

České vysoké učení technické v Praze

Fakulta Elektrotechnická



Diplomová práce

## Implementace metod počítačového hraní strategických her

*Bc. Petr Houdek*

Vedoucí práce: RNDr. Marko Genyk-Berezovskyj

Studijní program: Elektrotechnika a informatika, strukturovaný magisterský

Obor: Výpočetní technika - Systémové programování

Leden 2010



## **Prohlášení**

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 4.1 2010 .....  
.....



## **Poděkování**

Rád bych poděkoval vedoucímu mé diplomové práce RNDr. Marko Genyk-Berezovskému za jeho spolupráci, podnětné konzultace a trpělivost. Rovněž bych také rád poděkoval mé rodině, která mě morálně a duševně podporovala během tvorby této práce.



# **Abstrakt**

Diplomová práce se zabývá použitím algoritmů umělé inteligence ve strategických hrách. Teoreticky jsou popsány a implementovány algoritmy Minimax, Negamax, Alfa – Beta a jejich vylepšení. Tyto algoritmy jsou použity v implementaci obecného grafického rozhraní umožňujícího pracovat s libovolnou strategickou hrou. Pro tuto práci byly popsány a implementovány tři abstraktní antagonistické hry – Abalone, Terrace, Yinsh.

Úvodní část popisuje zvolené hry, jejich pravidla a historii. Druhá část se zabývá teoretickým popisem herních stromů, použitých algoritmů a jejich zlepšujících technik. Návrh a realizace jsou popsány ve třetí části, která se věnuje způsobu implementace algoritmů, heuristických funkcí a dalších vybraných částí aplikace. Čtvrtá část se věnuje testování a porovnávání vlastností jednotlivých algoritmů a ohodnocovacích funkcí. V závěrečné části se nachází shrnutí dosažených výsledků a náměty do budoucna pro další výzkum.

# **Abstract**

The master's thesis deals with an application of artificial intelligence algorithms in strategy games. Algorithms Minimax, Negamax, Alpha – Beta and theirs improvements are theoretically described and implemented. These algorithms are used in implementation of universal graphic interface, which allows us to work with the arbitrary strategical game. For this thesis, three abstract antagonistic games, Abalone, Terrace and Yinsh were described and implemented.

The introductory part describes chosen games, their rules and history. The second part addresses the theoretical description of the game trees, used algorithms and their improvements. Design and realization are described in the third part, which engage the way, how algorithms, heuristic functions and further selected parts of application are implemented. The fourth part is devoted to the measuring and comparing properties of each algorithm and heuristic function. The final part contains a summarization of achieved results and proposals to the future for the further research.



# Obsah

<b>Prohlášení.....</b>	<b>iii</b>
<b>Poděkování .....</b>	<b>v</b>
<b>Abstrakt.....</b>	<b>vii</b>
<b>Abstract .....</b>	<b>vii</b>
<b>Obsah .....</b>	<b>ix</b>
<b>Seznam obrázků .....</b>	<b>xv</b>
<b>Seznam tabulek.....</b>	<b>xvii</b>
<b>Část I - Úvod.....</b>	<b>1</b>
1. <i>Výběr her.....</i>	1
2. <i>Popis vybraných her .....</i>	2
2.1. <i>Abalone.....</i>	2
2.1.1. <i>Původ .....</i>	2
2.1.2. <i>Varianty a rozšíření.....</i>	2
2.1.3. <i>Specifikace hry .....</i>	3
2.1.4. <i>Pravidla hry.....</i>	4
2.1.4.1. <i>Cíl hry.....</i>	4
2.1.4.2. <i>Pohyb.....</i>	4
2.1.4.3. <i>Hra pro více než 2 hráče a alternativní rozestavení .....</i>	6
2.1.5. <i>Notace tahů .....</i>	7
2.1.6. <i>Diskuze.....</i>	9
2.1.6.1. <i>Předcházení remíze .....</i>	9
2.2. <i>Terrace .....</i>	9
2.2.1. <i>Původ .....</i>	9
2.2.2. <i>Varianty a rozšíření.....</i>	10
2.2.3. <i>Specifikace hry .....</i>	10
2.2.4. <i>Pravidla .....</i>	10
2.2.4.1. <i>Cíl hry.....</i>	12
2.2.4.2. <i>Pohyb.....</i>	12
2.2.4.3. <i>Další pravidla a alternativní rozestavení .....</i>	12
2.2.5. <i>Notace tahů .....</i>	13
2.3. <i>Yinsh.....</i>	14
2.3.1. <i>Původ .....</i>	14

2.3.2. Varianty a rozšíření.....	14
2.3.3. Specifikace hry.....	14
2.3.4. Pravidla hry.....	15
2.3.4.1. Cíl hry .....	15
2.3.4.2. Začátek hry .....	15
2.3.4.3. Fáze umisťování kroužků.....	15
2.3.4.4. Fáze tvorby řad.....	16
2.3.4.5. Vytvoření řady.....	16
2.3.4.6. Remíza.....	17
2.3.5. Notace tahů .....	17
<b>Část II - Principy umělé inteligence.....</b>	<b>19</b>
3. Optimální rozhodnutí ve hrách .....	19
4. Herní stromy .....	19
5. Nekompletní herní strom .....	20
6. Heuristická funkce.....	21
7. Složitost her .....	21
7.1. Složitost hry Abalone.....	22
7.1.1. Počet přípustných pozic .....	22
7.1.2. Složitost herního stromu .....	22
7.2. Složitost hry Terrace .....	23
7.2.1. Počet přípustných pozic .....	23
7.2.2. Složitost herního stromu .....	23
7.3. Složitost hry Yinsch.....	23
7.3.1. Počet přípustných pozic .....	23
7.3.2. Složitost herního stromu .....	24
8. Vyhledávací algoritmy umělé inteligence.....	24
8.1. Algoritmus Minimax .....	24
8.1.1. Použití.....	25
8.1.2. Pseudokód algoritmu .....	25
8.1.3. Popis práce pseudokódu .....	26
8.1.4. Složitost algoritmu Minimax.....	27
8.2. Algoritmus Negamax .....	27
8.2.1. Pseudokód algoritmu .....	28

