member



Sl. 4.7 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 na domeni path

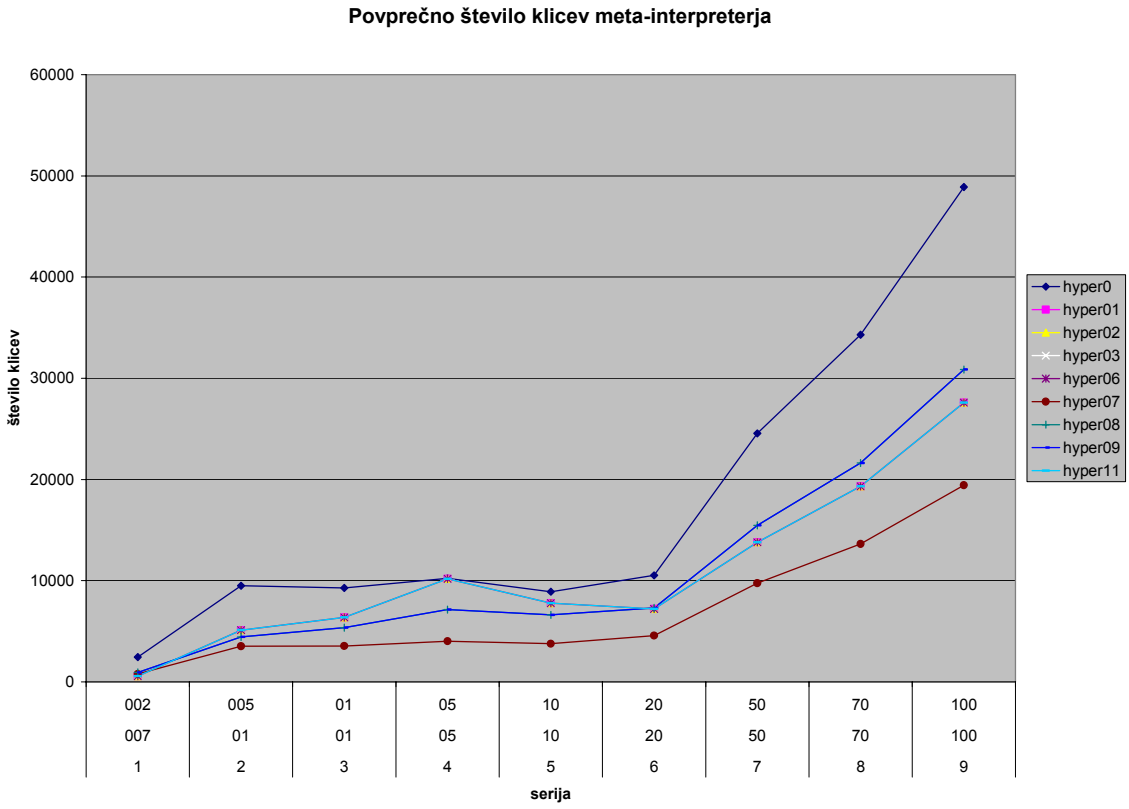
Medtem ko hyper03 (v tej verziji ne privzamemo, da je nadmnožica kompletne hipoteze tudi kompletna) v primerjavi s hyper01 ne deluje različno, pa hyper08 in hyper09 v primerjavi s hyper07 povsod delujeta malo drugače. Ker pri hyper01 in hyper03 pokritost s strani hipoteze dobimo tako, da naredimo unijo pokritosti s strani stavkov, že na ta način privzamemo, da je nadmožica kompletne hipoteze tudi kompletna (sami uniji, ki smo jo dobili že za prejšnjo kompletno hipotezo in vsebuje vse pozitivne učne primere, dodamo še nekaj množic pozitivnih učnih primerov – ne da bi se rezultat spremenil). Pri hyper07, hyper08 in hyper09 računamo pokritost na nivoju hipoteze (treba je spomniti, da je zaradi meta-interpreterja omejeno število korakov dokaza pokritosti), zato se lahko (in tudi se) zgodi dvoje: s preverjanjem kompletnosti primerne hipoteze (ki bi jo vse verzije sprejele) z iskanjem poznanih kompletnih podmnožic najdemo kompletno podmnožico, kar pomeni, da ni potrebno preverjati pokritosti pozitivnih učnih primerov, ampak le negativnih, kar privede do zmanjšanja klicev meta-interpreterja in tudi manjše porabe časa (kar se zgodi na domeni member). Lahko pa se zgodi, da preverjamo neprimerno hipotezo (ki jo originalni HYPER, hyper08 in hyper09 zavrnejo), ki vsebuje kot podmnožico kompletno hipotezo (recimo, da je prvi stavek neskončna zanka, ostali stavki pa sestavljajo primerno, kompletno hipotezo – če tako hipotezo sprejmemo v nadaljnjo obdelavo, ne obstaja možnost, da bi njene naslednike sistem sprejel kot pravilno rešitev, ne glede na način računanja kompletnosti). V tem primeru nam tudi ni potrebno preverjati pokritosti pozitivnih učnih primerov (hyper8 in hyper9 morata vsaj za en pozitivni učni primer preveriti pokritost), vendar pa moramo preveriti pokritost negativnih učnih primerov (kar pa hyper8 in hyper9 ne naredita, saj zavrneta hipotezo kot nekompletno), zato se poveča število klicev meta-interpreterja in tudi poraba časa (kar se zgodi na domeni path).

Če pogledamo porabo časa (Sl. 4.10 in Sl. 4.11), opazimo, da je na prvi pogled originalni HYPER (označen kot hyper0) predvsem na domeni path občutno hitrejši od vseh verzij sistema HYPER^2. Vendar če primerjamo posamezne primere, ki jih oba rešita pravilno (oziroma pri vseh, ko se oba zaustavita pred omejitvijo 700 hipotez), je HYPER^2, običajno hitrejši, v najslabšem primeru pa je rešitev najdena v podobnem času (povprečni časi in druge meritve v odvisnosti od pravilnosti rešitve so na voljo v dodatku A). Le ko se originalni HYPER (ali pa oba) zaustavi po 700 izostrenih hipotezah, je HYPER^2 počasnejši, ker preiskuje po nekoliko kompleksnejši poti. Drugo opažanje, ki ga lahko zaznamo glede porabljenega časa, je, da iz posameznih podskupin izstopata verziji hyper06 in hyper09, kljub temu da sta na ostalih meritvah enaka kot najbolj podobni verziji (hyper01 oziroma hyper08).

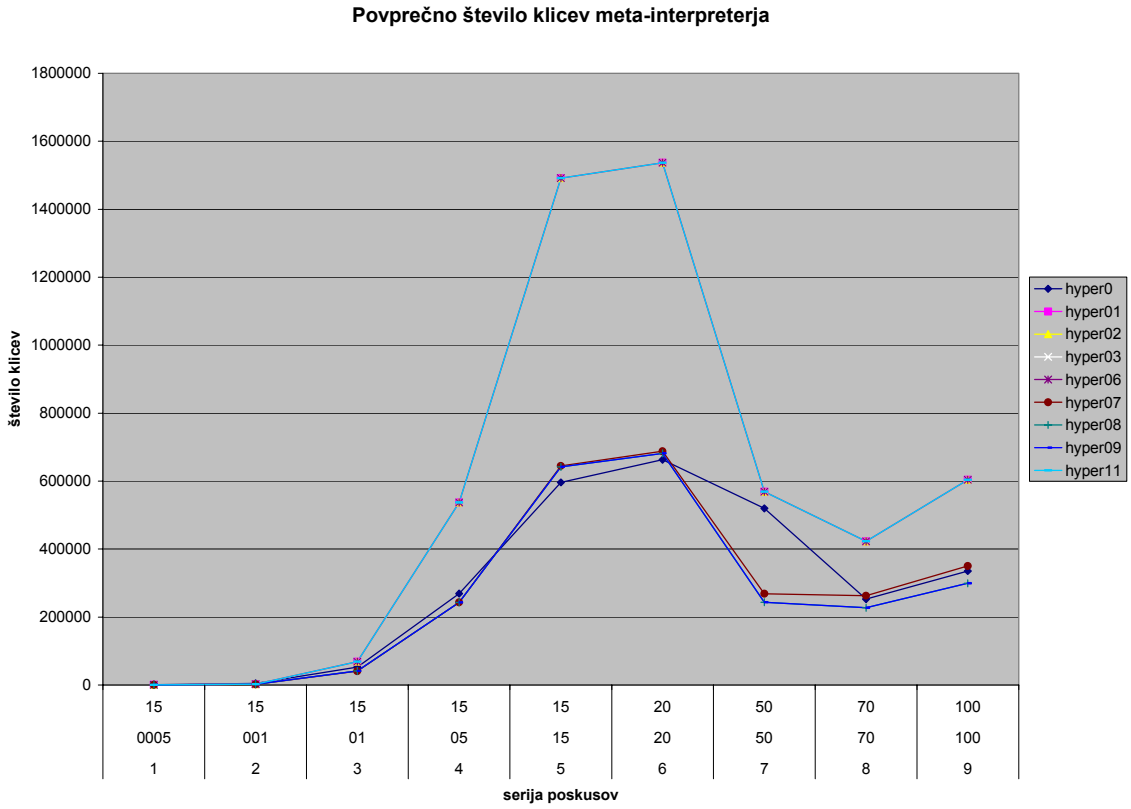
Samo ponovno generiranje stavkov (v nasprotju z uporabo naslednikov v grafu stavkov) sicer ne porabi veliko časa, vendar ponovno iskanje kopij (kar je potrebno le, ko generiramo stavek) zahteva večjo porabo časa, saj je potrebno preveriti veliko število stavkov - v domeni path je bilo običajno generiranih med 100 in 5000 stavkov.

Zmanjšanje porabe časa pri kasnejših serijah pri domeni path lahko razložimo s kombinacijo večanja uspešnosti in krajši enostavnejši poti do prave rešitve s povečanjem števila učnih primerov. Pri domeni member ta padec porabe časa pride že pri srednjih serijah, nato pa se čas poveča sorazmerno s številom učnih primerov.

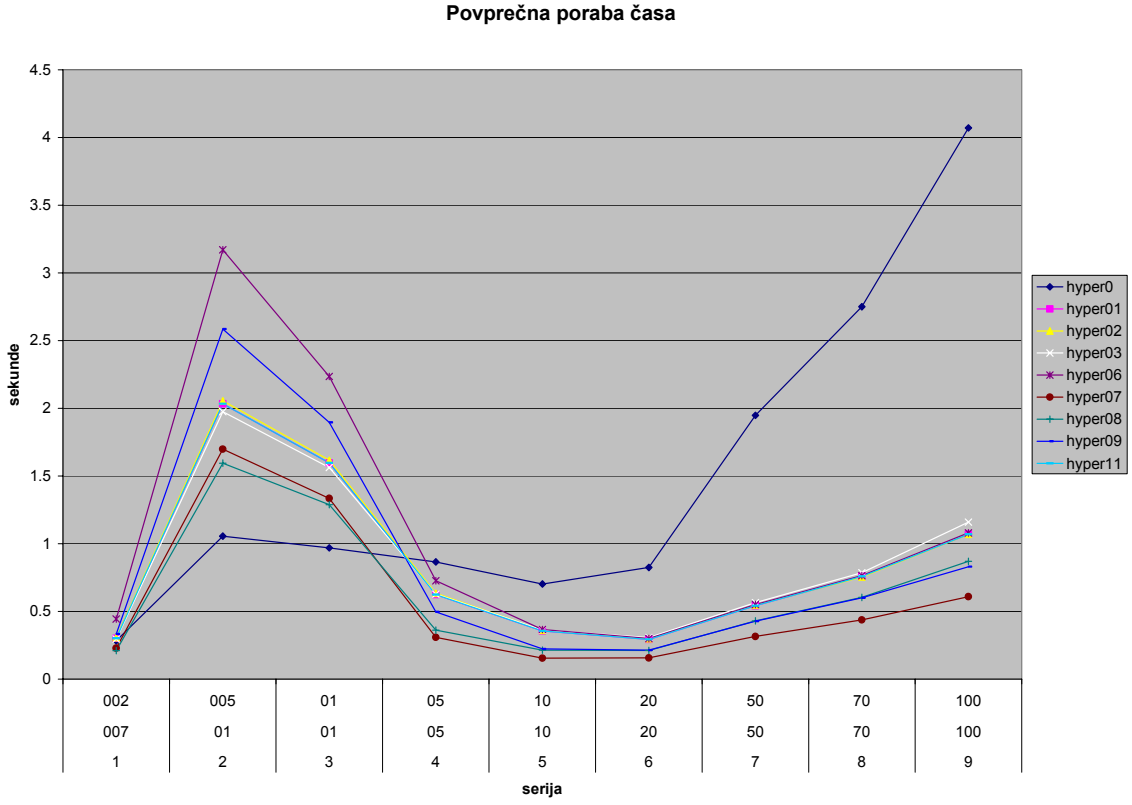
Pri testiranju verzij v tej skupini se je predvsem izkazalo, da nekatere modifikacije s seboj prinesejo poleg potencialne pohitritve delovanja tudi potencialno upočasnitev. Še več, občasno je ta upočasnitev veliko večja od pohitritve.



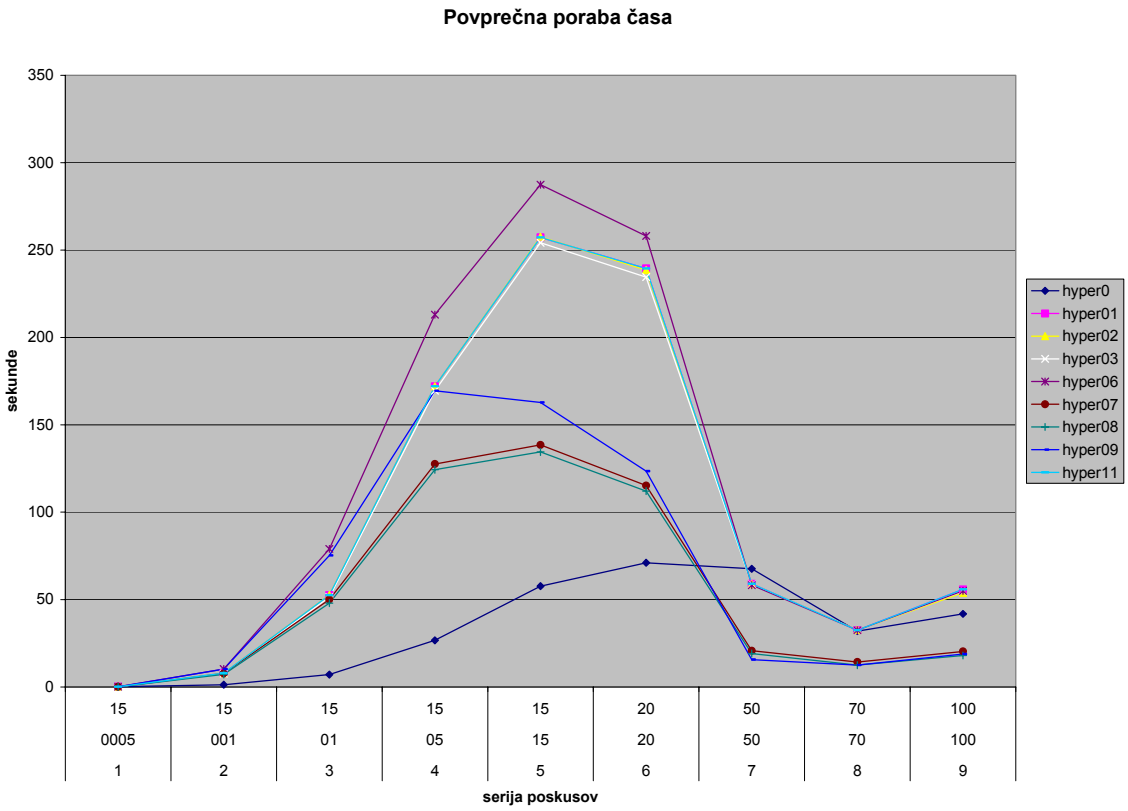
Sl. 4.8 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER[2] na domeni member



Sl. 4.9 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 na domeni path



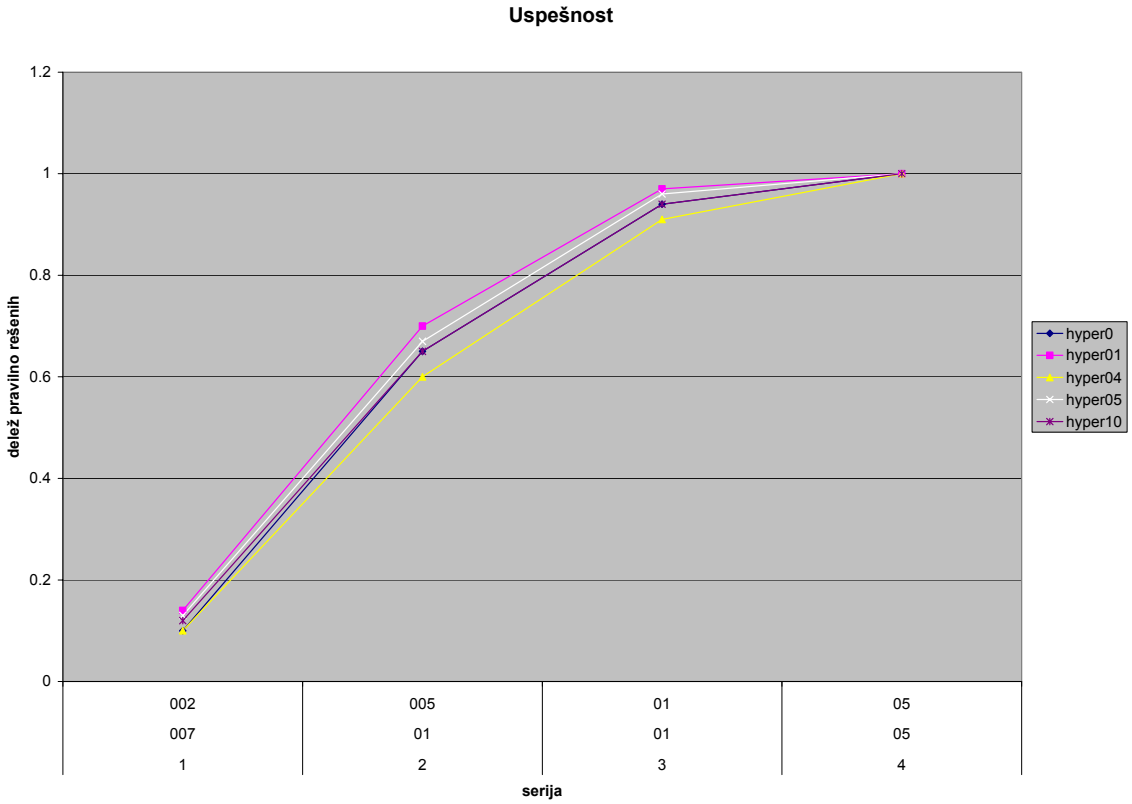
Sl. 4.10 Povprečna poraba časa nekaterih verzij sistema HYPER^2 na domeni member



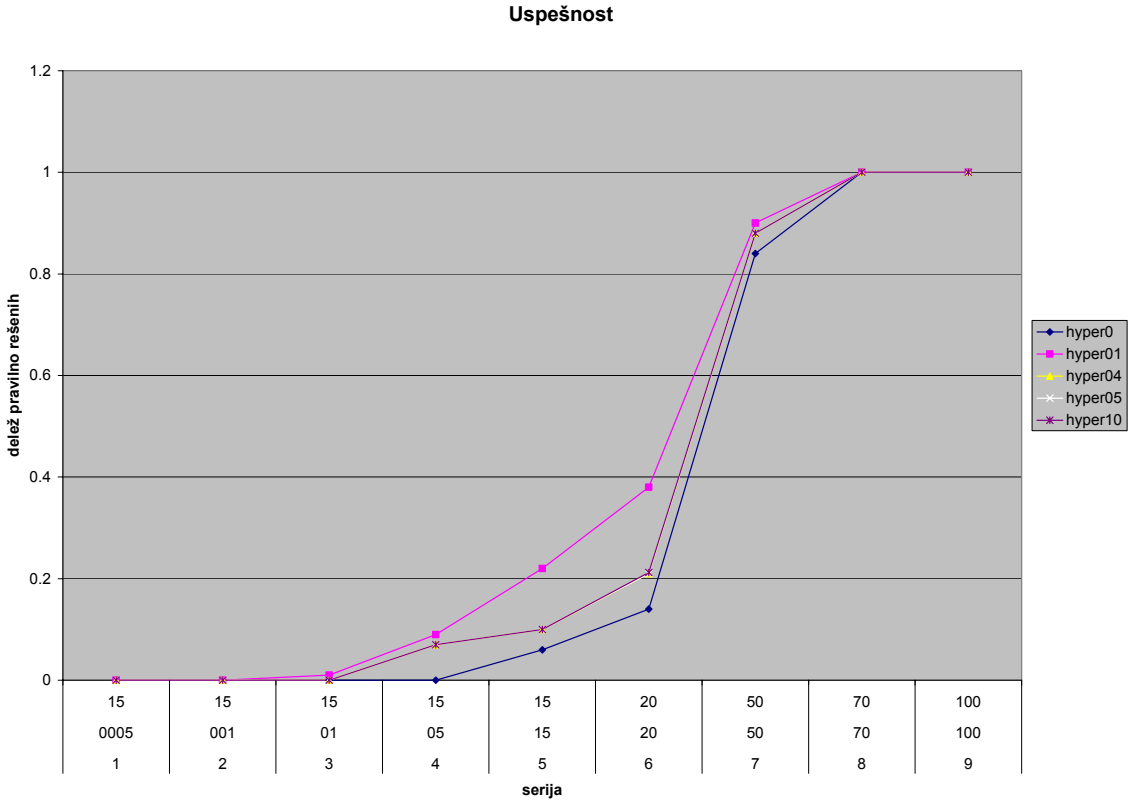
Sl. 4.11 Povprečna poraba časa nekaterih verzij sistema HYPER^2 na domeni path

4.2 Skupina B

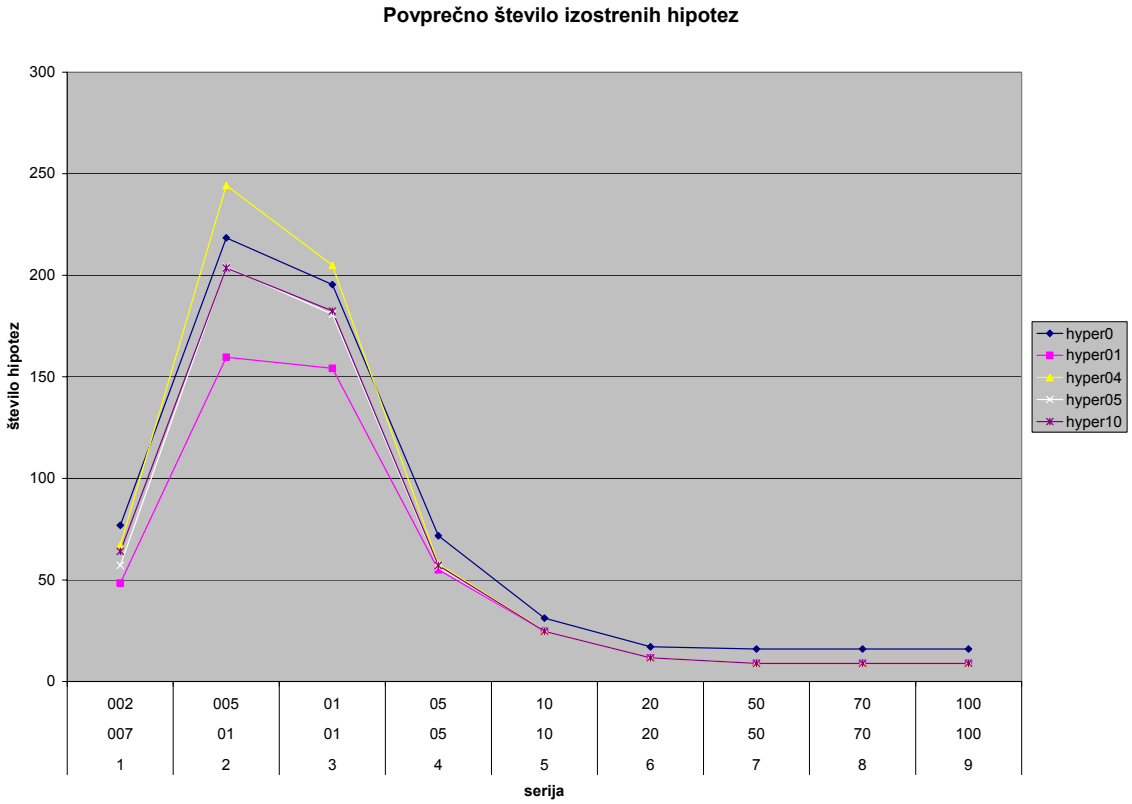
V skupini B so zbrane verzije sistema HYPER^2, ki imajo izklopljene modifikacije, ki vplivajo na pot preiskovanja skozi prostor možnih hipotez. Tako hyper04 ne preverja, če smo trenutno generirano hipotezo že generirali (ter mogoče celo izostrili), hyper05 pa ne preverja, ali trenutno generiran stavek že obstaja. Pri sistemu hyper10 je izklopljena kombinacija modifikacij - ne preverja, ali stavki že obstajajo, ne preverja, ali hipoteze vsebujejo kot podmnožice kompletne hipoteze, stavki si ne zapomnijo naslednikov (kar v kombinaciji z nepreverjanjem prejšnjega obstoja stavkov prinese večjo stopnjo ponavljanja) in stavki si ne zapomnijo primerov, ki jih pokrivajo. Vse te izklopljene modifikacije povzročijo manjšo uspešnost verzij sistema (Sl. 4.12 in Sl. 4.13). To je predvsem posledica dejstva, da pregledajo večje število hipotez (zaradi ponavljanja) in bolj pogosto pridejo do omejitve 700 izostrenih hipotez (kar se lahko vidi pri povprečnem številu izostrenih hipotez na Sl. 4.14 in Sl. 4.15). Pri tem se pri hyper04 lahko pojavi celo večje ponavljanje kot pri originalnem HYPER. Ta namreč ne preverja, ali je bila novo generirana hipoteza že izostrena, vendar pa ne dopušča kopij enake hipoteze v vrsti čakajočih na ostrenje. Posledično ima hyper04 lahko večje število izostrenih hipotez in manjšo uspešnost.



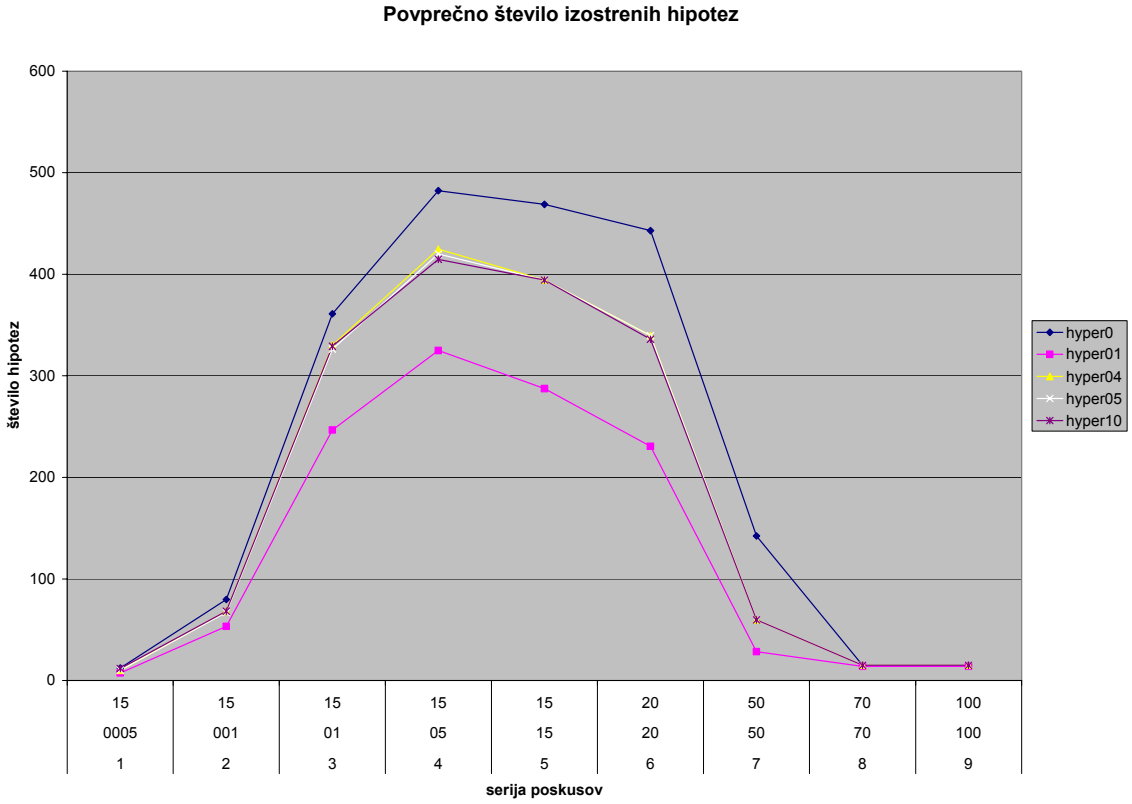
Sl. 4.12 Uspešnost nekaterih verzij sistema HYPER^2 na domeni member - prikazane so samo prve 4 serije (razen v prvi točki sta hyper0 in hyper10 enako uspešna)



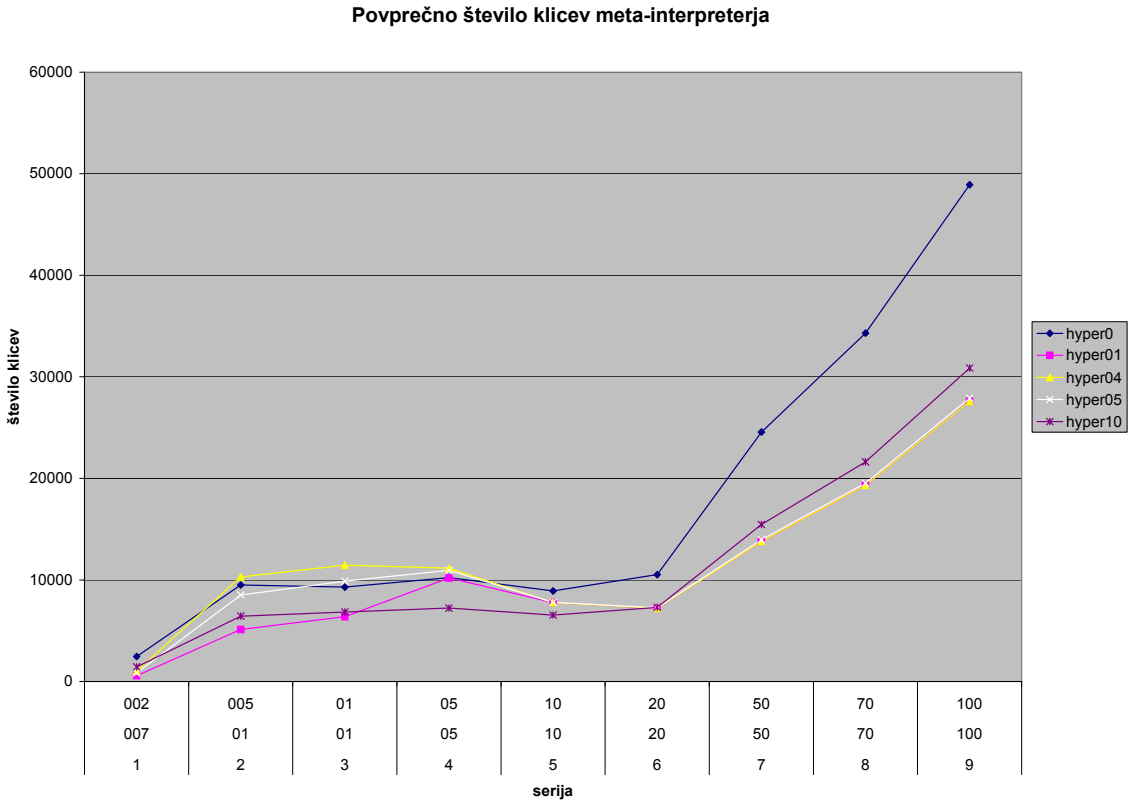
Sl. 4.13 Uspešnost nekaterih verzij sistema HYPER^2 na domeni path (vse verzije skupine B so enako uspešne)



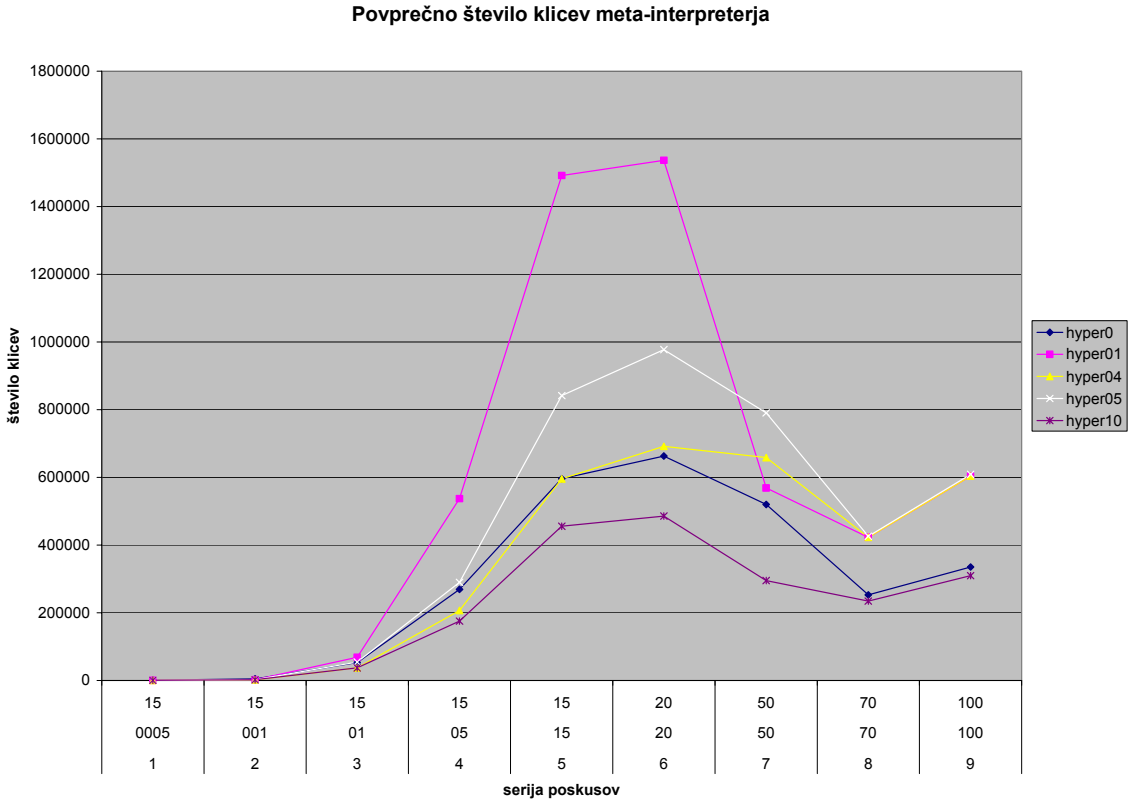
Sl. 4.14 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 na domeni member



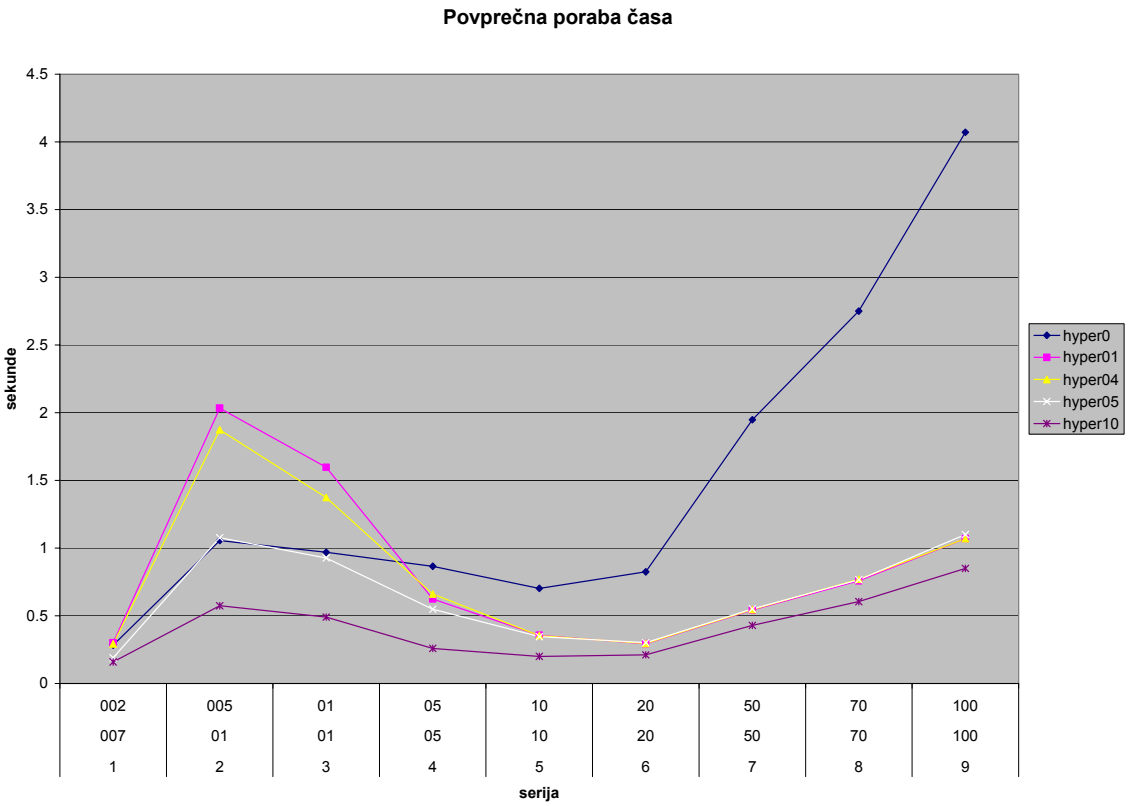
Sl. 4.15 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 na domeni path



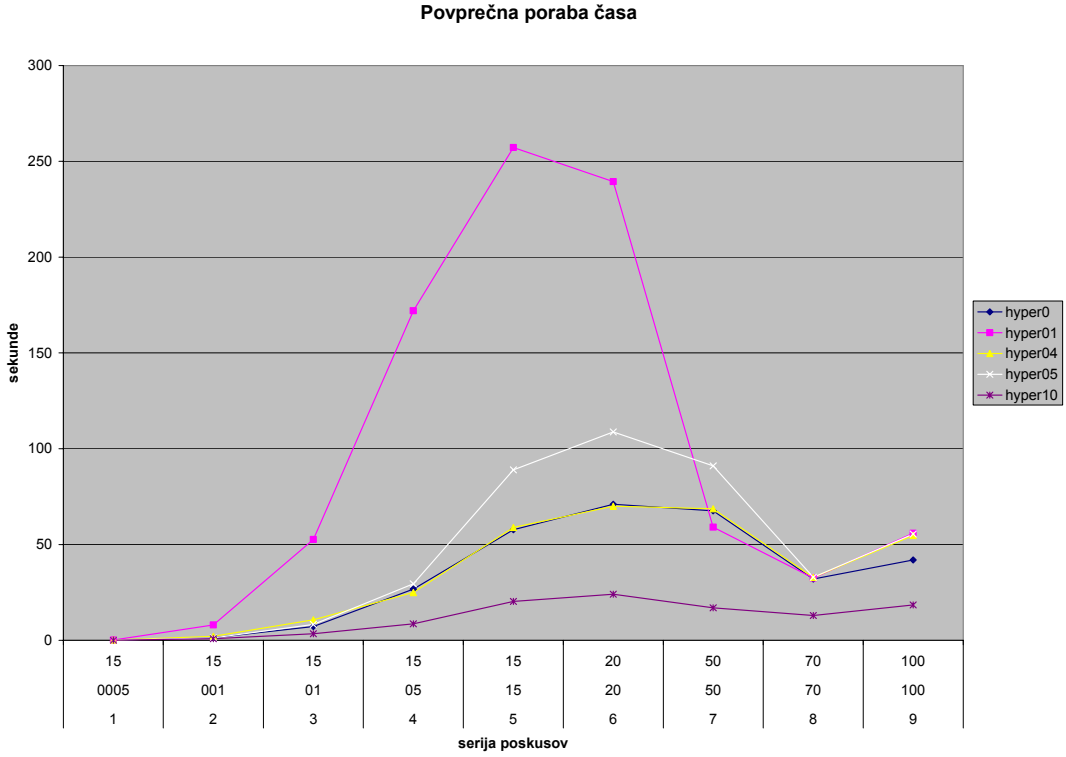
Sl. 4.16 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 na domeni member



Sl. 4.17 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 na domeni path



Sl. 4.18 Povprečna poraba časa nekaterih verzij sistema HYPER^2 na domeni member



Sl. 4.19 Povprečna poraba časa nekaterih verzij sistema HYPER^2 na domeni path

Zaradi večjega števila pregledanih hipotez je tudi število klicev meta-interpreterja večje kot pri osnovni verziji. Vendar to velja le, ko vse verzije pravilno rešijo problem. Ko le osnovna verzija reši problem, druge verzije pa naletijo na omejitev 700 izostrenih hipotez, je teh 700 hipotez običajno bolj enostavnih, kar povzroči manjše število klicev meta-interpreterja. Podobno je tudi, kadar vse verzije dosežejo omejitev 700 izostrenih hipotez. Manjše število klicev meta-interpreterja povzroči tudi krajši čas izvajanja (Sl. 4.18 Sl. 4.19). Zaradi enakih razlogov kot pri verziji hyper07 (le-te smo razložili v prejšnjem podpoglavju) tudi verzija hyper10 uporabi manj klicev meta-interpreterja (in tudi manj časa) kot primerljive verzije (v tem primeru verzija hyper5).

Modifikacije, ki nam preprečujejo uporabo kopij tako hipotez kot tudi stavkov, nam očitno pripomorejo k lepše usmerjeni poti skozi prostor možnih hipotez in na ta način tudi k večji učinkovitosti sistema.

4.3 Skupina C

Sistemi v skupini C so bili prvi sistemi narejeni z namenom, da se odstranijo pomanjkljivosti, ki so se izkazale pri testiranju osnovne verzije (tako pri testiranju modifikacij kot tudi pri primerjalnem testiranju z drugimi sistemi). Najprej smo, da bi dosegli večjo učinkovitost, spremenili način preiskovanja prostora možnih hipotez iz najprej najboljši na iskanje s snopom (hyper12), nato smo poskušali še odpraviti probleme, ki so se občasno pojavljali, ker je osnovna verzija nepravilno uporabljala rekurzivne klice. Zaradi tega smo dodali še informacijo o vhodno/izhodnem tipu spremenljivk v glavah stavkov hipoteze (hyper13).

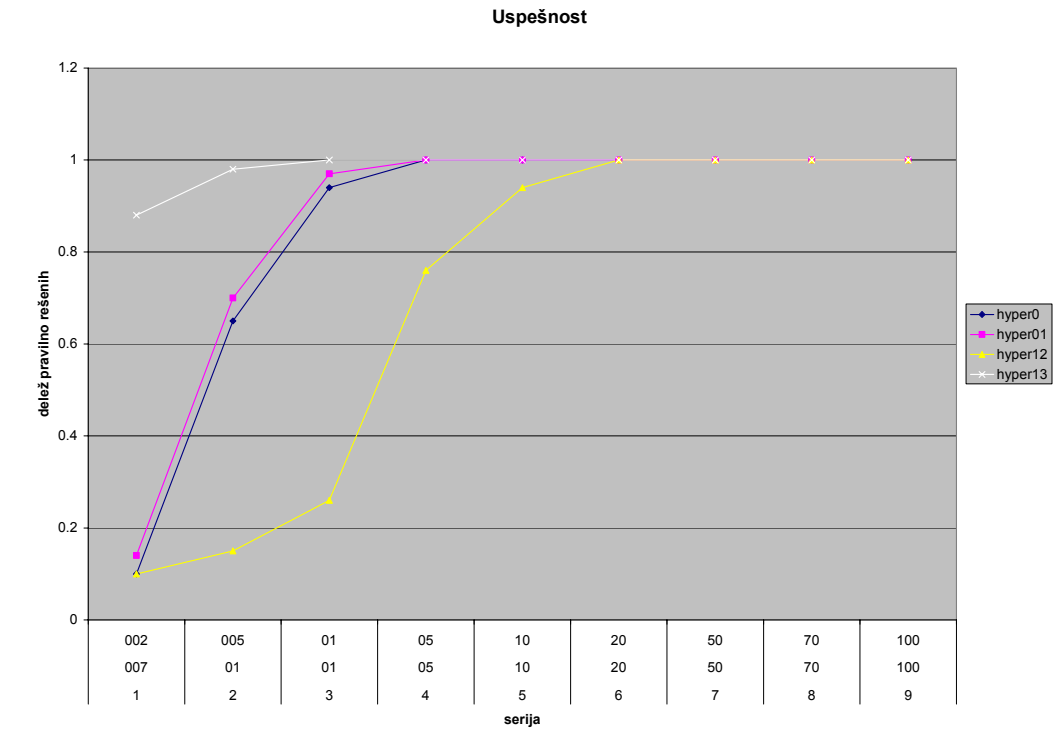
Če pogledamo uspešnost teh verzij na domeni member (Sl. 4.20) in na domeni path (Sl. 4.23), vidimo, da z iskanjem s snopom na teh domenah nismo nič pridobili (obstajajo domene, kjer se obnese bolje kot osnovna verzija). Še več, verzija z iskanjem s snopom se obnese celo slabše od originalnega sistema HYPER. Ker v snopu dopuščamo najboljše stare hipoteze in ko nobena izmed novih hipotez ni bolje ocenjena od starih hipotez, se (dokaj pogosto) dogodi, da se sistem zaustavi, ker so vse hipoteze v snopu že izostrene in nobena izmed novih hipotez ni dovolj dobra, da bi se uvrstila v snop. Če bi dovolj povečali širino snopa, bi se uspešnost te verzije povečala (kako narašča uspešnost s povečanjem širine snopa je razvidno na Sl. 4.21). Na domeni path se obnese izredno slabo, predvsem ker je izvajanje pogosto bilo ustavljeno po treh urah delovanja. Večja širina snopa je zaradi večje porabe časa (in pogostejših ustavitev) celo zmanjšala učinkovitost. Dosti več kot samo z iskanjem s snopom smo pridobili z informacijo o (vhodno/izhodnem) tipu spremenljivk v glavah stavkov iskanih hipotez.

Osnovna verzija sicer podobne informacije že uporablja, ampak samo za predznanje, medtem ko za spremenljivke v glavah predvideva, da so vse vhodne (kar pomeni, da imajo vse poznane vrednosti – kar je, če uporabimo učne podatke v prvem klicu, res, v rekurzivnih klicih pa ni nujno tako). Tako nastali konflikti so upočasnjevali delovanje, zahtevali posebne definicije predznanja in poleg vsega dopuščali nekatere nesmiselne definicije hipotez. Z dodatkom informacije o tipu spremenljivk v glavi stavkov so se te težave odpravile, predvsem pa so se prepovedale nekatere hipoteze, ki na napačen način kličejo predznanje (in izvajajo rekurzivne klice). Posledično se je nekaj zmanjšal celo preiskovalni prostor možnih hipotez. Vse to privede v povečano uspešnost te verzije v primerjavi z osnovno verzijo sistema HYPER^2. Poleg tega doseže veliko uspešnost z majhno širino snopa (na Sl. 4.22 se krivulje uspešnosti za širine snopa 15, 20, 30, 40 in običajnih 50 prekrivajo – kar pomeni, da maksimalno uspešnost na domeni member doseže že s snopom širine 15, namesto običajne 50). Na domeni path s povečanjem širine snopa pridobi na uspešnosti (kar je razumljivo, saj je path bolj kompleksna domena od domene member).

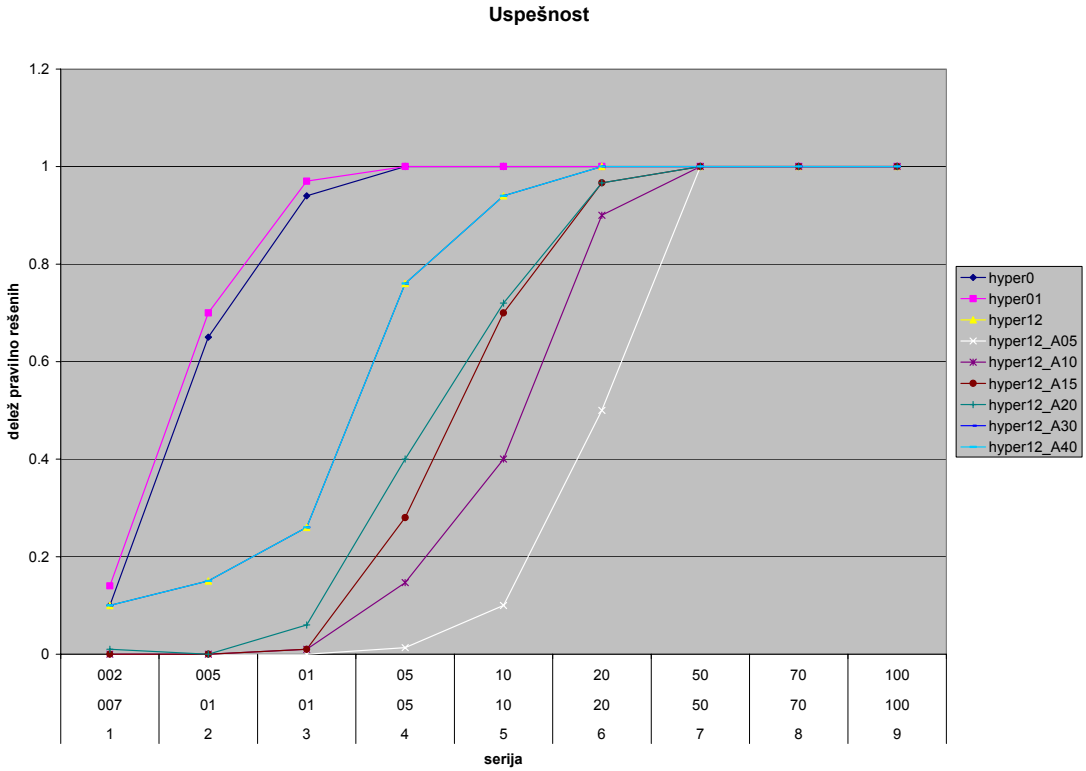
Število izostrenih hipotez verzij, ki uporabljata iskanje s snopom, je na domeni member dokaj konstantno in je odvisno predvsem od širine snopa (Sl. 4.24 do Sl. 4.26), ker najkasneje v peti generaciji najdeta rešitev (pravilno ali napačno), poleg tega pa je domena member dokaj omejena, saj vsebuje le delo s seznami in rekurzivne klice. Domena path pa potrebuje vsaj deset generacij (do pravilne rešitve), klici predznanja pa jo naredijo še nekoliko bolj kompleksno. Zaradi tega število izostrenih hipotez ni več tako konstantno, dodatno pripomore še slabša uspešnost sistemov na tej domeni. Če se osredotočimo samo na pravilno rešene probleme, je število izostrenih hipotez tudi v tej domeni dokaj konstantno. Grafi, ki prikazujejo število izostrenih hipotez, število klicev meta-interpreterja in porabo časa glede na pravilnost rešitve, so v dodatku A.

Tako za hyper12 kot za hyper13 velja, da je število klicev meta-interpreterja (Sl. 4.28 in Sl. 4.31) odvisno tako od števila izostrenih hipotez kot tudi števila učnih primerov v problemu. Enako velja tudi za porabljeni čas (krivulje na Sl. 4.32 do Sl. 4.35 so podobne krivuljam za število klicev meta-interpreterja, če izvzamemo hyper12 s širino snopa 100, ki je dosegla omejitev treh ur delovanja).

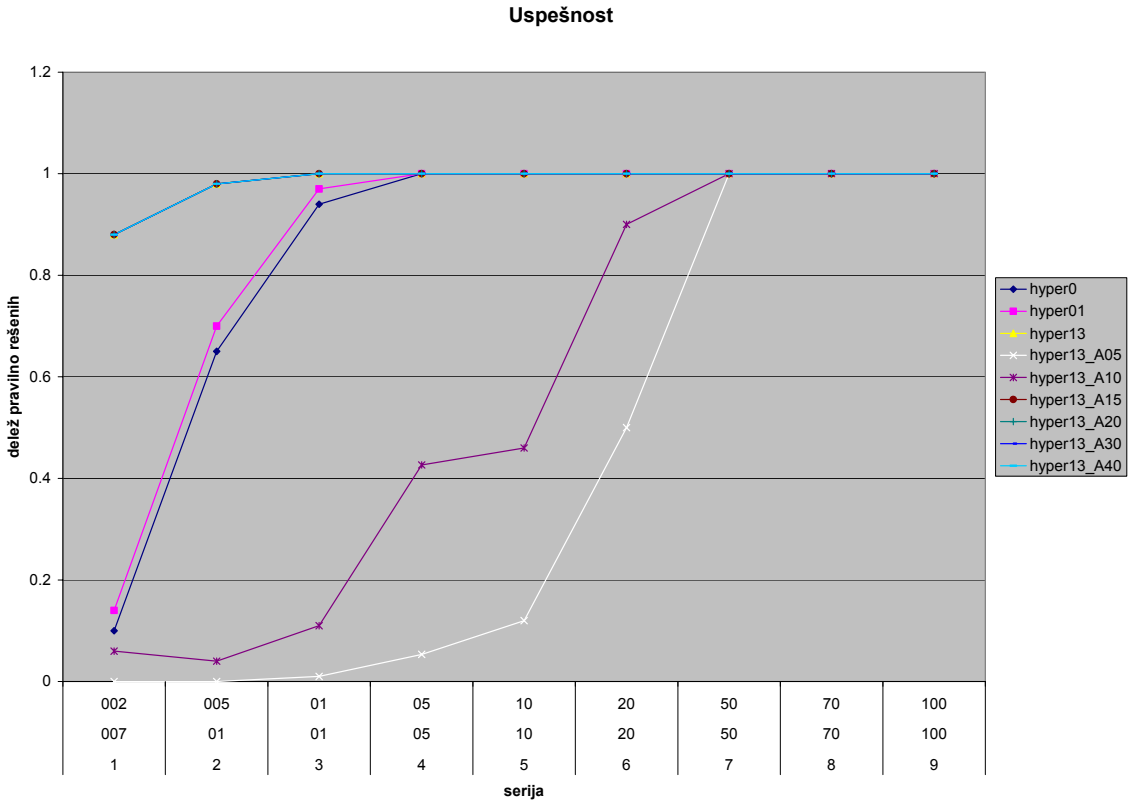
Iskanje s snopom sicer lahko poveča učinkovitost sistema, vendar občasno zahteva tako širok snop, da se porabljeni čas poveča preko meja sprejemljivosti.



