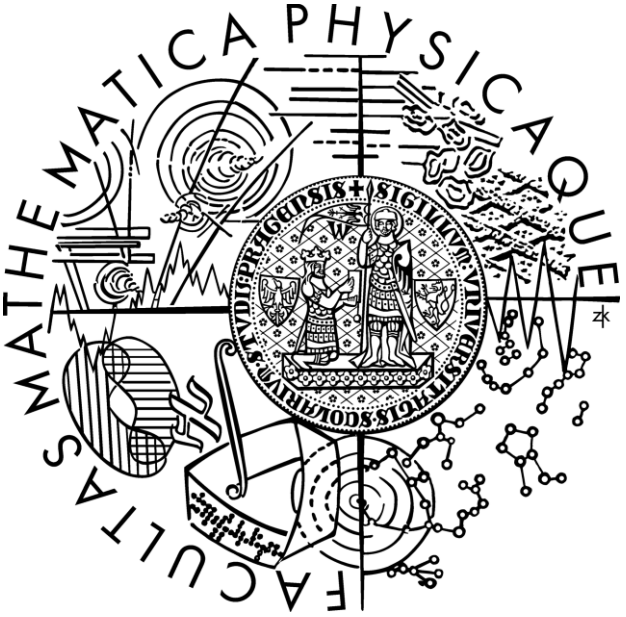
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Jan Pražma

Bridž: komplexní aplikace pro hraní bridže a vytváření/testování dražebních systémů

Katedra softwarového inženýrství
Vedoucí práce: Mgr. Pavel Ježek
Studijní program: Teoretická informatika

2012

Poděkování

Na tomto místě bych rád poděkoval vedoucímu své práce, Mgr. Pavlu Ježkovi, za cenné rady při vývoji aplikace a za to, že mě motivoval k dalšímu zlepšování bakalářské práce. Dále, stejně jako v bakalářské práci, děkuji Pavlu Trávníčkovi za zhotovení nových grafických materiálů použitých v programech Simulátor a Editor.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona

V Praze dne Bc. Jan Pražma

Obsah

Název práce: Bridž: komplexní aplikace pro hraní bridže a vytváření/testování dražebních systémů
Autor: Bc. Jan Pražma
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Pavel Ježek
e-mail vedoucího: pavel.jezek@ds3.mff.cuni.cz

Abstrakt: Cílem práce je zdokonalení stávajících aplikací Editor a Simulátor, které vznikly v rámci autorovy bakalářské práce. Dokument hovoří o hlavních aspektech bridžové hry, o licitaci a sehrávce. Celý dokument vychází z autorovi bakalářské práce, na níž je v dokumentu často odkazováno. Autor se nicméně snažil, aby dokument byl čitelný i bez znalosti jeho předchozí práce.

Dokument se konkrétněji zabývá autorovou snahou o vytvoření základní typologické množiny výroků, jakožto prostředku pro popsání situací bridžového rozdání a tyto výroky jsou pak použity při konstrukci dražebních konvencí a systémů. Autor diskutuje bridžové problémy a prezentuje zdokonalený algoritms pro řešení úlohy s odkrytými kartami. Práce se též zabývá samotnou softwarovou strukturou a krátce i grafickým rozhraním obou aplikací.

Klíčová slova: bridž, dražba, licitace, sehrávka, double-dummy

Title: Bridge: Complex application for playing contract bridge and the creation and analysis of Bridge auction systems
Author: Bc. Jan Pražma
Department: Department of Software Engineering
Supervisor: Mgr. Pavel Ježek
Supervisor's email: pavel.jezek@ds3.mff.cuni.cz

Abstract: The goal of the thesis is to improve the existing Editor and Simulator applications that have been created by the autor as his Bachelor Thesis. Document speaks about the main aspects of bridge; about the auction and the bridge play. The whole document is based on the author's Bachelor Thesis which is frequently referenced in the document. Nevertheless, the document should be readable without any previous knowledge of author's former work.

The author is discussing his effort to create logic expressions to describe bridge hands and truths in the bridge world. These expressions are then used to construct auction conventions and whole auction systems. The author is discussing the bridge problems and presenting the improved algorithm to solve double-dummy problems. The work also describes the software from the engineering point of view and speaks about the graphic user interface of both programmes.

Key words: bridge, bidding, double-dummy, auction, play,

1. Úvod

Bridž je komplexní hra. Skládá se totiž ze dvou komplementárních částí – dražby a sehrávky. Dražba i sehrávka jsou dva rozdílné problémy, se kterými se musí hráč, ať už „živý“ nebo počítačový, potýkat. Vše je navíc ztíženo faktem, že hráči si navzájem nevidí do karet - bridž je hra s neúplnou informací. Hru hrají čtyři hráči a sice dva páry hráčů proti sobě. Zcela legitimně se tak dá na problematiku pohlížet multiagentní optikou, kde v systému jsou celkem čtyři agenti, z nichž jsou dva nepřátelští a jeden kooperující.

Bridžové problémy jsou stále „živé“, zatím se nenalezl algoritmus, který by úlohy stabilně řešil lépe nebo stejně dobře jako bridžový expert. Bridžová sehrávka se nyní ocitá na pomyslné hraně již vyřešených problémů. Objevilo se několik implementací algoritmu na řešení sehrávky s odkrytými kartami (převedení na hru s úplnou informací, tzv. double-dummy úloha), které jsou s pomocí agresivního heuristického prořezávání schopny projít celý herní strom v řádu desítek nebo dokonce jednotek sekund [10]. Bohužel, konkrétní řešení originální úlohy se zakrytými kartami stále nedosahují expertních kvalit, ačkoliv se této hranici pomalu přibližují.

Dražba je proces, kdy se hráči (agenti) pomocí úzké množiny povolených hlášek snaží dosáhnout dvou věcí: předání informace o svých kartách partnerovi a vydražení nejvhodnějšího závazku, který definuje styl a „obtížnost“ následující sehrávky. Agent tedy musí mít implementován rozhodovací mechanismus, který se na základě předchozích informací rozhodne pro zvolení aktuálně nejvhodnější hlášky. Zároveň sleduje hlášky ostatních, obzvláště svého kooperujícího partnera a snaží se z nich vytěžit co nejvíce nových informací. Agresivní agenti mohou dokonce vhodnou hláškou zablokovat dražební prostor soupeřů. Bridžový software je opět pod úrovní experta, většinou dobře zvládá situace s větší pravděpodobností výskytu, ale u okrajových rozdání může být výsledek suboptimální. Velkým problémem pro počítač je agresivní dražba protihráčů [6].

Autor nabízí svůj pohled na řešení těchto problematik ve formě softwarového díla. Implementace řeší sehrávku a dražbu způsobem, který je diskutován níže v tomto textu. Hlavním parametrem pro řešení dražby byla možnost vytvářet a editovat dražební pravidla a obecnou zásadou byla jednoduchá ovladatelnost.

1.1 Cíle práce

Hlavním cílem diplomové práce bylo zdokonalení obou aplikací vytvořených v rámci autorovy bakalářské práce. Výsledek si měl ponechat to nejlepší z obou programů, měl opravit špatně implementované části a přidat novou funkcionalitu. Neduhem aplikace Editor byla malá informativnost a nepřehlednost, stejně jako nepříjemné blikání komponent při editaci větších systémů. Toto mělo být v nové verzi zlepšeno, stejně tak jako informace o chybně vytvořených hláškách a chybách v systému obecně. Dále mělo v aplikaci Editor dojít k rozšíření rozsahu možností pro pokrytí větší množiny rozdání a aplikace měla umožňovat zachycení obranné dražby, tj. uživatel měl mít možnost dotazovat se na pravdy o soupeřích, neboť starý Editor umožňoval pouze dotazy na pravdy týkající se pouze aktuálního hráče.

V aplikaci Simulátor mělo dojít ke zlepšení algoritmu pro řešení double-dummy úlohy, čímž by se automaticky zlepšil algoritmus sehrávky (single-dummy).

Pokud bude vytvořena rozumná implementace algoritmu, tak mělo být uživateli umožněno prohlédnout si optimální sehrávku pro dané rozdání. Toto vyžaduje, aby byl Simulátor schopen vyřešit celou double-dummy úlohu během desítek vteřin, jinak je řešení pro běžného uživatele nepoužitelné. Poslední novou vlastností Simulátoru měla být možnost otestovat vícero dražebních systémů na velkém počtu rozdání a tak změřit jejich úspěšnost.

Samostatnou kapitolou je pak vylepšení získávání informací jednotlivých hráčů z dražby a během sehrávky. Tyto informace měly být využity pro zpřesnění vstupních dat algoritmu sehrávky, ale zároveň získáváním těchto informací si hráč (agent) buduje množinu pravdivých tvrzeni o pozorovaném světě a na základě těchto skutečností vyhodnocuje i samotnou dražbu.

Konceptuálním požadavkem na obě aplikace je, a bylo tomu tak i v autorově bakalářské práci, aby výsledek co možná nejvíce připomínal moderní, komerční software, který je připraven na použití běžným uživatelem - bridžistou. Motivací bylo hlavně to, aby vytvořené programy mohli, třeba někdy v budoucnu, skuteční bridžisté opravdu používat. Proto ovládání software musí být pochopitelné i pro neinformatiky. Programy ale nikdy nebyly zamýšlené k výuce bridže, tzn. uživatel musí bridž již znát. Základním paradigmatem bylo, že vše musí být ovladatelné myší a ovládání by mělo být co nejvíce ergonomické a intuitivní. Nepřijatelný byl program běžící na příkazové řádce.

2. O bridži

Historie vzniku bridže a jeho pravidla jsou podrobně popsána v autorově bakalářské práci. Proto následující kapitola rozebírá tuto problematiku již jen stručně. V případě hlubšího zájmu o pravidla či o původ bridže autor odkazuje na citovanou bakalářskou práci [11] či na libovolné naučné stránky o bridži [7].

2.1 Pravidla hry

Bridž se hraje s balíčkem 52 klasických žolíkových karet (bez žolíků) a je velmi podobný českému mariáši. Jednotlivé karty se od sebe liší svojí hodnotou, eso (A) je nejvyšší karta, dvojka nejnižší. Karta kromě své hodnoty má i barvu, a sice nejnižší barvou jsou trefy ♣, dále kára ♦, srdce ♥ a nejvyšší barvou jsou piky ♠. Na tuto barevnou hierarchii se bere zřetel pouze během licitace.

Hru hrají čtyři hráči, kteří utvoří dvě partnerské dvojice hrající proti sobě. Hráči jsou běžně označováni světovými stranami (Z, S, V, J) dle posazení u hracího stolu, jih a sever tvoří jednu partnerskou linku, západ a východ linku druhou. Každému hráči se rozdá na začátku 13 karet, cílem hry je vydražit a uhrát co možná nejvyšší závazek.

Hra se, stejně jako mariáš, dělí na dvě části – licitaci (dražbu) a sehrávku (samotné hraní karet). V licitaci se hráči střídají po směru hodinových ručiček a snaží se najít stupeň a druh závazku, který je jejich linka schopná uhrát. Hodnota závazku (1-7) značí počet nutných zdvihů k úspěšnému splnění závazku, barva určuje trumfovou barvu. Je dovoleno použít speciální beztrumfovou barvu jako prostředek pro nespecifikování žádné trumfové barvy. Hráč může vyslovit hlášku pouze pokud je vyšší než posledně vyslovená hláška (4♣ > 3♠> 3♣**).** Nemá nebo nechce-li hráč nic říkat, tak použije hlášku pass. Poslední hláška následovaná třemi pasy je hláška vítězná a linka, jež tuto barvu dražila, se stane útočnou (sehrávající) linkou. „Hlavním hráčem“ se stává hráč z této linky, který poprvé dražil vítěznou barvu. Nemusí to tedy nutně být autor poslední, vítězné hlášky. Ještě je nutné dodat, že v dražbě lze použít další dvě umělé hlášky „kontra“ a „rekontra“, které se používají zcela stejně jako v mariáši hlášky „flek“ a „re“. Nicméně, moderní dražební systémy tyto hlášky značně „zneužívají“ a jejich původní význam coby „trestu“ za špatnou licitaci a zdvojnásobení získané bodové hodnoty se tak často vytrácí.

Po vydražení závazku přichází na řadu sehrávka, tedy samotná hra, kdy se útočná linka snaží uhrát vydražený závazek. Počet zdvihů nutný k vítězství sehrávající linky je roven hodnotě závazku + 6 (tzn. nejnižší možný závazek 1 předurčuje k uhrání 7 zdvihů). Hráč po levici hlavního hráče učiní otevírací výnos. Po úvodním výnosu se stane pro bridž naprosto zásadní věc (hlavní rozdíl v sehrávce oproti mariáši) a sice partner hlavního hráče položí všechny své karty na stůl lícem nahoru. Od tohoto momentu s nimi disponuje výhradně hlavní hráč. Hráči se střídají v hraní karet po směru hodinových ručiček, vynášející hráč může zvolit libovolnou kartu ve své ruce. Barva první karty ve zdvihu určí barvu zdvihu a následní hráči musí přikládat jen karty této barvy, pokud je v ruce mají (musí „držet“ barvu, na rozdíl od mariáše, ale nemusí brát, pokud nechtějí). Pokud nemají, mohou přiložit libovolnou kartu ze své ruky. Poté, co přiloží karty všichni čtyři hráči, se určí vítěz zdvihu tak, že nejvyšší karta, jejíž barva se shoduje s barvou zdvihu (nebo je to barva trumfová, trumfy jsou vždy vyšší než ostatní barvy) vyhrává a její bývalý majitel má právo vynést do dalšího zdvihu. Postup se opakuje až všem hráčům dojdou karty. Vítězem

se stává sehrávající linka, pokud se jí podařilo uhrát přesně nebo více zdvihů než určuje závazek. Obranná linka vítězí v opačném případě.

V turnajových podmínkách se karty nerozdávají, ale hraje se s předpřipravenými kartami pro všechny hráče stejnými. Vítězem pak je linka, která na daném rozdání dosáhne co nejlepší relativní výsledek vůči ostatním linkám. Vítězem tedy může být paradoxně partnerská dvojice, která sice prohrála, ale menším rozdílem než ostatní hráči.

2.2 Proč vůbec dražba

Přesnou odpověď asi nezískáme, museli bychom se zeptat anglických gentlemanů z přelomu 18. – 19. století. Hra se patrně stala zajímavější nejen z hlediska přidání trumfové barvy, ale i z hlediska toho, že přidání odhadu počtu odehraných zdvihů před hrou efektivně smazává náhodnost rozdaných karet. I v mariáši lze hrát takzvaného „betla“, což je modifikace mariáše, kdy se jeden z hráčů zavazuje, že neuhraje ani jediný zdvih. Trik spočívá v tom, že pokud chce hráč hrát „betla“, musí to oznámit před začátkem vlastní hry. Situaci je tedy nutné odhadnout pouze pohledem na své karty. Hráčské umění začne být mnohem výraznější a možná i v důsledku přidání dražby se bridž později stává respektovanou hrou.

Ale zpět k dnešnímu bridži. Výsledkem dražby je závazek, který určuje jak trumfovou barvu do následující sehrávky (nebo pátou speciální barvu „beztrumfy“), tak i počet nutných zdvihů k uhrání závazku pro sehrávající linku. Závazek má vlastně duální charakter. Určuje styl hry specifikací trumfové barvy a určuje obtížnost hry svým stupněm. Zde je důležité připomenout, že v bridži je závazek splněn v momentě, kdy je uhrán předepsaný nebo větší počet zdvihů. Tím se liší například od hry whist, coby prapůvodní varianty bridže, kde je nutné svůj závazek uhrát na zdvih přesně, jakýkoliv nadzdvih znamená neúspěch. Nutno však podotknout, že turnajový bridž bodově zvýhodňuje přesně vydražené a uhrané závazky, takže trend profesionálního bridže je být co nejpřesnější. Přesněji, v bridži existují tři hranice závazků, jejichž dosažení je oceněno bonusovými body: celoherní závazek, malý slam a velký slam[13]. Na rozdíl od mariáše, kde hraje každý hráč za sebe, zde stojí partnerská dvojice proti partnerské dvojici. To znamená, že partneři se pokouší najít nejlepší závazek vyhovující oběma spoluhráčům. K tomu je ale nutná určitá forma komunikace mezi hráči, protože z pouhého pohledu na své vlastní karty lze jen těžko usuzovat, jaké karty má spoluhráč. Můžeme jen spekulovat, jestli v momentě zavedení trumfových hlášek do hry (konec 19. století) bylo již počítáno s komunikací, která se dá na hláškách vystavět nebo měly hlášky pouze přirozený význam a komunikace, která vedla ke vzniku prvních dražebních systémů, se vyvinula až posléze.

2.3 Od hlášek k systému

Tato „komunikace“ probíhá v předem stanovených mantinelech a těmi jsou povolené dražební hlášky. Hlášek je celkem 38, 35 uspořádaných hlášek závazků (1♣, 1♦, 1♥, 1♠, 1**BT**, 2♣,2♦, .., 7♠, 7**BT**) a 3 speciální hlášky, které lze používat opakovaně (pass, kontra, rekontra). Vstup do dražby hlášením libovolného závazku předurčuje hráče k zisku alespoň nadpoloviční většiny zdvihů, proto se počty zdvihů nutné k uhrání závazku získají formulí 6 + stupeň závazku. Na rozdíl od mariáše se v bridži hráč nemůže sám dobrovolně dostat do pozice „betla“, tzn. sám od sebe nemůže deklarovat svojí linku jako obrannou, to musí být učiněno dražbou soupeře a tudíž, trochu nadsazeně, hráč se silnějšími kartami určuje styl sehrávky pro všechny

ostatní hráče. Každé pravidlo má ale svojí výjimku a v bridži je zcela běžná takzvaná „obranná“ dražba, kdy hráč vysokou hláškou (na kterou by neměl mít bodově nárok) zablokuje dražební prostor soupeřům a sníží tak jejich schopnost se domluvit na správném závazku. Občas se stane, že si žádný z hráčů netroufne závazek dražit, v takovém případě se sehrávka nekoná a hra končí remízou. Na větších turnajích je mezi hráče umístěna plenta, aby se zajistilo, že komunikace bude opravdu probíhat jen ve vymezených mantinelech a nezmění se v důmyslný systém úšklebků a zakašlání.