8.2.2. Popis práce pseudokódu.....	28
8.2.3. Složitost algoritmu Negamax.....	29
8.3. Algoritmus Alfa-Beta ořezávání .....	29
8.3.1. Použití .....	29
8.3.2. Pseudokód algoritmu.....	30
8.3.3. Popis práce pseudokódu.....	31
8.3.4. Složitost algoritmu Alfa-beta ořezávání.....	32
9. Zlepšující techniky vyhledávacích algoritmů .....	32
9.1. Řazení tahů ( <i>Move ordering</i> ) .....	32
9.2. Iterativní prohlubování ( <i>Iterative Deepening</i> ) .....	33
9.2.1. Pseudokód techniky.....	33
9.2.2. Popis práce pseudokódu.....	33
9.2.3. Navýšení počtu stavů.....	34
9.3. Transpoziční tabulka ( <i>Transposition Table</i> ) .....	34
9.3.1. Pseudokód techniky.....	34
9.3.2. Popis práce pseudokódu.....	36
9.3.3. Práce s transpoziční tabulkou.....	37
9.3.4. Zobrist Hash.....	37
9.3.5. Inkrementální hashování .....	38
10. Problematické aspekty vyhledávání.....	39
10.1. Nestabilita hledání ( <i>Search Instability</i> ) .....	39
10.2. Horizon efekt ( <i>Horizon Effect</i> ).....	39
<b>Část III - Návrh a realizace.....</b>	<b>40</b>
11. Použité technologie.....	40
12. Objektový model .....	40
12.1. Přehled tříd aplikace .....	40
12.2. Popis tříd a jejich funkcí .....	42
12.2.1. Třídy části <i>Logic</i> .....	42
12.2.2. Třídy části <i>View</i> .....	42
13. Grafická reprezentace herní desky.....	43
13.1. Hrací panel .....	43
13.1.1. Model vrstev hracího panelu.....	44
13.1.2. Vrstva hracího panelu.....	44

<i>14. Reprezentace startovních stavů</i> .....	45
<i>15. Logická a datová reprezentace herní desky</i> .....	46
<i>16. Reprezentace hracích prvků</i> .....	48
<i>17. Implementace heuristických funkcí</i> .....	49
17.1. Obecné vlastnosti heuristických funkcí .....	49
17.2. Abalone.....	50
17.2.1. Seznam vlastností.....	50
17.2.1.1. Vlastnosti při ohodnocování pozic .....	50
17.2.1.2. Vlastnosti při řazení tahů .....	52
17.2.2. Seznam heuristik hry Abalone.....	52
17.2.3. Tabulka vah jednotlivých heuristik.....	52
17.3. Terrace.....	53
17.3.1. Seznam vlastností.....	53
17.3.1.1. Vlastnosti při ohodnocování pozic .....	53
17.3.1.2. Vlastnosti heuristik při řazení tahů .....	55
17.3.2. Seznam heuristik hry Terrace.....	55
17.3.3. Tabulka vah jednotlivých heuristik.....	55
17.4. Yinsh .....	56
17.4.1. Seznam vlastností.....	56
17.4.1.1. Vlastnosti ve fázi pokládání kroužků .....	56
17.4.1.1.1. Vlastnosti při ohodnocování pozic .....	56
17.4.1.1.2. Vlastnosti heuristik řazení tahů .....	57
17.4.1.2. Vlastnosti ve fázi tvorby řad.....	57
17.4.1.2.1. Vlastnosti při ohodnocování pozic .....	57
17.4.1.2.2. Vlastnosti heuristik řazení tahů .....	59
17.4.2. Seznam heuristik hry Yinsh.....	60
17.4.3. Tabulka vah jednotlivých heuristik.....	60
18. Implementace algoritmů umělé inteligence .....	61
18.1. Implementace algoritmu Best Move .....	61
18.2. Implementace algoritmu Negamax.....	62
18.2.1. Pseudokód algoritmu .....	62
18.2.2. Popis práce pseudokódu .....	64
18.3. Implementace algoritmu Alfa-Beta prořezávání .....	65

18.3.1. Pseudokód algoritmu.....	65
18.3.2. Popis práce pseudokódu.....	68
19. Implementace zlepšujících technik algoritmů umělé inteligence.....	68
19.1. Detaily implementace transpoziční tabulky .....	68
20. Repetiční remíza.....	69
21. Plugin systém .....	70
22. Skórování her .....	70
23. Cache (vyrovnávací paměť).....	71
24. Přidání vlastní hry .....	71
25. Parametry příkazové řádky .....	72
<b>Část IV - Testování umělé inteligence .....</b>	<b>73</b>
26. Testovací sestava .....	73
27. Porovnání výkonnosti heuristických funkcí .....	73
27.1. Výsledky u hry Abalone.....	74
27.2. Výsledky u hry Terrace.....	75
27.3. Výsledky u hry Yinsh .....	76
28. Měření efektivity algoritmů umělé inteligence a jejich zlepšujících technik .....	77
28.1. Měření na začátku hry .....	77
28.2. Měření uprostřed hry .....	79
28.3. Měření za celou hru .....	81
29. Porovnání výkonnosti algoritmů umělé inteligence a jejich zlepšujících technik .....	82
<b>Část V - Závěr .....</b>	<b>85</b>
30. Zhodnocení.....	85
31. Náměty pro budoucí práci.....	86
<b>Reference .....</b>	<b>87</b>
<b>Internetové odkazy .....</b>	<b>87</b>
<b>Literatura .....</b>	<b>90</b>
<b>A Seznam zkratek .....</b>	<b>91</b>
<b>B Uživatelská příručka .....</b>	<b>92</b>
B.1 Obecné .....	92
B.2 Hlavní okno aplikace .....	92
B.2.1 Stavový řádek .....	93
B.2.2 Panel historie tahů.....	93