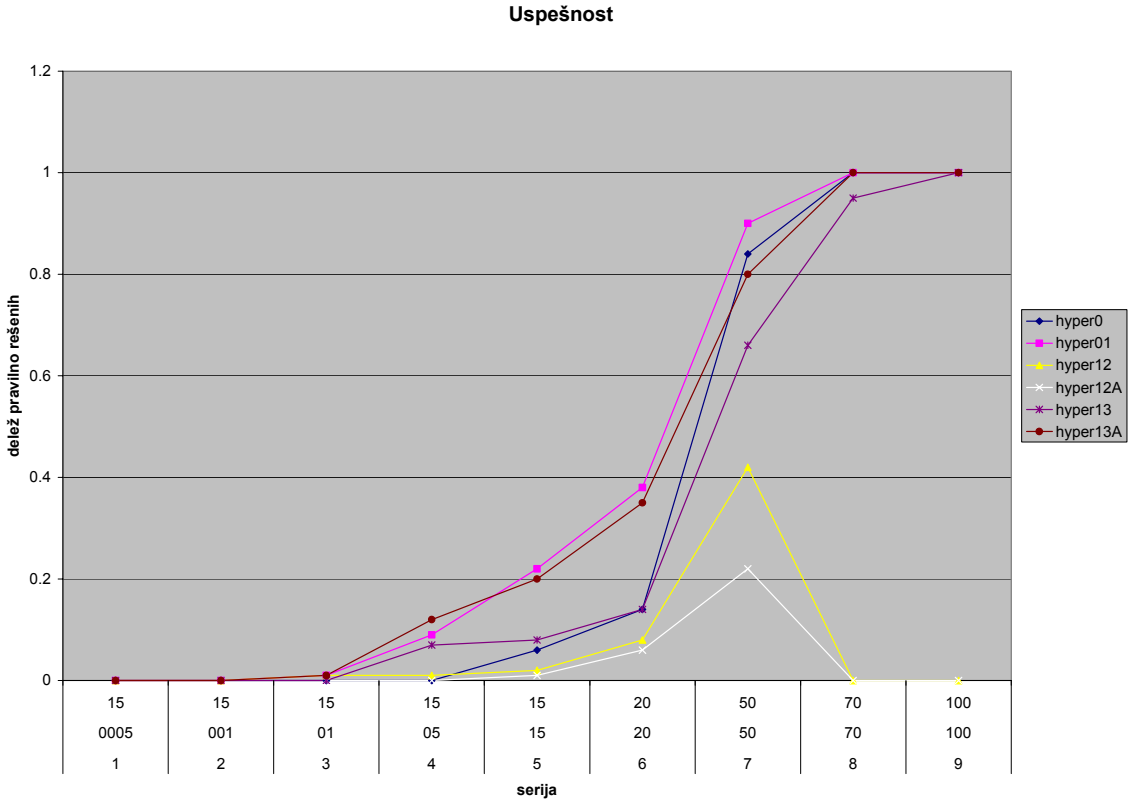
Sl. 4.20 Uspešnost nekaterih verzij sistema HYPER^2 na domeni member



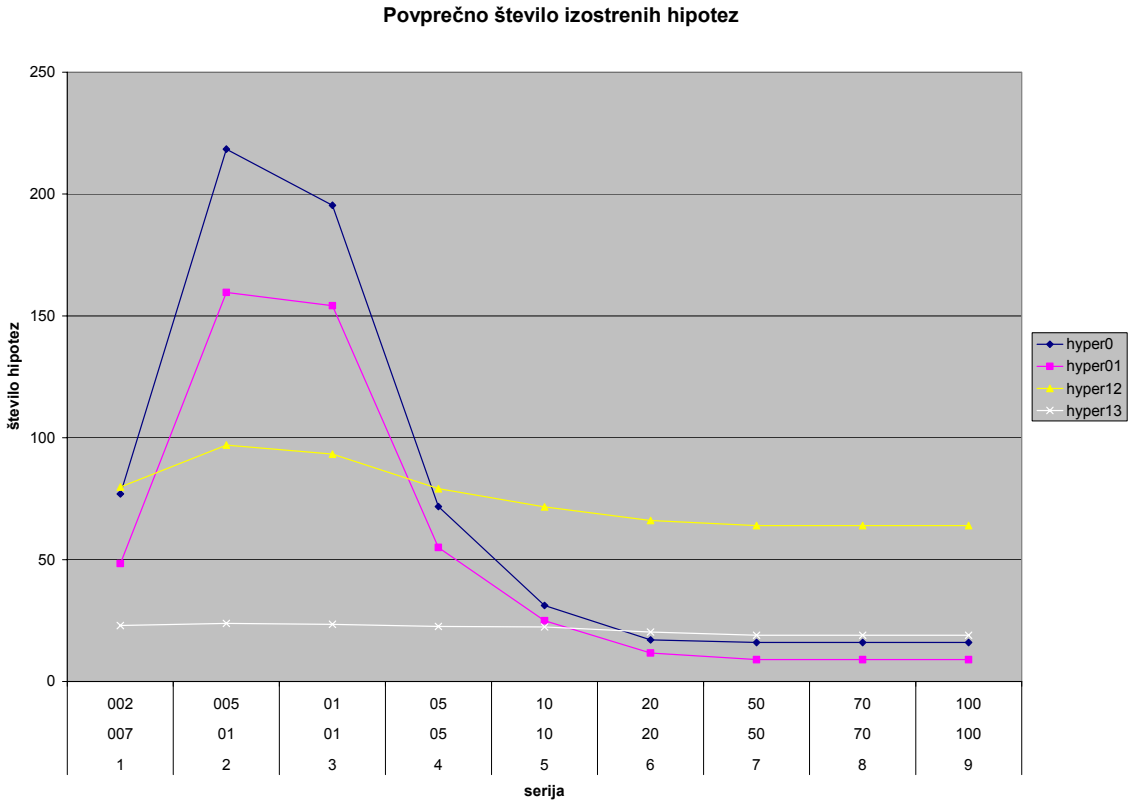
Sl. 4.21 Uspešnost nekaterih verzij sistema HYPER^2 glede na širina snopa na domeni member



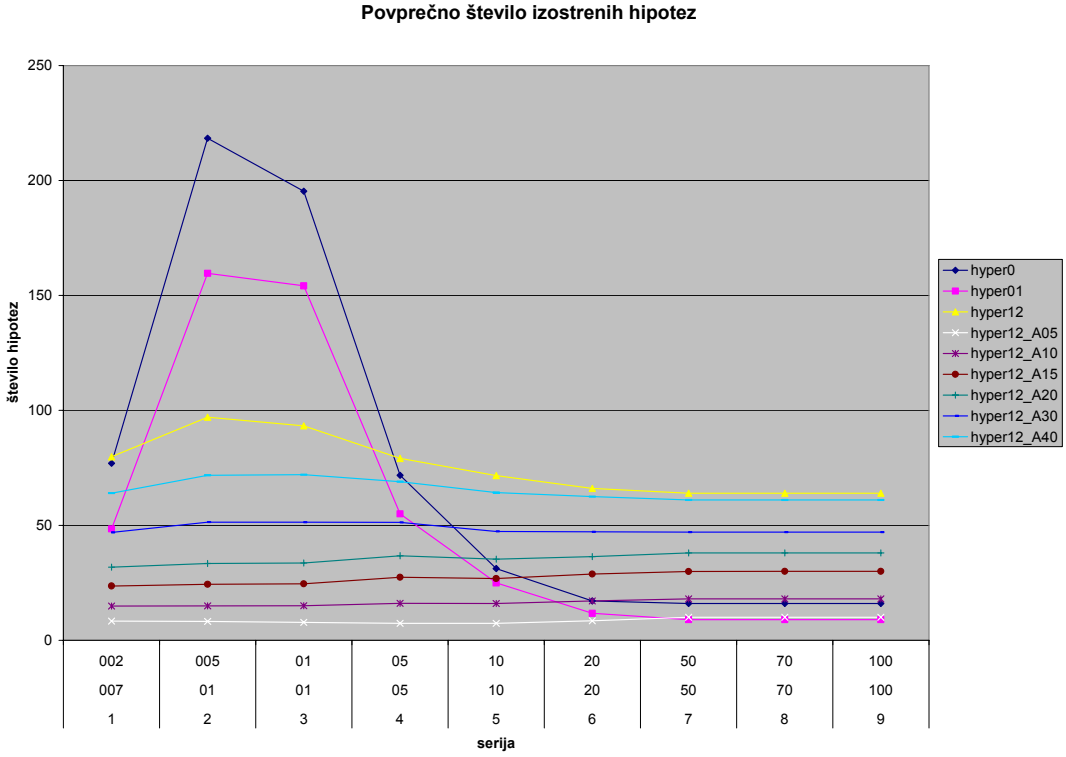
Sl. 4.22 Uspešnost nekaterih verzij sistema HYPER^2 glede na širina snopa na domeni member



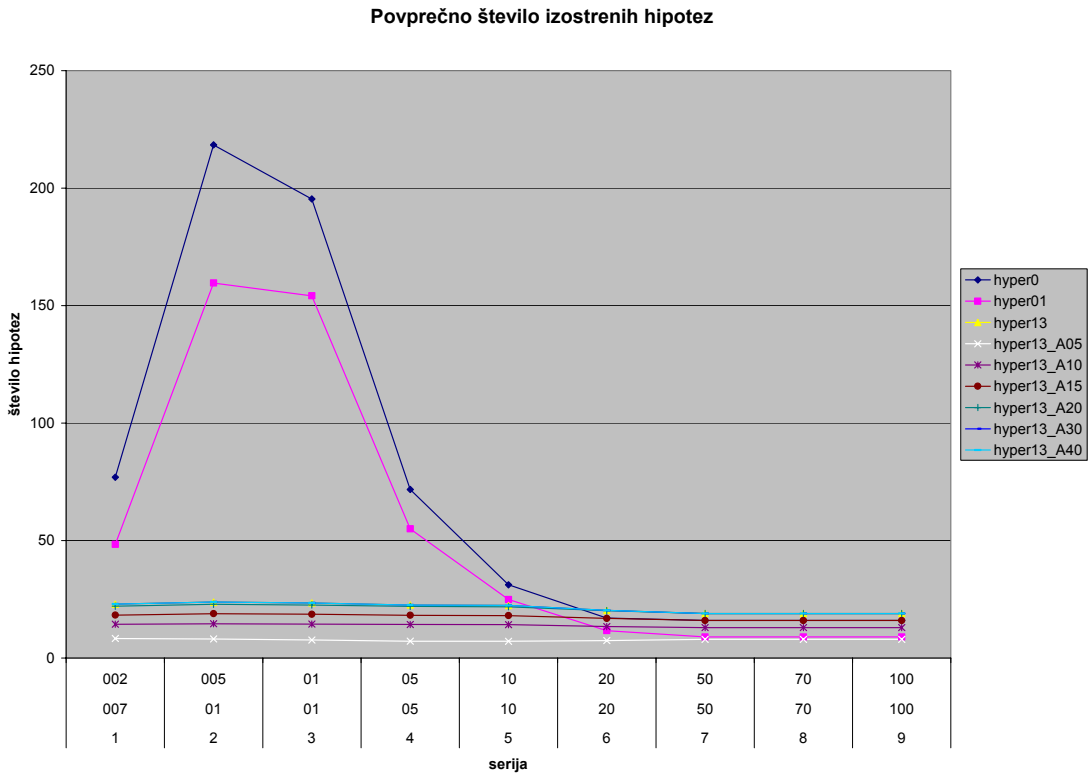
Sl. 4.23 Uspešnost nekaterih verzij sistema HYPER^2 na domeni path



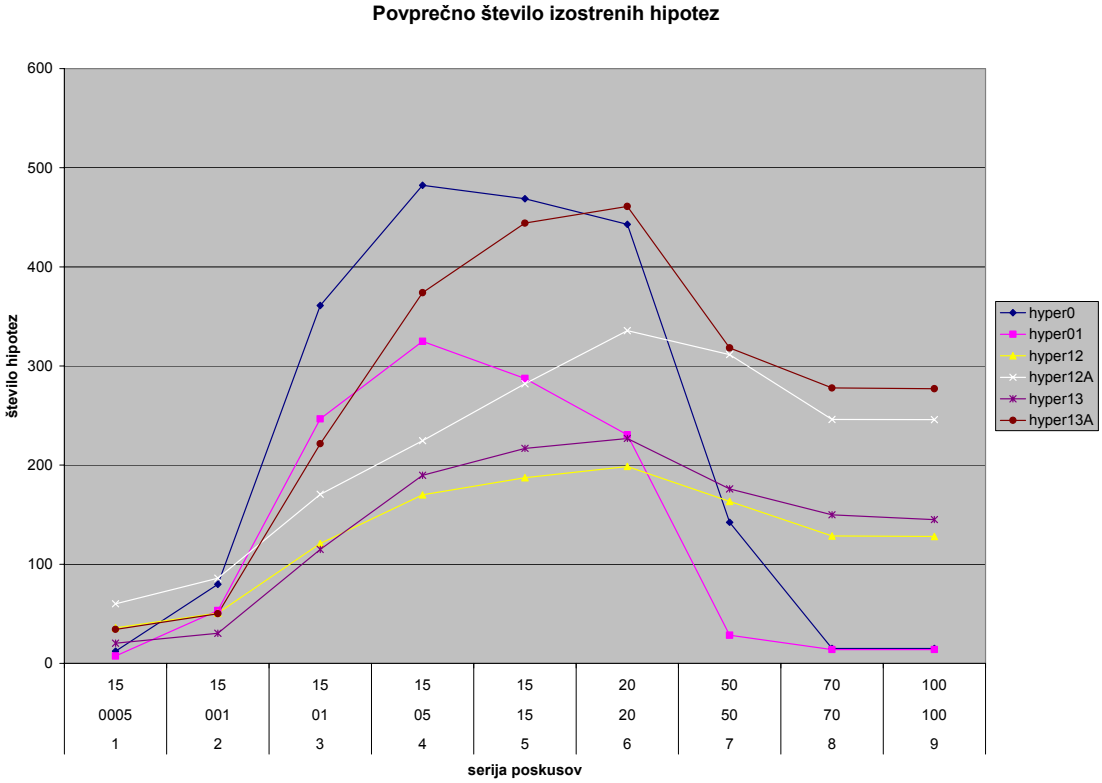
Sl. 4.24 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 na domeni member



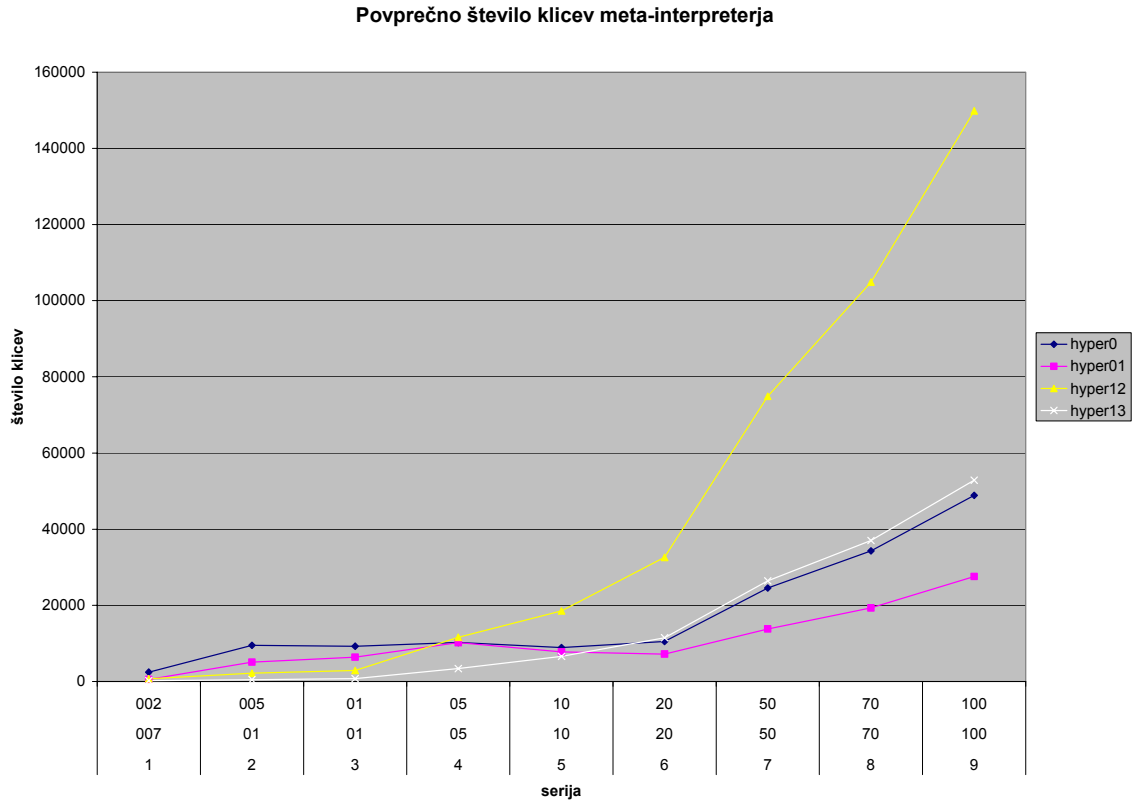
Sl. 4.25 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 glede na širino snopa na domeni member



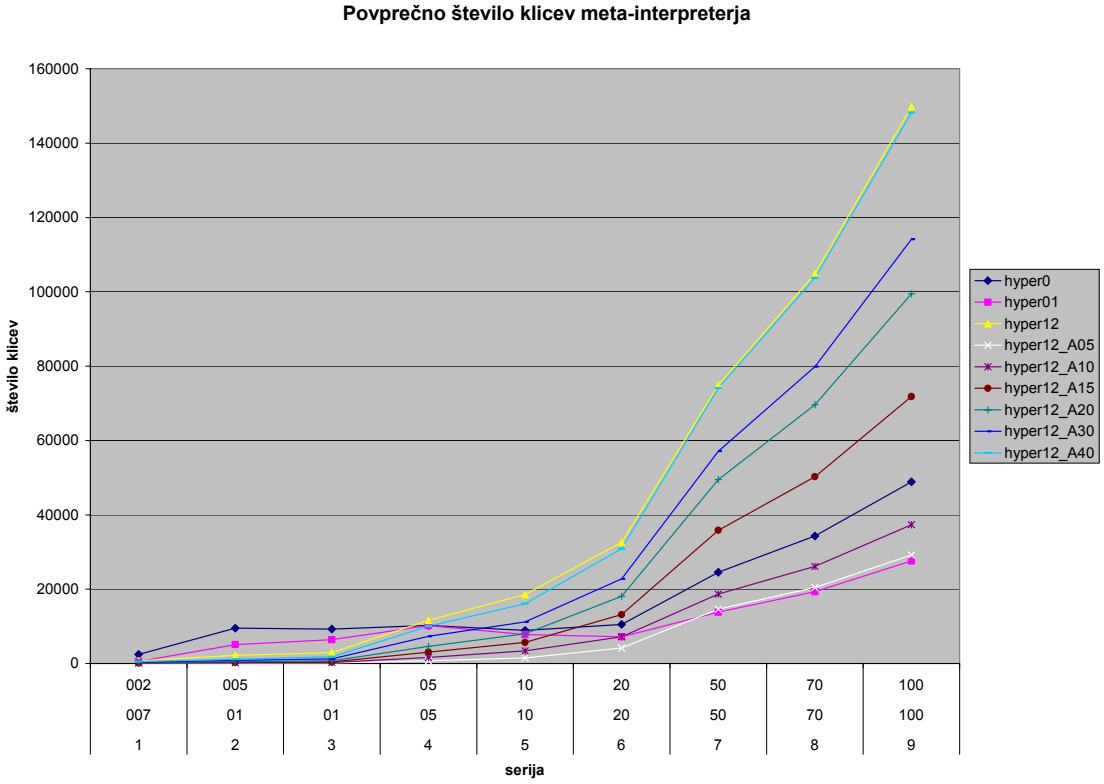
Sl. 4.26 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 glede na širino snopa na domeni member



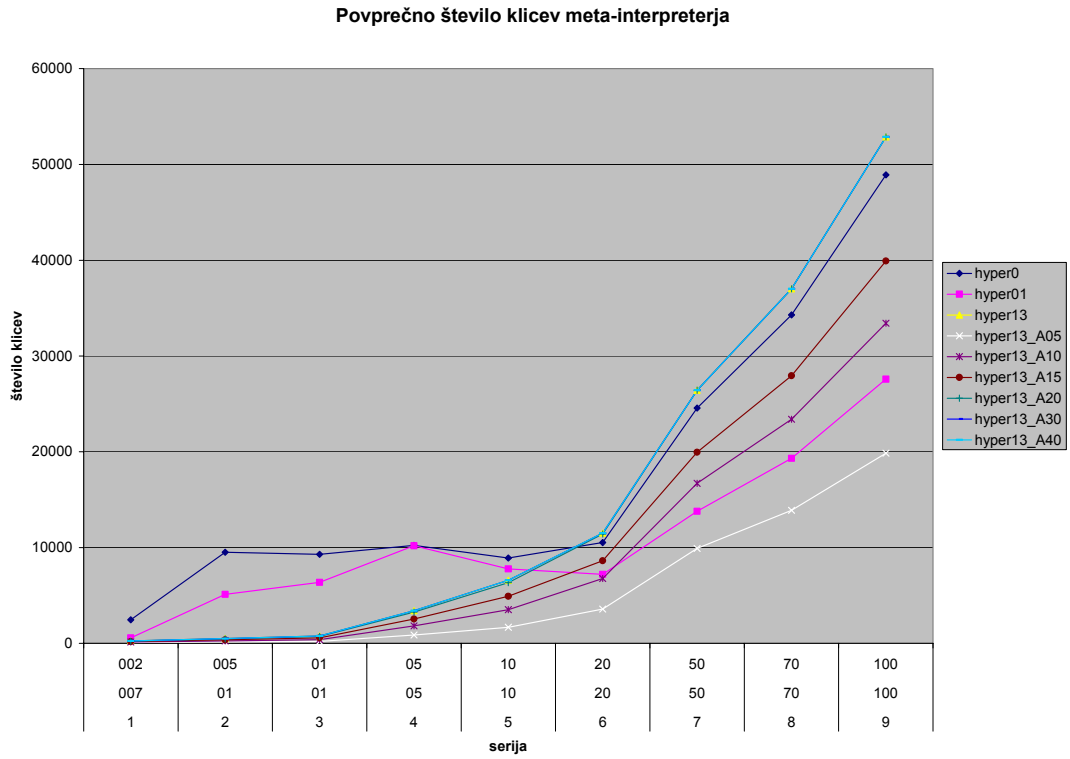
Sl. 4.27 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 na domeni path



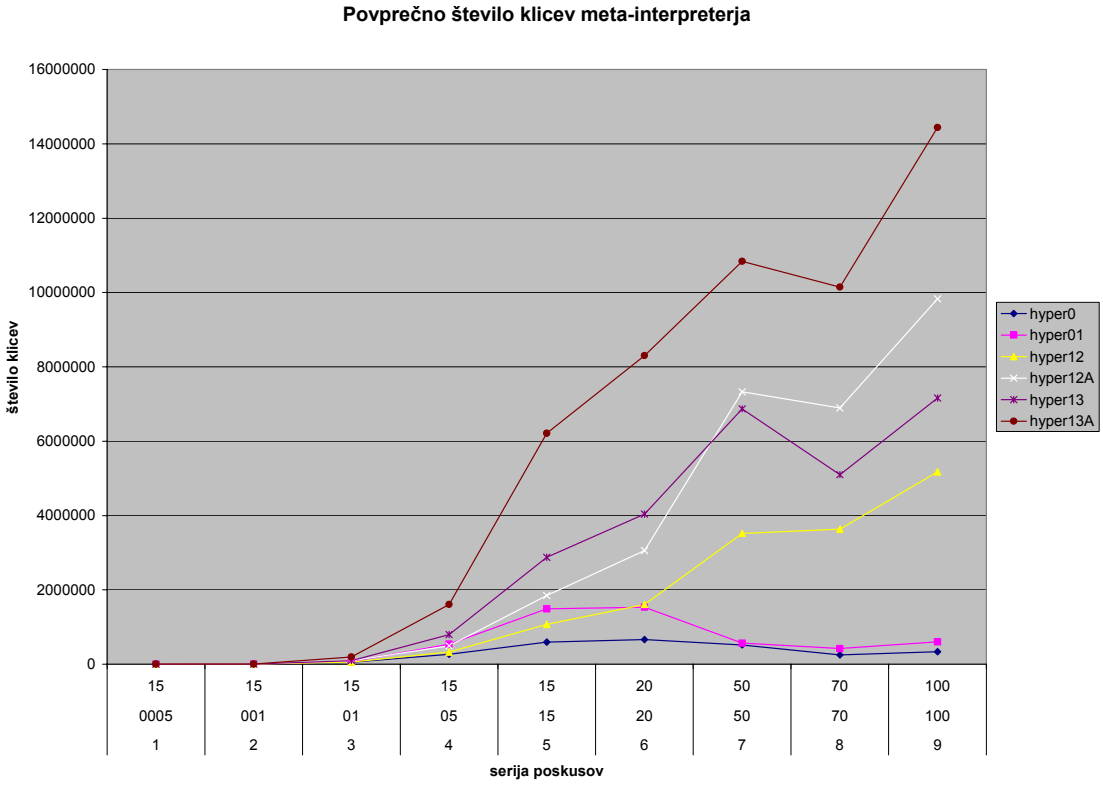
Sl. 4.28 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 na domeni member



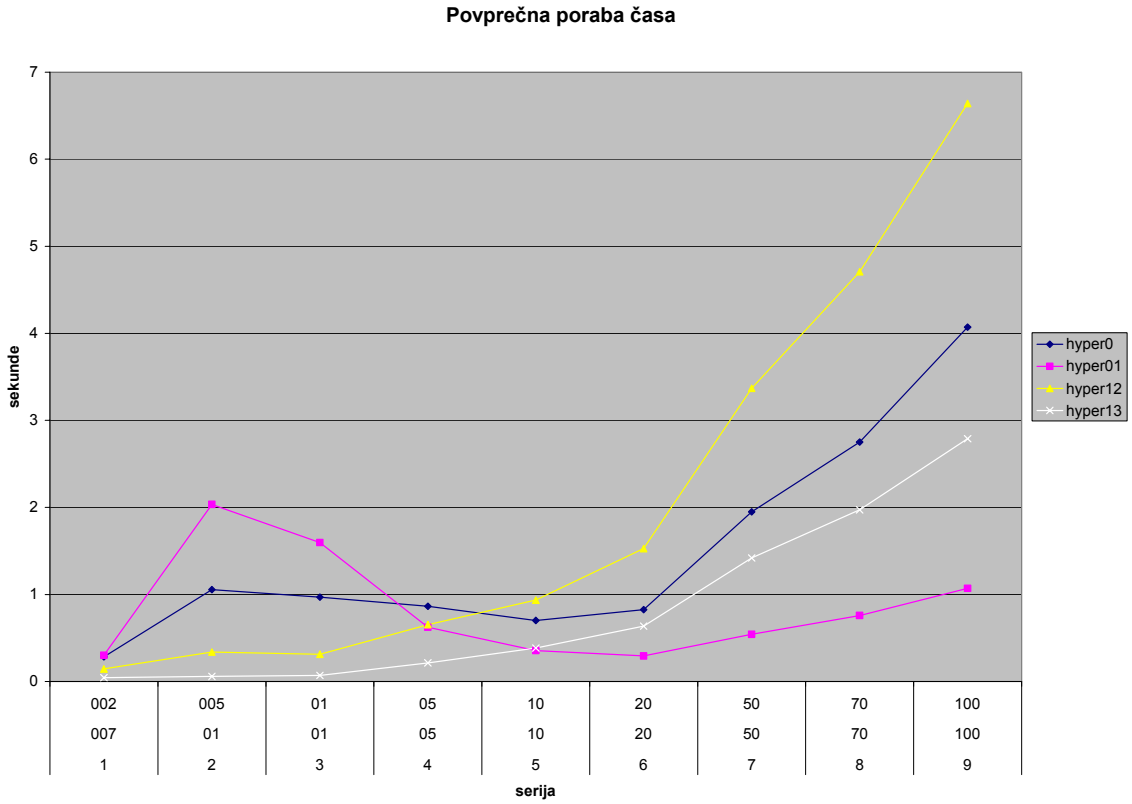
Sl. 4.29 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 glede na širino snopa na domeni member



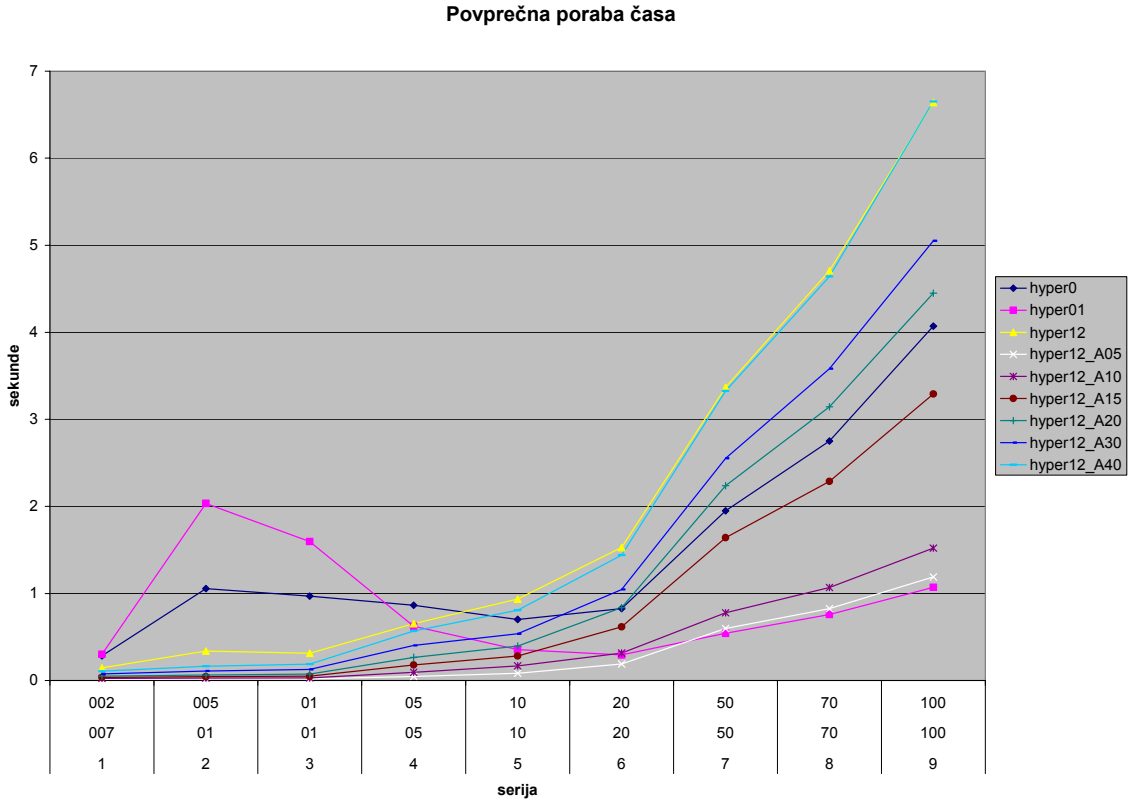
Sl. 4.30 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 glede na širino snopa na domeni member



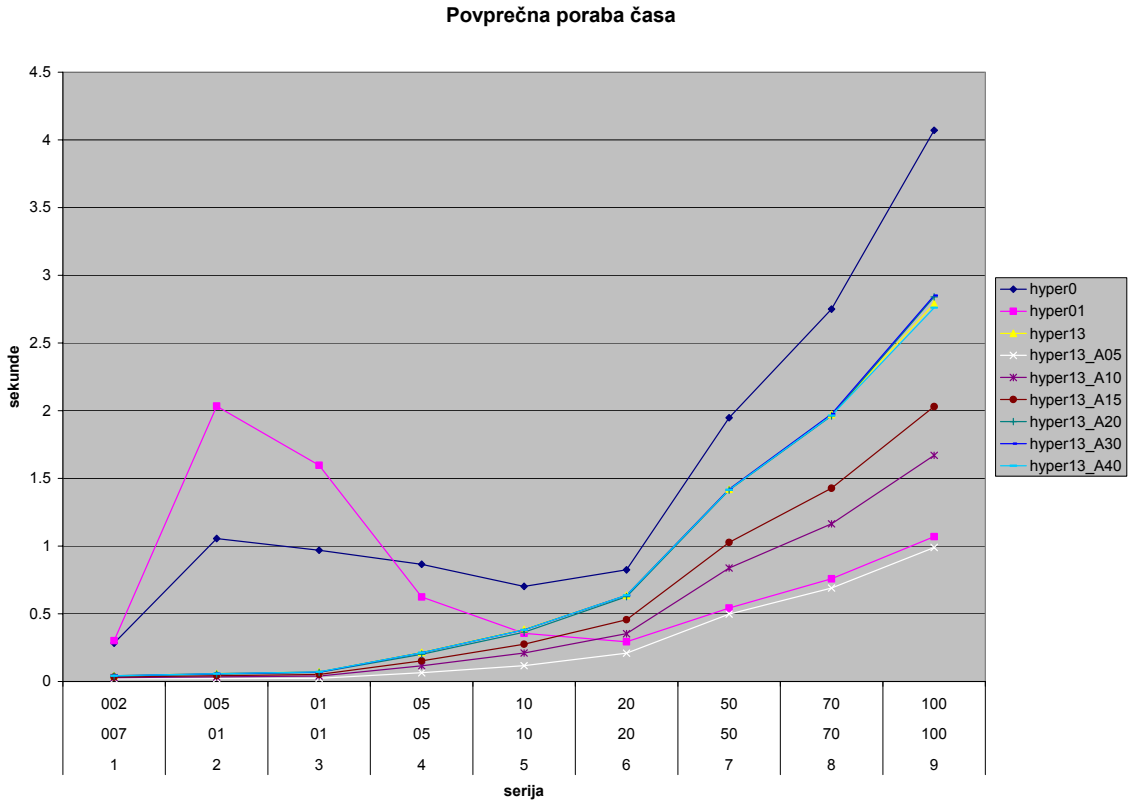
Sl. 4.31 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 na domeni path



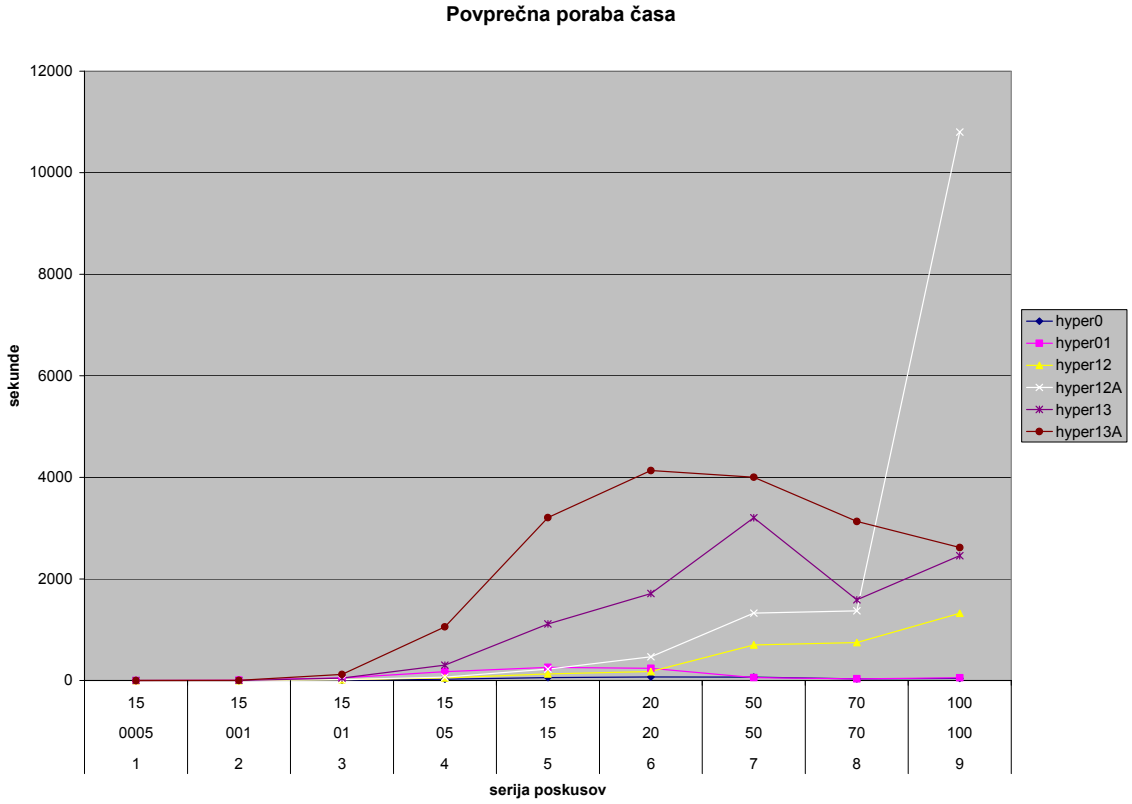
Sl. 4.32 Povprečna poraba časa nekaterih verzij sistema HYPER2 na domeni member



Sl. 4.33 Povprečna poraba časa nekaterih verzij sistema HYPER^2 glede na širino snopa na domeni member



Sl. 4.34 Povprečna poraba časa nekaterih verzij sistema HYPER^2 glede na širino snopa na domeni member



Sl. 4.35 Povprečna poraba časa nekaterih verzij sistema HYPER^2 na domeni path

4.4 Skupina D

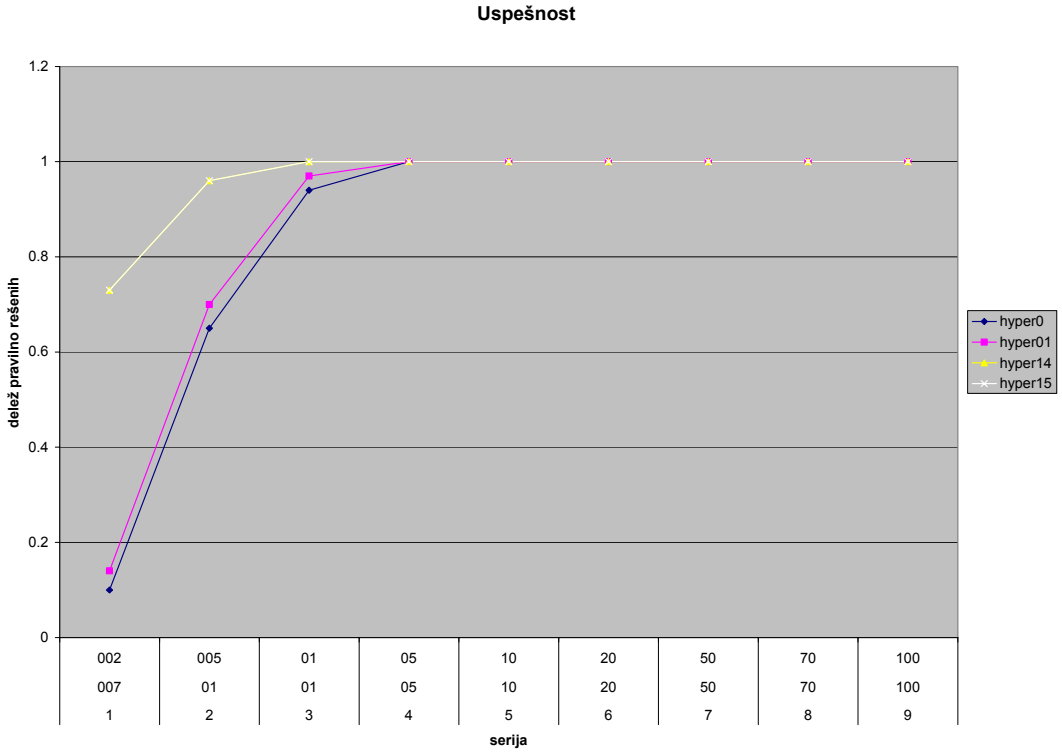
Verzije v tej skupini so predvsem rezultat prejšnjih meritev. Ker smo opazili, kako se je pri iskanju s snopom povečala učinkovitost in tudi druge performanse, ko smo v sistem dodali uporabo informacije o vhodno/izhodnem tipu spremenljivk v glavah stavkov iskane hipoteze, smo podobno, kot smo spremenili hyper12, da smo dobili hyper13, spremenili tudi hyper01, da smo dobili hyper14. Poleg tega smo dodatno optimizirali tudi nekatere, sicer časovno zahtevne operacije (kot je recimo iskanje kopij stavkov s tem, da se najprej hitro preveri enakost dolžine stavka). Glede na to, da se je izkazalo, da računanje pokritosti na nivoju stavkov ni prineslo zadovoljivega učinka, smo v verziji hyper15 vezali računanje pokritosti na hipoteze, vendar si v nasprotju s hyper07 in originalnih sistemom HYPER v tej verziji v okviru hipoteze zapomnimo, katere negativne učne primere pokriva hipoteza (vsaka hipoteza mora pokriti vse pozitivne učne primere, zato si teh ni potrebno posebej zapomniti).

Izkaže se, da smo na ta način izboljšali učinkovitost (na domeni member je učinkovitost malo manjša kot učinkovitost verzije hyper13, medtem ko je na domeni path večja). Pravzaprav sta obe tukaj testirani verziji po vseh kriterijih boljši od preostalih verzij sistema HYPER[2]. Na domeni member so razlike povsod občutne, ker je osnovna verzija napačno predvidevala, da so vse spremenljivke v glavah stavkov vhodne, čeprav je bil rekurzivni klic predznanja definiran z izhodno spremenljivko. Ta razlika privede do konfliktov, ki se jim verziji hyper14 in hyper15 izogneta. Takih konfliktov na domeni path ni bilo, zato na domeni path ni občutne izboljšave učinkovitosti, se pa pojavi občutna razlika v času izvajanja in številu klicev meta-interpreterja.

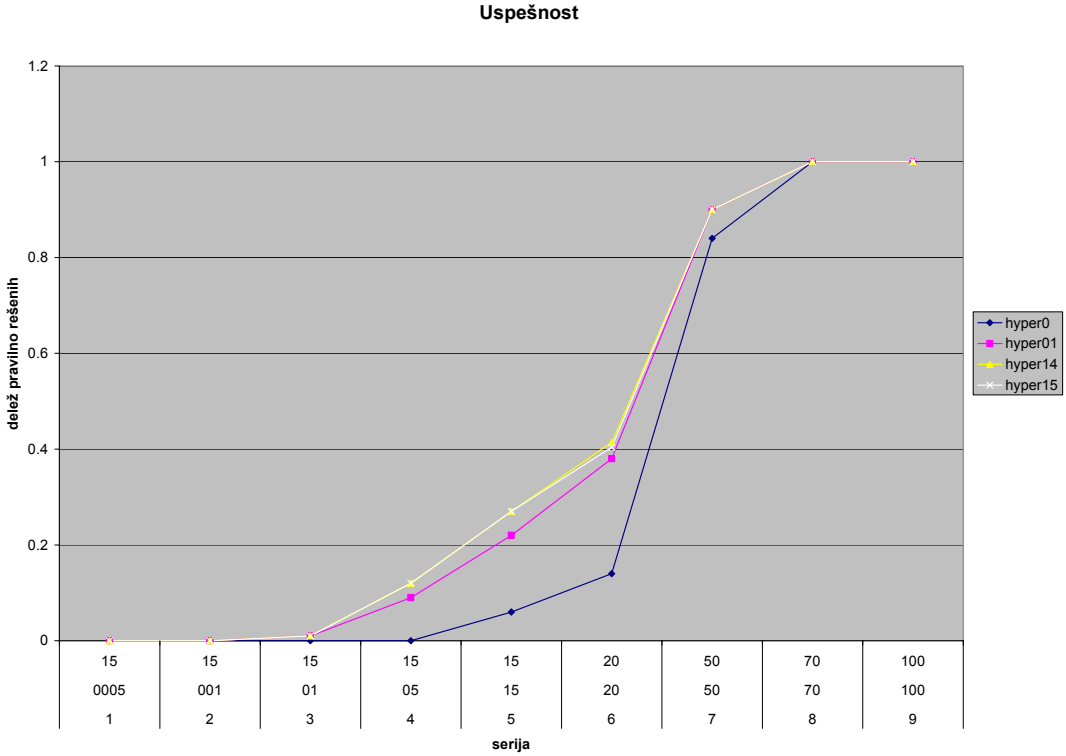
Opazne razlike med verzijama se pojavijo le pri številu klicev meta-interpreterja in porabi časa. Medtem ko je na domeni member malo boljša verzija hyper14 (tu pridejo do veljave pozitivni učinki računanja pokritosti na nivoju stavkov), je na domeni path občutno boljša verzija hyper15 (zaradi negativnih učinkov računanja pokritosti na nivoju stavkov).

Izkazalo se je, da informacija o vhodno/izhodnem tipu spremenljivk lahko prinese občutno prednost sistemu, ki jo uporablja (še posebej, kadar drugi sistemi napačno predvidevajo ta tip). Poleg tega je prinesla prednost tudi vrnitev na računanje pokritosti na nivoju hipotez (kot to počne originalni HYPER), s tem, da si v okviru hipotez zapomnimo, katere negativne primere pokrivajo. Zaradi tega je hyper15 pri vseh meritvah boljši (ali vsaj primerljiv) od ostalih verzij, še posebno, če upoštevamo, da je originalni HYPER pri serijah z manj primeri

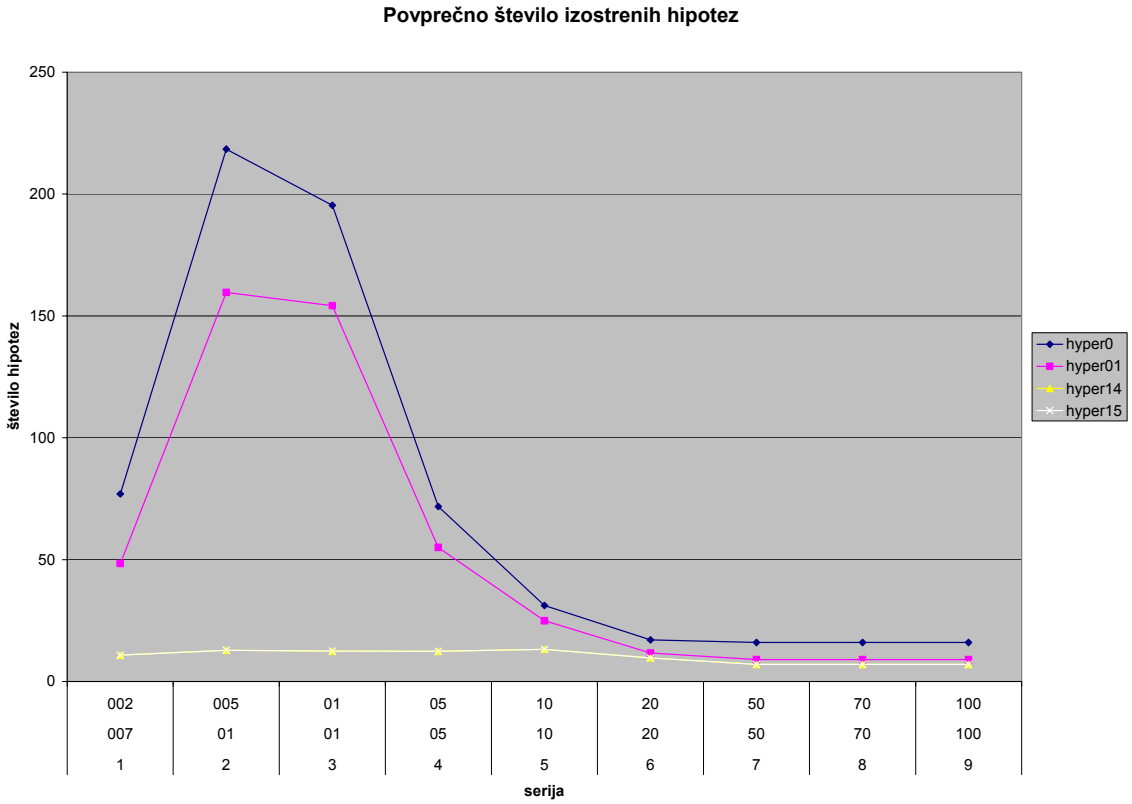
v domeni path hitrejši, ker preiskuje prostor možnih hipotez na enostaven način in ima zato manjšo možnost uspeha.



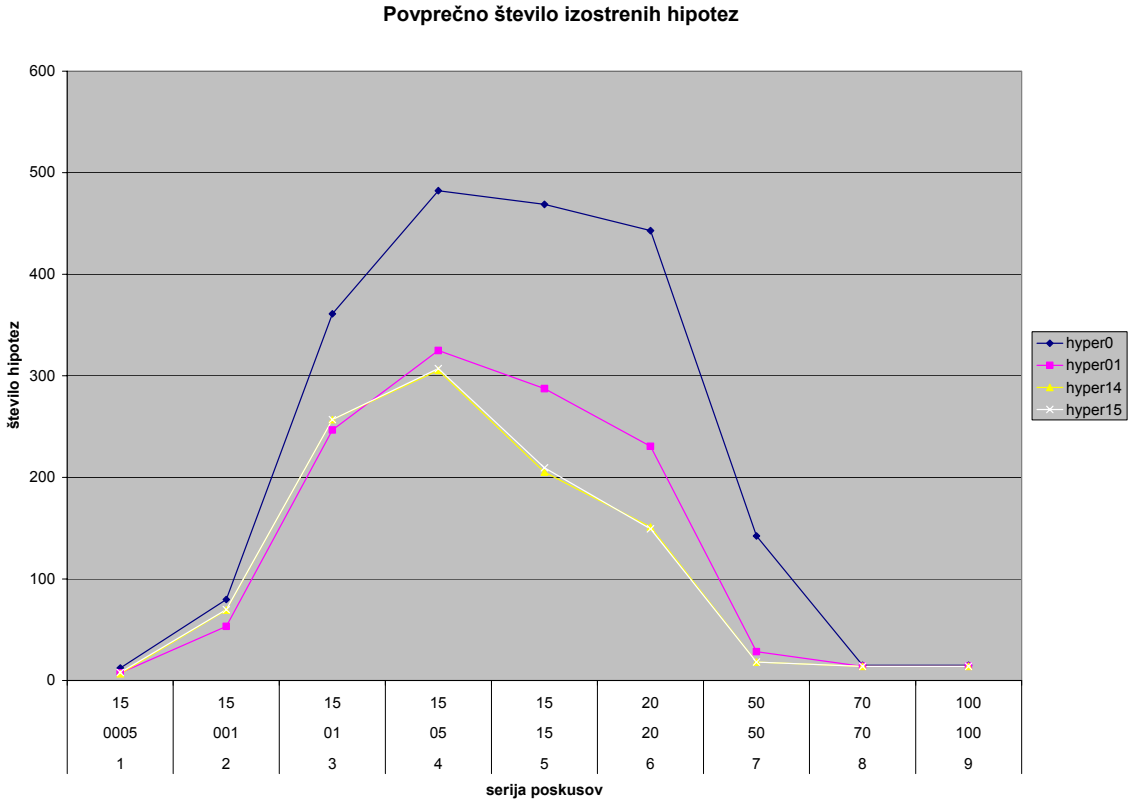
Sl. 4.36 Uspešnost nekaterih verzij sistema HYPER^2 na domeni member



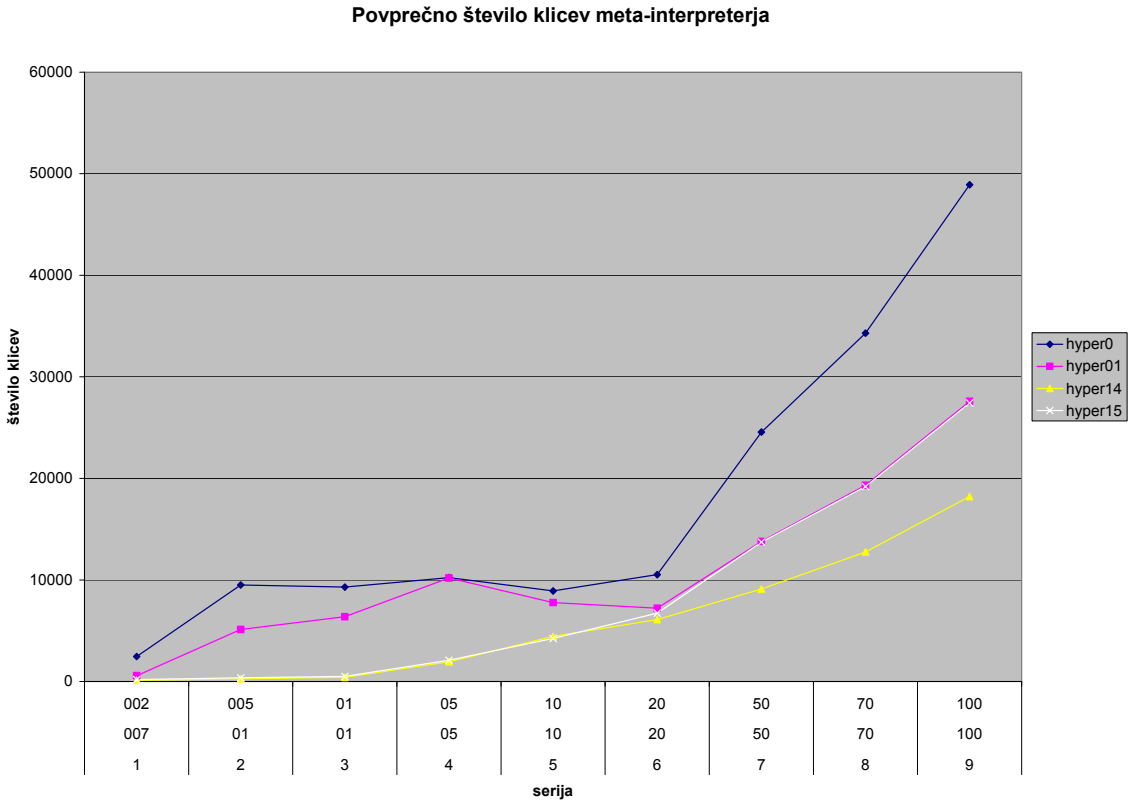
Sl. 4.37 Uspešnost nekaterih verzij sistema HYPER^2 na domeni path



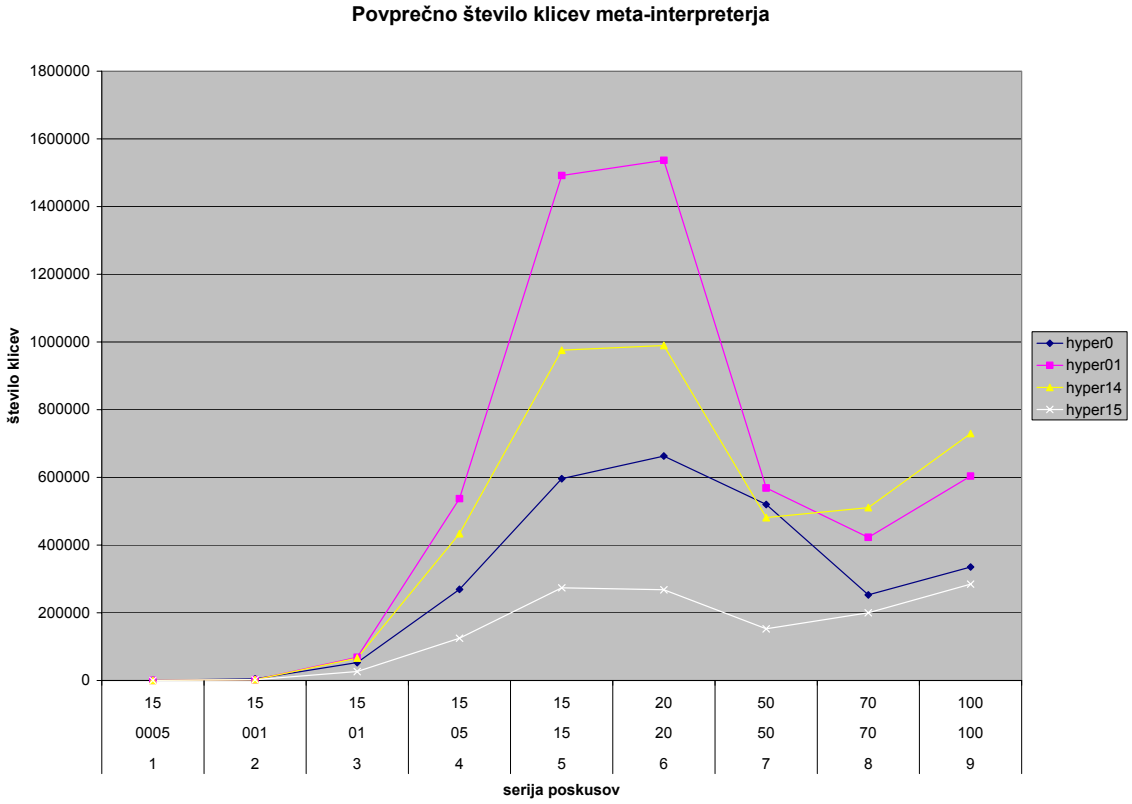
Sl. 4.38 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 na domeni member



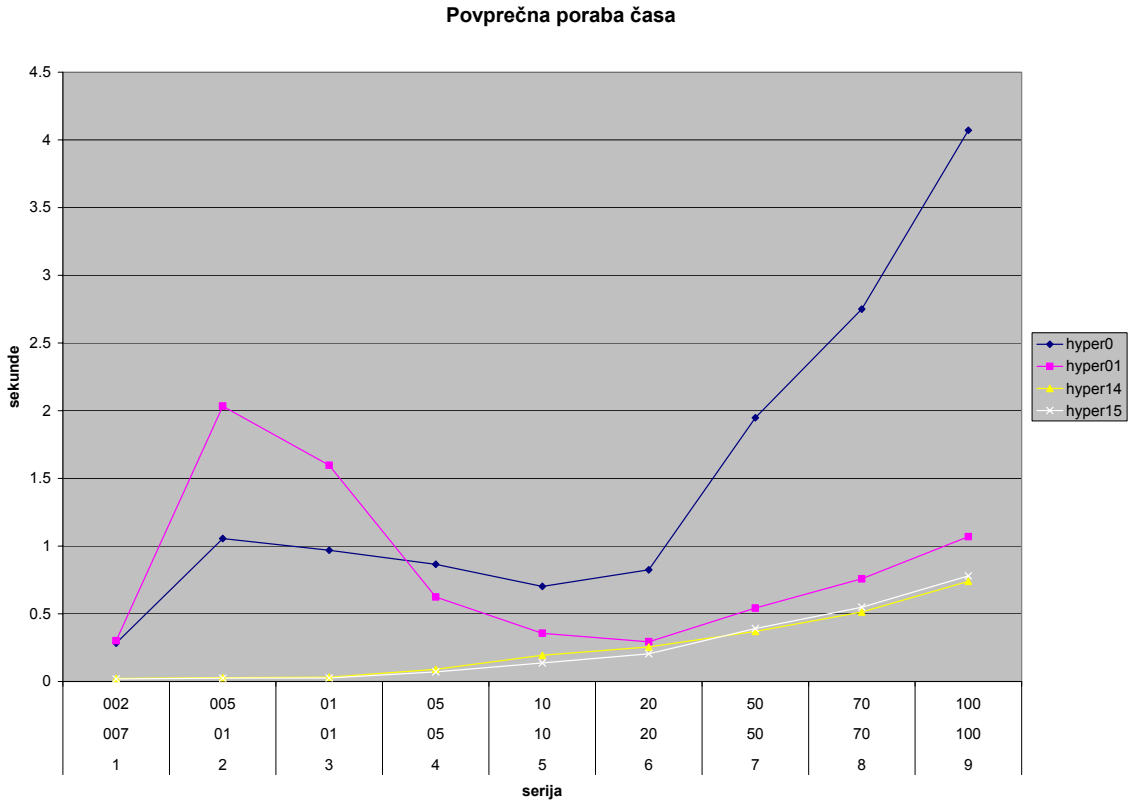
Sl. 4.39 Povprečno število izostrenih hipotez nekaterih verzij sistema HYPER^2 na domeni path



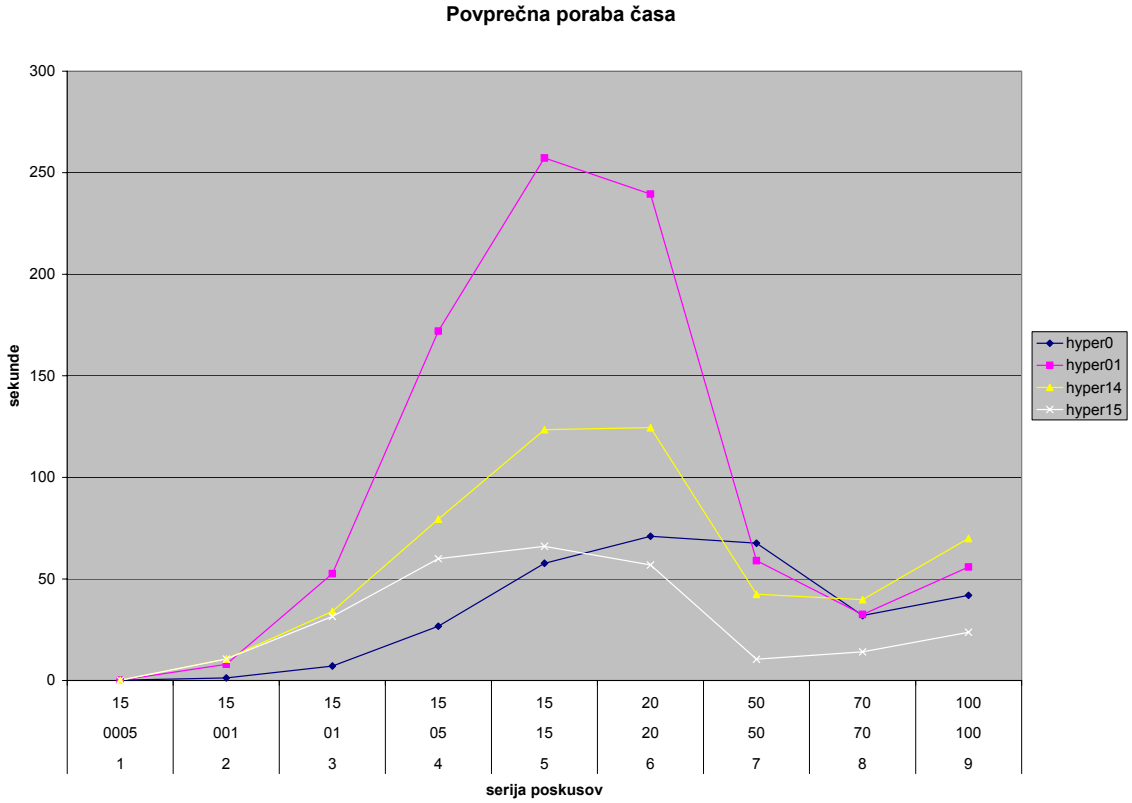
Sl. 4.40 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 na domeni member



Sl. 4.41 Povprečno število klicev meta-interpreterja nekaterih verzij sistema HYPER^2 na domeni path



Sl. 4.42 Povprečna poraba časa nekaterih verzij sistema HYPER^2 na domeni member



Sl. 4.43 Povprečna poraba časa nekaterih verzij sistema HYPER^2 na domeni path

4.5 Zaključek

V tem poglavju smo primerjalno testirali različne verzije sistema HYPER^2, ki se primarno razlikujejo v vklopljenih oziroma izklopljenih izboljšavah. Pri tem smo za vsako verzijo merili uspešnost, potreben čas, število izostrenih hipotez in število klicev meta-interpreterja (meritve, ki vsebujejo čas, število hipotez in število klicev v odvisnosti od pravilnosti rezultata, so prikazane v dodatku A).