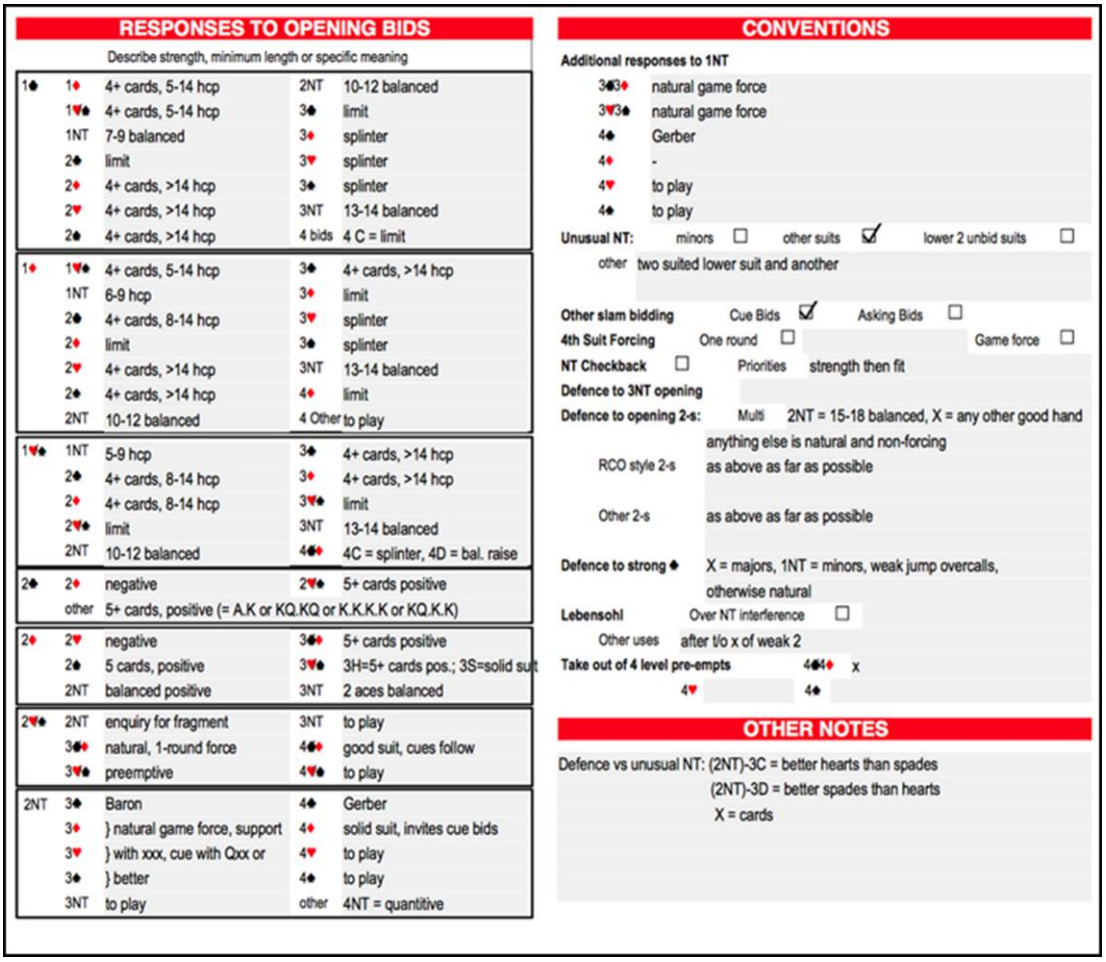
Dražební systém je předem domluvená komunikace, je to přidělený význam sekvencím hlášek, jejichž pomocí si oba partneři popisují své karty na ruce a upřesňují informace o společné síle. Každá partnerská dvojice může mít svůj vlastní systém, může si vytvořit své vlastní sekvence a významy hlášek. Systém ale není tajný, soupeř má právo znát význam hlášek protihráče, a to dokonce v takové míře, že může dražbu zastavit a nechat si vysvětlit význam právě dražené hlášky partnerem dražitele. Tady pak nastávají situace, kdy partner na dotaz vysvětlí hlášku zcela jinak, než kolega dražitel původně zamýšlel. O problému, kdy se z významu hlášky pokoušíme určit, o kterou hlášku v našem systému se jedná, pojednává kapitola 3.2.14 této práce. Problém vychází ze samé podstaty dražebního systému a ze snahy co nejvíce herních situací zachytit na fixně veliké množině hlášek, takže některé hlášky budou nutně přetížené vícero významy. Například hláška 2♥ může mít i v tom nejjednodušším systému hned několik významů – může značit 5+ list v srdcích a 18-24 bodů, jedná-li se o zahájení nebo jako odpověď navrhuje jinou vhodnější barvu (4+ list a 10+ bodů). V neposlední řadě se může jednat o potvrzení partnerem dražené barvy (8+ v srdcích na naší lince, 6-10 bodů).

2.4 Přirozené a umělé systémy

S trochou nadsázky by se dalo říct, že přirozená dražba je dražba, která nepotřebuje systém. To znamená, že hlášení barvy má opravdu znamenat karetní převahu. Úrovně hlášek nejsou blokující, ale vyjadřují reálnou karetní sílu. Karetní síla se v bridži popisuje takzvanými figurovými body; eso je za 4, král za 3, královna za 2 a kluk za 1 bod. Celkový součet figurových bodů v balíčku je 40. Například, v přirozeném systému bude hláška 1♥ zcela jistě nabízet 4 a více karet srdcové barvy a bude slibovat 11+ bodů (10 bodů je průměrná hodnota na ruku, aby mohlo být zahájeno, je třeba uhrát alespoň závazek hodnoty 1 a to znamená mít na lince nadprůměrnou hodnotu bodů). Nejtypičtějším zástupcem přirozených systémů je dražební systém Acol (obr 1).

Umělé systémy, nebo umělé hlášky obecně jsou takové hlášky, které nemusí nutně „mluvit pravdu“, tzn. dražená barva nemusí značit sílu v této barvě. Umělé hlášky (zvané také jako „konvence“) se sice vyskytují i v přirozených systémech, ale jejich použití je řídké. Nejznámějším umělým systémem na světě je nejspíše Polský Tref. Jeho princip spočívá v tom, že naprosto nebere v potaz barvu hlášek. Hláška 1♥ v umělém systému zpravidla vůbec nebude nabízet srdce, ani nemusí znamenat nadprůměrnou bodovou sílu.

Umělé hlášky a systémy neustále přibývají a fantazie jejich autorů nebere mezí. Podmínky pro jejich vytváření totiž skoro žádné nejsou – každá partnerská dvojice si může svůj vlastní systém navrhnout jak bude chtít.



RESPONSES TO OPENING BIDS

Describe strength, minimum length or specific meaning

1♠	1♦	4+ cards, 5-14 hcp	2NT	10-12 balanced
	1♥/♠	4+ cards, 5-14 hcp	3♠	limit
	1NT	7-9 balanced	3♦	splinter
	2♣	limit	3♥	splinter
	2♦	4+ cards, >14 hcp	3♠	splinter
	2♥	4+ cards, >14 hcp	3NT	13-14 balanced
	2♠	4+ cards, >14 hcp	4 bids	4 C = limit
1♦	1♥/♠	4+ cards, 5-14 hcp	3♠	4+ cards, >14 hcp
	1NT	6-9 hcp	3♦	limit
	2♣	4+ cards, 8-14 hcp	3♥	splinter
	2♦	limit	3♠	splinter
	2♥	4+ cards, >14 hcp	3NT	13-14 balanced
	2♠	4+ cards, >14 hcp	4♦	limit
	2NT	10-12 balanced	4 Other	to play
1♥/♠	1NT	5-9 hcp	3♠	4+ cards, >14 hcp
	2♣	4+ cards, 8-14 hcp	3♦	4+ cards, >14 hcp
	2♦	4+ cards, 8-14 hcp	3♥/♠	limit
	2♥/♠	limit	3NT	13-14 balanced
	2NT	10-12 balanced	4♣/♦	4C = splinter, 4D = bal. raise
2♣	2♦	negative	2♥/♠	5+ cards positive
	other	5+ cards, positive (= A.K or KQ.KQ or K.K.K.K or KQ.K.K)		
2♦	2♥	negative	3♣/♦	5+ cards positive
	2♠	5 cards, positive	3♥/♠	3H=5+ cards pos.; 3S=solid suit
	2NT	balanced positive	3NT	2 aces balanced
2♥/♠	2NT	enquiry for fragment	3NT	to play
	3♣/♦	natural, 1-round force	4♣/♦	good suit, cues follow
	3♥/♠	preemptive	4♥/♠	to play
2NT	3♣	Baron	4♣	Gerber
	3♦	} natural game force, support	4♦	solid suit, invites cue bids
	3♥	} with xxx, cue with Qxx or	4♥	to play
	3♠	} better	4♠	to play
	3NT	to play	other	4NT = quantitive

CONVENTIONS

Additional responses to 1NT

3♣/♦	natural game force
3♥/♠	natural game force
4♣	Gerber
4♦	-
4♥	to play
4♠	to play

Unusual NT: minors ☐ other suits ☑ lower 2 unbid suits ☐

other: two suited lower suit and another

Other slam bidding Cue Bids ☑ Asking Bids ☐

4th Suit Forcing One round ☐ Game force ☐

NT Checkback ☐ Priorities strength then fit

Defence to 3NT opening

Defence to opening 2-s: Multi 2NT = 15-18 balanced, X = any other good hand

anything else is natural and non-forcing

RCO style 2-s: as above as far as possible

Other 2-s: as above as far as possible

Defence to strong ♣ X = majors, 1NT = minors, weak jump overcalls, otherwise natural

Lebensohl Over NT interference ☐

Other uses: after t/o x of weak 2

Take out of 4 level pre-empts 4♣/♦ x

4♥ 4♠

OTHER NOTES

Defence vs unusual NT: (2NT)-3C = better hearts than spades

(2NT)-3D = better spades than hearts

X = cards

Obr. 1 – Ilustrační obrázek reálného dražebního systému

3. Rozbor dražby

3.1 Popis problému dražby

V této kapitole bude vysvětlena podstata problémů bridžové dražby, budou nastíněny způsoby, jakými problémy řeší dostupný bridžový software a prezentováno bude řešení, které vytvořil autor práce.

3.1.1 Popis problému

Dražba se obecně považuje za nejtěžší a nejdůležitější část hry. Bridžoví velmistři jsou známí tím, že jejich úroveň sehrávky je na téměř totožné, precizní úrovni a hra se většinou rozhoduje během dražby. Jak již bylo řečeno, účelem dražby je nalézt nejvhodnější závazek na základě komunikace mezi partnery, která probíhá v rámci dražby. Nejvhodnější závazek je takový, který je co nejvyšší, ale zároveň splnitelný. Zjednodušeně se dá říci, že partneři se na bázi vzájemné komunikace snaží o odhadnutí síly společných karet a nalezení co nejlepšího stylu hry, který by sliboval co nejvyšší bodový zisk. Komunikace mezi hráči probíhá na úrovni hlášek samotných a hláška má tak duální charakter – jednak reprezentuje hodnotu, čili konkrétní závazek, ale zároveň slouží jako nositel informace, kterou si mohou hráči do hlášek či jejich posloupností zakódovat.

Sémantiku tohoto „kódu“ si může každá partnerská dvojice zvolit zcela libovolnou a tato sémantika je dražební systém (viz 2.3). Dražební systém je sada pravidel, pravidlo se skládá z podmínky a efektu. Podmínka určuje platnost pravidla nad dražbou (sekvencí hlášek) a nad aktuální rukou, efekt je pak samotná hláška. Každé pravidlo tedy svojí podmínkou určuje množinu rozdání, na kterou se vztahuje. Některá rozdání mohou splňovat podmínky vícera různých pravidel. Ideální dražební algoritmus by měl být invariantní vzhledem k dražebnímu systému, tzn. měl by dosahovat optimálních výsledků nezávisle na zvoleném dražebním systému.

Na začátku dražby vidí agent svojí ruku z 13 kartami z celkových 52. Chybí mu tedy 75% informace z celkového stavu. Počet všech možných karetních kombinací, které může hráč dostat na ruku, je $6.35 * 10^{11}$. Každý z těchto možných částečných stavů může být zkompletován do úplného stavu celkem $8.45 * 10^{16}$ způsoby [2]. Agent tedy začíná s poměrně úzkou množinou znalostí reprezentujících část aktuálního stavu a během dražby (i během sehrávky) si tento stav postupně doplňuje získáváním informací od ostatních agentů. Aktuální stav znalostí agenta je tedy implikován pouze jeho rukou a posloupností dražených hlášek. Pokud bridžový software znalostní báze agentů implementuje, tak jsou téměř vždy reprezentovány jako množiny rozdání, které neodporují doposud získaným informacím. Velmi zřídka se setkáváme s jinou reprezentací znalostí, například program Micro Bridge [19] reprezentuje získané znalosti jako množinu platných pravidel, na rozdíl od množiny platných rozdání.

Na rozdíl od sehrávky, kterou doplněním na double-dummy problém můžeme optimálně řešit klasickým způsobem (minimax algoritmus), u dražby tomu tak není. Ani doplněním na úplný stav (tzn. odkrytím všech karet) nedokážeme zajistit, že rozhodovací algoritmus najde optimální závazek vzhledem ke zvolenému systému. Vychází to ze samotné podstaty dražebního systému, kdy velmi omezený soubor pravidel popisuje řádově větší množinu všech rozdání. Každý systém má svá slabá

místa, množiny rozdání, které nezachycuje jeho soubor pravidel. Perfektní informace nezaručuje schopnost systému nalézt optimální závazek [10].

Další problém pro minimax nad stromem hlášek je ten, že v bridži se kvalita dražby dá posoudit až s poslední hláškou, která značí finální závazek. Veškeré hlášky dražené před finální hláškou znamenaly většinou pouze komunikaci mezi partnery a tudíž je nemožné vytvořit univerzální ohodnocující funkci, která by dokázala ohodnotit vrcholy uprostřed herního stromu [2]. Algoritmus tedy musí vždy dojít až do listů herního stromu hlášek, aby mohl ohodnotit kvalitu zkoumané cesty. Pravděpodobně z výše uvedených důvodů se autor nesetkal s žádnou reálnou implementací minimaxového algoritmu, která by procházela možné sekvence hlášek.

Otázkou, které musí hráči-agenti také čelit, je častá víceznačnost hlášek v systému. Když hráč hledá v dražebním systému pravidlo, jehož podmínky vyhovují aktuálně pozorovanému světu, snadno se stane, že nalezne víc takových pravidel. Pokud se pak jeho partner snaží odvodit, jaká rozdání reprezentovala hláška partnera (jakou informaci nesla), tak výsledek nemusí být jednoznačný – hráč jednoduše neví, z čeho odvozovat, protože nalezl více pravidel, jejichž podmínka nezamítá aktuální situaci. S tímto problémem se autor setkal již ve své bakalářské práci, v níž ho pojmenoval „problém zpětného určení“. O tomto problému se zmiňují Amit a Markovitch [2] a Ginsberg[6], ale nenabízí uspokojivé řešení. V implementaci využívající kooperativní učení představené v [2] tento problém v průběhu učící fáze agentů postupně mizí, jak si agenti více a více „zvykají“ jeden na druhého, nikdy ale nevymizí úplně (více 3.1.2). Problém „zpětného určení“ lze zcela jistě zmírnit, či dokonce vyřešit, vhodnou konstrukcí dražebního systému. Protože většina dostupného bridžového software draží dle předdefinovaných dražebních systémů, lze usoudit, že tyto systémy jsou vytvořeny natolik „bytelně“, že problém nejednoznačnosti nemůže nastat. Vzhledem k tomu, že implementační řešení vypracovávané autorem zahrnuje editor dražebních systémů, je tento problém více viditelný a autor s ním musel počítat. (viz 3.2.14).

3.1.2 Jak je řešena bridžovým software

Bridžový software ještě nedávno řešil převážně sehrávku, dražba byla dlouho upozaděná. Protože se kvalita sehrávky v poslední dekádě výrazně zlepšila, kvalita počítačové dražby začala být problém. Poslední roky se dražba dostává opět do hledáčku informatiků a objevují se zcela nové, zajímavé implementace řešící problém dražby. Je třeba dodat, že konkrétní implementační detaily o bridžovém software se získávají poměrně obtížně, pokud jsou vůbec získatelné. Ty nejlepší bridžové programy se spolu utkávají každý rok v klání elektronického bridže [14] a jejich autoři si svá implementační řešení hlídají částečně i z komerčního hlediska.

Nejklasičtější a nejrozšířenější řešení je dražba pomocí sady pravidel. Toto řešení je nejvíce věrné skutečnosti, sada pravidel není nic jiného než dražební systém. Pravidla bývají nejčastěji fixně zvolena, uživatel má jen omezenou možnost vybrat si předvytvořené sady pravidel.

Pravidlový přístup řešení problému dražby skýtá několik nedostatků částečně zmíněných v předchozí kapitole. Žádný dražební systém nemůže jednoznačně svojí sadou pravidel pokrývat všechna myslitelná rozdání a dražební situace. Vždy bude existovat rozdání, které bude splňovat podmínky několika různých pravidel, nebo dokonce nebude popsáno pravidlem žádným. Podobným problémem je případ, kdy partner použije špatnou hlášku nebo v případě zúžení dražebního prostoru agresivní obrannou dražbou soupeře (toto je dle [6] místo, kde lidský protivník stále výrazně

převyšuje protivníka počítačového). S uvedenými problémy se samozřejmě běžně potýká každý bridžový hráč, člověk je ale schopen díky předešlým zkušenostem a „heuristickému“ uvažování tyto situace zvládat a zvolit vhodnou či podobnou hlášku. Počítačový hráč, striktně se řídící pravidly, toto ale učinit nemůže a výsledek dražby bude při konfrontaci s těmito problémy narušen, v horším případě program přestane dražit úplně. Pro menší bridžový software toto není až tak zásadní problém, protože moderní dražební systémy jsou poměrně dobře zkonstruované a pokrývají velkou množinu rozdání.

Komerční bridžové programy, ve snaze překonat herní kvalitou člověka, ale tento fakt ignorovat nemohou. Logickým postupem je zachovat množinu pravidel, kterými se počítač při své dražbě řídí a v případě problému s nalezením vhodného pravidla spustit dodatečný rozhodovací mechanismus, který rozhodne o finální hlášce. Jak podobnou situaci řeší asi nejznámější program GIB [17], vyvinutý Matthew Ginsbergem, je podrobně popsáno v [6]. GIB používá sadu pravidel pro popis dražby. Zároveň si ale udržuje databázi dražebních sekvencí získanou z reálných, nepočítačových her (údajně obsahuje více jak 60000 reálných dražeb). Rozhodovací proces začne vybráním vhodných pravidel z dražebního systému, které svými podmínkami neodporují situaci. Nahlédnutím do databáze reálných dražeb GIB zjistí, jakými finálními závazky běžně končívají dražby pokračující z těchto pravidel (zajímavý pro GIB je pouze finální závazek každé dražební posloupnosti). Nyní pomocí Monte-Carlo vzorkování vytvoří množinu rozdání, která neodporují doposud získaným znalostem a na tato rozdání pustí algoritmus sehrávky (double-dummy). Hláška (pravidlo) vybrané rozhodovacím mechanismem je hláška, která slibuje nejvyšší a stále splnitelný závazek, kde splnitelnost je určena výsledky double-dummy analýzy.

Zajímavý postup zvolili Amit a Markovitch [2].Řešení spočívá ve strojovém učení agentů na náhodně generované bázi rozdání. Základním konceptem je společný trénink obou agentů – partnerů. Agenti, fungující na bázi rozhodovacích sítí, se učí od svého vzniku společně a mohou tak vzniknout vzorce, kterým rozumí jen agenti navzájem. Agenti si tak vlastně vytvářejí vlastní dražební systém, vlastní soubor pravidel, podle kterých pak draží. Důkladným učením lze dle [2] překonat i problémy zpětného určení. Dle výsledků prezentovaných uvedenými autory se podařilo vytrénovat agenty, kteří dosahovali lepších dražebních výsledků než program GIB. Nevýhodou tohoto postupu je nemožnost vyměnit partnera. Oba agenti jsou na sebe natolik naučení, že mohou fungovat pouze společně. Bridžový expert je ale schopen hrát s libovolným jiným bridžovým expertem bez větší újmy na výkonosti. Autor si není vědom toho, že by výsledky této či podobné metody byly reálně implementovány v nějakém konkrétním bridžovém software.

3.2 Jak řeší dražbu autor práce

Autor práce implementuje rozhodovací mechanismus agentů jako sadu pravidel, z kterých si agent během dražby vybírá. Množina pravidel ale není statická, uživatel má možnost pravidla vytvářet a hierarchizovat. Možnost vytvořit si vlastní dražební systém byl definující faktor při vypracovávání této práce. Každý agent si drží v paměti množiny rozdání, které neodporují zatím získané informaci a během dražby je tato informace zpřesňována. Následující kapitoly popisují reprezentaci pravidel, podmínek, znalostí hráčů a veškeré operace, které se s nimi provádějí.

