alternatívne riešenia, pretože nahrádzané a nahrádzajúce hodnoty nemusia spĺňať relevantné ohraničenia rovnakým spôsobom.

Okrem jednoduchej substitúcie jednej hodnoty inou hodnotou tej istej premennej je možný aj všeobecnejší prípad, keď sa nahrádza kombinácia hodnôt rôznych premenných inou kombináciou hodnôt tých istých premenných [80]. Takáto úprava nie je realizovaná redukciou domén premenných, ale úpravami príslušných ohraničení. Príkladom môže byť napr. čiastočná zameniteľnosť [54], keď dve hodnoty nejakej premennej sú vtedy čiastočne zameniteľné, ak ľubovoľné riešenie obsahujúce prvú hodnotu je možné zámenou tejto hodnoty za hodnotu druhú transformovať na iné riešenie, pričom nejaká podmnožina premenných musí zmeniť svoje hodnoty tiež. Ak by napr. domény premenných z príkladu na obr. 2.2 mali tvar

D_1 : žirafa, európa
D_2 : puška, kurín
D_3 : kvér, íver
D_4 : pór, eso

tak riešenie

(európa, kurín, íver, pór)

môže byť transformované na iné náhradou hodnoty premennej V_3 “íver” za hodnotu “kvér”, pričom sa súčasne musí vhodne zmeniť aj hodnota premennej V_2.

4.2 Prehľadávacie algoritmy

Základným algoritmom pre hierarchické prehľadávanie priestoru riešení je algoritmus *spätného navracania*, ktorý realizuje vlastne slepé prehľadávanie tohto priestoru do hĺbky. Samotný algoritmus spätného navracania bol v histórii niekoľkokrát znovuobjavený [19].

Premenným sa priraďujú hodnoty sekvenčne. Platnosť každého ohraničenia je testovaná hneď ako je to možné (teda hneď po priradení hodnôt všetkým tým premenným, nad ktorými je dané ohraničenie definované). Ak sa zistí porušenie nejakého ohraničenia, algoritmus sa vráti k premennej, ktorá získala svoju hodnotu ako posledná a pokúsi sa jej priradiť hodnotu novú. Ak to nie je možné, vracia sa k premennej, ktorá získala svoju hodnotu ako predposledná atď. Pri tomto navracaní sa vlastne používa obrátené chronologické poradie premenných než pri priraďovaní hodnôt.

Akonáhle sa zistí, že v nejakom podpriestore priestoru riešení (danom nejakým parciálnym priradením hodnôt niektorým premenným) nie je splnené nejaké ohraničenie, tak tento podpriestor už nebude ďalej prehľadávaný delením na menšie a menšie podpriestory, ale celý tento podpriestor sa vylúči z ďalšieho prehľadávania, pretože počet porušení ohraničení by priradením hodnôt aj ostatným (zatiaľ neviazaným) premenným už neklesol. Takýmto spôsobom dochádza k orezávaniu priestoru prehľadávania, čím sa znižuje počet vykonaných priradení.

V prípade, že riešenie existuje, sa algoritmu v určitom štádiu hľadania podarí zúžiť priestor prehľadávania iba na jeden bod a priradiť hodnoty všetkým premenným bez toho, aby nejaké ohraničenie bolo porušené. Ak však riešenie neexistuje, tak po určitej dobe sa algoritmus navracaním vráti až k premennej, ktorá získala hodnotu ako prvá, s tým, že pokus o priradenie novej hodnoty tejto premennej zlyhá.

Premenné sú usporiadané v určitom poradí a podľa tohto usporiadania sú postupne vyberané pre priradenie hodnoty. Podobne hodnoty v doménach premenných sú tiež určitým spôsobom usporiadané a vyberané z domén pre priradenie premenným podľa daného poradia. Tieto usporiadania premenných a hodnôt v doménach premenných môžu byť statické (platné počas celej doby prehľadávania priestoru riešení) alebo dynamické (počas hľadania riešenia sa môžu meniť).

Algoritmus spätného navracania (backtracking – BT) možno potom vyjadriť napr. v tvare: [8]

```
define method backtracking( i::<integer> )
   if ( and ( 0 < i, i <= n ) )
      V[i] := next( D[i], V[i] );
      if ( not ( V[i] ) )
         backtracking ( i-1 );
      end if;
      if ( satisfy_all_constraints ( ) )
         backtracking ( i+1 );
      else
         backtracking ( i );
      end if;
   end if;
end method backtracking;
```

[8] Funkcia `next` ako svoj vstup očakáva zoznam hodnôt a hodnotu. Ak zadaná hodnota sa nenachádza v zozname, tak vracia prvú hodnotu zoznamu. Ak sa zadaná hodnota nachádza v zozname, tak vracia nasledujúcu hodnotu zo zoznamu. Ak zadaná hodnota je poslednou hodnotou v zozname, tak funkcia vracia hodnotu `#f` (false).

Pomocou zaznamenávania (a vizualizácie) činnosti algoritmu, menovite toho, kedy ktorej premennej priradil ktorú hodnotu z jej domény a s akým úspechom, je možné získať *strom prehľadávania*. Jeho uzly reprezentujú stavy riešenia úlohy, charakterizované určitým (parciálnym) priradením hodnôt premenným. Hrany označujú prechody medzi stavmi. Tieto prechody môžu byť dvojakého typu:

- premennej sa priradí nejaká hodnota z jej domény (a teda v strome prehľadávania sa vygeneruje nový uzol)
- premenná s priradenou hodnotou sa uvoľní – nebude mať priradenú žiadnu hodnotu a teda sa stáva neviazanou.

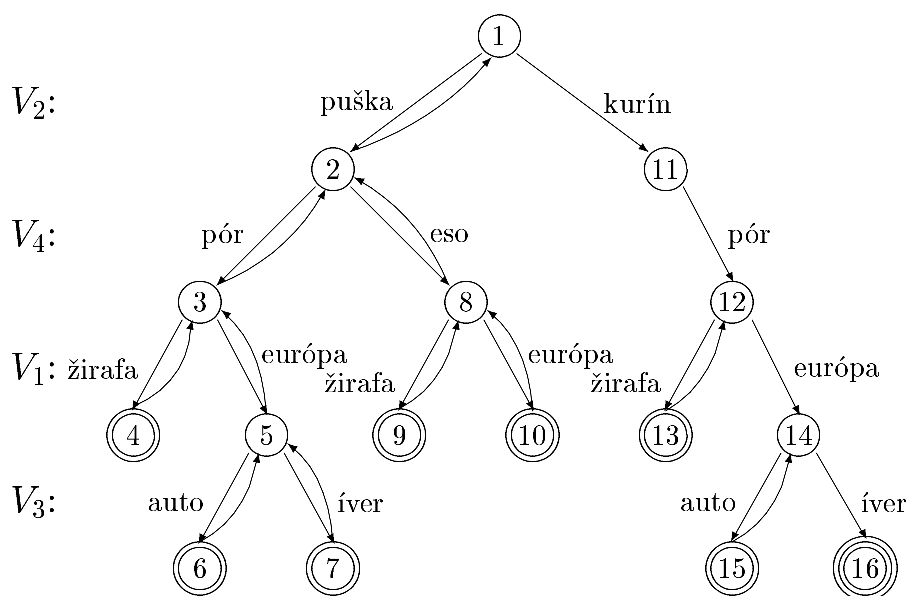
Obidva uvedené typy prechodov znamenajú pohyb v hierarchii podpriestorov priestoru prehľadávania. Priradenie zužuje podpriestor na jednu jeho časť (charakterizovanú priradenou hodnotou), uvoľnenie naopak znamená rozšírenie podpriestoru.

Konkrétny tvar stromu prehľadávania závisí od spôsobu výberu premenných a hodnôt. Ak by sa použili rôzne usporiadania premenných alebo hodnôt v doménach premenných, získali by sa stromy prehľadávania rôznych tvarov. Ak riešený problém má práve jedno riešenie, tak všetky možné stromy prehľadávania by smerovali priamejším alebo spletitejším spôsobom k tomuto riešeniu. Ak by však problém mal viac riešení, tie by mohli byť nájdené rôznymi stromami v rôznom poradí (a pri hľadaní iba jedného riešenia by nebolo nájdené vždy to isté riešenie).

Ak by bol algoritmus spätného navracania použitý pre ilustračný príklad na obr. 2.2, pričom by bolo použité zmenené usporiadanie premenných V_2, V_4, V_1 a V_3 a usporiadanie hodnôt v doménach by ostalo nezmenené, tak by bol vygenerovaný strom prehľadávania podľa obr. 4.6.

V tomto strome operácie priradenia hodnôt premenným sú znázornené ako orientované hrany smerujúce nadol a operácie uvoľnenia premenných zase ako hrany orientované nahor. Dvojitý krúžok označuje situáciu keď bolo detekované porušenie nejakého ohraničenia a následne bolo vyvolané navracanie k predošlým stavom hľadania riešenia. Trojitý krúžok označuje úspešné vygenerovanie riešenia – stav keď priestor prehľadávania bol redukovaný na jeden bod a pritom nebolo detekované žiadne porušenie definovaných ohraničení.

Z obrázku je zrejmé orezávanie priestoru prehľadávania. Pri prehľadávaní došlo štyrikrát k orezaniu (v uzloch 4, 9, 10 a 13), pričom celkovo sa ušetrilo



Obr. 4.6: Trasovanie riešenia ilustračného príkladu pomocou spätného navracania pre poradie premenných V_2, V_4, V_1 a V_3.

generovanie ôsmich uzlov v strome prehľadávania (v podstromoch týchto štyroch uzlov).

Algoritmus spätného navracania nemá však len prednosti ale aj nevýhody. Obzvlášť sú mu vyčítané tieto dve necnosti:

- opakované zlyhávanie hľadania riešenia v rôznych častiach priestoru prehľadávania z tých istých príčin (označované ako *trashing* [86])
- redundantné prehľadávanie.

Strom prehľadávania na obr. 4.6 ilustruje obe tieto nevýhody spätného navracania. Príkladom opakovaného zlyhania sú uzly 6 a 15. V oboch zlyhalo priradenie hodnoty "auto" premennej V_3 z toho istého dôvodu – premenná V_4 mala priradenú hodnotu "pór" (uzly 3 a 12) a následkom toho bolo porušené ohraničenie $C^{3,4}$. Dôvodom pre toto chovanie je to, že algoritmus spätného navracania nemá pamäť.

Absencia pamäti je tiež príčinou nasledujúceho redundantného chovania algoritmu. Na začiatku prehľadávania bolo zistené, že hodnota "pór" je kompatibilná iba s jednou hodnotou z domény D_1 (hodnota "európa") – uzly 3, 4 a 5. Keďže však hodnota "puška" (uzol 2) nebola kompatibilná so žiadnou hodnotou z domény D_3, navracanie sa vrátilo do uzla 2 a naučená znalosť o vzťahu hodnoty "pór" k doméne D_1 bola zabudnutá. V neskoršom štádiu prehľadávania táto znalosť musela byť znovuobjavená (uzly 12, 13 a 14). Teda algoritmus to isté musel redundantne objavovať druhý raz.