Ker se je pokazalo, da določene izboljšave (oziroma spremembe, za katere smo predvidevali, da bi lahko izboljšale delovanje sistema) niso prinesle izboljšanja, ampak so celo poslabšale delovanje v enem ali več vidikih, smo poskušali pomanjkljivosti izboljšati ali nadomestiti z drugačno rešitvijo. Na ta način so nastale zadnje verzije sistema HYPER^2. Najprej verzije skupine C, ki uporabljajo iskanje s snopom in nato (ko se je videlo, da določene spremembe v verziji hyper13 prinesejo več kot iskanje s snopom) tudi novejše verzije, ki tako kot prve uporabljajo iskanje najprej najboljši. Kot najboljša se je izkazala verzija, ki uporablja iskanje najprej najboljši, zna upoštevati informacije o vhodnem/izhodnem tipu spremenljivk v glavah stavkov hipoteze in si zapomni, katere (negativne) učne primere pokriva posamezna hipoteza.

Izkazalo pa se je, da bi bilo potrebno dodati še nekatere spremembe. Predvsem bi bilo potrebno sistem pospešiti in mu omogočiti delo z več učnimi primeri, za kar bi bilo potrebno HYPER^2 napisati v kakšnem drugem narečju jezika PROLOG, saj SICStus omogoča delo le z 256MB spomina, poleg tega pa ni med najhitrejšimi, ko je potrebno najti določeno dejstvo med množico podobnih. Oboje bi se dalo rešiti v kakšnem drugem programskem jeziku, a bi s tem lahko prišli do omejitev pri predznanju, ki bi ga sistem znal uporabljati.

Poleg tega bi bilo potrebno spremeniti način preiskovanja prostora možnih hipotez. HYPER^2 občasno zaide na napačno pot, s katere se ne zna vrniti. Ker take poti običajno vsebujejo stavke, ki ne pokrivajo nobenega pozitivnega učnega primera, bi lahko take stavke kaznovali, vendar bi bilo potrebno kazen določiti zelo natančno, da ne bi pretirano škodovali dejstvu, da HYPER^2 lahko poišče hipoteze tudi, ko ni pokrit celoten prostor učnih primerov (prav zato, ker dovolimo stavke, ki ne pokrivajo pozitivnih učnih primerov).

5 Primerjava sistema HYPER^2 z drugimi ILP sistemi

V tem poglavju bomo primerjali sistem HYPER^2 (nekaj najbolj uspešnih verzij) z nekaterimi drugimi ILP sistemi na domenah programske sinteze. Pri večini domen se bomo osredotočili na sposobnosti sistemov rešiti probleme iz pomanjkljive množice učnih podatkov. Zato bomo v domenah izvedli več serij poskusov, v vsaki z od 10 do 100 različnimi poskusi (en poskus je test vseh izbranih ILP sistemov na enem kompletu podatkov). Serije v domeni se bodo med seboj razlikovale po številu razpoložljivih učnih podatkov. Pri tem bomo opazovali, kakšen delež problemov v seriji posamezen sistem reši pravilno, koliko časa porabi za rešitev (v povprečju, v povprečju samo nepravilnih rešitev in v povprečju samo pravilnih rešitev) in kako veliko hipotezo je sistem vrnil (prav tako v povprečju, v povprečju samo nepravilnih rešitev in v povprečju samo pravilnih rešitev).

Sisteme smo testirali na naslednjih domenah:

- odd-even
- member
- memberA (od domene Member se razlikuje po vhodno-izhodnem tipu atributov)
- append
- last
- next
- path
- instertionsort
- quicksort

Ker nekatere sisteme poznamo bolje, kot druge, bi bolj poznani sistemi imeli prednost, če bi sisteme poskušali prilagoditi posameznim domenam. Iz tega razloga smo vse sisteme poganjali s privzetimi opcijami (le sistemu FOIL smo prepovedali negativne literale).

Poglejmo si najprej kratke opise preostalih ILP sistemov:

- ALEPH [33] Verzija 3 (25. 7. 2001) teče v Yap Prologu, za vhod dobi tri datoteke – eno z deklaracijami in definicijo predznanja, eno s pozitivnimi učnimi primeri in eno z negativnimi učnimi primeri. Sistem ne zna delati s seznami, zato je potrebno dodati

predznanje za delo s seznami (v obliki predikata, ki zna seznam spremeniti v glavo in rep, ali obratno). Sistem ne podpira učenja več predikatov hkrati. Pogosto je bilo potrebno dodati varovalen sistem, ki preprečuje klavzule, ki se zazankajo (A ← A). Ta varovalni sistem je v obliki rezalnega mehanizma (ki ga ALEPH omogoča) in odreže take klavzule (ta rezalni mehanizem je bil nastavljen s pomočjo avtorjev sistema). Sistem ne odstranjuje redundantnih klavzul iz teorije. Pogosto se zgodi, da v teoriji obstajajo podvojene klavzule. Če ne more pokriti vseh učnih primerov, doda nepokrite pozitivne učne primere v teorijo kot dejstva (torej v primeru neuspeha samo našteje pozitivne učne primere). Sistem deluje v štirih korakih:

1) Izbere primer, ki ga bo posplošil, če ni nobenega primera, se ustavi
2) Zgradi najbolj specifično klavzulo, ki pokrije izbrani primer. To je običajno t.i. spodnja klavzula, ki vsebuje veliko literalov. Postopek grajenja je podrobno opisan v [21]
3) Poišče bolj splošno klavzulo, ki pokriva izbran primer. Pregleduje klavzule, katerih literali so podmnožice literalov spodnje klavzule. Privzeta ocena je razlika med številom pokritih pozitivnih in negativnih učnih primerov (pri rekurzivnih klicih upošteva tudi že prej zgrajene klavzule). Doda najboljšo klavzulo v teorijo
4) Odstrani učne primere, ki so postali redundantni z dodatkom nove klavzule v teorijo.

- CHILLIN [34, 35] (Version 1.0 Alpha) je bil prvotno napisan za Quintus prolog. Ker tega komercialnega programa ni bilo na voljo, smo sistem s pomočjo originalnih avtorjev prevedli v SICstus narečje jezika prolog. Na žalost niti originalni avtorji nimajo več na voljo Quintus prologa, zato ni bilo možno preveriti, da se s prevodom ni spremenilo obnašanje sistema (prevedena verzija ne zna obdelovati več predikatov hkrati). Kot vhod vzame tri datoteke (definicije predznanja, pozitivni učni primeri in negativni učni primeri). Ne predznanju ne iskanemu predikatu ne določimo niti tipov argumentov niti ali so vhodni ali izhodni. Sistem zna sam obdelovati sezname, zato posebno predznanje ni potrebno. Sistem uporablja kombinacijo pristopov od zgoraj navzdol (ostrenje) in od spodaj navzgor (posploševanje), ko poskuša zgraditi teorijo, ki pokriva pozitivne učne primere in ne pokriva negativnih učnih primerov. Pri tem spet in spet zgošča teorijo (gostoto teorije meri z njeno sintaktično velikostjo).

Algoritem začne z najbolj specifično teorijo – to je z množico vseh pozitivnih primerov. Nato poskuša posplošiti trenutno teorijo s ciljem najti čim manjšo teorijo, ki še vedno pokriva vse pozitivne primere. To počne tako, da naključno izbere pare klavzul v teoriji, ki jih nato nadomesti z *lgg* para. Če po posplošenju klavzula pokriva negativne primere, se nato še izostri z algoritmom, podobnim FOIL-u. Če še to ni zadosti, se poskuša klavzula dodatno izostriti z uvedbo novih predikatov. V vsakem koraku CHILLIN pregleda več možnih posplošitev in izbere tisto, ki najbolj zmanjša teorijo. Sistem se zna učiti rekurzivne definicije in ima vdelane omejitve, ki preprečujejo neskončno rekurzijo. Vendar sistem lahko vrne rekurzivno teorijo, ki pokriva negativne primere. Sistem v primeru neuspeha ne vrne ničesar.

- CPROGOL [21, 22] (4.4) je samostojen program (ne teče v prologu). Kot vhod vzame tri datoteke (definicije in deklaracije predznanja in tipov, pozitivne učne primere ter negativne učne primere). Sistemu lahko povemo tudi, kako se obdeluje posamezne tipe argumentov, zato posebno predznanje za obdelavo seznamov ni potrebno. Sistem zna »hkrati« učiti se več predikatov, vendar to v resnici počne zaporedno. Najprej se nauči prvi predikat, nato naslednjega, itn., pri tem lahko uporablja učne primere za druge predikate in tudi že naučene definicije kot predznanje. Sistem uporablja prekrivni algoritem. Izbere primer, ki ga bo posplošil. Iz tega primera naredi spodnjo klavzulo, nato pa preiskuje mrežo, ki jo spodaj omejuje tako določena spodnja klavzula, zgoraj pa prazna klavzula. Preiskuje od splošnega proti specifičnemu s pomočjo algoritma A* [15], ki ga vodi ocena na osnovi ocene zgoščenosti teorije (vsak korak čimbolj zmanjša število bitov potrebnih za kodiranje teorije). V vsakem koraku prekrivnega algoritma iskanje najde klavzulo, ki najbolj zgosti trenutno teorijo, vendar ni nujno, da je končni rezultat najbolj zgoščena teorija, ki ustreza učnim podatkom. Sistem primere, ki jih ni mogel drugače pokriti, navede v teoriji kot dejstva (torej v primeru neuspeha samo našteje pozitivne učne primere).
- FOIL [29, 30] (6.4) je samostojen program, ki za vhod vzame datoteko, ki vsebuje tako pozitivne in negativne učne podatke kot tudi deklaracije in definicije predznanja (pri iskanem predikatu določimo samo tipe argumentov, ne pa tudi ali so vhodni ali izhodni, kot lahko določimo pri predikatih predznanja). Samo predznanje mora biti podano v obliki množic n-teric. Drugače rečeno, za vsako predznanje moramo našteti vse možne kombinacije (ali vsaj vse, ki se lahko pojavijo pri naših učnih podatkih). Sistem se zna naučiti več predikatov, vendar le zaporedno. FOIL uporablja prekrivni

algoritem. V vsakem koraku se indukcija posamezne klavzule začne z najbolj splošno klavzulo (s praznim telesom), ki jo postopno ostri z dodajanjem literalov. V vsakem koraku preveri vse smiselne literale – vse predikate predznanja in tudi sam iskani predikat, z vsemi možnimi kombinacijami spremenljivk (ki ustrezajo deklaracijam predznanja) in (posebej deklariranih) konstant. Prav tako se preverijo (ne)enakosti spremenljivk. Posebni pristopi preprečujejo možnost neskončnih rekurzivnih zank. Izmed kandidatov FOIL nato izbere najboljšega s pomočjo hevristike informacijskega prispevka in ga doda telesu klavzule (v vsakem trenutku ima FOIL v obdelavi samo eno kandidatno klavzulo, zaradi česar je zelo učinkovit, vendar lahko prezre pravilne rešitve). Dodajanje literalov se konča, ko klavzula preseže nastavljeno točnost (nastavljiva preko parametrov). Indukcija klavzul se ustavi, ko so pokriti vsi pozitivni učni primeri (ali večina, če ne more pokriti vseh). Nato v postprocesiranju FOIL odstrani redundantne klavzule in literale iz klavzul, ker pričakuje šumne podatke je pri tem konzervativen.

- MARKUS [13, 14] (V1.1) deluje znotraj prologa (SICstus, Quintus ali Arity) in pričakuje, da so deklaracije in definicije predznanja ter sami učni podatki že v prologovi bazi. MARKUS se ne zna učiti več medsebojno odvisnih predikatov. Za učenje uporablja prekrivni algoritem, ki vsak korak začne z najbolj splošno klavzulo in nato od tam preiskuje graf izostritev, kot ga določa ostrilni operator (le-ta je bil prevzet od sistema MIS [32], s tem da je bil ostrilni operator optimiziran – torej je graf izostritev brez podvojenih vozlišč). Graf izostritev se preiskuje z iterativnim poglabljanjem, dokler se ne najde klavzula, ki pokriva vsaj en do sedaj nepokrit pozitivni učni primer in nobenega negativnega učnega primera. Ta klavzula je nato dodana v hipotezo. Redundantne klavzule se odstranijo iz hipoteze.

  Ker MARKUS uporablja izčrpno iskanje, je zelo pomembno omejiti prostor možnih hipotez. To MARKUS doseže z omejitvami, ki so poleg deklaracij tipov argumentov predznanja tudi največje število literalov v klavzuli ter največja dovoljena strukturna globina argumentov v glavi predikatov. Poleg tega lahko uporabnik sam doda nekatere dodatne omejitve (da so že privzete omejitve dokaj močne, se pokaže v domeni next, kjer pravilna rešitev pade izven privzetega prostora možnih hipotez).

5.1 Domena odd – even

5.1.1 Opis domene

Celotna domena je sestavljena iz seznamov z največjo dolžino pet, s petimi različnimi elementi, brez ponavljanja elementov v seznamih. Domena opisuje sodost oziroma lihost dolžine seznamov. Za generiranje učnih primerov smo uporabili kodo, vidno na Sl. 5.1. Pozitivnih primerov za odd je 141, za even 185. Negativnih primerov za odd je 185 in za even 141. Poskusi so potekali s sistemi HYPER^2 (v tabelah in slikah razviden kot hyper, uporabljena je bila verzija, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, brez informacije o vhodno/izhodnih spremenljivkah v glavah stavkov – glede na to, da so vsi argumenti vhodni, se obnaša enako kot če bi uporabljali to informacijo z iskanjem najprej najboljši), Hyper^{2}Beam (HyperB – pokrivanje na nivoju stavkov, informacija o vhodno/izhodnih spremenljivkah in iskanje s snopom), FOIL 6.4 (foil) in CPROGOL 4.4 (progol). Sistemi ALEPH, CHILLIN in MARKUS ne znajo reševati večpredikatnih problemov. Možno bi jih bilo sicer prisiliti, tako da rešujejo dva podproblema in s podatki za drugi podproblem kot predznanjem (kaže, da se problema na podoben način avtomatsko lotita tudi FOIL in PROGOL). Vendar bi tak postopek lahko šteli med trike, za katere pa smo se odločili, da jih pri poskusih ne bomo uporabljali.

```
even([]).
even([A,B|C]):- even(C).
odd([A|B]):- even(B).
```

Sl. 5.1 Pravilna rešitev (ena izmed mnogih), ki je bila uporabljena za generiranje učnih primerov

Zgradili smo deset serij problemov z različnim številom primerov (tab. 5.1). V vsaki seriji (razen zadnji z vsemi primeri) je bilo opravljenih 100 poskusov z naključno izbranimi primeri (vsi sistemi so v isti ponovitvi dobili enake primere).

5.1.2 Rezultati

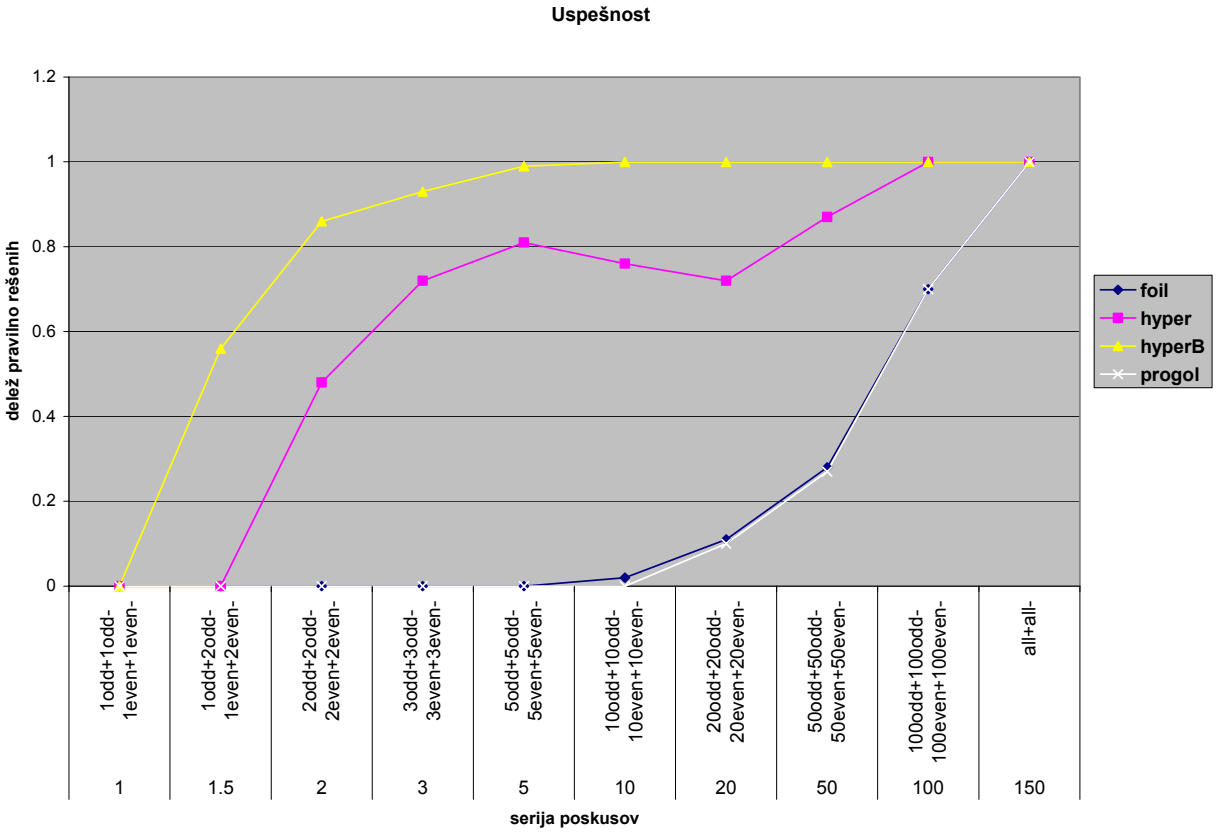
Primerjava uspešnosti sistemov na domeni odd-even (Sl. 5.2) pokaže veliko premoč sistema HYPER^2. To je razumljivo, kajti pravilna definicija mora vsebovati definicijo za sodost praznega seznama, kar pa se sistemi lahko naučijo le iz enega učnega primera. Ker pa so učni primeri v poskusih naključno izbrani, se ta učni primer z praznim seznamom pojavi bolj redko (vsaj v serijah z manj primeri). HYPER^2 ima tu občutno prednost, saj ko sestavlja celotne hipoteze, dopušča, da posamezen stavek ne prekriva nobenega učnega primera. Hipoteza, ki

vsebuje stavek, ki sam po sebi ne pokriva nobenega učnega primera, se (ker je cena hipoteze odvisna tako od dolžine kot od števila pokritih negativnih primerov) uporabi le, če tak stavek ugodno vpliva na delovanje celotne hipoteze. To se običajno zgodi (kot v tem primeru) v primeru rekurzivne definicije, kjer nimamo učnega primera za mejno vrednost rekurzije. Ostala sistema v tej primerjavi pa sestavljata hipoteze stavek po stavek in v takem primeru je nespametno v hipotezo vstaviti stavek, ki ne pokriva nobenega primera. Brez stavka, ki pokriva mejne primere, pa rekurzija ne deluje pravilno in se je pravzaprav sploh ni možno pravilno lotiti. Sistema FOIL in PROGOL v tem primeru poiščeta nadomestni mejni primer za rekurzijo. Rekurzijo ustavita na seznamih dolžine ena (ki so lihi) in v resnici pogosto prideta zelo blizu pravilni definiciji – najdena hipoteza pokriva vse primere domene, razen praznega seznama. HYPER^2 bi prav tako lahko kot mejni primer vzel lih seznam z dolžino ena, a izbere sod prazen seznam, ker je bolj enostaven (oziroma krajši).

Oznaka serije	Število pozitivnih primerov za odd	Število negativnih primerov za odd	Število pozitivnih primerov za even	Število negativnih primerov za even
1	1	1	1	1
1.5	1	2	1	2
2	2	2	2	2
3	3	3	3	3
5	5	5	5	5
10	10	10	10	10
20	20	20	20	20
50	50	50	50	50
100	100	100	100	100
All	141	185	185	141

tab. 5.1 Opis posameznih serij domene odd - even

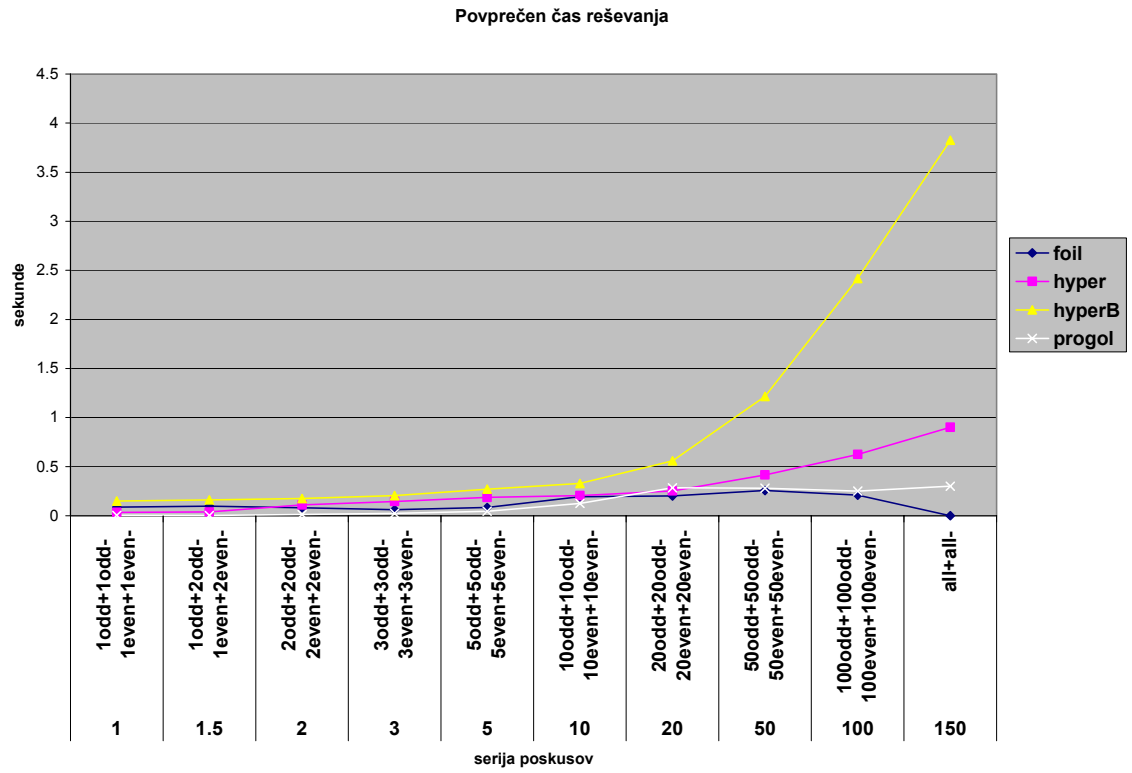
Ko primerjamo med seboj različni verziji sistema HYPER^2, je očitno in razumljivo, da je nekaj bolj uspešna verzija z iskanjem s snopom in tudi počasnejša, saj išče bolj v širino (največja dovoljena širina snopa je bila nastavljena na 50). Verzija z iskanjem najprej najboljši je malo manj uspešna kot verzija z iskanjem s snopom, je pa tudi (še posebno pri serijah z več učnimi primeri) občutno hitrejša. Padec natančnosti pri serijah 10 in 20 pa, kakor kaže, nastane zaradi nekaj več izrojenih učnih množic (na primer vsi pozitivni učni primeri zaeven so dolžine štiri), kot pri prejšnji seriji. A še vedno je pri večjem številu primerov približno še enkrat počasnejša od ostalih sistemov (Sl. 5.3, Sl. 5.4 in Sl. 5.5).



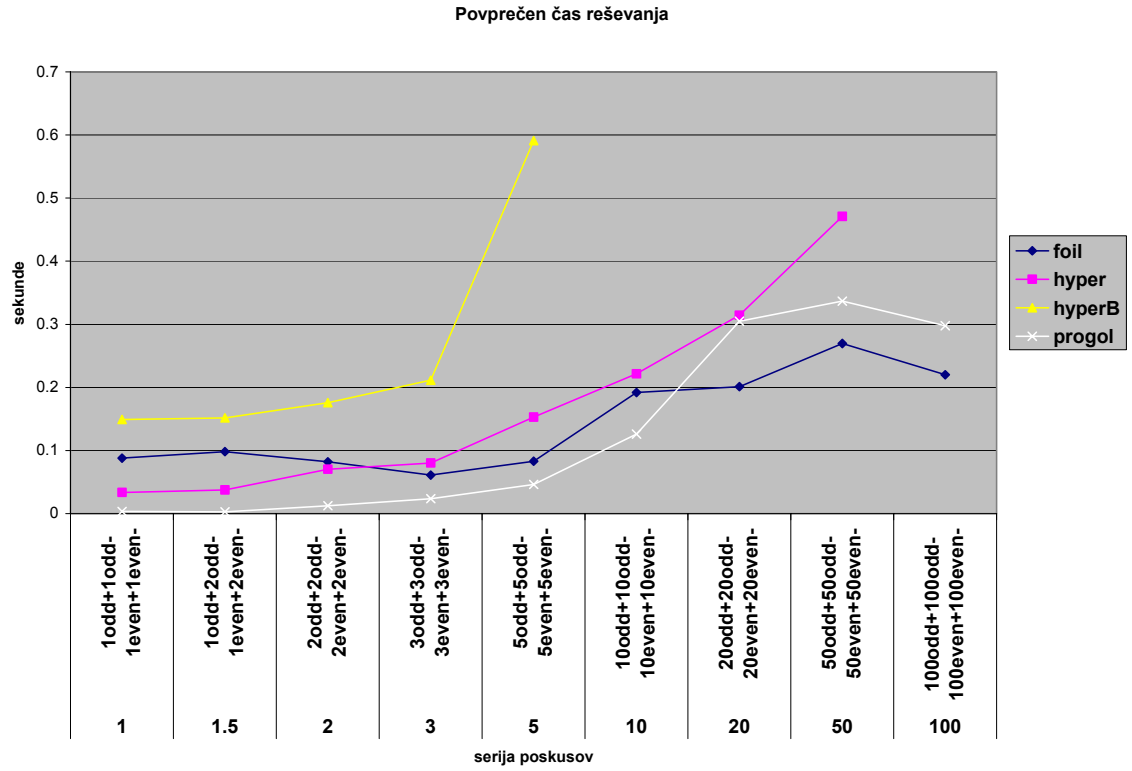
Sl. 5.2 Delež pravilno rešenih problemov

Če pogledamo Sl. 5.6, kjer je predstavljena uspešnost sistemov na posameznih problemih, opazimo veliko korelacijo med uspešnostjo sistemov FOIL in PROGOL. Po pregledu problemov, ki sta jih uspešno rešila, je bilo opaženo, da je minimalna zahteva, da FOIL in PROGOL rešita problem, da:

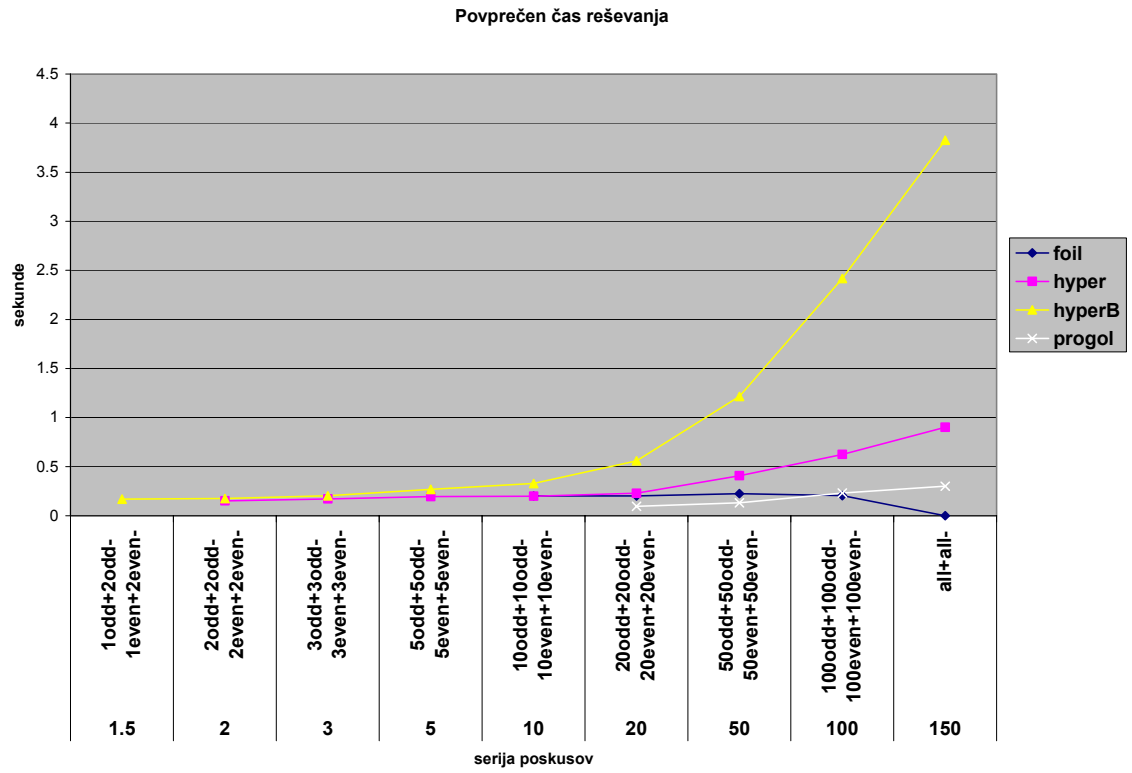
1. obstaja pozitivni učni primer za even([]) – mejni primer za rekurzijo
2. obstaja pozitivni učni primer za:
    a. even(L), tako da L = [A|L1] in obstaja učni primer za odd(L1) ali
    b. even(L), tako da L = [A,B|L1] in obstaja učni primer za even(L1)
3. obstaja pozitivni učni primer za:
    a. odd(L), tako da L = [A|L1] in obstaja učni primer za even(L1) ali
    b. odd(L), tako da L = [A,B|L1] in obstaja učni primer za odd(L1) in obstaja učni primer za odd([C])



Sl. 5.3 Povprečna poraba časa v posamezni seriji



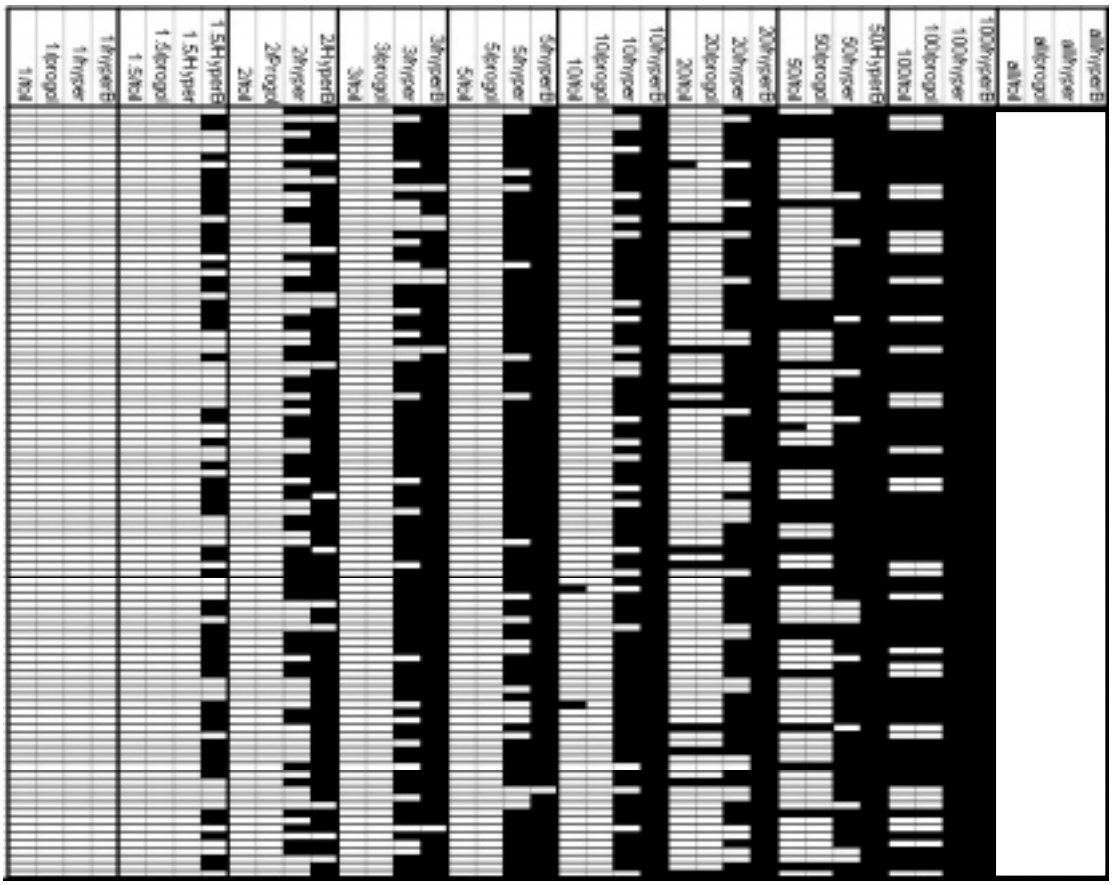
Sl. 5.4 Povprečna poraba časa v posamezni seriji, ko sistem ni pravilno rešil problema



Sl. 5.5 Povprečna poraba časa v posamezni seriji, ko je sistem pravilno rešil problem

Na slikah Sl. 5.7, Sl. 5.8 in Sl. 5.9 vidimo povprečno velikost induciranih hipotez. Šteli smo samo literale (vključno z glavo stavkov), nismo pa šteli literalov za manipulacijo seznamov, ki jih potrebuje FOIL (kot tudi ALEPH), saj ne zna obravnavati seznamov kot struktur (dela s strukturami različnih tipov lahko z deklaracijami naučimo PROGOL in HYPER, medtem ko zna MARKUS že sam delovati s seznami). Zaradi tega je bilo potrebno sistemu FOIL podati posebno predznanje za obdelavo seznamov. Klicev tega dodanega predznanja nismo šteli v velikost hipoteze.

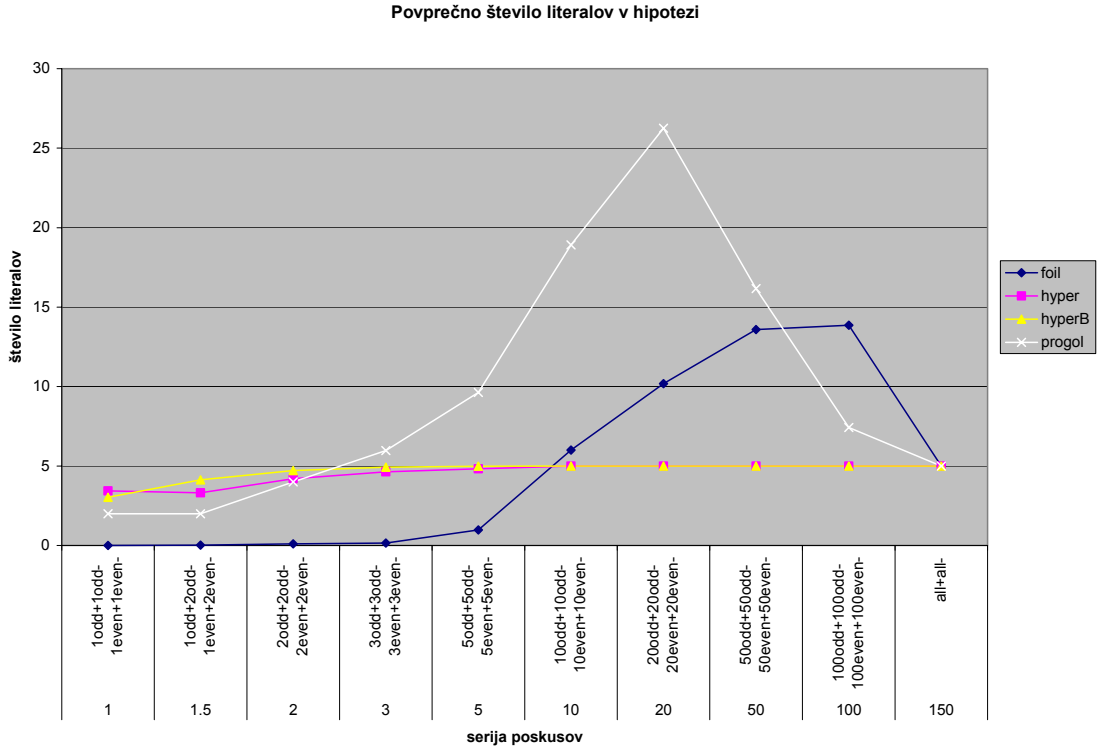
Ko pogledamo rezultate meritev velikosti induciranih hipotez, vidimo, da obe uporabljeni verziji sistema HYPER^2 inducirata hipoteze dokaj konstantnih velikosti. To je razumljivo, saj sistem HYPER^2 postopoma povečuje velikost hipoteze. Poleg tega je velikost hipoteze tudi del cene same hipoteze, kar pripelje HYPER^2 do ene manjših možnih hipotez, ki še ustrezajo podatkom. V nasprotju s HYPER^2 pa FOIL in predvsem PROGOL povečujeta velikost hipotez s številom pozitivnih učnih primerov. PROGOL v primeru, ko mu ne uspe inducirati primerne hipoteze, našteje pozitivne primere, ki mu jih ni uspelo pokriti. Ko FOIL in PROGOL začenjata pravilno reševati problem, se postopoma znižuje tudi velikost induciranih hipotez. Pri tem je bolj uspešen PROGOL, ki dokaj uspešno najde in izloči redundantne stavke in s tem zmanjšuje velikost hipoteze. Tako FOIL kot tudi PROGOL konsistentno inducirata minimalno hipotezo šele, ko imata na voljo vse možne učne primere.



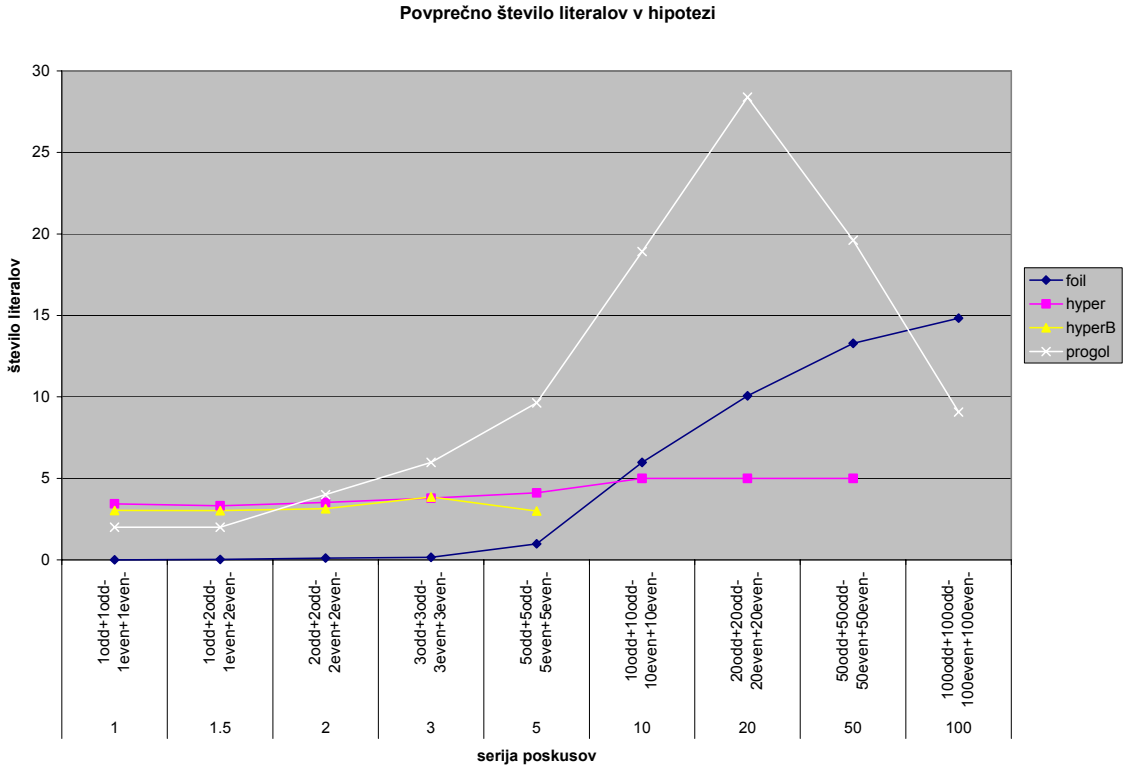
Sl. 5.6 Primerjava pravilno rešenih problemov (pravilno rešeni problemi so predstavljeni s črno celico, nepravilno rešeni pa z belo. Na vodoravni osi so serije in sistemi, na navpični pa problemi v seriji. Zadnji stolpci imajo le po eno črno celico, ker je v zadnji seriji le en problem)

Kot je razvidno v tej in preostalih domenah, velikost pravilnih hipotez pri sistemu FOIL najprej zelo naraste, nato pa (nekako v isti točki, kjer začne uspešnost strmo naraščati) velikost hipotez pade proti optimalni. Razlaga za to ugotovitev je, da FOIL za pravilno definicijo rekurzivnega predikata potrebuje med pozitivnimi učnimi primeri tudi sam primer rekurzivnega klica (na primer da rekurzivno reši primer odd([a,b,c]), potrebuje tudi even([b,c])), torej je za pravilnost definicije zelo odvisen od polnosti učne množice. Ko le-ta ni polna, pogosto poskuša poiskati še dodatne rešitve, le-te prinesejo k velikosti hipoteze in tudi povečajo verjetnost napake v hipotezi (tako se zgodi, da hipoteza vsebuje pravilne klavzule, vendar tudi nekaj dodatnih, nepravilnih). Za primer podamo tipično, veliko, a pravilno hipotezo, ki jo je FOIL vrnil v enem izmed poskusov v seriji 50 (literale za delo z seznami smo zamenjali s izrazi):

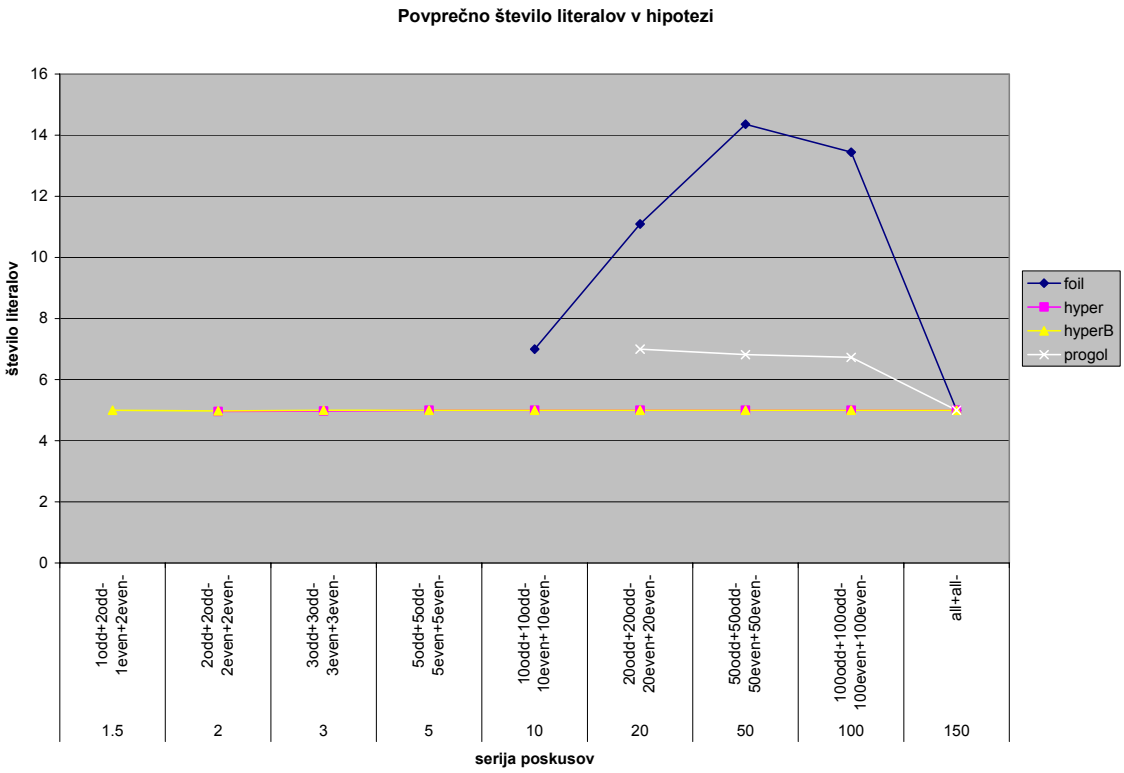
```
odd([B|C]) :- even(C).
odd([B,D,F|G]) :- even(G).
odd([B,D|E]) :- even([B|E]).
odd([B,D|E]) :- odd(E).
odd([B,D,F|G]) :- odd([D|G]).

even([]).
even([B|C]) :- odd(C).
even([B,D,F|G]) :- odd(G).
even([B,D|E]) :- even(E).
even([B,D,F|G]) :- even([D|G]).
```



Sl. 5.7 Povprečno število literalov v hipotezi



Sl. 5.8 Povprečno število literalov v hipotezi, ko sistem ni pravilno rešil problema



Sl. 5.9 Povprečno število literalov v hipotezi, ko je sistem pravilno rešil problem

5.2 Domena member

5.2.1 Opis domene

Celotna domena je sestavljena iz seznamov z največjo dolžino pet, s petimi različnimi elementi, brez ponavljanja elementov v seznamih. Domena opisuje, ali so elementi člani seznama. Učni primeri so bili generirani s predikatom na Sl. 5.10. Pozitivnih primerov je 1.305, negativnih primerov pa je 325. V tem primeru sta bila oba parametra predikata member definirana kot vhodna.

```
member(A,[A|_]).
member(A,[_|B]):-member(A,B).
```

Sl. 5.10 Definicija predikata member, ki je bila uporabljena za generiranje učnih primerov

Poskusi so potekali s sistemi HYPER^2 (v tabelah in slikah razviden kot hyper, uporabljena je bila verzija, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, brez informacije o vhodno/izhodnih spremenljivkah v glavah stavkov, z iskanjem najprej najboljši), Hyper^{2}Beam (HyperB – pokrivanje na nivoju stavkov, informacija o vhodno/izhodnih spremenljivkah in iskanje s snopom), FOIL 6.4 (foil), CPROGOL 4.4 (progol), ALEPH v3 (aleph), SPChillin 1.0A prenesen na Sicstus Prolog (chillin) in Markus V1.1 (markus).

Opravili smo devet serij poskusov z različnim številom primerov (tab. 5.2). V vsaki seriji je bilo opravljenih do 100 poskusov z naključno izbranimi primeri (vsi sistemi so v isti ponovitvi dobili enake primere).

5.2.2 Rezultati

Na Sl. 5.12 inSl. 5.11 vidimo, kako so bili sistemi uspešni pri reševanju problema. Tudi na tej domeni prevladata obe verziji sistema HYPER^2. Že ob zelo majhnem številu učnih primerov (trije pozitivni in trije negativni učni primeri) je HYPER^2 v večini primerov sposoben najti pravilno rešitev. Po pričakovanjih je pri tem rahlo boljša verzija z iskanjem s snopom, saj le ta usmerja iskanje malo bolj v širino in ima pri tem nekaj boljše možnosti, da najde pravilno rešitev. Po pregledu nepravilno rešenih problemov smo opazili, da sta sistema prišla v probleme, če so bili vsi pozitivni primeri daljši od vseh negativnih primerov. V tem primeru sta kot rešitev postavila definicijo, ki pravi, da je pravilni rezultat dosežen, če ima seznam neko minimalno dolžino, iskani element pa je lahko poljuben. Ker je taka definicija praviloma bolj enostavna (ima manjše število literalov) od iskane definicije, jo sistem HYPER^2 preferira. Da je verzija z iskanjem s snopom ob nekaterih teh primerih kljub temu prišla do pravilne rešitve, je rezultat dejstva, da se do pravilne rešitve pride v manj korakih kot pa do definicije z omejitvijo dolžine, če je le-ta dolžina dovolj dolga. Izkaže se, da so za iskano, pravilno, rešitev potrebni štirje koraki. Posledično je v primeru, da so bili vsi pozitivni učni primeri dolžine pet, in vsaj negativni učni primer dolžine štiri, verzija z iskanjem s snopom našla pravilno rešitev, verzija z iskanjem najprej najboljši pa ne. Podobne probleme so sistemu HYPER^2 povzročali primeri, kjer je bil iskani element na istem mestu pri vseh pozitivnih primerih. V takih primerih je sistem definiral, da je iskani element na istem mestu, kot je bil pri vseh podanih učnih primerih. Verzija z iskanjem s snopom se je obnesla bolje (zaradi enakega razloga kot prej – ker je prej naletela na pravilno definicijo), le če je bil pri vseh učnih primerih iskani element na četrtem ali petem mestu.

Poglejmo še delovanje ostalih sistemov. Najbolj uspešen je MARKUS, ki pa v resnici generira preveč kompleksno hipotezo, saj zahteva, da se trenutno prvi element seznama ne pojavi v preostanku seznama (kar pa je pri naši domeni pravilno, saj smo uporabili sezname brez ponavljanja elementov). Da sistemu MARKUS uspe najti pravilno hipotezo, mora v učni množici biti vsaj en pozitivni primer ustavitvenega pogoja rekurzije in vsaj en pozitivni primer prvega klica rekurzije (primer, ki se ga da rešiti z rekurzijo globine ena). Vendar lahko tudi, če ima to podano, zaide v težave. Zgodi se, ko imajo vsi pozitivni primeri sezname z liho

dolžino in vsi negativni primeri sezname s sodo dolžino, da definira predikat member kot negacijo rekurzivnega klica z repom seznama. Ker tak klic ne uspe na seznamu dolžine nič, posledično taka definicija uspe na vseh seznamih lihe dolžine in ne uspe na vseh seznamih sode dolžine. V primeru, ko so vsi negativni primeri sode dolžine in samo nekaj pozitivnih primerov lihe dolžine, MARKUS doda tako negirano rekurzijo kot del hipoteze, nato pa (običajno že prej najde ustavitveni pogoj rekurzije) kot redundantnega odstrani ustavitveni pogoj; v naslednjem koraku ugotovi, da ne more najti konsistentne in kompletne hipoteze ter se ustavi. Opazi se, da bi, če ne bi odstranil ustavitvenega pogoja, ostala možnost, da najde pravilno definicijo; po odstranitvi pa MARKUS nima več te možnosti.

Oznaka serije	Št. poskusov v seriji	Št. pozitivnih primerov	Št. negativnih primerov
1 (002 007)	100	3 (0,2%)	3 (0,7%)
2 (005 01)	100	7 (0,5%)	4 (1%)
3 (01 01)	100	13 (1%)	4 (1%)
4 (05 05)	75	66 (5%)	17 (5%)
5 (10 10)	50	131 (10%)	33 (10%)
6 (20 20)	30	261 (20%)	65 (20%)
7 (50 50)	20	653 (50%)	163 (50%)
8 (70 70)	10	914 (70%)	228 (70%)
9 (100 100)	1	1305 (100%)	325 (100%)

tab. 5.2 Opis posameznih serij domene member

Da ALEPH pravilno reši problem, potrebuje najmanj tri pozitivne primere ustavitvenega pogoja rekurzije (kar je pri ALEPHU razumljivo, ker za delo s seznami potrebuje klic predikata in sta zato dva našteta primera krajša kot ustavitveni pogoj rekurzije). Zanimivo je, da potrebuje le en pozitiven primer prvega rekurzivnega klica (ostali primeri rekurzivnih klicev so lahko večje globine, vendar, morajo kljub temu biti skupaj vsaj trije). ALEPH v primeru, da mu ne uspe pravilna definicija, samo našteje nepokrite primere kot dejstva (prav tako občasno našteje primere, ki jih je kasneje pokril z ustreznim predikatom).

PROGOL prav tako kot ALEPH za pravilno definicijo predikata, ki opisuje zaustavitveni pogoj rekurzije, potrebuje vsaj tri pozitivne učne primere, ki pokrivajo ustavitveni pogoj. Poleg tega potrebuje še vsaj tri primere, ki pokrijejo prvi klic rekurzije (vendar je to, kakor kaže naslednja domena, odvisno od deklaracije učnega primera). Prav tako potrebuje še dodatne primere, ki predstavljajo rekurzivne klice večjih globin, drugače ne uporabi rekurzije, ampak le definira predikata za element prvem in na drugem mestu v seznamu. Izkaže pa se, da, tudi če je vsemu zadoščeno, PROGOL občasno definira napačno hipotezo. To se še posebej pogosto dogaja pri majhnem številu negativnih primerov. Tako PROGOL občasno

definira, da je določen element vedno v seznamu (ko se v učnih primerih pojavi dosti pozitivnih primerov s tem elementom in ni nobenega negativnega primera, ki išče ta element).