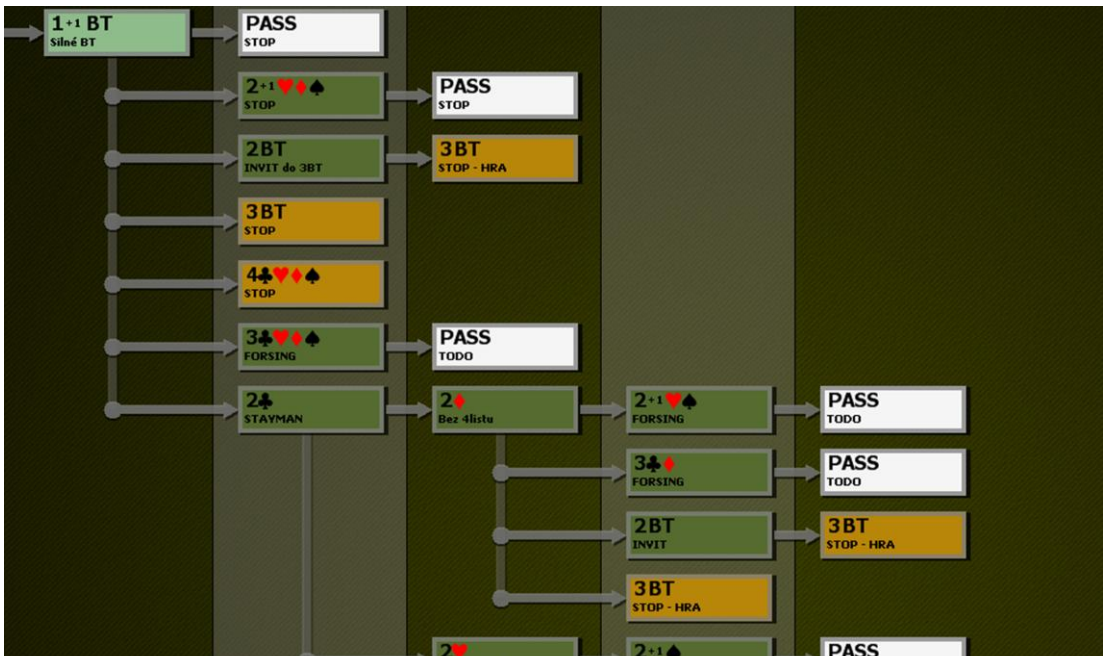
3.2.1 Reprezentace zahlášených hlášek

Nejprve si označme zahlášenou hlášku jako **e**, kde **e** je dvojice složená ze stupně hlášky a její barvy, **e** = ({**0**,**1**,..,**7**}, {**pass**, **kontra**, **rekontra**, **♣**, **♦**, **♥**, **♠**, **BT**}). Dále je nutné držet přehled o již zahlášených hláškách, nechť je toto množina **D** = {$\mathbf{e_1}$,$\mathbf{e_2}$,..}. Množina **D** je na začátku hry prázdná a postupně se plní.

3.2.2 Reprezentace struktury systému

Na dražební systém je nahlíženo jako na množinu pravidel nebo ještě lépe na množinu dvojic **H** = (**P**,**e**), kde **P** = { $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$ } je množina podmínek, které musí být splněny, aby hráč zahlásil hlášku **e**, platí $\mathbf{p_1} \wedge \mathbf{p_2} \wedge \mathbf{p_3} \Rightarrow \mathbf{e}$. Dražební systém **DS** je množina těchto pravidel, **DS** = { $\mathbf{H_1}$,$\mathbf{H_2}$, ...,$\mathbf{H_n}$} a vyhodnocení by se, zcela formálně, provádělo nad výrokem $\mathbf{H_1} \vee \mathbf{H_2} \vee \ldots \vee \mathbf{Hn}$. Jakmile je pravidlo $\mathbf{H_x}$= ($\mathbf{P_x}$,$\mathbf{e_x}$)kladně vyhodnoceno, tak se hláška $\mathbf{e_x}$ zapíše do množiny **D**. Podmínky **p** se mohou ptát na existenci konkrétního **e** v množině **D**. Tento model pro názornost funguje, ale máme-li být technicky přesní a chceme-li respektovat reálnou strukturu použitou v programu Editor, tak jej musíme mírně pozměnit.

Model má jednu nevýhodu, nerespektuje přirozenou hierarchii dražebního systému. Dražební systémy se totiž běžně konstruují formou „odpovědí“ na zahájení či na konvence, jinými slovy – dražební systémy mají běžně stromovou strukturu. Pozměníme tedy definici pravidla, nově bude: **H** = (**P**,**e**,**o**), kde **o** je uspořádaná množina ve tvaru {$\mathbf{H_1}$, $\mathbf{H_2}$, $\mathbf{H_3}$ ..}, **o**∈**DS**. Nově přidaný parametr o je vlastně nová pozice, nový vrchol ve stromu, do kterého hráč při dražbě přejde – je to seznam odpovědí na hlášku reprezentovanou pravidlem **H**. Požadavek na uspořádání pravidel v odpovědích **o** je činěn čistě ze sémantických důvodů.



Obr. 2 – Reprezentace stromové struktury dražebního systému v aplikaci Edtior

Tento model by již měl fungovat dobře, stromová struktura byla zaimplementována jako základní princip aplikace Editor a to i z toho důvodu, že je pro bridžisty přirozená a značně snižuje nároky na zadávání podmínek uživatelem.

3.2.3 Reprezentace pravidel

Pravidlo **H** je tedy tvaru $\mathbf{p_1} \wedge \mathbf{p_2} \wedge \mathbf{p_3} \Rightarrow \mathbf{e}$. Každá z podmínek **p** může mít zcela jinou strukturu, může se dotazovat na dražbu, na karty v ruce, na nalezenou shodu u oponenta, jednoduše téměř na cokoliv. V Editoru1 bylo vše řešeno tak, že každá podmínka měla své explicitní vyjádření v grafickém rozhraní. Jakmile se autor setkal se situací, kterou neuměl řešit, potřebný dotaz na tuto situaci si přidělal. Byly přidávány podmínky tak, aby bylo možné zachytit nejzákladnější dražební obraty dle [13].

Při implementaci nového Editoru se nabízelo několik řešení. Bylo možné prostě jen přidat pár nových prvků do uživatelského rozhraní ve stejném duchu jako při implementaci starého Editoru. Problém ale byla nesourodost podmínek a to jak z implementačního, tak z uživatelského hlediska. Autor se tedy pokusil o to, co se mu nepodařilo v bakalářské práci a sice o vymyšlení nějakého jednotícího rámce pro zadávání podmínek. Pokusil se o vytvoření gramatiky podmínek.

3.2.4 Reprezentace podmínek – filtry

Ještě před popsáním konstrukce podmínek, je nutné připomenout koncept filtrů, který byl úspěšně použit již v autorově bakalářské práci [11]. Každému pravidlu **H** = (**P**,**e**) tedy přísluší množina podmínek **P**, které musí být splněny, aby hráč zahlásil hlášku **e.** V bridži je běžné, že jedno pravidlo reprezentuje více barevných hlášek najednou. Například zjednodušené pravidlo „draž nejdelší a nejdražší barvu na ruce“ může vést k různým hláškám. Formálně bychom mohli psát **H** = (**P**,{$\mathbf{e_1}$, $\mathbf{e_2}$, ..}). Samozřejmě, výše uvedené pravidlo by se dalo rozložit na několik separátních pravidel a v každém pravidle je možné mít podmínku pro právě jednu barvu stylu: „máš-li piků více než ostatních barev, tak draž piky“. Pravidla bychom potom seřadili sestupně od nejdražší barvy po nejlevnější a dostali bychom ekvivalentní zápis situace. Výsledný dražební strom by však byl až čtyřikrát košatější, protože odpovědi na podobně definované barevné hlášky jsou vždy stejné. Jinými slovy, psali bychom neustále to samé pro každou barvu zvlášť.

Proto je ve starém Editoru používán koncept filtrů tak, že nejprve se dle zadaných podmínek určí „vítězná“ barva pravidla a poté proběhne vyhodnocení ostatních pravidel. Může se stát, že filtry neprojde žádná barva nebo filtry projde víc barev naráz. V takovém případě pravidlo selhalo stejně, jakoby selhala některá z podmínek. Systém filtrů byl zcela oddělen od ostatních podmínek, záleželo na pořadí a vždy se filtrovaly všechny barvy. Odstranění konkrétní barvy z filtrované množiny muselo být provedeno explicitně přidáním filtru, který nepovoluje tuto barvu.

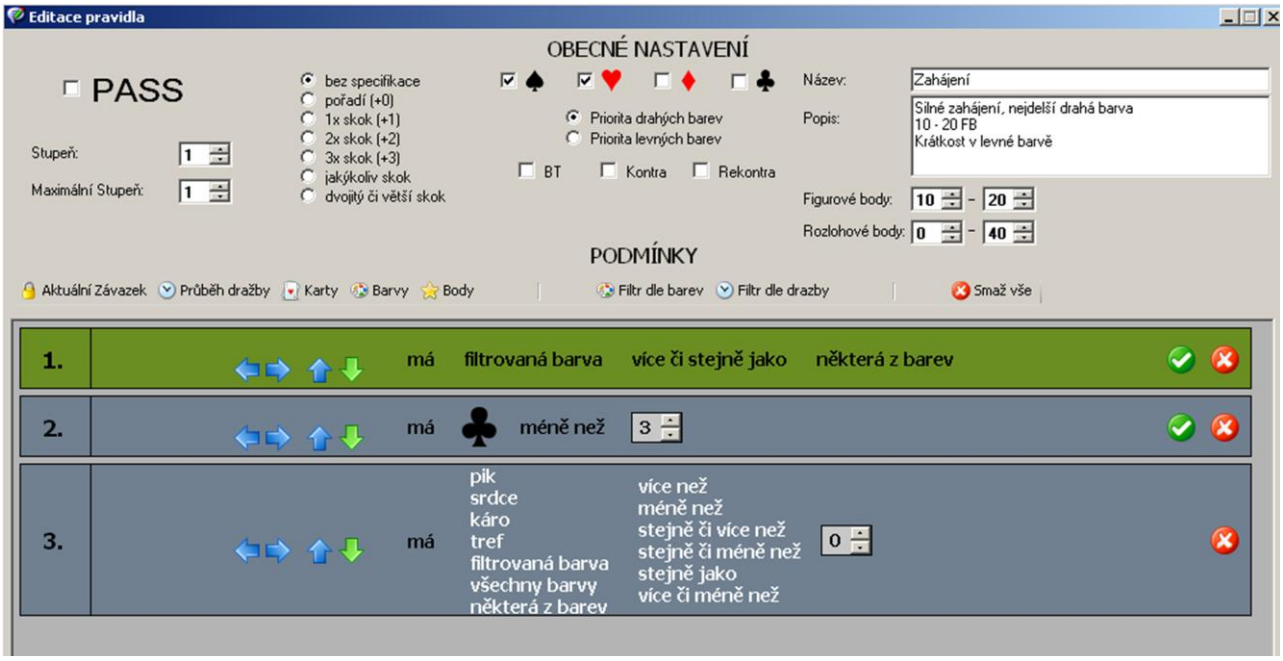
V novém Editoru systém filtrů zůstává, ale v mírně pozměněné podobě. Každé pravidlo povoluje explicitní nastavení konkrétních barev, kterých může nabývat dražená hláška **e**. Zvolením více jak jedné barvy se aktivuje nastavení filtrů, jež jsou realizovány naprosto stejným principem jako ostatní podmínky (na obr 3 jsou filtry vyznačeny zeleně). Rozdílem je, že už nezáleží na pořadí filtrů a barev, které projdou filtrovacími podmínkami může být více. Nastaveni „priorita levných barev“ a „priorita drahých barev“ resp. „priorita delší barvy“ a „priorita kratší barvy“ zajistí, že

z množiny profiltrovaných barev bude vybrána vždy právě jedna. Pokud nedojde k vybrání barvy (množina profiltrovaných barev bude prázdná), tak pravidlo selže. Po vybrání vítězné barvy se pokračuje dále ve vyhodnocení ostatních podmínek. V těchto podmínkách se může vyskytovat odkaz na profiltrovanou barvu.

3.2.5 Reprezentace podmínek - atomy

Nyní se dostáváme ke konstrukci samotných podmínek ze základních stavebních bloků gramatiky. Těmto základním blokům říkejme atomy, byť jsou to formálně terminální symboly gramatiky podmínek. Každá podmínka je složena z těchto atomů, tedy z terminálů, které uživatel vybere v editačním okně pravidla v Editoru. Celkem je rozlišováno pět základních druhů podmínek: podmínky týkající se barev, podmínky týkající se figurových bodů, podmínky týkající se dražby a podmínky týkající se konkrétního závazku.

Základním atomem, který se musí vyskytovat v každé podmínce, je atom KDO (na obrázku 3 je to vždy obrázek zcela vlevo se čtyřmi šipkami). Atom KDO reprezentuje množinu jednoho či více hráčů, každý hráč je reprezentován šipkou a šipky jsou vždy myšleny z pohledu dražitele, tzn. šipka dolů značí „já“, šipka nahoru mého partnera a levá / pravá šipka značí oponenty. Při vyhodnocování podmínky se za relativní hodnoty dosadí konkrétní hodnoty z pohledu aktuálního hráče. Atom KDO dále, v případě zaškrtnutí více jak jednoho hráče, umožňuje nastavit kvantifikátor ve stylu predikátové logiky. Kvantifikace „každý“ je ekvivalentem univerzálního kvantifikátoru ∀, „někdo z“ je existenční kvantifikátor ∃ . Dále umožňuji nastavit třetí a poslední typ „dohromady“, který reprezentuje součet znalostí o hráčích (např. „já mám víc figurových bodů než levý+pravý dohromady“).



Obr. 3 – Zadávání podmínek při editaci pravidla v programu Editor 2.0

Každá podmínka musí obsahovat atom KDO na prvním místě, nad hráči obsaženými v množině reprezentované primárním atomem KDO je pak prováděno vyhodnocení a získávání znalostí. Podmínka může (ale nemusí) obsahovat sekundární atom KDO na poslední pozici.

Dále každá podmínka, kromě podmínek dotazujících se na dražbu, musí obsahovat operátor porovnání (dále v textu je označen jako **op**), který určuje vztah nutný k platnosti podmínek. Dalším důležitým atomem je atom barvy, specifikující porovnávanou barvu. Atom barvy umožňuje, podobně jako KDO, nastavení kvantifikátoru ∀ (všechny barvy) a kvantifikátoru ∃ (některá z barev). Dále umožňuje nastavit hodnotu „filtrovaná barva“, která značí profiltrovanou barvu dle kapitoly 4.4. Byly popsány nejdůležitější atomy, ostatní používané atomy fungují v podobném duchu.

3.2.6 Reprezentace znalostí v bázi

Před popisem vyhodnocování samotného systému, je nutné zmínit, jakým stylem si hráč uchovává znalosti o pravdách okolního světa. Pro účely dalšího textu si označme Φ jako množinu všech hráčů, Φ = { **S**, **J**, **V**, **Z** } (**S** jako sever, **J** je jih, **V** východ a **Z** západ)

Každý hráč si ze svého úhlu pohledu buduje lokální (personální) znalostní bázi **B**. Nad touto bází pak probíhá vyhodnocení podmínek **p** z předchozí kapitoly. Lokální znalostní báze $\mathbf{B_s}$ je množina pravdivých tvrzení o světě z pohledu hráče **S**. Zpravidla platí, že $\mathbf{B_x} \neq \mathbf{B_y}$ a tudíž je možné, že podmínka **p** může být v bázi $\mathbf{B_x}$ vyhodnocena kladně a v bázi $\mathbf{B_y}$ záporně. Důvodem, proč jsou báze $\mathbf{B_x}$ a $\mathbf{B_y}$ rozdílné je fakt, že hráči **X** a **Y** mají různé karty na ruce, což v důsledku vede k tomu, že **X** nemusí pozorovanou situaci interpretovat stejně jako **Y**.

Příklad: Mějme podmínku **p = „**hráči **S** + **J** mají dohromady méně jak 8♥**“**. Může se stát, že **S** ani **J** nebudou moci dle své $\mathbf{B_s}$ respektive $\mathbf{B_j}$ určit, zda je podmínka splněna, ale o splnění podmínky bude vědět hráč **Z**, třeba proto, že má na ruce 6♥ (celkový počet karet od jedné barvy je 13). Podmínka **p** bude platná dle $\mathbf{B_z}$, ale nebude platná dle $\mathbf{B_s}$ a $\mathbf{B_j}$.

Pro získávání znalostí a pro řešení zpětného určení, ale potřebuji být schopen z pohledu hráče **Y** zjistit, zda mohl hráč **X** vyhodnotit **p** kladně v $\mathbf{B_x}$. Svojí bází $\mathbf{B_y}$ hráč **Y** použít nemůže, protože je zatížena personální znalostí vlastní ruky a výsledek vyhodnocení **p** dle $\mathbf{B_y}$ se, jak jsme ukázali v příkladu, nemusí shodovat s **p** dle $\mathbf{B_x}$. Z tohoto důvodu autor zavedl pátou, globální bázi znalostí **G**. Funguje na zcela stejném principu jako **B**, jen není zatížena lokálními znalostmi hráčů. Platí, že pokud **p** dle **G** = true, pak **p** dle $\mathbf{B_x}$ = true pro $\forall \mathbf{x} \in \Phi$. Do globální báze dále formálně přidávám i průběh dražby **D**.

Znalosti jsou reprezentovány intervaly (kromě **D**, **D** je seznam hlášek). Celočíselným intervalem <**a**,**b**> symbolizujeme nejistotu pozorovaného parametru **x** a platí, že $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$. Dále obecně platí, že vždy $\mathbf{a} \leq \mathbf{b}$. Je-li $\mathbf{a} = \mathbf{b}$ tak se jedná o jistou znalost a hodnota parametru $\mathbf{x} = \mathbf{a} = \mathbf{b}$. Na začátku hry se všem intervalům nastaví **a**=0 a **b**=max, kde max je maximální možná hodnota pozorovaného parametru (startovní hodnoty viz níže). V průběhu hry, se získáváním nových znalostí, se odhad zpřesňuje a interval „konverguje“ k reálné hodnotě parametru **x**. Vždy platí, že **a** roste (je neklesající) a **b** klesá (je nerostoucí).

V každé bázi $\mathbf{B_x}$ jsou uchovávány následující informace pro $\forall \mathbf{x} \in \Phi$, intervaly vpravo jsou startovní hodnoty:

a) figurové body celkem $<\mathbf{fbmax}, \mathbf{fbmin}>_{\mathbf{x}}$, startovní hodnota <0,40>

b) figurové body pro každou barvu ♣♦♥♠, např. <**fb♣min, fb♣max**>$_{\mathbf{x}}$, startovní hodnota <0,10>
c) délka každé barvy ♣♦♥♠, např. <**♣min**, **♣max**>$_{\mathbf{x}}$, startovní hodnota <0,13>
d) počty figur **AKQJ,** např. <**Amax**, **Amin**>$_{\mathbf{x}}$, startovní hodnota <0,4>

3.2.7 Provázanost znalostí

Znalosti v rámci báze **B** jsou svázány jistými omezujícími podmínkami vyplývajícími z logiky věci. Například, má-li hráč **S** jistá 4 esa, pak je zřejmé, že ostatní hráči mají 0 es. Vyplývá to z faktu, že v balíčku jsou celkem 4 esa. Tato podmínka tedy udržuje konzistentní intervaly stejného typu napříč hráči. Další příkladem může být situace, kdy jistě víme o hráči S, že nemá žádná esa. To ale znamená, že má maximálně 24 figurových bodů oproti možným 40 (eso = 4 body, král = 3 body, královna = 2 body, kluk = 1 bod). Tato podmínka udržuje konzistentní intervaly v rámci jednoho hráče. Je tedy důležité zajistit, aby znalosti v rámci báze zůstávaly konzistentní. První pravidlo je, že intervaly stejného typu musí zůstat konzistentní napříč hráči. Toto pravidlo využívá faktu, že každý druh intervalu má svůj maximální součet, který nemůže být překročen. Pravidlo platí pro všechny znalosti (figurové body, barvy i karty). Hodnota **max** je tedy maximální hodnota odvislá od typu intervalu (40 pro celkové figurové body, 10 pro figurové body v barvě, 13 je maximální počet karet v jedné barvě a 4 je počet karet jednoho typu v balíčku). Mějme interval <$\mathbf{a_x}$,$\mathbf{b_x}$> popisující celkové figurové body hráče **x**, kde $\mathbf{x} \in \Phi$ Pro $\forall \mathbf{x} \in \Phi$ platí:

a) $\sum_{\mathbf{y} \in \Phi,\ \mathbf{y} \neq \mathbf{x}} \mathbf{a_y} + \mathbf{b_x} < \mathbf{max}$
b) $\sum_{\mathbf{y} \in \Phi,\ \mathbf{y} \neq \mathbf{x}} \mathbf{b_y} + \mathbf{a_x} < \mathbf{max}$

Pokud podmínka a) resp. b) neplatí, tak položíme $\mathbf{b_x} = \mathbf{max} - \sum \mathbf{a_y}$ respektive $\mathbf{a_x} = \mathbf{max} - \sum \mathbf{b_y}$.