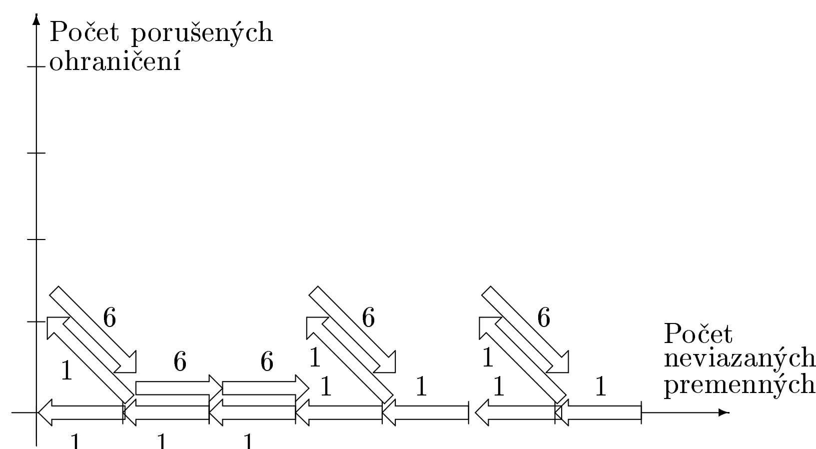
Toto bolo príčinou vzniku radu ďalších algoritmov snažiacich sa tieto nevýhody štandardného spätného navracania eliminovať. V zásade tieto algoritmy možno rozdeliť do dvoch hlavných skupín – skokové algoritmy (snažiace sa o znižovanie počtu nutných navracaní) a pamäťové algoritmy (založené na znižovaní počtu kontrol platnosti jednotlivých ohraničení). Samozrejme sú možné aj hybridy využívajúce princípy oboch skupín [82], [107].

Niektoré z týchto algoritmov sú uvedené v nasledujúcich podkapitolách. Tieto vylepšenia algoritmu spätného navracania generujú vo všeobecnosti menšie stromy prehľadávania. Cenou za to však je to, že majú väčšiu výpočtovú réžiu ako samotný algoritmus spätného navracania. Z tohto dôvodu je možné, že niekedy štandardný algoritmus spätného navracania nájde riešenie skôr než jeho zdokonalené varianty.

Zmenšovanie priestoru prehľadávania pri použití prehľadávacích algoritmov má iný charakter ako v prípade redukčných algoritmov (aj keď v prípade binárnych premenných to nie je viditeľné navonok). Zatiaľ čo pri redukčných algoritmoch bol prítomný princíp "vypustiť všetky kombinácie hodnôt premenných obsahujúce hodnotu, ktorá je vypúšťaná z domény nejakej premennej", tak teraz to je "ponechať všetky kombinácie hodnôt premenných obsahujúce hodnotu priraďovanú nejakej premennej".

Aplikáciou tohto princípu v kombinácii s postupným viazaním premenných vzniká tvar trajektórie v stavovom priestore, ktorý je typický pre prehľadávacie algoritmy. Táto trajektória je znázornená na obr. 4.7.

Z hľadiska stavového priestoru podobne ako algoritmy šírenia ohraničení aj uvedené prehľadávacie algoritmy využívajú iba dva typy riešiacich krokov. V tomto prípade to sú priradenie a uvoľnenie. Keďže je možné aj také priradenie, ktoré spôsobí porušenie nejakého ohraničenia, tak priraďovací krok môže byť znázorňovaný dvojako. Naproti tomu je uvoľňovaná vždy tá premenná ako prvá, ktorej viazanie na konkrétnu hodnotu z jej domény spôsobilo všetky porušenia ohraničení – a teda uvoľňovací krok vždy končí na vodorovnej osi. Pri algoritme spätného navracania dĺžka kroku (vo vodorovnom smere) je vždy jednotková. Pri skokových algoritmoch táto dĺžka



Obr. 4.7: Typická trajektória pre prehľadávacie algoritmy.

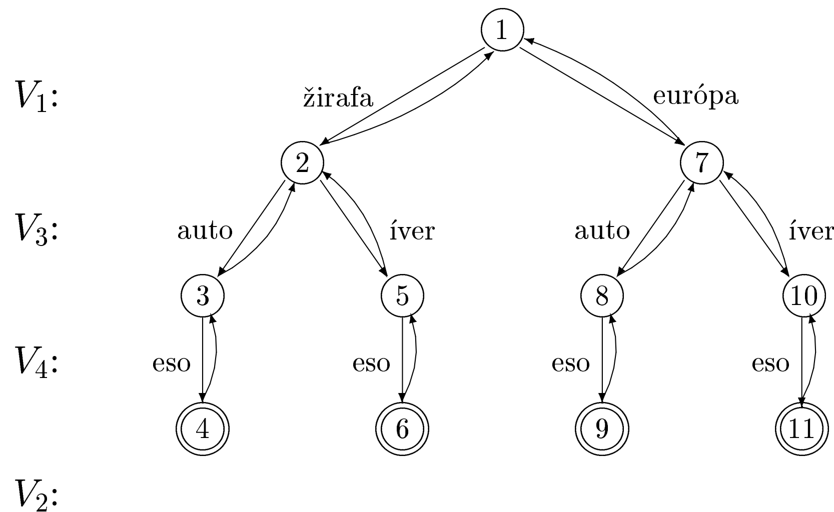
môže byť väčšia (a teda jeden uvoľňovací krok nie je inverziou jedného priraďovacieho).

Tieto dva typy riešiacich krokov sa striedajú podľa toho, či momentálnej etape hľadania riešenia zodpovedá v strome prehľadávania pohyb smerom nadol alebo nahor. Ich počet nemôže byť vopred presne daný. Počet priraďovacích krokov je minimálne n (je potrebné priradiť hodnoty všetkým premenným) a maximálne $\| D_1 \| \times \ldots \times \| D_n \|$ (v najhoršom prípade je potrebné vyskúšať všetky možné kombinácie hodnôt z domén premenných). V ideálnom prípade nie je nutné realizovať žiadny uvoľňovací krok. Celkový počet uvoľňovacích krokov nepresiahne počet priraďovacích krokov (v prípade, že úloha nemá riešenie, je potrebné uvoľniť všetky realizované priradenia, a teda počet priraďovacích krokov je rovnaký ako počet uvoľňovacích krokov).

Ak úloha má riešenie, tak trajektória končí v počiatku. Ak však úloha riešenie nemá, tak potom trajektória končí v tom istom bode, v ktorom začala (žiadna premenná nemá priradenú hodnotu a teda žiadne ohraničenie nemôže byť porušené).

4.2.1 Návrat k príčine neúspechu

Ak by v príklade na str. 51 v doméne D_4 nebola prítomná hodnota "pór" a ak by sa zmenilo usporiadanie premenných na V_1, V_3, V_4 a V_2, tak by strom prehľadávania vygenerovaný pri použití algoritmu spätného navracania mal tvar podľa obr. 4.8.



Obr. 4.8: Trasovanie riešenia ilustračného príkladu pomocou spätného navracania pre poradie premenných V_1, V_3, V_4 a V_2.

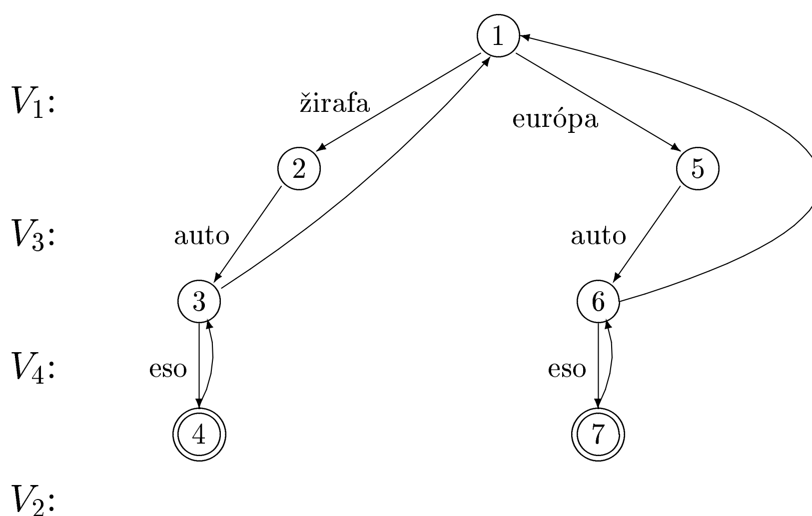
Pretože v tomto prípade úloha nemá riešenie, všetky listové uzly stromu signalizujú zahájenie navracania. V tomto strome možno nájsť inú podobu opakovaného zlyhávania. Prehľadávanie v uzle 4 zlyhalo preto, lebo hodnota "žirafa" nie je konzistentná so žiadnou hodnotou z domény premennej V_4. Algoritmus sa však vrátil k premennej V_3 a snažil sa jej priradiť inú hodnotu. Keďže neriešil príčinu neúspechu v uzle 4, tak následné prehľadávanie zlyhalo z tej istej príčiny (uzol 6). Bola vykonávaná zbytočná práca, pretože algoritmus nedetekoval príčinu neúspechu, a tým pádom nereagoval optimálnym spôsobom.

Takýto prípad opakovaného zlyhania z tej istej príčiny sa snaží riešiť napr. *spätné navracanie riadené závislosťami* [123], ktoré pri výskyte konfliktu v priradení hodnôt premenným sa pokúša meniť premennú, ktorá je

príčinou tohto konfliktu. Podobný prístup reprezentuje Gaschnigov algoritmus *spätného skoku* [61].

Tento algoritmus zaručuje, že strom prehľadávania nebude väčší ako ten, ktorý je dosiahnutý pomocou spätného navracania. Jeho podstatou je to, že pri detekovaní *slepého konca* (deadend) prehľadávania (po vyčerpaní všetkých možností výberu hodnoty z nejakej domény) sa dokáže vrátiť späť naraz aj o niekoľko úrovní – vykoná spätný skok až priamo k tej premennej, ktorá je príčinou neúspechu. Ak neúspech bol zapríčinený viacerými premennými, tak sa vracia k tej z nich, ktorá je najbližšie (teda ktorá získala svoju momentálnu hodnotu najneskôr).

Ak by algoritmus spätného skoku bol použitý pre riešenie toho istého problému ako bol riešený na obr. 4.8, tak by bol vygenerovaný zjednodušený strom prehľadávania podľa obr. 4.9.



Obr. 4.9: Trasovanie riešenia ilustračného príkladu pomocou algoritmu spätného skoku pre poradie premenných V_1, V_3, V_4 a V_2.

Po neúspechu v uzle 4 a následnom návrate k uzlu 3 bola detekovaná nemožnosť konzistentného priradenia hodnoty premennej V_4. Preto je potrebné realizovať navracanie k jednej z už naviazaných premenných (premenné V_1

a V_3). Pretože premenná V_4 bola v konflikte s priradením V_1 a nie s V_3, tak sa vykoná spätný skok od uzla 3 k uzlu 1. Rovnaký dôvod mal za následok spätný skok od uzla 6 k uzlu 1.

Nevýhodou tohto variantu je to, že ak od premennej V_i sa spätne skočí k premennej V_j a tá už nemá žiadnu ďalšiu hodnotu vo svojej doméne na vyskúšanie, tak návrat k premenným pred premennou V_j už prebieha ako pri spätnom navracaní, pretože premenná V_j nie je momentálne v konflikte so žiadnou z premenných ktoré získali svoju hodnotu skôr ako ona.