Primer učne množice, na kateri sta bila tako ALEPH kot tudi PROGOL uspešna (vsebuje štiri učne primere, ki predstavljajo zaustavitveni pogoj rekurzije in pet učnih primerov, ki predstavljajo prvi rekurzivni klic, s + so označeni pozitivni učni primeri, z – pa negativni):

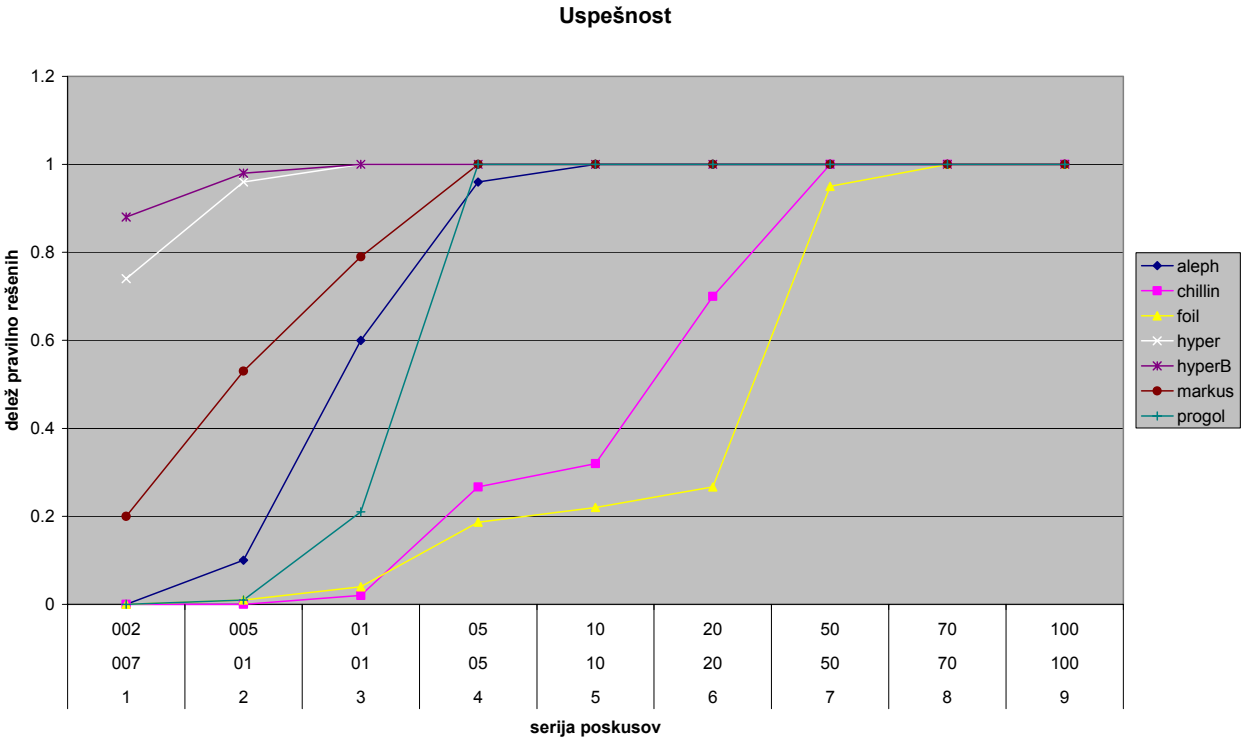
```
+member( d , [c,d,e] )
+member( d , [b,d,e,a] )
+member( a , [a,d,e,b] )
+member( a , [c,e,a,d] )
+member( b , [c,b,e,d] )
+member( e , [c,e,b,d,a] )
+member( c , [a,e,d,c,b] )
+member( c , [c,e,a,d,b] )
+member( e , [d,a,e,b,c] )
+member( b , [b,e,c,a,d] )
+member( e , [e,b,a,c,d] )
+member( c , [b,c,a,e,d] )
+member( a , [c,b,a,d,e] )
-member( d , [a,c,b] )
-member( b , [d,a,e] )
-member( e , [b,c,a,d] )
-member( a , [c,d,b,e] )
```

Primer učne množice, na kateri tako ALEPH kot PROGOL ne uspeta najti pravilne rešitve (ker vsebuje le dva učna primera zaustavitvenega pogoja rekurzije):

```
+member( c , [d,e,c,b] )
+member( a , [e,c,a,d] )
+member( c , [c,e,a,d] )
+member( c , [b,a,c,d] )
+member( c , [e,b,c,d] )
+member( b , [b,a,c,e] )
+member( a , [d,a,c,e] )
+member( c , [e,b,c,d,a] )
+member( c , [d,b,c,e,a] )
+member( d , [c,a,d,e,b] )
+member( a , [d,b,a,e,c] )
+member( a , [c,a,e,b,d] )
+member( b , [a,b,e,c,d] )
-member( b , [c,e] )
-member( e , [c,b,d] )
-member( e , [d,b,a,c] )
-member( b , [a,c,e,d] )
```

Sistem CHILLIN ima zaradi drugačnega postopka generiranja hipotez drugačne probleme kot ostali sistemi. Ker začenja s pozitivnimi učnimi primeri in jih združuje ter posplošuje, se pogosto (še posebej, ko ima na voljo manjše število učnih primerov) zgodi, da jih ne posploši dovolj. Na primer občasno definira ustavitveni pogoj rekurzije, tako da mu zadoščajo le primeri s seznami dolžine tri ali več. Ta napaka se odpravi, ko ima sistem na voljo več pozitivnih učnih primerov. Druga napaka, ki jo CHILLIN dela, se pojavi ob pomanjkanju negativnih učnih primerov in je pravzaprav ravno nasprotna od prejšnje – v tem primeru CHILLIN preveč posploši in definira recimo; da relacija velja pri poljubnem elementu in poljubnem seznamu dolžine pet.

Kot se vidi iz grafa uspešnosti (Sl. 5.11), ima sistem FOIL najbolj ostre zahteve glede učnih primerov, da pravilno reši problem. Za definicijo ustavitvenega pogoja rekurzije potrebuje sicer le en pozitivni primer, vendar za definicijo rekurzivnega klica potrebuje vsaj dva pozitivna primera, ki predstavljata klic predikata in prvi rekurzivni klic, ki je posledica originalnega klica.



Sl. 5.11 Uspešnost različnih sistemov na domeni member

Ko pogledamo čase, potrebne za reševanje podanih problemov (Sl. 5.13, Sl. 5.14 in Sl. 5.15), opazimo, da se poraba časa povečuje polinomsko glede na število učnih primerov. To še posebej velja pri sistemih CHILLIN, MARKUS in PROGOL, nekoliko manj občutno pri obeh verzijah sistema HYPER^2, (pri tem, da je HYPER^2 z iskanjem najprej najboljši občutno hitrejši od verzije z iskanjem s snopom in, razen pri vseh podanih primerih, celo na enakem nivoju kot FOIL), medtem ko tako ALEPH kot FOIL z vsemi ali celo z večino primerov rešita problem celo hitreje kot z manj primeri. Kljub temu je treba poudariti, da vsi sistemi rešijo probleme dokaj hitro, saj je bil najpočasnejši čas izmerjen na 8.3sekund.

Na Sl. 5.16, Sl. 5.17 in Sl. 5.18 vidimo, da po velikosti sestavljenih hipotez občutno izstopa FOIL, ki kljub temu, da tako ALEPH kot PROGOL, ko jima ne uspe pokriti vseh primerov, samo naštejeta nepokrite primere kot dejstva, ustvari daleč največje hipoteze. Zanimivo je, da se te velike hipoteze pojavljajo ne le pri nepravilnih rešitvah (Sl. 5.17) temveč tudi pri pravilnih rešitvah (Sl. 5.18).

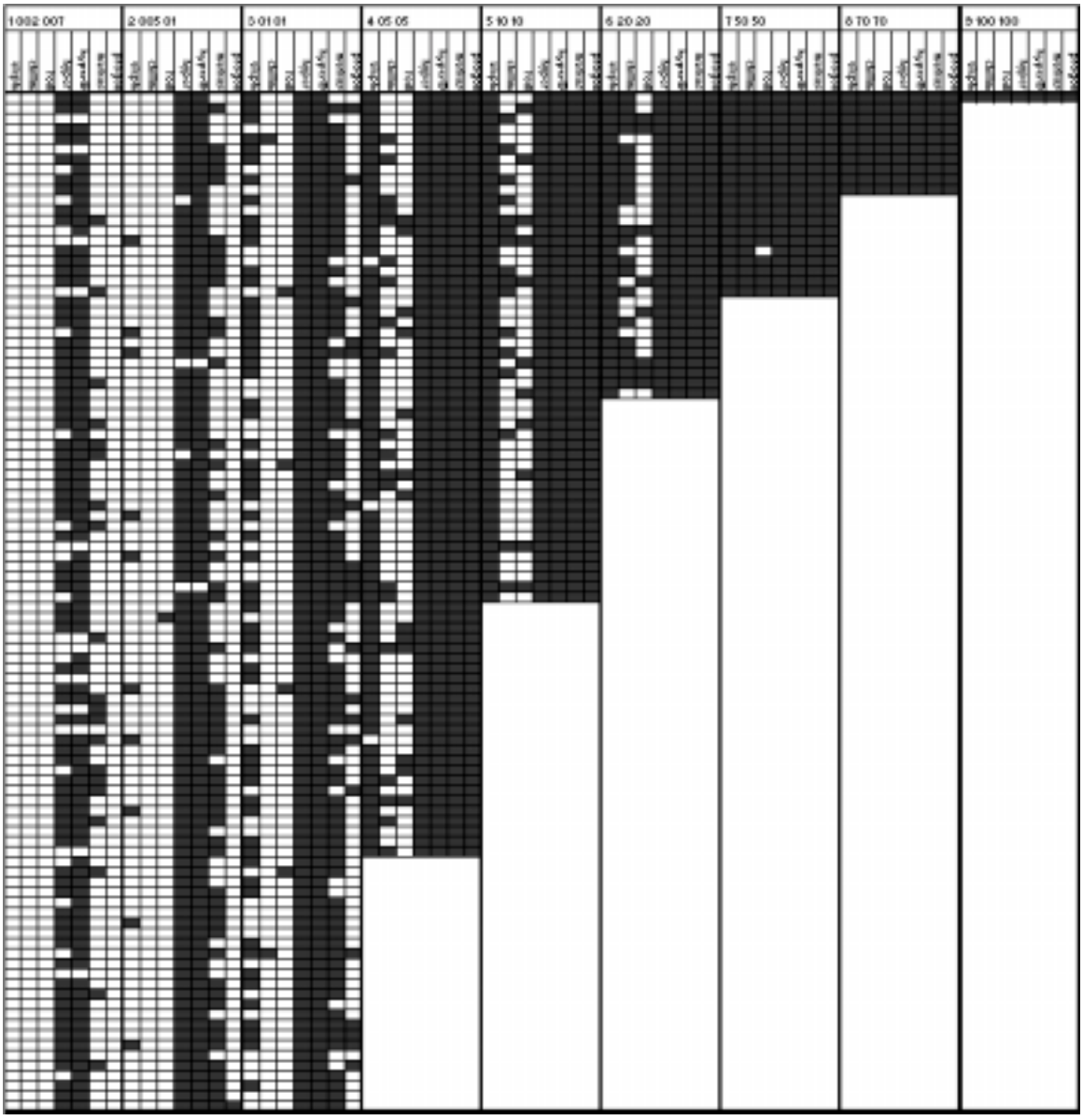
Primer hipoteze, ki jo je ALEPH generiral na učni množici s 131 pozitivnimi in 33 negativnimi primeri v seriji pet (klice predikata za delo s seznami smo zamenjali z izrazi):

```
member(a,[c,b,d,e,a]).
member(d,[b,e,d,c,a]).
member(c,[a,e,c,b,d]).
member(c,[e,b,c,a,d]).
member(A,[C,A|E]).
member(A,[A|C]).
member(b,[a,c,e,b]).
member(A,[C|D]) :-
  member(A,D).
```

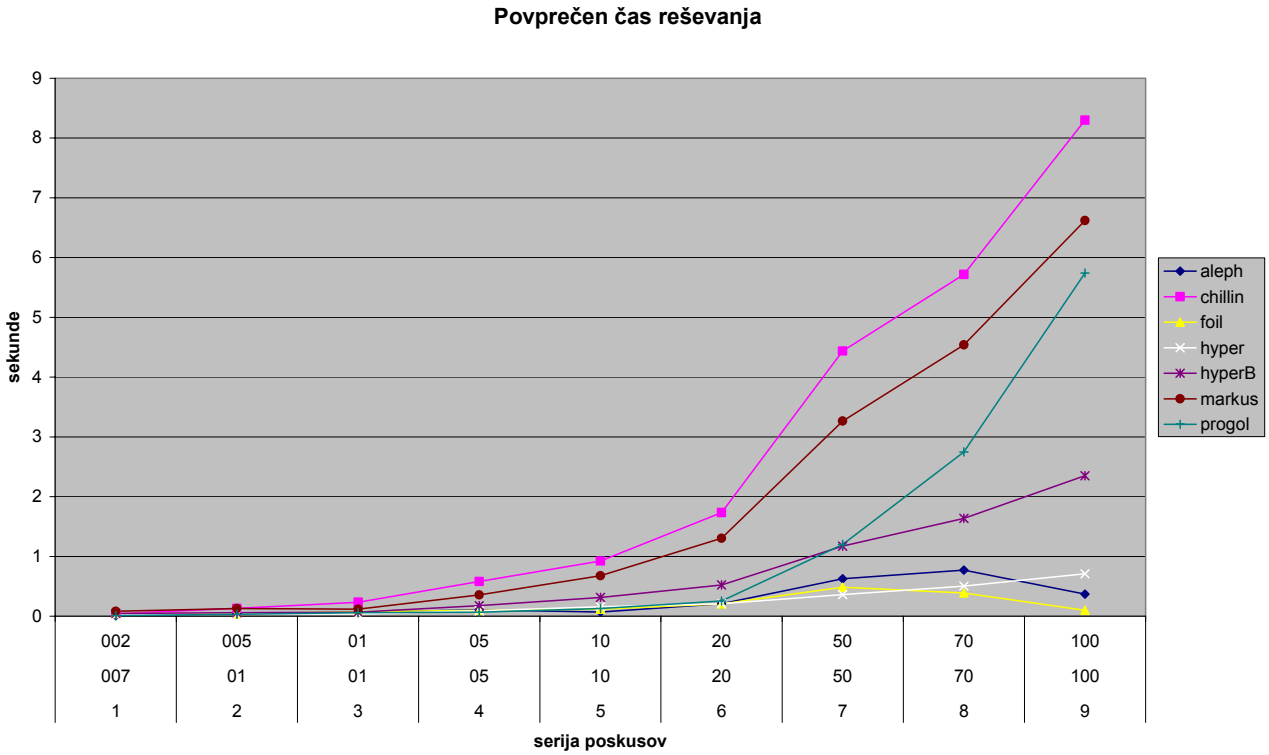
Ostali sistemi generirajo pravilne hipoteze do približno dvakratne velikosti. Pri tem se opazita dve anomaliji;

- ALEPH ima nekatere hipoteze prevelike, ker v hipotezo doda primere, ki jih nato pokrije (drugače rečeno, ne odstrani redundantnih predikatov),
- MARKUS, ki v tej domeni v vseh pravilnih rešitvah doda negiran rekurzivni klic, ki zahteva, da se glava seznama ne pojavi v repu.

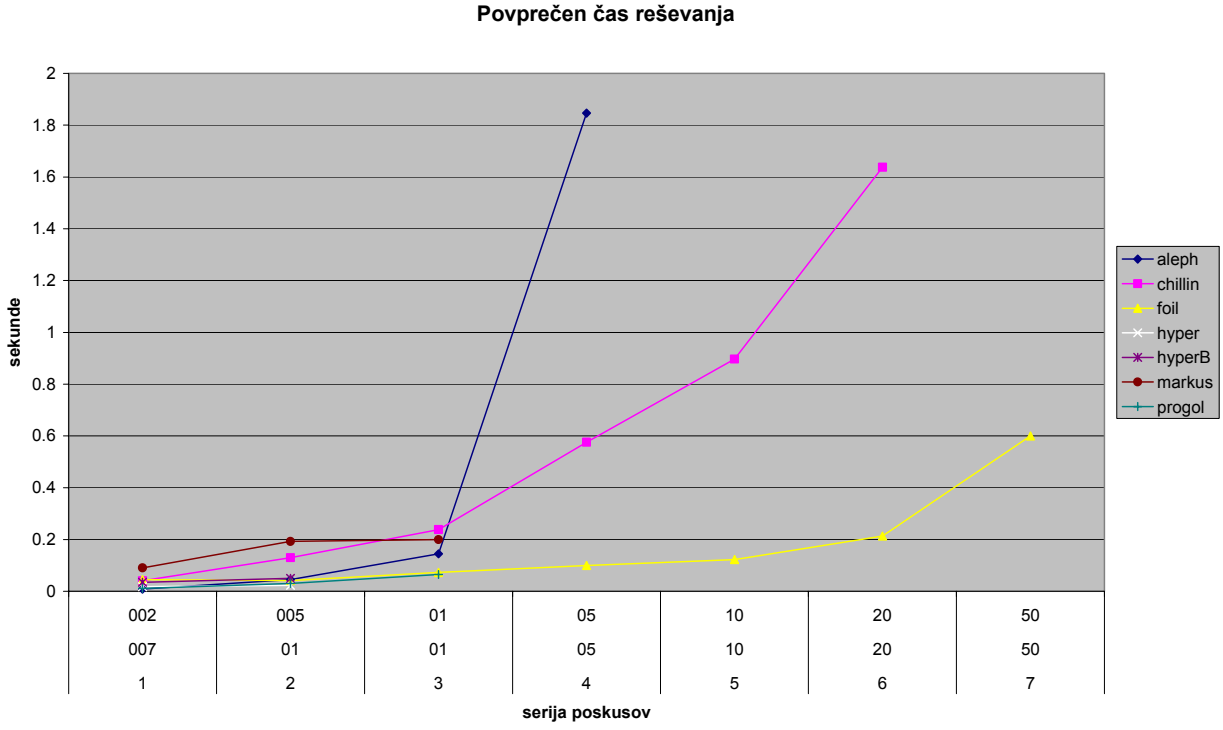
Pri nepravilnih hipotezah izstopa (poleg sistema FOIL) predvsem ALEPH, kar je posledica naštevanja nepokritih primerov kot dejstev.



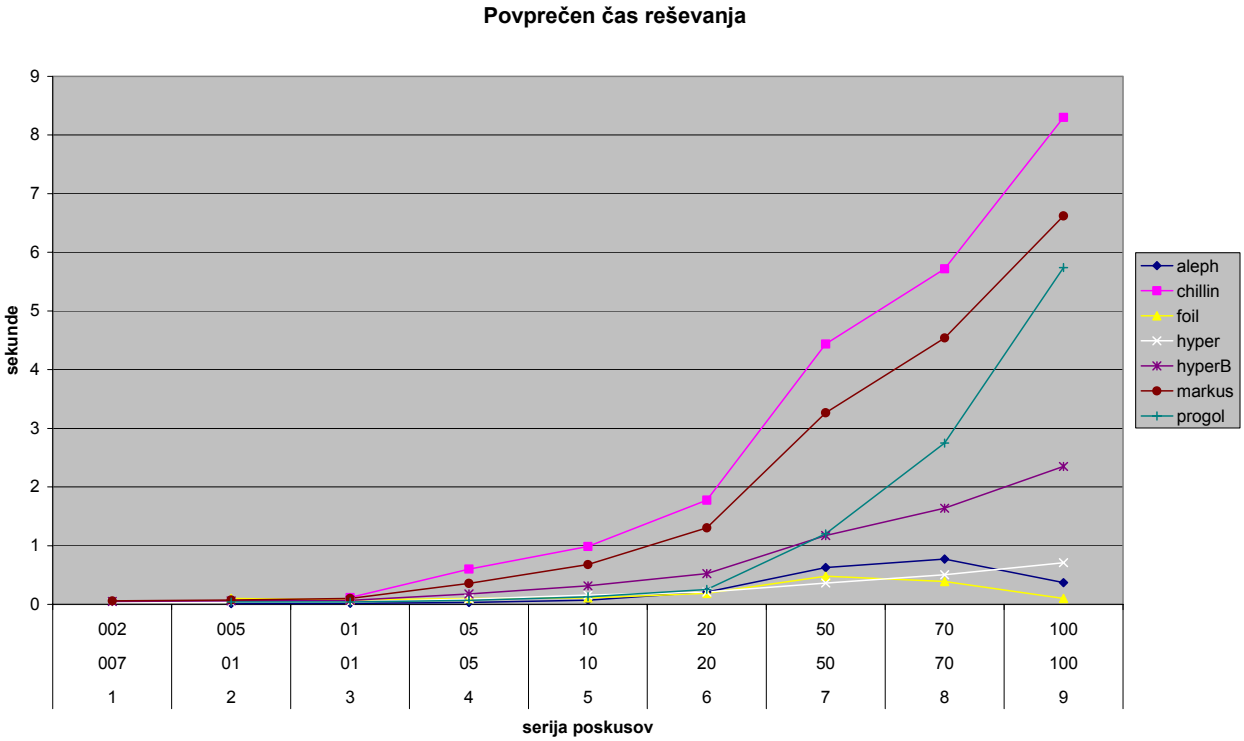
Sl. 5.12 Primerjava pravilno rešenih problemov domene member (kasnejše serije imajo manj kot 100 problemov)



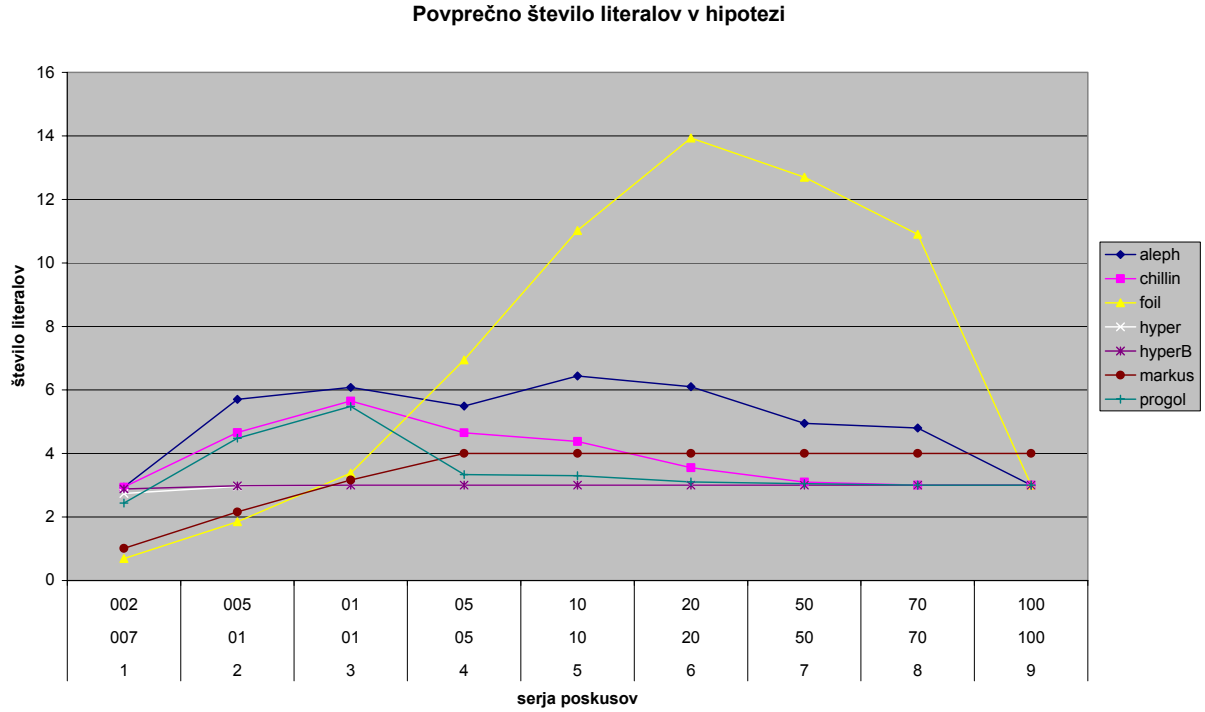
Sl. 5.13 Povprečna poraba časa sistemov v posamezni seriji



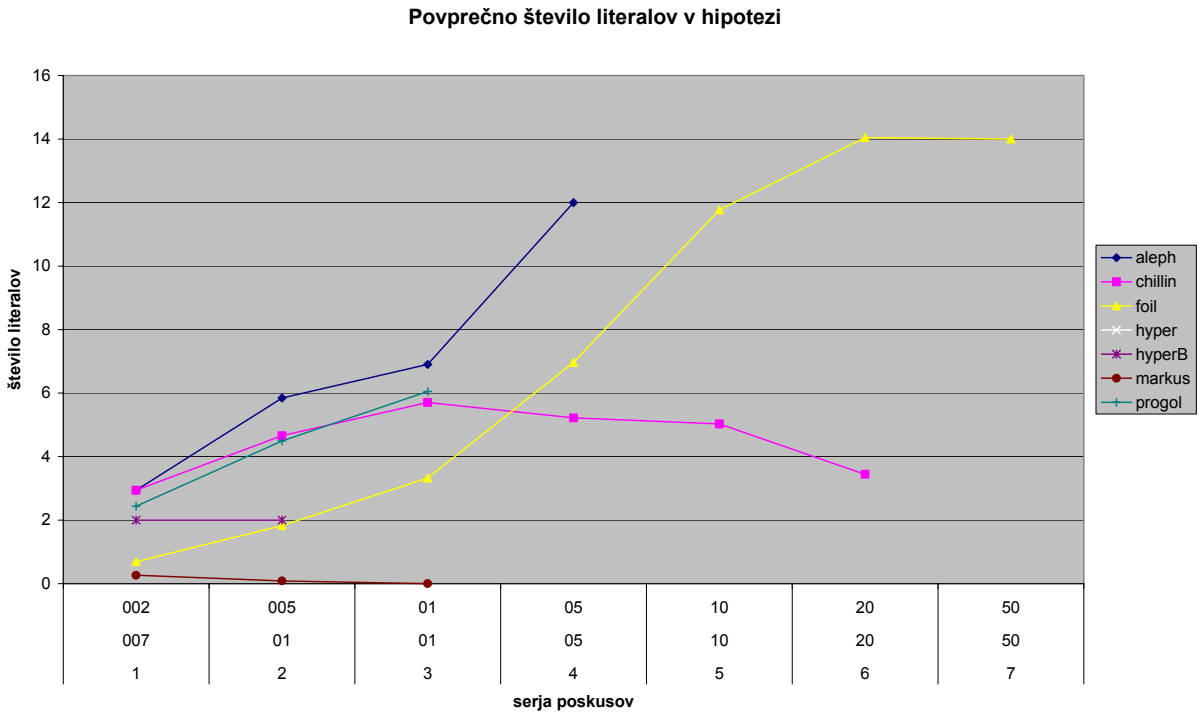
Sl. 5.14 Povprečna poraba časa različnih sistemov v posamezni seriji, ko sistem ni pravilno rešil problema



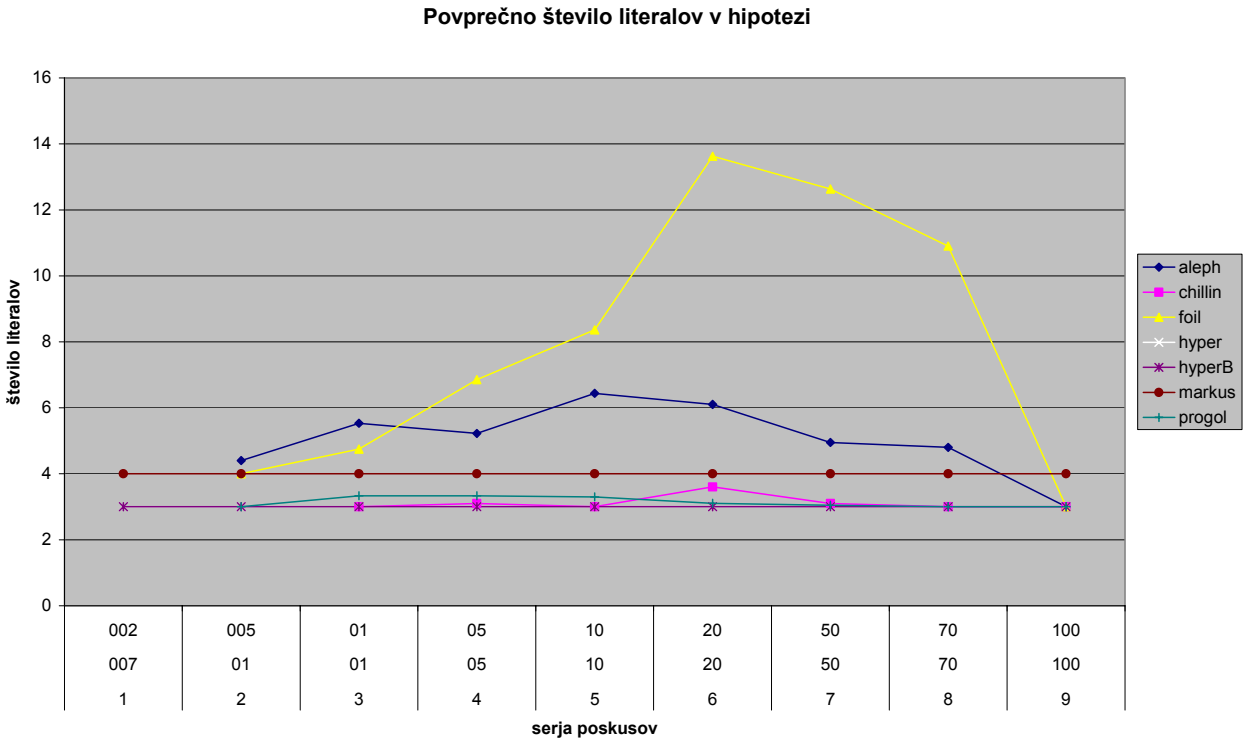
Sl. 5.15 Povprečna poraba časa različnih sistemov v posamezni seriji, ko je sistem pravilno rešil problem



Sl. 5.16 Povprečno število literalov v hipotezah različnih sistemov v posamezni seriji



Sl. 5.17 Povprečno število literalov v hipotezah različnih sistemov v posamezni seriji, ko sistem ni pravilno rešil problema



Sl. 5.18 Povprečno število literalov v hipotezah različnih sistemov v posamezni seriji, ko je sistem pravilno rešil problem

5.3 Domena memberA

5.3.1 Opis domene

Celotna domena je sestavljena iz seznamov z največjo dolžino pet, s petimi različnimi elementi, brez ponavljanja elementov v seznamih. Domena opisuje ali so elementi člani seznama. Pozitivnih primerov je 1.305, negativnih primerov pa je 325. V tem primeru je bil seznam definiran kot vhodni parameter, element pa kot izhodni parameter (kot se uporablja za naštevanje elementov seznama). To je tudi edina razlika s prejšnjo domeno, in s tem smo poskušali oceniti občutljivost sistemov na definicijo iskane hipoteze. Ker se vhodni in izhodni parametri ne definirajo pri sistemih CHILLIN in FOIL, za ta dva sistema ni sprememb glede na prejšnjo domeno. Vse učne množice so bile enake kot pri prejšnji domeni.

Poskusi so potekali z različnimi verzijami sistema HYPER^2:

- v tabelah in slikah razviden kot hyperE, verzija, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, brez informacije o vhodno/izhodnih spremenljivkah v glavah stavkov, z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyper, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyperG, ki si zapomni podatke o pokrivanju primerov na nivoju hipotez, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- Hyper^{2}Beam (HyperB), ki uporablja pokrivanje na nivoju stavkov, informacijo o vhodno/izhodnih spremenljivkah in iskanje s snopom.

Poleg tega smo uporabili tudi sisteme FOIL 6.4 (foil), CPROGOL 4.4 (progol), ALEPH v3 (aleph), SPChillin 1.0A prenesen na Sicstus Prolog (chillin) in Markus V1.1 (markus). Opravili smo devet serij problemov z različnimi števili pozitivnih in negativnih učnih primerov (tab. 5.2). V vsaki seriji je bilo opravljenih do 100 poskusov z naključno izbranimi primeri (vsi sistemi so v isti ponovitvi dobili enake primere).

5.3.2 Rezultati

Ko pogledamo rezultate na domeni memberA (Sl. 5.19 in Sl. 5.20), opazimo, da je tudi tu v prednosti HYPER^2, vendar pa to velja le za verzije, ki imajo informacijo o vhodno/izhodnih

spremenljivkah v glavah stavkov. Pri verziji, ki te informacije ne zna upoštevati in predvideva, da so vse spremenljivke v glavi vhodne, v rekurzivnem klicu pa ne, se zaradi te nekonsistence pojavijo problemi, ki povzročijo slabše delovanje. Medtem pa verzije, ki znajo upoštevati to informacijo, delujejo enako kot pri prejšnji domeni (to je verzija z iskanjem s snopom, ki deluje enako kot prej; drugi dve verziji pa delujeta kot je prej delovala verzija brez informacije o vhodno/izhodnih spremenljivkah, saj je le ta pri prejšnji domeni pravilno predvidevala, da so vse spremenljivke vhodne).

Če pogledamo delovanje ostalih sistemov, opazimo največjo razliko pri sistemu MARKUS, ki je občutno bolj uspešen z manj učnimi primeri (vendar doseže 100% uspešnost na isti točki kot pri prejšnji domeni). Poleg tega MARKUS tudi preneha izdelovati prekomplicirane hipoteze, saj ne zahteva več, da se glava seznama ne ponovi v repu. Ko pregledamo primere, ki jih v tej domeni MARKUS reši in jih ni rešil v prejšnji domeni, opazimo, da ne potrebuje več pozitivnega učnega primera, ki pokriva prvi rekurzivni klic, ampak potrebuje le primer, ki se ga reši z rekurzivnim klicem (seveda poleg primera ustavitvenega pogoja rekurzije).

Večja razlika se pojavi tudi pri sistemu PROGOL, vendar le pri tretji seriji poskusov, kjer ima sistem na voljo trinajst pozitivnih in štiri negativne učne primere. Ko primerjamo probleme, ki jih je v tem primeru sistem uspešno rešil, v prejšnji domeni pa ne, opazimo, da ni več potrebe po treh primerih, ki pokrivajo prvi rekurzivni klic. Poleg treh primerov, ki pokrivajo ustavitveni pogoj rekurzije, je dovolj več primerov, ki pokrivajo rekurzivne klice (lahko tudi z globino večjo od ena).

ALEPH se glede uspešnosti obnaša približno enako kot pri prejšnji domeni, vendar če pogledamo, katere probleme je pravilno rešil v prejšnji domeni (Sl. 5.12) in v tej domeni (Sl. 5.19) opazimo, da je sedaj rešil pet problemov, ki jih prej ni, in ni mu uspelo rešiti sedem problemov, ki jih je prej rešil. Zakonitost, po kateri je prišlo do te spremembe, ni bila odkrita tudi po podrobnem ogledu učnih množic teh problemov.

Sistema FOIL in CHILLIN se glede na to, da nimata mehanizmov za nastavljanje tipa spremenljivk v glavi iskanega predikata, obnašata v tej domeni natanko tako kot v prejšnji (saj je bila razlika samo v tem, da je ena spremenljivka sedaj izhodna).

Ko pogledamo čase potrebne za reševanje problemov (Sl. 5.21 do Sl. 5.25), opazimo, da najbolj občutno izstopa ALEPH, ki pri večjem številu primerov v učni množici potrebuje za nekaj velikostnih razredov več časa kot pri manjšem številu primerov in tudi kot pri prejšnji domeni (kjer povprečna poraba časa v eni seriji ni nikoli presegla 1 sekunde). Treba je

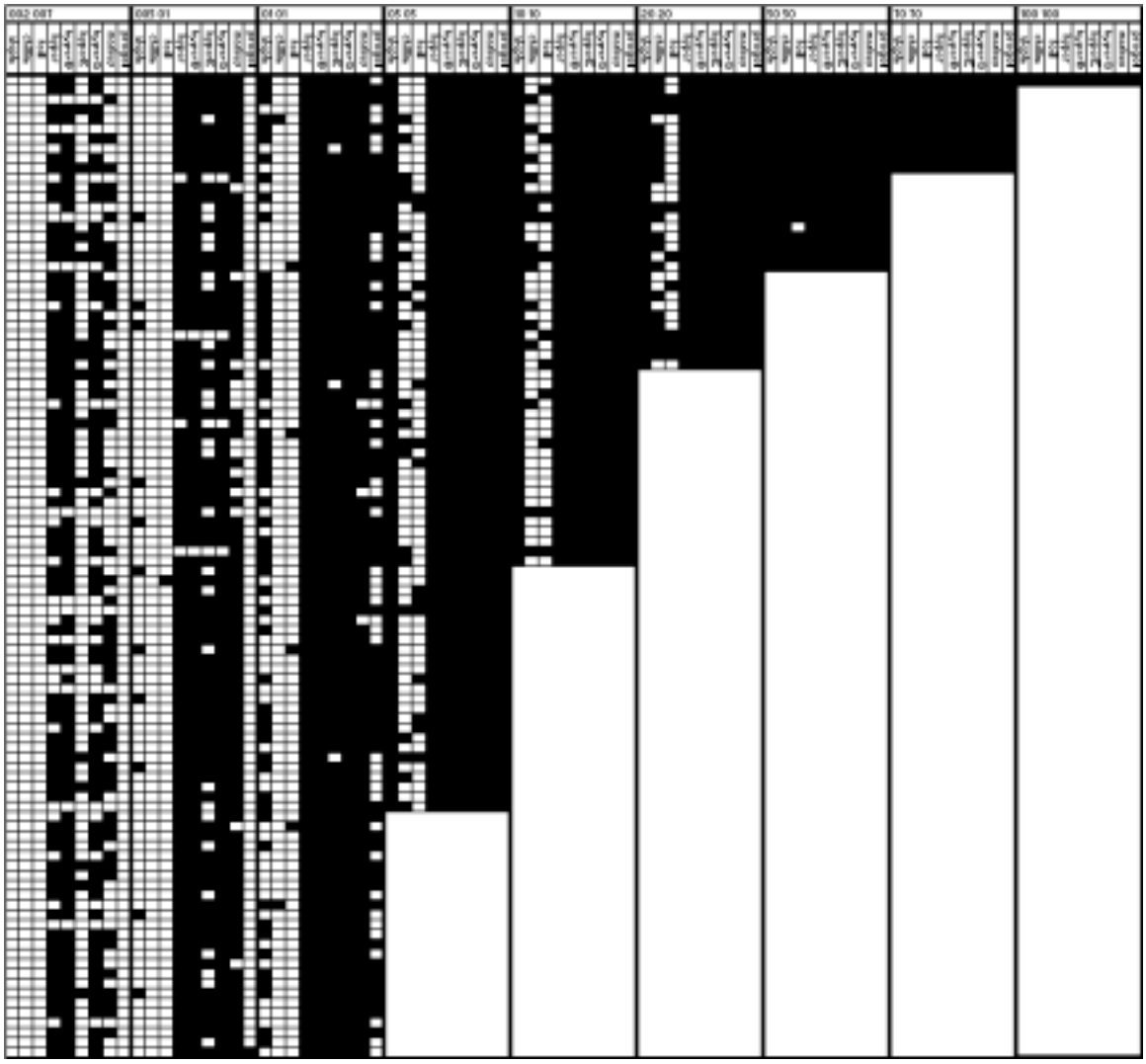
poudariti, da tako visoko povprečje ni posledica enega ali dveh slabih rezultatov. Najhitrejši čas, ki ga je ALEPH dosegel v osmi seriji, je 85 sekund, najpočasnejši pa 711 sekund, pri tem je 50% časov preseglo 500 sekund.

Tudi povprečje časov, ki jih je porabil sistem PROGOL v sedmi seriji (s 50% primerov), je nenavadno, saj je v tej seriji porabil skoraj dvakrat več časa kot v najpočasnejšem poskusu ostalih serij (dva najdaljša časa, če izvzamemo sedmo serijo, sta 7.8 sekund v deveti seriji in 7.4 sekund v osmi seriji, najhitrejši čas v sedmi seriji pa je 13.8 sekund). Edina razlaga takega obnašanja, brez natančnega poznavanja notranjega delovanja sistema, je, da se s številom učnih primerov povečuje čas potreben za obdelavo posameznega predikata, ki bi bil možen za vstavitev v hipotezo, hkrati se povečuje usmerjenost proti pravim predikatom oziroma hipotezi in s tem zmanjšuje število korakov. In v sedmi seriji se, kakor kaže, zelo poveča potreben čas, ne da bi se za enako mero zmanjšalo število korakov.

MARKUS je poleg tega, da je v tej domeni dosti bolj uspešen kot v prejšnji, tudi dosti hitrejši. Čas, ki ga porabi MARKUS, se, kot je pričakovano, povečuje s številom učnih primerov. HYPER^2 (vse verzije) se obnaša podobno kot v prejšnji domeni – torej se porabljen čas povečuje s številom učnih primerov, s tem da je najpočasnejša verzija z iskanjem s snopom, ker pregleda več hipotez. FOIL in CHILLIN se seveda obnašata enako kot v prejšnji domeni (za ta dva sistema sta domeni enaki).

Razen pri vseh verzijah sistema HYPER^2, ki konstantno generirajo hipoteze z največ tremi literali (pri tem se štejejo tudi glave stavkov) in pri sistemu MARKUS, ki, kadar ne najde primerne hipoteze, vrne prazno hipotezo, se (kot se lahko vidi na slikah Sl. 5.26, Sl. 5.27 in Sl. 5.28) velikost hipotez, ki jih producirajo sistemi, sprva povečuje in nato zmanjša do optimalne hipoteze, ki jo vsi sistemi vrnejo, ko imajo na razpolago vse možne učne primere. Pri tem daleč izstopa FOIL, ki producira največje hipoteze. Povprečje velikosti hipotez sistema FOIL v šesti seriji doseže štirinajst literalov, kar je skoraj petkratna velikost optimalne hipoteze (ki je sestavljena iz treh litreralov). Prevelika velikost hipotez je verjetno posledica tega, da FOIL pričakuje šumne podatke in da ni zelo uspešen pri odstranjevanju redundantnih stavkov iz hipotez. V serijah, ki vsebujejo majhno število učnih podatkov, PROGOL in predvsem ALEPH generirata hipoteze, ki so celo večje od hipotez sistema FOIL, vendar sta ta dva sistema bolj uspešna pri generiranju pravilnih hipotez, ki so pravilom manjše od nepravilnih (še posebej pri teh dveh sistemih, ki v primeru, da jima ne uspe pokriti vseh pozitivnih učnih primerov, preostale naštejeta kot dejstva v hipotezi). Učinkovito

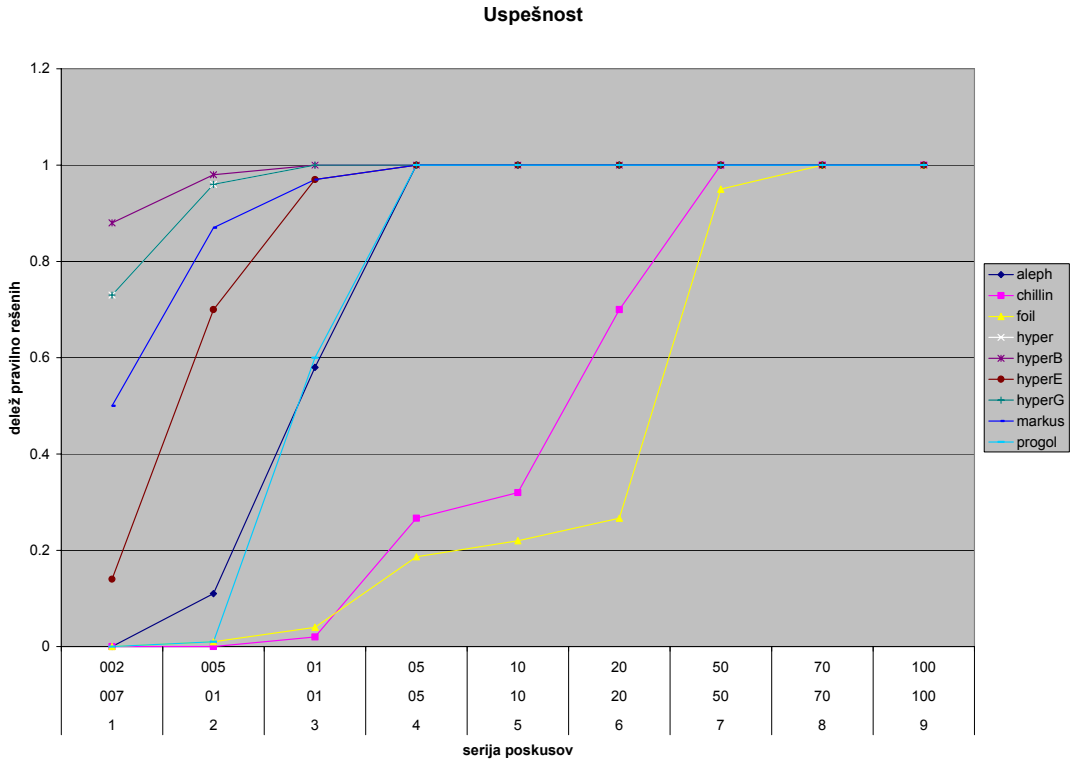
odstranjevanje redundatnih stavkov iz hipotez dodatno zmanjšuje povprečno velikost hipotez sistema PROGOL.



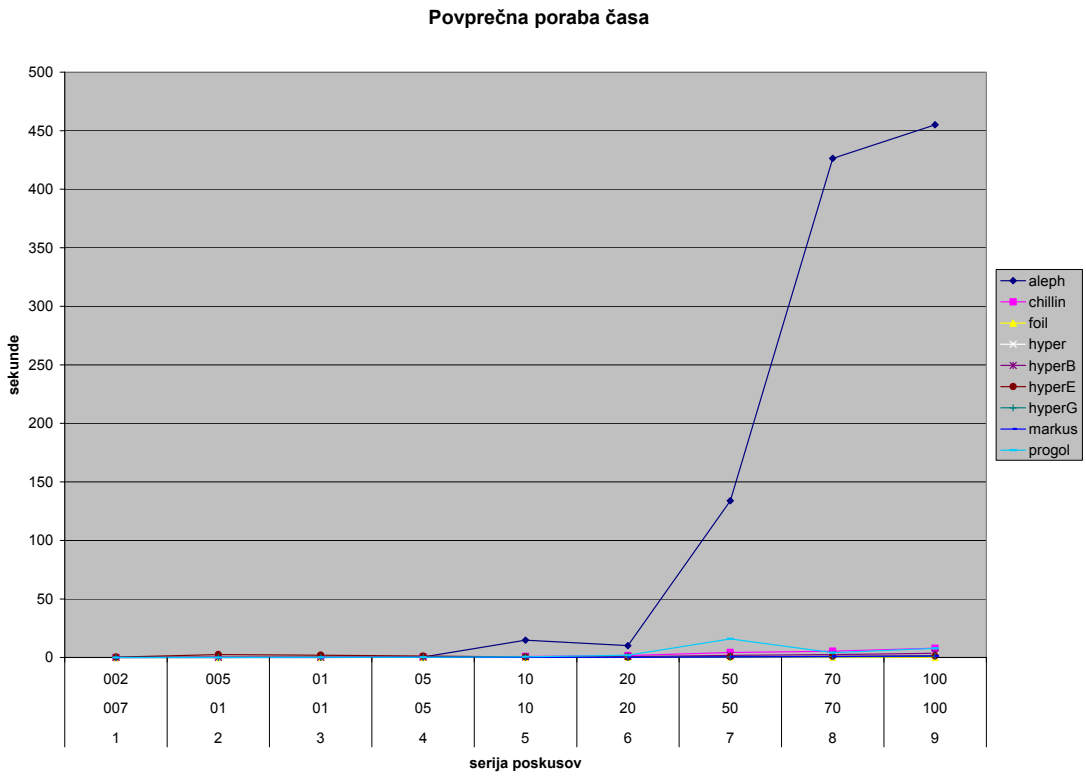
Sl. 5.19 Primerjava pravilno rešenih problemov (kasnejše serije imajo manj kot 100 problemov)

Če primerjamo vse rezultate iz te in prejšnje domene, lahko sklepamo, kako so različni sistemi občutljivi na (vhodno/izhodni) tip argumentov iskane relacije. FOIL in CHILLIN sta, ker ne uporabljata teh informacij, seveda čisto neobčutljiva. Verzije sistema HYPER[2], ki uporabljajo informacije o tipu argumentov, so tudi neobčutljive. V nasprotju pa je verzija, ki teh podatkov ne uporablja (uporablja jih le pri predznanju) – kot jih tudi originalni HYPER ni, dokaj občutljiva, če se izkaže, da je tip drugačen od predvidenega (predvideva se, da so argumenti vhodni), vendar se občutljivost kaže le v uspešnosti sistema, ne pa v času reševanja. MARKUS se na drugi strani je z definicijo argumentov, ki mu očitno bolj ustreza, občutno izboljšal, tako po učinkovitosti kot tudi časovno. Poglavje zase pa sta sistema ALEPH in PROGOL. Prvi je po učinkovitosti zelo stabilen, vendar se izredna občutljivost

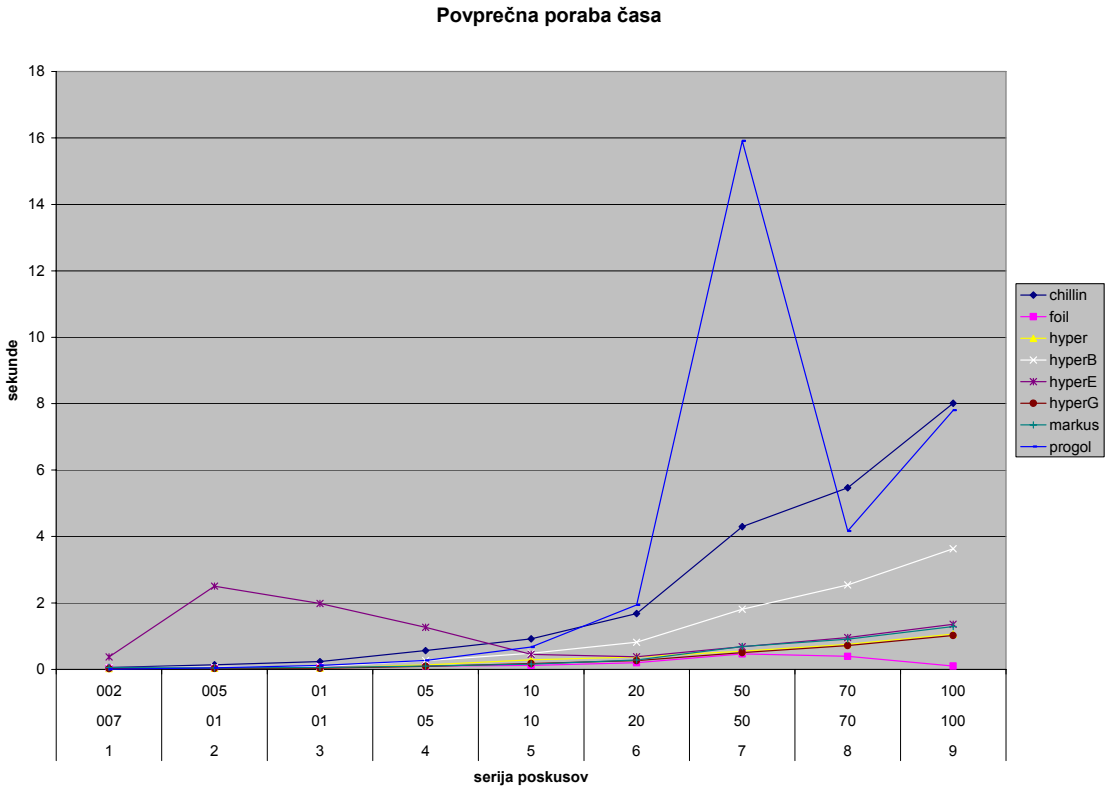
pokaže na času izvajanja, drugi v določenem trenutku naredi občuten skok pri učinkovitosti, vendar naredi tudi skok pri porabi časa.