Druhé pravidlo vyjadřuje vztah karet k bodům. Označíme-li si intervaly reprezentující figury jako <**Amin**, **Amax**>$_{\mathbf{x}}$, <**Kmin**, **Kmax**>$_{\mathbf{x}}$, <**Qmin**, **Qmax**>$_{\mathbf{x}}$, <**Jmin**, **Jmax**>$_{\mathbf{x}}$ (A eso, K král, Q královna, J kluk) a interval celkových figurových bodů <**fbmin**, **fbmax**>$_{\mathbf{x}}$, tak pro $\forall \mathbf{x} \in \Phi$ platí:

a) 4***Amin** + 3***Kmin** + 2***Qmin** + **Jmin** ≤ **fbmin**
b) 4***Amax** + 3***Kmax** + 2***Qmax** + **Jmax** ≥ **fbmax**
c) **Amax** ≤ **fbmax**/ 4
d) **Kmax** ≤ **fbmax**/ 3
e) **Qmax** ≤ **fbmax**/ 2
f) **Jmax** ≤ **fbmax**

Pokud podmínka a) či b) neplatí, tak položíme $\mathbf{fbmin_x}$ = 4***Amin** + 3***Kmin** + 2***Qmin** + **Jmin** respektive **fbmax** = 4***Amax** + 3*$\mathbf{Kmax_x}$ + 2***Qmax** + **Jmax** . Pokud neplatí podmínky c) - f), tak položíme **Max** = **fbmax** / 4 respektive **Kmax** = **fbmax** / 3 respektive **Qmax** = **fbmax** / 2 respektive **Jmax** = **fbmax.**

A konečně třetí pravidlo vyjadřuje vztah figurových bodů v barvě vůči celkovým figurovým bodům. Označíme-li si intervaly reprezentující figurové body v barvě jako <**fb♣min**, **fb♣max**>$_{\mathbf{x}}$, <**fb♦min**, **fb♦max**>$_{\mathbf{x}}$, <**fb♥min**, **fb♥max**>$_{\mathbf{x}}$, <**fb♠min**, **fb♠max**>$_{\mathbf{x}}$, tak pro $\forall \mathbf{x} \in \Phi$ platí:

a) **fb♣min** + **fb♦min** + **fb♥min** + **fb♠min**≤**fbmin**
b) **fb♣max** + **fb♦max** + **fb♥max** + **fb♠max**≥**fbmax**

Pokud podmínka a) či b) neplatí, tak položíme **fbmin** = **fb♣min** + **fb♦min** + **fb♥min** + **fb♠min** respektive **fbmax**= **fb♣max** + **fb♦max** + **fb♥max** + **fb♠max**.

3.2.8 Vyhodnocení podmínky dle báze znalostí

Zde bude popsán způsob, kterým se vyhodnocuje podmínka dle báze znalostí **B**. Nyní nebudeme brát v úvahu dotazy na dražbu (množinu **D** dražených hlášek), ty jsou pro nás nezajímavé, neboť jsou to v podstatě jen dotazy na existenci prvku v množině **D**. Nás stále hlavně zajímá nejistota a intervaly.

Vyhodnocení podmínky **p** dle **B** neznamená nic jiného, než porovnání dvou intervalů. Ve speciálním případě můžeme porovnávat interval s číslem, ale vzhledem k tomu, že na číslo **a** lze nahlížet jako na interval <**a**,**a**>, tak můžeme s klidným svědomím zůstat u porovnávání dvou intervalů. Porovnáním se rozumí logické vyhodnocení výrazu <**a**,**b**> **op** <**c**,**d**>, kde **op** je operátor, **op**∈ {=, ≠,≤, ≥, <, >}. Výsledek této operace autor definoval poměrně striktně: <**a**,**b**>**op**<**c**,**d**> = **true** pokud pro ∀**x**∈<**a**,**b**>, ∀**y**∈<**c**,**d**> platí **x op y** = **true**. Jinými slovy, porovnání je vyhodnoceno kladně, pokud je výsledek zcela jistý. Například, aby mohlo být tvrzeno, že <**a**,**b**> ≤ <**c**,**d**>, tak musí platit **b** ≤ **c**. Vzhledem k výše (a níže) uvedeným vlastnostem platí zajímavý fakt, máme-li eval(<**a**,**b**> ≤ <**c**,**d**>) = false, pak nutně nemusí platit, že eval(<**a**,**b**> > <**c**,**d**>) = true. Například pokud je <**a**,**b**> uvnitř <**c**,**d**>.

Zde jsou uvedené všechny druhy porovnání intervalů a jejich převod na jednoduché porovnání dvou čísel:

a) <**a**,**b**> ≤ <**c**,**d**>, jinými slovy ∀**x**∈<**a**,**b**>, ∀**y**∈<**c**,**d**> platí **x** ≤ **y**, převedeme na **b**≤**c**
b) <**a**,**b**> ≥ <**c**,**d**>, převedeme na **a** ≥ **d**
c) <**a**,**b**> < <**c**,**d**>, převedeme na **b** < **c**
d) <**a**,**b**> > <**c**,**d**>, převedeme na **a** > **d**
e) <**a**,**b**> = <**c**,**d**>, převedeme na **a** = **b** ∧ **c** = **d** ∧ **a** = **c**
f) <**a**,**b**> ≠ <**c**,**d**>, převedeme na **a** > **d** ∨ **b** < **c**

3.2.9 Zpřesňování znalostí

Zbývá probrat poslední operaci, která je nad intervaly prováděna, a to upravování intervalů dle nově získaných znalostí. Dejme tomu, že se dozvíme, že podmínka **p**: <**a**,**b**> ≤ <**c**,**d**> určitě platí. Tedy víme, že musí platit ∀**x**∈<**a**,**b**>, ∀**y**∈<**c**,**d**> je **x** ≤ **y** . A tedy můžeme usoudit, že **b** ≤ **d** (pokud by **b** > **d** , tak bychom porušili podmínku o řádek výše a to tak, že za hledané **x** bychom dosadili **b** a tudíž ∀**y**∈<**c**,**d**> by platilo **x** > **y**) Důsledkem této dedukce je, že pokud **b** > **d** ,tak je možné zpřesnit interval <**a**,**b**> a položit **b** = **d**. Ekvivalentním způsobem lze vyvodit, že **a** ≤ **c** a v případě, že **a** > **c** položíme **a** = **c**. V následujícím textu je pro operaci zpřesnění intervalu dle podmínky **p** používáno označení update(**p**).

Následuje seznam všech typů zpřesnění intervalů, které autor uvedeným způsobem provádí. Pro názornost a přehlednost je u každého typu vždy upravován pouze interval <**a**,**b**>:

a) **p**: <**a**,**b**> ≤ <**c**,**d**> a pokud platí **b** > **d** tak položíme **b** = **d**
b) **p**: <**a**,**b**> ≥ <**c**,**d**> a pokud platí **a** < **c** tak položíme **a** = **c**
c) **p**: <**a**,**b**> < <**c**,**d**> a pokud platí **b** ≥ **d** tak položíme **b** = **d**-1
d) **p**: <**a**,**b**> > <**c**,**d**> a pokud platí **a** ≤ **c** tak položíme **a** = **c**+1
e) **p**: <**a**,**b**> = <**c**,**d**> převede se na $\mathbf{p_1}$: <**a**,**b**> ≤<**c**,**d**> a $\mathbf{p_2}$: <**a**,**b**> ≥ <**c**,**d**>

f) **p**: <**a**,**b**> ≠ <**c**,**c**> a pokud **a** = **c** položíme **a** = **c**+1, pokud **b** = **c** tak položíme **b** = **c**-1

g) **p**: <**a**,**b**> ≠ <**c**,**d**> a pokud **c** = **d** tak použijeme f)

3.2.10 Vyhodnocení systému a pravidel

Na dražební systém **DS** je nahlíženo jako na množinu odpovědí o tvaru **o** ={ $\mathbf{H_1}$, $\mathbf{H_2}$ …}, kde **H** = ({ $\mathbf{p_1}$, **..** } = **P**, **e**, {$\mathbf{H_x}$, ...} = **o**). Označme si nově množinu všech množin odpovědí **O**, **o**∈**O**. Vytvořme startovní množinu odpovědí **z** (zahájení), kdy **z** má stejný tvar jako odpověď, tzn. je to uspořádaná množina pravidel, **z** ={$\mathbf{H_1}$, $\mathbf{H_2}$, …}. Na množinu **z** není nikde v rámci podmínek **H** odkazováno, je to startovní množina každého systému, začínající pozice všech hráčů. Nově tedy platí: **DS** = **O** ∪ **z.**

Hráči si při průchodu systémem udržují svoji aktuální pozici.V našem formálním jazyce si vždy pamatují $\mathbf{o_{last}}$ náležící posledně kladně vyhodnocenému pravidlu **H.** V momentě, kdy je hráč při dražbě na tahu, tak může vybírat pouze z pravidel (hlášek) obsažených v množině odpovědí reprezentovaných jeho uloženou aktuální pozici $\mathbf{o_{last}}$. Může vybrat pouze odpovědi na předchozí hlášku. Na startu hry je všem hráčům jako jejich aktuálně uložená pozice nastaveno **z** dle jejich příslušných systémů, $\mathbf{o_{last}}$ = **z**. Spoluhráči v rámci jedné partnerské linie mají stejné **z**, protože využívají stejný **DS**.

3.2.11 Vyhodnocení a průchod dražebním systémem

Hráč draží jednoduchým způsobem. Prochází uspořádanou množinu $\mathbf{o_{last}}$ možných odpovědí (pravidel), které má k dispozici a u každého $\mathbf{H_x}$=($\mathbf{P_x}$,$\mathbf{e_x}$,$\mathbf{o_x}$) se pokouší splnit jeho podmínky $\mathbf{P_x}$ = {$\mathbf{p_{1x}}$,..,$\mathbf{p_{nx}}$}. První pravidlo $\mathbf{H_x}$, pro které se povede kladně vyhodnotit $\mathbf{p_1} \wedge \mathbf{p_2}$ = true, je vybráno jako vítězné, jeho $\mathbf{e_x}$ je přidáno na dražbu a hráčův $\mathbf{o_{last}}$ = $\mathbf{o_x}$. Partner hráče nyní provede zpětné určení dle $\mathbf{e_x}$, výsledkem mu bude nová hodnota $\mathbf{o_{last}}$ (viz kapitola 3.2.14). Hodnota $\mathbf{o_{last}}$ se u partnerů vždy shoduje, pokud tedy nedojde k omylu při zpětném určení.

Z výše nastíněného způsobu procházení stromem je zřejmé, proč je v autorově modelu důležité uspořádání pravidel v rámci množin odpovědí **o** respektive **z.** Jakmile se najde první pravidlo $\mathbf{H_x}$, tak všechna pravidla $\mathbf{H_y}$, **y**>**x** jsou zahozena. Může se stát, že hráč projde celou množinu $\mathbf{o_{last}}$ a není schopen najít žádné kladně vyhodnocené pravidlo. V takový moment je vybráno implicitní pravidlo $\mathbf{H_{default}}$ = ({true},pass,{}).

3.2.12 Vyhodnocení pravidel

Nyní se podíváme, jakým stylem probíhá vyhodnocení samotných pravidel $\mathbf{H_x}$=($\mathbf{P_x}$,$\mathbf{e_x}$,$\mathbf{o_x}$), $\mathbf{P_x}$ = {$\mathbf{p_{1x}}$,..,$\mathbf{p_{nx}}$}. Hráč **S** se snaží zjistit, zda platí pravidlo $\mathbf{H_x}$ tak, že provede $eval_{Bs}(\mathbf{p_{1x}} \wedge \ldots \wedge \mathbf{p_{nx}})$. Nejprve se vyhodnotí proměnná barva, je-li přítomna. Formálně to takto je správně, bohužel, zavedením proměnné barvy a vícebarevných hlášek se musí vyhodnocení pravidla rozdělit na dvě fáze.

Nejprve najdeme „vítěznou“ barvu, kterou pak použijeme jako argument pro hlášku e a také tuto barvu dosadíme za jakýkoliv případný výskyt terminálu „proměnná“ v normálních, nefiltrujících podmínkách. Pro všechny povolené barvy pravidla se dle stanoveného pořadí (od ♣ nahoru či od ♠ dolu, dle nastavení „priorita levných“ resp. „drahých barev“) zkusí vyhodnotit filtrovací podmínky dosazením iterované barvy za terminál „proměnná“. Pokud všechny barvy selžou, tak selže i celý

vyhodnocovací proces – pravidlo nelze vybrat. První úspěšná barva se nastaví jako vítězná barva. V tenhle moment už můžeme provést $eval_{Bs}(\mathbf{p_{1x}} \wedge \ldots \wedge \mathbf{p_{nx}})$

3.2.13 Vyhodnocení podmínek

Hráč **S** vyhodnocuje podmínku **p** tak, že za hodnoty všech atomů, ze kterých je podmínka složena, dosadí znalosti obsažené ve své bázi $\mathbf{B_s}$. Jak již bylo uvedeno v kapitole, každá podmínka obsahuje hlavní množinu hráčů, hlavní atom KDO, pro který je dotaz položen. Vyhodnocení probíhá tak, že se otestují všechny hodnoty intervalů (dle kap. 3.2.8) pro všechny hodnoty definované v podmínce. Výsledek je pak závislý na typu kvantifikátoru použitém v množině KDO. Jedná-li se o univerzální kvantifikátor, tak musí platit pro všechny hráče, existenční kvantifikátor definuje platnost pro alespoň jednoho hráče. Pokud je typ kvantifikátoru nastaven jako DOHROMADY, tak se jednoduše sečtou příslušné intervaly.

Uveďme příklad, ze kterého by mělo být vše zřejmé. Ostatní situace jsou řešeny obdobně. Hráč **S** vyhodnocuje dle své báze $\mathbf{B_S}$ podmínku **p**, která praví: „Někdo ze skupiny Já + partner má některou z barev delší než hráč po levici“. Nejprve dojde k dosazení identifikátorů hráčů dle pohledu, jakým je vidí **S**, protože podmínka hovoří relativně. Přepíšeme tedy podmínku na: „Někdo ze skupiny **S** + **J** má pikovou barvu delší než **V**“. Nyní si můžeme podmínku přepsat do konkrétní podoby: $\exists \mathbf{x}, \mathbf{x} \in \{\mathbf{S},\mathbf{N}\}: <\mathbf{fb\spadesuit min}, \mathbf{fb\spadesuit max}>_{\mathbf{x}} > <\mathbf{fb\spadesuit min}, \mathbf{fb\spadesuit max}>_{\mathbf{V}}$. Počítač nyní dosadí konkrétní hodnoty za oba intervaly a provede dva testy intervalů dle 3.2.8. Uspěje-li alespoň v jednom z nich, tak vrátí true. Ekvivalentním způsobem jsou řešena všechna porovnání podmínek. Pro účely dalšího textu si označme operaci vyhodnocení podmínky **p** dle báze $\mathbf{B_S}$ jako $eval_{Bs}(\mathbf{p})$. Zobrazení eval můžeme formalizovat jako $eval:(\mathbf{B},\mathbf{p}) \rightarrow \{true, false\}$. V textu bude pojednáno o negaci podmínky, která bude označena non(**p**). Negace je provedena dle pravidel predikátové logiky tak, že každý atom s kvantifikátorem (KDO a barva) změní svůj kvantifikátor na opačnou hodnotu (z $\forall$ se stane $\exists$ a opačně). Operátor **op** je převeden následovně: $\leq \rightarrow <$, $\geq \rightarrow >$ = $\rightarrow$!= a opačně.

3.2.14 Zpětné určení

Zpětné určení je termín, který autor používá pro pojmenování problému, s nímž se setkal již ve své bakalářské práci. Zpětné určení provádí partner dražitele k tomu, aby si aktualizoval svoji aktuální pozici v systému $\mathbf{o_{last}}$. Může se stát, že $\mathbf{o_{last}}$ nalezená zpětným určením není správná, tzn. neshoduje se s $\mathbf{o_{last}}$ partnera. V takovém případě se dražba vymkne z kolejí, hráči si přestanou rozumět, dojde pravděpodobně k vydražení nevhodného závazku a hráči si mohou naplnit své báze znalostí nepravdivými informacemi. Vhodnou tvorbou systému lze špatnému vyhodnocení předejít nebo se mu úplně vyhnout.

Popišme problém zpětného určení nově budovaným aparátem. Nechť probíhá dražba a na tahu je právě hráč **S**. Hráč **S** prochází pravidla obsažená v jeho $\mathbf{o_{last}} = \{\mathbf{H_1}, \mathbf{H_2}, \ldots, \mathbf{H_n}\}$. První pravidlo, které se mu podaří vyhodnotit kladně je pravidlo $\mathbf{H_x} = (\mathbf{P_x},\mathbf{e_x},\mathbf{o_x})$, tzn. podmínky obsažené v $\mathbf{P_x}$ byly dle $\mathbf{B_S}$ vyhodnoceny kladně. Hláška $\mathbf{e_x}$ se přidá do dražby **D** a hráč **S** si nastaví $\mathbf{o_{last}} = \mathbf{o_x}$. Hráč **N**, snažící se o zpětné určení, má k dispozici staré $\mathbf{o_{last}}$ hráče **S**, o kterém předpokládáme, že bylo při minulém zpětném určení určeno dobře. Úkol hráče **N** je najít $\mathbf{H_y} \in \{\mathbf{H_1}, \mathbf{H_2}, \ldots, \mathbf{H_n}\}$ tak, aby $\mathbf{H_y} = \mathbf{H_x}$. Problém je v tom, že jedinou informaci, kterou **N** od **S** získal je globální informace

přidání $\mathbf{e_x}$ do **D**, co když byla ale některá z podmínek $\mathbf{P_x}$ vyhodnocena kladně díky lokální bázi $\mathbf{B_S}$? V jednotlivých pravidlech **H** se navíc mohou hodnoty **e** libovolně opakovat, tzn. může existovat víc různých **H** se stejnými **e** (viz příklad pro **e** = (2,♥) na konci sekce 2.3).

Jediným způsobem pro **N**, jak se pokusit najít hodnotu $\mathbf{H_x}$, je začít procházet poslední známé $\mathbf{o_{last}}$ pro **S** a u hlášek $\mathbf{H_y}$ takových, že $\mathbf{e_y} = \mathbf{e_x}$, se pokusit vyhodnotit podmínky $\mathbf{P_y}$. Použít svoji bázi $\mathbf{B_N}$ ale nemůže, protože zcela jistě obsahuje jiná data než $\mathbf{B_S}$. Ze stejného důvodu nemůže použít ani **G**, protože by vyhodnocení mohla uniknout některá z podmínek $\mathbf{P_y}$, kterou **S** ve své lokální bázi vyhodnotí kladně. Ve snaze vyřešit uvedený problém, použil autor opačný přístup, a sice pokusil se najít pravidlo, které nejde zamítnout. Hráč **N** tedy určí za $\mathbf{H_x}$ první pravidlo $\mathbf{H_y}$ ($\mathbf{e_y}$= $\mathbf{e_x}$), pro které nevyhodnotí eval(non($\mathbf{P_y}$)) = true, tzn. eval(non($\mathbf{p_{1y}}$) ∨ … ∨ non($\mathbf{p_{ny}}$)) = false. Vyhodnocování non($\mathbf{P_y}$) musí samozřejmě probíhat dle **G**, jinak se vystavujeme riziku, že bude vyhodnocení ovlivněno lokální znalostí hráče a mohlo by být falešně pozitivní.

3.2.15 Získávání znalostí

Nyní bude vysvětleno, jakým způsobem a kdy se aktualizují znalosti hráčů uložené v bázi **B** a globální znalosti **G**. Znalosti jsou důležitým aspektem celého konceptu. Využívají se hlavně při dražbě, kdy na základě znalostí je schopen hráč dělat přesnější a přesnější rozhodnutí a vlastně je vůbec schopen pokročileji dražit a reagovat na informace získané od partnera. Znalosti se dále používají v algoritmu sehrávky, kde slouží k omezení množiny náhodně generovaných rozdání Monte Carlo vzorkováním. Hráč vstupuje do hry bez jediné znalosti o světě, pozorováním okolního světa si své znalosti rozšiřuje.