Túto nevýhodu sa snaží odstrániť algoritmus spätného skoku riadený konfliktmi [82]. Pre každú premennú je udržiavaná *konfliktná množina* – zoznam tých premenných, s ktorými daná premenná bola v konflikte (pri ich momentálnom spôsobe naviazania) pri priraďovaní hodnôt z jej domény (vo všeobecnosti rôzne hodnoty z jej domény môžu zapríčiňovať konflikt s rôznymi premennými). Tento algoritmus spätného skoku riadený konfliktmi (dependency-directed backjumping – DDBJ) môže mať nasledujúci tvar: [9]

```
define method backjumping( i::<integer> )
   if ( and ( 0 < i, i <= n ) )
      V[i] := next( D[i], V[i] );
      if ( not ( V[i] ) )
         let j = max ( CS[i] );
         CS[j] := union ( CS[j], del( CS[i], j ) );
         for ( k::<integer> from j+1 to i )
            CS[k] := #();
         end for;
         backjumping ( j );
      end if;
      if ( satisfy_all_constraints ( ) )
         backjumping ( i+1 );
      else
         CS[i] := add ( CS[i], conflict_variables ( i ) );
         backjumping ( i );
      end if;
   end if;
end method backjumping;
```

Ak teda po priradení nejakej hodnoty nejakej premennej V_i nastáva porušenie jedného alebo viacerých ohraničení, tak do zoznamu CS_i prislúchajúceho danej premennej sa vložia všetky tie premenné, s ktorými je daná premenná momentálne v konflikte. Pri nutnosti návratu k predchádzajúcim

[9] Znak `#()` označuje prázdny zoznam.

premenným je zoznam usporiadaný – premenné sa v ňom zoradia podľa toho, v akom poradí im boli priradené hodnoty – a algoritmus pokračuje skokom k tej premennej V_j, ktorá získala hodnotu najneskoršie. Zároveň zvyšok konfliktnej množiny CS_i premennej V_i sa pridáva ku konfliktnej množine tej premennej, na ktorú sa skáče, takže informácia o konfliktoch sa nestráca.

Pretože premenná V_i po realizácii spätného skoku nebude mať priradenú žiadnu hodnotu, tak súčasne nebude v konflikte so žiadnou inou premennou a teda zoznam prislúchajúci tejto premennej sa vyprázdni. Po návrate k opätovnému pokusu naviazať premennú V_i na nejakú hodnotu sa zoznam prislúchajúci tejto premennej začne budovať z prípadných neúspechov nanovo.

Ak by bol použitý pre riešenie rovnakého problému, ako bol riešený na obr. 4.8, tak by bol opäť vygenerovaný strom prehľadávania zobrazený na obr. 4.9. Pretože v uzle 4 nastal konflikt (bolo porušené ohraničenie $C^{1,4}$), do zoznamu CS_4 zodpovedajúcemu premennej V_4 bola vložená premenná V_1. Po vyčerpaní všetkých možností ako priradiť hodnotu premennej V_4 (návrat do uzla 3) bolo rozhodnuté o návrate k nejakej predchádzajúcej premennej. Zo zoznamu CS_4 bola vybratá premenná V_1. Opäť došlo k väčšiemu orezaniu priestoru prehľadávania ako v prípade použitia spätného navracania.

Existuje aj iný variant algoritmu spätného skoku, ktorý sa snaží eliminovať nutnosť udržiavať konfliktnú množinu pre každú premennú tým, že používa iba jeden zoznam uzlov, ktoré majú byť skontrolované pri výskyte nasledujúceho slepého konca. Je to algoritmus spätného skoku založený na grafe [36].

Znalosti o závislostiach vyberá zo samotnej siete ohraničení. Kedykoľvek sa počas procesu prehľadávania vyskytne slepý koniec pri pokuse naviazať nejakú premennú V_i, algoritmus sa vracia na najneskôr naviazanú premennú V_j spomedzi tých premenných, ktoré sú spojené v sieti ohraničení s premennou V_i nejakým ohraničením. Ak V_j už nemá viac hodnôt na vyskúšanie, tak sa algoritmus vracia k najposlednejšej premennej V_k spomedzi tých premenných, ktoré v sieti ohraničení súvisia s V_i alebo V_j atď.

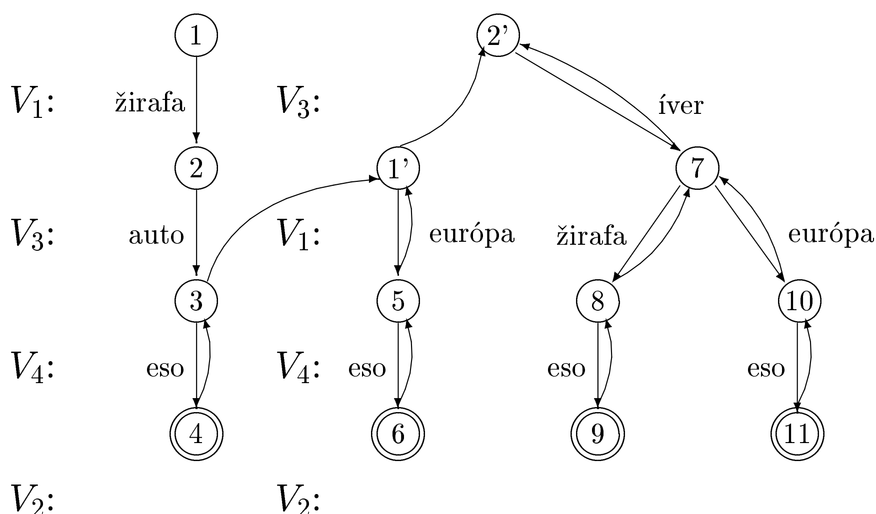
Je to o niečo slabšia verzia ako variant riadený konfliktmi – nevracia sa priamo k príčine neúspechu, ale k možnej príčine neúspechu, čím jeho spätný skok môže byť kratší. Príkladom je opäť problém podľa obr. 4.8. Teraz nebude vygenerovaný strom prehľadávania podľa obr. 4.9, pretože konfliktná premenná V_4 v sieti ohraničení susedí s oboma premennými V_1 a V_3. Výsledkom by v tomto prípade bol strom prehľadávania rovnaký ako na obr. 4.8.

Všetky podoby algoritmu spätného skoku majú tú nevýhodu, že pri navracaní od premennej V_i až k premennej V_j sa spätne uvoľnia všetky pre-

menné, ktorým bola priradená hodnota po naviazaní V_j a pred priradením hodnoty V_i. Tieto premenné budú v ďalších pokusoch o nájdenie riešenia naväzované znova.

Dynamické spätné navracanie [63] toto rieši zmenou poradia niektorých z tých premenných, ktoré už majú priradenú hodnotu. Vráti sa síce k premennej V_j ale tá sa zároveň stane poslednou z minulých premenných a všetky tie premenné ktoré boli naviazané po nej a pred V_i budú presunuté pred V_j. Potom je teda možné zmeniť hodnotu premennej V_j bez modifikácie priradení premenným, ktoré boli naviazané až po nej. To umožní uchovať výsledky predchádzajúcej práce.

Pri použití dynamického spätného navracania pre rovnaký problém ako v predošlom (riešený chronologickým spätným navracaním na obr. 4.8) by bol vygenerovaný strom prehľadávania podľa obr. 4.10 (zároveň to je príklad toho, že použitie zložitejšieho (inteligentnejšieho) variantu voči jednoduchšiemu nie je vždy výhodné).



Obr. 4.10: Trasovanie riešenia ilustračného príkladu pomocou dynamického spätného navracania pre prvotné poradie premenných V_1, V_3, V_4 a V_2.

V tomto prípade vykonanie spätného skoku z uzla 3 nemá za následok stratu hodnoty "auto" pre premennú V_3. Hodnotu stratí iba premenná V_1 (hodnotu "žirafa") a premenné V_1 a V_3 si navzájom vymenia poradie.

4.2.2 Algoritmy s pamäťou

Doplnenie pamäti k prehľadávaciemu algoritmu umožní tomuto algoritmu v rámci riešenia nejakého problému poučiť sa z výsledkov svojej činnosti v skorších etapách prehľadávania, a tieto svoje skúsenosti využiť v neskorších etapách prehľadávania priestoru riešení. Typicky sa jedná o učenie z neúspechov – algoritmus si tieto svoje neúspechy pamätá a to mu umožní nezopakovať tieto chyby ešte raz.

Najjednoduchším z takýchto algoritmov je algoritmus *spätnej kontroly* [68]. Vždy po priradení hodnoty nejakej premennej je toto priradenie kontrolované na konzistenciu s predchádzajúcimi priradeniami. V prípade, že nejaká podmnožina priradených hodnôt porušuje nejaké ohraničenie, tak ešte pred začiatkom navracania sa táto nekonzistentná podmnožina hodnôt zapamätá.

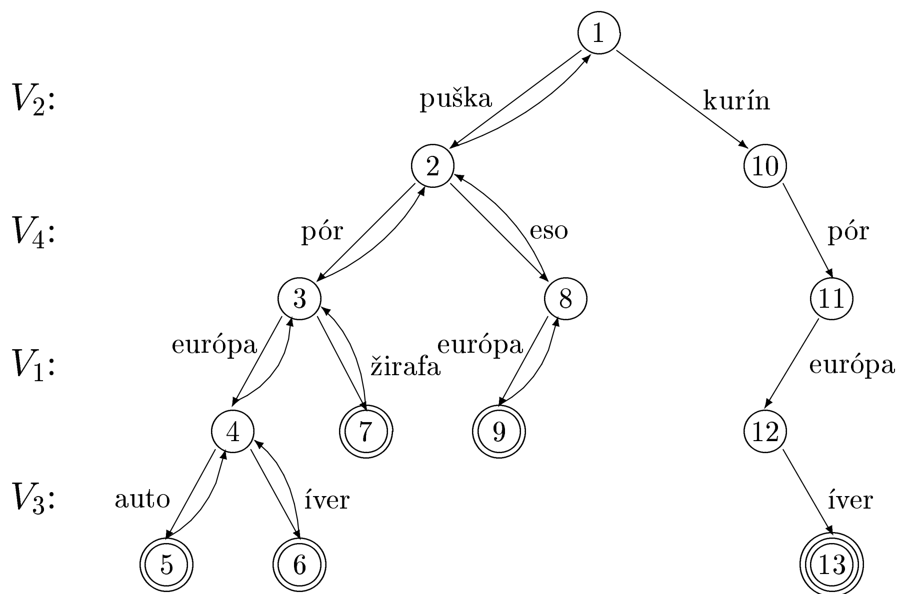
Ak by sa v budúcnosti priradením určitej hodnoty nejakej premennej táto pamätaná kombinácia hodnôt mala opäť vyskytnúť ako súčasť budovaného riešenia, tak k priradeniu nedôjde. Teda vždy pred priradením nejakej hodnoty nejakej premennej sa kontroluje, či rozšírenie aktuálneho riešenia o uvažovanú hodnotu nepovedie k výskytu nejakej kombinácie hodnôt, o ktorej už bolo zistené, že je nekonzistentná.

Ak by algoritmus spätnej kontroly bol využitý pre riešenie príkladu z obr. 2.2 pričom by bolo použité usporiadanie premenných V_2, V_4, V_1 a V_3 a poradie hodnôt v doméne D_1 by bolo opačné, tak by vygenerovaný strom prehľadávania mal tvar podľa obr. 4.11.

Tento strom prehľadávania je menší o dva negenerované uzly než strom, ktorý by bol získaný pomocou chronologického spätného navracania. V uzle 8 nie je pokus priradiť hodnotu "žirafa" premennej V_1. Je to preto, lebo tento pokus by bol neúspešný – táto hodnota spolu s hodnotou "puška" priradenou premennej V_2 by porušovala ohraničenie $C^{1,2}$. A pretože tento fakt bol zistený už v uzle 7, tak v uzle 8 sa tento neúspešný pokus už nezopakuje. Z rovnakého dôvodu v uzle 12 nie je pokus o použite hodnoty "auto" – nevhodnosť tohto pokusu bola zistená v uzle 5.

V princípe spätnú kontrolu je možné použiť na kontrolu ohraničení ľubovoľnej árnosti. Ak sa však uvažujú iba binárne ohraničenia, je potrebné si pamätať iba nekonzistentné dvojice.