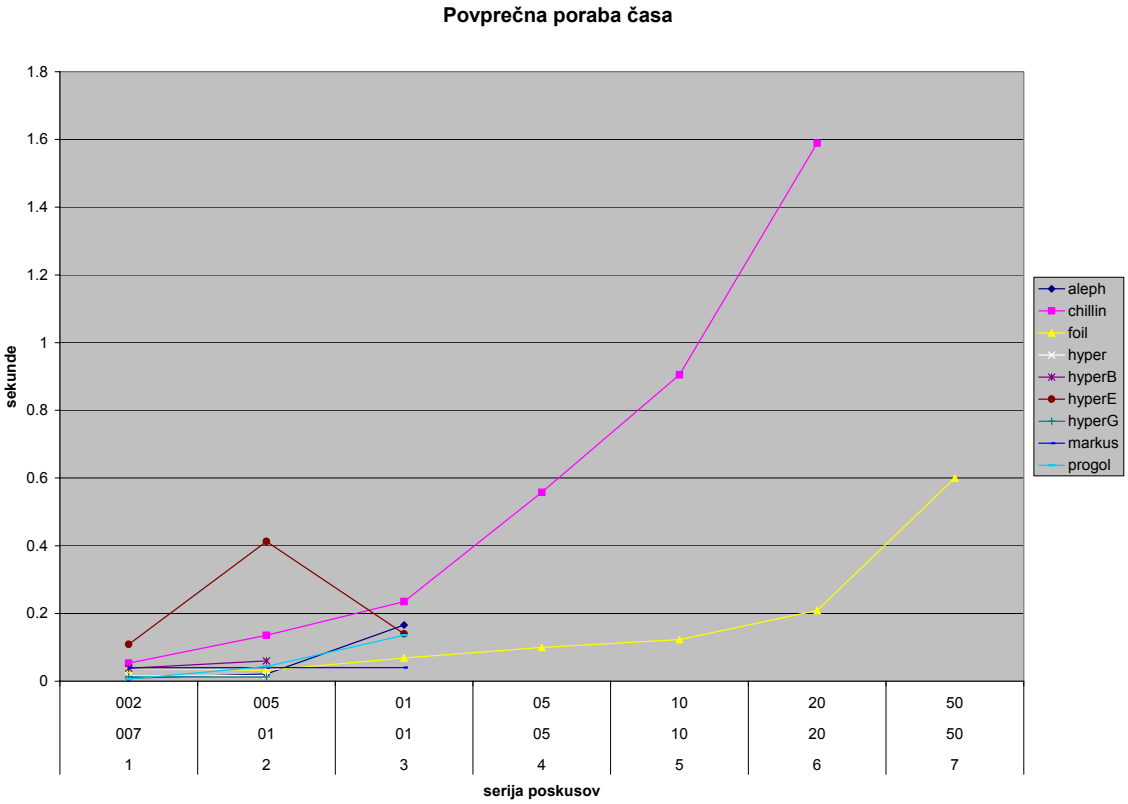
Sl. 5.20 Uspešnost različnih sistemov na domeni memberA



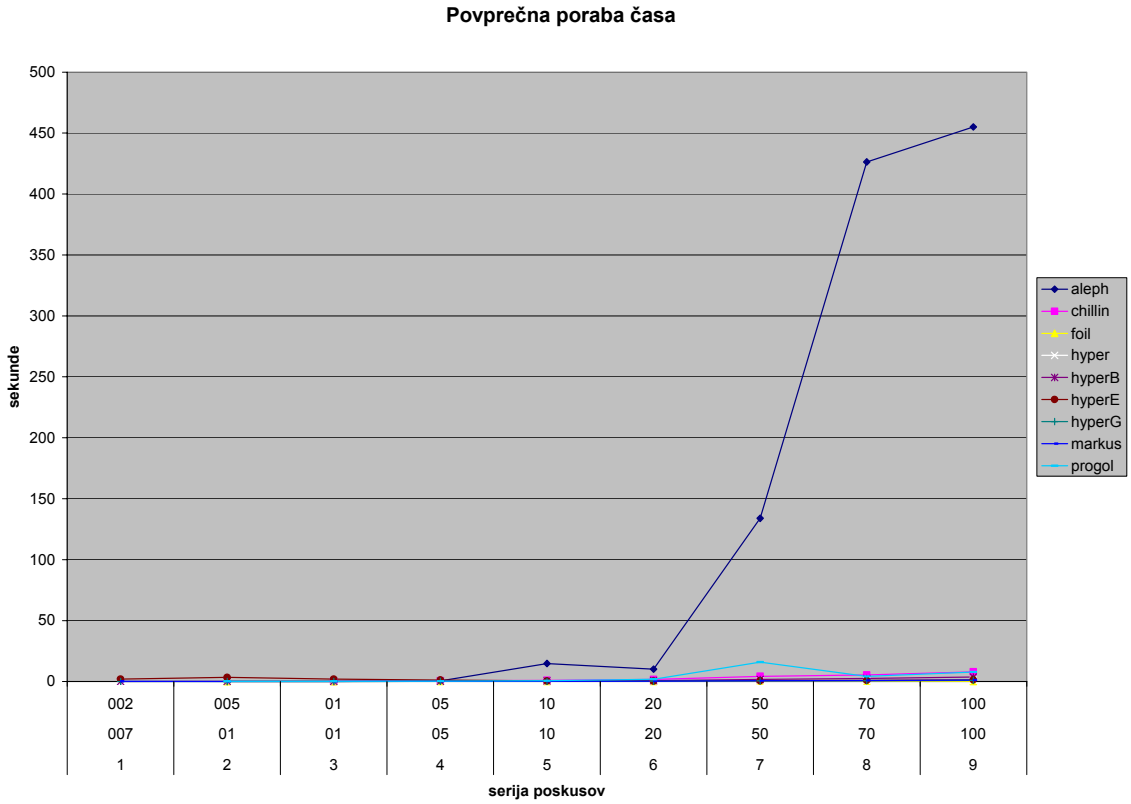
Sl. 5.21 Povprečna poraba časa sistemov v posamezni seriji



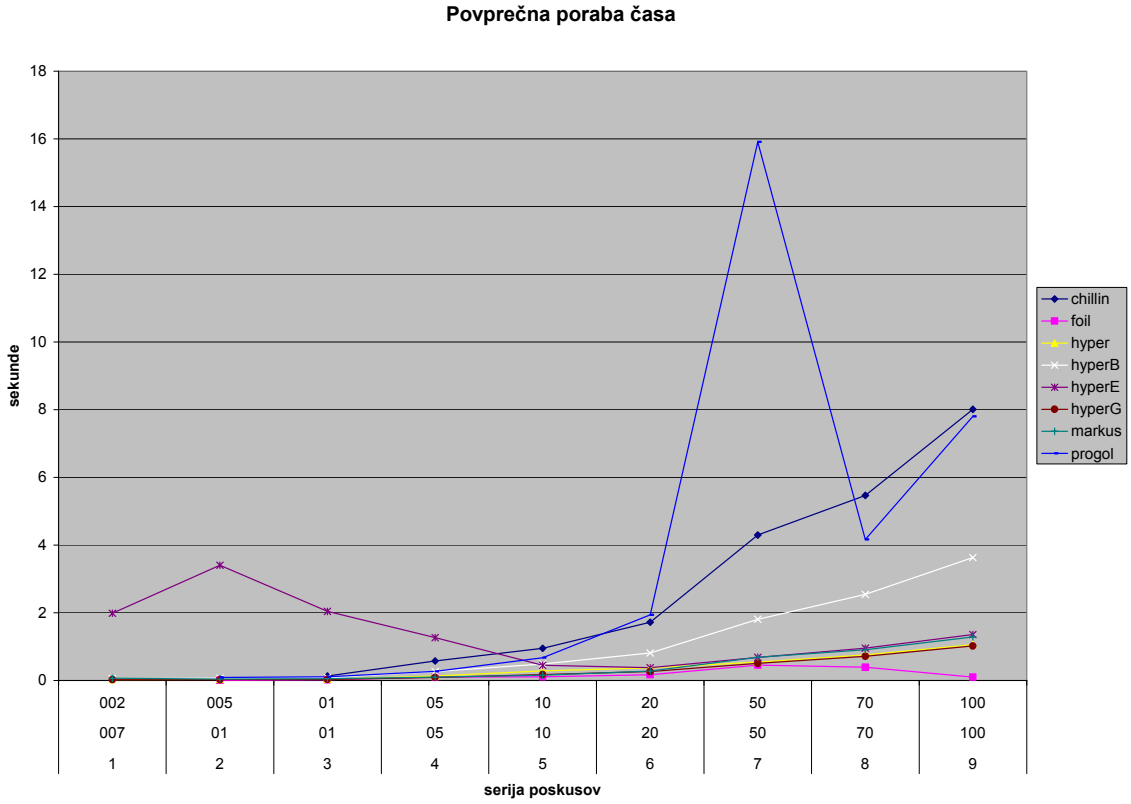
Sl. 5.22 Povprečna poraba časa sistemov (brez sistema ALEPH) v posamezni seriji



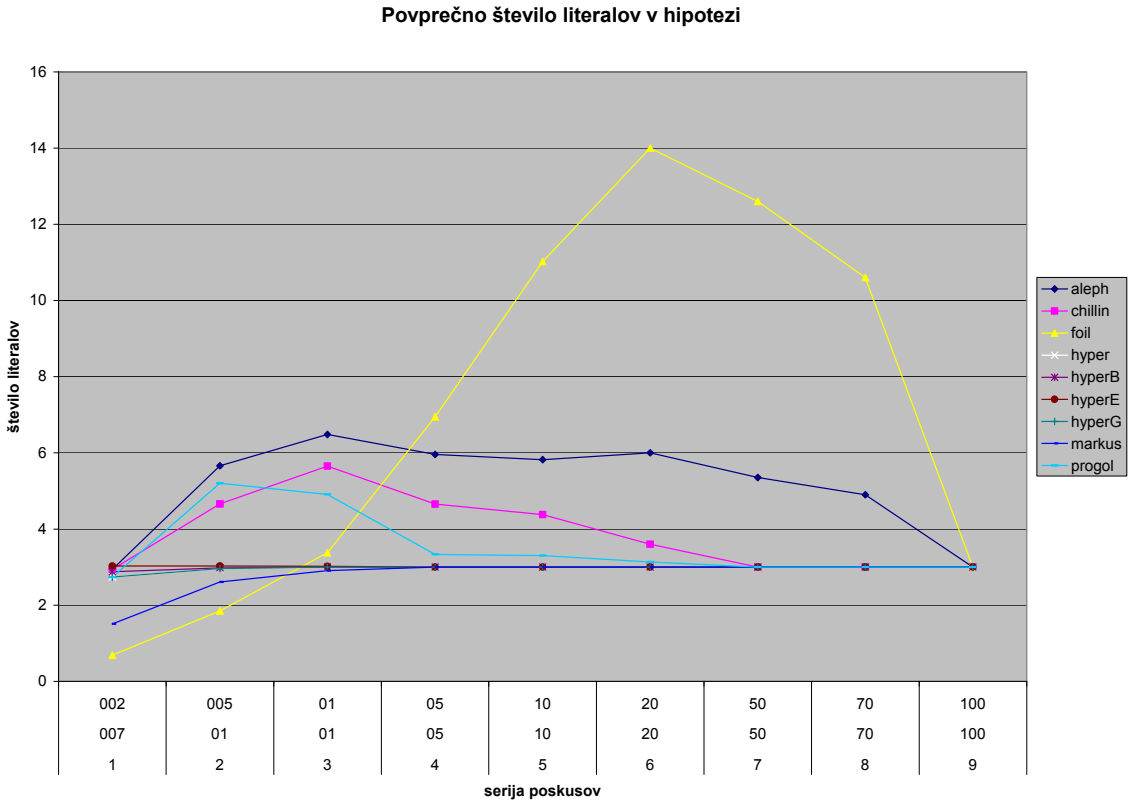
Sl. 5.23 Povprečna poraba časa različnih sistemov, ko le ti niso pravilno rešili problema



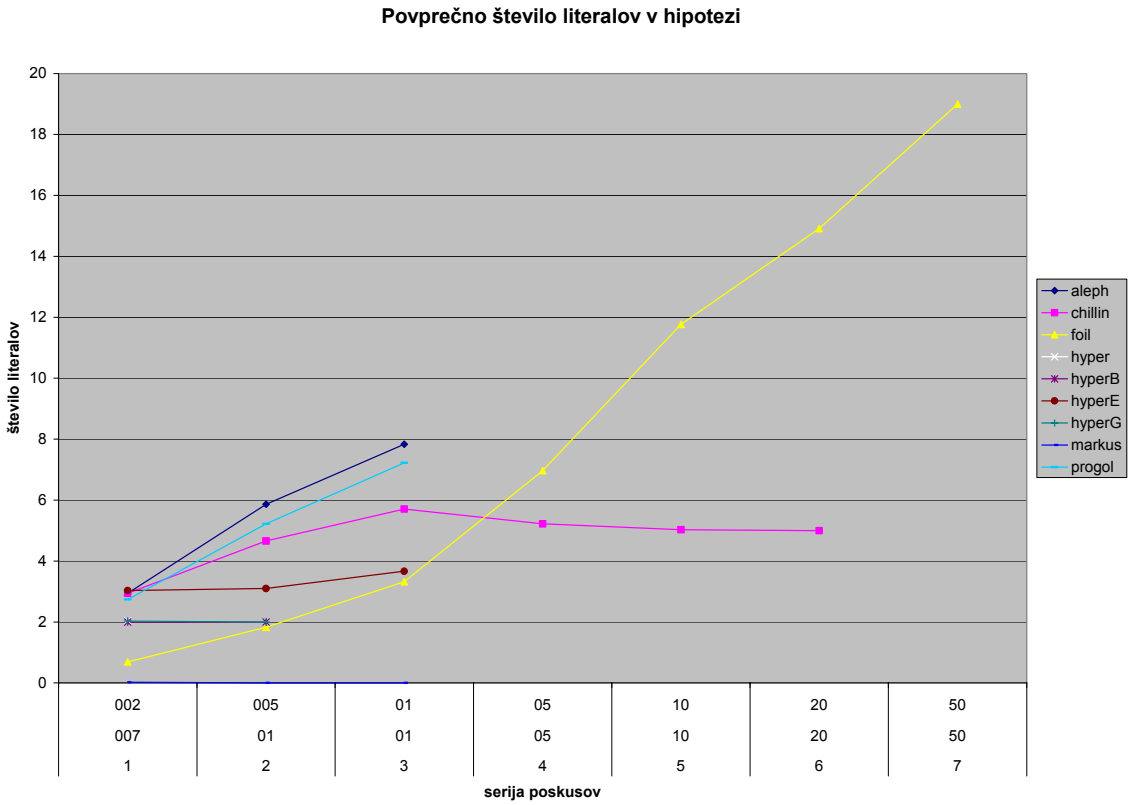
Sl. 5.24 Povprečna poraba časa različnih sistemov, ko so pravilno rešili problem



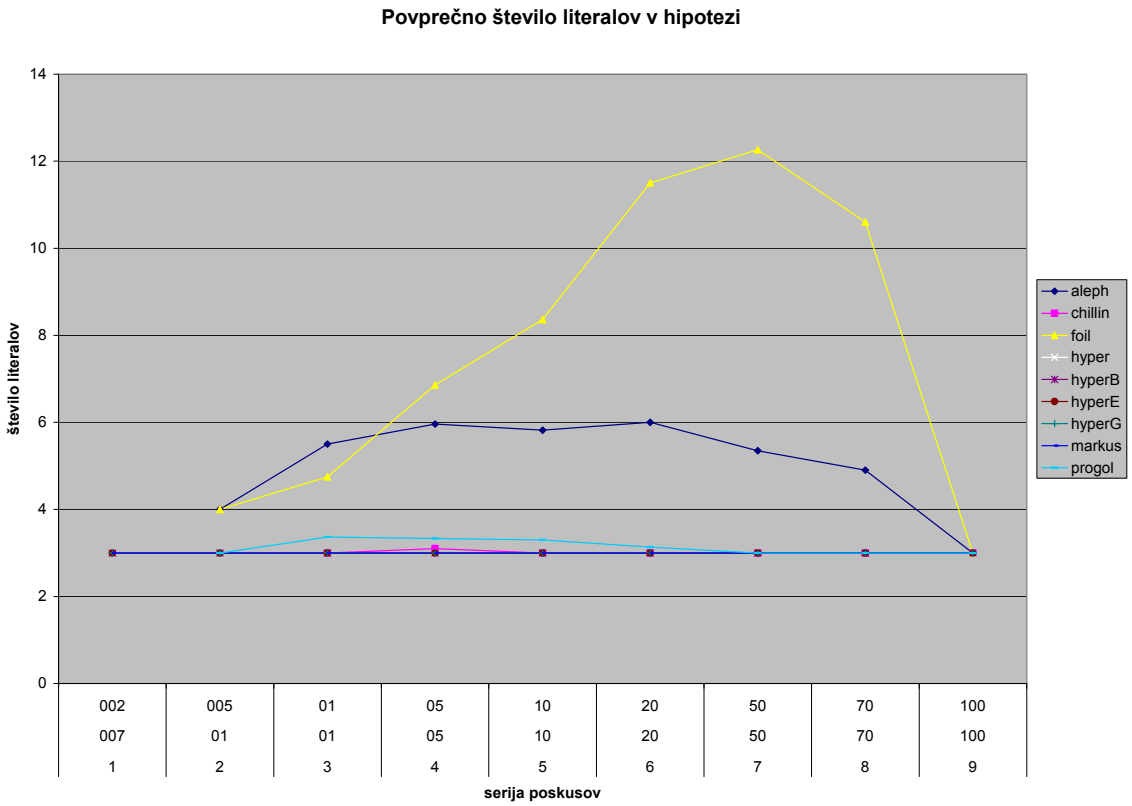
Sl. 5.25 Povprečna poraba časa različnih sistemov (brez sistema ALEPH), ko so pravilno rešili problem



Sl. 5.26 Povprečno število literalov v hipotezah različnih sistemov



Sl. 5.27 Povprečno število literalov v hipotezah, ko sistemi niso pravilno rešili problema



Sl. 5.28 Povprečno število literalov hipotezah, ko sistemi pravilno rešijo problem

5.4 Domena append

5.4.1 Opis domene

Celotna domena je sestavljena iz seznamov z največjo dolžino pet, s petimi različnimi elementi, brez ponavljanja elementov v seznamih. Domena opisuje sestavljanje dveh seznamov v tretjega. Pozitivnih primerov je 1.631, negativnih primerov pa je 1.783.545. Če bi povečali dolžino seznama na osem, kar bi lahko pomagalo sistemom pri reševanju problemov, bi število pozitivnih primerov naraslo na 876.809, negativnih pa na približno $1{,}3*10^{15}$, kar bi ali povečalo redkost učne množice ali pa bi morali izredno povečati velikost le-teh; s tem bi še podaljšali čas izvajanja. V tej domeni sta bila prva dva seznama definirana kot vhodna parametra, tretji seznam pa je bil izhodni parameter. Primeri so bili geneririani s predikatom vidnim na Sl. 5.29.

```
append([],A,A).
append([A|B],C,[A|E]):-
        append(B,C,E).
```

Sl. 5.29 Definicija uporabljena za generiranje učnih primerov

Poskusi so potekali z različnimi verzijami sistema HYPER^2:

- v tabelah in slikah razviden kot hyper, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyperG, ki si zapomni podatke o pokrivanju primerov na nivoju hipotez, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- Hyper^{2}Beam (HyperB), ki uporablja pokrivanje na nivoju stavkov, informacijo o vhodno/izhodnih spremenljivkah in iskanje s snopom.

Poleg tega smo uporabili tudi sisteme FOIL 6.4 (foil), CPROGOL 4.4 (progol), ALEPH v3 (aleph), SPChillin 1.0A prenesen na Sicstus Prolog (chillin) in Markus V1.1 (markus). Opravili smo 11 serij poskusov z različnim številom primerov (tab. 5.3). V vsaki seriji je bilo opravljenih 100 poskusov z naključno izbranimi primeri (vsi sistemi so v isti ponovitvi dobili enake primere).

Oznaka serije	Število pozitivnih primerov (delež od vseh poz. primerov)	Število negativnih primerov (delež od vseh neg. primerov)
1 (2ex 2ex)	2 (0,12%)	2 (0,000011%)
2 (5ex 5ex)	5 (0,31%)	5 (0,000028%)
3 (10ex 10ex)	10 (0,61%)	10 (0,000056%)
4 (15ex 15ex)	15 (0,92%)	15 (0,000084%)
5 (20ex 20ex)	20 (1,23%)	20 (0,000112%)
6 (30ex 30ex)	30 (1,84%)	30 (0,000168%)
7 (50ex 50ex)	50 (3,07%)	50 (0,000280%)
8 (100ex 100ex)	100 (6,13%)	100 (0,000560%)
9 (200ex 200ex)	200 (12,26%)	200 (0,001120%)
10 (500ex 500ex)	500 (30,66%)	500 (0,002800%)
11 (1000ex 1000ex)	1000 (61,31%)	1000 (0,005600%)

tab. 5.3 Opis posameznih serij domene Append

Čeprav je bil čas izvajanja omejen že pri prejšnjih domenah, do sedaj noben izmed sistemov ni nikoli dosegel meje. Pri tej domeni pa je drugače, zato je treba poudariti, da je bil čas, ki so ga sistemi imeli na voljo, omejen na eno uro (3600 sekund) za vsak poskus.

5.4.2 Rezultati

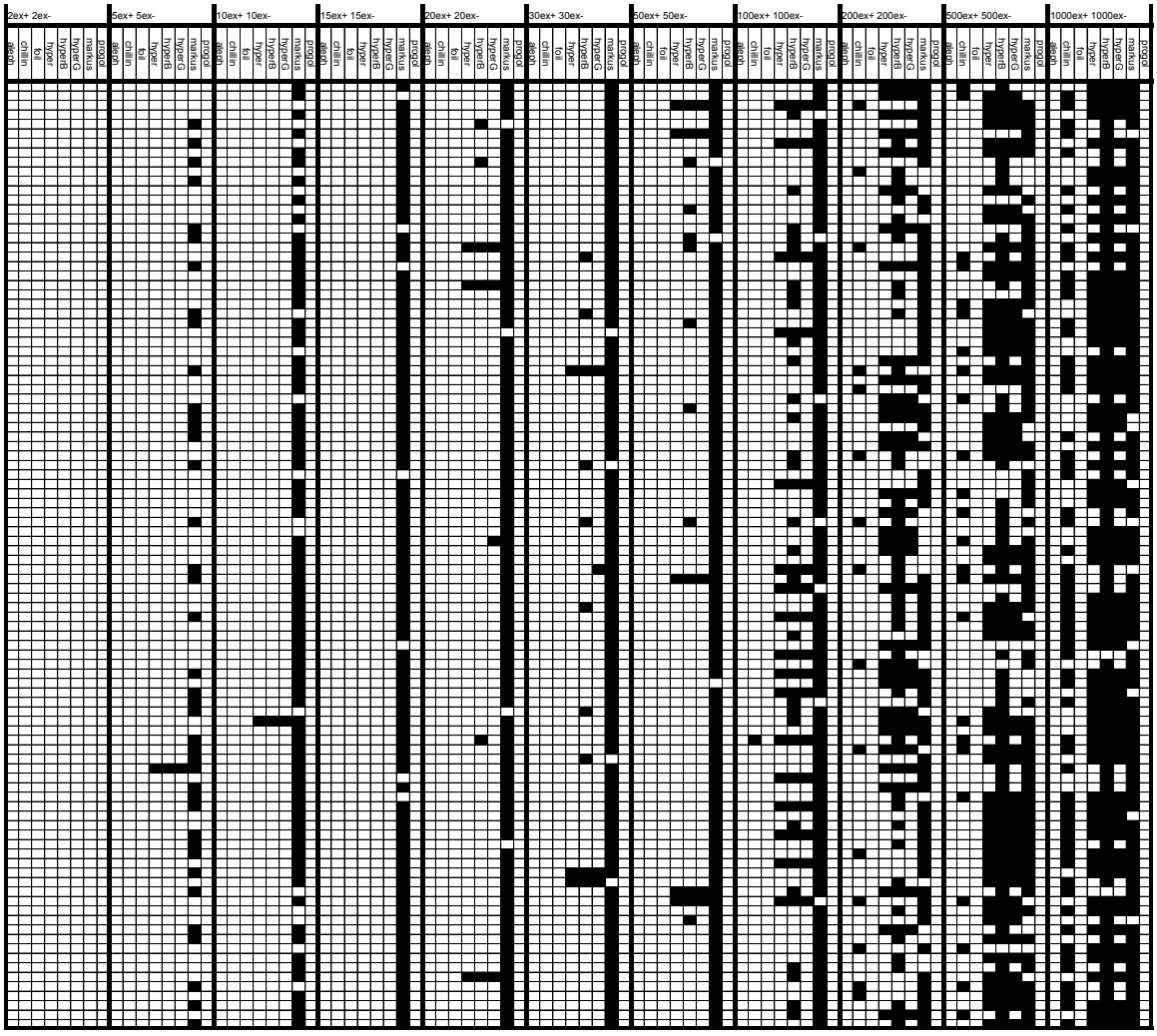
Od vseh sistemov so samo CHILLIN, HYPER^2 in MARKUS uspešno rešili probleme v domeni (generirali hipotezo, ki pokrije vse pozitivne in nobenega negativnega primera v

domeni) znotraj ene ure, ALEPH, FOIL in PROGOL pa so bili neuspešni v vseh 1.100 poskusih, kar je lahko posledica krajših seznamov (vendar je recimo testiranje sistema FOIL opisano v [30] potekalo na seznamih dolžine tri in štiri, vendar na polnih množicah). ALEPH je vedno dosegel časovno omejitev in je bil ustavljen po eni uri. FOIL je v vseh hipotezah dodal stavek, ki je imel sezname z nedefiniranim repom. PROGOL pa je vedno samo naštel pozitivne učne primere. Pri uspešnih sistemih pa izstopa MARKUS, ki (kot je vidno na Sl. 5.30 in Sl. 5.31) daleč pred ostalimi sistemi doseže večino pravilno rešenih problemov. Po uspešnosti nato sledi HYPER^2 z iskanjem s snopom, kmalu za njim sta obe verziji HYPER^2 z iskanjem najprej najboljši. Zanimivo je, da od šeste serije dalje večino poskusov, ki povzročajo probleme sistemu MARKUS, reši vsaj ena verzija sistema HYPER^2. Zadnji med še (delno) uspešnimi sistemi je CHILLIN. Kot že rečeno, noben izmed preostalih sistemov ni uspel rešiti niti enega problema v katerikoli seriji.

MARKUS ima v tej domeni najmanj problemov, vendar kljub temu, da zelo kmalu doseže uspešnost nad 80%, nikoli ne doseže 100%. Neuspehi so kombinacija napačnih rezultatov, ko v hipotezo vstavi stavek, ki praktično določi, da je rezultat spenjanja seznamov enak prvemu seznamu, ne glede na to, kakšen je drugi seznam, in poskusov, kjer je dosegel časovno omejitev ene ure. Število napak prvega tipa se s številom učnih primerov zmanjšuje, število drugih napak pa se povečuje do devete serije. V deveti je, kakor kaže, število primerov dovolj veliko, da se poveča čas obdelave stavkov, ki so kandidati za vključitev v hipotezo, a ne dovolj veliko, da bi se preprečilo zahajanje sistema v neproduktivne smeri. Posledica take kombinacije je padec v učinkovitosti.

Največji problem, ki so ga imele vse verzije sistema HYPER^2, je bil določiti ustavitveni pogoj rekurzije. Zaradi pomanjkanja ustreznih negativnih učnih primerov, je bil le-ta običajno presplošen. Občasno sta se tudi verziji z iskanjem najprej najboljši zaustavili ob interni omejitvi izostrenih 700 hipotez, kar običajno pomeni, da sta zašli iz ustrezne smeri.

Sistemu CHILLIN je največje težave delala zaustavitev združevanja primerov in s tem posploševanja stavkov. V tej domeni je običajno stavke premalo posplošil in so tako pokrivali premalo primerov. Tipičen primer je recimo stavek, ki naj bi pokrival zaustavitveni pogoj rekurzije. Tak pravilen stavek naj bi deloval vse od sezamov brez elementov (torej dolžine nič) dalje, CHILLIN pa je pogosto sestavil stavek, ki recimo deluje le od dolžine tri dalje (ali pa celo deluje samo za dolžino tri).



Sl. 5.30 Primerjava pravilno rešenih problemov domene Append

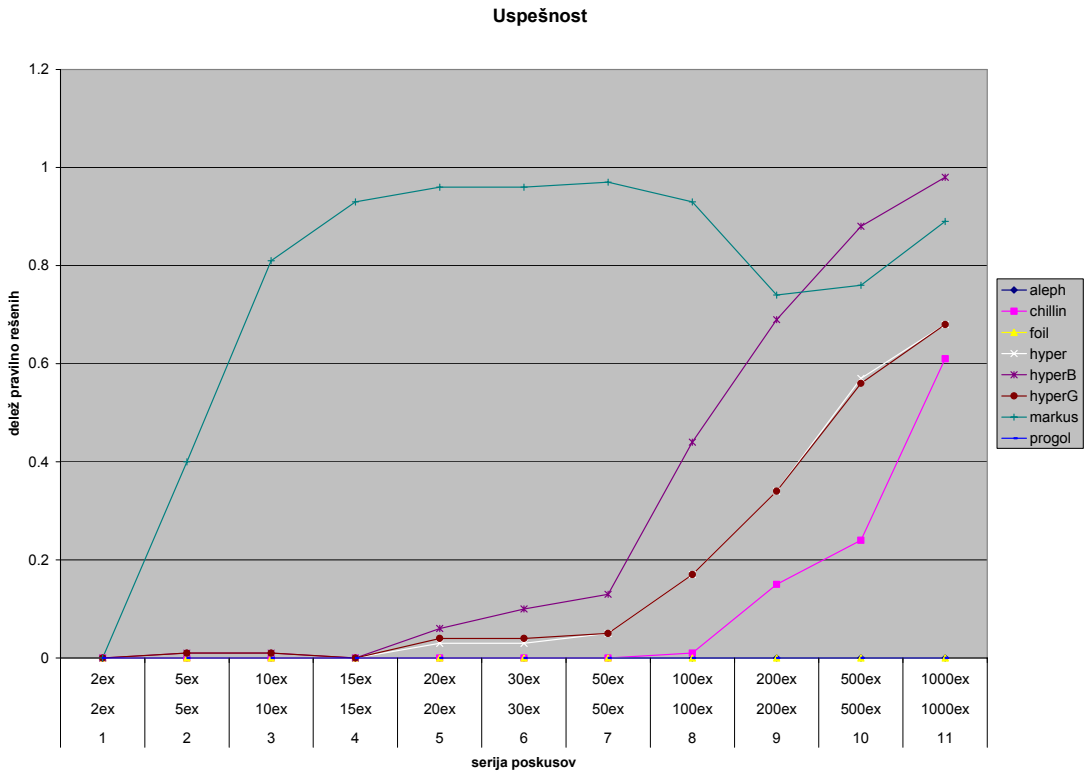
Če pogledamo čase, potrebne za obdelavo (Sl. 5.32) vidimo, da je bil ALEPH vedno zaustavljen v eni uri. Poleg tega se na prvi sliki, ki prikazuje čas, vidi tudi, da sta bili obe verziji sistema HYPER[2], ki uporabljata iskanje najprej najboljši, počasnejši od verzije z iskanjem s snopom. To je posledica tega, da ti dve verziji občasno zaideta na stranske poti (in se včasih z njih vrneta) pri preiskovanju prostora. Verzija z iskanjem s snopom pa pogosto najde rezultat v sedmi generaciji ali pa celo prej (če najde nepravilen rezultat), kar pomeni, da v tej domeni iskanje s snopom preišče manj hipotez kot iskanje najprej najboljši. Poleg tega je s slike razvidno, da se povprečni čas, ki ga potrebujeta verziji z iskanjem najprej najboljši, povečuje s številom učnih primerov, dokler konsistentno neuspešno rešujeta probleme. Ko se uspešnost začenja povečevati, se začenja zmanjševati tudi povprečni čas reševanja.

Čase sistema MARKUS (bolje vidne na Sl. 5.33) si je težje razlagati. Najbolj verjetna razlaga je, da so povprečni časi in predvsem povprečni časi nepravilnih rešitev (Sl. 5.34), sorazmerni s številom zaustavitev sistema po pretečeni uri delovanja, ki je posledica preiskovanja preiskovalnega prostora v napačni smeri. Na to kažejo predvsem povprečni časi potrebni za

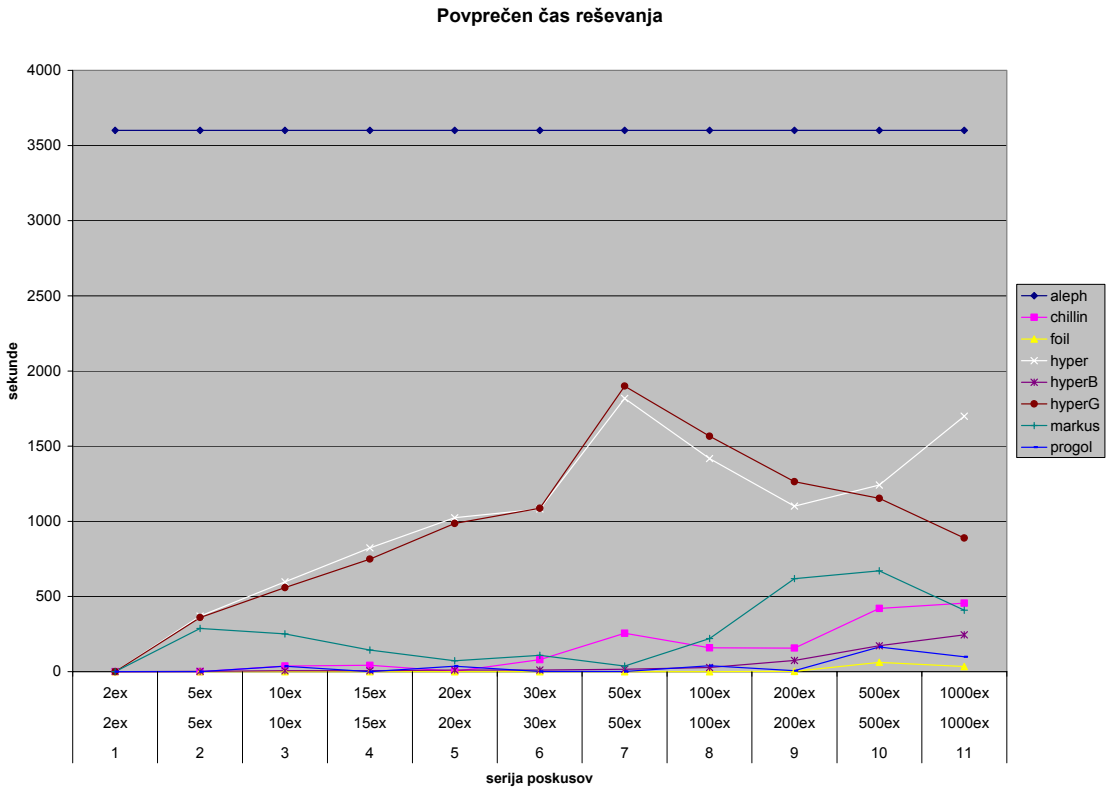
pravilno rešitev (Sl. 5.35), ki se pri sistemu MARKUS še zdaleč ne približajo eni uri. Najdaljši posamičen čas, potreben za pravilno rešitev, je bil v enajsti seriji 1.990 sekund, drugi pa, prav tako v enajsti seriji, 534 sekund. To, da je bil že deseti najdaljši čas potreben za pravilno rešitev že pod 30 sekund, kaže, da se je MARKUS včasih sicer sposoben vrniti na pravo pot pri preiskovanju, vendar se to zgodi le redko.

Pravzaprav vsem sistemom narašča porabljen čas s številom učnih primerov. Vendar pa porabljen čas pri nepravilnih rešitvah praviloma narašča dosti hitreje kot pri pravilnih, zato se povprečni čas pri vseh rešitvah ne povečuje monotono, ampak se pri povečani učinkovitosti lahko celo zmanjša.

V tej domeni se tudi prvič pojavi razlika med obema verzijama HYPER[2], ki preiskujeta prostor po sistemu najprej najboljši. HyperG, ki računa pokritost na nivoju hipotez, je namreč predvsem pri pravilnih rešitvah v serijah z več učnimi primeri hitrejši od verzije, ki računa pokritost glede na stavke. Izkaže se, da se pri računanju pokritosti glede na hipoteze včasih izvržejo nekatere hipoteze kot neprimerne, medtem ko se pri drugi verziji ne. In te hipoteze lahko zapeljejo sistem v slepo ulico, kjer se porablja čas.

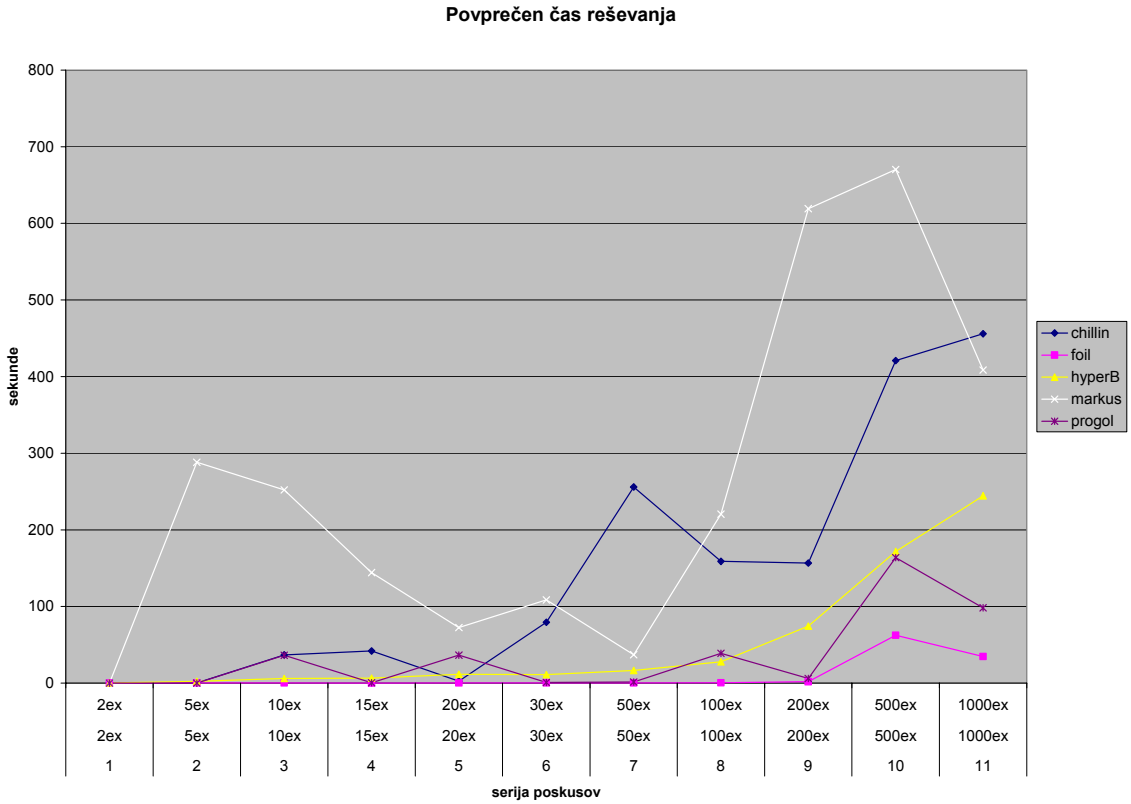


Sl. 5.31 Uspešnost različnih sistemov na domeni Append

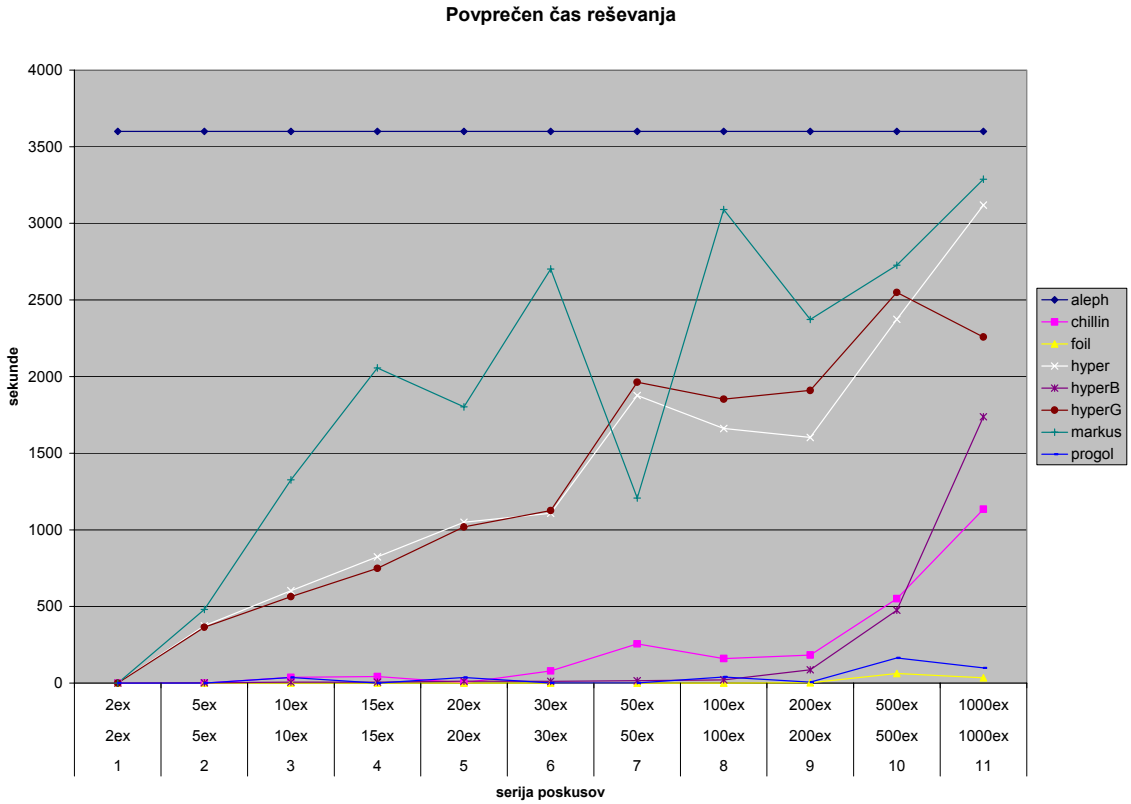


Sl. 5.32 Povprečna poraba časa sistemov na en poskus v posamezni seriji

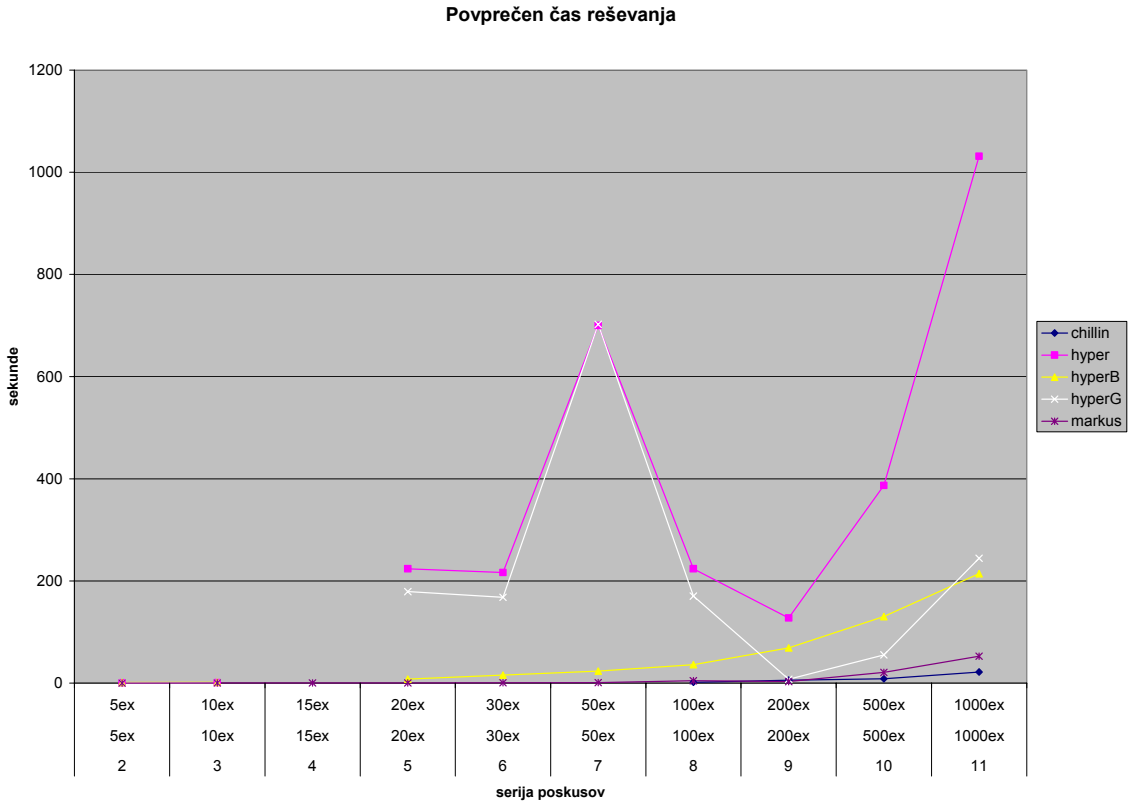
Pri velikosti induciranih hipotez, vidnih na Sl. 5.36, izstopa PROGOL, ki samo našteje učne primere; od preostalih sistemov (Sl. 5.37) prav tako izstopa FOIL, ki mu uspe generirati kompleksne, a nepravilne hipoteze (največja hipoteza, ki jo generira, ima 15 stavkov in 51 literalov, ne da bi šteli literale potrebne za obdelavo seznamov). Od preostalih sistemov (torej brez sistemov PROGOL in FOIL), vidnih na Sl. 5.38, so hipoteze večinoma velike med dvema in štirimi literali, kar se giblje zelo blizu velikosti pravilne hipoteze. Rahlo izstopata samo MARKUS, za katerega je treba vedeti, da vrne prazno hipotezo če mu ne uspe rešiti problema in CHILLIN, ki ob neuspehih običajno premalo posploši stavke ter jih zaradi tega potrebuje več, da pokrije primere. Če pogledamo samo neuspešne poskuse (Sl. 5.39 - Sl. 5.41), se vsa zgodba ponovi, s tem da je razpon pri sistemih CHILLIN, HYPER^2 in MARKUS še nekoliko večji. Pri pravilnih rešitvah (Sl. 5.42) so sistemi, z izjemo sistema CHILLIN, ki občasno pusti kakšen redundanten stavek v hipotezi, konsistentno generirali hipoteze optimalne velikosti.



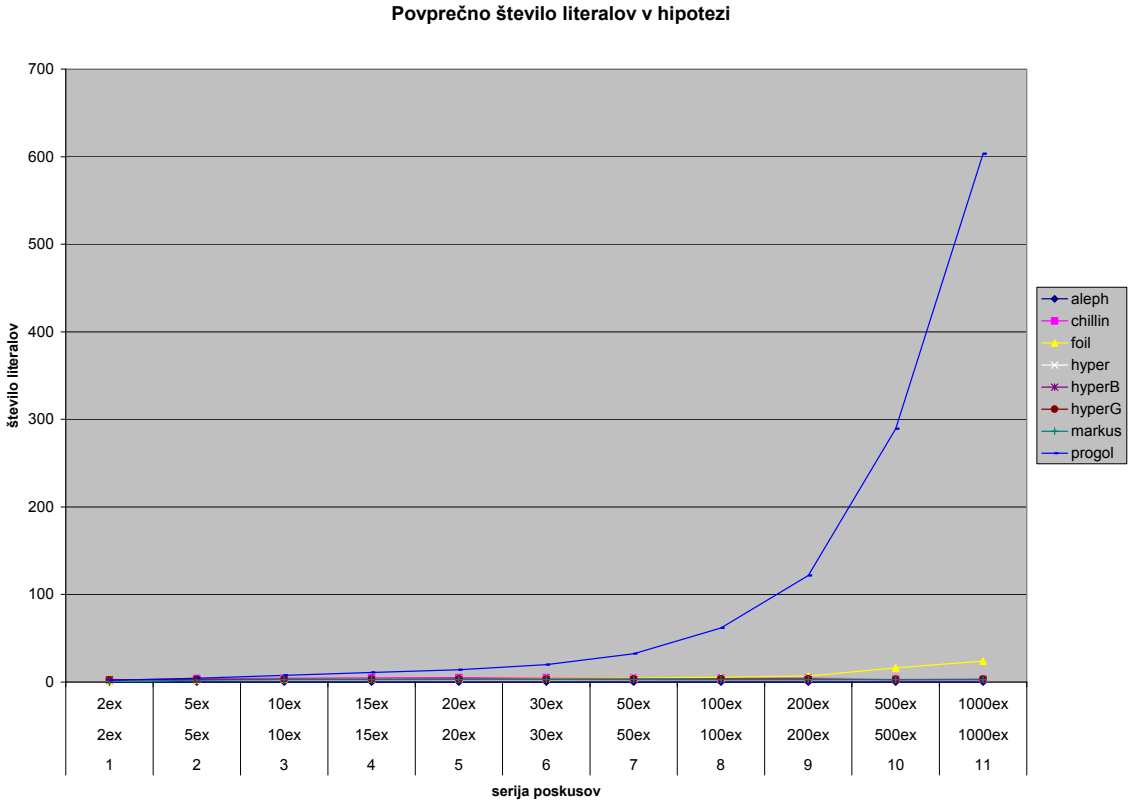
Sl. 5.33 Povprečna poraba časa sistemov (brez sistemov ALEPH, Hyper in HyperG) v posmeznih serijah



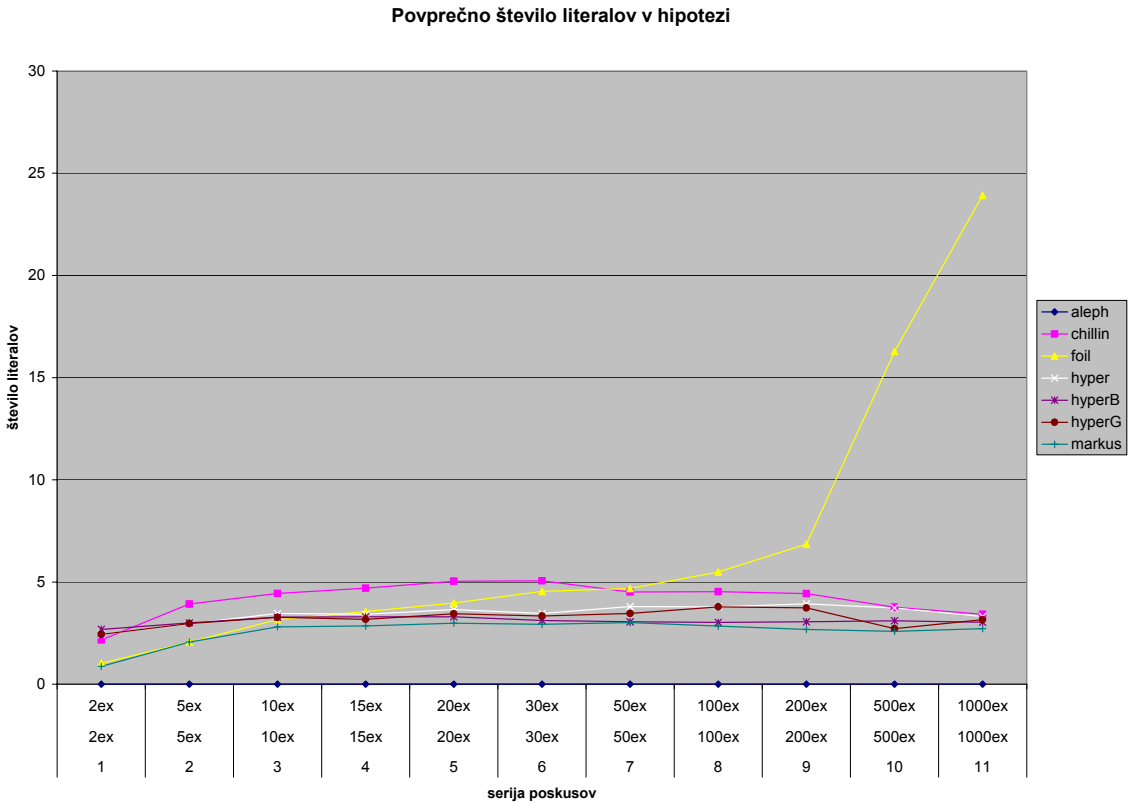
Sl. 5.34 Povprečna poraba časa različnih sistemov v posamezni seriji, ko sistem ni pravilno rešil problema



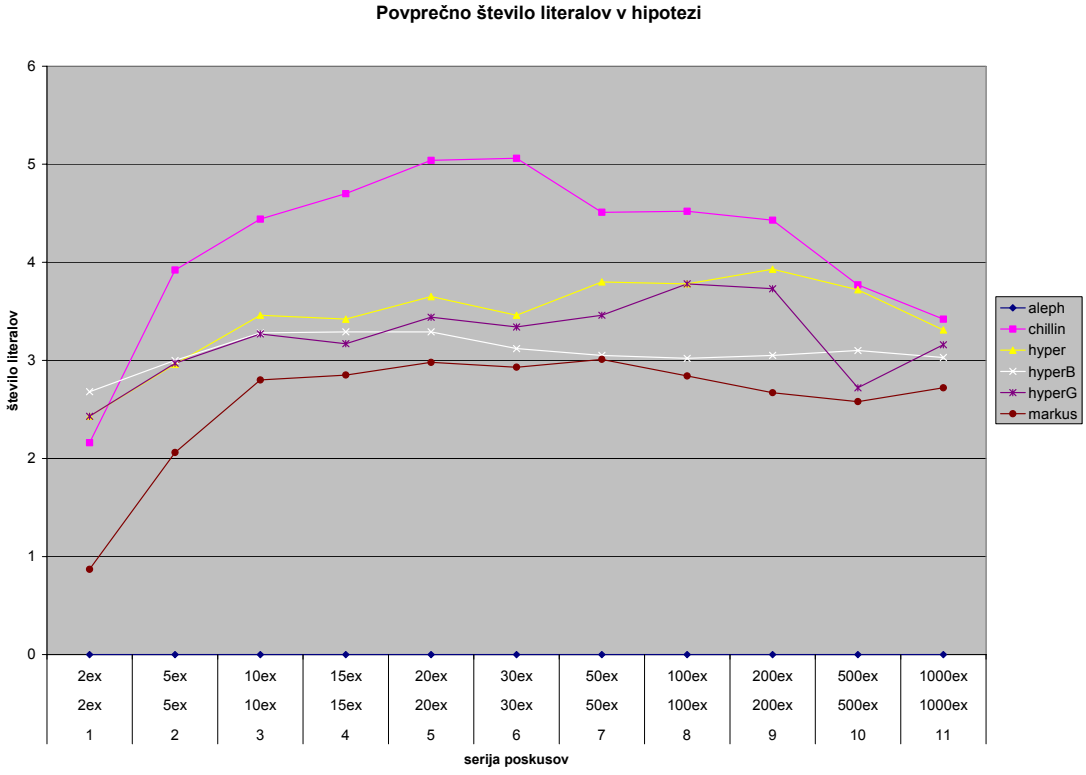
Sl. 5.35 Povprečna poraba časa različnih sistemov v posamezni seriji, ko je sistem pravilno rešil problem



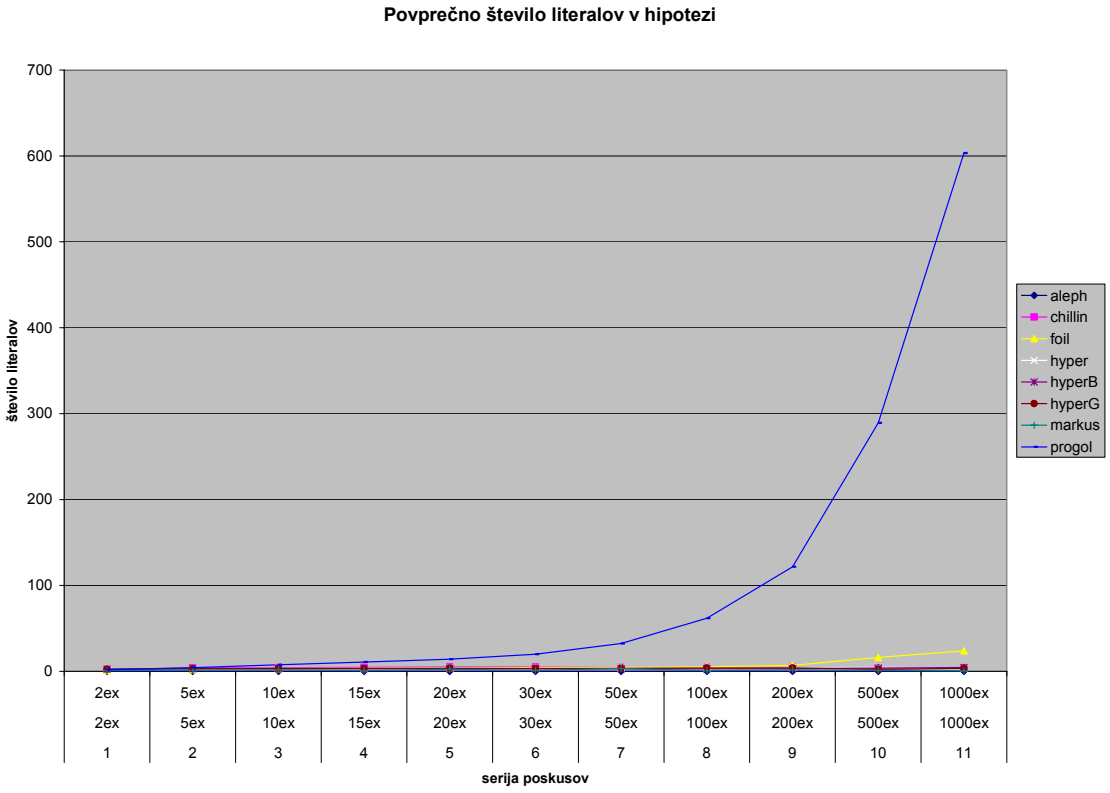
Sl. 5.36 Povprečno število literalov v hipotezah različnih sistemov v posamezni seriji



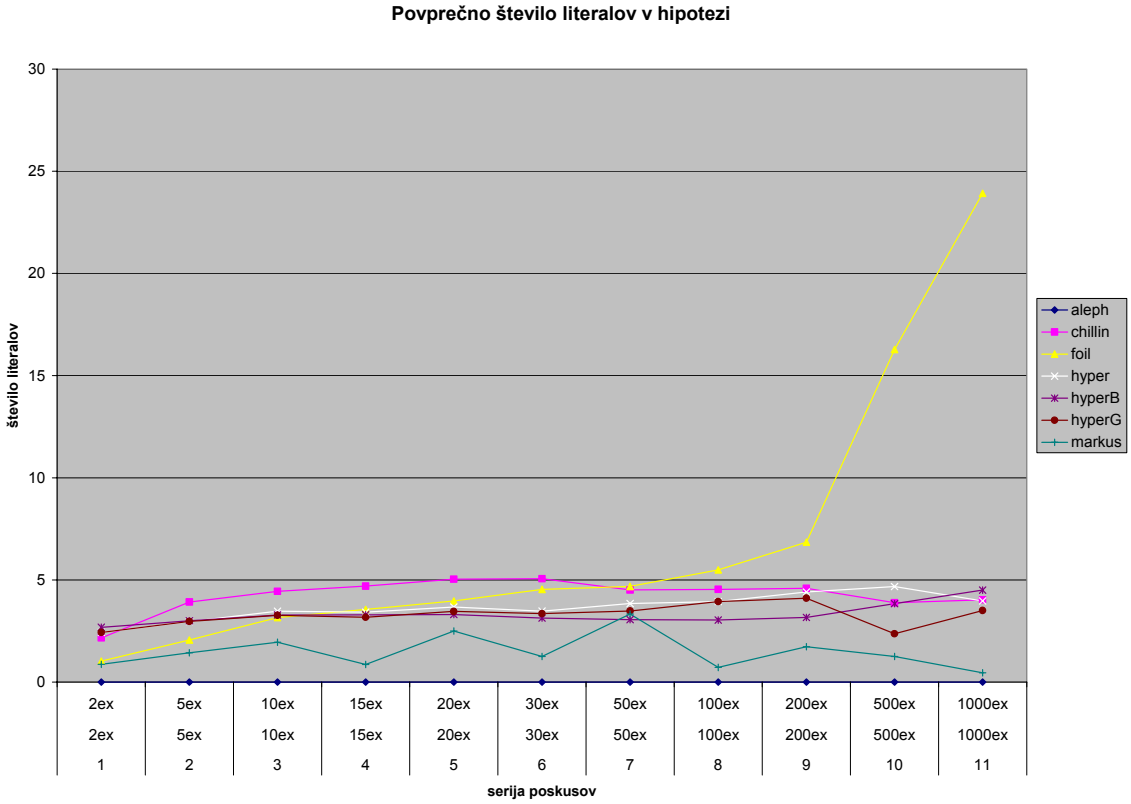
Sl. 5.37 Povprečno število literalov v hipotezah različnih sistemov (brez sistema PROGOL) v posamezni seriji



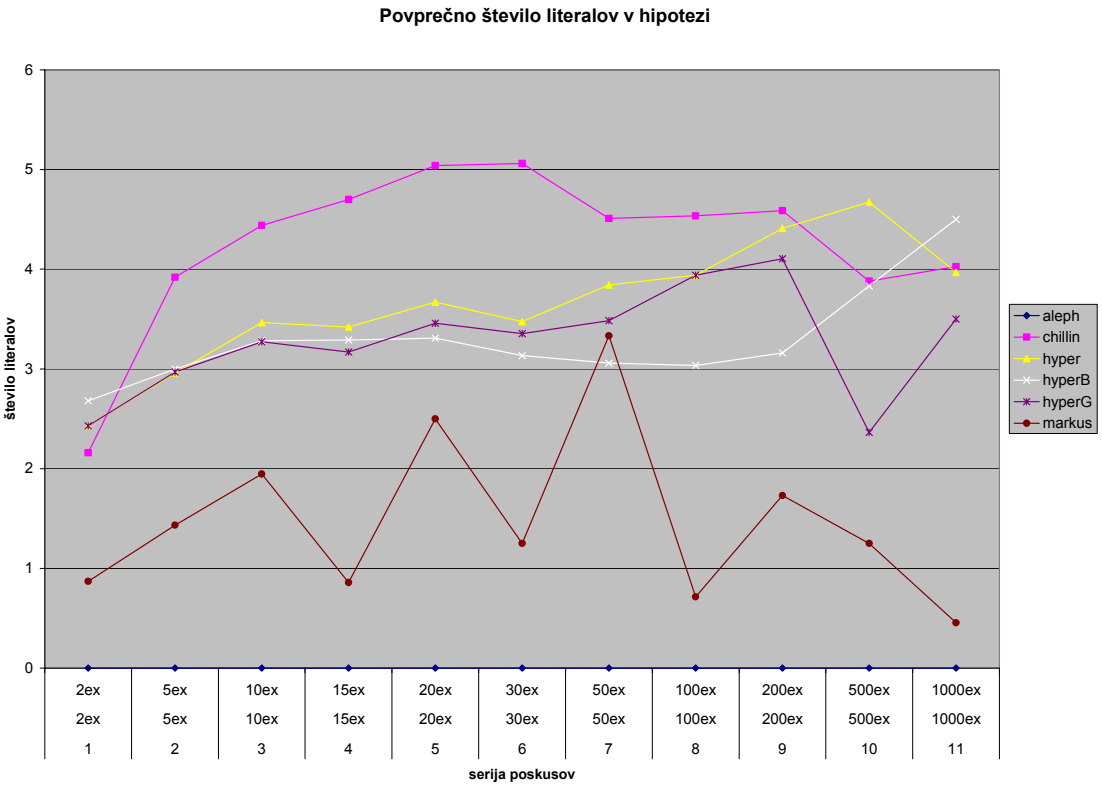
Sl. 5.38 Povprečno število literalov v hipotezah različnih sistemov (brez sistemov FOIL in PROGOL) v posamezni seriji



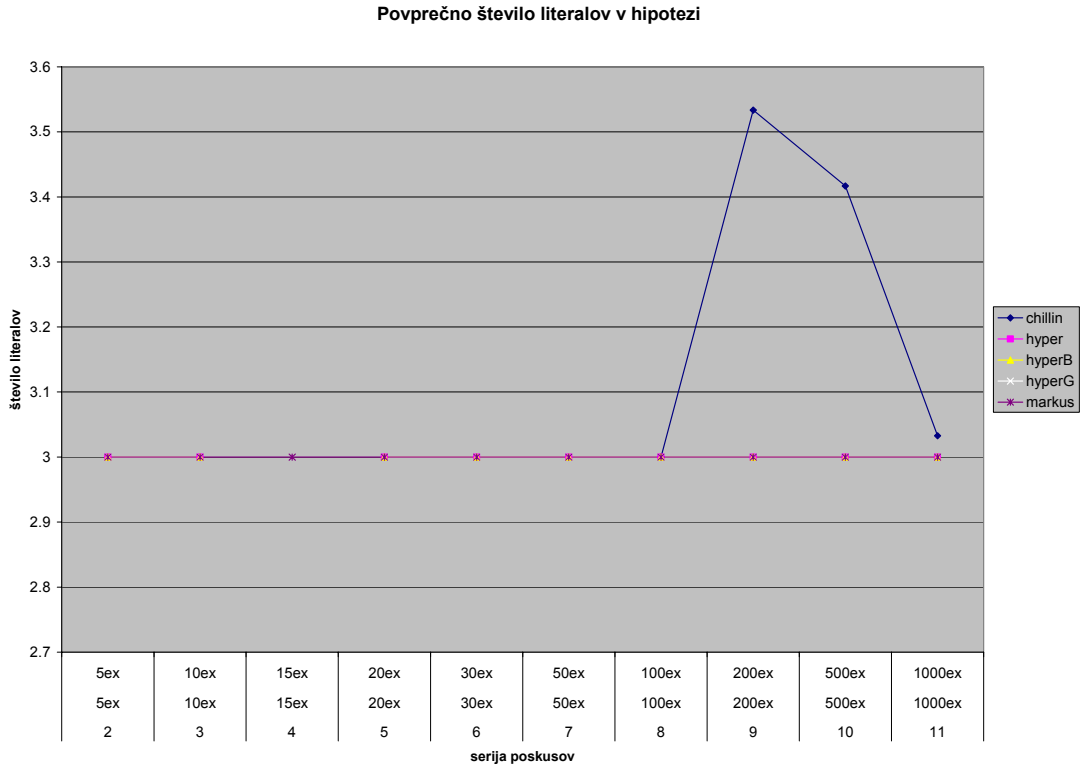
Sl. 5.39 Povprečno število literalov v hipotezah, ko sistem ni pravilo rešil problema



Sl. 5.40 Povprečno število literalov v hipotezah, ko sistem ni pravilno rešil problema (brez sistema PROGOL)



Sl. 5.41 Povprečno število literalov v hipotezah, ko sistem ni pravilno rešil problema (brez sistemov FOIL in PROGOL)



Sl. 5.42 Povprečno število literalov v hipotezah, ko je sistem pravilno rešil problem

5.5 Domena last

5.5.1 Opis domene

Celotna domena je sestavljena iz seznamov z največjo dolžino pet, s petimi različnimi elementi, brez ponavljanja elementov v seznamih. Domena opisuje izpis zadnjega elementa iz seznama. Pozitivnih primerov je 325, negativnih primerov pa je 1305. Učni primeri so bili generirani s pomočjo predikata na Sl. 5.43. V tej domeni je bil seznam vhodni parameter in element izhodni parameter.

```
last(A,[A]).
last(A,[_|B]):-
        last(A,B).
```

Sl. 5.43 Definicija predikata, uporabljena za generiranje učnih primerov

Poskusi so potekali z različnimi verzijami sistema HYPER^2:

- v tabelah in slikah razviden kot hyperE, verzija ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, brez informacije o vhodno/izhodnih spremenljivkah v glavah stavkov, z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyper, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyperG, ki si zapomni podatke o pokrivanju primerov na nivoju hipotez, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- Hyper^{2}Beam (HyperB), ki uporablja pokrivanje na nivoju stavkov, informacijo o vhodno/izhodnih spremenljivkah in iskanje s snopom.