3.2.16 Kdy se znalosti získávají

Momenty, kdy hráč získává novou znalost jsou tyto: a) hráč se poprvé podívá na svou ruku, b) dražení hlášky hráčem **X**, $\mathbf{X} \neq \mathbf{S}$, c) odkrytí karet tichého hráče **T**, d) hráč **X** nese kartu na stůl. Všechny tyto momenty zároveň aktualizují globální znalost **G**, samozřejmě až na moment a). Momenty a) a c) jsou pro nás nezajímavé. Jedná se totiž o jistou znalost, tzn. příslušné intervaly mohu upravit na konkrétní číslo, čímž odstraním z rozhodování veškerou „nejistotu“ nad tímto intervalem. Například, vidím-li, že tichý hráč **T** má dvě esa, tak vlastně vidím, že platí podmínka **p**: $\langle \mathbf{Amin}, \mathbf{Amax} \rangle_{\mathbf{T}} = \langle 2,2 \rangle$ a já mohu interval dle kapitoly 3.2.9 zpřesnit tak, že **Amin** = **Amax** = 2. Samozřejmě, kdykoliv přibude v bázi nějaká nová znalost (tzn. nějaký interval je zpřesněn), tak se automaticky provede kontrola konzistence znalostí dle jejich provázanosti 3.2.7, toto budu nadále brát jako automatický proces.

3.2.17 Získávání znalostí během sehrávky

Nepočítáme-li odkrytí karet tichého hráče, tak jediný moment, kdy hráč získá novou informaci během sehrávky, je samotné „pokládání“ karet na stůl. V ten moment si každý hráč může upravit svojí bázi znalostí **B** dle pozorované karty. Následně se, jako vždy při jakémkoliv zpřesnění intervalu, zkontrolují a případně upraví ostatní znalosti za účelem dodržení jejich vzájemné provázanosti (3.2.6). Hraní karet při sehrávce je jediná pozorovatelná činnost, kdy informace ze systému „mizí“ a hráči si

tedy zmenšují jak horní, tak dolní hodnoty intervalů reprezentujících znalosti v **B**. Například, hráč **S** hraje eso a o hráči **S** víme (z průběhu dražby), že jeho $<\mathbf{Amin},\mathbf{Amax}>_{\mathbf{S}} = <1,3>$. Z toho vyplývá, že data můžeme upravit na novou hodnotu $<\mathbf{Amin},\mathbf{Amax}>_{\mathbf{S}} = <0,2>$.

3.2.18 Získávání znalostí z podmínek

Zisk znalosti z podmínky se provádí dle pravidel v 3.2.9 a je proveden pouze pro hráče uvedené v hlavním atomu KDO, tzn. v bázi se upraví intervaly hovořící pouze o hráčích v primárním atomu KDO. Pokud se nedá jednoznačně určit, jakým způsobem mají být intervaly zpřesněny, tak se neprovede nic. Například, víme-li, že podmínka $eval_{Bs}(\mathbf{p})$ = true a **p** zní: „Já nebo partner máme 5 piků“, tak můžeme zpřesnění provézt pouze tehdy, zjistíme-li, dle jaké informace hráč S usoudil $eval_{Bs}(\mathbf{p})$ = true.

Na podmínku **p** dle 3.2.13 nahlížíme jako na: $\exists \mathbf{x},\ \mathbf{x}\in\{\mathbf{S},\mathbf{N}\}$: $<\mathbf{fb\spadesuit min},\ \mathbf{fb\spadesuit max}>_{\mathbf{x}} = 5$, **p** si ale můžeme rozložit na dvě podmínky $\mathbf{p_1}$ a $\mathbf{p_2}$, kde $\mathbf{p_1}$: $<\mathbf{fb\spadesuit min},\ \mathbf{fb\spadesuit max}>_{\mathbf{S}} = 5$, $\mathbf{p_2}$: $<\mathbf{fb\spadesuit min},\ \mathbf{fb\spadesuit max}>_{\mathbf{N}} = 5$. Podaří-li se nám dokázat, že právě jedna z podmínek $\mathbf{p_1},\mathbf{p_2}$ nemůže ve světě platit, tak je jasné, že se hráč **S** musel při svém rozhodování postupovat dle podmínky druhé. Pokusíme se tedy vyhodnotit negaci podmínky dle globální znalosti, pokud uspějeme právě u jedné (pokud uspějeme u obou, někde se stala chyba) tak můžeme zpřesnit svou bázi znalostí dle druhé, neúspěšné podmínky. Tzn.: $eval_G(non(\mathbf{p_1})) = true \wedge eval_G(non(\mathbf{p_2})) = false \Rightarrow update(\mathbf{p_2})$. A obdobně $eval_G(non(\mathbf{p_2})) = true \wedge eval_G(non(\mathbf{p_1})) = false \Rightarrow update(\mathbf{p_1})$. Pokud **p** obsahuje sekundární množinu KDO, tak se vyhodnocení provede i pro tuto skupinu zrcadlovým otočením operátoru **op** obsaženého v podmínce **p** a zaměněním sekundární množiny KDO za primární množinu. Stejným způsobem je prováděno získávání znalostí ze všech typů podmínek **p** obsažených v pravidlech.

3.2.19 Získávání znalostí během dražby

Ocitáme se v momentě, kdy hráč **S** zahlásil hlášku $\mathbf{e_x}$ a my jsme zpětným určením našli hráčem **S** vybrané pravidlo $\mathbf{H_x}$. Víme tedy, že podmínky $\mathbf{P_x}$ byly kladně vyhodnoceny dle $\mathbf{B_s}$. Hráč **N** využije znalost, že všechny podmínky $\mathbf{P_x}$platí a provede update(**p**) pro$\forall \mathbf{p}\in\mathbf{P_x}$. Pokud se v podmínce vyskytuje proměnná barva, tak se za ní dosadí barva z $\mathbf{e_x}$.

To ale není všechno. Víme, že **S** vybral pravidlo $\mathbf{H_x}$, tzn. $eval_{Bs}(\mathbf{P_x})$ = true. Pak ale víme, že u pravidel $\mathbf{H_y},\mathbf{y}<\mathbf{x}$, tj. pravidel, pro které se **S** nerozhodl, musí $\forall \mathbf{P_y}$ platit: $eval_{Bs}(non(\mathbf{p_{1y}}) \vee \ldots \vee non(\mathbf{p_{ny}})) = true$. Jinak řečeno, můžeme těžit z vědomosti, že bylo dané pravidlo zamítnuto. Musíme ale najít pravý důvod, proč se tomu tak stalo - jednu konkrétní podmínku **p**, která při vyhodnocování selhala. Přesněji, abychom mohli vytěžit nějakou novou informaci, musíme najít právě jednu podmínku, u které nemůžeme dle **G** určit, že platí. A to se nemusí nutně povést. Zkusíme tedy postupně pro $\forall \mathbf{P_y},\ \mathbf{y}<\mathbf{x}$ zjistit, zda (právě jedna) $!\exists \mathbf{p}\in\mathbf{P_y}\ eval_G(\mathbf{p}) = false$, ale pro všechny ostatní podmínky $\forall \mathbf{q}\in\mathbf{P_y},\ \mathbf{q}\neq\mathbf{p}\ eval_G(\mathbf{q}) = true$. Pokud takové **p** najdeme, našli jsme pravidlo, které muselo selhat – všechna ostatní pravidla jsme vyhodnotili přes globální znalost, takže i hráč **S** je musel vyhodnotit stejně. Zbývající podmínka **p** musela selhat, jinak by hráč **S** musel zvolit pravidlo $\mathbf{H_y}$. Můžeme tedy pustit update(**p**).

4. Rozbor sehrávky

4.1 Popis problému sehrávky

Sehrávka následuje ihned po dražbě. Hráči se střídají v odehrávání karet systémem „větší bere“ a snaží se o (ne)naplnění vydraženého závazku (viz kapitola 2). Tichý hráč (angl. dummy player) pokládá po úvodním výnosu karty na stůl lícem nahoru, takže je všichni ostatní hráči mohou vidět. Každý hráč během sehrávky má, kromě situace před úvodním výnosem, informace o polovině všech karet ve hře (vidí karty tichého hráče a svoje vlastní karty). K úplné znalosti by mu stačilo vidět ještě karty jednoho hráče a zbývající karty by si dopočítal. Stačilo by tedy, aby se ve hře objevil druhý tichý hráč, odtud anglické označení double-dummy.

Double-dummy úloha je bridžová úloha s odkrytými kartami (je to tedy hra s úplnou informací), kde cílem je najít nejlepší herní strategii, která maximalizuje zisk zdvihů. V podobném duchu se tedy klasický bridžový problém označuje jako single-dummy úloha (tentokrát už se jedná o hru s neúplnou informací). A nakonec se někdy vyskytuje termín zero-dummy úloha, což je vlastně situace, kdy hráč před úvodním výnosem vidí pouze karty na svojí ruce.

Pravděpodobně nejtěžším momentem sehrávky, i pro lidského hráče, je právě úvodní výnos. Profesionální bridžové programy mají většinou mírně pozměněné herní strategie pro tahy obránců či pro úvodní výnos. Najít vhodný úvodní výnos je zatím asi nejsložitějším bridžovým problémem a v bridžové (nepočítačové) literatuře najdeme mnoho knih, které se výhradně zabývají jen problematikou úvodních výnosů.

4.2 Jak řeší sehrávku bridžový software

Asi nejrozšířenější a nejúspěšnější postup, který zpopularizoval svými úspěchy na mistrovství počítačového bridže [14] program GIB (Ginsberg’s Intelligent Bridgeplayer [18]), je převedení sehrávky na množinu double-dummy problémů. Ty se pak vyřeší minimax algoritmem a karta, která byla nejčastěji vybrána jako vítězná, se zahraje. Převedení na double-dummy je provedeno Monte-Carlo vzorkováním koherentním s dosud získanou informací. Kvalita tohoto postupu je přímo úměrná kvalitě implementace double-dummy algoritmu.

Double-dummy úlohu už lze řešit konvenčními metodami pro řešení problémů s úplnou informací. Základní stavební kameny byly položeny již v minulém století prací Ming-ShenChanga[8]. Jeho idey doplnil a úspěšně implementoval Ginsberg [4],[5],[6]. Klasický minimax algoritmus s alfa-beta prořezáváním je doplněn o transpoziční tabulky, eliminující symetrické tahy [8]. Transpoziční tabulka je v podstatě hashovací tabulka, která má za úkol řešit zrcadlovou povahu her. V bridži, stejně jako například v šachu, totiž nezáleží na tom, v jakém pořadí jsou provedeny dva tahy za předpokladu, že vedou do stejné situace. Dle Changa se klíč do hashovací tabulky zkonstruuje jako šestnáctimístné číslo, složené ze dvou čísel reprezentujících již odehrané figury v barvě (AKQJ, 16 míst) a z druhého čísla reprezentující odehrané „středně“ velké karty (16 míst, karty 10,9,8,7). Výsledný klíč získáme provedením operace XOR na obě hodnoty. Záznamem v tabulce je aktuální herní situace, tzn. číslo hráče na tahu, počet kol do konce hry a počet uhraných zdvihů. Každá prozkoumaná situace je zahashována a při vstupu do nového vrcholu herního stromu se prozkoumá, zda už nebyla někdy vyřešena. Pokud ano, vrátí se výsledek z tabulky a situace se dále neřeší.

Dalším důležitým krokem je aplikace heuristických prořezávání. Příklad některých prořezání dle Changa:

- Posloupnosti karet ve stejné barvě (např.: KQJ10) se může považovat jako jedna karta. Ostatní karty z posloupnosti budou mít při zahrání stejný efekt.
- Když je nejvyšší karta hráče P menší, než nejnižší karta libovolného hráče, který ještě nenesl kartu, pak z pohledu hráče P zahrajeme pouze jednu kartu a ostatní prořežeme.
- Ze všech karet, které nejsou schopny přebít aktuálně nejvyšší kartu na stole, se vybere pouze jedna karta.
- Malé karty (9 a níže) lze považovat za ekvivalentní.

Dalším zlepšením, které představil GIB, je metoda zvaná Partition Search vymyšlená a popsaná Ginsberkem [5]. Metoda spočívá v tom, že tahy, které očividně nevedou k úspěchu, nejsou vůbec prozkoumávány a je realizována na úrovni transpozičních tabulek (kóduje podobné situace do stejné). Bohužel, jakým způsobem zakódovat bridžovou situaci tak, aby se dvě významově (herně) podobné situace přeložily do stejného klíče tabulky, se autorovi diplomové práce nepodařilo zjistit ani vymyslet. Ginsberk bridžovou úlohu takto řešil, ale podrobněji se o tom nezmiňuje. Přiznává však, že aby šel tento koncept použít, je třeba nějakým netriviálním způsobem herní stav reprezentovat a většinou to vyžaduje expertní znalost samotné hry.

Pozoruhodná implementace pro řešení double-dummy úlohy využívající neuronové sítě se začíná rodit v Polsku [9]. Autoři tvrdí, že nejlepší sítě dokáží správně odhadnout maximální počet získatelných zdvihů zhruba na jedné třetině rozdání. Povolíme-li chybu jednoho zdvihu, tak se výsledek zlepší na zhruba 80%. Pouze 5% rozdání vykazuje chybu sítě o více jak dva zdvihy. Tento výzkum je poměrně aktuální, autoři nyní pracují na dalším zlepšení sítí na double-dummy problémech, další na řadě je dražba a nakonec single-dummy problém.

Momentálně nejlepší implementaci algoritmu pro řešení double-dummy úlohy vytvořil Williamem Bailey a jmenuje se DeepFinesse [16]. Spekuluje se, že W.Bailey pro svůj algoritmus použil výše uvedený přístup zkombinovaný s nadstandardně agresivním heuristickým prořezáváním. DeepFinesse potřebuje v průměru 6 sekund na analyzování double-dummy úlohy. Bohužel, W.Bailey zatím nepublikoval žádný článek o jeho implementaci.

Jak již bylo řečeno, většina software řeší single-dummy úlohu rozložením na double-dummy úlohy. Zřejmě nejúspěšnější program přistupující k single-dummy problému jinak, je program Bridge Baron[15], který se zhruba před deseti lety pravidelně umisťoval ve finále klání elektronického bridže. Bridge Baron využívá HTN plánování a více probabilistický přístup. Bridge Baron definuje základní bridžové akce, které může hráč provést. Tyto akce vždy v konkrétní situaci dekomponuje na atomické úkony, z nichž postaví síť, kterou ještě doplní o možné výsledky akcí. Jednotlivé listy této sítě ohodnotí pravděpodobností výskytu a „užitkovou“ hodnotou. Z množiny možných akcí vybere ten, který má nejvyšší vážený průměr z možných „užitkových“ hodnot svých listů (více v [12] či [11]).

Dalším přístupem, který jde v podobném duchu jako Bridge Baron, je snaha o konstrukci herního stromu z možných herních „taktik“ namísto hraní konkrétních karet. Basin a kolektiv autorů ve své práci [3] identifikuje celkem 7 těchto strategií, které expertní hráč při hře využívá. Autoři proklamují, že jejich řešení dosahuje úrovně expertního hráče. Bohužel, tento princip testují na zjednodušeném single-

dummy problému, omezeném na pouze jednu barvu (tzn. uvažují pouze 13 karet jedné barvy) a tudíž je použití v reálném software zatím v nedohlednu.

Sumář implementací a pokroků za posledních několik let nabízí ve svém článku P. Nyu [10]. Nejúspěšnější bridžové programy současnosti jsou programy SharkBridge [20], Jack Bridge [18] a Wbridge5 [21]. Tyto programy zvítězily na posledních kláních počítačového bridže WCBC [14]. Bohužel, protože se jedná o komerční software, tak jejich implementace není přesně známá. Pravděpodobně se opět jedná o Monte Carlo metody [10] a v podstatě jen o zlepšení konceptu poprvé představeného v GIB.

4.3 Jak sehrávku řešil autor v Simulátoru 1.0

Označením „Simulátor 1.0“ budeme myslet program Simulátor vypracovaný v rámci autorovy bakalářské práce (pro detailní popis viz [11]). Autor se rozhodl řešit sehrávku (single-dummy úlohu) Monte Carlo simulací rozložením hry na konkrétní double-dummy úlohy. Na dílčí double-dummy úlohy je již možno použít konvenční metody. Hlavní důvod je prozaický – ostatní řešení vyžadují k uspokojivému fungování řadu heuristických úvah a zlepšení vycházející z expertní znalosti bridžových pravidel a zákonitostí. Monte Carlo simulace, podpořená klasickým prohledáváním herního stromu, by měla fungovat alespoň uspokojivě dobře, bez nutnosti algoritmus nějak výrazně upravovat a přizpůsobovat různým nepravidelným rozdáním a výjimkám v herním světě. Výhodou tohoto přístupu je i to, že funguje stejně i na úvodní výnos hráče, tzn. nerozlišuje rozdíl mezi single-dummy a zero-dummy. Toto řešení je navíc ověřeno řadou úspěšných implementací.

Základní strukturou pro držení karetních informací v algoritmu implementovaném v Simulátoru 1.0 byly cyklické spojové seznamy s hlavou. Pro každou barvu je držen vlastní seznam karet. Operace, které je nutné provádět nad těmito strukturami, jsou: iterace, přidávání a odebrání prvku ve struktuře a získání nejmenšího a největšího prvku (next, first, last, add, delete). Zvolená struktura je pro toto vskutku ideální, první a poslední prvek získáme přímo přes hlavu seznamu, iteraci provádíme se zarážkou reprezentovanou hlavou a přidávání a odebírání provedeme jako jednoduché připojení resp. odpojení prvku v seznamu. Přidávání a odebírání prvků bylo navíc vždy prováděno na iterovaném místě, nemuselo tedy předtím docházet k průchodu seznamu a hledání místa k přidání resp. odebrání prvku. Každá operace má tedy konstantní složitost O(1). Jedna konkrétní karta byla reprezentována objektem, který v sobě udržoval informaci o barvě a hodnotě karty.

Double-dummy algoritmus samotný byl implementovaný jako klasický rekurzivní minimax algoritmus. Docházelo tedy k průchodu stromu herních situací, jednotlivé situace byly jednoznačně ohodnoceny počtem získaných zdvihů. Algoritmus byl doplněn o transpoziční tabulky a řadu heuristických prořezávání zmíněných v [6] a [8]. Celková rychlost algoritmu byla dostačující pro simulaci sehrávky, nebyla však úplně ideální. Starý algoritmus byl schopen prohledat herní strom do hloubky 4 tahů, tzn. 12 odehraných karet v řádu několika stovek milisekund. Maximální hloubka stromu, kterou byl starý algoritmus schopen uspokojivě řešit, byla hloubka 5 tahů (16 karet). Zde se výpočet uskutečnil v řádu několika vteřin. Hloubka 6 a více byla už časově zcela mimo schopnosti starého algoritmu.

4.4 Jak sehrávku řeší autor v Simulátoru 2.0

Když autor řešil implementaci nového algoritmu sehrávky s odkrytými kartami, měl v podstatě dvě základní cesty, kterými se vydat. Buď mohl být vyzkoušen zcela jiný přístup jak algoritmus řešit (zvažovány byly zejména neuronové sítě, ale například i naivní Bayesovský klasifikátor) nebo bylo možné se pokusit o zlepšení a preciznější implementaci algoritmu starého. Zvolena byla druhá cesta hlavně proto, že jakékoliv experimenty by sice mohly přinést zajímavé výsledky, ale autorovo očekávání bylo v tomto ohledu spíše negativní. Rychlé řešení double-dummy problému je totiž otázkou na úrovni samostatné práce a protože autorova diplomová práce se zabývá i jinými věcmi, než algoritmem sehrávky, rozhodl se vydat jistou cestou. Hlavním požadavkem bylo, aby algoritmus hrál rozumně dobře už jen kvůli možnosti otestovat více dražebních systémů proti sobě. Se špatným algoritmem by byly výsledky značně zkreslené.