Mechanizmus spätnej kontroly sa samostatne nepoužíva, ale zvyčajne sa dopĺňa o mechanizmus eliminácie niektorých redundantných kontrol konzistencie do tvaru algoritmu *spätného značkovania*. Ten vynecháva niektoré kontroly, ktoré boli vykonané v minulosti, boli vtedy úspešné a ak by boli



Obr. 4.11: Strom prehľadávania generovaný algoritmom spätnej kontroly pre ilustračný príklad pre poradie premenných V_2, V_4, V_1 a V_3.

vykonané opäť, tak by boli opäť úspešné [68]. Výsledný strom prehľadávania má rovnaký tvar ako pri spätnej kontrole.

Pre vysvetlenie možno predpokladať, že pokus o priradenie hodnoty nejakej premennej V_i stroskotal, pretože táto premenná nemala vo svojej doméne žiadnu takú hodnotu, ktorá by bola konzistentná s aktuálnym parciálnym riešením. Z toho dôvodu bolo vyvolané navracanie cez premenné $V_{i-1}, \ldots, V_{j+1}$ až k premennej V_j. Tej bola priradená nejaká iná hodnota z jej domény. Premenným $V_{j+1}, \ldots, V_{i-1}$ boli taktiež znovu priradené nejaké hodnoty. Potom pri novom pokuse o priradenie nejakej hodnoty premennej V_i (takej, o ktorej ešte nie je známe podľa algoritmu spätnej kontroly, že by bola nekonzistentná s aktuálnym parciálnym priradením) túto hodnotu pri binárnych ohraničeniach nie je potrebné kontrolovať voči hodnotám tých premenných ktoré svoje hodnoty nadobudli skôr ako V_j – voči tým je hodnota premennej V_i určite konzistentná (keďže tie hodnoty sa nemenili, ale ostali rovnaké aké boli pred začatím navracania od V_i, tak potrebné kontroly už boli realizované

pri prvom pokuse priradiť hodnotu premennej V_i).

Pri použití spätného značkovania na rovnaký príklad ako vyššie by bol opäť získaný strom prehľadávania podľa 4.11. Rozdiel oproti spätnej kontrole je ten, že pri spätnom značkovaní v uzle 9 nie je potrebné kontrolovať hodnotu priradenú premennej V_1 voči hodnote premennej V_2, pretože rovnaká kontrola v uzle 4 potvrdila konzistenciu tejto dvojice hodnôt. V uzle 9 sa skontroluje konzistencia iba dvojice hodnôt ("európa", "eso").

Nevýhodou predchádzajúcich algoritmov je to, že si iba zapamätávajú nekonzistentné kombinácie hodnôt, porušujúce existujúce ohraničenia. Nemajú schopnosť sa učiť – nevedia indukovať ohraničenia a pracovať s nimi rovnakým spôsobom ako s ostatnými explicitnými ohraničeniami (teda vytvárať nové alebo sprísňovať už existujúce explicitné ohraničenia). A práve o toto sa snaží prístup *plytkého* a *hlbokého učenia* [36] [58].

Príležitosť indukovať nejaké ohraničenie sa vyskytne vždy pri detekcii slepého konca. Ak nejaké parciálne riešenie, charakterizované priradením hodnôt premenným $V_1 \ldots V_i$, nie je možné konzistentne rozšíriť o žiadnu hodnotu z domény premennej V_{i+1} tak principiálne je možné vytvoriť resp. upraviť ohraničenie $C^{1,\ldots,i}$ tak, aby daná kombinácia hodnôt nebola dovolená (parciálne riešenie vlastne vytvára konfliktnú množinu). Problémom je to, že by to bolo zbytočné – tá istá kombinácia hodnôt sa vďaka stratégii spätného navracania už v ďalšej fáze prehľadávania nevyskytne. Ak by však nejaká podmnožina nájdenej konfliktnej množiny bola v konflikte s hodnotami v doméne premennej V_{i+1}, tak táto podmnožina sa ešte v ďalšom prehľadávaní môže znovu vyskytnúť a preto má cenu si ju zapamätať vo forme ohraničenia. Cieľom potom je vybrať z danej konfliktnej množiny všetky minimálne konfliktné podmnožiny a tie si pamätať. [10].

Ak nejaká konfliktná množina je v konflikte s nejakou premennou V_{i+1} (nedá sa konzistentne rozšíriť priradením žiadnej hodnoty tejto premennej), tak je z nej možné získať nejakú konfliktnú podmnožinu vypustením priradenia hodnoty nejakej premennej V_k, ak toto priradenie je irelevantné voči V_{i+1}. Toto je splnené napr.:

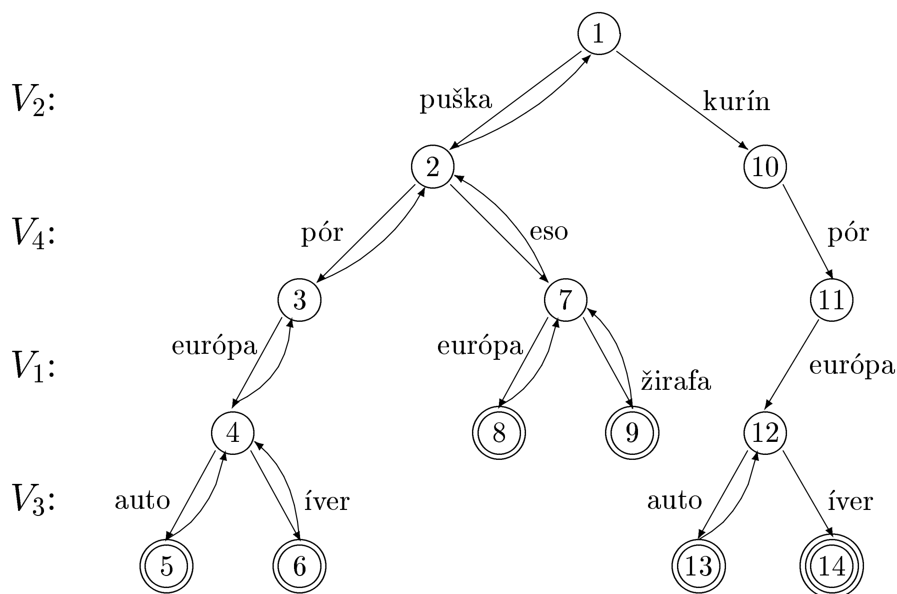
1. ak žiadne ohraničenie nie je definované súčasne nad V_k a V_{i+1}
2. ak priradenie premennej V_k je konzistentné so všetkými možnými hodnotami z domény premennej V_{i+1}.

[10] Je možný aj selektívny výber – pamätať sa budú napr. iba konfliktné podmnožiny určitej veľkosti (napr. binárne) alebo iba tie, ktoré povedú na sprísnenie už existujúceho ohraničenia a nevyžadujú vytvorenie nového ohraničenia, čím by došlo k zmene topológie siete ohraničení.

Takýmto spôsobom sa hľadajú konfliktné podmnožiny pri plytkom učení. Ak sa použije iba prvý bod, tak sa jedná o plytké učenie založené na grafe. Sieť ohraničení poskytuje jednoduchý návod ako identifikovať navzájom irelevantné premenné. Nevýhodou je to, že samotná topológia siete negarantuje identifikáciu všetkých irelevantných priradení v konfliktnej množine.

Na rozdiel od plytkého učenia hlboká varianta postupne analyzuje všetky možné podmnožiny konfliktnej množiny a vyhľadáva všetky minimálne konfliktné podmnožiny.

Pri použití princípu plytkého učenia spolu so spätným navracaním na rovnaký príklad ako je riešený na obr. 4.11 by bol vygenerovaný strom prehľadávania podľa obr. 4.12.



Obr. 4.12: Strom prehľadávania generovaný algoritmom plytkého učenia pre ilustračný príklad pre poradie premenných V_2, V_4, V_1 a V_3.

Po návrate do uzla 4 z uzla 6 bol vlastne detekovaný slepý koniec s tým, že kombinácia hodnôt ("európa", "puška", "pór") je v konflikte s premennou V_3, pretože sa nedá konzistentne rozšíriť ani o hodnotu "auto" ani o hodnotu "íver". Jedinou irelevantnou hodnotou v tejto kombinácii je "európa", pre-

tože neexistuje žiadne ohraničenie definované nad V_1 a V_3. Následkom učenia bude vytvorené nové ohraničenie $C^{2,4}$, ktoré nebude povoľovať kombináciu hodnôt ("puška", "pór"). Toto bude dôvodom toho, že po návrate do uzla 3 sa nevyskytne neúspešný pokus priradiť premennej V_1 hodnotu "žirafa". Podobne po detekcii slepého konca v uzle 7 sa ohraničenie $C^{2,4}$ sprísni tak, aby nepovoľovalo kombináciu ("puška", "eso").

Ak by však bolo použité hlboké učenie namiesto plytkého, tak po detekcii slepého konca v uzle 4 by z konfliktnej množiny ("európa", "puška", "pór") bola vybraná minimálna konfliktná podmnožina ("puška") a táto hodnota by bola zakázaná unárnym ohraničením C^2. Následkom toho by sa negenerovali ani uzly 7, 8 a 9 a navracanie by išlo z uzla 4 až späť k uzlu 1.

Ešte ďalej ide prístup nazývaný *zaznamenávanie nevhodností* [116] [117]. Nevhodnosť je vlastne objekt, zahŕňajúci nekonzistentnú kombináciu hodnôt a zároveň aj dôvod tejto nekonzistencie. Pritom ako dôvod nekonzistencie danej kombinácie hodnôt slúži nejaká kombinácia ohraničení. Tak napr. v uzle 5 podľa obr. 4.12 bolo zistené, že kombinácia hodnôt ("európa", "puška", "auto", "pór") nie je konzistentná, pretože porušuje ohraničenia $C^{2,3}$ a $C^{3,4}$. Na základe toho možno zostrojiť nevhodnosť

$$((\text{európa, puška, auto, pór}), (C^{2,3})).$$

To nové, čo prináša tento prístup, je to, že s nevhodnosťami (reprezentujúcimi nejaké ohraničenia) je možné určitým spôsobom narábať tak, že vznikajú nevhodnosti nové (teda je možné vlastne z existujúcich ohraničení odvádzať ohraničenia nové). Pritom sú možné dve operácie:

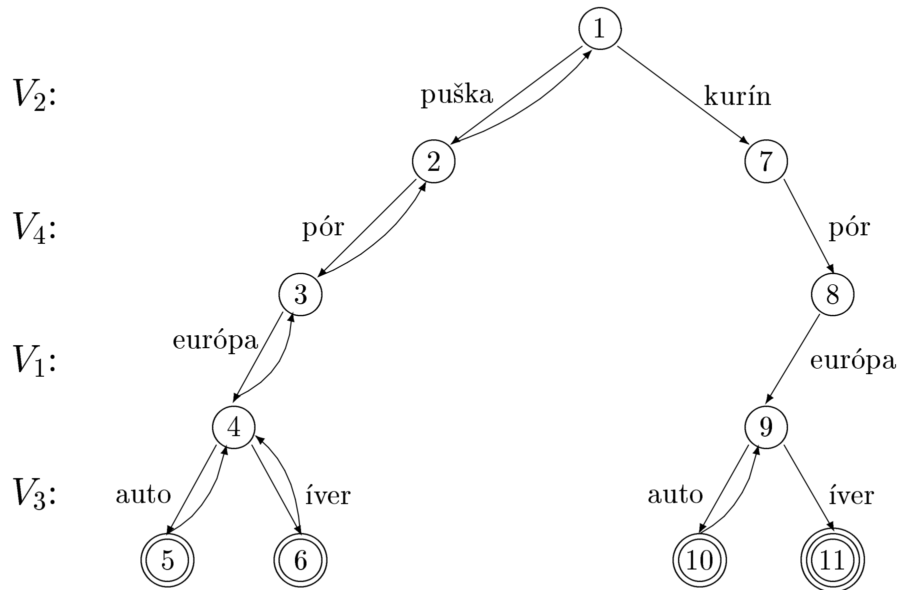
- spájanie

  Je možné spojiť viacero nevhodností do jednej, ak ich kombinácie hodnôt sú také, že obsahujú všetky možné hodnoty z domény nejakej premennej V_i a súčasne sa navzájom líšia iba v hodnote priradenej tejto premennej. Vo výslednej nevhodnosti kombinácia hodnôt nebude obsahovať hodnotu priradenú premennej V_i a kombinácia ohraničení bude vytvorená zjednotením kombinácií ohraničení príslušných spájaných nevhodností.