Poleg tega smo uporabili tudi sisteme FOIL 6.4 (foil), CPROGOL 4.4 (progol), ALEPH v3 (aleph), SPChillin 1.0A prenesen na Sicstus Prolog (chillin) in Markus V1.1 (markus). Opravili smo devet serij problemov z različnim številom primerov (tab. 5.4). V vsaki seriji razen zadnje je bilo opravljenih 100 poskusov z naključno izbranimi primeri (vsi sistemi so v isti ponovitvi dobili enake primere).

Oznaka serije	Število poskusov	Število pozitivnih primerov (delež od vseh poz. primerov)	Število negativnih primerov (delež od vseh neg. primerov)
1 (2ex 8ex)	100	2 (0,62%)	8 (0,61%)
2 (3ex 12ex)	100	3 (0,92%)	12 (0,92%)
3 (5ex 20ex)	100	5 (1,54%)	20 (1,53%)
4 (10ex 80ex)	100	10 (3,08%)	80 (6,13%)
5 (20ex 150ex)	100	20 (6,15%)	150 (11,49%)
6 (50ex 200ex)	100	50 (15,38%)	200 (15,33%)
7 (100ex 400ex)	100	100 (30,78%)	400 (30,65%)
8 (200ex 800ex)	100	200 (61,54%)	800 (61,30%)
9 (325ex 1305ex)	1	325 (100%)	1305 (100%)

tab. 5.4 Opis posameznih serij domene Last

5.5.2 Rezultati

Če pogledamo uspešnost posameznih sistemov (Sl. 5.44 in Sl. 5.45) opazimo, da je to zopet ena izmed domen, ki ustrezajo sistemu HYPER^2, še posebno verzijam, ki znajo upoštevati informacijo o vhodnih in izhodnih spremenljivkah v glavi iskanega predikata. Verzija, ki teh podatkov ne upošteva, napačno predvideva, da so vse spremenljivke vhodne, kar pa lahko povzroči konflikte pri rekurzivnih klicih (za rekurzivne klice pa pozna tip spremenljivk). Posledično so lahko spremenljivke neinstancirane, ko bi morale biti instancirane. To pa kot posledico prinese padec uspešnosti.

Kot smo že rekli, je HYPER^2 občutno bolj uspešen kot ostali sistemi. Od vseh verzij se glede uspešnosti najbolje obnese verzija z iskanjem s snopom, kar je razumljivo, saj zaradi širine snopa preišče tudi del prostora, ki bi ga lahko iskanje najprej najboljši izpustil in ima zaradi tega iskanje s snopom manjšo verjetnost, da zgreši pravo hipotezo. Sledita ji obe verziji HYPER^2, ki upoštevata informacije o vhodno/izhodnem tipu spremenljivk. Ti verziji imata probleme le s primeri, ko imajo vsi pozitivni primeri iste dolžine seznama (ter noben izmed negativnih primerov nima enake dolžine seznama), le-te pa lahko verzija z iskanjem s snopom celo reši, če je dolžina seznama večja ali enaka štiri. Z obsežnim zaostankom sledi tudi verzija, ki ne upošteva vhodno/izhodnega tipa spremenljivk. Kljub zaostanku je tudi ta verzija bolj uspešna kot preostali sistemi. Kakor je bilo razvidno iz učnih primerov problemov, ki so bili uspešno (in neuspešno) rešeni, potrebuje ta verzija, da uspešno reši problem (pravzaprav, da pravilno postavi ustavitveni pogoj rekurzije) negativni učni primer, ki ima dolžino seznama največ ena.

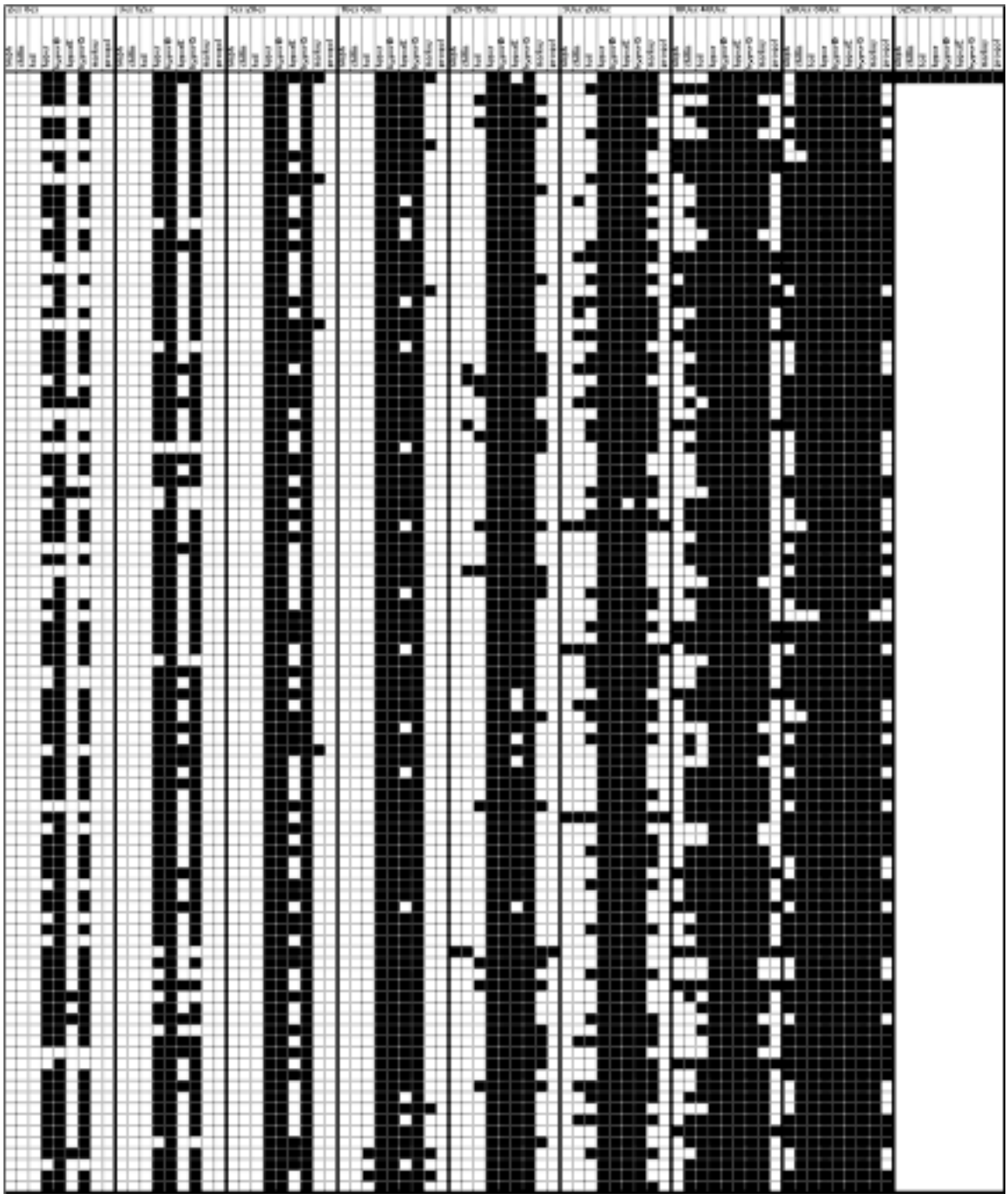
MARKUS po učinkovitosti vodi med preostalimi sistemi. MARKUS, kakor kaže, potrebuje en pozitivni učni primer, ki pokriva zaustavitveni pogoj rekurzije (teh je v celotni domeni le pet), poleg tega pa tudi pozitivni učni primer, ki pokriva prvi rekurzivni klic. Drugače rečeno, potrebuje pozitivni učni primer z dolžino seznama dve.

FOIL, drugi sistem po uspešnosti, za rešitev problema potrebuje učni primer, ki pokriva ustavitveni pogoj rekurzije, poleg tega pa potrebuje pozitivni učni primer, ki se po nekaj korakih lahko prevede na drug pozitivni učni primer. Če se tak primer prevede z več kot enim korakom, mora sistem dobiti tudi primere, ki imajo sezname krajše od števila korakov. Tako recimo, da ima pozitivni učni primer z seznamom dolžine pet, ki se s tremi koraki lahko prevede na drug pozitivni učni primer s seznamom dolžine dve. Iz teh dveh primerov generira stavek, ki seznam skrajša za tri elemente in nato izvede rekurzivni klic. Da taka rekurzija deluje, potrebuje ne le običajni ustavitveni pogoj s seznamom dolžine ena, ampak tudi s seznami dolžin dve in tri. Da pa FOIL vstavi tudi take stavke v hipotezo, potrebuje pozitivne učne pogoje s seznami dolžin ena, dve in tri.

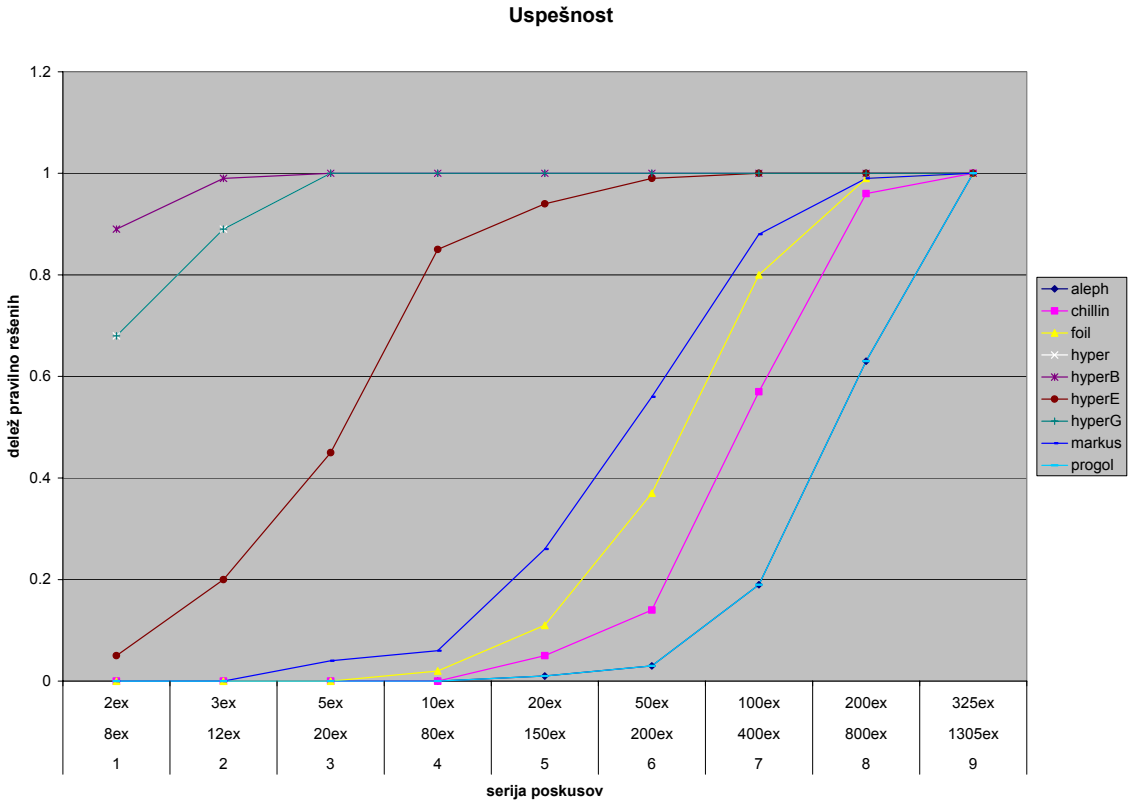
Sledi CHILLIN, ki, samo da pravilno postavi ustavitveni pogoj rekurzije potrebuje vsaj dva pozitivna učna primera s seznami dolžine ena. ALEPH in PROGOL (ki v tej domeni rešita iste probleme) potrebujeta vsaj tri take primere (od možnih petih). To so tako hude zahteve, da se običajno izpolnijo le v učnih množicah z velikim številom primerov, kar pomeni, da so vse ostale zahteve (koliko in kakšne primere potrebuje sistem, da pravilno postavi rekurzivni klic) izpolnjene.

Če pogledamo čas (Sl. 5.46 do Sl. 5.50), ki so ga sistemi porabili pri reševanju problemov, lahko opazimo, da je iskanje s snopom prednost glede učinkovitosti, je pa slabost glede porabljenega časa, saj pregleda več hipotez, in je zaradi tega običajno počasnejši. Ostali sistemi so večino časa v istem velikostnem razredu, le ob zadnjih dveh serijah izstopata ALEPH in FOIL, ki v nasprotju z ostalimi sistemi skrajšata čas delovanja, ko imata na razpolago večino vseh učnih primerov. Edina anomalija je daljši čas, ki ga porabi za pravilne rešitve verzija sistema HYPER^2, ki ne zna upoštevati informacij o vhodno/izhodnem tipu spremenljivk v glavi iskanega predikata, ko ima na voljo manj učnih primerov. Razlaga takega početja je, da, ko ima sistem na voljo najmanj učnih podatkov, vedno najde hipotezo (običajno napačno) v malo korakih, in ko ima na voljo nekaj več učnih podatkov (tako pozitivnih kot negativnih), se zgodi, da sistem preiskuje najprej v napačni smeri in se nato obrne v pravo smer ter najde pravilno rešitev. Posledica tega je porast porabljenega časa. Ko

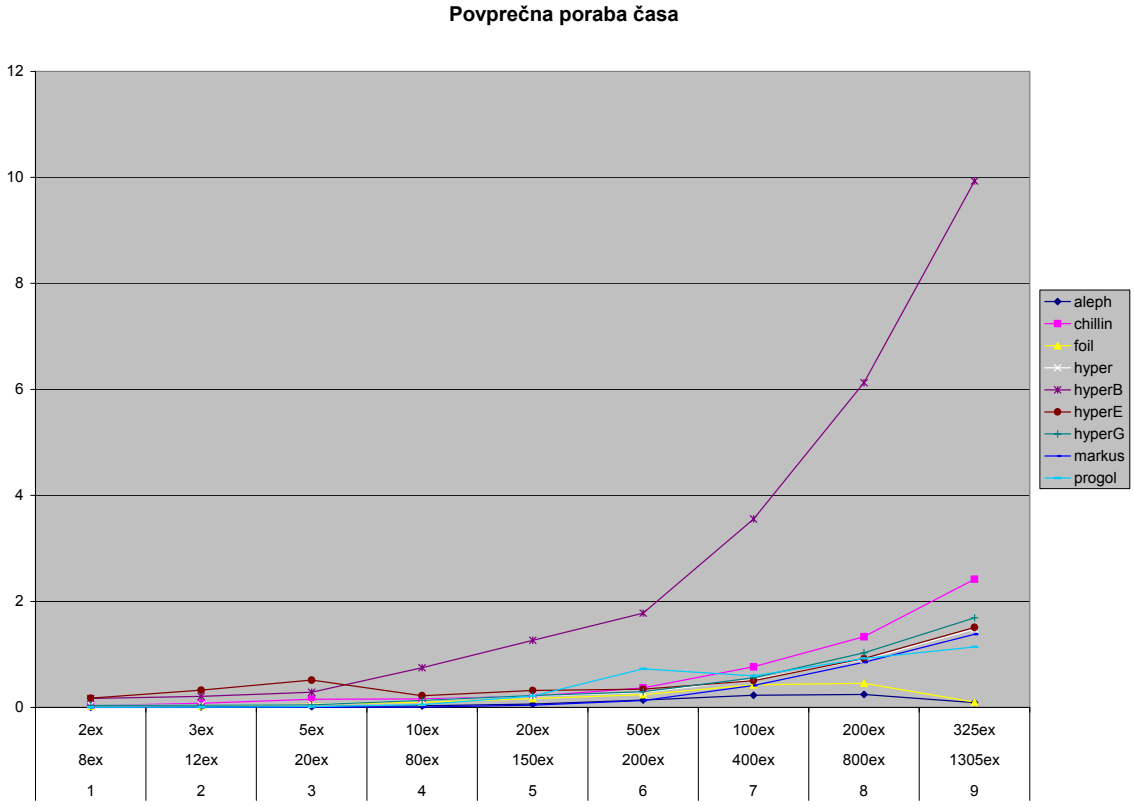
pa ima na voljo še več podatkov, se običajno že od začetka obrne v pravo smer in tako porabi manj časa.



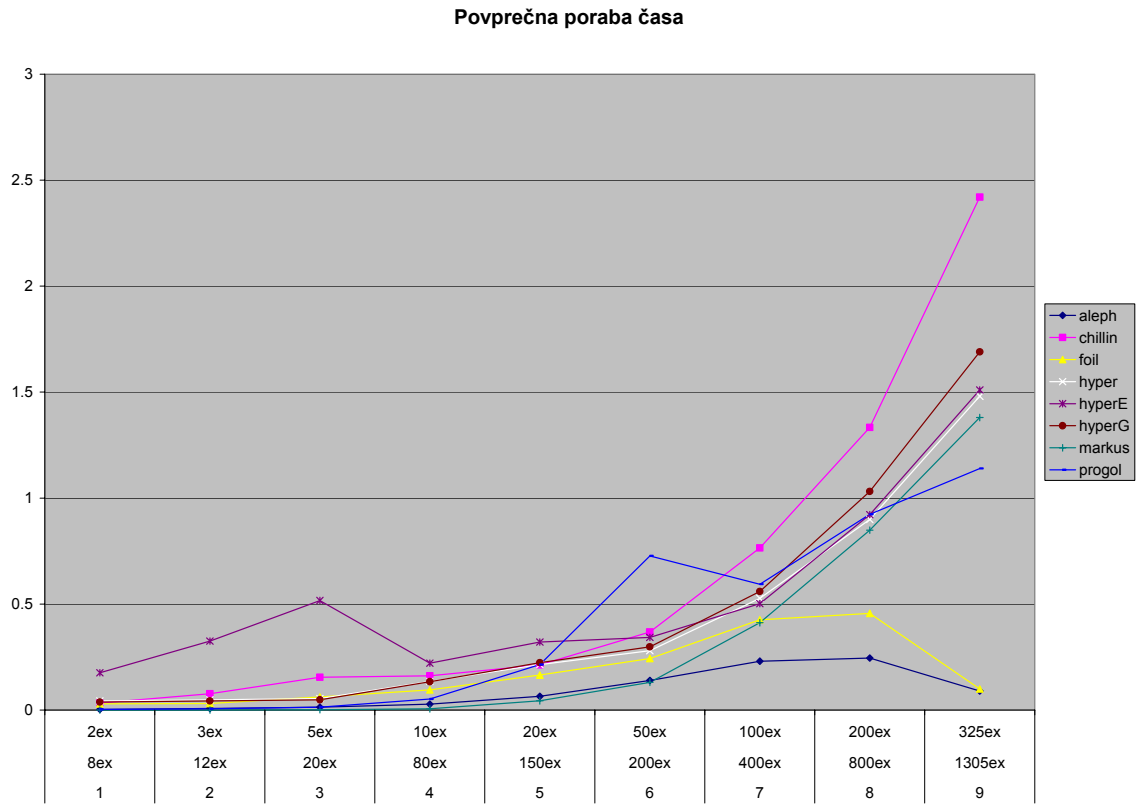
Sl. 5.44 Primerjava pravilno rešenih problemov domene Last



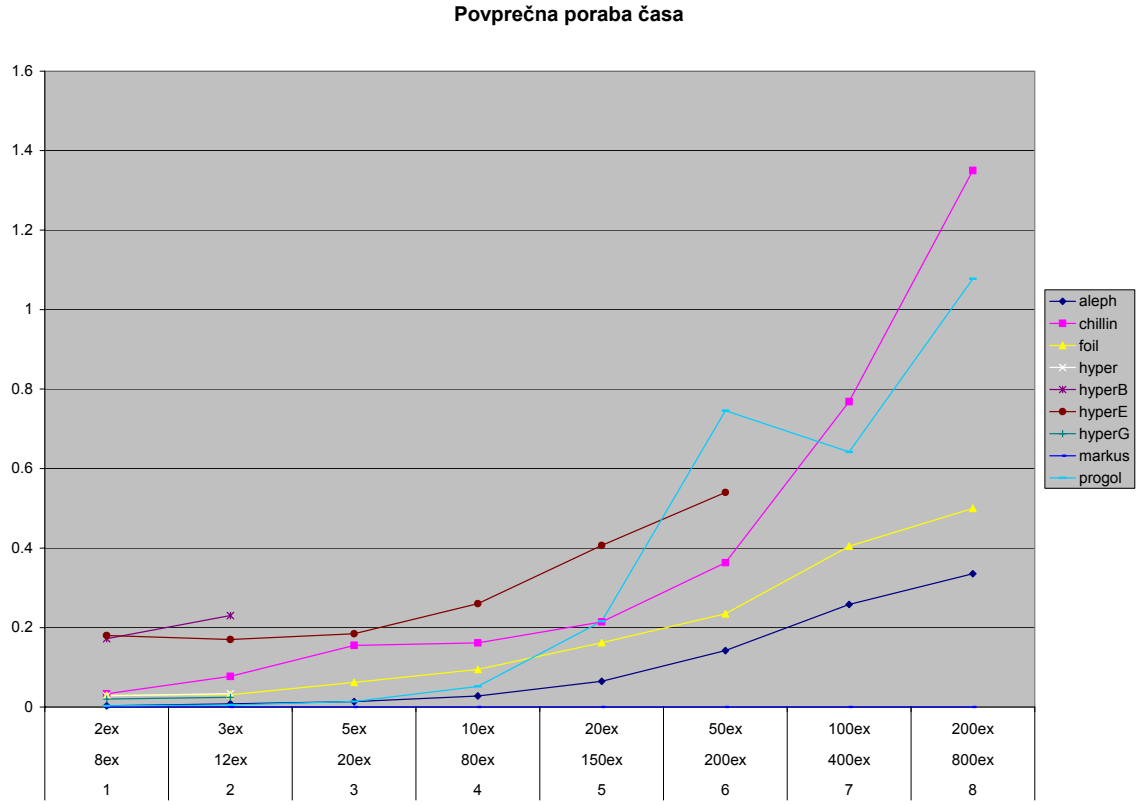
Sl. 5.45 Uspešnost različnih sistemov na domeni Last



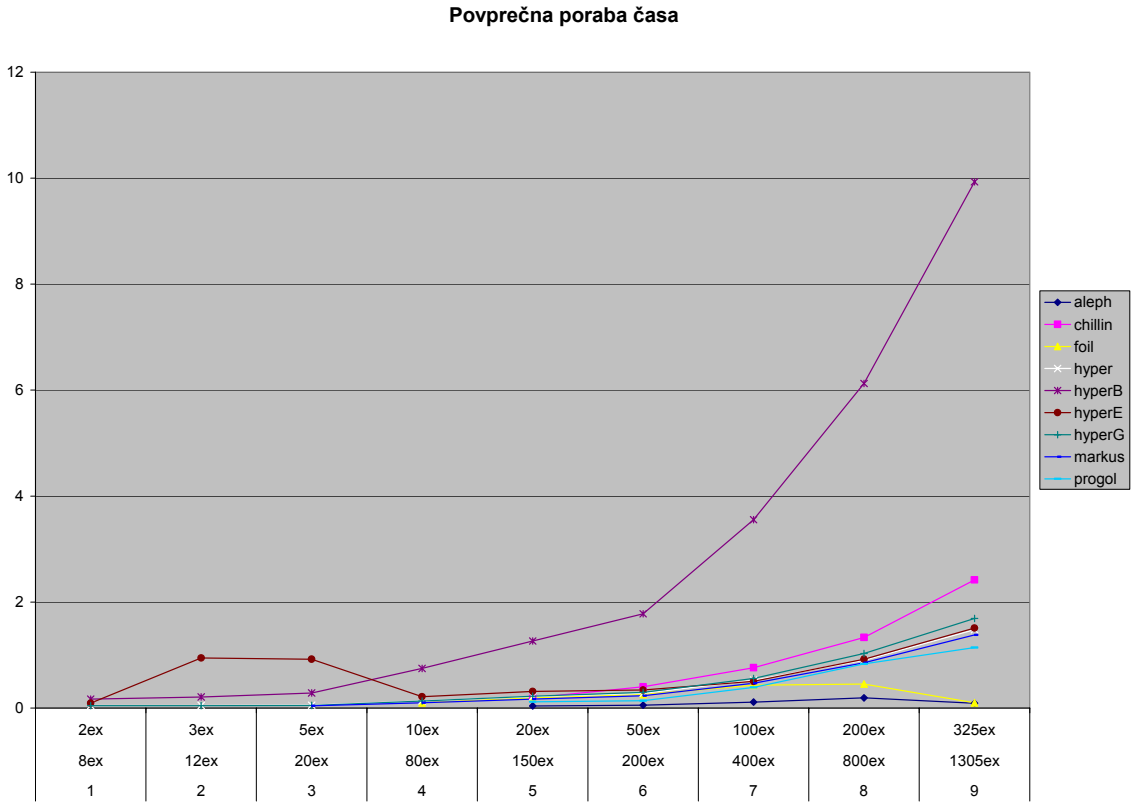
Sl. 5.46 Povprečna poraba časa sistemov v posamezni seriji



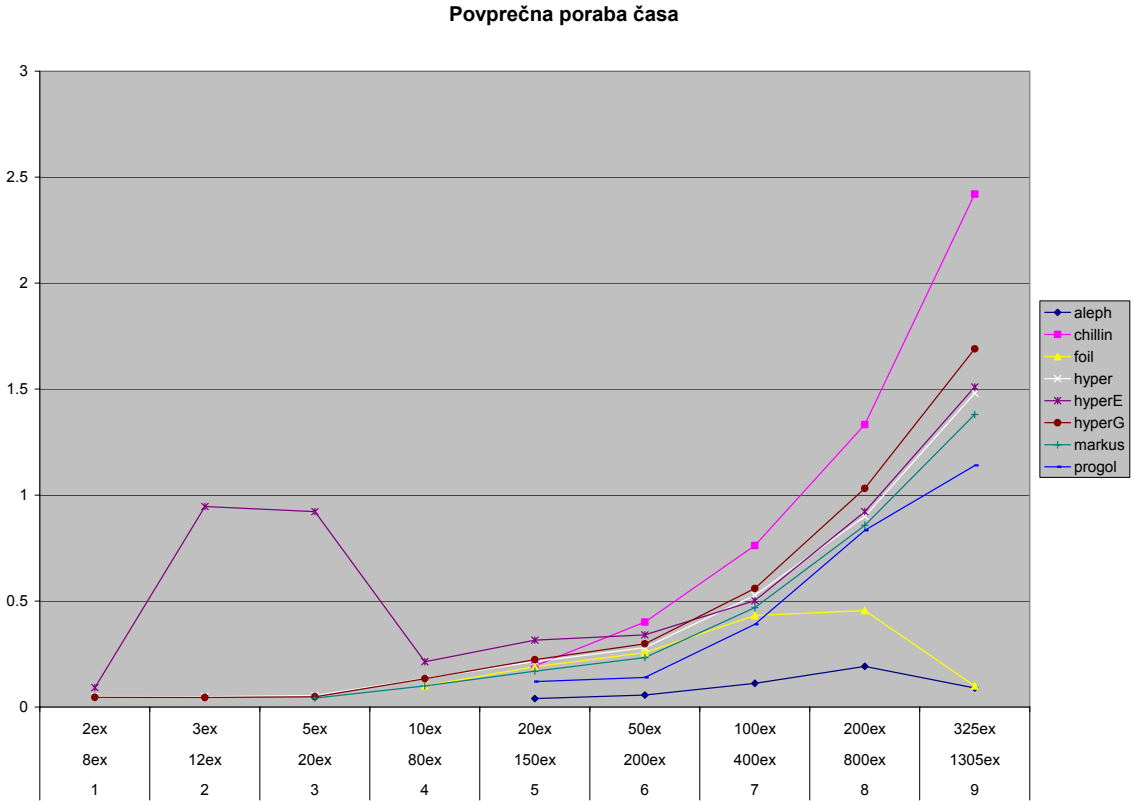
Sl. 5.47 Povprečna poraba časa sistemov v posamezni seriji, brez sistema hyperB



Sl. 5.48 Povprečna poraba časa različnih sistemov v posamezni seriji, ko sistem ni pravilno rešil problema

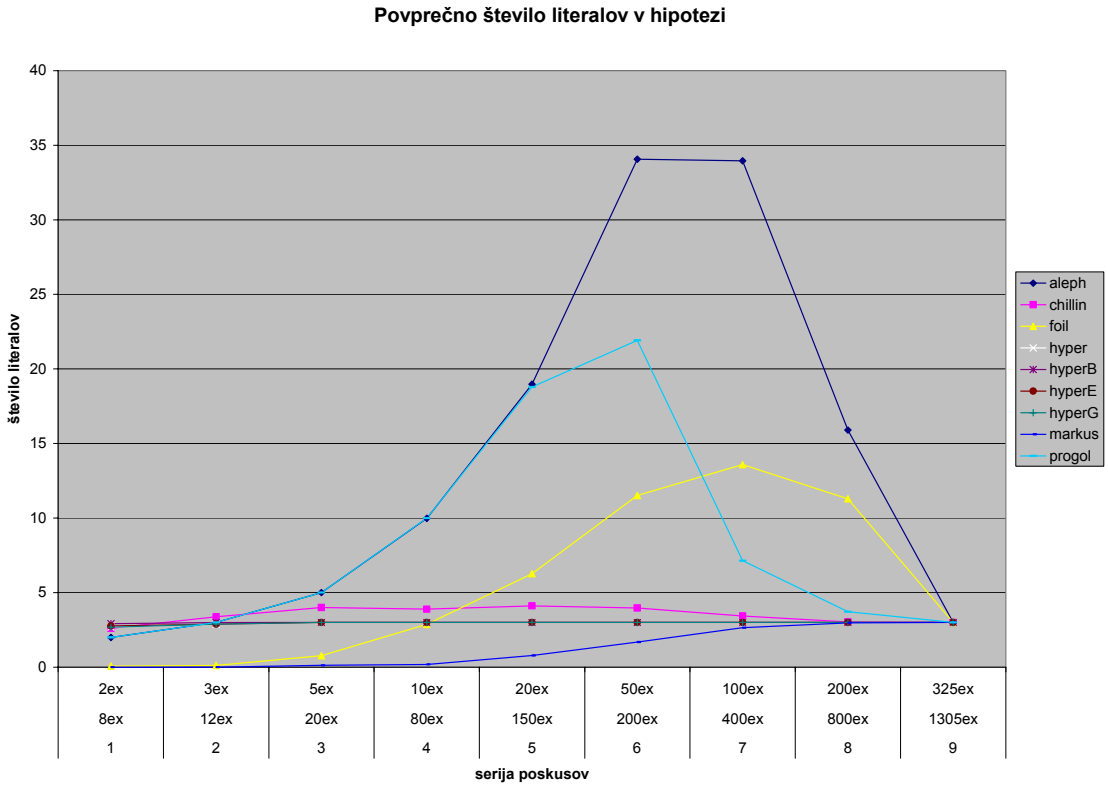


Sl. 5.49 Povprečna poraba časa različnih sistemov v posamezni seriji, ko je sistem pravilno rešil problem

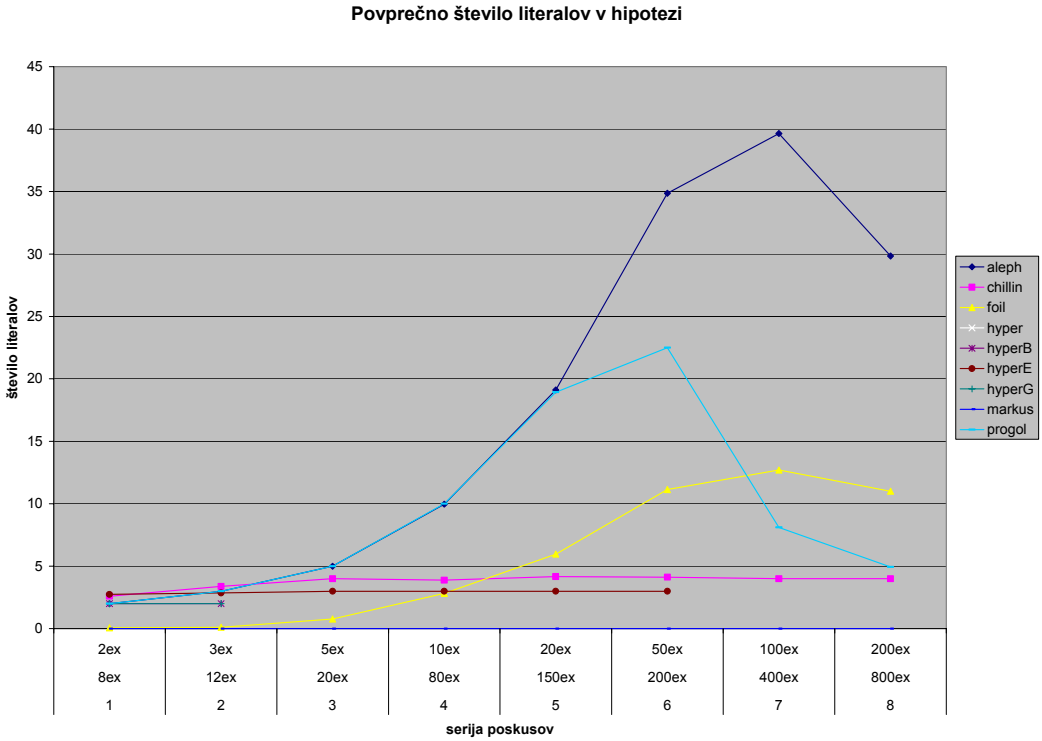


Sl. 5.50 Povprečna poraba časa različnih sistemov v posamezni seriji, ko je sistem pravilno rešil problem (brez sistema hyperB)

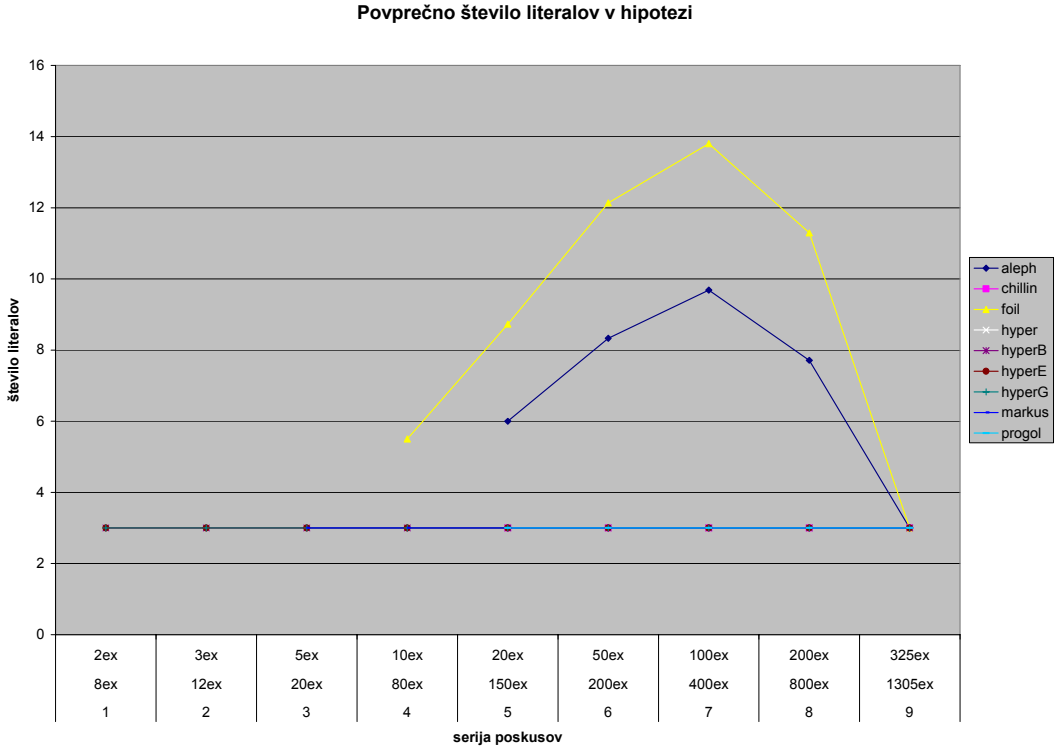
Velikost hipotez (Sl. 5.51 do Sl. 5.53), ki jih generirajo vse verzije sistema HYPER[2], se vedno giblje okoli velikosti optimalne pravilne hipoteze. Podobno je tudi pri sistemu CHILLIN. Ker MARKUS, kadar ne more najti primerne hipoteze, vrne prazno hipotezo, povprečna velikost hipotez tega sistema narašča do optimalne velikosti. Medtem ko ti sistemi vedno vrnejo majhne hipoteze, pa velikost hipotez, ki jih generirajo sistemi ALEPH, FOIL in PROGOL, sprva narašča do izrednih velikosti. ALEPH in PROGOL namreč, kadar ne moreta pokriti vseh pozitivnih učnih primerov, nepokrite primere naštejeta kot dejstva. ALEPH poleg tega celo ne zna odstraniti redundantnih stavkov iz hipoteze, zato so hipoteze še večje. Če ne štejemo naštetih primerov v hipotezah, pa daleč največje hipoteze generira FOIL. Največja hipoteza vsebuje deset stavkov in šestnajst literalov (štete so tudi glave stavkov), hipoteza je pravilna in vsebuje štiri nerekurzivne in šest rekurzivnih stavkov (vsi so različni), čeprav bi zadoščala le dva izmed teh stavkov.



Sl. 5.51 Povprečno število literalov v hipotezah različnih sistemov v posamezni seriji



Sl. 5.52 Povprečno število literalov v hipotezah različnih sistemov v posamezni seriji, ko sistem ni pravilno rešil problema



Sl. 5.53 Povprečno število literalov v hipotezah različnih sistemov v posamezni seriji, ko je sistem pravilno rešil problem

5.6 Domena next

5.6.1 Opis domene

Celotna domena je sestavljena iz seznamov z največjo dolžino pet, s petimi različnimi elementi, brez ponavljanja elementov v seznamih. Domena opisuje izpis elementa, ki v seznamu sledi podanemu elementu. Pozitivnih primerov je 980, negativnih pa je 7.170. Učni primeri so bili generirani s predikatom na Sl. 5.54. V tej domeni sta bila seznam in prvi element podana kot vhodna parametra, naslednji element pa kot izhodni parameter.

```
next(A,B,[A,B|_]).
next(A,B,[_|C]):-
        next(A,B,C).
```

Sl. 5.54 Definicija predikata uporabljena za generiranje učnih primerov domene next

Poskusi so potekali z različnimi verzijami sistema HYPER^2:

- v tabelah in slikah razviden kot hyperE, verzija, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, brez informacije o vhodno/izhodnih spremenljivkah v glavah stavkov, z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyper, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyperG, ki si zapomni podatke o pokrivanju primerov na nivoju hipotez, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- Hyper^{2}Beam (HyperB), ki uporablja pokrivanje na nivoju stavkov, informacijo o vhodno/izhodnih spremenljivkah in iskanje s snopom.

Poleg tega smo uporabili tudi sisteme FOIL 6.4 (foil), CPROGOL 4.4 (progol), ALEPH v3 (aleph), SPChillin 1.0A prenesen na Sicstus Prolog (chillin) in Markus V1.1 (markus). Opravili smo devet serij problemov z različnim številom primerov (tab. 5.5). V vsaki seriji razen zadnje je bilo opravljenih 100 poskusov z naključno izbranimi primeri (vsi sistemi so v isti ponovitvi dobili enake primere).

Oznaka serije	Število poskusov	Število pozitivnih primerov (delež od vseh poz. primerov)	Število negativnih primerov (delež od vseh neg. primerov)
1 (2ex 5ex)	100	2 (0,20%)	5 (0,07%)
2 (5ex 10ex)	100	5 (0,51%)	10 (0,14%)
3 (10ex 30ex)	100	10 (1,02%)	30 (0,42%)
4 (20ex 70ex)	100	20 (2,04%)	70 (0,98%)
5 (50ex 300ex)	100	50 (5,10%)	300 (4,18%)
6 (100ex 800ex)	100	100 (10,20%)	800 (11,16%)
7 (200ex 1500ex)	100	200 (20,41%)	1500 (20,92%)
8 (500ex 3000ex)	100	500 (51,02%)	3000 (41,84%)
9 (980ex 7170ex)	1	980 (100%)	7170 (100%)

tab. 5.5 Opis posameznih serij domene Next

5.6.2 Rezultati

Pri pregledu uspešnosti sistemov (Sl. 5.55 in Sl. 5.56) se opazita dve dejstvi. Najprej, da je v nasprotju z ostalimi domenami MARKUS na tej domeni (tako kot na vseh domenah in kot vsi ostali sistemi je bil pognan s standardnimi parametri) neuspešen. MARKUS ima probleme, ker seznamov nikoli ne razvije dovolj, da bi lahko prišel do uporabnega rezultata (sezname je tu potrebno razviti vsaj za dva elementa). Drugo, nekoliko presenetljivo dejstvo je, da se na tej domeni nekoliko bolje obnese verzija HYPER^2, ki ne zna upoštevati vhodno/izhodnih tipov spremenljivk v glavi iskanega stavka. Izkaže se, da verzije, ki to upoštevajo in imajo zaradi tega tudi malo spremenjen način preiskovanja prostora, zaradi tega v enem koraku dobijo več naslednikov. Povečano število naslednikov povzroči, da se v omejenem številu 700 hipotez ne znajo vrniti iz slepe ulice, medtem ko se verzija, ki tega ne upošteva in generira manj naslednikov v enem koraku, pravočasno vrne iz slepe ulice.

Poleg tega se presenetljivo slabo obnese verzija z iskanjem s snopom (obnese se primerljivo s sistemoma CHILLIN in FOIL). Ker je bila uporabljena verzija iskanja s snopom, ki vedno v snopu obdrži najboljše stare hipoteze, se zgodi, da nobena izmed novih hipotez ni boljša od najslabše v snopu, se iskanje zaustavi. Če bi uporabili širši snop ali pa si ne bi zapomnili starih, a bolje ocenjenih hipotez, bi sistem rešil več problemov, vendar bi bil v drugih domenah počasnejši in bi lahko presegel časovno omejitev (primerjava sistemov naj bi bila narejena brez prilagajanja posameznih sistemov na posamezno domeno – boljše poznavanje določenih sistemov bi jim lahko prineslo nepošteno prednost).

Sistemu HYPER^2 z iskanjem naprej najboljši po učinkovitosti sledita ALEPH in PROGOL. Oba imata minimalno zahtevo, da so med pozitivnimi primeri vsaj trije primeri ustavitvenega pogoja rekurzije (torej, da je podani element prvi v seznamu in iskani element drugi v

seznamu). PROGOL, kakor kaže, potrebuje tudi vsaj tri učne primere, ki pokrivajo prvi rekurzivni klic in dodatno tudi nekaj učnih primerov globljih rekurzivnih klicev. Preostale zahteve sistema ALEPH so malo manj zahtevne, saj potrebuje le tri učne primere prvega klica rekurzije.

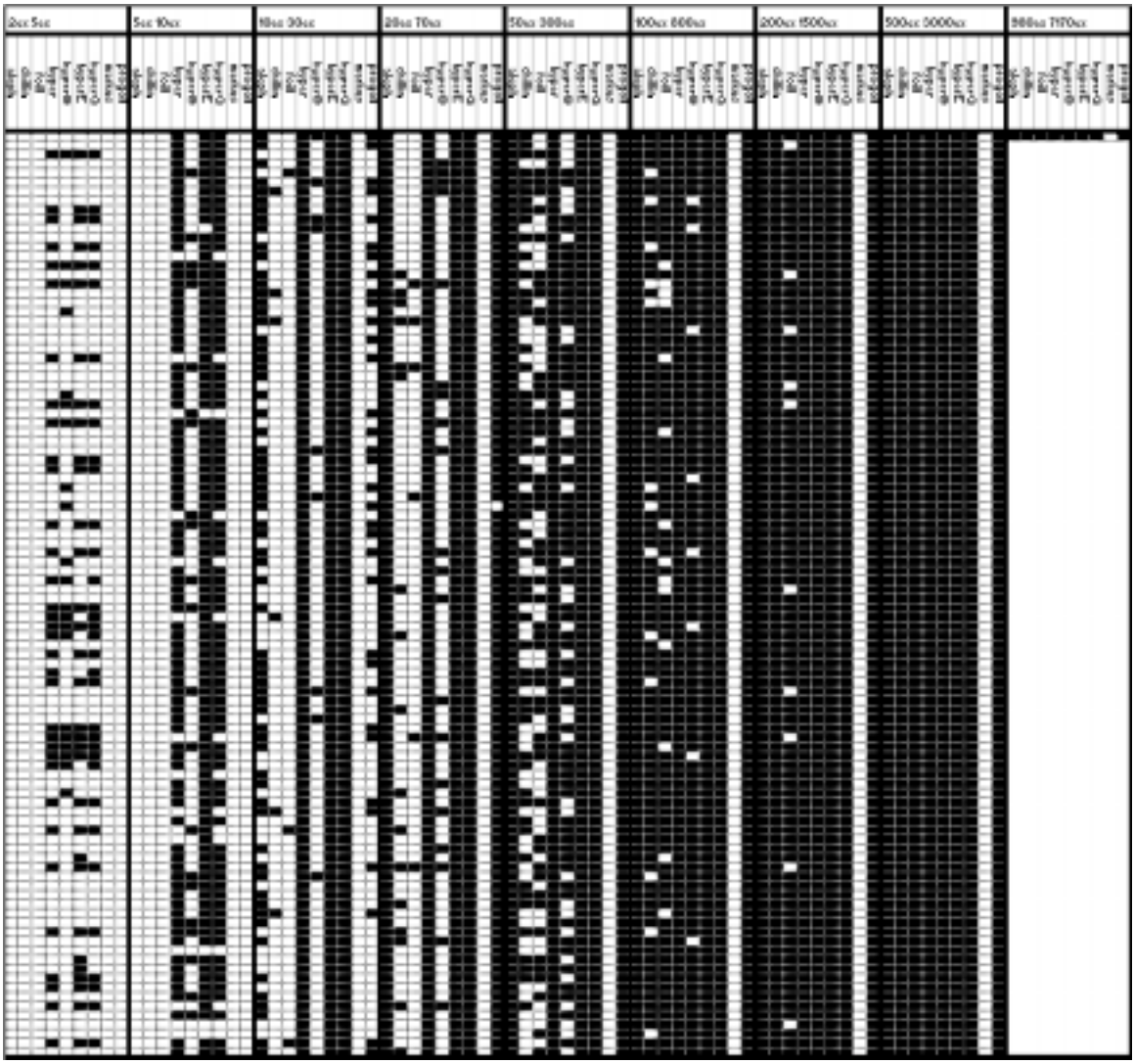
Sledijo (kot že rečeno) HYPER^2 z iskanjem s snopom, CHILLIN in FOIL. CHILLIN potrebuje dobro enakomerno pokritost vseh možnih dolžin učnih primerov, tako pozitivnih kot tudi negativnih, drugače se pri posploševanju in združevanju stavkov ne ustavi v pravem trenutku (če manjkajo pozitivni učni primeri določenega tipa, se običajno ustavi prezgodaj in je hipoteza preveč specifična; če pa manjkajo določeni negativni učni primeri, se ustavi prepozno in je hipoteza presplošna).

FOIL kot vedno poleg pozitivnih učnih primerov, ki pokrivajo ustavitveni pogoj rekurzije, potrebuje pozitivni učni primer, ki se lahko s prvim rekurzivnim klicem prevede na nek drug pozitivni učni primer (oziroma če se lahko prevede z dvema klicema, potrebuje tudi primere, s pomočjo katerih lahko sestavi zaustavitvene pogoje rekurzije večjih globin).

Povprečna poraba časov vseh sistemov (Sl. 5.57 - Sl. 5.62) najprej pokaže izstopanje vseh verzij sistema HYPER^2. Od teh izstopajo verzije z iskanjem najprej najboljši le pri majhnem številu učnih primerov, vendar se izkaže, da je nadpovprečna poraba časa samo v primerih, ko HYPER^2 ne najde primerne rešitve in se zato ustavi, ko predela 700 hipotez. Če bi zmanjšali dovoljeno število hipotez, ki jih HYPER^2 pregleda, bi se v tej domeni občutno zmanjšali časi potrebni za obdelavo, z majhnim vplivom na učinkovitost. Časi reševanja so v primerih, ko tudi drugi sistemi uspešno rešijo probleme, primerljivi s časi sistema PROGOL, medtem ko je CHILLIN nekaj hitrejši, ALEPH in FOIL pa sta občutno hitrejša. Prav na drugo stran odstopa verzija z iskanjem s snopom. Ta časovno odstopa v serijah, ki imajo veliko učnih primerov in ko pravilno reši problem. Ker je pri iskanju s snopom potrebno za vsak specializacijski korak pregledati več hipotez, se povečanje števila učnih primerov pozna bolj občutno kot pri drugih verzijah. Ostali sistemi se obnašajo kot običajno (le MARKUS-u ni bilo možno izmeriti porabljenega časa), vsi povečujejo porabo časa s povečanim številom primerov, (PROGOL in CHILLIN v večji meri kot ALEPH in FOIL), le FOIL pospeši reševanje, ko ima na voljo vse možne učne primere.

Povprečna velikost hipotez (Sl. 5.63 - Sl. 5.65), ki jih sistemi generirajo kot rešitve problemov na domeni next, se gibljejo okoli velikosti optimalne hipoteze. Večina rešitev vsebuje do šest predikatov. Še najbližje so vedno različne verzije sistema HYPER^2, ki uporabljajo iskanje najprej najboljši. Verzija z iskanjem s snopom pogosto vrne prazno hipotezo. Kadar pa vrne

pravilno rešitev, je le-ta vedno optimalne velikosti, kot je razvidno na Sl. 5.65. Posledično se povprečna velikost vrnjene hipoteze z večjo učinkovitostjo (z več učnimi primeri) povečuje do velikosti optimalne hipoteze. Le pri majhnem številu učnih primerov občasno generira nepravilno neprazno hipotezo. Zaradi tega in padca učinkovitosti v začetnih serijah, povprečna velikost hipotez najprej pade, preden se poveča do velikosti optimalne hipoteze.



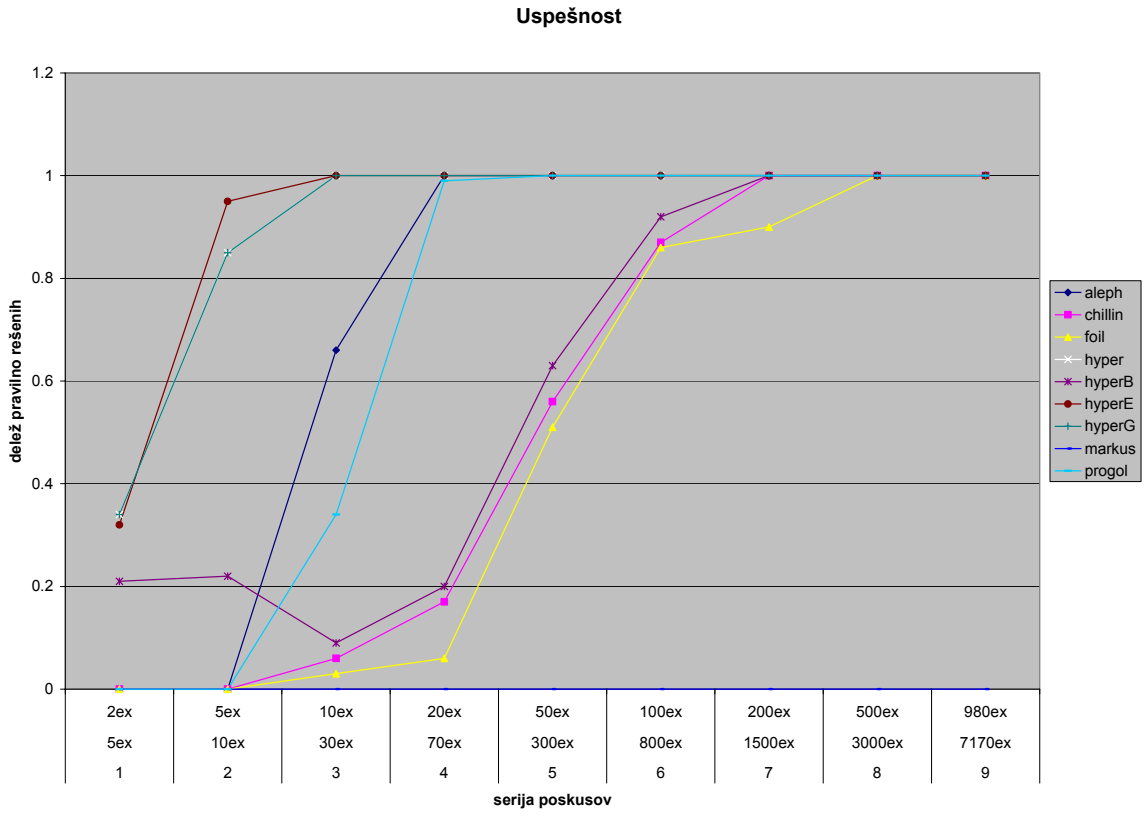
Sl. 5.55 Primerjava pravilno rešenih problemov domene next

PROGOL je edini od preostalih sistemov, ki večinoma vrne pravilne rešitve optimalne velikosti. Le občasno generira hipotezo, ki potrebuje dva zaustavitvena pogoja rekurzije in je zaradi tega nekaj večja od optimalne. Kadar mu ne uspe generirati pravilne hipoteze, nepokrite učne primere preprosto našteje, zato so nepravilne hipoteze dosti večje.

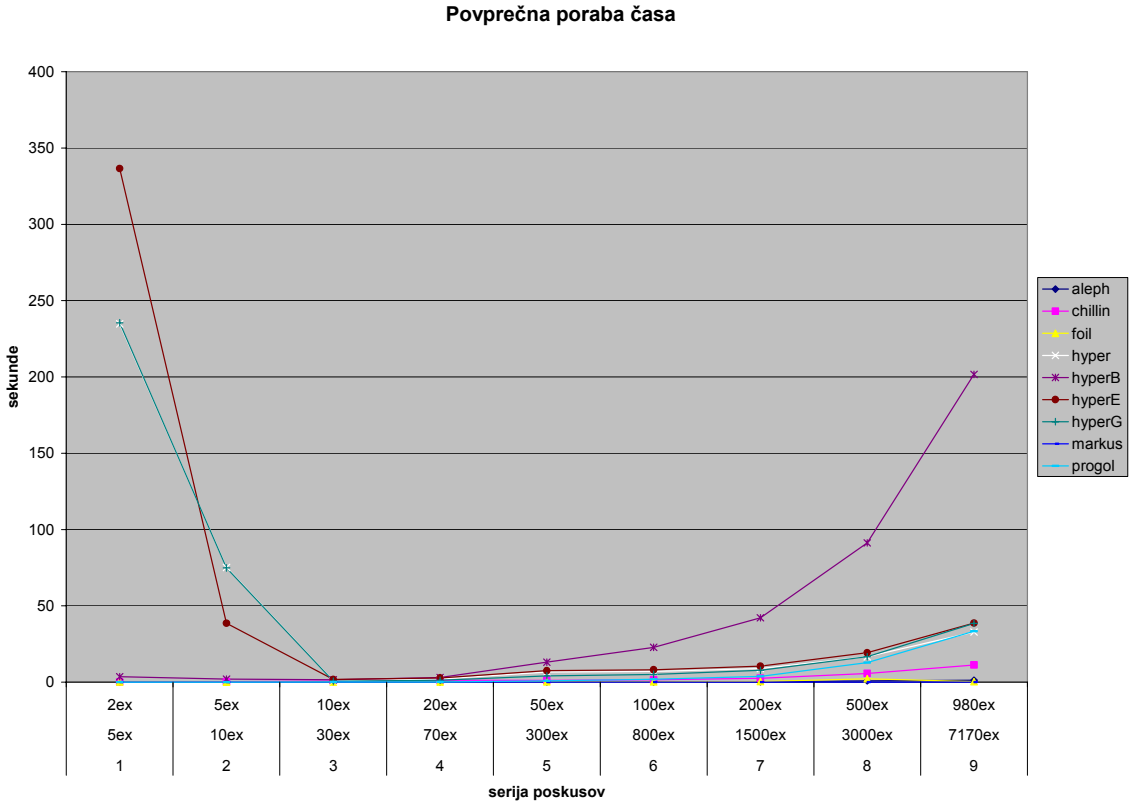
CHILLIN na drugi strani vrača hipoteze dokaj konstantnih velikosti (obstajajo redka odstopanja), vendar nikoli ne generira optimalne hipoteze (je edini izmed sistemov, ki mu uspe rešiti problem, kljub temu niti z vsemi učnimi primeri ne najde optimalne hipoteze).