Single-dummy úloha se tedy obdobným způsobem jako v Simulátoru 1.0 rozloží na double-dummy úlohy tak, že se z neznámých karet vygeneruje konkrétní rozložení, které pro daného hráče neodporuje jeho znalostem uloženým ve znalostní bázi. Na toto rozložení je spuštěn double-dummy algoritmus a spočtený výsledek je uložen. Monte Carlo vzorků je vygenerováno několik (autor se konkrétně snaží generovat 20 vzorků, vše je ale omezeno časovým limitem 5 vteřin, aby hráč nečekal příliš dlouho) a z nalezených výsledků je vybrán ten, který se vyskytuje nejčastěji. Tato metoda bude tím úspěšnější, čím přesnější jsou znalosti hráče v bázi.

4.5 Double-dummy v Simulátoru 2.0

Prvním krokem bylo revidovat použitou datovou strukturu k uchování karetních informací. Cyklické spojové seznamy použité v Simulátoru 1.0 se jeví jako ideální, alespoň co se teorie složitosti týče. Bylo však jasné, že samotná iterace, porovnávání a i mazání/přidávání prvků v seznamu, byť se jedná o O(1) operace, s sebou nese jistou režii. Autor totiž již od počátku přemýšlel o reprezentaci karty coby jednoho třináctimístného binárního čísla, s právě jednou jedničkou reprezentující hodnotu karty.

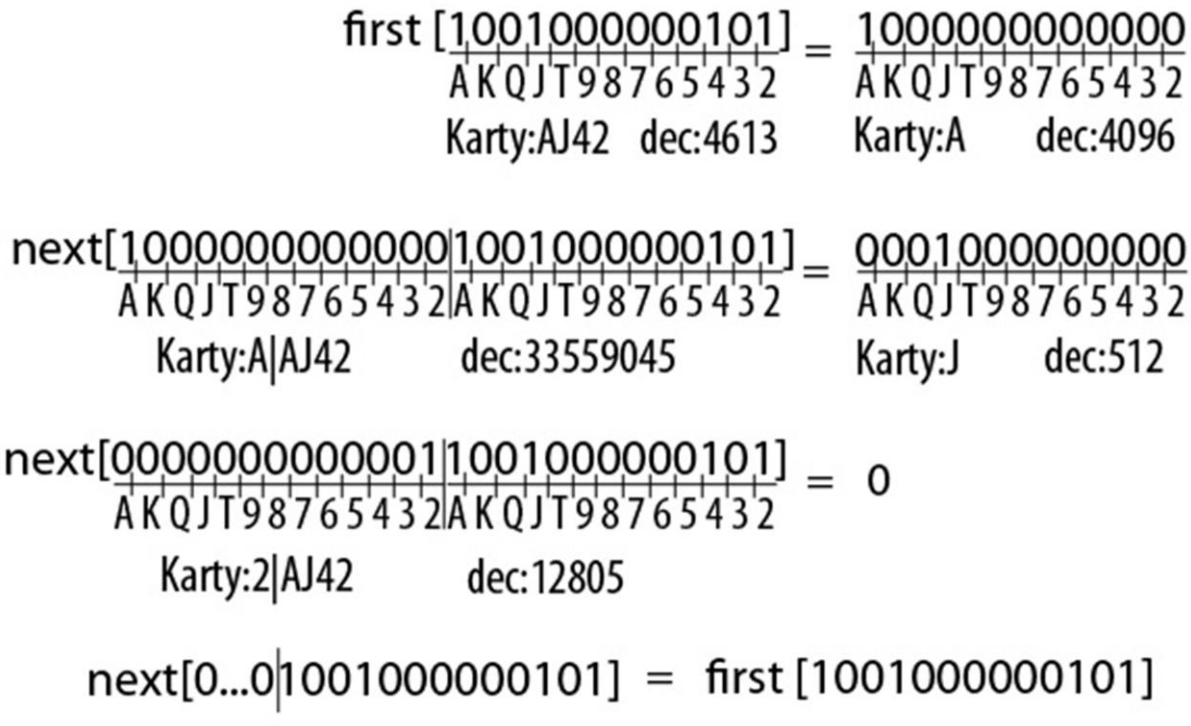
Binární reprezentace jedné karty má hlavní výhodu v tom, že všechny karty jedné barvy se dají reprezentovat opět jako třináctimístné binární číslo, protože v rámci jedné barvy existuje vždy 13 různých karet. Například list AJ42 je reprezentován jako číslo 4613, v binárním zápise 1001000000101. Ruka jednoho hráče je tedy reprezentována čtyřmi čísly pro každou barvu. Nejvyšší bit byl zvolen jako eso záměrně proto, aby bylo jednoduší a intuitivnější vzájemné porovnávání karet. Operace přidání a odebrání prvku nejsou žádný problém – k jejich implementaci se použijí rychlé binární operace OR a AND, OR pro přidání a AND pro odstranění. Autor očekával, že operace insert a delete, implementované použitím binárních AND a OR, budou rychlejší než v případě přepojování spojového seznamu.

Vyvstal však problém při hledání rychlé implementace operace next (last a first už jsou podružné). Jinými slovy, mějme binární číslo předem známé fixní délky. Otázka je, jak co nejrychleji určit index první jedničky (nejvíce vlevo)? Samozřejmě, číslo bychom mohli iterovat přes všechny číslice, dokud bychom jedničku nenalezli. Problém je, že ve zvolené reprezentaci bude mít většina čísel na většině míst nuly. Mnoho průchodů přes nuly tedy budeme provádět zcela zbytečně a nastávala by

otázka, jaké zlepšení (či dokonce zhoršení) by tento přístup přinesl oproti spojovým seznamům, kde byla operace first prováděna v konstantním čase.

Autor se nejprve marně pokoušel vyřešit tento problém vhodnou kombinací binárních operací. Nakonec, protože možných kombinací karet od jedné barvy je relativně málo, se rozhodl všechny možné iterace (volání operace next, first, last) předpočítat do pole. Pro operaci first (respektive last) bude indexem do pole číslo reprezentující list karet v barvě a hodnota v poli bude nejvyšší resp. nejnižší karta (číslo s právě jednou jedničkou) v listu obsažená. Pole first bude mít 2^{13}(8192) záznamů a všechny budou využity. Pole last je konstruováno obdobně.

Ještě bylo třeba vyřešit operaci next, která má fungovat jako iterace rukou (číslem) a proto je třeba „volání“ next doplnit o nějaký druh iterátoru. Index do pole bude tedy doplněn o „kód“ předchozí nalezené karty a výstupem bude následující karta v listu obsažená. Pokud je návratová hodnota operace next nulová, tak iterace došla na konec listu. Autor nakonec pole reprezentující operaci next doplnil o hodnoty uložené v poli first, toto lze bezpečně udělat, protože průnik polí first a next je nula. Jinými slovy, operace first je reprezentována voláním operace next, kde kód předchozí karty je 0. Pole next bude mít velikost 2^{26} (67108864) a ne všechny záznamy budou využité. Příklady operace next jsou znázorněny na obrázku č.4. (Pozn.: reálná implementace nakonec využívá pouze 221184 záznamů. Iterátor, coby „kód“ předchozí nalezené karty, má totiž jen 13 hodnot a v reálné implementaci dochází k převedení decimálního čísla karty (1 – 4096) na hodnoty 1-13.)



Obr. 4 – Ukázka realizace operací first a next v Simulátoru 2.0. Next je předpočítané pole čísel pro všechny možné iterace. T je zkrácené označení pro kartu hodnoty 10.

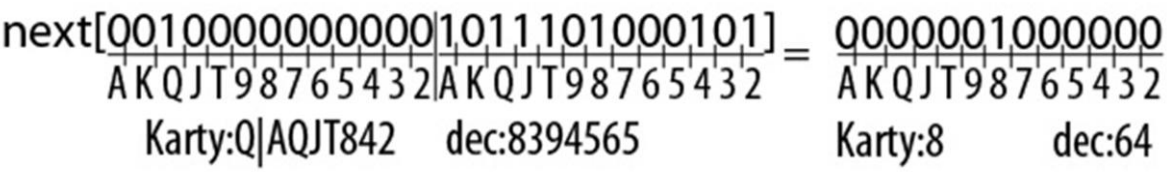
Další krok, který měl vést ke zlepšení algoritmu, byl nahrazení rekrurze `while` cyklem. Volání funkcí obecně s sebou nese určitou režii při alokaci nové paměti a

kopírování argumentů na zásobník. `While` cyklus je časově úspornější ačkoliv výsledný kód je možná o něco méně přehledný. Na podobné téma nalezl autor zajímavou úvahu. Některé problémy se, dle tvrzení J.D. Allena[1], dají zpřehlednit a zrychlit v hodným použitím příkazu `goto`. Allen tento postup demonstruje právě při nahrazování rekurze `while` cyklem za účelem dosažení maximální rychlosti. `Goto` používá například k předčasnému vyskočení z několikanásobně zanořených cyklů na zcela jinou pozici v algoritmu. Autor této diplomové práce `goto` použít nehodlal, ale Allenův přístup jej utvrdil v tom, že by k odstranění rekurze mělo dojít.

V tento moment, ještě před aplikováním prořezávání a transpozičních tabulek, se autor rozhodl algoritmus implementovat a otestovat rychlost. Všechny testy byly prováděny na počítači s procesorem Intel Core 2 Duo 2GHz. Algoritmus byl v tomto stavu schopen během pár jednotek vteřin vyhodnotit strom do hloubky 2 tahů (tzn. 8 zahraných karet) a v průměru navštívil 5 milionů listů za jednu vteřinu (ve stromě hloubky 8).

Na řadu přicházejí jednotlivá prořezávání. Nejprve klasické alfa-beta prořezávání, které v Simulátoru 1.0 nebylo implementované. S alfa-beta prořezáváním bylo možné během několika desítek milisekund projít stromy hloubky 16, tzn. algoritmus je nyní na úrovni starého algoritmu..

Dále byla postupně implementována další prořezávání zmíněná v [6]. Největší vliv mělo prořezávání rychlých zdvihů (tento výsledek byl pozorován již v autorově bakalářské práci [11]). Jedná se o prořezávání, které neuvažuje karty na ruce, jež nemohou vyhrát aktuální zdvih. Z těchto karet je testována pouze ta nejmenší. Algoritmus je nyní schopen prohledávat stromy hloubky 24. Další prořezávání nahlíží na karty v „postupce“ jako na jednu kartu (z karet AKQ zkouší jen A). Toto prořezávání autor realizoval tak, že upravil výstupy výše zmíněné operace next. Předpočítané hodnoty byly pozměněny tak, aby operace next „přeskočila“ sekvenci za sebou jdoucích jedniček. Pozměněná operace next je znázorněna na obrázku č.5.



Obr 5 – Ukázka prořezávání posloupnosti karet jako součást operace next. T je zkrácené označení pro kartu hodnoty 10.

Poslední prořezávání, které bylo implementováno, je založené na znalosti maximálně získatelného počtu zdvihů pro danou hloubku prohledávání. Jestliže hledáme 6 tahů dopředu, tak maximální počet získatelných zdvihů je 6. Podobnou úvahu je možné učinit v jakékoliv hloubce a tím úspěšně ořezat ostatní tahy, bude-li již nalezen maximální možný. Aplikací všech výše zmíněných postupů bylo dosaženo výpočtu, který netrvá déle než stovky milisekund pro stromy hloubky 28. Algoritmus byl schopen prohledávat i hlubší stromy, ale už nebyl stabilní na době výpočtu, například při nastavení hloubky prohledávání na 32 se na určitých rozdáních výpočet zpomalil na deset či více vteřin, zatímco jiná rozdání byla vyřešena do sta milisekund. Je to tím, že například rozdání s krátkou barvou jsou obecně složitější než rozdání s rovnoměrně rozloženými kartami. Důvod je jednoduchý – algoritmus musí prohledávat až třikrát více možností, protože se velmi brzy dostane do stavu, kdy je

možné hrát všechny karty z ruky (jakmile některý z hráčů nese barvu, kterou jiný hráč v ruce nemá).

Finálním konceptem, který byl implementován, byla již zmíněná transpoziční tabulka [8]. Po zakomponování tabulky do algoritmu ale nedošlo k výraznému zrychlení. Na některých rozdáních byl výsledek patrný, ale neplatilo to obecně. Autor se tedy pokusil výrazně zagresivnit prořezávání generované tabulkou tím, že jako klič byla použita pouze jedna množina karet (AKQJ). Výsledek se mírně zlepšil, nyní je dosahováno desítek milisekund pro stromy hloubky 28, stromy hloubky 32 a 36 jsou již reálně řešitelné, ale v těchto hloubkách již velmi záleží na druhu rozdání. Čím nerovnoměrnější je rozloha karet, tím déle trvá výpočet.

4.6 Zhodnocení

Celkem se autorovi podařilo zlepšit algoritmus sehrávky zhruba o 80 až 100%. Maximální hloubka, do které počítá počítač v aplikaci Simulátor, je 28. Algoritmus sice zvládá výpočty do hloubky 32 a více, ale vzhledem k nestabilitě doby výpočtu danou různorodostí rozdání, se autor rozhodl nastavit výchozí hodnotu o jeden tah menší.

Možným zlepšením by mohlo být implementování metody zvané PartitionSearch popsané Ginsberkem [5]. Metoda spočívá v tom, že tahy, které očividně nevedou k úspěchu, nejsou vůbec prozkoumávány a je realizována na úrovni transpozičních tabulek (kóduje podobné situace do stejné). Bohužel, jakým způsobem zakódovat bridžovou situaci tak, aby se dvě významově (herně) podobné situace přeložily do stejného klíče tabulky, se autorovi diplomové práce nepodařilo zjistit ani vymyslet. Ginsberk bridžovou úlohu takto řešil, ale podrobněji se o tom nezmiňuje. Přiznává však, že aby šel tento koncept použít, je třeba nějakým netriviálním způsobem herní stav reprezentovat a většinou to vyžaduje expertní znalost samotné hry.

5. Implementační dokumentace

Autor při implementaci vycházel ze své bakalářské práce, ze které převzal hlavně koncepty uživatelského rozhraní, které shledal jako vyhovující a nebylo je třeba více měnit. Zejména konstrukci stromové hierarchie dražebního systému považoval za funkční a proto ji konceptuálně použil znovu. Nicméně, vzhledem k nutnosti přepracovat prakticky veškeré grafické zobrazení, vzhledem ke změnám celé dražební „logiky“ a vzhledem k tomu, že autor při psaní bakalářské práce nekladl důraz na „znovupoužitelnost“ kódu, bylo 95% kódu smazáno a přepsáno do nové podoby. Autor též zanechal rozdělení díla na dvě aplikace: Editor dražebních systémů a testovací rozhraní pro hraní bridže - Simulátor. Stejně tak se autor rozhodl i nové aplikace vyvíjet v jazyku C# nad rozhraním .NET.

Oba programy byly implementovány v prostředí .NET 2.0 ve Visual Studiu 2005. K jejich spuštění je nutné mít nainstalované příslušné knihovny (jsou součástí CD). Projekt se skládá ze sdíleného projektu (knihovny) BridgeBase a svou separátních projektů Editor a Simulátor. Zkompilovat jednotlivé projekty lze z prostředí Visual Studia, vždy je ale nutné referencovat projekt BridgeBase.

5.1. Úvodní myšlenky

Snahou autora bylo nově dílo hierarchizovat a modularizovat na menší části s předem definovanou funkčností a polem působnosti. Trend staré implementace byl sdružovat různé funkčnosti do jedné třídy (souboru), což není vhodné při konstrukci většího softwarového díla. V podobném duchu používá nově napříč celou aplikací Simulátor statické odkazy na nejdůležitější výkonné třídy. Ve staré implementaci vyvstávala velmi nepříjemná nutnost přidávat odkazy na veškeré výkonné třídy jako argumenty volání metod a parametry konstruktorů. Zápis metod byl pak neúměrně dlouhý a zavlékal do problému nadměrnou složitost. Autor nově používá kontejnery dostupné ve frameworku .NET, nejčastěji je to univerzální kontejner *ArrayList*. Ve staré implementaci si autor veškeré datové struktury (převážně spojové seznamy) implementoval sám. Vzhledem k tomu, že autor na díle pracoval v průběhu několika let, tak bylo důležité, aby programy byly psány stylem, který usnadňuje „znovupochopení“ a udržitelnost již napsaného kódu.

Dále mělo být zachováno přívětivé grafické rozhraní a měly být učiněny další kroky ke zlepšení přehlednosti programů. Autor se proto, mimo jiné, rozhodl do aplikace Editor implementovat možnost zoomu celé plochy formuláře a k přidání tooltipů nad pravidly dražebního systému. Simulátor umožňoval zobrazení pro 4 předdefinovaná rozlišení obrazovky, což je v dnešní době širokoúhlých monitorů nedostatečné. Nově se tedy měla grafika Simulátoru přizpůsobit libovolné velikosti okna.

Při implementaci nového Simulátoru se autor inspiroval takzvaným MVC (model, view, controller) konceptem. MVC architektura pojmenovává a odděluje logickou část aplikace od zobrazovací a datové části. Každá aplikace se dá takto myšlenkově rozložit a tento konstrukt se ukázal jako úspěšný mimo jiné i z důvodu, že jednotlivé části architektury se dají vyměnit (zobrazovací část se vymění za jinou, zatímco datová část může zůstat stejná). MVC se dnes asi nejvíce uplatňuje při konstrukci webových aplikací. Autor se rozhodl vytvořit nový Simulátor v podobném duchu, byť nepředpokládal, že by jednotlivé části Simulátoru mohly být vyměňovány za jiné. Autor však není úplně striktní a některé menší třídy (kontrolery) zastávají

funkčnost pohledu (view). Pro Simulátor to znamená, že téměř každá třída je rozložena na tři podtřídy Model, View a Controller. Rozložení na podtřídy není dědičného charakteru, jedná se jednoduše o tři různé, avšak kooperující třídy.

5.2. Grafické zobrazení

Samostatným problémem, se kterým se autor potýkal, bylo grafické zobrazení aplikací. Aplikace Editor narážela na nepříjemné blikání formuláře při scrollování, změně velikosti a při pohybu jiného okna nad formulářem. Hlavně při vytváření většího systému bylo blikání znatelné a znepříjemňovalo práci s aplikací. Autor toto vnímal jako velký problém, zejména vzhledem k tomu, že se snažil o co největší přehlednost. V nové aplikaci Editor chtěl navíc ke zvýšení přehlednosti implementovat plynulý zoom celého formuláře a tak bylo potřeba vykreslování zlepšit.

Problém byl v tom, že jednotlivé grafické elementy (pravidla systému) autor reprezentoval jako komponenty .NET prostředí (`Panel`, `Label`,..). Autor vypozoroval, že tyto komponenty se vykreslují až příliš často, aniž by proběhla jejich změna. Obecně platilo, že komponenty prostředí .NET nebyly příliš přívětivé co se týče plynulosti vykreslování, obzvláště pokud se s nimi hýbalo, nebo se hýbalo jiným oknem nad formulářem. Kromě nepříjemného blikání a vlnění bylo vykreslování i poměrně náročné pro procesor počítače, zvláště v momentě, kdy bylo na ploše umístěno velké množství komponent. Autor se rozhodl cachovat grafická data a využít metodu zvanou double-buffering k vyřešení grafických obtíží. Cacheování mělo pomoci v momentě, kdy se má překreslit celá aplikace, aniž by se změnil její obsah (scrollování, posun cizího okna nad formulářem). (Pozn.: autor pracuje s komponentami dostupnými v rozhraní .NET 2.0, k vykreslování používá GDI.).

Double-buffering je obecná technika používaná k vykreslování obsahu na obrazovku. Spočívá v tom, že namísto přímého kreslení do zobrazovacího „bufferu“ (pamět, jejíž obsah je přímo zobrazován na monitoru), se nejprve obraz předkreslí do záložního bufferu a teprve po skončení kreslení se data ze záložní paměti překopírují na obrazovku (ideálně se nový buffer vymění za starý, aby nedocházelo k dalšímu zdržení způsobenému kopírováním obrazu). Tento přístup má hlavní výhodu v tom, že se minimalizuje nepříjemné vlnění obrazu při zobrazování pohyblivé grafiky. „Vlnění“ vznikne v momentě, kdy se monitor začne překreslovat během měnění obsahu bufferu. Tím pádem se na monitoru objeví část nového obrazu zkombinovaná se starým obrazem a objeví se zmíněné „vlnění“ obrazu.