- projekcia

  Z nevhodnosti je možné vytvoriť novú vypustením tej hodnoty z kombinácie hodnôt, ktorá je priradená takej premennej, nad ktorou nie je definované žiadne ohraničenie z kombinácie ohraničení danej nevhodnosti.

Pri použití tohto princípu spolu so spätným navracaním na rovnaký príklad ako vyššie by bol vygenerovaný strom prehľadávania podľa obr. 4.13.



Obr. 4.13: Strom prehľadávania generovaný algoritmom zaznamenávania nevhodností pre ilustračný príklad pre poradie premenných V_2, V_4, V_1 a V_3.

V uzloch 5 a 6 bola detekovaná nekonzistencia aktuálneho riešenia a preto bolo možné vygenerovať tri základné nevhodnosti:

uzol 5

((európa, puška, auto, pór), ($C^{2,3}$))
((európa, puška, auto, pór), ($C^{3,4}$))

uzol 6

((európa, puška, íver, pór), ($C^{2,3}$))

Dve nevhodnosti je možné spojiť do jednej, pričom to je možné urobiť dvoma spôsobmi (vždy pre jednu z tých ktoré vznikli v uzle 5). Výsledkom sú dve nové nevhodnosti, ktoré už povoľujú ľubovoľnú hodnotu priradenú premennej V_3:

((európa, puška, pór), ($C^{2,3}$))
((európa, puška, pór), ($C^{3,4}$, $C^{2,3}$)).

Na obidve je možné aplikovať operáciu projekcie. V prvom prípade v kombinácii hodnôt sa vyskytuje priradenie premenných V_1 a V_4, hoci nad týmito premennými nie je definované žiadne ohraničenie, ktoré by sa súčasne nachádzalo aj v kombinácii ohraničení danej nevhodnosti. Preto tieto dve priradenia je možné z danej kombinácie hodnôt vypustiť. V druhom prípade je možné vypustiť iba priradenie premennej V_1. Pomocou projekcie sa získajú

((puška), ($C^{2,3}$))
((puška, pór), ($C^{3,4}$, $C^{2,3}$)).

Prvé z nich je vlastne ekvivalentné úprave unárneho ohraničenia C^2, pri ktorom sa zakáže hodnota "puška". A práve vďaka tejto úprave v uzle 3 nie je vyskúšaná hodnota "žirafa" pre premennú V_1 a v uzle 2 zase nie je pokus priradiť hodnotu "eso" premennej V_4.

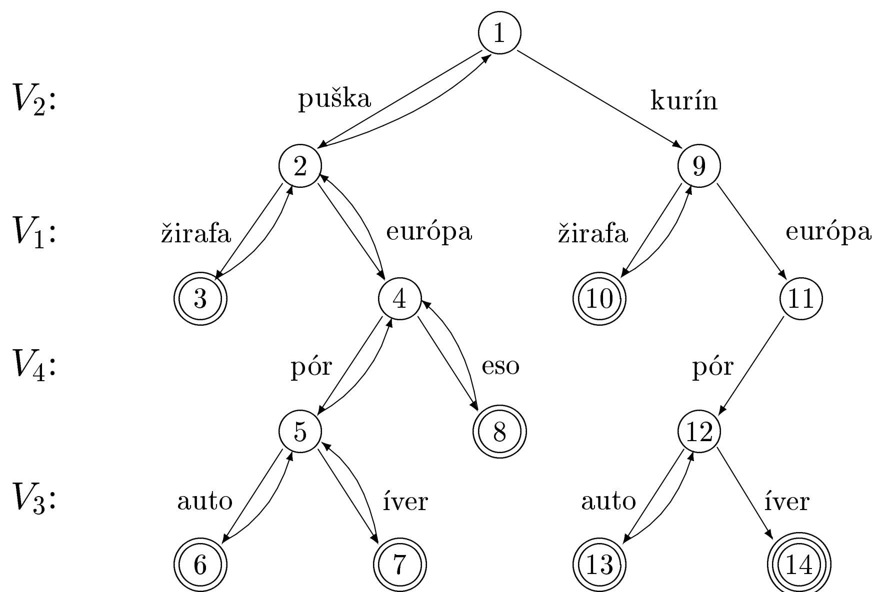
4.2.3 Usporiadanie premenných a ich domén

Proces hľadania riešenia prehľadávaním priestoru riešení závisí nielen od použitého algoritmu prehľadávania, ale je ovplyvnený aj tým, v akom poradí sa daný algoritmus pokúša priradiť hodnoty jednotlivým premenným a v akom poradí tieto hodnoty vyberá z domén týchto premenných. Vhodnou voľbou poradia je možné ovplyvniť tvar prehľadávacieho stromu a tým aj efektívnosť použitia daného algoritmu.

Napríklad použitie spätného navracania pre ilustračný príklad z obr. 2.2 vytvorilo strom prehľadávania so šestnástimi uzlami a dvoma slepými koncami (obr. 4.6). Ak by sa však napr. vymenilo navzájom poradie premenných V_1 a V_4 (a teda celkové poradie premenných by bolo V_2, V_1, V_4 a V_3) a opäť by sa použil algoritmus spätného navracania, výsledný strom prehľadávania by mal iba 14 uzlov a jeden slepý koniec. Jeho tvar je na obr. 4.14.

Z obrázku je zrejmé, že ak by napríklad súčasne v doménach D_1 a D_3 bolo použité opačné poradie hodnôt, tak výsledný strom prehľadávania by bol menší o dva uzly (uzly č. 10 a 13).

V predchádzajúcich kapitolách v ilustračných príkladoch aplikácie prehľadávacích algoritmov bolo vždy uvedené poradie, v akom sa jednotlivým premenným priraďujú hodnoty (poradie hodnôt v doménach sa väčšinou



Obr. 4.14: Strom prehľadávania generovaný algoritmom spätného navracania pre ilustračný príklad pre poradie premenných V_2, V_1, V_4 a V_3.

nemenilo)[11]. Pri riešení nejakého konkrétneho problému je však cieľom vyriešiť tento problém čo najskôr – a teda je vhodné voliť také poradie premenných a hodnôt, ktoré umožňuje čo najrýchlejšie nájdenie hľadaného riešenia (vo všeobecnosti tomu zodpovedá čo najmenší strom prehľadávania).

Veľmi často je možné v literatúre nájsť intuitívny postup, odporúčajúci inštalovať "obtiažnejšie" premenné najprv, čím sa vo všeobecnosti je možné vyhnúť budovaniu parciálnych riešení, ktoré nebude možné dobudovať. Pri takomto výbere ak sa systém dostane do slepej uličky, tak má šancu na to prísť skôr. To je "fail-first" princíp [68] zaručujúci, že každá vetva stromu prehľadávania, ktorá nevedie k riešeniu, bude orezaná čo najskôr a teda celkovo sa bude veľkosť stromu prehľadávania minimalizovať.

Podobne pre výber hodnôt sa odporúča vyberať najmenej ohraničujúce hodnoty – také, o ktorých sa predpokladá, že vystupujú v čo najväčšom počte

[11] Toto poradie bolo volené účelovo, kde účelom bolo získať taký tvar stromu prehľadávania, ktorý by čo najlepšie vystihoval tú-ktorú popisovanú vlastnosť.

riešení, kompatibilných s daným stavom prehľadávania. Takýmto spôsobom ostáva vlastne čo možno najväčší počet možností pre budúce priradenie ostatným voľným premenným. Tento postup zosobňuje "succeed-first" princíp, ktorý volí takú hodnotu, ktorá najpravdepodobnejšie vedie k riešeniu a tým minimalizuje riziko, že bude potrebné navracanie kvôli zmene tejto hodnoty. V praxi sa hodnota, ktorá najpravdepodobnejšie vedie k riešeniu, zamieňa za hodnotu, ktorá najmenej pravdepodobne vedie ku konfliktu.

Na základe mnohých publikovaných porovnaní sa zaužívala predstava, že voľba využiť "fail-first" princíp pre usporiadanie premenných a "succeed-first" princíp pre usporiadanie hodnôt nie je zlou voľbou. Zvyčajne to funguje, ale neplatí to všeobecne – existujú príklady, ktoré protirečia tomuto všeobecnému výberu [121] [122].

V nasledujúcich dvoch kapitolách budú prezentované niektoré všeobecné heuristiky pre výber premenných a hodnôt z domén týchto premenných. Tieto všeobecné heuristiky sú však niekedy málo účinné, lebo neberú do úvahy špeciálne charakteristiky riešených úloh, a preto v praktických aplikáciách môžu byť nahrádzané heuristikami, špeciálne orientovanými na konkrétny typ riešenej úlohy (napr. prideľovanie zdrojov [113]).

4.2.3.1 Usporiadanie premenných

Voľba poradia premenných môže byť dvojakého typu:

- statická
- dynamická.

Pri statickom zotriedení sa určí poradie premenných ešte pred začiatkom priraďovania hodnôt premenným. Toto zotriedenie sa potom používa počas celého hľadania riešenia. Naproti tomu dynamické zotriedenie sa určuje vždy nanovo v okamihu, keď je potrebné rozhodnúť, ktorej premennej sa bude priraďovať hodnota[12]. V tomto prípade sa jedná o zoradenie iba tých premenných, ktorým ešte nebola priradená žiadna hodnota a vlastne nie je potrebné určiť ich celkové usporiadanie ale iba to, ktorá premenná bude na začiatku tohto usporiadania.

[12] Aby dynamická voľba mala zmysel, použitá metóda usporiadania musí mať "dopredný charakter" – musí byť založená na takej charakteristike premenných, ktorá je ovplyvnená aktuálnym priradením hodnôt ostatným premenným. Rôzne priradenia hodnôt nejakej skupine premenných musia spôsobovať rôzne hodnoty tejto charakteristiky pre dosiaľ voľné premenné.

Vo všeobecnosti statické zotriedenia sú výpočtovo menej náročné ako dynamické, pretože tie je potrebné opakovať viacnásobne (čím je väčší strom prehľadávania, tým viackrát musia byť opakovane určované). Na druhej strane však dynamické triedenia umožňujú použiť rôzne poradia premenných v rôznych častiach stromu prehľadávania, čím ich účinok na znižovanie rozmernosti stromu môže byť väčší.

Vo všeobecnosti pre menšie a stredné problémy sú výhodnejšie statické metódy, nevyžadujúce veľa počítania. Pre veľké problémy sa oplatí investovať do dynamických metód [113].

Metódy statického zotrieďovania premenných sú viazané na sieť ohraničení. Najčastejšie v literatúre uvádzané heuristické metódy tohto typu sú [40]:

- minimálna šírka

  Vzostupné usporiadanie podľa počtu tých susediacich uzlov, ktoré zodpovedajú premenným bez priradenej hodnoty. Ako nasledujúci sa vyberá vždy uzol s minimálnym stupňom v podgrafe voľných premenných.

- maximálny stupeň

  Zostupné usporiadanie podľa počtu hrán, ktorými je daný uzol spojený s inými uzlami. Vyberá sa vždy tá premenná, ktorá zodpovedá uzlu s maximálnym stupňom v sieti ohraničení.