B.2.3 Panel skóre .....	94
B.2.4 Hlavní menu .....	94
B.2.5 Panel nástrojů.....	96
<i>B.3 Přeuspořádání panelů grafického rozhraní.....</i>	97
<i>B.4 Dialogy aplikace.....</i>	98
B.4.1 Dialog konzole .....	98
B.4.2 Dialog pravidel.....	99
B.4.3 Dialog konce hry .....	99
B.4.4 Dialog výběru hry .....	100
B.4.5 Dialog nové hry.....	101
<i>B.5 Ovládání aplikace v průběhu hry .....</i>	103
B.5.1 Hra Abalone.....	103
B.5.2 Hra Terrace .....	104
B.5.3 Hra Yinsh.....	106
B.3.5.1 Fáze umisťování kroužků.....	106
B.3.5.2 Fáze tvorby řad.....	106
<b>C Instalační příručka .....</b>	<b>109</b>
C.1 Instalační požadavky aplikace .....	109
C.2 Instalace prostředí Java .....	109
C.3 Spuštění aplikace .....	109
<b>D Obsah přiloženého CD .....</b>	<b>110</b>
<b>E Fyzické přílohy diplomové práce .....</b>	<b>110</b>

# Seznam obrázků

Obr. 1 - Abalone z [1].....	3
Obr. 2 - Abalone Quattro z [39]. .....	3
Obr. 3 - Abalone Travel z [1].....	3
Obr. 4 - Základní postavení pro 2 hráče z [37].....	4
Obr. 5 - Ukázka pohybu 1,2 a 3 koulemi. ....	5
Obr. 6 - Tah do strany z [37]. .....	5
Obr. 7 - Dopředný tah z [37]. .....	5
Obr. 8 - Demonstrace všech možných způsobů přetlačení. ....	6
Obr. 9 - Podporované rozestavení pro 3 a 4 hráče. ....	6
Obr. 10 - Podporované nestandardní rozestavení pro 2 hráče. ....	7
Obr. 11 - Souřadnice notací hry Abalone z [37].....	8
Obr. 12 - Příklad notace pro posun 1 koule. ....	8
Obr. 13 - Příklad notace pro více koulí. ....	9
Obr. 14 - Terrace 6x6 z [1]. .....	10
Obr. 15 - Terrace z [1].....	10
Obr. 16 - Základní postavení pro 2 hráče.....	11
Obr. 17 - Základní postavení pro 3 hráče.....	11
Obr. 18 - Základní postavení pro 4 hráče.....	11
Obr. 19 - Vyhodení, úprava z [13].....	12
Obr. 20 - Pohyb na stejně úrovni, úprava z [13]. .....	12
Obr. 21 - Pohyb nahoru, úprava z [13].....	12
Obr. 22 - Pohyb dolů, úprava z [13]. .....	12
Obr. 23 - Alternativní rozestavení pro 2 hráče - <i>Short</i> .....	13
Obr. 24 - Alternativní rozestavení pro 2 hráče – <i>Second Player Master</i> .....	13
Obr. 25 - Yinsh z [1].....	14
Obr. 26 - Prázdná deska hry Yinsh. ....	15
Obr. 27 - Ukázka odebrání samostatné řady. ....	16
Obr. 28 - Ukázka odebírání více řad či vzájemně propojených. ....	17
Obr. 29 - Herní Strom z [54].....	20
Obr. 30 - Procházení algoritmu Minimax herním stromem. ....	25
Obr. 31 - Průchod algoritmu Alfa-Beta herním stromem. ....	30
Obr. 32 - Přehled tříd. ....	41

Obr. 33 - Poloha komponenty GamePanel v aplikaci .....	43
Obr. 34 - Rozložení vrstev komponenty GamePanel.....	44
Obr. 35 - Detailní pohled na jednu vrstvu komponenty GamePanel. ....	45
Obr. 36 - Diagram užití startovních stavů. ....	46
Obr. 37 - Souřadnice polí hry Abalone v implementaci. ....	47
Obr. 38 - Souřadnice polí hry Terrace v implementaci. ....	47
Obr. 39 - Souřadnice polí hry Yinsch v implementaci. ....	48
Obr. 40 - Základní pozice položení kroužků u hry Yinsch pro měření. ....	77
Obr. 41 - Výchozí rozpoložení kamenů u hry Abalone pro měření uprostřed hry. ....	79
Obr. 42 - Výchozí rozpoložení kamenů u hry Terrace pro měření uprostřed hry.....	79
Obr. 43 - Výchozí rozpoložení u hry Yinsch pro měření uprostřed hry. ....	80
Obr. 44 - Popis úvodní obrazovky.....	92
Obr. 45 - Popis stavového řádku v tahu lidského hráče .....	93
Obr. 46 - Popis stavového řádku v tahu počítače .....	93
Obr. 47 - Panel historie tahů. ....	94
Obr. 48 - Panel skóre.....	94
Obr. 49 - Hlavní menu a jeho funkce.....	94
Obr. 50 - Popis panelu nástrojů.....	96
Obr. 51 - Oblast panelu, určená na vyvolání přetažení. ....	97
Obr. 52 - Demonstrace přetažení panelu k jiné straně aplikace. ....	97
Obr. 53 - Demonstrace změny polohy panelu skóre, historie tahů po přetažení. ....	97
Obr. 54 - Panel přemístěný mimo okno aplikace. ....	98
Obr. 55 - Dialog konzole. ....	98
Obr. 56 - Dialog pravidel.....	99
Obr. 57 - Popis dialogu konce hry.....	99
Obr. 58 - Popis dialogu výběru hry. ....	100
Obr. 59 - Popis dialogu nové hry. ....	101
Obr. 60 - Dialog nové hry se zobrazením dalších specifických vlastností.....	101
Obr. 61 - Abalone - výběr kamenu.....	103
Obr. 62 - Abalone – výběr cílové pozice. ....	104
Obr. 63 - Terrace – výběr kamenu.....	105
Obr. 64 - Terrace – výběr cílové pozice. ....	105
Obr. 65 - Yinsch – výběr pole pro kroužek. ....	106