ALEPH podobno kot PROGOL, ko mu ne uspe pokriti primerov, samo našteje nepokrite primere, zato so nepravilne hipoteze dokaj velike, vendar v nasprotju s sistemom PROGOL slabše odstranjuje redundance iz hipotez, zato so pogosto primeri našteti tudi v drugače pravilnih hipotezah, kar jim povečuje velikost. Ko se število razpoložljivih učnih primerov poveča, začenja generirati manjše hipoteze in konča z optimalno hipotezo.

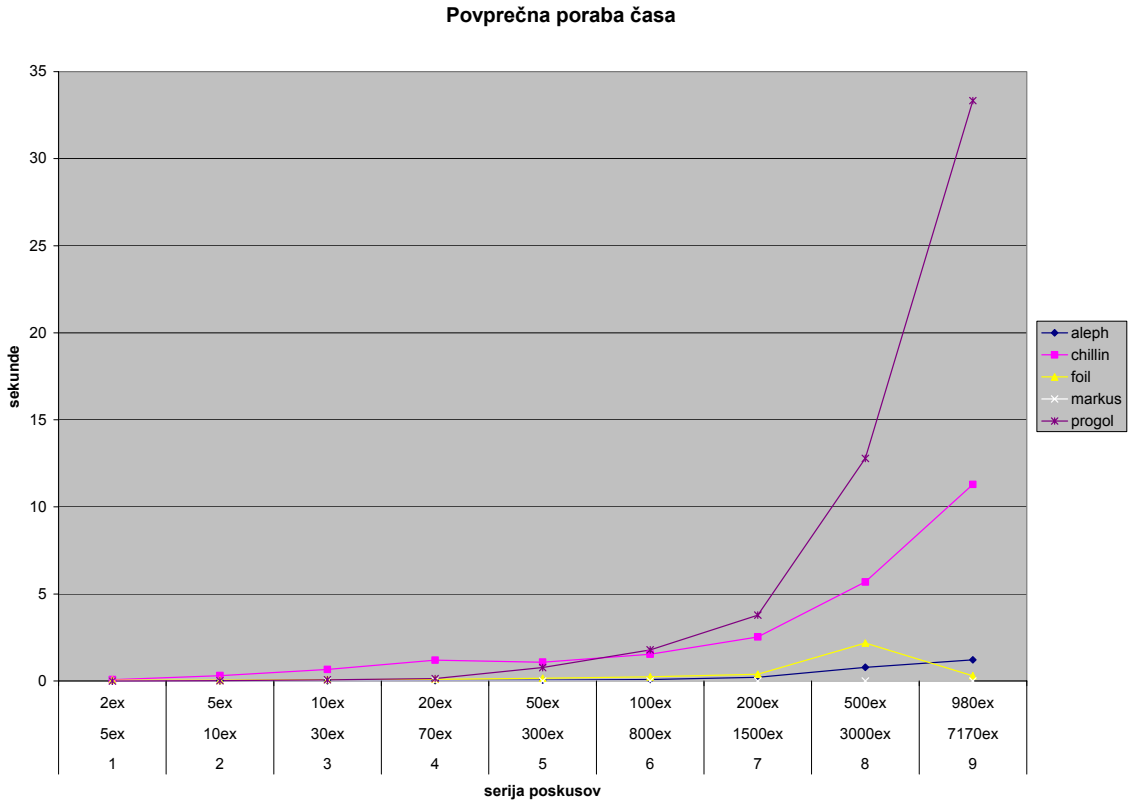
FOIL kot običajno generira zelo kompleksne hipoteze, katerih velikost je odvisna le od števila učnih primerov (velikost narašča skoraj linearno po serijah, kar bi pomenilo, da narašča približno logaritemsko s številom primerov). Le ko ima na voljo vse možne učne primere, mu uspe generirati optimalno hipotezo.



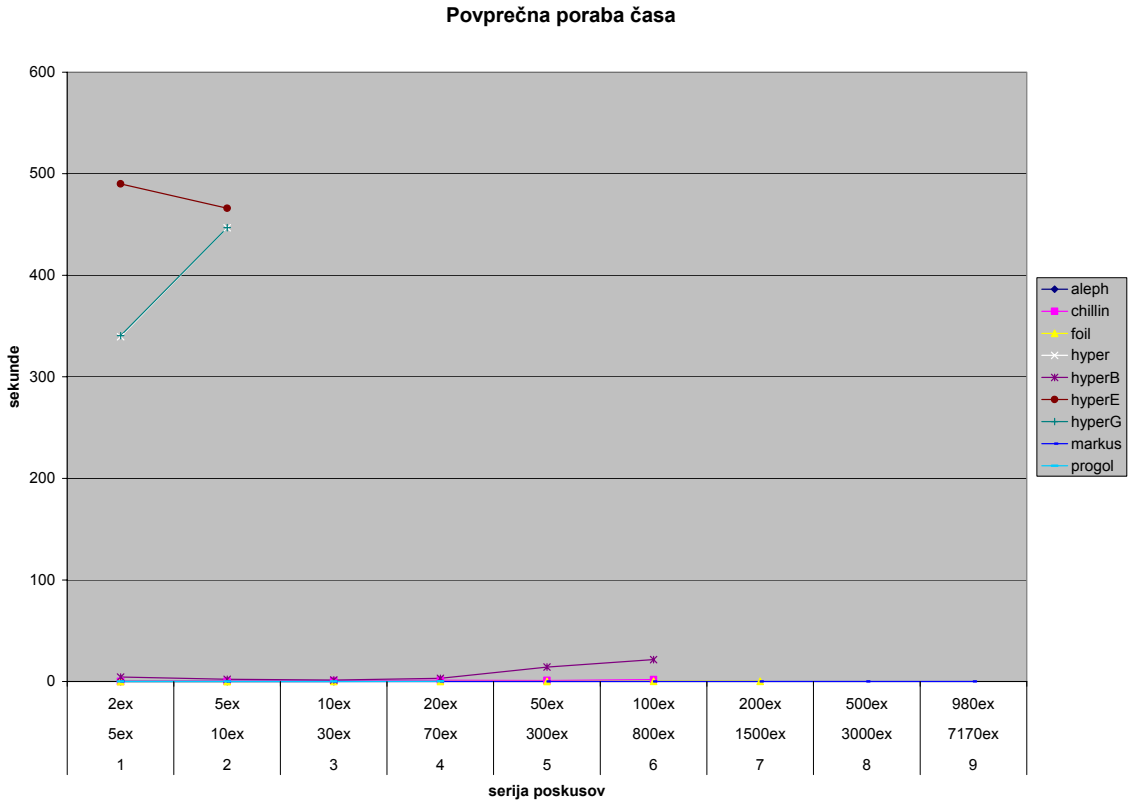
Sl. 5.56 Uspešnost različnih sistemov na domeni next



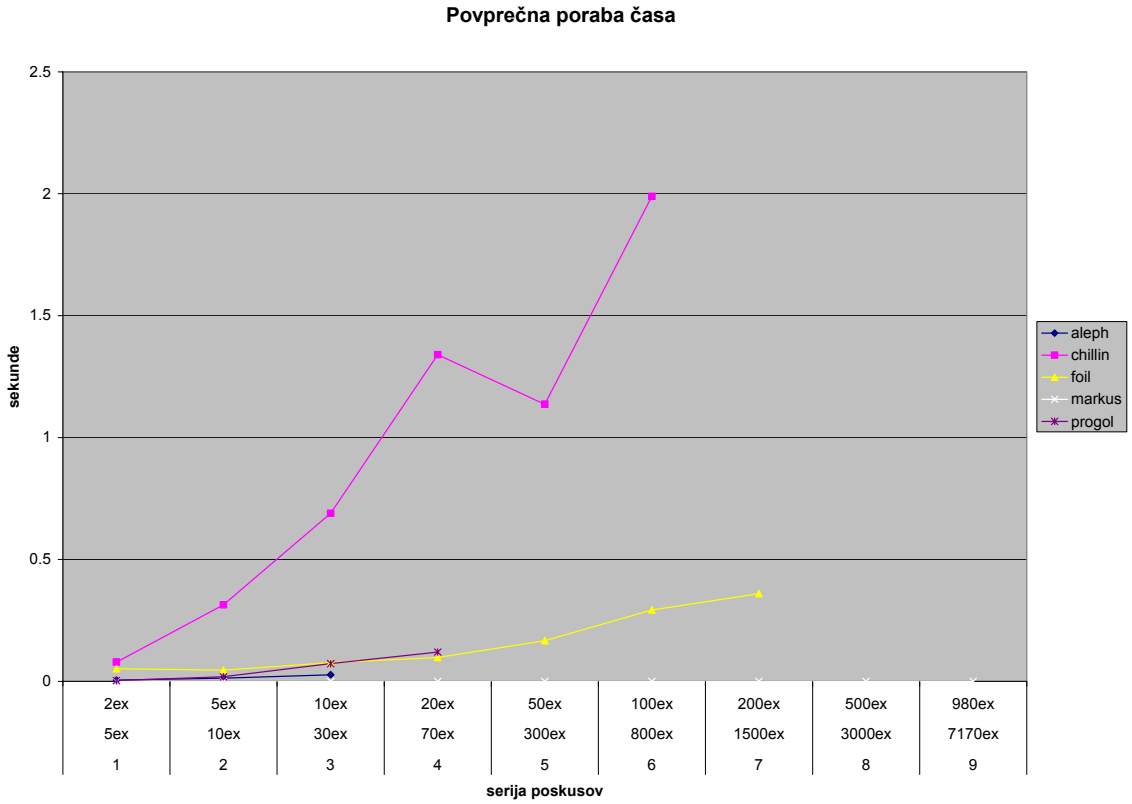
Sl. 5.57 Povprečna poraba časa različnih sistemov na domeni next



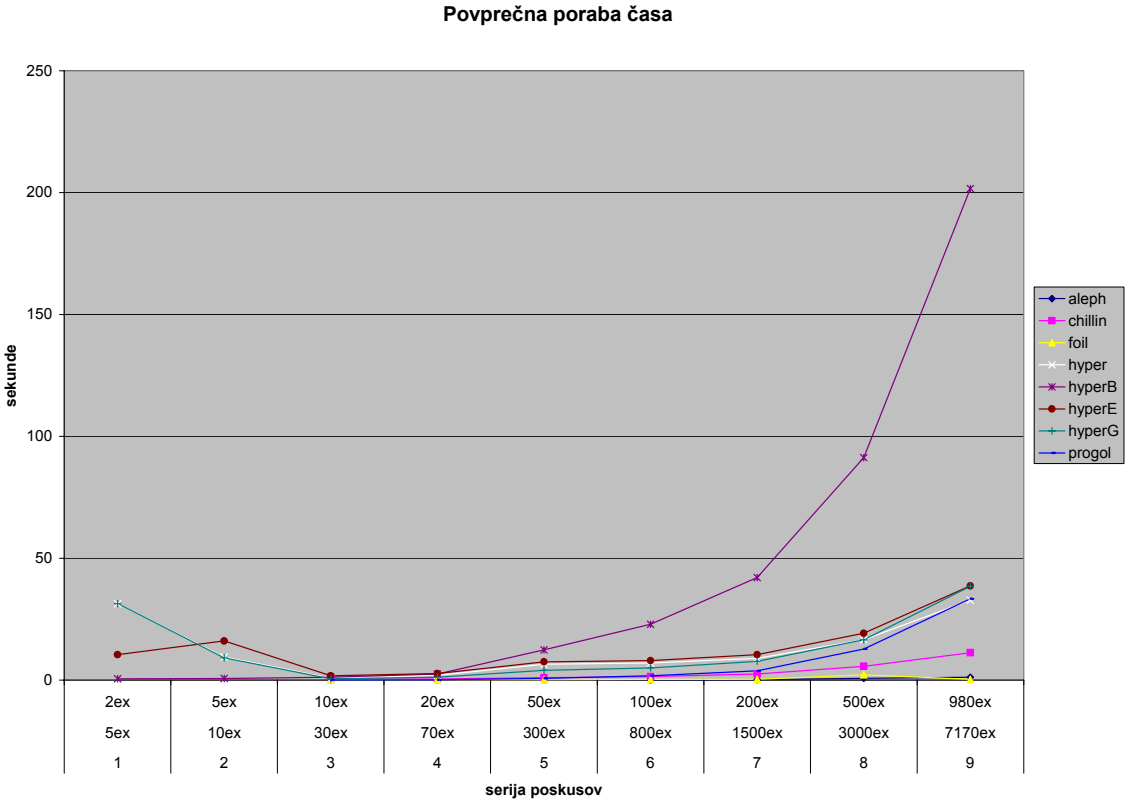
Sl. 5.58 Povprečna poraba časa sistemov (brez različnih verzij HYPER^2) v domeni next



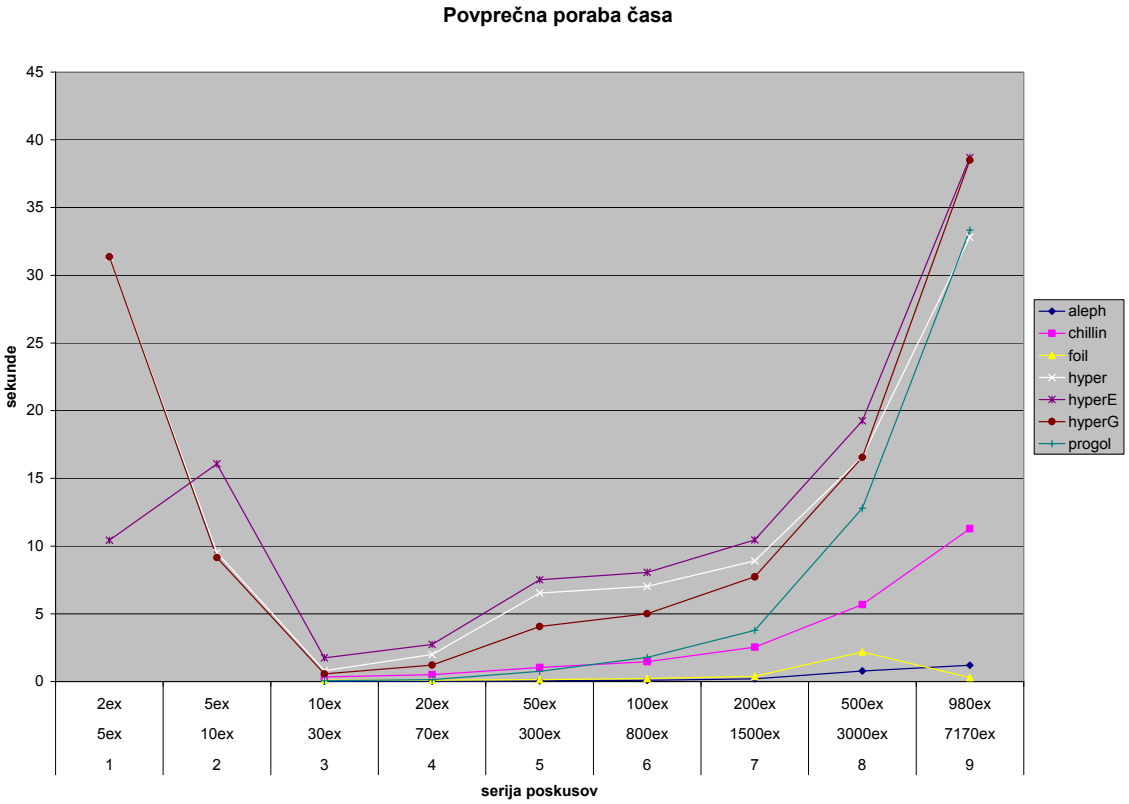
Sl. 5.59 Povprečna poraba časa sistemov v domeni next, ko niso pravilno rešili problema



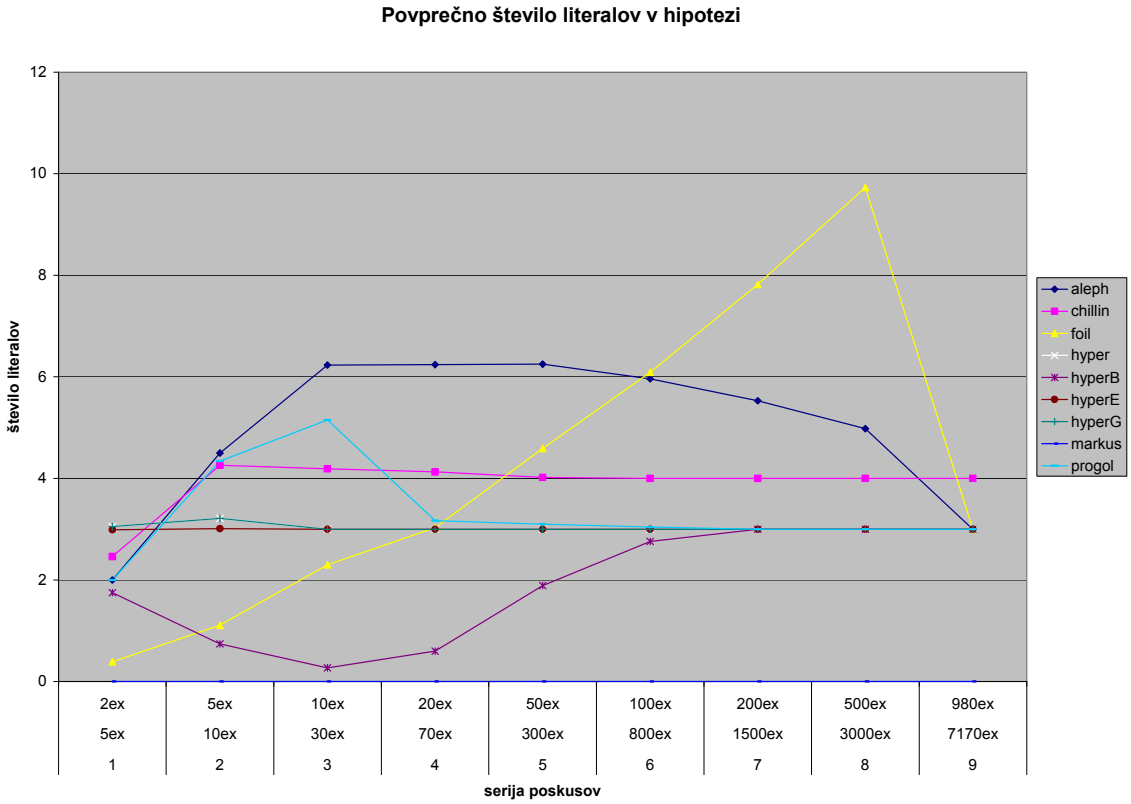
Sl. 5.60 Povprečna poraba časa sistemov (brez različnih verzij HYPER^2) v domeni next, ko niso pravilno rešili problema



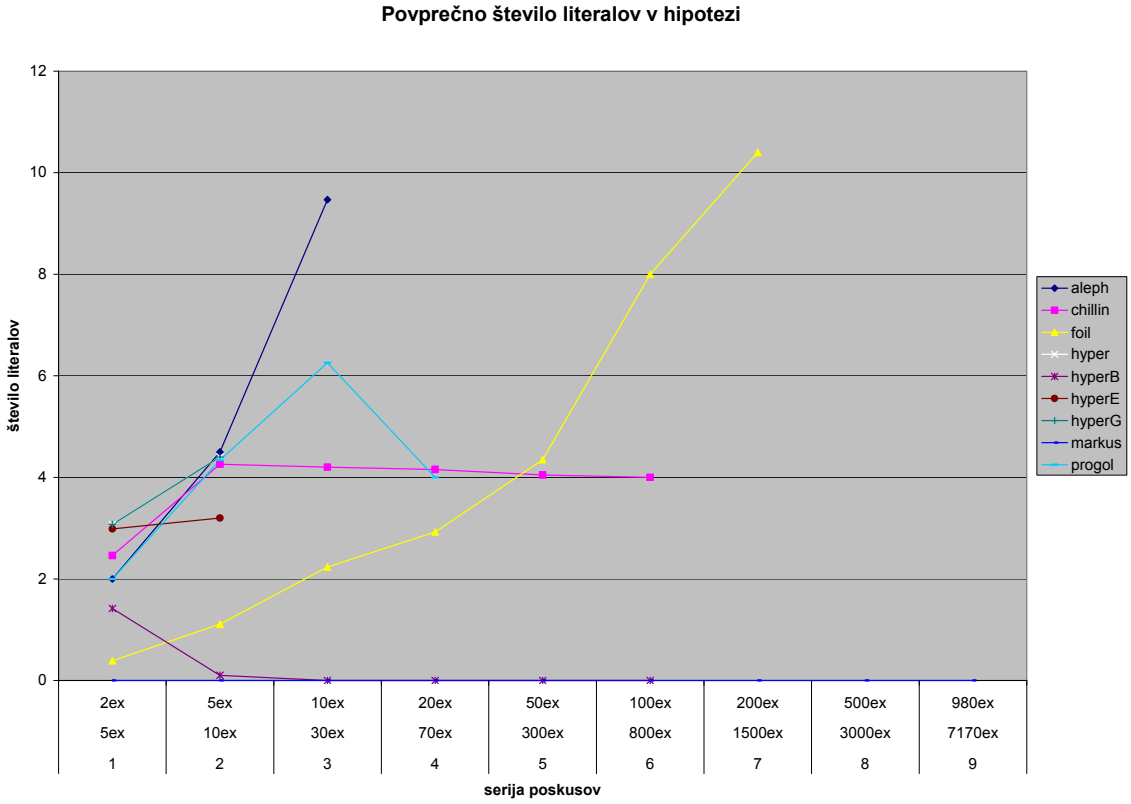
Sl. 5.61 Povprečna poraba časa sistemov v domeni next, ko so pravilno rešili problem



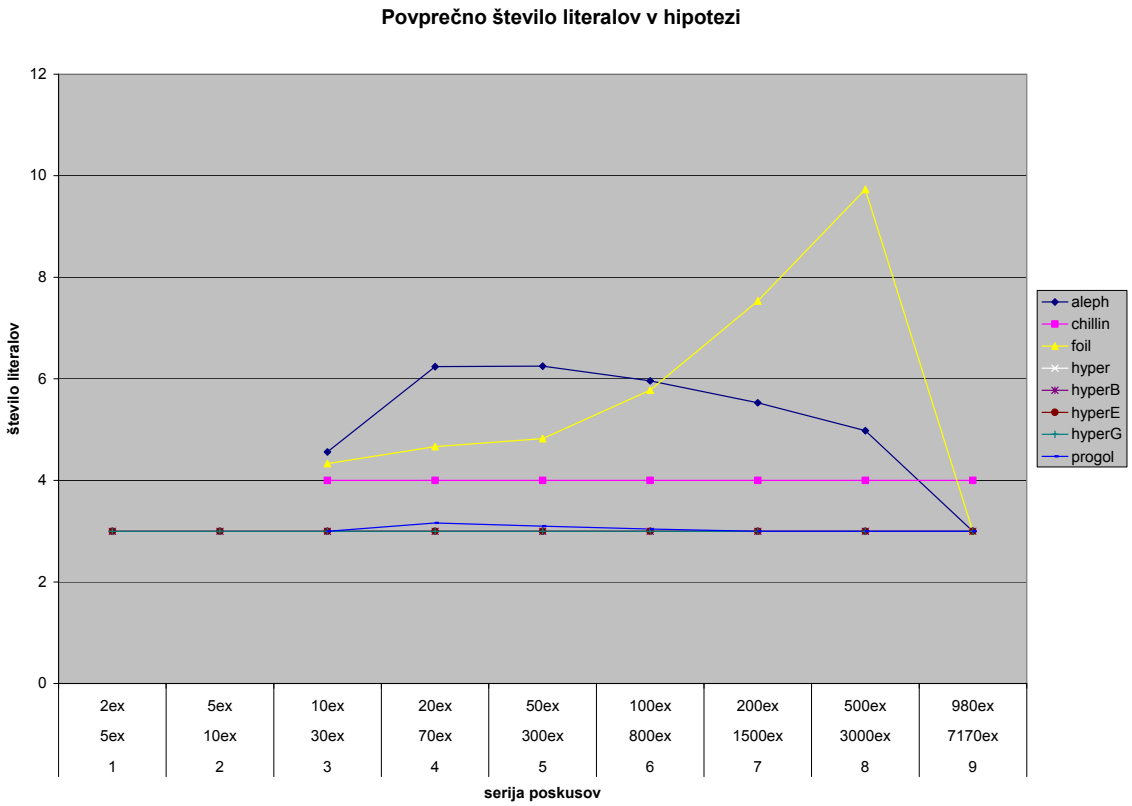
Sl. 5.62 Povprečna poraba časa sistemov (brez sistema hyperB) v domeni next, ko so pravilno rešili problem



Sl. 5.63 Povprečno število literalov v hipotezah sistemov v domeni next



Sl. 5.64 Povprečno število literalov v hipotezah sistemov v domeni next, ko niso pravilno rešili problema



Sl. 5.65 Povprečno število literalov v hipotezah sistemov v domeni next, ko so pravilno rešili problem

5.7 Domena path

5.7.1 Opis domene

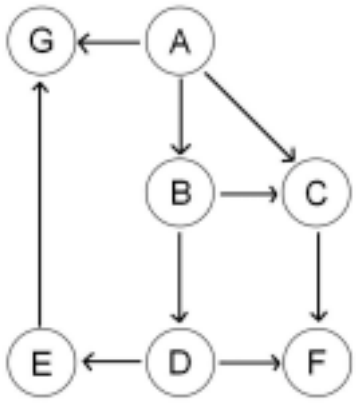
Celotna domena predstavlja iskanje poti (opisanih s seznami) od enega podanega vozlišča v grafu do drugega. Za generiranje učnih primerov je bil uporabljen predikat na Sl. 5.66, s tem, da je bila struktura grafa na Sl. 5.67 podana kot predznanje (tako za generiranje učnih primerov kot tudi sistemom za generiranje hipotez). Vseh pozitivnih primerov je 28, negativnih primerov pa je 177.352. V tej domeni sta bila začetno in končno vozlišče podana kot vhodna parametra, pot pa kot izhodni parameter.

```
path(A,A,[A]).
path(A,B,[A|C]):-
        link(A,D),
        path(D,B,C).
```

Sl. 5.66 Definicija predikata path, ki je bila uporabljena za generiranje učnih primerov

Poskusi so potekali z različnimi verzijami sistema HYPER^2 (ker je bila domena path ena izmed prvih testnih domen, smo lahko testirali samo prve verzije sistema HYPER^2):

- v tabelah in slikah razviden kot hyper, verzija, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, brez informacije o vhodno/izhodnih spremenljivkah v glavah stavkov, z iskanjem najprej najboljši,
- Hyper^{2}Beam (HyperB), ki uporablja pokrivanje na nivoju stavkov, informacijo o vhodno/izhodnih spremenljivkah in iskanje s snopom.



Sl. 5.67 Graf, uporabljen za generiranje učnih primerov in katerega struktura je bila podana sistemom kot predznanje

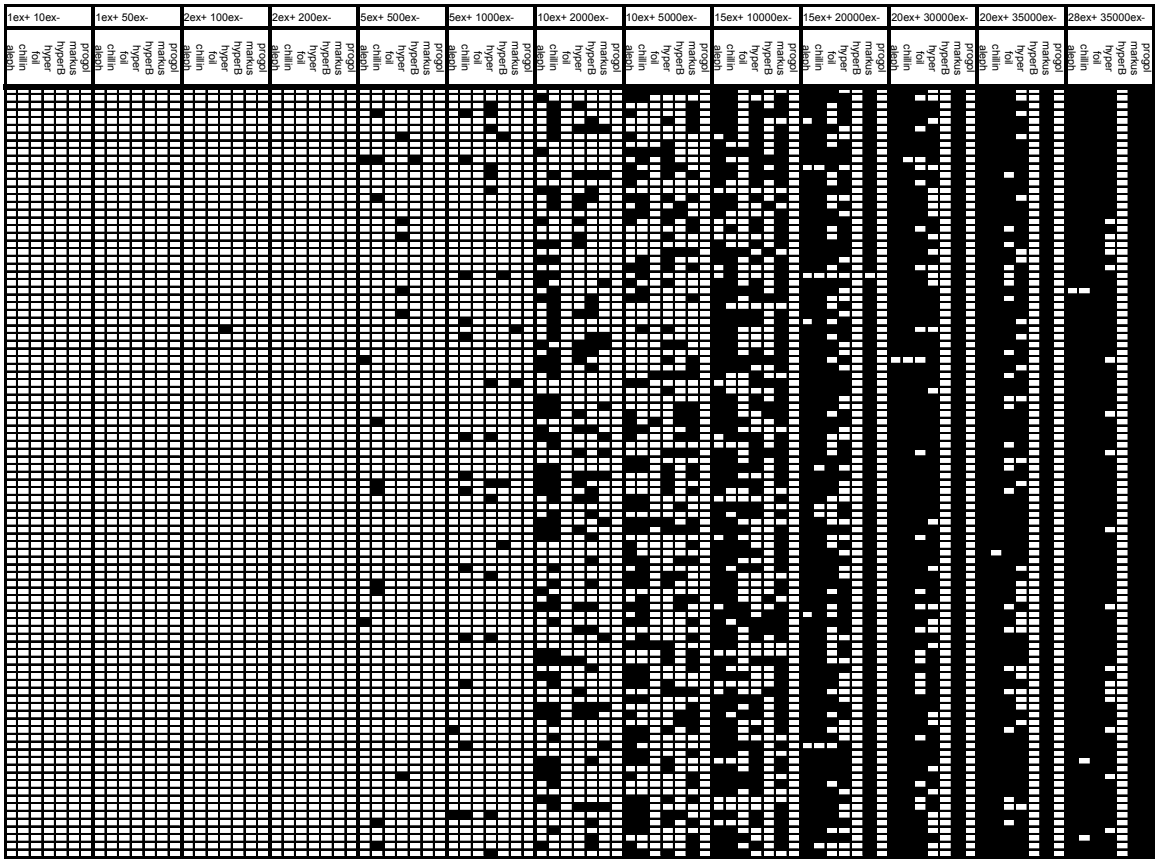
Oznaka serije	Število poskusov	Število pozitivnih primerov (delež od vseh poz. primerov)	Število negativnih primerov (delež od vseh neg. primerov)
1 (1ex 10ex)	100	1 (3,57%)	10 (0,006%)
2 (1ex 50ex)	100	1 (3,57%)	50 (0,028%)
3 (2ex 100ex)	100	2 (7,14%)	100 (0,056%)
4 (2ex 200ex)	100	2 (7,14%)	200 (0,11%)
5 (5ex 500ex)	100	5 (17,86%)	500 (0,28%)
6 (5ex 1000ex)	100	5 (17,86%)	1000 (0,56%)
7 (10ex 2000ex)	100	10 (35,71%)	2000 (1,13%)
8 (10ex 5000ex)	100	10 (35,71%)	5000 (2,82%)
9 (15ex 10000ex)	100	15 (53,57%)	10000 (5,64%)
10 (15ex 20000ex)	100	15 (53,57%)	20000 (11,28%)
11 (20ex 30000ex)	100	20 (71,43%)	30000 (16,92%)
12 (20ex 35000ex)	100	20 (71,43%)	35000 (19,73%)
13 (28ex 35000ex)	100	28 (100%)	35000 (19,73%)

tab. 5.6 Opis posameznih serij domene Path

Poleg tega smo uporabili tudi sisteme FOIL 6.4 (foil), CPROGOL 4.4 (progol), ALEPH v3 (aleph), SPChillin 1.0A prenesen na Sicstus Prolog (chillin) in Markus V1.1 (markus). Opravili smo 13 serij problemov z različnim številom primerov (tab. 5.6). V vsaki seriji je bilo opravljenih 100 poskusov z naključno izbranimi primeri (vsi sistemi so v isti ponovitvi dobili enake primere).

5.7.2 Rezultati

Ko pogledamo uspešnost sistemov (Sl. 5.68 in Sl. 5.69) najprej opazimo, da ko število učnih primerov preseže 20000 (predvsem negativnih) primerov, HYPER^2 z iskanjem s snopom ne reši več nobenega problema, kar je posledica povečane porabe časa (ter posledične zaustavitve izvajanja) in (bolj pogosto) prekomerne potrebe po pomnilniku, ki ji SICStus prolog ne more zadostiti (ne glede na količino razpoložljivega pomnilnika lahko SICStus razpolaga le z 256MB prostora). Že sama osnovna varianta sistema HYPER^2 samo zaradi dejstva, da razvija celotne hipoteze, porabi več prostora kot ostali sistemi, pri iskanju s snopom pa je ta poraba še toliko večja. Iz podobnih razlogov je osnovna verzija sistema HYPER^2 pri serijah z več učnimi podatki slabša od ostalih sistemov.

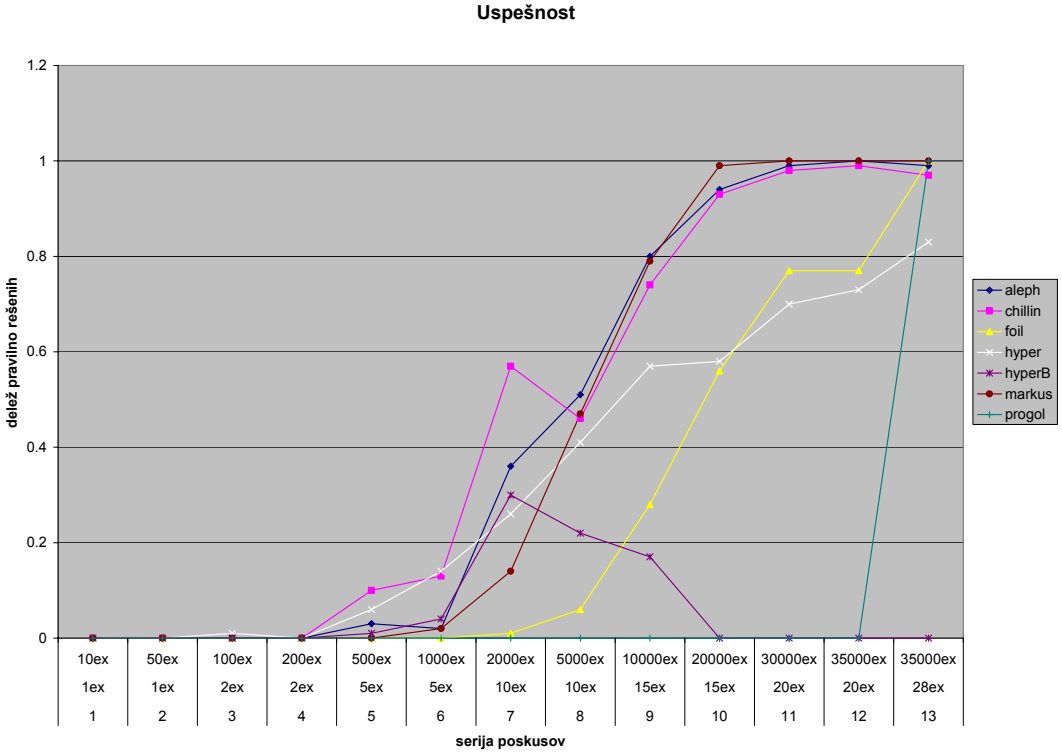


Sl. 5.68 Primerjava pravilno rešenih problemov

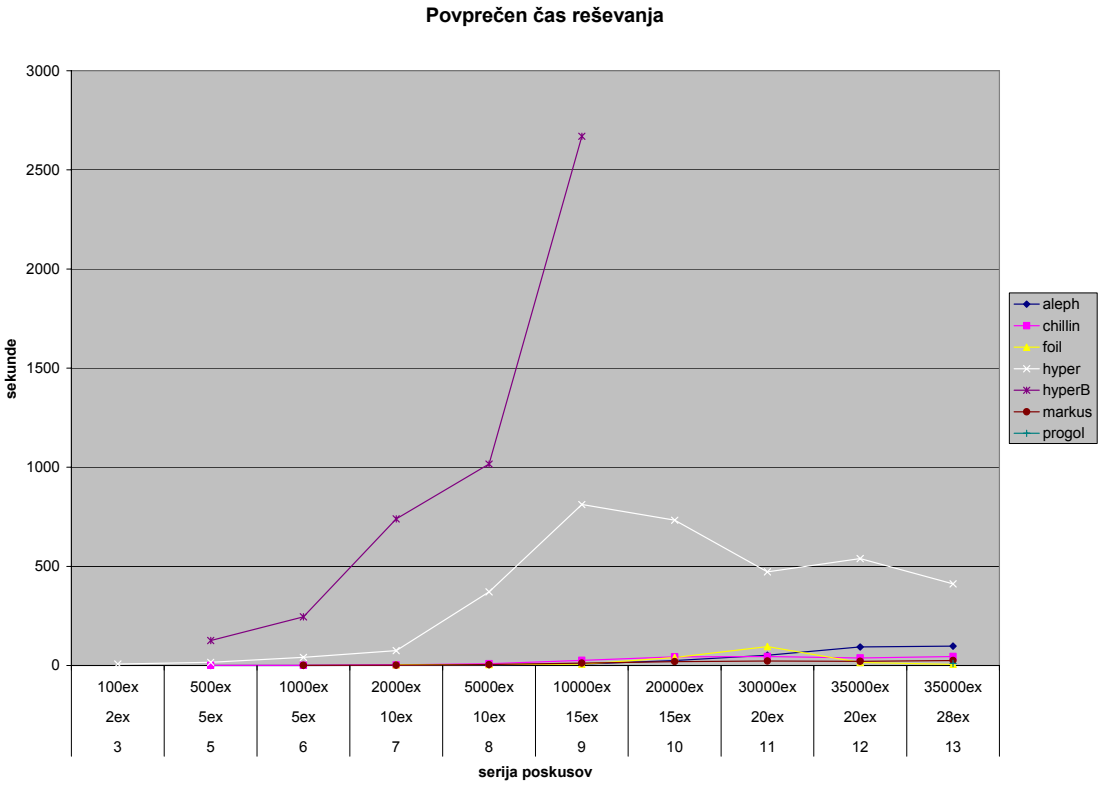
Prav tako se zelo hitro opazi, da PROGOL reši problem pravilno samo, če ima na voljo vse možne pozitivne učne primere. Pa še to velja bolj pogojno kot v resnici, saj PROGOL v tej domeni vedno samo našteje vse pozitivne učne primere kot enostavna dejstva, ne da bi našel kakršno koli zakonitost. Od preostalih sistemov se nekoliko slabše obnaša FOIL, zopet zaradi zahteve, da ima med pozitivnimi učnimi primeri primer, ki se lahko prevede na drug pozitivni učni primer. ALEPH, CHILLIN in MARKUS pa delujejo približno enako uspešno, s tem da je CHILLIN nekoliko bolj uspešen pri serijah z manj učnimi primeri.

Poglejmo si še čase potrebne za generiranje rešitev (v tej domeni imamo zanesljive meritve za vse sisteme le v primeru pravilnih rešitev, zato so predstavljeni samo ti rezultati). Kot že rečeno, je HYPER^2 zelo občutljiv na veliko število učnih podatkov, kar je razvidno na Sl. 5.70, s tem da je verzija z iskanjem s snopom posebej bolj občutljiva na število podatkov. Osnovna verzija je pri serijah z največ podatki hitrejša, ker se z več podatki običajno laže usmerja proti pravilni rešitvi; kadar se to ni zgodilo, običajno ni pravilno rešila problema, ker ji je zmanjkalo časa ali prostora (medtem ko je pri manjših serijah lahko še našla pravilno rešitev v daljšem času).

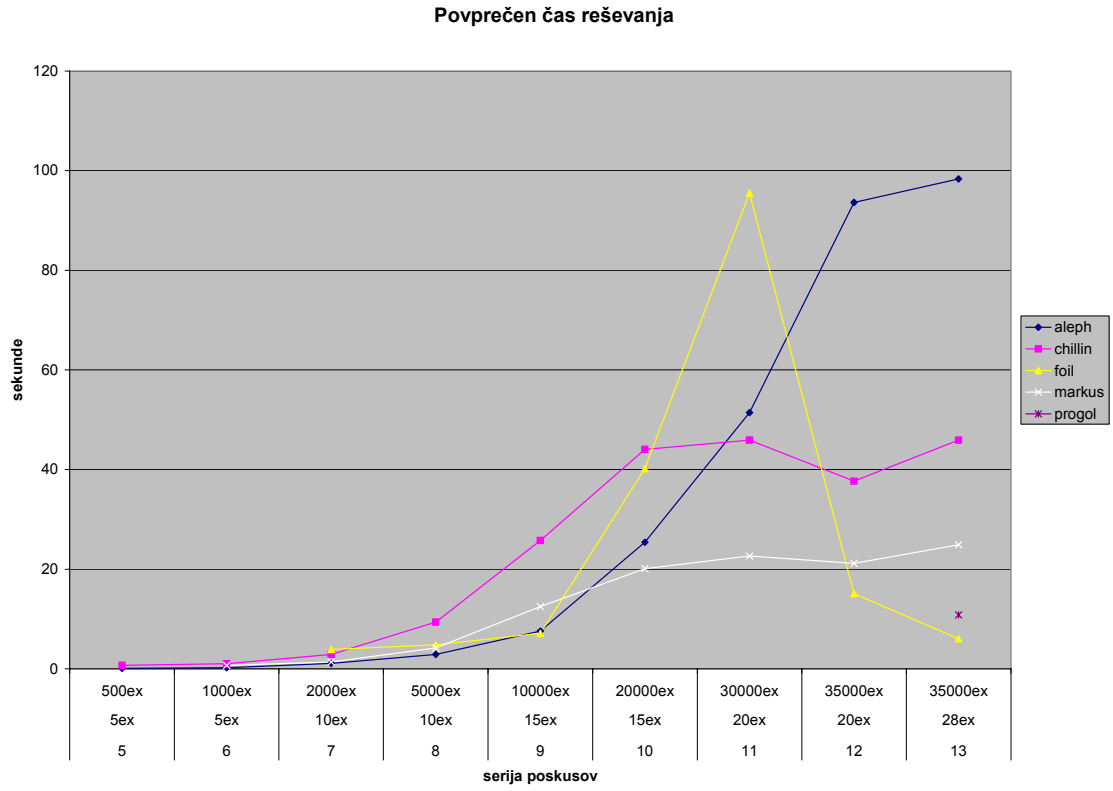
Preostali sistemi (Sl. 5.71) so precej hitrejši, s tem da pri njih največjo občutljivost na število učnih podatkov pokažeta ALEPH in do neke mere FOIL (ki tudi v tej domeni precej pospeši delovanje, ko ima na voljo večino pozitivnih učnih podatkov). Tako CHILLIN kot MARKUS ne pokažeta izrazitejše občutljivosti na večje število učnih podatkov in od trenutka, ko imata na voljo dovolj pozitivnih učnih podatkov, da pravilno rešita skoraj vse probleme, rešujeta le-te v skoraj konstantnem času (s tem da je treba omeniti, da MARKUS reši probleme približno enkrat hitreje kot CHILLIN).



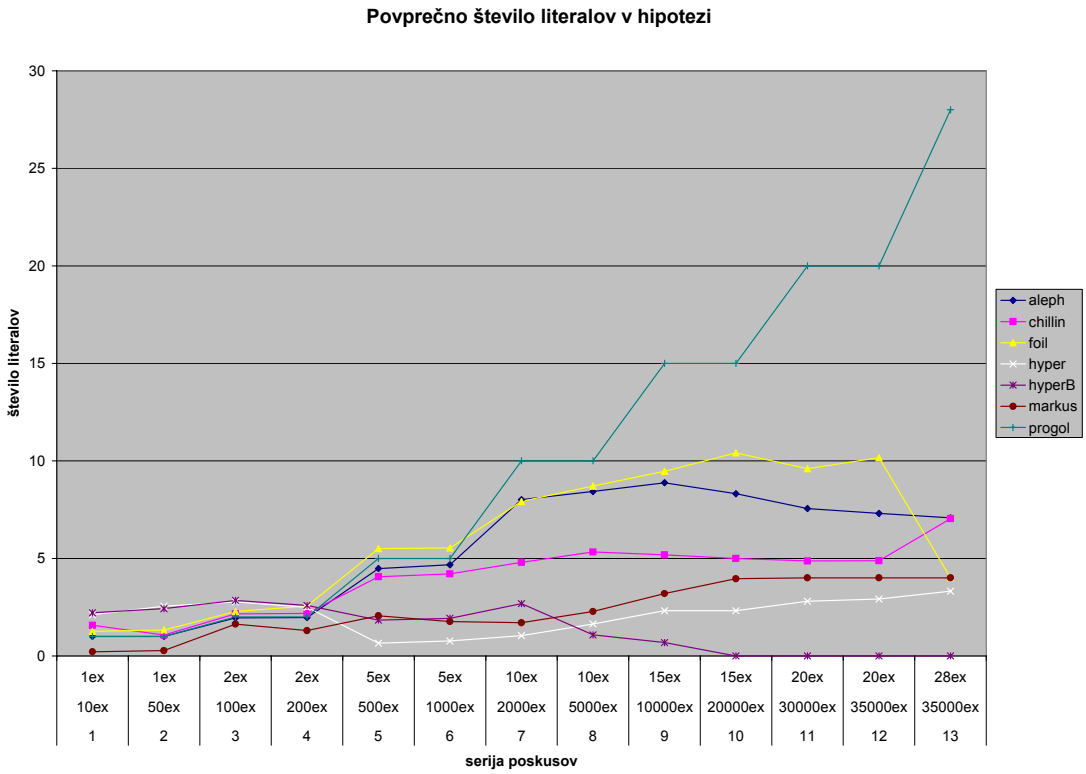
Sl. 5.69 Uspešnost sistemov na domeni Path



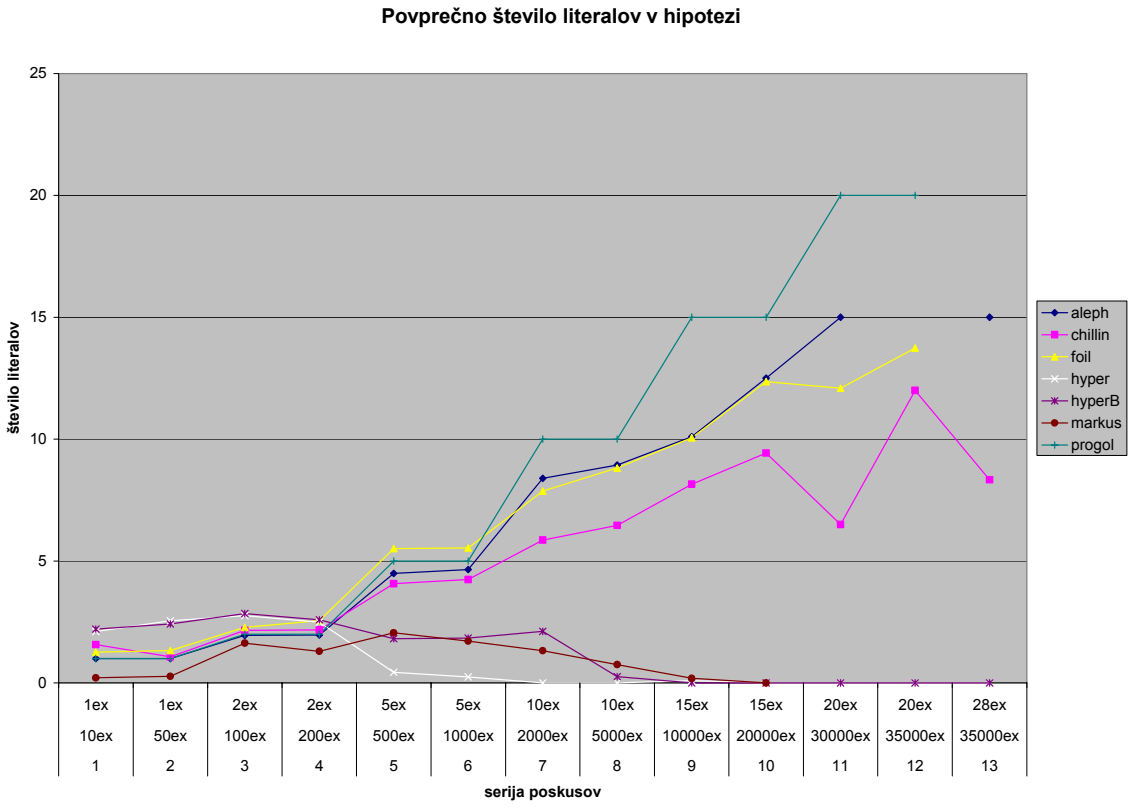
Sl. 5.70 Povprečna poraba časa sistemov v domeni Path, ko so pravilno rešili problem



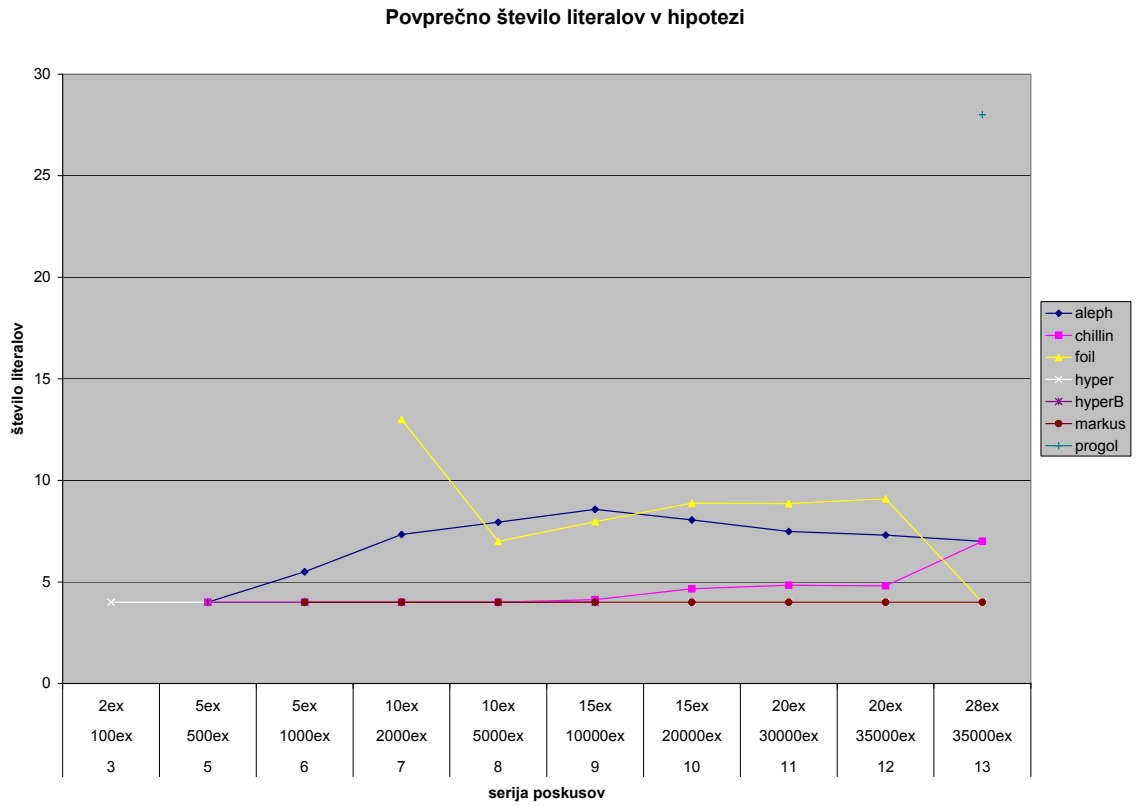
Sl. 5.71 Povprečna poraba časa sistemov (brez različnih verzij sistema HYPER[2]) v domeni Path, ko so pravilno rešili problem



Sl. 5.72 Povprečno število literalov v hipotezah sistemov v domeni Path



Sl. 5.73 Povprečno število literalov v hipotezah sistemov v domeni Path, ko niso pravilno rešili problema



Sl. 5.74 Povprečno število literalov v hipotezah sistemov v domeni Path, ko so pravilno rešili problem

Poglejmo še velikosti hipotez (Sl. 5.72 do Sl. 5.74). Najbolj očitna ugotovitev je, da PROGOL res samo našteje učne primere. FOIL, kot že v predhodnih domenah, vse dokler nima na voljo vseh pozitivnih učnih primerov, generira izredno kompleksne hipoteze (tudi pravilne), ALEPH pogosto našteva pozitivne učne primere (tudi če so že pokriti s strani drugih delov hipoteze), kar prinese povečanje končne hipoteze (tudi pri pravilnih rešitvah). Zanimivo je, da CHILLIN, ko ima na voljo večino pozitivnih učnih podatkov, generira nekoliko prevelike hipoteze (le-te vsebujejo stavek za poti dolžine ena, stavek za poti dolžine dve in stavek za daljše poti), medtem ko z manj podatki običajno najde optimalno hipotezo. Kompleksnost nepravilnih hipotez pa se, tako kot pri vsej prej naštetih sistemih, povečuje. MARKUS in obe verziji HYPER^2 pa se obnašajo drugače – pravilne hipoteze so vedno optimalne velikosti, medtem ko velikost nepravilnih hipotez z večjim številom učnih podatkov pada proti nič, kar pomeni, da od določene točke sistemi vračajo le pravilne rešitve ali pa ne vrnejo nič.

5.8 Domena insertionsort

5.8.1 Opis domene

Celotna domena je sestavljena iz seznamov z največjo dolžino pet, s petimi različnimi elementi, brez ponavljanja elementov v seznamih (rezultati bi lahko bili drugačni na domeni z

daljšimi seznami, vendar so razlogi zakaj nismo uporabili daljših seznamov, enaki kot v domeni append). Domena opisuje urejanje seznama. Pozitivnih primerov je 326, negativnih pa je 105.950. Učni primeri so bili generirani s predikatom na Sl. 5.75, s pomočjo predznanja na Sl. 5.76, ki je bil v enaki obliki podan vsem sistemom. Prvi del predznanja je varovalni, ker prepreči možno zazankanje ob napačni uporabi (CHILLIN ne uporablja informacije o vhodnih in izhodnih spremenljivkah predznanja, nekatere verzije sistema HYPER^2 pa lahko predznanje narobe uporabijo, ker napačno domnevajo, da je spremenljivka inicializirana, čeprav ni). V tej domeni je bil prvi, neurejen seznam podan kot vhodni parameter, urejen seznam pa kot izhodni parameter.

```
insertion_sort([], []).
insertion_sort([A|B], C):-
        insertion_sort(B, D),
        insert_sorted(A, D, C).
```

Sl. 5.75 Definicija predikata uporabljena za generiranje učnih primerov domene insertionsort

```
insert_sorted(X, L,_):-
        var(X), !, fail
        ;
        var(L), !, fail
        ;
        L=[Y|_], var(Y), !, fail.
insert_sorted(X, [], [X]).
insert_sorted(X, [Y|L], [X,Y|L]):-
        X@<Y,!.
insert_sorted(X, [Y|L], [Y|L1]):-
        insert_sorted(X, L, L1).
```

Sl. 5.76 Definicija predznanja uporabljenega za generiranje učnih primerov in podanega sistemom

Poskusi so potekali z različnimi verzijami sistema HYPER^2:

- v tabelah in slikah razviden kot hyperE, verzija, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, brez informacije o vhodno/izhodnih spremenljivkah v glavah stavkov, z iskanjem najprej najboljši,

- v tabelah in slikah razviden kot hyper, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyperG, ki si zapomni podatke o pokrivanju primerov na nivoju hipotez, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- Hyper^{2}Beam (HyperB), ki uporablja pokrivanje na nivoju stavkov, informacijo o vhodno/izhodnih spremenljivkah in iskanje s snopom.

Oznaka serije	Število poskusov	Število pozitivnih primerov (delež od vseh poz. primerov)	Število negativnih primerov (delež od vseh neg. primerov)
1 (3ex 30ex)	100	3 (0,92%)	30 (0,03%)
2 (5ex 100ex)	100	5 (1,53%)	100 (0,09%)
3 (10ex 200ex)	100	10 (3,07%)	200 (0,19%)
4 (20ex 500ex)	100	20 (6,13%)	500 (0,47%)
5 (50ex 1000ex)	100	50 (15,34%)	1000 (0,94%)
6 (100ex 2000ex)	100	100 (30,67%)	2000 (1,89%)
7 (200ex 4000ex)	100	200 (61,34%)	4000 (3,76%)

tab. 5.7 Opis posameznih serij domene InsertionSort

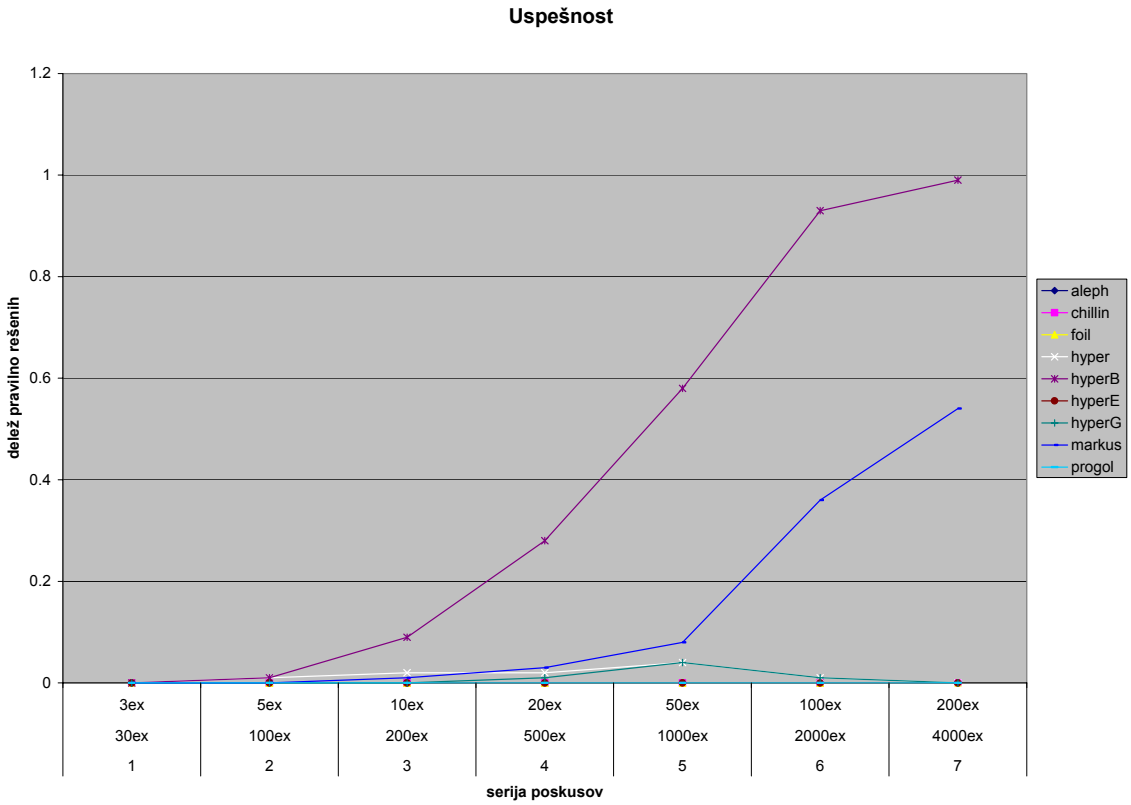
Poleg tega smo uporabili tudi sisteme FOIL 6.4 (foil), CPROGOL 4.4 (progol), ALEPH v3 (aleph), SPChillin 1.0A prenesen na Sicstus Prolog (chillin) in Markus V1.1 (markus). Opravili smo devet serij problemov z različnim številom primerov (tab. 5.7). V vsaki seriji (razen zadnje) je bilo opravljenih 100 poskusov z naključno izbranimi primeri (vsi sistemi so v isti ponovitvi dobili enake primere).