- maximálna kardinalita

  Prvá premenná sa volí ľubovoľne. Ostatné sa zoradia zostupne podľa počtu tých susediacich uzlov, ktoré už majú priradenú hodnotu. Teda na každom stupni (okrem prvého) sa vyberá tá premenná, ktorá je spojená s čo najväčším počtom už naviazaných premenných.

- prehľadávanie do hĺbky

  Prehľadávaním do hĺbky (pozdĺž hrán) sa vygeneruje cesta, obsahujúca všetky uzly.

Uvedené metódy zotriedenia nie sú úplné – existujú situácie, keď podľa nich nie je možné rozhodnúť jednoznačne o poradí premenných (napr. ktorým uzlom začať pri prehľadávaní do hĺbky alebo pri rovnakom počte susediacich

uzlov daného typu či pri rovnakom počte hrán). Vtedy je možné rozhodnúť náhodne[13] alebo použiť ako pomocné kritérium nejakú inú z uvedených metód zotriedenia. Metóda usporiadania teda nezaručuje vygenerovanie iba jedného poradia premenných.

Z popisu týchto metód by sa mohlo zdať, že niektoré z nich (napr. minimálna šírka) majú dynamický charakter – zohľadňujú uzly siete ohraničení, ktoré už majú alebo ešte nemajú priradenú hodnotu. Jedná sa však o simulované priraďovanie, pri ktorom sa nezohľadňuje, aká konkrétna hodnota bola priradená tej-ktorej premennej.

Uvažujme sieť ohraničení podľa obr. 2.2. Pre metódu minimálnej šírky všetky premenné majú stupeň rovnaký (2). Ak by napríklad bola vybratá premenná V_2, tak stupeň premenných V_1 a V_3 klesol na 1. Pretože stupeň premennej V_4 sa nezmenil, nemôže byť vybratá. Ak teraz bude vyberaná premenná V_1, tak stupeň oboch ostávajúcich premenných je rovnaký a je možné vybrať ľubovoľnú z nich. Pomocou tejto metódy teda možno získať napr. usporiadanie V_2, V_1, V_4 a V_3 (strom prehľadávania zodpovedajúci tomuto usporiadaniu je na obr. 4.14), ale nie je možné získať napr. poradie V_2, V_4, V_1 a V_3.

Pre metódu maximálnej kardinality by bolo možné ako prvú premennú zvoliť V_2 (ľubovoľná voľba prvej premennej to umožňuje). Potom uzol reprezentujúci premennú V_4 nesusedí so žiadnym uzlom reprezentujúcim viazanú premennú a nemôže byť vybraný ako druhý, pretože kardinalita uzlov V_1 a V_3 je vyššia ako jeho kardinalita. Použitím tejto metódy by bolo tiež možné získať poradie premenných použité na obr. 4.14.

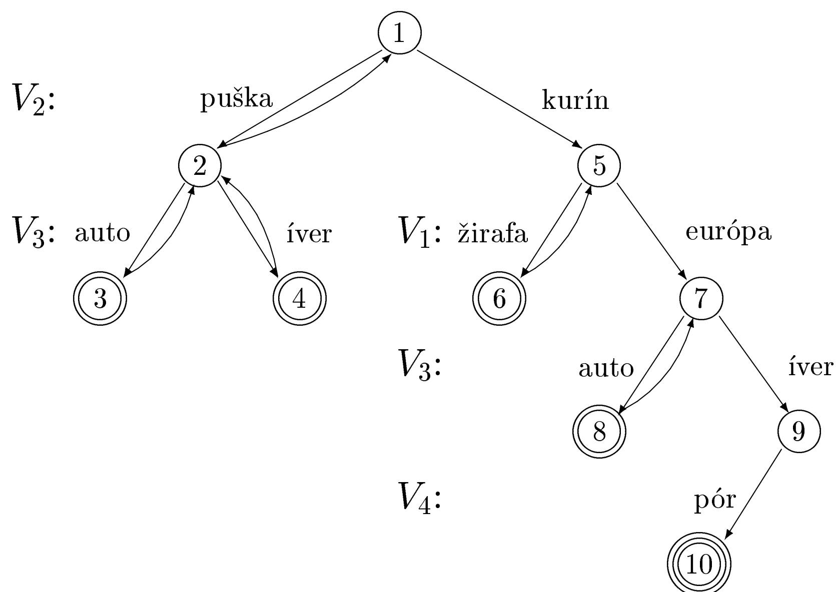
Jednotlivé metódy môžu viesť na rovnaké usporiadanie ako vo vyššie uvedenom prípade. To však neznamená, že by tieto metódy boli ekvivalentné – pre každú dvojicu metód existuje taký tvar siete ohraničení, pri ktorom produkujú rôzne usporiadania.

Ak by sme chceli použiť metódu prehľadávania do hĺbky, mohli by sme ako počiatočný uzol zvoliť V_2. Z tohto uzla sa dá prejsť iba do uzla V_1 alebo V_3. Nech sa zvolí ktorýkoľvek z nich, treťou v poradí bude premenná V_4. Touto metódou nie je možné získať usporiadanie, v ktorom by hneď za sebou nasledovali premenné V_2 a V_4 alebo premenné V_1 a V_3.

Na rozdiel od statických metód, dynamické metódy zotrieďovania zohľadňujú aj konkrétne hodnoty, ktoré už sú priradené premenným. Jednou z

[13] Náhodnosť v riešení nerozhodných situácií je použitá v [64], kde je stanovený prah na počet kontrol. Ak pri riešení úlohy sa dosiahne tento prah, tak riešenie je odštartované odznovu, pričom sa použije iná inicializácia generátora náhodných čísel ako v predchádzajúcom behu.

dynamických metód je *dynamická zmena prehľadávania* [40] (niekedy označovaná ako *minimum ostávajúcich hodnôt* [9]). Táto metóda vyberá vždy tú premennú, ktorá má vo svojej doméne najmenší počet hodnôt konzistentných s aktuálnym priradením. Pre použitý ilustračný príklad je jeden z možných stromov prehľadávania generovaných touto metódou na obr. 4.15.



Obr. 4.15: Strom prehľadávania generovaný algoritmom spätného navracania pre ilustračný príklad pre dynamicky určované poradie premenných.

Na začiatku všetky premenné majú rovnakú šancu na výber, pretože každá z nich má vo svojej doméne dve hodnoty. Po priradení napr. hodnoty "puška" premennej V_2 je vyberaná premenná V_3 – ako jediná z neviazaných premenných nemá v doméne žiadnu hodnotu konzistentnú s hodnotu premennej V_2. Po návrate k premennej V_2 a priradení hodnoty "kurín" tejto premennej je možné voliť spomedzi premenných V_1 a V_3 – obe majú jednu hodnotu zo svojej domény konzistentnú s aktuálnym parciálnym priradením. Po naviazaní premennej V_1 všetky zvyšné premenné majú po jednej konzistentnej hodnote vo svojej doméne a opäť je možno medzi nimi voliť náhodne.

V práci [62] je uvedených niekoľko ďalších heuristík pre dynamický výber premenných. Tieto heuristiky sú založené na snahe odhadovať ohraničenosť subproblému, tvoreného voľnými premennými. Snažia sa o to, aby sa vybrala taká premenná, po naviazaní ktorej subproblém, tvorený zvyšnými premennými, je ohraničený čo najmenej.

Jednou takou heuristikou je *E(N)*, ktorá sa snaží maximalizovať očakávaný počet riešení. Teda snaží sa zároveň o čo najväčší subproblém (s najväčším priestorom prehľadávania) a o čo najväčšiu hustotu riešení (teda čo najmenšiu ohraničenosť). Jej cieľom je vybrať takú premennú V_i, ktorej výber maximalizuje vzťah[14]

$$\left(\prod_{V_k \in V^{free} - V_i} \parallel D_k \parallel \right) \left(\prod_{c \in (C^{free} - C_i^{all})} (1 - p_c) \right) \tag{4.2}$$

teda vzťah zvažujúci veľkosti domén voľných premenných (bez ohodnocovanej premennej) a prísnosť ohraničení týkajúcich sa iba voľných premenných (s výnimkou práve ohodnocovanej premennej).

To je vlastne identické so snahou vybrať tú premennú, ktorá najmenej prispieva k ohraničenosti subproblému tvorenému voľnými premennými (po naviazaní vybranej premennej) a teda minimalizuje

$$\parallel D_i \parallel \prod_{c \in (C^{free} \cap C_i^{all})} (1 - p_c) \tag{4.3}$$

kde $\parallel D_i \parallel$ je veľkosť domény D_i premennej V_i a $(C^{free} \cap C_i^{all})$ sú všetky ohraničenia nad voľnými premennými zahŕňajúce aj V_i a p_c je prísnosť ohraničenia (udávajúca akú časť možných kombinácií hodnôt premenných vylučuje). Pri rovnakej prísnosti všetkých ohraničení sa táto heuristika redukuje na už spomínanú dynamickú zmenu prehľadávania.

Zaujímavou myšlienkou je použitie viacerých heuristík súčasne nielen v tom zmysle, že jedna je primárna a ak tá nevie rozhodnúť, tak sa použije sekundárna, ale ako kombinácia viacerých heuristík do jednej výslednej. Pri dom/deg usporiadaní [17] sa vyberá tá premenná, ktorá má najmenší pomer veľkosti domény a stupňa (jedna časť uvažuje domény hodnôt premenných a druhá zase štruktúru siete ohraničení).

Uvedené zotriedenia je možné používať v súčinnosti s ľubovoľným prehľadávacím algoritmom. Nie vždy to je však vhodné, pretože spôsob zotriedenia

[14] C^{free} označuje všetky ohraničenia definované nad voľnými (dosiaľ neviazanými) premennými a C_i^{all} zase všetky ohraničenia, ktoré sú definované nad premennou V_i.

premenných môže kolidovať s činnosťou algoritmu. Tak napr. použitie dynamickej zmeny prehľadávania v spojení s algoritmom spätného skoku je zbytočné, pretože pri navracaní sa bude tento vracať vždy na najbližšiu úroveň (bude sa chovať ako obyčajné spätné navracanie, ibaže bude mať väčšiu réžiu) [9]. Inou možnosťou je návrh zotriedenia určený pre použitie s nejakým konkrétnym prehľadávacím algoritmom. Takéto zotriedenie má možnosť vychádzať z vlastností tohto konkrétneho algoritmu a využívať ich. Príkladom je usporiadanie premenných pre algoritmus dopřednej kontroly, založené na minimalizovaní očakávaného počtu uzlov a očakávaného počtu kontrol [100].

4.2.3.2 Usporiadanie domén

Z pravej vetvy stromu na obr. 4.15 je zrejmá úloha usporiadania hodnôt v doménach (podľa obrázku je zrejmé nevhodné usporiadanie domén D_1 a D_3). V tomto prípade hrajú dominantnú úlohu dynamické heuristické metódy. Niektoré z nich sú:

- lepkavé hodnoty

  preferovanie hodnôt, ktoré sa v minulosti ukázali úspešné

- dopredné heuristiky

  všetky hodnoty selektovanej premennej sú porovnávané voči doménam budúcich[15] premenných podľa nejakého kritéria

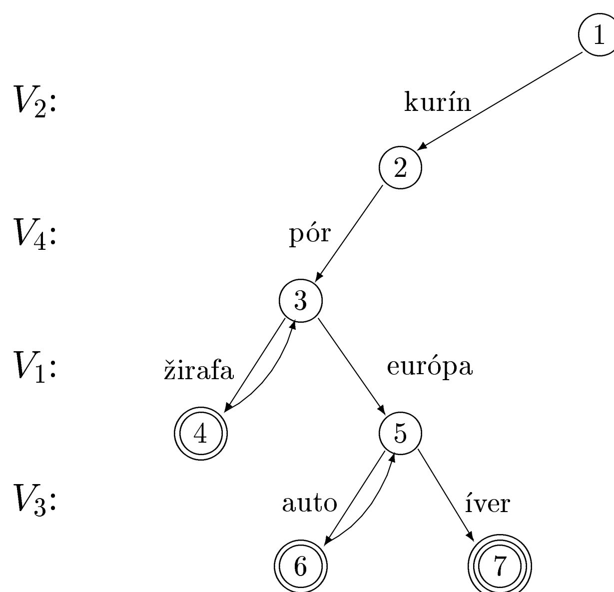
- minimalizácia zložitosti

  preferencia hodnoty, ktorá vedie na jednoduchší problém.