Obr. 66 - Yinsh – výběr cílové pozice kroužku.....	107
Obr. 67 - Yinsh – výběr řady na odebrání z více možností.....	107
Obr. 68 - Yinsh – výběr kroužku na odebrání.....	108

## Seznam tabulek

Tab. 1 - Váhy vlastností heuristik hry Abalone.....	53
Tab. 2 - Váhy vlastností heuristik hry Terrace.....	56
Tab. 3 - Váhy vlastností heuristik hry Yinsh. ....	61
Tab. 4 - Definované skórování pro jednotlivé hry.....	70
Tab. 5 - Akce měnící skórování .....	71
Tab. 6 - Skóre získané v jednotlivých vzájemných hrách heuristik u Abalone. ....	74
Tab. 7 - Body udělené za jednotlivé vzájemné hry heuristik u Abalone. ....	74
Tab. 8 - Průměrné parametry naměřené během všech vzájemných her heuristik. ....	74
Tab. 9 - Skóre získané v jednotlivých vzájemných hrách heuristik u Terrace. ....	75
Tab. 10 - Body udělené za jednotlivé vzájemné hry heuristik u Terrace. ....	75
Tab. 11 - Průměrné parametry naměřené během všech vzájemných her heuristik u Terrace. ....	75
Tab. 12 - Skóre získané v jednotlivých vzájemných hrách heuristik u Yinsh. ....	76
Tab. 13 - Body udělené za jednotlivé vzájemné hry heuristik u Yinsh.....	76
Tab. 14 - Průměrné parametry naměřené během všech vzájemných her heuristik u Yinsh.....	76
Tab. 15 - Počet prozkoumaných stavů k získání výsledného tahu na začátku hry. ....	78
Tab. 16 - Počet prozkoumaných stavů k získání výsledného tahu na začátku hry. ....	78
Tab. 17 - Počet prozkoumaných stavů k získání výsledného tahu na začátku hry. ....	78
Tab. 18 - Počet prozkoumaných stavů k získání výsledného tahu uprostřed hry. ....	80
Tab. 19 - Počet prozkoumaných stavů k získání výsledného tahu uprostřed hry. ....	80
Tab. 20 - Počet prozkoumaných stavů k získání výsledného tahu uprostřed hry. ....	81
Tab. 21 - Průměrný počet prozkoumaných stavů k získání výsledného tahu v průběhu celé hry. .	81
Tab. 22 - Průměrný počet prozkoumaných stavů k získání výsledného tahu v průběhu celé hry. .	82
Tab. 23 - Průměrný počet prozkoumaných stavů k získání výsledného tahu v průběhu celé hry. .	82
Tab. 24 - Skóre získané v jednotlivých vzájemných hrách různých algoritmů. ....	83
Tab. 25 - Body udělené v jednotlivých vzájemných hrách různých algoritmů. ....	83
Tab. 26 - Naměřená průměrná hloubka tahů v rámci vzájemných her různých algoritmů. ....	83



# Část I - Úvod

Cílem diplomové práce bylo prozkoumat použití algoritmů a technik umělé inteligence deskových her. Pomocí těchto technik poté vytvořit rozhraní, kde by byla možnost tyto techniky porovnat a vyzkoumat jejich vlastnosti. U implementace byl kladen důraz na její schopnost zahrát kvalitní tah, a tím konkurovat lidskému protivníku. Stávající implementace deskových her se velmi často konkrétně zaměřují na zvolenou hru. Na rozdíl od tohoto přístupu, byla v implementaci této práce snaha se nespecializovat na zvolenou hru, nýbrž vytvořit univerzální rozhraní pro více her s možností budoucího doprogramování dalších. Aplikace nemůže konkurovat zaběhlým úzce specializovaným implementacím, ale za to díky volbě neobvyklých deskových her nebo možnosti doimplementování dalších her, přináší možnost si tyto hry vůbec zahrát na počítači, popř. prověřit schopnosti hráče proti různým úrovním umělé inteligence.

## 1. Výběr her

Jak již bylo uvedeno v úvodu, vytvořená aplikace bude univerzální a nebude se úzce specializovat na konkrétní hru. Aby mohly být algoritmy umělé inteligence rozumně testovány, a aby si proti nim mohl vůbec někdo "zahrát" je nutné zvolit nějakou konkrétní hru. Jelikož jsou velké rozdíly ve hraní mezi jednotlivými hrami, tak jsem se rozhodl, že bude pro implementaci vybráno několik různých her. Pomocí nichž se demonstruje univerzálnost rozhraní a vliv různorodých hracích prvků na umělou inteligenci.