Treba je povedati, da medtem ko so bili v prejšnjih domenah sistemi omejeni na eno uro delovanja (na računalniku PIII 700MHz), so bili v tej domeni omejeni na 20 minut (na računalniku Athlon XP 1800+, narejenih je bilo nekaj poskusov na obeh računalnikih in ocenjeno, da je Athlon tri do štirikrat hitrejši).

5.8.2 Rezultati

Če pogledamo uspešnost (Sl. 5.78 in Sl. 5.77), opazimo, da samo MARKUS in HYPER^2 rešita to domeno. Izmed različnih verzij sistema HYPER^2 samo verzija z iskanjem s snopom učinkovito reši probleme, medtem ko jih osnovna verzija nikoli ne reši pravilno; verziji, ki znata uporabljati informacije o vhodno/izhodnem tipu spremenljivk v glavi stavkov pa

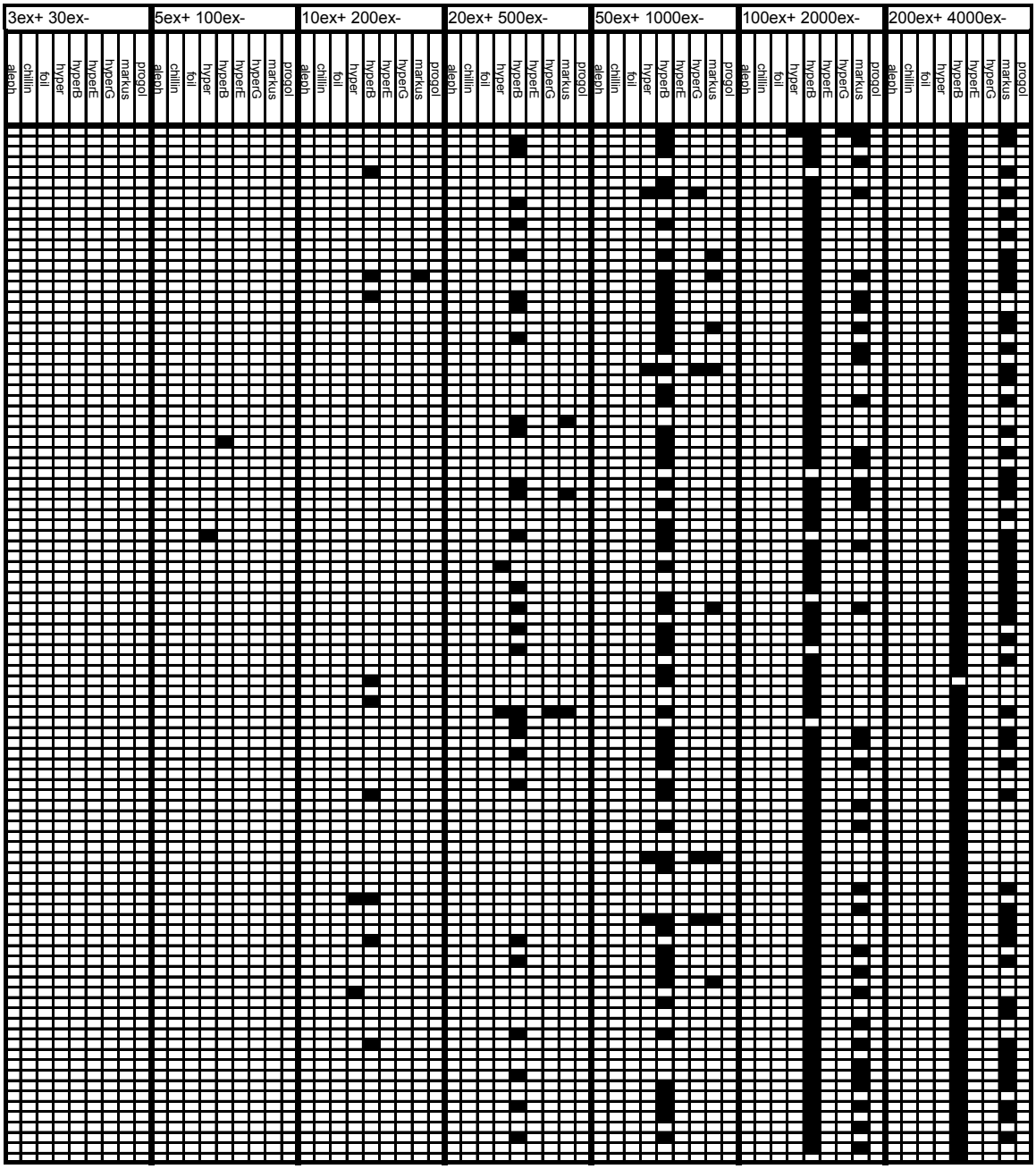
občasno rešita problem, običajno pa sta zaustavljeni zaradi prekoračitve dovoljenega časa. Očitno najprej zaideta v slepo ulico, in preden se lahko vrneta na pravo pot preiskovanja, jima zmanjka časa, medtem ko verzija z iskanjem s snopom ne zaide v slepo ulico – zaradi širine snopa obdeluje več možnih poti hkrati. Opazimo tudi, da je MARKUS pri reševanju slabši od HYPER^2 z iskanjem s snopom, vendar je to rezultat dejstva, da MARKUS pogosto najde rešitev, ki je pravilna za vse sezname dolžine večje od nič (ko pozitivnega učnega primera s praznim seznamom ni v učni množici). Poudariti je še treba, da pri nekaj poskusih, ki smo jih naredili z vsemi možnimi pozitivnimi primeri in pri približno 5% negativnih učnih primerov, poleg sistemov, ki so že prej rešili probleme (s tem, da hyper in hyperG rešita problem pred pretekom 20 minut) problem reši tudi FOIL. Vendar so ti poskusi potekali na drugačnem računalniku in časovni rezultati ne bi bili primerljivi.



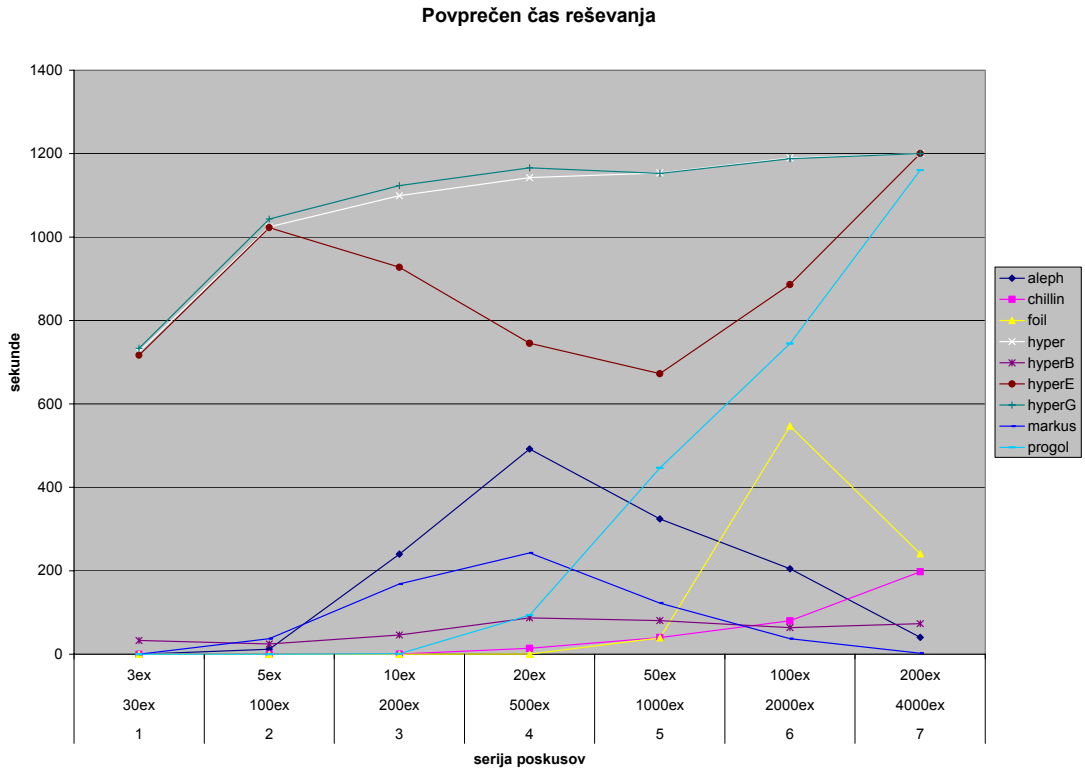
Sl. 5.77 Uspešnost različnih sistemov na domeni InsertionSort

Poraba časa (Sl. 5.79 do Sl. 5.81) pokaže, da sta bila hyper in hyperG (obe verziji sistema HYPER^2, ki iščeta najprej najboljši in znata uporabljati informacije o vhodno/izhodnem tipu spremenljivk) izredno pogosto zaustavljena zaradi prekoračitve časovne omejitve, saj je povprečen porabljen čas od tretje serije dalje skoraj 20 minut, medtem ko je povprečen čas pri nepravilno rešenih problemih od pete serije dalje točno 20 minut. Izkaže pa se, da je HYPER^2 z iskanjem s snopom v povprečju na serijah z manj učnimi primeri celo hitrejši od sistema

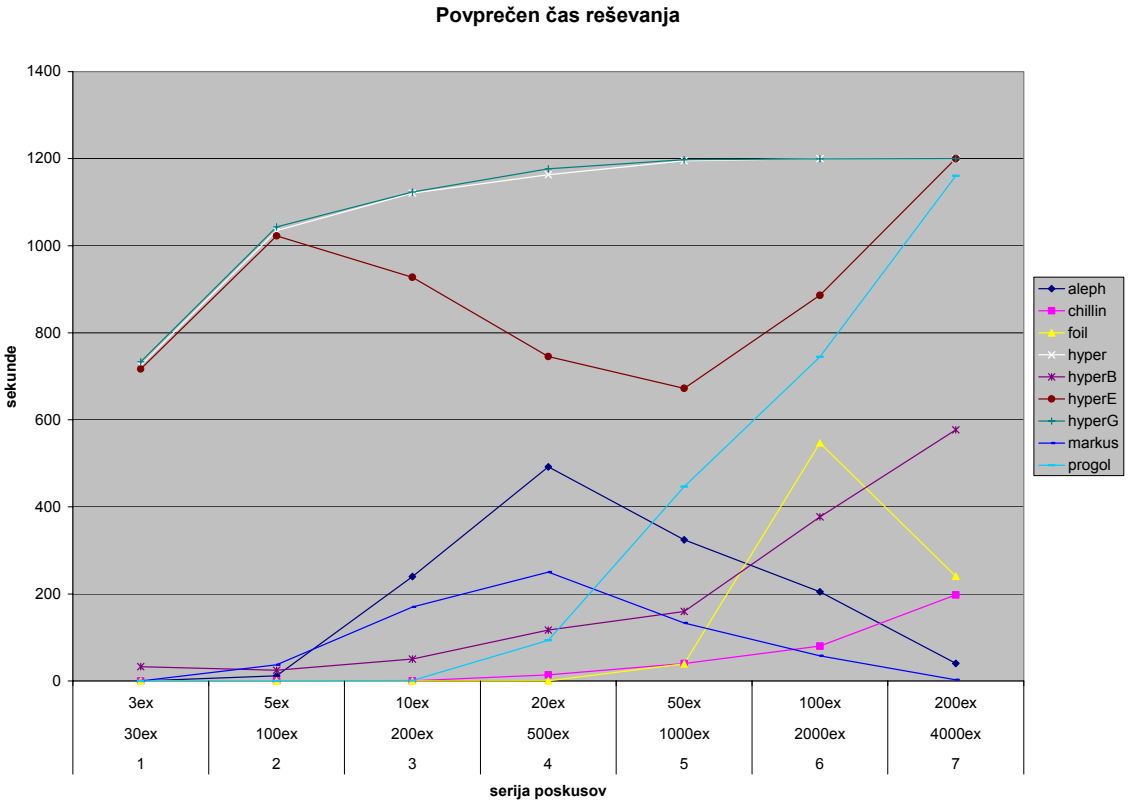
MARKUS, ta pa je hitrejši, če vzamemo samo pravilno rešene probleme - vendar je potrebno poudariti, da MARKUS pravilno reši le bolj enostavne probleme, ki imajo boljše učne primere, HYPER[2] (z iskanjem s snopom) pa tudi primere, ki imajo nekoliko slabše učne primere, kar lahko privede do večje porabe časa.



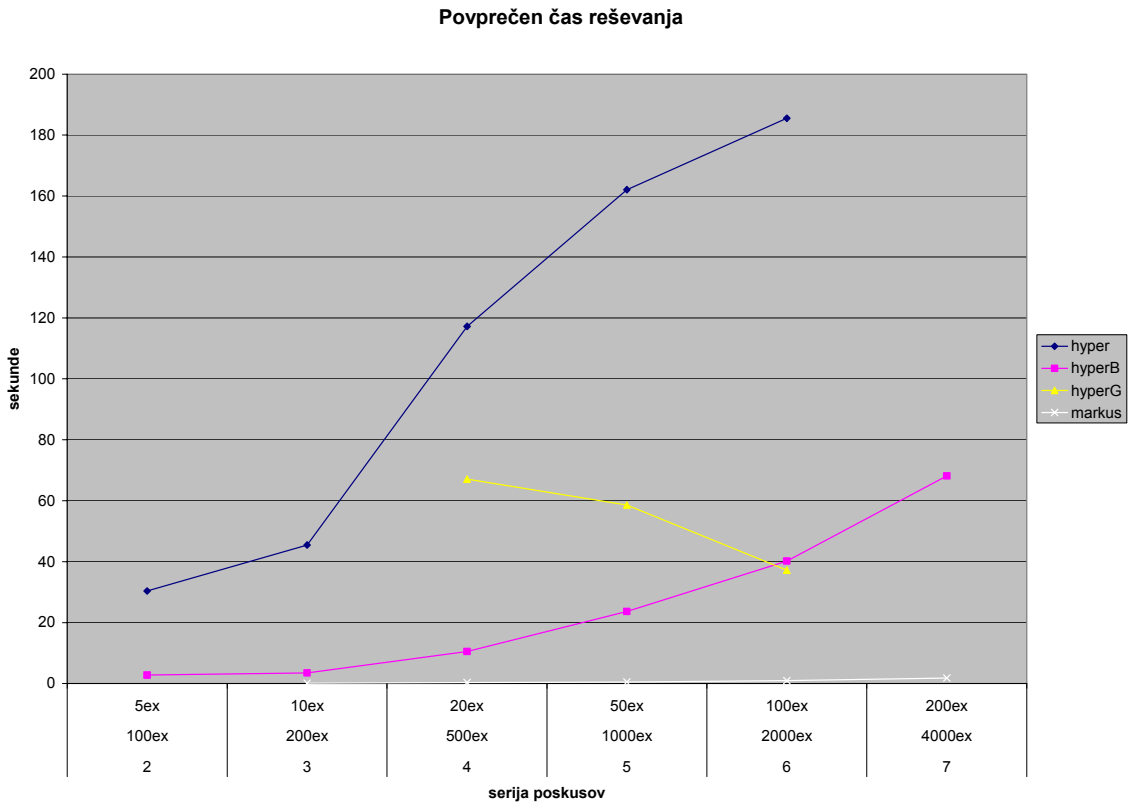
Sl. 5.78 Primerjava pravilno rešenih problemov domene insertionsort



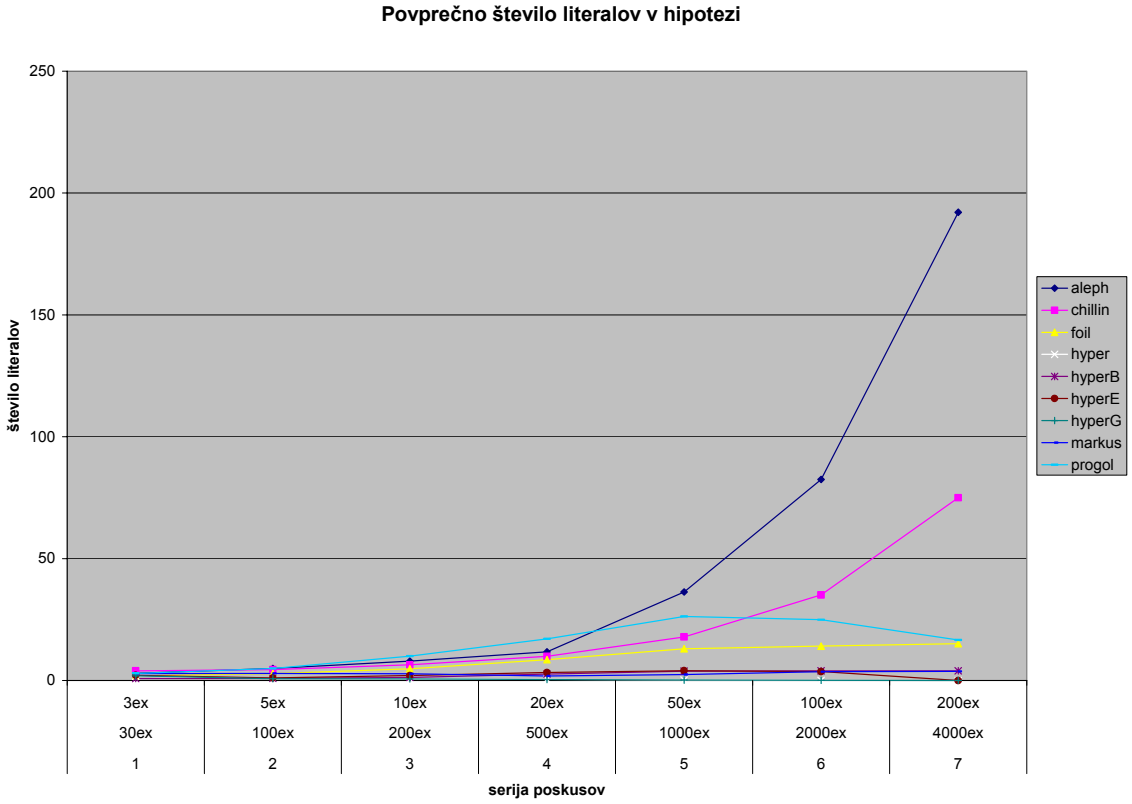
Sl. 5.79 Povprečna poraba časa različnih sistemov na domeni InsertionSort



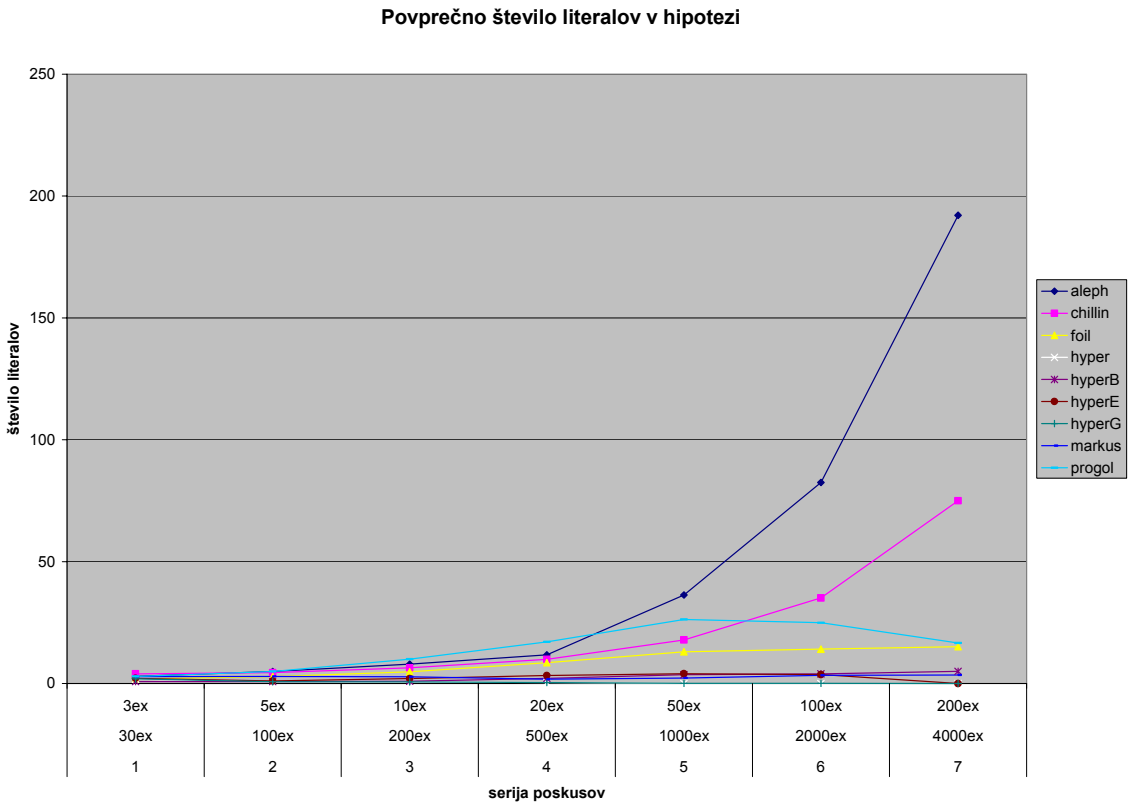
Sl. 5.80 Povprečna poraba časa sistemov na domeni InsertionSort, ko niso pravilno rešili problema



Sl. 5.81 Povprečna poraba časa sistemov v domeni InsertionSort, ko so pravilno rešili problem



Sl. 5.82 Povprečno število literalov v hipotezah sistemov v domeni InsertionSort



Sl. 5.83 Povprečno število literalov v hipotezah sistemov v domeni InsertionSort, ko niso pravilno rešili problema

Velikost hipotez (Sl. 5.82 in Sl. 5.83) pokaže, da ALEPH večinoma samo našteje pozitivne učne primere (nekajkrat je bil ustavljen po izteku časa in to zmanjša povprečni čas). CHILLIN prav tako poskuša z dokaj kompleksnimi hipotezami, ki pogosto vsebujejo tudi naštete nekatere pozitivne učne primere. Isto velja tudi za PROGOL, ki pa je pogosto ustavljen po poteku dovoljenega časa (ne da bi vrnil hipotezo), kar povzroči nekaj manjše povprečne hipoteze. Lastnost sistema FOIL, da ne našteje nepokritih učnih primerov, tukaj povzroči, da ne generira tako velikih hipotez, kot prej našteti sistemi, vendar tudi vrača prevelike hipoteze s približno 15 literali (vsaj v serijah z več učnimi primeri). MARKUS in HYPER^2 vračata (kadar sploh kaj vrneta) hipoteze z velikostjo približno okoli optimalne. Pravilne hipoteze pa so bile vedno optimalne velikosti (ta graf smo izpustili).

5.9 Domena quicksort

5.9.1 Opis domene

Celotna domena je sestavljena iz seznamov z največjo dolžino pet, s petimi različnimi elementi, brez ponavljanja elementov v seznamih (pred tem je bil tudi opravljen en poskus na manjši domeni s seznami z največjo dolžino štiri, s 65 pozitivnimi in 4.260 negativnimi

primeri). Rezultati bi lahko bili drugačni na domeni z daljšimi seznami, vendar so razlogi, zakaj nismo uporabili daljših seznamov, enaki kot v domeni append. Domena opisuje urejanje seznama. Pozitivnih primerov je 326, negativnih pa je 105.950. Učni primeri so bili enaki kot pri domeni insertionsort. V tej domeni je bil prvi, neurejen seznam podan kot vhodni parameter, urejeni seznam pa kot izhodni parameter. Algoritmom je bilo podano predznanje prikazano na Sl. 5.84. Predikat split razdeli seznam glede na podani element na dva seznama z elementi, manjšimi (v prvem vrnjenem seznamu) oziroma večjimi (v drugemu vrnjenemu seznamu) podanemu elementu. Predikat append pa združi seznam, element in seznam (v tem vrstnem redu) v nov seznam. V običajni definiciji predikata quicksort, kot bi ga sestavil človek (Sl. 5.85), bi uporabili predikat append, ki bi sestavil seznam iz dveh seznamov. Ker pa tako obliko algoritem težje sestavi (predvsem je problem v klicu predikata append, ki mora imeti en sestavljen vhodni argument, in tu bi bila v prednosti ALEPH in FOIL, ki morata uporabljati klice predznanja za delo s seznami in enostavno sestavita tak argument še pred klicem predikata append), je bila izbrana nekoliko drugačna definicija, ki simulira tak sestavljen argument, predstavljena na Sl. 5.86 (možne so tudi druge definicije). In tej definiciji je bilo prilagojeno tudi podano predznanje.

```
split(X,[],[],[]).
split(X,[Y|T],[Y|S],B):-
        X @>Y ,!,
        split(X,T,S,B).
split(X,[Y|T],S,[Y|B]):-
        split(X,T,S,B).

append([],B,C,[B|C]).
append([A|B],C,D,[A|E]):-
        append(B,C,D,E).
```

Sl. 5.84 Predznanje podano algoritmom pri domeni Quicksort

```
quicksort([],[]).
quicksort([A|B],C):-
        split(A,B,D,E),
        quicksort(D,F),
        quicksort(E,G),
        append(F,[A|G],C).
```

Sl. 5.85 Običajna definicija predikata quicksort

```
quicksort([],[]).
quicksort([A|B],C):-
        split(A,B,D,E),
        quicksort(D,F),
        quicksort(E,G),
        append(F,A,G,C).
```

Sl. 5.86 Pričakovana definicija predikata quicksort

Poskusi so potekali z različnimi verzijami sistema HYPER^2:

- v tabelah in slikah razviden kot hyperE, verzija, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, brez informacije o vhodno/izhodnih spremenljivkah v glavah stavkov, z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyper, ki si zapomni podatke o pokrivanju primerov na nivoju stavkov, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- v tabelah in slikah razviden kot hyperG, ki si zapomni podatke o pokrivanju primerov na nivoju hipotez, z informacijo o vhodno/izhodnih spremenljivkah v glavah stavkov in z iskanjem najprej najboljši,
- Hyper^{2}Beam (HyperB), ki uporablja pokrivanje na nivoju stavkov, informacijo o vhodno-izhodnih spremenljivkah in iskanje s snopom.

Poleg tega smo uporabili tudi sisteme FOIL 6.4 (foil), CPROGOL 4.4 (progol), ALEPH v3 (aleph), SPChillin 1.0A prenesen na Sicstus Prolog (chillin) in Markus V1.1 (markus). Opravili smo en poskus z vsemi učnimi primeri z manjšo domeno (primeri do dolžine štiri), ter deset poskusov z vsemi pozitivnimi in z 10% negativnih učnih primerov (z dolžino do pet). Vsi sistemi so v isti ponovitvi dobili enake primere.

Treba je povedati, da, medtem ko so bili v prejšnjih domenah sistemi omejeni na eno uro delovanja (na PIII 700MHz) oziroma 20 minut na Athlon XP 1800+, so bili v tej domeni omejeni na tri ure na Intel Pentium IV 1800.

5.9.2 Rezultati

Poskus, ki je potekal na krajši domeni, je uspešno rešil samo MARKUS in še ta ni našel definicije za quicksort, ampak nekakšno definicijo za insertionsort, ki je tudi možna s takimi predikati predznanja (seznam razdeli na element in ostanek, uredi ostanek, nato razcepi urejen seznam glede na element, ki ga je prej odstranil ter razcepljen seznam in element združi v urejen seznam). Ostali sistemi niso našli pravilne definicije.

Na daljši domeni je MARKUS prav tako našel definicijo za insertionsort, v sedmih od desetih primerov, na istih primerih je FOIL v petih od desetih primerov našel pravilno definicijo za quicksort. Čeprav je bil vrstni red klicev drugačen kot je bil pričakovan (rekurzivna klica sta na koncu, za združitvijo seznamov), predikat deluje pravilno. Ostali sistemi niso našli pravilne rešitve. Samo PROGOL je občasno še vrnil rešitev, ampak je običajno vsebovala večinoma naštete učne primere in kakšno zakonitost (recimo za sezname dolžine ena). FOIL je pri napačnih definicijah narobe definiral ustavitveni pogoj rekurzije, MARKUS pa je bil dvakrat ustavljen zaradi prekoračitve časovne omejitve, medtem ko se je enkrat sam ustavil brez najdene definicije.

Časi, ki so jih sistemi potrebovali za delo, so se gibali med desetimi sekundami in tremi urami. ALEPH in CHILLIN sta bila vedno ustavljena po treh urah. PROGOL se je samo dvakrat ustavil pred iztekom treh ur (obakrat z napačnim rezultatom in porabo 2.500 sekund oziroma 9.600 sekund). Različne verzije sistema HYPER so porabile med 24 in 2.024 sekundami (običajni čas je bil od 50 do 130 sekund, odvisno od verzije, povprečni časi so bili 269 sekund za hyper, 381 sekund za hyperB, 537 sekund za hyperE in 242 sekund za hyperG). FOIL je vrnil rezultat po 10-ih do 560-ih sekundah, s tem da je pravilni rezultat običajno našel po 430 sekundah (skupni povprečni čas je 277 sekund, povprečni čas pravilnih rešitev pa 389 sekund). MARKUS je bil kot že rečeno dvakrat ustavljen po treh urah, enkrat se je ustavil brez rešitve po 6.100 sekundah, medtem ko je rešitve našel praviloma v 18-ih do 40-tih sekundah (povprečni čas 2.787 sekund, povprečni čas pravilnih rešitev pa 22 sekund).

Najdene rešitve so bile optimalne velikosti (MARKUSOVA za insertionsort, ki je nekoliko krajša od definicije za quicksort, ki jo je našel FOIL). Samo napačni rešitvi, ki jih je našel PROGOL, sta bili večji in sta vsebovali več kot 200 literalov.

5.10 Zaključek

V tem poglavju smo nekaj osnovnih verzij sistema HYPER^2 primerjali z nekaj najbolj znanimi ILP sistemi (ALEPH, CHILLIN, CPROGOL, FOIL in MARKUS) na domenah programske sinteze. Ti sistemi so bili izbrani tako, da pokrijejo različne pristope k ILP.

V primerjavi se je pokazalo, da je HYPER^2 manj kot ostali sistemi občutljiv na manjkajoče učne podatke, od ostalih se mu še najbolj približa MARKUS, medtem ko je FOIL najbolj občutljiv. HYPER^2 konsistentno najde majhne hipoteze kot tudi MARKUS in CHILLIN, medtem ko ALEPH, CPROGOL in FOIL pogosto generirajo prevelike in prekompleksne hipoteze. Pravzaprav je HYPER^2 celo sposoben generirati stavke, za katere ni nobenega pozitivnega učnega primera, vendar le v primeru, če so drugi deli iskane hipoteze odvisni od teh stavkov (kar se naredi pri rekurzivnih in medsebojno rekurzivnih hipotezah).

Vendar se je tudi izkazalo, da predvsem pri domenah z več učnimi podatki pride do izraza večja kompleksnost preiskovanja celotnih hipotez, in zato HYPER^2 v takih domenah lahko deluje občutno počasneje od ostalih sistemov, med katerimi je običajno najhitrejši FOIL.

V domenah, kjer so iskani predikati nekaj bolj kompleksni (obe domeni z urejanjem seznamov), se kot konstantno dobrega izkaže predvsem MARKUS, ki najde rešitve v obeh domenah (sicer ne vedno, in v domeni quicksort najde definicijo, ki ustreza urejanju z vstavljanjem), vendar ne najde pravilne rešitve v dosti preprostejši domeni next. Domeno insertionsort poleg sistema MARKUS reši še HYPER^2 z iskanjem s snopom (ter občasno tudi druge verzije sistema HYPER^2), ki pa ne rešijo problema v domeni quicksort. Obe domeni reši tudi FOIL, vendar, kakor kaže, za to potrebuje vse pozitivne učne primere.

Torej, če je na voljo malo učnih primerov ter obstaja sum, da nekateri ključni učni primeri manjkajo, je najbolje uporabiti sistem HYPER^2; če je na voljo več učnih primerov, a kljub temu še ne vsi, je priporočljivo uporabiti MARKUS in poskusiti tudi z HYPER^2; če so pa na voljo vsi učni primeri (ali vsaj velika večina), in jih je veliko, je priporočljiva uporaba sistema FOIL.

6 Zaključek

6.1 Dosežki in priporočila

Naloga doktorske disertacije je bila izboljšati algoritem HYPER. Izboljšave je bilo nato potrebno temeljito preizkusiti, in algoritem primerjati z nekaj standardnimi ILP algoritmi.

V okviru razvijanja in testiranja verzij sistema HYPER^2 smo razvili okoli 30 verzij. Nekatere se že od začetka niso izkazale kot primerne (ali smo postavili prehude omejitve in običajno niso našle rešitve ali pa smo si pri obdelavi zapomnili preveč podatkov, ki nam bi lahko koristili pri nadaljnji obdelavi in prekoračili omejitev SICStus prologa, ki lahko naslavlja le 256MB pomnilnika). 15 verzij sistema HYPER^2, ki so se med seboj razlikovale po posameznih izboljšavah, smo temeljito testirali na dveh domenah programske sinteze in jih temeljito primerjali med seboj z originalnih algoritmom glede uspešnosti, porabljenega časa, števila izostrenih hipotez in števila klicev meta-interpreterja. Pravzaprav je nekaj teh verzij nastalo med samim testiranjem, ker so se pri testiranju izkazale očitne pomakljivosti nekaterih pristopov.

Dosežki tega doktorskega dela vsebujejo razvoj sistema HYPER^2 in predvsem ugotovitve kateri pristopi (poskusi izboljšav) delujejo in kateri ne. Kot najbolj pomembne izboljšave so se izkazale:

- Grajenje gozda stavkov, kjer iz vsakega podanega začetnega stavka izhaja eno drevo. Same hipoteze so sestavljene iz kazalcev na stavke. Na ta način nam ni potrebno večkrat izračunavati naslednikov enega stavka (ki se lahko pojavi v več hipotezah). Najbolj pomembna prednost grajenja gozda stavkov pa je enostavnost preverjanja obstoja kopij stavkov in še lažjega preverjanja kopij hipotez, kar preprečuje ponavljanje procesiranja. Grajenje gozda stavkov in preprečevanje ponavljanja procesiranja zmanjša število generiranih in izostrenih hipotez (v odvisnosti od domene in učnih primerov). Na domeni path originalni HYPER generira do 75% kopij hipotez in izostri do 79% že izostrenih hipotez.
- Same hipoteze poznajo učne primere, ki jih pokrivajo. Ker se pri ostrenju pokrivanje primerov lahko samo zmanjša, je treba pri računanju pokritosti učnih primerov s strani naslednikov preveriti samo primere, ki so jih pokrivali starši. Zaradi tega se kompleksnost obdelave ene hipoteze zmanjša za delež primerov, ki

zanesljivo niso pokriti (ker jih že starševska hipoteza ni pokrivala). Skupna kompleksnost se zmanjša za povprečen delež pokritih primerov (ki ga je težko oceniti, ker je odvisno od domene in vrstnega reda preiskovanja).

- Uporaba informacije o vhodno/izhodnem tipu spremenljivk v glavah stavkov hipoteze. Če uporabimo tako informacijo, se nekoliko zmanjša prostor možnih hipotez, kar že samo po sebi lahko pospeši delovanje. Poleg tega pa se izkaže, da je lahko poskus ovrednotenja hipoteze, ki s tem znanjem ne bi bila upoštevana (in recimo uporablja drugače izhodno spremenljivko, ki še nima znane vrednosti kot vhod pri klicu predznanja) lahko povzroči upočasnitev delovanja (zaradi možnega "neskončnega" števila odgovorov napačno klicanega predznanja). Ta sprememba sicer potencialno poveča kompleksnost izboljšave ene hipoteze, vendar dodatno usmerja iskanje po prostoru, ki se zaradi tega hitreje konča.

Nekaj "izboljšav", ki niso prinesle pričakovanega izboljšanja:

- Stavki poznajo primere, ki jih pokrivajo. Ta osnovna izboljšava, od katere smo pričakovali eno izmed večjih izboljšanj, se ni obnesla po pričakovanjih (zato je bila tudi narejena verzija, ki si zapomni pokritost na nivoju hipotez). Izkazalo se je, da sta glavna problema za to izboljšavo prekrivanje pokritosti – ko več stavkov v hipotezi pokrije isti primer. To izniči prednost, da je potrebno za vsak stavek le enkrat izračunati pokritost. Glavna težava pa je bila dejstvo, da stavki, odvisni od preostanka hipoteze, pokrivajo primere drugače v vsaki hipotezi (tega smo se sicer zavedali, ampak nismo pričakovali tako velikega učinka). Izboljšava je teoretično dosti bolj učinkovita, saj bi v idealnem primeru zmanjšala kompleksnost na 2/L (kjer je L število stavkov v ciljni hipotezi)originalne kompleksnosti, vendar je to le v idealnem primeru (nerekurzivna domena, kjer se pokritosti posameznih stavkov ne prekrivajo).
- Izboljšava, ki upošteva, da je nadmnožica kompletne hipoteze (pogosto) tudi kompletna, ni prinesla pričakovanega izboljšanja. Problem je predvsem veliko število kompletnih hipotez, ki so kandidati za podmnožico hipoteze, ki jo trenutno preverjamo. Dodatne težave prinese dejstvo, da v okviru meta-interpreterja vrstni red stavkov lahko vpliva na to, ali hipoteza pokriva določen primer ali ne.

Dodaten dosežek tega doktorskega dela je analiza kompleksnosti algoritma in velikosti preiskovanega prostora. Le ta je izkazala, da je prostor eksponentno odvisen od mestnosti predikatov predznanja, največje dovoljene dolžine stavkov in največjega dovoljenega

števila predikatov v stavku. Sam algoritem pa je več kot eksponentno (x^x) odvisen od mestnosti predikatov predznanja, medtem ko je od ostalih parametrov polinomsko odvisen. Pri vsem tem pa je treba upoštevati, da je odvisnost običajno dosti večja od kvalitete učnih podatkov. Tako se recimo izkaže (kljub temu, da enačba napoveduje linearno odvisnost od števila učnih podatkov), da ko povečujemo število podatkov čas procesiranja najprej naraste, nato pa pogosto začne padati (na začetku, ko je na voljo malo podatkov sistem hitro najde nepravilno rešitev, ko se število podatkov povečuje sistem bolj pogosto po daljšem iskanju najde pravilno rešitev, ko še povečamo število podatkov se hitreje usmeri proti ciljni hipotezi, ter zaradi tega hitreje najde rešitev). Tako je lahko čas reševanja dveh podobnih problemov (ista domena, enako število pozitivnih učnih podatkov, enako negativnih učnih podatkov) zelo različen.

Dosežek tega doktorskega dela je tudi primerjalno testiranje sistema HYPER^2 in nekaterih standardnih ILP sistemov (ALEPH, CHILLIN, CPROGOL, FOIL, MARKUS) na devetih domenah programske sinteze (testiranje je bilo osredotočeno predvsem na občutljivost sistemov na manjkajoče podatke). Ti sistemi so bili izbrani tako, da pokrijejo različne pristope k ILP.

Pri teh testiranjih se je izkazalo, da je HYPER^2 sposoben generirati pravilne rešitve iz zelo majhnega števila učnih primerov. Občasno generira pravilne rešitve tudi, če nekateri ključni učni primeri niso na voljo, če so od delov hipoteze, ki bi se jih moral naučiti iz manjkajočih primerov, odvisni preostali deli hipoteze (ker generiramo celotne hipoteze, lahko manjkajoče pozitivne učne primere nadomesti odvisnost ostalih delov hipoteze, za katere pa ima na voljo dovolj pozitivnih učnih primerov). Tako na primer, lahko zgradi zaustavitveni pogoj rekurzije brez neposrednega pozitivnega učnega primera. To se še posebej pozna na domenah odd – even, obeh domenah member ter last in next.

Na domenah path in predvsem insertionsort pride do izraza kompleksnost specializacije celotnih hipotez. Sistem je sicer sposoben rešiti oba problema, vendar je (predvsem na domeni insertionsort) občasno ali celo pogosto ustavljen, ker zaradi velikega števila učnih primerov preseže dovoljen čas obdelave. Kompleksnost same domene pride do izraza v domeni quicksort, kjer sistem ni sposoben najti rešitve (prej pregleda dovoljeno število hipotez, če mu to število povečamo, pa običajno porabi ves razpoložljiv spomin).

Iz primerjave lahko zaključimo, da je HYPER^2 uporaben predvsem za manjše in enostavnejše rekurzivne probleme, saj celotni prostor hipotez postane prevelik za učinkovito preiskovanje pri večjih problemih. Poleg tega v realnih problemih običajno v

podatkih najdemo šum. Obdelavi šuma sistem HYPER^2 ni namenjen. Možne bi bile enostavne modifikacije, ki bi dovoljevale hipoteze, ki pokrijejo le določen delež pozitivnih učnih podatkov, ter bi v končni množici dovoljevale določen delež negativnih učnih primerov, vendar bi bilo potrebno z dodatnim eksperimentiranjem poiskati primerne (še sprejemljive) deleže.

Enostavne nerekurzivne probleme je HYPER^2 prav tako sposoben reševati kot rekurzivne, vendar pri teh nima prednosti pred ostalimi sistemi (saj so pri nerekurzivnih problemih stavki hipoteze neodvisni med seboj, ter ne zahtevajo sprotne obdelave).

Iz same primerjave različnih sistemov na domenah programske sinteze lahko zaključimo: če je na voljo malo učnih primerov in obstaja sum, da nekateri ključni učni primeri manjkajo, je najbolje uporabiti sistem HYPER^2, če je na voljo več učnih primerov, a kljub temu še ne vsi, je priporočljivo uporabiti MARKUS in poskusiti tudi s HYPER^2; če so pa na voljo vsi učni primeri (ali vsaj velika večina), in jih je veliko, pa je priporočljiva uporaba sistema FOIL. Taki zaključki so predvsem posledica opaženih zahtev sistemov za uspešno rešitev problema: HYPER^2 običajno najde pravilno rešitev, če je le ta najkrajša definicija, ki ustreza učnim podatkom, vendar je postopek lahko dolgotrajen (ob velikem številu učnih podatkov) in občasno se sistem izgubi v kakšni slepi ulici (v bolj kompleksnih domenah). MARKUS zahteva vsaj en pozitivni učni primer, ki pokriva zaustavitveni pogoj rekurzije in vsaj en pozitivni učni primer prvega rekurzivnega koraka. ALEPH in PROGOL potrebujeta vsaj tri pozitivne učne primere, ki pokrivajo zaustavitveni pogoj rekurzije in vsaj tri pozitivne učne primere prvega rekurzivnega koraka. FOIL potrebuje vsaj en pozitivni učni primer zaustavitvenega pogoja rekurzije in vsaj en pozitivni učni primer, ki se z enim rekurzivnim korakom prevede na drug pozitivni učni primer. CHILLIN pa potrebuje lepo porazdelitev pozitivnih in negativnih učnih primerov vseh globin, da v pravem trenutku zaustavi generalizacijo (pogosto se zgodi, da zaradi pomanjkanja nekaterih pozitivnih učnih primerov, premalo generalizira in naredi prespecifične hipoteze, ali pa zaradi pomanjkanja nekaterih negativnih učnih primerov preveč generalizira in naredi presplošne hipoteze).

Torej je priporočljivo glede na kompleksnost domene, število učnih primerov in verjetnost manjkajočih ključnih učnih primerov, izbrati med sistemi HYPER^2, MARKUS in FOIL.

Izvirni prispevki znanosti so:

- razvoj in implementacija sistema HYPER^2
- analiza velikosti preiskanega prostora
- analiza kompleksnosti algoritma
- ugotovitve testiranja modifikacij:
  - največje izboljšanje učinkovitosti prinese uporaba dodatne informacije o vhodno/izhodnem tipu spremenljivk v glavi iskanih predikatov
  - večjo pohitritev prinese gradnja gozda hipotez
  - večjo pohitritev prinese informacija, katere primere pokriva posamezna hipoteza
  - nepričakovano slabo se obnese uporaba informacije o tem, katere primere pokriva posamezen stavek v hipotezi
  - nepričakovano slabo se obnese upoštevanje dejstva, da je nadmnožica kompletne hipoteze (običajno) kompletna
- ugotovitve primerjalnega testiranja z drugimi ILP sistemi:
  - HYPER^2 se običajno bolje obnese na zelo redkih učnih množicah in je pogosto sposoben generirati pravilne rešitve iz zelo majhnega števila učnih podatkov
  - HYPER^2 je zelo občutljiv na kompleksnost domene
  - FOIL je najbolj občutljiv na manjkajoče podatke, a je običajno najhitrejši
  - ALEPH in PROGOL potrebujeta večje število učnih primerov določene oblike
  - CHILLIN potrebuje lepo porazdeljene pozitivne in negativne učne primere
  - MARKUS je običajno dober kompromis, a mu lahko enostavne domene povzročajo težave

6.2 Nadaljnje delo

Kot že rečeno, ima HYPER^2 največje težave, ko ima domena veliko učnih primerov. Pri takih domenah namreč pogosto prekorači ali našo omejitev časa izvajanja ali pa prostorsko omejitev SICStus prologa. Zaradi tega bi bilo potrebno sistem prevesti v kakšen drug jezik, z

dodanim prologovim interpreterjem, kjer bi lahko določene poizvedbe poiskali na hitrejši način, kot ga uporablja prolog.

Drug opaznejši problem pa je, ko HYPER^2 zaide v slepo ulico. Hipoteze na tej poti pogosto vsebujejo stavke, ki ne pokrivajo nobenega pozitivnega primera. Da bi to odpravili, bi bilo potrebno kaznovati hipoteze, ki vsebujejo take stavke. Vendar bo potrebno najti primeren kompromis, drugače bo HYPER^2 izgubil večji del sposobnosti, da najde pravilno hipotezo kljub pomanjkljivim učnim primerom. Drug mogoč način je, da poskusimo kaznovati hipoteze, ki vsebujejo nespecializirane, stavke in spremenimo izbiranje stavka v hipotezi tako, da se ne vzame prvi, ki pokriva negativne učne primere ampak najkrajši stavek, ki pokriva negativne učne primere. Oba načina pa zahtevata kar nekaj poskusov, da se najde primerna velikost kazni.

Dodatno bi bilo potrebno nekaj najboljših verzij združiti v eno verzijo, ki bi znala iskati po načelu najprej najboljši ali s snopom ter znala izračunavati pokritost na nivoju stavkov (kar bi bilo bolj učinkovito za nerekurzivne domene) ali pa na nivoju hipotez (za rekurzivne domene).

Poleg tega bi bilo potrebno preizkusiti nekatere modifikacije, ki bi omogočale delovanje sistema na šumnih podatkih, kot je recimo dopuščanje določenega deleža pokritosti negativnih učnih podatkov in nepokritosti pozitivnih učnih podatkov hipotezah.

7 Literatura

[1] Bergadano, F., Gunetti, D. *Inductive Logic Programming: From Machine Learning to Software Engineering.* MIT Press, Cambridge, MA, 1995.

[2] Blockeel, H., de Raedt L. Top-down induction of first order logical decision trees. *Artificial Intelligence* 101, 285-297, 1998

[3] Bratko, I. Prolog Programming for Artificial Intelligence, Third Edition, Addison-Wesley, 2001

[4] Bratko, I. Refining complete hypotheses in ILP. *Proc. 9th Int. Workshop ILP-99* (ur. S. Džeroski and P. Flach) Springer-Verlag, 1999

[5] Bratko, I., Grobelnik, M. Inductive learning applied to program construction and verification. *Proc. ILP Workshop 93* (ur. S. Muggleton), 1993.

[6] Brazdil, P., Jorge, A. Exploiting algorithm sketches in ILP. *Proc ILP Workshop 93* (ur. S. Muggleton), 1993

[7] Buntime W., Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2): 149 – 176, 1988

[8] Cameron-Jones, R.M. and Quinlan, J.R., Efficient Top-down Induction of Logic Programs. *SIGART Bulletin*, 5(1):33—42, 1994.

[9] De Raedt, L. Inductive logic programming made easy. *Machine Learning and Applications* (Lecture Notes), 1999

[10] De Raedt, L., Džeroski S., First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70 375-392, 1994

[11] De Raedt, L., (ur.), *Advances in Inductive Logic Programming*. IOS Press, 1996

[12] Džeroski, S., Lavrač, N. (ur.), *Relational Data Mining*, Springer, 2001

[13] Grobelnik, M. Induction of Prolog programs with MARKUS. *Proc. 3rd International Workshop on Logic Program Synthesis and Transformation*, 55-63. Springer, 1993

[14] Grobelnik, M. Markus – an optimised model inference system. *Proc. ECAI Workshop on Logical Approaches to Machine Learning*, 1992

[15] Kononenko, I. *Strojno učenje*, Založba FE in FRI, 1997

[16] Lavrač N., Džeroski S. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994 (http://www-ai.ijs.si/SasoDzeroski/ILPBook/).

[17] Midelfart, H. A bounded search space of clausal theories. *Proc. 9th Int. Workshop ILP-99* (eds S. Džeroski and P. Flach) Springer-Verlag, 1999.

[18] Mitchell, T. *Machine Learning,* McGraw Hill, 1997

[19] Mozetič, I. The Role of abstractions in learning qualitative models. *Proc 4th Int. Conf. Machine Learning*. Morgan Kaufmann, 1987.

[20] Muggleton, S. editor, *Inductive Logic Programming*. Academic Press, 1992

[21] Muggleton, S. Inverse entailment and Progol. *New Generation Computing*, 13, 245-286, 1995

[22] Muggleton, S. Learning from positive data. V S. Muggleton, (ur.), *Proc. 6th International Workshop on Inductive Logic Programming*, 225-244, 1996

[23] Muggleton, S. Buntine, W. Machine invention of first-order predicates by inverting resolution. *Proc. 5th International Conference on Machine Learning,* 339-352, Morgan-Kaufmann, 1988

[24] Muggleton, S., Feng, C. Efficiant induction of logic programs. *Proc. 1st Conference on Algoritmic Learning Theory,* 368 – 381, Ohmsha, 1990

[25] Niblett, T. A study of generalization in logic programs. *Proc. 3rd Euroopean Working Session on Learning,* 131-138, Pitman, 1988

[26] Nienhuys-Cheng, S.-H., de Wolf, R. *Foundations of Inductive Logic Programming*. Springer, 1997

[27] Plotkin, G. A note on inductive generalization. V Meltzer, B., Michie, D. (ur.), *Machine Intelligence 5*, 153-163, Edinburgh University Press, 1969

[28] Pompe, U. Inductive Constraint Logic Programming. *Ph.D. Thesis*, University of Ljubljana, Faculty of Computer and Information Science, 1998

[29] Quinlan, J.R. Learning logical definitions from relations. *Machine Learning*, 5, 239-266, 1990

[30] Quinlan, J. R., Cameron-Jones, R. M. FOIL: A midterm report. V P. Brazdil, (ur.), *Proc. 6th European Conference on Machine Learning, volume 667 of Lecture Notes in Artificial Intelligence*, 3-20, Springer Verlag, 1993

[31] Sakakibira, Y. Kobayashi, S. Inductive inference of formal languages. *Journal of the Japanese Society of Artificial Intelligence*, 14, 781-789, 1999

[32] Shapiro, E. *Algoritmic Program Debugging.* MIT Press, 1983

[33] Srinivasan, A. The Aleph Manual. *Technical Report*, Computing Laboratory, Oxford University, 2000 (http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/)

[34] Zelle, J. M., Mooney, R. J. Learning to parse database queries using inductive logic programming. *Proc. 14th National Conferenece on Artificial Intelligence*, 1050-155, AAAI Press/MIT Press, 1996

[35] Zelle, J. M., Mooney, R. J. And Konvisser, J.B. Combining top-down and bottom-up techniques in inductive logic Programming. *Proc. 11th International Conference on Machine Learning*, 343-351, Morgan Kaufmann, 1994

Dodatek A Grafična primerjava performans različnih verzij sistema HYPER^2

Dodatek A* je razširitev poglavja 4, kjer je bila prikazana primerjava različnih verzij sistema HYPER^2.

V dodatku bomo s pomočjo (dodatnih) grafov prikazali različno obnašanje različnih verzij sistema HYPER^2. Pri poskusih smo merili:

- uspešnost (ali je rešitev pravilna)
- čas reševanja
- število izostrenih hipotez
- število klicev meta-interpreterja

Poleg grafov, ki so v poglavju 4, ki prikazujejo te meritve na celotnih domenah, tu prikažemo še meritve ločene na pravilno in nepravilno rešene probleme.

Merili smo na domenah:

- member
- path

* Celoten dodatek je na dodanem CD.

Dodatek B Kratka navodila za uporabo sistema HYPER2

V dodatku B bomo na kratko opisali navodila za uporabo sistema HYPER^2. Glede na to, da je delovanje sistema opisano v tretjem poglavju se bomo tu osredotočili le nastavitve s pomočjo katerih usmerjamo iskanje sistema. Poleg tega bomo tudi razložili kako se sistemu poda opis problema ter množico učnih primerov. Ker je sam sistem pisan v SICStus narečju jezika Prolog, ter teče v intrepreterju tega jezika, so vse nastavitve in podatki podani kot dejstva v Prologu.

Nastavitve

Z nastavitvami, ki so del kode sistema HYPER^2, lahko določimo sledeče lastnosti preiskovanega prostora in načina preiskovanja (prikazane so privzete vrednosti):

- **max_proof_length(15).** Določi največjo dovoljeno dolžino dokaza pokritosti primera, ki ga računa meta-interpreter.
- **min_clauses(1).** Določi najmanjše število stavkov v hipotezi.
- **max_clauses(3).** Določi največje število stavkov v hipotezi.
- **max_clause_length(5).** Določi največje število predikatov v stavku.
- **max_hypothesis_refined(700).** Določi največje število specializiranih hipotez.
- **max_beam_width(50).** Določi največjo širino snopa.

Opis domene

Z opisom domene določimo začetne stavke, definiramo transformacije spremenljivk v izraze, ter deklariramo predznanje.

Z start_clause določimo začetne stavke:

- **start_clause([predikat(L1, L2)] / [L1:tip1, L2:tip2]).**
- **start_clause([predikat(L1,L2)] / [L1:tip1], [L2:tip2]).**

kjer je **predikat** ime iskanega predikata, **L1** in **L2** sta argumenta tega predikata, **tip1** in **tip2** pa sta tipa le-teh. Prva varianta je za verzije, ki ne upoštevajo vhodno-izhodne informacije v glavah stavkov, druga varianta je pa za verzije, ki to informacijo upoštevajo. Pri tem predstavlja **[L1:tip1]** množico vhodnih argumentov in **[L2:tip2]** množico izhodnih argumentov.

Z backliteral deklariramo predikate predznanja:

- **backliteral(predikat(L1,L2),[L1:tip1],[L2:tip2]).**

kjer je **predikat** ime predikata predznanja, **L1** in **L2** sta argumenta tega predikata, **tip1** in **tip2** pa sta tipa le-teh. Pri tem predstavlja **[L1:tip1]** množico vhodnih argumentov in **[L2:tip2]** množico izhodnih argumentov. Če bi radi poiskali rekurzivno definicijo iskanega predikata, moramo sam iskani predikat tudi deklarirati kot predznanje. Vse predikate predznanja (razen iskanega predikata) je potrebno tudi definirati.

Z term lahko definiramo pretvorbo spremenljivk v izraze. Poglejmo si to na primeru pretvorbe seznamov:

- **term(seznam, [X|L], [X:element, L:seznam]).**
- **term(seznam, [], []).**

Prva vrstica določa, da se lahko spremenljivka tipa **seznam** lahko spremeni v izraz **[X|L]**, kjer je **X** tipa **element** in **L** tipa **seznam**. Druga vrstica pa določa da se spremenljivka tipa **seznam** lahko spremeni v izraz **[]**, ki je brez spremenljivk.

Z prolog_predikate dopovemo meta-interpreterju, da je določen predikat zunanji predikat in nočemo, da se njegovo izvajanje šteje v dolžino dokaza pokritosti primerov:

- **prolog_predicate(predikat(L1,L2)).**

kjer je **predikat** ime tega predikata, **L1** in **L2** pa sta njegova argumenta.

Definicija učnih primerov

Učne primere podamo sistemu HYPER^2 z naslednjo definicijo:

- **ex(predikat(a,b)).**
- **nex(predikat(b,a)).**

kjer z **ex** določimo pozitivne učne primere in z **nex** negativne učne primere. Pri tem je **predikat** ime iskanega predikata za katerega velja ta učni primer, **a** in **b** pa sta argumenta tega predikata.