Autor se nejprve pokoušel o optimalizaci vykreslování již existujících komponent reprezentující pravidla dražebního systému v aplikaci Editor (`Panel`). To se mu ale nakonec nepodařilo. Ačkoliv se pozorovaná odezva vykreslování zlepšila, občas stále došlo k „svévolnému“ spuštění události `Paint`, kterou .NET komponenty používají k vykreslování své grafické reprezentace. Dále implementace zoomu přinášela problémy při scrollování formulářem. Autorovi se nepodařilo učinit, aby přednastavené scrollbary reflektovaly novou velikost zazoomovaných komponent. Zoom chtěl navíc ovládat kombinací kolečka na myši a klávesy ctrl, pohyb kolečkem ale vyvolává vertikální scroll formuláře. Autor byl nakonec nucen implementovat své vlastní komponenty jako finální řešení tohoto problému. Je možné, že k zajištění bezproblémového vykreslováni vedou jednodušší kroky. V momentě implementace ale bylo nutné učinit rozhodnutí, které by vedlo ke splnění úkolu a vlastní implementace se jevila jako časově úspornější řešení než zjišťování co se vlastně děje „pod kapotou“ .NET komponent.

Obr 6 – Ukázka grafického zpracování Simulátoru 2.0

Nově bude veškeré kreslení řešit pouze hlavní formulář aplikace, který bude též jedinou původní .NET komponentou. Při spouštění události `Paint` na hlavním formuláři se vždy kontroluje, zda byl obsah na formuláři znevalidněn a rozlišují se dva druhy znevalidnění – scroll a změna obsahu. Pokud obsah nebyl změněn, či proběhl pouze scroll formulářem, tak se použije již uložený (cacheovaný) obsah bufferu z minulého volání události `Paint`. Pokud byl celý obsah znevalidněn, tak se překreslí celý obsah. Jsou volány metody `Draw` všech vytvořených komponent, které vykreslí svůj obsah do double-bufferu. Poté je obsah double-bufferu poslán na obrazovku. Tento postup zamezil nadbytečnému kreslení v klidových stádiích aplikace a výsledná implementace již nevykazovala žádné blikání. K reprezentaci double-bufferu je využita .NET třída `ManagedBackBuffer`.

5.3 Projekt BridgeBase

Hlavním účelem projektu *BridgeBase* je implementace formuláře pro práci s dražebním systémem, stejně tak jako implementace samotného dražebního systému coby stromové struktury pravidel. Zároveň je v tomto projektu ukládána většina obrázků, které jsou používány v ostatních projektech. K ukládání obrázků je používán Resource Designer dostupný ve Visual Studiu. Obrázky jsou uloženy ve vygenerovaném souboru *resource.resx*, stejně jako gramatika podmínek, soubor *GrammarBig.xml*.

Základní zobrazovací třídou, kterou autor využívá jak v projektu Editor, tak v projektu Simulátor, je třída `BiddingTreeFormMassive`. Je to klasický .NET formulář typu `Form`. K zobrazení grafiky je používána .NET třída `ManagedBackBuffer`. Kreslení je prováděno naslouchání události `Paint` na hlavním formuláři `BiddingTreeFormMassive`. Ve formuláři je držen indikátor typu

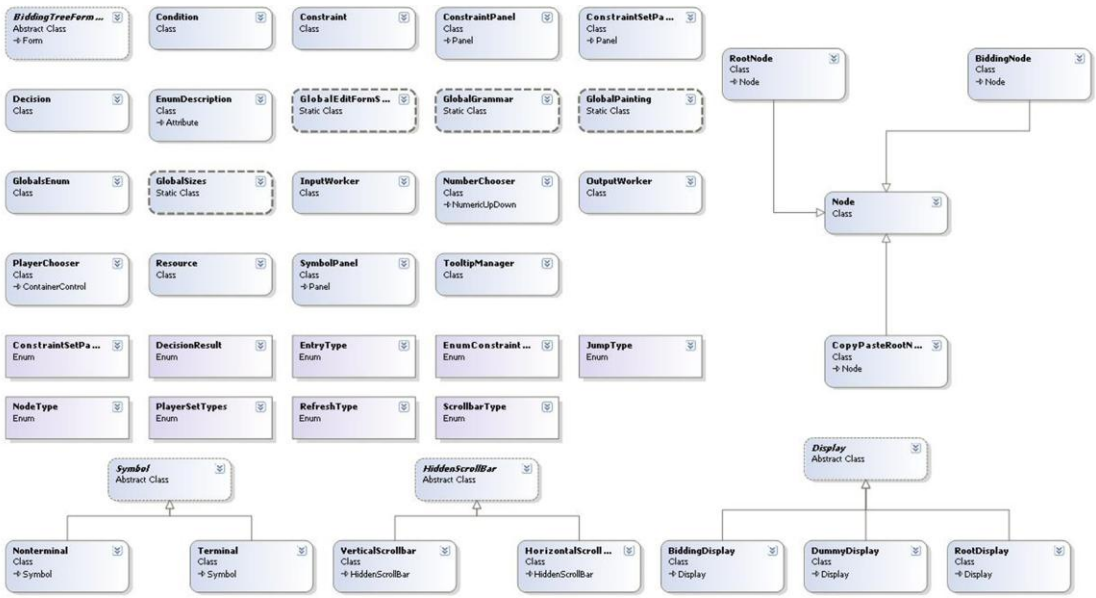
RefreshType s možnými hodnotami: NO_REDRAW_NO_SCROLL, NO_REDRAW_SCROLL, REDRAW_SCROLL. Kreslení se provádí pouze v momentě, pokud byl dražební systém od minulého kreslení změněn (REDRAW_SCROLL). Pokud je nutnost kreslit, hierarchicky se volají metody Draw jednotlivých pravidel (které jsou reprezentovány barevnými „obdélníky"). Těmto metodám se předá managedBackBuffer.Graphics, což je objekt typu Graphics, který v GDI (graphics device interface) reprezentuje grafický buffer či bitmapu, do které se kreslí. Ke konci je volána metoda managedBackBuffer.Render, která zajistí vykreslení obsahu bufferu na tělo formuláře. Obsah objektu managedBackBuffer není mezi voláními Paint mazán, takže se rychle použije znovu, pokud od minulého volání Paint neproběhla žádná obsahová změna.

Nejdůležitější skupinou tříd, kterou autor využívá napříč všemi aplikacemi, je abstraktní třída Node. Ta reprezentuje jedno pravidlo **H**, jeden vrchol dražebního stromu. Třída Node si udržuje odkazy na děti v kontejneru typu ArrayList. Grafickou reprezentaci a řešení uživatelského vstupu zajišťuje její potomek BiddingNode. V BiddingNode se provádí kreslení na formulář (metodou Draw). Je využita třída BiddingDisplay, potomek abstraktní třídy Display, která zajišťuje kreslení obdélníkového pravidla do bufferu typu Graphics. RootNode reprezentuje abstraktní kořen stromu, samotný kořen není na formulář zobrazen, ale RootNode zajišťuje pomocí svého objektu RootDisplay kreslení pomocných čar oddělující jednotlivé sloupce hlášek (a značí tak střídání partnerů při dražbě). Odkaz na kořen systému, na objekt typu RootNode, je umístěn ve formuláři BiddingTreeFormMassive. Při kreslení a při průchodech stromem se tak vždy začíná od objektu typu RootNode, který volání propaguje přes svoje děti. CopyPasteRootNode je využívána při kopírování částí stromu. Při označení vrcholu, který chceme zkopírovat, se reference uloží do instance CopyPasteRootNode. Při vkládání se na cílové místo uloží celý podstrom vrcholu CopyPasteRootNode.

Každá třída BiddingNode si udržuje jednu instanci objektu Condition. Condition reprezentuje seznam podmínek **p** v konkrétním pravidle, jednotlivé podmínky jsou reprezentovány objekty typu Constraint a jsou uloženy v kontejneru ArrayList. Constraint obsahuje ArrayList symbolů, tedy objektů typu Terminal či Neterminal, jež jsou potomky společné nadtřídy Symbol. Načítání a ukládání systému ve formátu XML je realizováno třídou InputWorker respektive OutputWorker. Třída InputWorker umí též načítat soubory ve formátu PBN (Portable Bridge Notation). PBN je oficiální formát k ukládání bridžového rozdání. Velké bridžové turnaje často nabízejí ke stažení turnajová rozdání ve tvaru PBN.

Grafická reprezentace podmínek je realizována komponentami ConstraintSetPanel, ConstraintPanel a SymbolPanel. Všechny tři komponenty rozšiřují .NET komponentu Panel. Jedno pravidlo je reprezentováno komponentou (grafickým kontejnerem) ConstraintPanel, která v sobě drží ArrayList symbolů-atomů typu SymbolPanel. ConstraintSetPanel je grafický kontejner a zobrazuje všechny podmínky (objekty typu ConstraintPanel) ve formě scrollovacího okna. Atomy jsou tedy reprezentovány jako textové objekty typu SymbolPanel a mohou to být buď terminály či neterminály gramatiky. Jedná-li se o neterminál, SymbolPanel nabídne uživateli výběr možných přepisů tohoto neterminálu. Klikem na přepis se SymbolPanel přemění na nové symboly (SymbolPanel) gramatiky (přidají se do ConstraintPanel). V případě, že terminály jsou čísla či atomy KDO, budou nahrazeny speciálními komponentami NumberChooser či PlayerChooser, které zprostředkovávají uživateli výběr konkrétního čísla či množiny hráčů.

Dalšími třídami obsaženými v projektu `BrideBase` jsou grafické, autorem implementované, scrollbary `HorizontalScrollbar` a `VerticalScrollbar` (nadtřídou je `HiddenScrollbar`). Scrollbary mají též svou metodu `Draw`, která zprostředkovává vykreslení do objektu typu `Graphics`. Další pomocnou grafickou třídou na zobrazování tooltipů je `TooltipManager`. `TooltipManager` drží pozici aktuálně zobrazovaného tooltipu a spouští `Timer` při nájezdu na další vrchol. Ke kreslení se jako pomocné třídy využívají statické objekty `GlobalPainting` a `GlobalSizes`. `GlobalPainting` udržuje objekty typu `Brush`,`Font` a `Pen` potřebné ke kreslení v prostředí .NET. `GlobalSizes` obsahuje gettery, které vracejí všechny potřebné hodnoty využívané ke kreslení (layoutové hodnoty - vzdálenosti, velikosti a hodnotu zoomu).



Obr 7 – Třídy projektu BridgeBase

5.4 Projekt Editor

Projekt Editor obsahuje formulář `MainFormMassive`, který je potomkem `BiddingTreeMassive` a umožňuje vytváření a editování hlášek. K funkčnosti `BiddingTreeMassive` přidává hlavní (`MenuStrip`) a kontextové menu (`ContextMenuStrip`), jejichž prostřednictvím uživatel ovládá konstrukci a editaci dražebního stromu. Projekt dále obsahuje formulář `Edit`. Formulář `Edit` v podstatě slouží k zobrazení a k editaci podmínek pravidla (`Condition`) a využívá k tomu komponenty zmíněné výše: `ConstraintSetPanel`, `ConstraintPanel`. Jednotlivé atomy jsou zobrazeny pomocí tříd `SymbolPanel`, `NumberChooser` a `PlayerChooser`. Ostatní atributy podmínky jsou zobrazeny a editovány pomocí klasických komponent .NETu `RadioButton`, `CheckBox`, `Numeric` a `TextBox`. Formulář `Edit` neimplementuje grafiku pomocí dvojitého bufferu.

5.5 Projekt Simulátor

Vstupním bodem aplikace Simulátor je hlavní formulář `PlayTable`. Stejně jako formulář `BiddingTreeFormMassive`, i `PlayTable` realizuje zobrazování pomocí dvojitého bufferu. Metody `Draw` tříd reprezentující příslušné pohledy (views) jsou volány z centrálního bodu, delegátu `PlayTable_Paint` (naslouchající události `Paint` na formuláři `PlayTable`). Autor tento druh zobrazení implementoval již ve staré verzi Simulátoru, nicméně v době implementace nepoužíval výhody `ManagedBackBuffer` a výsledek nedosahoval kvalit nové implementace (místy se objevilo blikání obrazovky). Na rozdíl od formuláře `BiddingTreeMassive`, u kterého se počítá s tím, že svůj obsah nemusí zobrazit celý (při scrollování či zoomování), `PlayTable` vždy zobrazuje veškerý svůj obsah (karty a dražební tabulku). `PlayTable`, nebo lépe řečeno všechny třídy realizující kreslení (views), musí svůj obsah pozicovat a případně zmenšovat tak, aby se vešel do libovolné velikého okna. Zmenšování je prováděno při zachování poměru originálních obrázků. Kreslení je prováděno za pomoci getterů ze statické třídy `LAYOUT` a podobně, statická třída `Painting` udržuje použité objekty typu `Pen`, `Brush` a `Font`. Objekty typu `Font` se při změně velikosti okna zkonstruují znovu v nové velikosti.

Ostatní formuláře využívané v projektu Simulátor již neimplementují dvojité bufferování. Dalšími formuláři, které jsou v projektu používány jsou:

- Formulář `TestingGrounds` - slouží k zobrazení testovacího rozhraní. Zde je možno otestovat dražební systémy navzájem proti sobě. Testování probíhá tak, že je vygenerováno náhodné rozdání a na tomto rozhrání odehrají hry všechny kombinace dvojic vstupních systémů proti sobě. Každý systém z dvojice na jednom rozdání odehraje dvě hry, jednu z pozice Sever-Jih a druhou z pozice Západ-Východ. Výsledky zápasů jsou počítány pomocí IMP bodování (bridžové turnajové bodování). Po odehrání všech her pro testované rozdání se rozdání zahodí, vygeneruje se nové a vše začíná nanovo. Uživatel může kdykoliv proces testování zastavit, též lze nastavit fixní počet testovaných rozdání, po kterém testování skončí.
- `Settings` - zde se provádí načítání dražebních systémů, reprezentovaných třídou `RootNode`, pomocí třídy `InputWorker`. Dále se zde nachází nastavení, zda jsou hráči reprezentováni počítačem nebo se jedná o lidského hráče. Nakonec lze zapnout „podvádění“ algoritmu sehrávky. Tzn. algoritmus sehrávky bude vidět do karet všech hráčů a nebude prováděna Monte Carlo simulace. Nastavením všech hráčů jako počítačových, společně s nastavením „podvádění“ a odkrytím karet, lze sledovat jak by nejoptimálněji řešil dané rozdání počítač s čistě jen s použitím double-dummy algoritmu.
- `SystemViewer`, který je potomkem třídy `BiddingTreeFormMassive`, slouží k zobrazení aktuálně načteného systému. Na formuláři `SystemViewer` nelze dělat úpravy dražebního stromu. Na druhou stranu, `SystemViewer` umí zobrazit dodatečné informace z průběhu dražby, jako jsou doposud dražené hlášky a případné chyby zpětného určení.

- Formulář `MemoryDebug`, který slouží k zobrazení obsahů znalostí všech hráčů (`MemoryModel`).

Všechny kontrolery jsou umístěny ve statické třídě `G` a jsou tudíž dosupné ze všech částí programu. Základním řídící třídou aplikace je třída `MainController`, která řídí veškerý průběh hry a případně přeposílá informace dalším kontrolerům. `PlayController`, `AuctionController` jsou hned po `MainControlleru` na řadě v pořadí důležitosti, protože řídí průběh sehrávky respektive dražby. `PlayController` i `AuctionController` mají své příslušné modely `PlayModel` a `AuctionModel`, které drží informace o stavu aktuálně hrané sehrávky resp. dražby. Graficky je sehrávka resp. dražba zobrazována pomocí objektu `PlayView` a `AuctionView`.

Dalšími řídícími třídami jsou `ContractHandController`, který prostřednictvím třídy `ContractHandView` zobrazuje skóre, stav aktuálního závazku (v tabulce dole vlevo) a indikátor hráče na tahu. Dále `DealController`, který se stará o rozdání konkrétních karet - načítá a ukládá rozdání reprezentované objektem `DealModel`, případně generuje náhodná rozdání. `HistoryController` uchovává odehrané karty a globální znalost reprezentovanou objektem `MemoryModel`, dalším kontrolerem je `TableController`, který reprezentuje odehrané karty na stole. `SettingsController` řídí nastavení aplikace a načítání systémů a údajů o hráčích (formulář `Settings`). Tyto informace uchovává v `SettingsModel`. A konečně velmi důležitý je `PlayersController`, který reprezentuje hráče a fyzicky spouští algoritmy dražby a sehrávky, pokud aktuální hráč na tahu je počítač. Údaje o hráči, jeho aktuální karty, aktuální pozice v systému a jeho paměť,ve formě instance třídy `MemoryModel`, jsou v `PlayersController` uloženy v objektu `PlayerModel`, který de fakto reprezentuje data jednoho hráče.

Některé modely a pohledy (views) nemají svůj kontroler. Je to proto, že jsou zpravidla řízeny jinými kontrolery a autor považoval za zbytečné kontroler implementovat. Jedná se hlavně o třídy, které mají reprezentovat nějakou informaci. `PlayerModel`, jak už bylo řečeno výše, reprezentuje údaje o hráči. `PlayerView` je pomocná třída, která má za úkol zobrazit karty hráče na hrací ploše. Karta je reprezentovaná dvojicí `CardModel` a `CardView`, dražební hláška pak dvojicí `CallModel` a `CallView` (zobrazuje hodnotu hlášky a obrázek

Naprosto zásadní třídou v celém programu je `MemoryModel`. `MemoryModel` je reprezentací báze znalostí **B** jednoho hráče. Zde se zároveň provádí veškeré logické vyhodnocení a zpřesňování znalostí. Podmínka **p** je v Simulátoru reprezentována třídou `LogicExpression`. `LogicExpression` obsahuje řadu statických metod, které řeší konverzi enumů požívaných v *BridgeBase* a v Editoru na enumy, používané v Simulátoru. `LogicExpression` dále obsahuje statickou metodu `ConstructExpression`, která vezme množinu symbolů `Symbol` a vrátí konkrétní objekt, reprezentující logický výraz. Možnými logickými výrazy, coby potomky `LogicExpression`, jsou:

- `ActualContractExpression` (podmínky týkající se aktuálního závazku)
- `AuctionQueryExpression` (podmínky týkající se dražby)
- `CardComparingExpression` (karetní podmínky, porovnávaní s počty konkrétních karet)
- `PointsComparingExpression` (porovnání bodů)
- `ColorComparingExpression` (a porovnání délky barev).

Každá `LogicExpression` implementuje metodu `Evaluate`, která jako argument dostává `MemoryModel` hráče (případně profiltrovanou barvu) a provede evaluaci podmínky (eval) dle zadaného objektu `MemoryModel` (tzn. báze znalostí). Samotné ověření platnosti podmínky se provádí voláním příslušné metody objektu typu `MemoryModel`, která jako svoje argumenty obdrží jednotlivé atomy zkoumané podmínky. Tyto metody přesně reflektují všechny druhy zkonstruovatelných podmínek (kromě podmínek týkajících se průběhu dražby a aktuálního závazku, protože tyto informace jsou uloženy ve třídě `AuctionModel`) a jsou to:

- `COMPARE_CARD_WITH_BOUNDS`
- `COMPARE_CARD_WITH_CARDS`
- `COMPARE_CARD_WITH_NUMBER`
- `COMPARE_CARD_WITH_PLAYER_SET`
- `COMPARE_COLOR_WITH_BOUNDS`
- `COMPARE_COLOR_WITH_COLOR`
- `COMPARE_COLOR_WITH_NUMBER`
- `COMPARE_COLOR_WITH_PLAYER_SET`
- `COMPARE_POINTS_WITH_BOUNDS`
- `COMPARE_POINTS_WITH_COLOR`
- `COMPARE_POINTS_WITH_PLAYER_SET`

Podobně, `MemoryModel` obsahuje sadu metod, které slouží k vytěžení informace z platné podmínky. Tyto metody v podstatě kopírují výše zmíněné s rozdílem, že nevyhodnocují podmínku (kterou obdrží jako parametry), ale pokusí se z podmínky vytěžit informaci, tzn. zpřesnit znalosti v bázi. Důležitou metodou ve třídě `MemoryModel` je metoda `SynchroniseBoundsAmongPlayers`, která je spouštěna vždy, je-li zpřesněna nějaká znalost. Každé zpřesnění znalosti může vyvolat kaskádu změn ostatních znalostí a přesně to se děje uvnitř této metody. Metoda se spouští tak dlouho, dokud probíhají změny znalostí. Tento proces musí skončit, protože každá znalost se dá zpřesnit zvětšením dolní resp. zmenšením horní meze příslušného intervalu. A to lze dělat pouze do doby, než se dolní mez začne rovnat horní.