Pokud se podíváme na svět deskových her, tak zjistíme, že existuje velké množství deskových her [1]. Deskové hry můžeme dělit podle několika kategorií.

- Podle tematicnosti [2] ( např. abstraktní, karetní, fantasy, válečné, ekonomické ).
- Podle použitých herních mechanismů [3] ( např. házení kostkou, modulární deska, obchodování, spekulativní, kooperativní, vybíraní karet ).
- Podle počtu hráčů.
- Podle stáří hry [4] ( zde hlavně rozlišujeme klasické hry - např. šachy a hry moderní např. monopoly ).
- Podle typu hry z pohledu teorie her [5] ( kooperativní x nekooperativní, hry s nulovým x nenulovým součtem, hry s neúplnou informací x úplnou informací, ad. ).

Z těchto kategorií jsem se zaměřil na rozčlenění z pohledu teorie her, které má velký vliv na použitelnost popsaných algoritmů umělé inteligence. **Vybral jsem nekooperativní hry s nulovým součtem a úplnou informací.**

Nekooperativní hra [5] ( *non-cooperative game* ), znamená, že jednotliví hráči provádějí své rozhodnutí nezávisle a nemají možnost mezi sebou vytvářet dohody, které se musí dodržovat.

Hra s nulovým součtem [5] ( *zero-sum game* ) je taková hra, kdy celkový zisk všech hráčů pro každou možnou strategii je v součtu nula ( resp. zisk jednoho hráče je úměrný ztrátě ostatních hráčů ). Do této kategorie spadá velké množství her pro 2 hráče.

Hry s úplnou informací [5] (*perfect information game*) jsou takové, kde všichni hráči vědí o všech tazích, které byly provedeny a hráči znají všechny informace o aktuálním stavu hry. Z toho například vyplývá, že karetní hry nejsou hry s úplnou informací, protože hráči nevidí karty ostatních hráčů nebo karty v balíku.

Do těchto typů her, které byly popsány spadá většina klasických her, jako šachy, dáma nebo hry jako Go nebo piškvorky. Právě na tuto kategorii her se výzkum umělé inteligence v deskových hrách zaměřuje.

Z této kategorie (nekooperativní hry s nulovým součtem a úplnou informací) jsem vybral hry, které mají odlišné herní mechanizmy a různou hrací desku. Jedná se o hry abstraktní ( abstract games ). V následující kapitole budou jednotlivé hry představeny.

## 2. Popis vybraných her

### 2.1. Abalone

#### 2.1.1. Původ

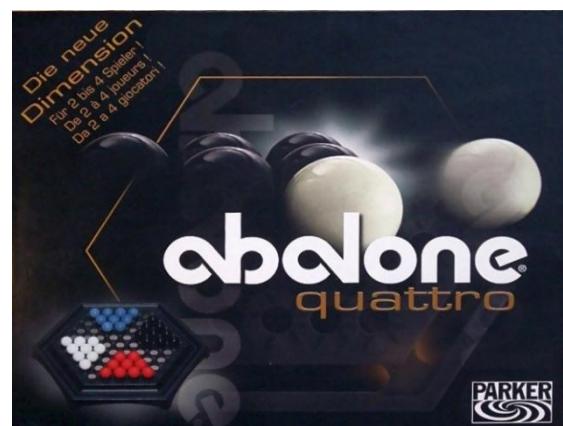
Hra Abalone byla vynalezena v roce 1987 Laurentem Levim a Michelem Laletem. Hře se podařilo uspět především v Evropě, díky čemuž se začaly od roku 1997 pořádat mezinárodní turnaje v rámci MSO [7]. Byl to první rok, kdy MSO začalo pořádat roční mezinárodní finále. MSO je považováno jako hlavní turnaj, kde se soutěží v Abalone, vzhledem k tomu, že hra nemá vlastní turnaj. V roce 1999 byla v MSO podepsána dohoda o změně startovní pozice na "Belgian Diasy", aby se hra revitalizovala [6] (důvody viz níže). V turnajích se staly hráči České Republiky úspěšní, když Jan Šťastna a Vojtěch Hrabal dohromady čtyřikrát vyhráli turnaj. Hra za svůj krátký život získala několik ocenění, mezi kterými zmiňme např. v roce 1990 ocenění Mensa Select [8].

#### 2.1.2. Varianty a rozšíření

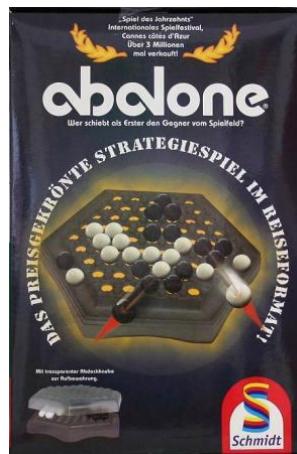
Hra Abalone se vyskytuje na trhu v několika provedeních, klasické Abalone [9], Abalone Travel (popř. Abalone Mini) se stejnými pravidly v zmenšeném cestovním provedení a nejnovější Abalone Quattro [11], které má nový design desky a je až pro 4 hráče oproti 2 hráčům v klasické variantě. Počet hráčů lze také navýšit přídavnou sadou "Abalone Extra Player Marbles" [10], která umožní hrát hru s 1 hráčem navíc na sadu. Takto lze hrát až s 6 hráči. **V této prací se budeme soustředit na variantu Abalone Quattro.**



Obr. 1 - Abalone z [1].