Použitie lepkavých hodnôt [57] možno ilustrovať na strome prehľadávania uvedeného na obr. 4.6. Pri prvom obsadzovaní premennej V_1 v uzle 3 bola hodnota "európa" úspešná (teda aspoň dočasne). Preto pri druhom pokuse o priradenie hodnoty premennej V_1 v uzle 8 bude najprv vyskúšaná hodnota "európa" a až potom hodnota "žirafa". Keďže teraz ani jedna hodnota nebola úspešná, tak toto druhé priraďovanie nebude mať vplyv na poradie hodnôt v doméne D_1. Pri treťom pokuse o priradenie hodnoty premennej V_1 sa opäť začne s hodnotou "európa" – a dôsledkom bude, že uzol 13 nebude generovaný.

[15] Sú to voľné premenné, ktorým v ďalšom postupe prehľadávania bude priradená nejaká hodnota z ich domény.

Príkladom doprednej heuristiky je maximalizácia počtu budúcich možností [83], ktorá maximalizuje počet možností pre budúce priraďovanie hodnôt. Pri tomto spôsobe výberu je preferovaná taká hodnota, po priradení ktorej počet hodnôt v doménach susedných uzlov, ktoré sa stanú nekonzistentnými s aktuálnym priradením, bude minimálny. Ak v príklade na obr. 2.2 sa ako prvej bude priraďovať hodnota premennej V_2, tak sa bude preferovať priradenie hodnoty "kurín", ktorá spôsobí nekonzistenciu dvoch hodnôt ("žirafa" v D_1 a "auto" v D_3), zatiaľ čo použitie hodnoty "puška" by negatívne ovplyvnilo tri hodnoty ("žirafa", "íver" a "auto"). Pri výbere hodnoty pre nasledujúcu premennú V_4 je voľba tiež jednoznačná. Pri obsadení tretej premennej V_1 obe hodnoty sú rovnako možné (neovplyvnia žiadnu hodnotu z D_3). Pre priradenie hodnoty poslednej premennej nie je možné použiť tento spôsob zotriedenia hodnôt a teda výber môže byť náhodný. Ukážka jedného možného stromu prehľadávania je na obr. 4.16.



Obr. 4.16: Strom prehľadávania generovaný algoritmom spätného navracania pre ilustračný problém pre dynamický výber hodnôt z domén.

Iným príkladom doprednej heuristiky je preferencia takej hodnoty, ktorá vytvára najväčšiu minimálnu doménu (z domén voľných premenných) [59] – vychádza z intuície, že je pravdepodobnejšie, že subproblém má riešenie, ak nemá premenné s malými doménami.

Pri metóde minimalizácie zložitosti musí byť daný nejaký spôsob ako určiť zložitosť problému. V [83] je uvedený postup, pri ktorom sieť ohraničení je modifikovaná na stromovú štruktúru vypustením minimálneho počtu hrán. Pre tento jednoduchší problém sa zistí, koľko existuje riešení, obsahujúcich danú skúmanú hodnotu, a tento počet sa použije ako miera zložitosti problému (čím viac riešení, tým je problém jednoduchší)[16].

Sieť ohraničení na obr. 2.2 možno redukovať na stromový tvar vypustením jednej hrany. Existujú štyri také možnosti, ktoré principiálne môžu viesť k rôznym výsledkom. Preto je potrebné nejaké rozhodnutie, ktorá redukcia sa použije. Je možné použiť aj všetky redukcie a o výsledku rozhodnúť hlasovaním.

Ak opäť sa ako prvej bude priraďovať hodnota premennej V_2, tak je možné pre redukovaný problém zistiť nasledujúce počty riešení:

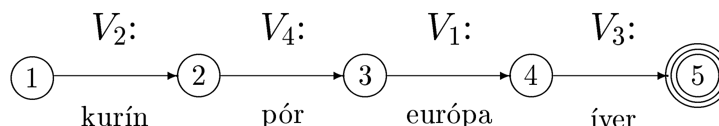
vypustená hrana	priradené hodnoty	počty riešení	
		puška	*kurín*
(V_1,V_2)		0	1
(V_2,V_3)		1	1
(V_3,V_4)		0	1
(V_1,V_4)		0	1

Z uvedeného je zrejmá preferencia hodnoty "kurín". Ak by sa uvažovala iba jedna redukcia na stromový tvar siete ohraničení, tak pri vypustení hrany (V_2,V_3) by mohla byť selektovaná aj hodnota "puška". Po naviazaní prvej premennej sa to isté zopakuje pre druhú premennú. Ak je ňou V_4, tak sa získa (pričom sa uvažuje aj už pridelená hodnota "kurín"):

vypustená hrana	priradené hodnoty	počty riešení	
		eso	*pór*
(V_1,V_2)	*kurín*	0	1
(V_2,V_3)	*kurín*	0	1
(V_3,V_4)	*kurín*	0	1
(V_1,V_4)	*kurín*	0	1

[16] V [41] sa pre tento účel používa minimálna kostra siete ohraničení, kde ohodnotenie každej hrany je dané počtom dvojíc hodnôt príslušných premenných, ktoré sú kompatibilné s ohraničením reprezentovaným danou hranou.

Rovnakým spôsobom je možné zotriediť dynamicky hodnoty v každej doméne pred výberom nejakej hodnoty z tejto domény. Výsledný strom prehľadávania je uvedený na obr. 4.17.



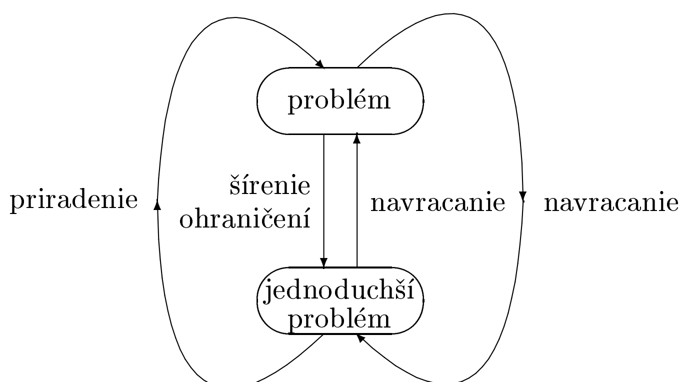
Obr. 4.17: Trasovanie riešenia ilustračného príkladu pomocou spätného navracania pre poradie premenných V_2, V_4, V_1 a V_3.

4.3 Kombinované algoritmy

Redukčné algoritmy zabezpečením prísnej k-konzistencie (kde k je nejaká postačujúca hodnota menšia alebo rovná počtu premenných) umožňujú pohodlne nájsť riešenie. Bohužiaľ ich výpočtová náročnosť zvyčajne presahuje možnosti zdrojov, ktoré sú k dispozícii pre riešenie nejakej konkrétnej úlohy. Na druhej strane prehľadávacie algoritmy sú síce pomerne jednoduché, ale pri "nevhodnej" konfigurácii problému dokážu práve vďaka skúšaniu rozličných nekonzistentných hodnôt vygenerovať obrovské stromy prehľadávania.

Obe triedy uvedených algoritmov majú teda svoje výhody aj nevýhody. A práve ich výhody sa snažia využiť kombinované algoritmy, ktoré spájajú princíp jednoduchého prehľadávania priestoru riešení s metódou zjednodušovania riešeného problému zabezpečovaním nejakého stupňa redukcie ohraničení a domén premenných. Všeobecná schéma takéhoto kombinovaného algoritmu je uvedená na obr. 4.18.

Na aktuálny problém sa aplikuje nejaký redukčný algoritmus. Môže to byť algoritmus pre zabezpečenie plnej (prísnej) k-konzistencie ($1 \leq k \leq n$), nejaký algoritmus, ktorý zabezpečí k-konzistenciu iba čiastočne (negarantuje jej zabezpečenie v celej sieti ohraničení), alebo nejaký substitučný algoritmus. Takýmto spôsobom je vo všeobecnosti pôvodný problém transformovaný na problém jednoduchší. Na tento jednoduchší problém je aplikovaný jeden prehľadávací krok, majúci za následok priradenie hodnoty jednej neviazanej premennej. Po tomto priradení sa problém opäť o niečo zjednoduší ďalšou redukciou atď.



Obr. 4.18: Schéma spojenia prehľadávacieho algoritmu so zabezpečovaním určitého stupňa konzistencie.

Prehľadávacie a redukčné algoritmy je možné rôzne kombinovať. No nie vždy každá kombinácia musí byť účelná. Ak sa kombinuje štandardné chronologické spätné navracanie s redukčnou zložkou, tak výsledok býva pozitívny. Ak sa však uvažujú inteligentné formy prehľadávania, tak ich spojenie s nejakým konkrétnym spôsobom redukcie nemusí vždy priniesť efekt, ale naopak môže celý proces hľadania riešenia spomaliť [17], [111].

Pretože strom prehľadávania bol zobrazovaný takým spôsobom, že priraďovanie hodnôt premenným bolo reprezentované hranami a uzly predstavovali stavy pred resp. po priradení hodnôt, tak jeho rozšírenie pre zobrazovanie činnosti kombinovaných algoritmov je priamočiare – vlastne v každom uzle môže nastať zabezpečenie určitého stupňa redukcie.

Ak v procese redukcie ostane v doméne každej neviazanej premennej iba jedna hodnota a bola zabezpečená minimálne hranová konzistencia siete ohraničení, tak bolo nájdené riešenie. Ak však doména ľubovoľnej premennej sa stane prázdnou (nezávisle od toho, aký stupeň konzistencie je zabezpečovaný), riešenie buď neexistuje alebo niektoré priradenie hodnoty premennej nebolo konzistentné. V oboch prípadoch sa vyvolá navracanie. Podobne ak nejakej premennej je priradená taká hodnota, ktorá spôsobila porušenie nejakého ohraničenia, tak sa tiež vyvoláva navracanie.

Navracanie v kombinovaných algoritmoch je zložitejšie ako v prehľadávacích algoritmoch – je potrebné nielen uvoľniť jednu alebo viac premenných, ale taktiež je potrebné spätne odstrániť redukcie domén a ohraničení, spôsobené redukčnou zložkou algoritmu.

Samotnú redukciu, ktorá môže nastať v nejakom uzle stromu prehľadávania, možno rozdeliť do troch tried v závislosti od toho, ktoré premenné sa zúčastňujú tejto redukcie:

- iba viazané premenné (premenné s priradenou hodnotou)
- iba neviazané premenné (premenné bez priradenej hodnoty), označované aj ako budúce premenné
- viazané aj neviazané premenné.

Zabezpečovanie redukcie iba medzi premennými s priradenými hodnotami nemá zmysel – pretože priradením hodnôt premenným sa vlastne ich domény zredukovali na jednu hodnotu, tak už ďalšia redukcia nie je možná, a súčasne ak by tieto hodnoty neboli konzistentné, museli by spôsobiť porušenie nejakého ohraničenia (a to by automaticky malo za následok navracanie). Ak je teda k viazaných premenných, tak medzi nimi je zaručená prísna k-konzistencia.