Intervaly v bázi znalostí reprezentované třídou `MemoryModel` jsou realizovány ve třídě `Bounds`. `CardBounds`, `PointBounds` a `ColorBounds` jsou nadstavbové třídy, které obsahují karetní resp. bodové resp. barevné intervaly pro všechny typy karet resp. bodů resp., barev (instance třídy `Bounds`). `PlayerSet` logicky reprezentuje atom KDO.

Třída `AuctionEvaluator` slouží k nalezení vítězného pravidla v průběhu dražby (hlášené hlášky). `AuctionEvaluator` prochází dražebním stromem (metoda `EvaluateBidding`) z aktuální pozice pro danou partnerskou dvojici a kontrolou podmínek se snaží najít první pravidlo, které splní všechny své podmínky. `AuctionEvalutor` nejprve vyhodnotí všechny „jednoduché“ podmínky (bodové hodnoty, úroveň skoku..), než se přejde k samotné evaluaci využívající `LogicExpression`. `InformationEvaluator` plní báze znalostí `MemoryModel` daty získanými z pohledu na svou ruku (začátek hry), na ruku tichého hráče (po úvodním výnosu) a na jednotlivé karty, hež se postupně objevují na stole během běžného hraní. `InformationEvaluator` ještě implementuje metodu `DoBackwardTrace`, která realizuje zpětné určení, tzn. nalezne novou pozici v systému (dražené pravidlo) po hlášce partnera.

Třída `DoubleDummySolver` reprezentuje implementaci algoritmu s odkrytými kartami, statická metoda `start` začíná výpočet. `Metoda` start zkonstruuje novou instanci objektu `DoubleDummySolver`, naplní ji aktuálními daty (metoda

`InitializeSituation`) a zahájí výpočet (`SolveSituation`). Metoda `GenerateHands` vygeneruje náhodné rozložení karet tak, aby rozložení neporušovalo aktuální znalosti. Průchod herním stromem je realizován metodami `GoUp` a `GoDown`. Pole `HANDS` obsahuje 16 čísel reprezentující barevné listy (ruce) všech hráčů (13 místné binární číslo, viz 4.4). V poli `TRANSITIONS` (pole čísel) jsou uloženy změny vyvolané průchodem herního stromu. Políčko v `TRANSITIONS` je číslo složené z:

- Aktuálních hodnot alfa a beta (2*4 bity)
- Aktuální hraná karta (převedená z 13 místné binární reprezentace do decimální 1-13, 4 bity).
- Aktuální vítězná karta na stole (4 bity)
- Barva zdvihu (2 bity)
- Barva hrané karty (2 bity)
- Indikátor, zda je vítězná karta trumfová (1 bit)
- Indikátor, zda naše linka vyhrává (1 bit)
- ID hráče na tahu (2 bity)

Při navštívení nového syna se volá metoda `GoDown`, která do políčka v poli `TRANSITIONS`, příslušného aktuální hloubce zanoření ve stromu, uloží informace, která při návratu umožní vrátit stav hry o kolo zpět. Při návratu výš se provede metoda `GoUp`, která navrátí herní stav dle informací uložených v příslušném záznamu v `TRANSITIONS`. `GoDown` provádí samotné hraní, `GoUp` vrací stav o kolo zpět, ale zároveň zde dochází k případné aktualizaci aktuálně hrané karty, pokud byla nalezena optimálnější sehrávka. Dále se zde mohou též aktualizovat hodnoty alpha a beta. `TRANSITIONS` nemůže mít více jak 52 záznamů, protože 52 je maximální hloubka stromu. Algoritmus si drží situaci, která vedla k zatím nejlepšímu výsledku (`WINNING_SITUATION`, z ní se pak dekóduje vítězná karta). Pole `GET_HIGHEST` a `GET_LOWEST` reprezentují operace next a last zmíněné v kapitole 4.4.

K výpočtu je použita třída `BackgroundWorker`, která je v prostředí .NET realizací jednoho vlákna. Výpočet trvá maximálně pět vteřin (ke stopování času využívám .NET třídu `Timer`).

Obr 8 – Třídy projektu Simulátor

```
public void SolveSituation()
{
  uint cardToPlay = 0;
  while(true)
  {
    // Alpha-Beta prorezani + dalsi prorez, pokud k nemu doslo jinde
    if ((ALPHA >= BETA || IS_CUTOFF)&& DEPTH_LEFT != STARTING_DEPTH_LEFT)
    {
      IS_CUTOFF = false;
      cardToPlay = GoUp();
      continue;
    }

    // Vyber hrane karty pomoci predpocitane iterace (operace next)
    cardToPlay = GET_HIGHEST[HANDS[(HAND_INDEX * 4) + EXAMINED_COLOR] +
                 (CARD2INDEX[cardToPlay] << 13)];
    // .. prosli jsme vsechny vrcholy na teto urovni pro danou barvu ..
    if (cardToPlay == 0)
    {
      // Dosli jsme na konec barvy a zrovna vynasim ci mohu hrat lib. barvu ..
      if (EXAMINED_COLOR != TABLE_COLOR )
      {
         EXAMINED_COLOR = (EXAMINED_COLOR + 1) % 4;
         continue;
      }

      // Uplny konec vypoctu ..
      if (DEPTH_LEFT == STARTING_DEPTH_LEFT)
      {
        // Nastaveni vitezne situace, pokud behem vypoctu nebyla jiz nastavena
        if (WINNING_SITUATION == 0)
          WINNING_SITUATION = TRANSITIONS[STARTING_DEPTH_LEFT];
        break;
      }

      // Navrat o uroven vyse po prozkoumani vsech vrcholu
      cardToPlay = GoUp();
      continue;
    }

    // Zanoreni se o uroven hloubs ..
    GoDown(cardToPlay);
    cardToPlay = 0;

    // Jsme-li na "dne", tak se vratime ..
    if (DEPTH_LEFT == 0)
    {
      ALPHA = BETA = NUM_OF_TRICKS;
      cardToPlay = GoUp();
    }
  }
}
```

***Obr** 9 – Implementace těla double-dummy algoritmu, metody SolveSituation, v Simulátoru 2.0*

6. Závěr

6.1 Zhodnocení

Obě aplikace byly od základů předělány a přepsány, zachovány byly pouze funkční koncepty - hlavně pokud se jedná o uživatelské rozhraní. U obou aplikací došlo k výraznému zlepšení odezvy uživatelského rozhraní použitím double-buffered vykreslování.

Aplikace Simulátor 2.0 nyní výrazným způsobem čerpá a interpretuje informace získané z dražby a ze sehrávky, informace získává i ze situací, které se nestaly znegováním jejich podmínek. Simulátor 1.0 prováděl pouze základní interpretaci bodového rozhraní dražených hlášek a pamatoval si hrané karty, které pak nevyužil ke generování náhodných rozdání Monte Carlo vzorkováním.

Algoritmus sehrávky byl výrazně zlepšen, dá se hovořit o téměř zdvojnásobení hloubky zkoumaného stromu. Simulátor 1.0 byl schopen prohledávat herní strom hloubky 16 odehraných karet během několika stovek milisekund. Hodnota 16 byla fixně nastavena pro Simulátor 1.0. Při zvětšení hloubky nad 20 karet se výpočet zpravidla odmlčel na dlouhé minuty. Simulátor 2.0 je schopen prohledávat stromy hloubky 32 i více, rekordem je odehraná hra hloubky 36 (double-dummy s odkrytými kartami), kde jeden výpočet trval řádově pár vteřin. Nutno ale podotknout, že vzhledem k různorodosti bridžových rozdání, je tato hodnota značně nestabilní – některá rozdání, hlavně ty s krátkostí v nějaké barvě, se nedají uspokojivě řešit s novým algoritmem ani při zmenšení hloubky výpočtu na 32. Výpočet může trvat několik desítek sekund a to je nepřijatelné pro normálního uživatele. Simulátor 2.0 má tedy fixně nastavenou hodnotu 28 jako maximální hloubku prohledávaného stromu.

Algoritmus sehrávky, přestože je jeho úspěšnost uspokojivá, stále ještě nedosahuje dostatečných kvalit, aby jeho výsledek mohl být označen za optimální. Dostatečnou kvalitou je myšlena schopnost prohledat celý herní strom hloubky 52. Proto se autor rozhodl neimplementovat ukázku optimální sehrávky přímo do grafického rozhraní aplikace. Implementace této funkcionality byla zamýšlena jako nabídka všech možností hraní karet, kde u každé karty by byl vidět maximální počet získatelných zdvihů při optimální hře všech hráčů. Bylo by to plnohodnotné výukové rozhraní Simulátoru. V rámci přiblížení se zadání práce přidělal autor možnost, jak si nechat sehrát rozdání počítačem využívající algoritmus sehrávky s odkrytými kartami. Uživatel tak může na libovolném rozdání vidět nejlepší hru, které je počítač aktuálně schopen, nejedná se však o optimální sehrávku (v nastavení lze všechny hráče označit jako „počítač“ a zaškrtnout políčko „počítač vidí do karet“). Platforma 2.0 nadále obsahuje jednoduché testovací rozhraní, kde je možno načíst více systémů naráz a otestovat jejich bodový zisk na náhodných rozdáních. Můžeme tedy mluvit o testování „reálné“ síly vytvořených systémů.

Popisná síla Editoru 2.0 byla výrazně rozšířena oproti veskrze předdefinovaným možnostem Editoru 1.0. Nově se pravidla zadávají použitím atomických prvků definovaných gramatikou, jednotlivá pravidla jsou tak více homogenní a díky tomu lze nad celou platformou budovat nadstavbové mechanismy, například celý systém získávání znalostí, jež je představen v této práci. Obranná dražba (dotazy na soupeře) se nyní zadává stejně jednoduše, jako dražba obyčejná.

Jediná věc, která se autorovi nepodařila implementovat, je hledání míst nepokrytých systémem. Při řešení tohoto problému se nepodařilo najít žádný

jednotící rámec. Původně autor zamýšlel místa, která reprezentují rozdání nepokrytá systémem, zobrazit v Editoru 2.0 jako speciální pravidla, jejichž podmínka se vygeneruje tak, aby přesně zachycovala nepokrytou část. Tzn. program by uměl detekovat „mezery“ v systému a tyto mezery se měl pokusit nahradit pravidly. Generování pravidel a nalezení volných míst tímto způsobem je však značně obtížná záležitost, hlavně ve složitějších systémech.

Při implementaci testovacího rozhraní Simulátoru 2.0 autor přišel na zajímavý způsob, jak automaticky řešit a dokonce doplňovat slepá místa v systémech. Na náhodné rozdání by se, bez předchozí dražby, spustil algoritmus sehrávky s odkrytými kartami pro každou možnou barvu závazku (celkem pětkrát). Výstupem takového algoritmu by byly různé hodnoty uhraných zdvihů pro různé trumfové barvy. Tímto způsobem je možné získat nejvyšší uhratelný počet zdvihů pro každé rozdání a přiřazenou trumfovou barvu. Optimální závazek by nesl barvu (či více barev), pro níž nalezl algoritmus nejvyšší hodnotu (tato hodnota by byla použita jako stupeň závazku). Nyní by se nad rozdáním otestovaly dražební systémy a vítězem by byl ten, který by se svým výsledkem co nejméně lišil od nalezeného optimálního závazku. Naopak v systému, který by se výrazně lišil svým vydraženým závazkem od právě nalezeného optimálního závazku, by bylo vygenerováno pravidlo („pojmenování“ právě nalezeného slepého místa), které by reflektovalo stav aktuálního rozdání. Vyžadovalo by to však algoritmus sehrávky s odkrytými kartami, který je schopen prohledat celý herní strom. Generování pravidel též nebude triviální, vzhledem k potencionálnímu velkému počtu takto nalezených míst by bylo třeba ještě implementovat sjednocování pravidel. Pokud by se povedlo implementovat tento mechanismus, tak by bylo možné automatizovat vytváření systémů a hledání slepých míst v systému.

Přestože byl, dle názoru autora, cíl diplomové práce splněn, dána problematika je natolik složitá, že nabízí nepřeberné možnosti dalšího zlepšování a skýtá celou řadu nových podnětů.

6.2 Pohled do budoucna

Oblastí, ve kterých by se mohla platforma 2.0 nadále zlepšovat, je nepřeberně. Od dalšího zlepšení uživatelského rozhraní a ovladatelnosti programů (například přidání drag and drop techniky při vytváření stromů), přes zlepšení algoritmu sehrávky, zlepšení nově vytvořeného prostředí pro testování systémů, hra po síti, možnost stáhnout si rozdání z velkých světových turnajů a v pohodlí domova si je „odehrát“ nebo nechat odehrát počítačem, přidání přirozené dražby počítače v momentě, kdy se ocitne mimo systém a tak dále. Možností a cest, jakým směrem programy nadále zlepšovat, je opravdu mnoho. Platforma 2.0 by mohla fungovat jako odrazový můstek pro další experimenty a pokusy ze světa bridže - například pro vytváření systémů genetickými algoritmy.

6.3 Zamyšlení se nad genetickými algoritmy

V průběhu implementování Editoru 2.0, konkrétně po naimplementování gramatiky podmínek, se autor zamýšlel nad tím, jak by asi vypadal systém, postavený čistě počítačem za použití genetických algoritmů. Struktura podmínky může dokonce genom vzdáleně připomínat. Nově vybudovaný jednotný systém zaměnitelných podmínek složených z navzájem zaměnitelných terminálů (v rámci svých definičních oborů) k tomuto přímo vybízí. Fascinující je myšlenka, co by mohlo z takových

pokusů vzniknout. Pokud je autorovi známo, nikdo se ještě nikdy nepokoušel o vytvoření dražebního systému počítačem.

Genom by tedy byl celý dražební strom, v testovacím rozhraní Simulátoru 2.0 by počítač odehrával hry systémem každý s každým na náhodně generovaných rozdání. Aby byla zajištěna co možná největší přesnost, hrálo by se turnajovým stylem, kdy se u každého rozdání vystřídají postupně všichni hráči. Po odehrání určitého počtu kol by se vybrali jedinci s největším bodovým ziskem IMP (International match points). V rámci množiny šampiónů by proběhlo klasické křížení a mutace (na dražebních systémech) a vše by začalo znovu. Mohl by se nechat generovat celý systém na zelené louce, stejně jako „zasadit“ již hotový systém nebo jen jeho část.

Mutace by probíhala na třech úrovních :

1) Na úrovni terminálů v konkrétních podmínkách. Terminál může být zaměněn za jiný terminál stejného typu od stejného rodiče – neterminálu.
2) Na úrovni podmínek – celá podmínka může být zaměněna za jiný typ podmínky, nebo může být přidána celá nová podmínka (nebo stará smazána).
3) Na úrovni pravidel a struktury dražebního stromu – mohla by vznikat nová pravidla či zanikat stará, mohla by se měnit priorita pravidel v rámci stejné množiny odpovědí a nakonec by bylo zajímavé umožnit, aby pravidla mohla cestovat od větve k větvi, tzn. aby se mohly měnit odpovědi napříč pravidly.

Křížení by mohlo fungovat jak na úrovni celého stromu, tak na úrovni pravidel. Při křížení pravidel by si jednotlivá pravidla vyměnila část svých podmínek. Křížení na úrovni celého stromu by se realizovalo jednoduše pouhým přepojením dražebních stromů.

Literatura

[1] ALLEN, J.D.,*Solve Double-Dummy Bridže Problems*.[online] [cit. 2012-03- 08]. Dostupné z: <http://fabpedigree.com/james/dbldum.htm>

[2] AMIT, A., MARKOVITCH, S. *Learn to Bid in Bridge*. Department of Computer Science, Israel institute of Technology [online] [cit. 2012-07- 10]. Dostupné z: <http://www.cs.technion.ac.il/~shaulm/papers/pdf/Amit-Markovitch-mlj2005.pdf>

[3] BASIN,D., BUNDY,A., FRANK,T.*Combining Knowledge and Search to Solve Single Suit Brdidge*. In Proceedingsofthe 10th Europeanconference on Artificialintelligence table ofcontents, pages 72-76, John Wiley&Sons, Inc. New York, NY, USA, 1992. [cit. 2012-07-07].
Dostupné z: <http://www.inf.ethz.ch/~basin/pubs/aaai2000.ps.gz>

[4] GINSBERK, M. *How computers will play bridge*. [online]. 1995 [cit. 2012-03-16]. Dostupné z: <ftp://ftp.cirl.uoregon.edu/pub/users/ginsberg/papers/bridge.ps.gz>

[5] GINSBERK, M. *Partition Search* [online].1996 [cit. 2012-03-16]. Dostupné z: <ftp://ftp.cirl.uoregon.edu/pub/users/ginsberg/papers/partition.ps.gz>

[6] GINSBERK, M. GIB: *Steps Toward an Expert-Level Bridge-Playing Program* [online]. 1997[cit. 2012-03-26]
Dostupné z: <ftp://ftp.cirl.uoregon.edu/pub/users/gins>

[7] HEBÁK, P. *Bridž pro každého*, 1. vydání. Praha: nakladatelství Informatorium, 1997.239s.1. ISBN 80-85646-41-2.

[8] CHANG,M.*Building a fast Double-dummy solver* [online].1996 [cit. 2012-02-16].Dostupné z: <http://cs.nyu.edu/web/Research/TechReports/TR1996-725/TR1996-725.pdf>

[9] MOSSAKOWSKI, K., MANDZIUK,J. *Artificial Neural Network for Solving DoubleDummy Bridge Problems.* Fakulty of Mathematics and Information Science, Warsaw University ofTechnology .[online] [cit. 2012-04-05]. Dostupné z: < http:// www.mini.pw.edu.pl/~mandziuk/PRACE/ICAISC04-2.pdf>

[10] NYU,P,. *The State of Automated Bridge Play*. [online] [cit. 2012-04-05]. Dostupné z: <http://ephman.org/bridgeReview200908.pdf>

[11] PRAŽMA, J. *Bridž a analyzování bridžové sehrávky*. Praha 2007. Bakalářská práce na MFF UK na katedře softwarového inženýrství. Vedoucí bakalářské práce Mgr. Pavel Ježek

[12] SMITH, S., NAU, D., THROOP, T., *Computer bridge: A big Win for AI planning.* [online] [cit. 2012-20-07].
Dostupné z: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.6977>

[13] Přehled dražebních systémů .[online] [cit. 2012-04-02].
Dostupné
z:<http://homepage.mac.com/bridgeguys//Conventions/BiddingSystems.html>

[14] World Computer-Bridge Championship.[online] [cit. 2012-04-02].
Dostupné z: <http://www.ny-bridge.com/allevy/computerbridge>

Odkazy na bridžový software zmiňovaný v textu:

[15] Bridge Baron [počítačový program]. Dostupné z:
<http://www.bridgebaron.com/home.shtml>

[16] DeepFinesse [počítačový program]. Dostupné z:
<http://www.deepfinesse.com>

[17] GIB. [počítačový program]. Dostupné z:<http://www.gibware.com>

[18] Jack Bridge. [počítačový program].Dostupné z:<http://www.jackbridge.com>

[19] Micro Bridge. [počítačový program]. Dostupné z:
<http://www.osk.3web.ne.jp/~mcbridge >

[20] SharkBridge. [počítačový program]. Dostupné z:<http://www.sharkbridge.dk>

[21] WBridge5. [počítačový program]. Dostupné z:<http://www.wbridge5.com>

Příloha 1 – Obsah CD

1) Práce ve formátu pdf (Bridge.pdf)
2) Návod na spuštění obou programů (Navod.txt)
3) Uživatelský manuál (Manual.pdf)
4) Zcompilované programy (ve složce bin)
5) Zdrojové soubory a .sln soubory pro VisualStudio 2005 (složka src)