Obr. 2 - Abalone Quattro z [39].



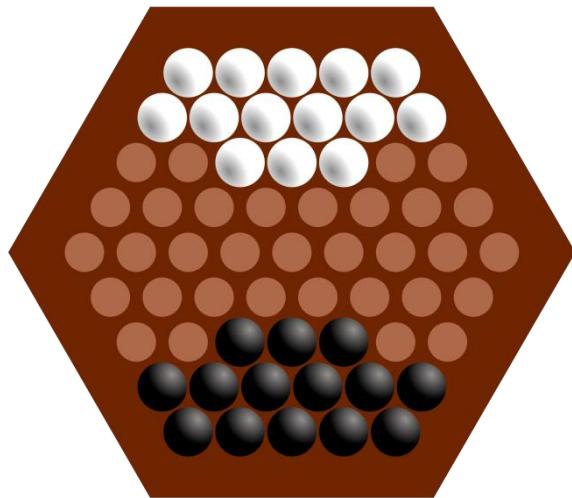
Obr. 3 - Abalone Travel z [1].

### 2.1.3. Specifikace hry

<i>Název hry</i>	Abalone Quattro (dále bude uváděno jako Abalone)
<i>Návrh</i>	Laurent Levi, Michel Valet
<i>Publikováno</i>	2003
<i>Počet hráčů</i>	2-4
<i>Hrací čas</i>	40m
<i>Kategorie</i>	Abstraktní hra
<i>Herní mechaniky</i>	pohyb po ploše, konstrukce vzorů

## 2.1.4. Pravidla hry

Abalone se hraje na šestiúhelníkové desce, která se skládá z 61 šestiúhelníkových polí seskupených do šestiúhelníku o straně 5 polí. Každý hráč má 14 koulí jedné barvy. Základní postavení pro 2 hráče je zobrazeno na následujícím obrázku 4<sup>1</sup>.



Obr. 4 - Základní postavení pro 2 hráče z [37].

### 2.1.4.1. Cíl hry

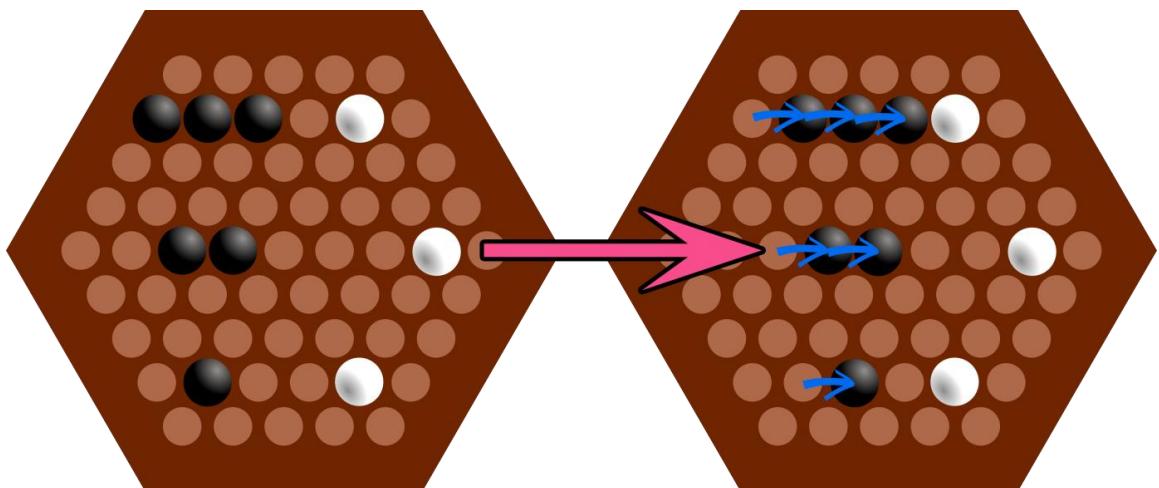
Cílem hry je vytlačit 6 soupeřových kouli z desky. Hráči se střídají v tazích, přičemž bílý začíná.

### 2.1.4.2. Pohyb

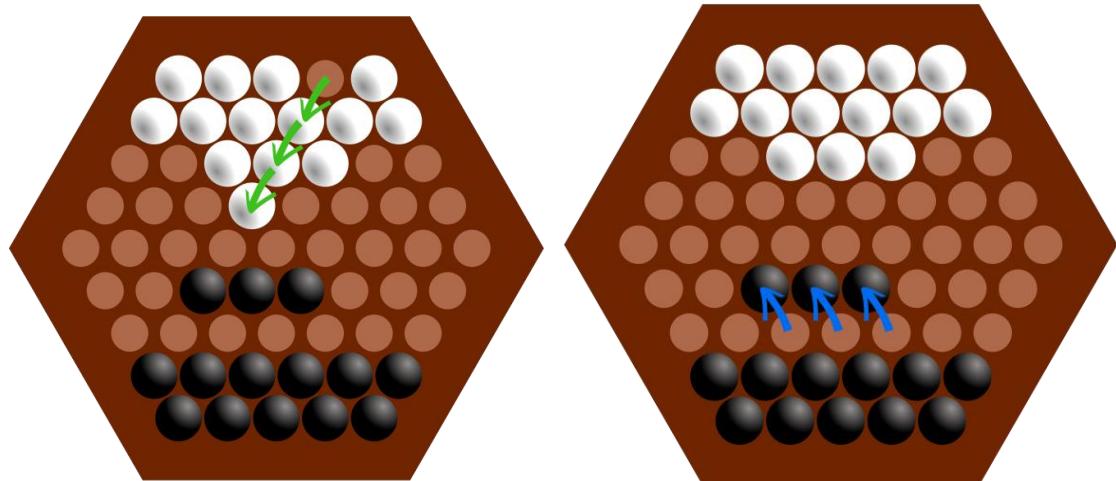
Hráč na tahu může posunout 1, 2 nebo 3 jeho koule o jedno pole, v kterémkoliv ze 6 možných směrů. V případě posouvání 2 nebo 3 koulí, koule musí stát v řadě a vzájemně sousedit. Pak každá tato koule z nich se posouvá o jedno pole a všechny se posouvají ve stejném směru. Pokud se koule v řadě pohybují paralelně s ní, jde o dopředný tah, jinak jde o tah do strany.