Redukcia iba medzi neviazanými premennými zmysluplná je (v koreňovom uzle stromu prehľadávania, keď ešte ani jedna premenná nemá priradenú hodnotu, ani nič iné nie je možné). Ak by sa však iba tento krok zabudoval do prehľadávacieho algoritmu, tak by mal zmysel iba v koreňovom uzle – v žiadnom ďalšom uzle by už nedošlo k žiadnej redukcii domén neviazaných premenných či ohraničení, definovaných nad týmito neviazanými premennými.

Pri treťom type vlastne dochádza k redukcii domén neviazaných premenných a s nimi súvisiacich ohraničení v závislosti na hodnotách priradených premenným. Ak sa takáto redukcia realizuje v každom uzle stromu prehľadávania, tak potom sa zvyčajne uvažuje redukcia iba v súvislosti s posledne priradenou hodnotou (pretože zabezpečenie konzistencie neviazaných premenných voči skôr priradeným hodnotám bolo už realizované v predchádzajúcich uzloch).

Priradenie hodnoty nejakej premennej môže zapríčiniť redukciu domén neviazaných premenných alebo ohraničení, definovaných nad týmito premennými. Následný pokus o redukciu medzi neviazanými premennými môže spôsobiť ďalšiu redukciu. Ak teda sa v jednotlivých uzloch stromu prehľadávania používa zabezpečovanie nejakého stupňa redukcie medzi viazanými a

neviazanými premennými, tak má zmysel v týchto uzloch sa snažiť následne zabezpečiť aj redukciu medzi neviazanými premennými navzájom.

Vzájomné spojenie prehľadávania priestoru riešení a zabezpečovania redukcie do jedného kombinovaného algoritmu môže mať napríklad nasledovnú podobu[17]:

```
define method backtracking( i::<integer> )
   if ( and ( 0 < i, i <= n ) )
      V[i] := next( D[i], V[i] );
      if ( not ( V[i] ) )
         restore ( i-1 );
         backtracking ( i-1 );
      end if;
      if ( satisfy_all_constraints ( ) )
         check_present ( i );
         check_future ( i+1 );
         if ( empty_domain ( i+1 ) )
            restore ( i );
            backtracking ( i );
         else
            backtracking ( i+1 );
         end if;
      else
         backtracking ( i );
      end if;
   end if;
end method backtracking;
```

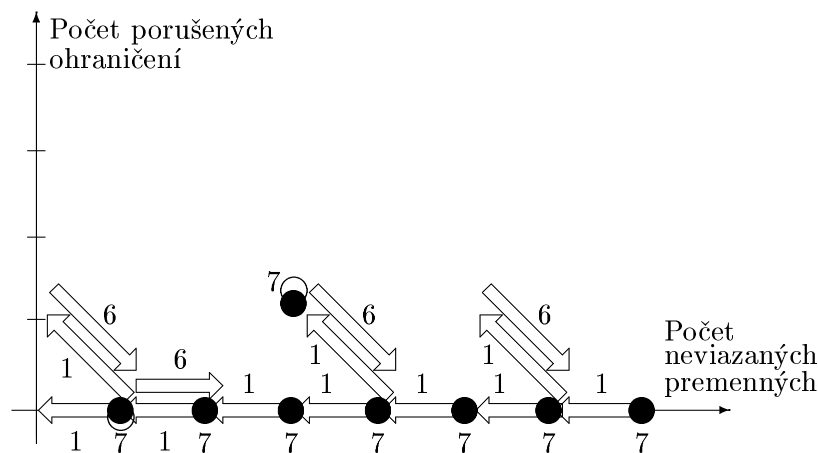
V uvedenej schéme sa používa spätné navracanie pre prehľadávaciu zložku – ale možno tu použiť aj iný prehľadávací algoritmus. Funkcia `check_present` reprezentuje redukčnú zložku algoritmu, kontrolujúcu neviazané premenné voči už priradeným hodnotám, a `check_future` zase redukčnú zložku, pracujúcu iba nad ešte neviazanými premennými. Keďže priraďovanie hodnôt jednotlivým premenným je realizované v určitom poradí, tak v prípade, že toto poradie je vopred známe (čo však nemusí byť – môže byť určované dynamicky), je postačujúce použiť smerové varianty redukčných algoritmov.

Rozhodnutie o tom, ktorý z uvedených typov redukcie sa použije, nemusí byť vopred dané ale môže byť dynamické počas riešenia úlohy. V [114] sa adaptívne určuje, či nejaká hrana bude spracovávaná iba ak jedna z príslušných premenných sa stane viazanou zatiaľ čo druhá ostáva neviazanou

[17] Volanie funkcie `restore(i)` zabezpečí odstránenie všetkých zmien v sieti ohraničení, realizovaných ako výsledok redukcie po poslednom priradení hodnoty premennej V_i.

(teda prípad viazanej a neviazanej premennej), alebo aj v prípade, že obe príslušné premenné sú neviazané. V [60] je zvolený zase iný prístup – stále sa začína tým, že sa využíva aj redukcia medzi neviazanými premennými. Táto sa používa dovtedy, pokiaľ nejaká podmienka je splnená. Ak táto podmienka prestane platiť, tak sa v ďalšom používa už iba redukcia medzi viazanými a neviazanými premennými. Možno použiť rôzne podmienky, napr. hĺbku stromu prehľadávania, meranie úspešnosti orezávania domén, veľkosť domén ap.

Z hľadiska stavového priestoru, hybridné algoritmy používajú širšiu paletu základných typov riešiacich krokov: priradenie, uvoľnenie a redukciu. Typický tvar výslednej trajektórie je na obr. 4.19.



Obr. 4.19: Typická trajektória pre kombinované algoritmy.

Počet redukčných krokov závisí od toho, v ktorých fázach prehľadávania sa šírenie ohraničení používa. Niekedy (napr. pri navracaní k predchádzajúcej premennej) je potrebné aplikovať redukčný krok v opačnom zmysle – je potrebné obnoviť stav siete ohraničení a jednotlivých domén do tvaru pred redukčným krokom (na obrázku je takýto spätný redukčný krok znázornený ako nevyplnený krúžok).

Slepý koniec s následným návratom k jednej z už naviazaných premenných môže byť detekovaný pri takomto type algoritmu v dvoch prípadoch: po aplikácii priraďovacieho kroku alebo po aplikovaní kroku redukčného. V oboch prípadoch prvé použitie uvoľňovacieho kroku končí na vodorovnej osi.

4.3.1 Kontrola budúcich premenných voči aktuálnej

Reprezentantom algoritmov pre zabezpečovanie konzistencie neviazaných premenných vzhľadom na už priradené hodnoty je *algoritmus doprednej kontroly* (forward checking – FC)[18]:

```
define method forward_checking ( i::<integer> )
   for ( j::<integer> from i+1 to n )
      for ( hj::<value> in D[j] )
         if ( not ( satisfy ( C[i,j], list ( V[i], hj ) ) )
            D[j] := remove ( D[j], hj );
         end if;
      end for;
   end for;
end method forward_checking;
```

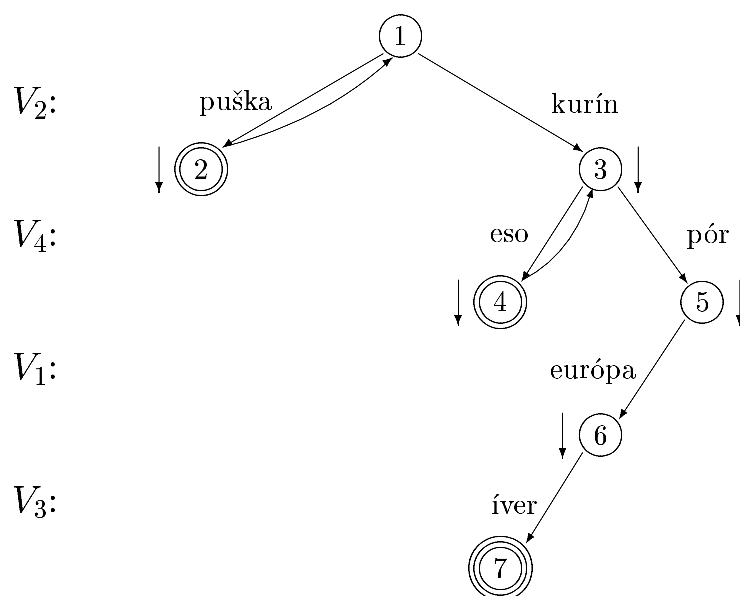
Algoritmus doprednej kontroly zabezpečí hranovú konzistenciu domén všetkých neviazaných premenných voči hodnote tej premennej, ktorá získala svoju hodnotu ako posledná. To znamená, že z domén všetkých neviazaných premenných sú vypustené všetky tie hodnoty, ktoré by po prípadnom priradení týmto premenným viedli k porušeniu binárnych ohraničení, definovaných nad premennou, ktorej práve bola priradená hodnota a jednou z neviazaných premenných.

Ukážka účinnosti tohto algoritmu je ilustrovaná na obr. 4.20. Obrázok zobrazuje strom prehľadávania pre problém podľa obr. 2.2, pričom sa uvažovali premenné v poradí V_2, V_4, V_1 a V_3 a hodnoty v D_4 sa uvažovali v obrátenom (nevýhodnejšom) poradí. Šípka pri uzle znamená, že v danom uzle bol aplikovaný algoritmus pre šírenie ohraničení.

Z obrázku je zrejmé, že dopredná kontrola nebola aplikovaná v koreňovom uzle stromu prehľadávania. To znamená, že pred priradením hodnoty prvej premennej nedošlo k žiadnej redukcii domén premenných. Oba neúspešné pokusy (s následným navracaním) boli detekované práve doprednou kontrolou. Priradenie hodnoty "puška" premennej V_2 spôsobilo, že šírením ohraničení došlo k úprave domén premenných V_1 (vypustená hodnota "žirafa") a V_3 (vypustené obe hodnoty). Prázdnosť domény D_3 signalizovala neexistenciu riešenia a preto bolo vyvolané navracanie. Navracanie v bode 4 bolo zase spôsobené vyprázdnením domén D_1 a D_3.

Ak by sa použil iba samotný algoritmus spätného navracania, tak by vznikol strom prehľadávania, ktorý by mal 19 uzlov a 9 z nich by reprezen-

[18] Je to vlastne funkcia `check_present` zo všeobecného hybridného algoritmu.



Obr. 4.20: Strom prehľadávania generovaný algoritmom doprednej kontroly pre ilustračný príklad pre poradie premenných V_2, V_4, V_1 a V_3.

tovalo nutnosť navracania. Doplnenie doprednej kontroly umožnilo vygenerovať strom iba so siedmimi uzlami a len dvoma prípadmi navracania. Z toho vidieť pomerne veľkú účinnosť algoritmu doprednej kontroly – aj keď je veľmi jednoduchý a zabezpečuje hranovú konzistenciu iba medzi niektorými uzlami siete ohraničení, tak jeho opätovné využívanie v každom uzle stromu prehľadávania umožňuje pomerne veľké orezanie priestoru prehľadávania.

Slabinou dopredných schém je to, že robia niekedy zbytočné kontroly. Ak by napríklad v príklade podľa obr. 2.2 boli k dispozícii ešte aj hodnoty "budova" (D_1), "voz" (D_4) a sto ďalších päťpísmenových slov (D_2), tak by časť stromu prehľadávania pri určitom usporiadaní premenných a hodnôt v ich doménach mohla vyzerať podľa obr 4.21.

V tomto prípade pri doprednej kontrole v uzle 2 boli zbytočne testované všetky hodnoty z domény D_2 premennej V_2 voči hodnote "budova" priradenej premennej V_1, ktorá aj tak neskôr bola zamenená inou hodnotou (a teda bolo nutné opäť zisťovať konzistenciu hodnôt z domény D_2).