---

<sup>1</sup> Přiřazení barev kamenů u obrázků Abalone zde v teoretické sekci se neshoduje s přiřazením



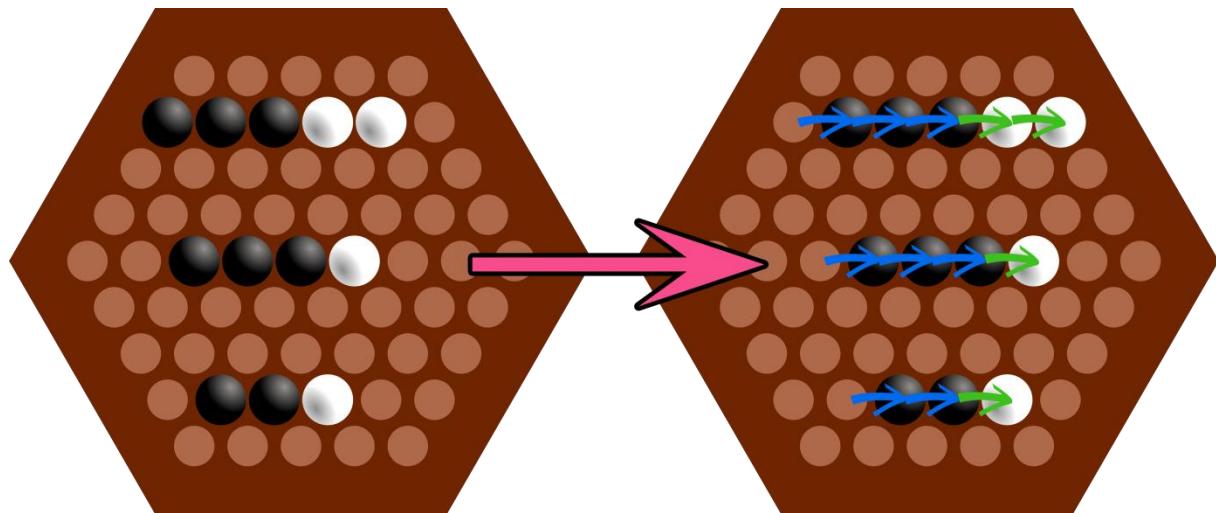
Obr. 5 - Ukázka pohybu 1,2 a 3 koulemi. Levý obrázek je stav před tahem, pravý po tahu. Úprava z [37]



Obr. 7 - Dopředný tah z [37].

Obr. 6 - Tah do strany z [37].

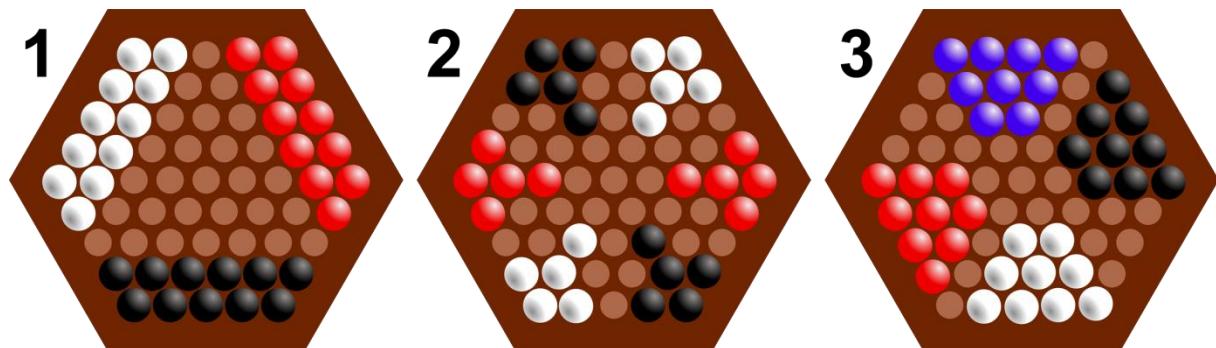
Pohybem koulí lze přetlačovat koule soupeřů. Při přetlačování koulí platí vždy, že více koulí přetlačí méně koulí. To znamená, že hráč může 2 koulemi přetlačit 1 soupeřovu kouli, 3 koulemi přetlačit 2 nebo 1 soupeřovu kouli. Přetlačovat lze pouze při dovedném tahu a přetlačují se pouze soupeřovy koule, ne vlastní. Za přetlačovanými kameny musí být volné místo nebo kraj desky.



Obr. 8 - Demonstrační všechny možné způsoby přetlačení. Levý obrázek je stav před tahem, pravý po tahu. Úprava z [37]

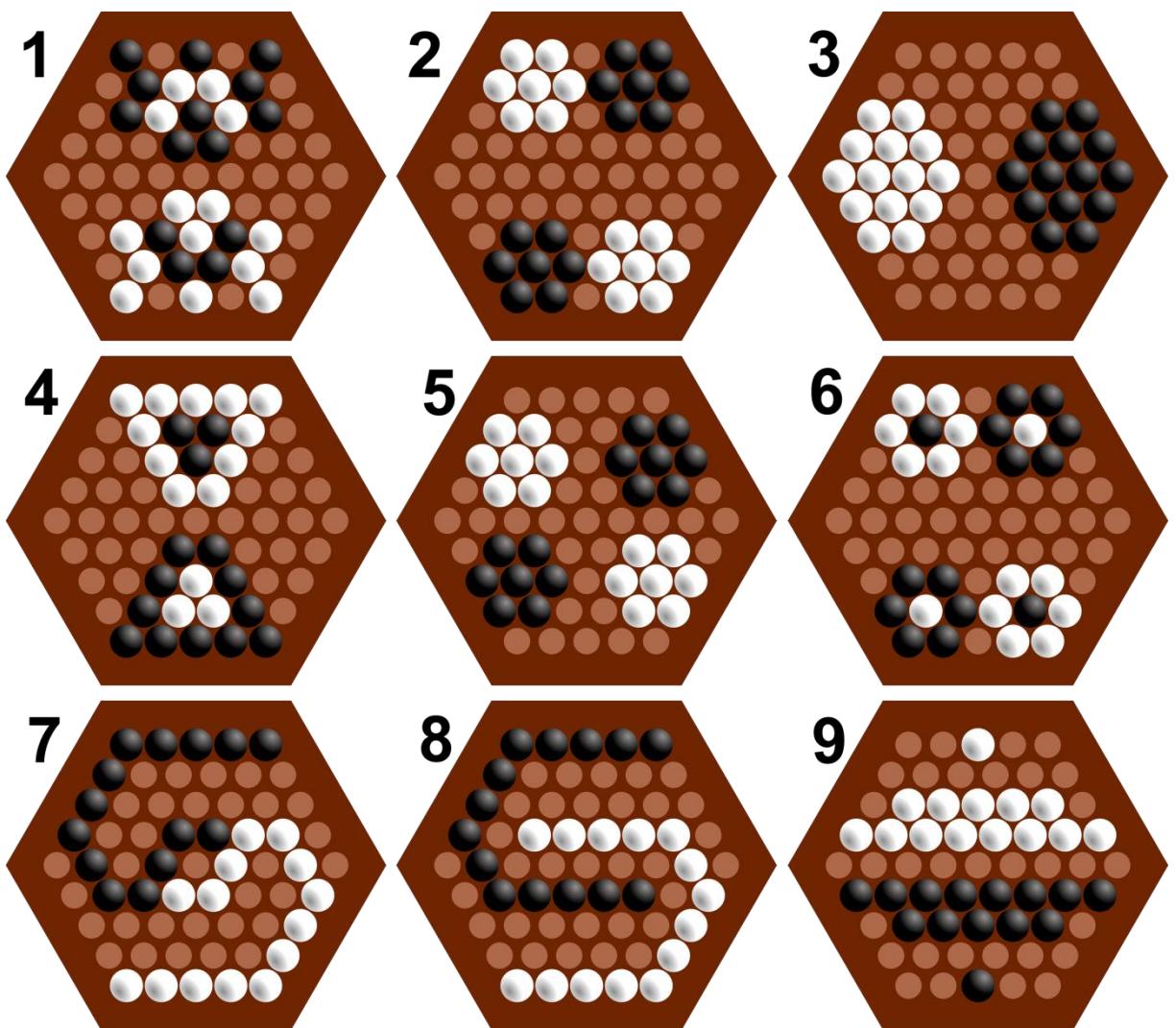
#### 2.1.4.3. Hra pro více než 2 hráče a alternativní rozestavení

Pravidla pro hru 3, 4 hráčů se liší tím, že se používá více barev koulí, přičemž každý hráč má svou vlastní barvu. Liší se také základním rozestavením kouli a jejich počátečním počtem, viz následující obrázek 9. Ostatní pravidla jsou stejná.



Obr. 9 - Podporované rozestavení pro 3 a 4 hráče. 1 – základní rozestavení pro 3 hráče, 2 – alternativní rozestavení pro 3 hráče (variate), 3 – základní rozestavení pro 4 hráče. Úprava z [37].

Hra kromě základního rozestavení umožňuje alternativní rozestavení. V následujícím obrázku 10 jsou vyobrazeny alternativní rozestavení podporované implementací.

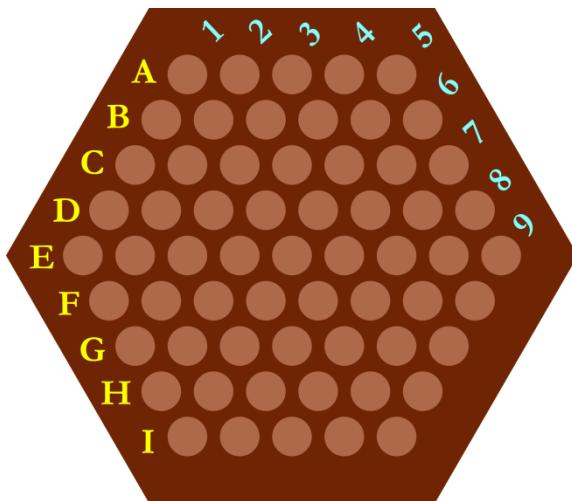


Obr. 10 - Podporované nestandardní rozestavení pro 2 hráče. 1 – Alien Attack, 2 – Belgian Daisy, 3 – Face 2 Face, 4- Fujiyama, 5 – German Daisy, 6- Dutch Daisy, 7 – Snakes, 8 – Snakes Variation, 9 – The Wall Úprava z [37].

### 2.1.5. Notace tahů

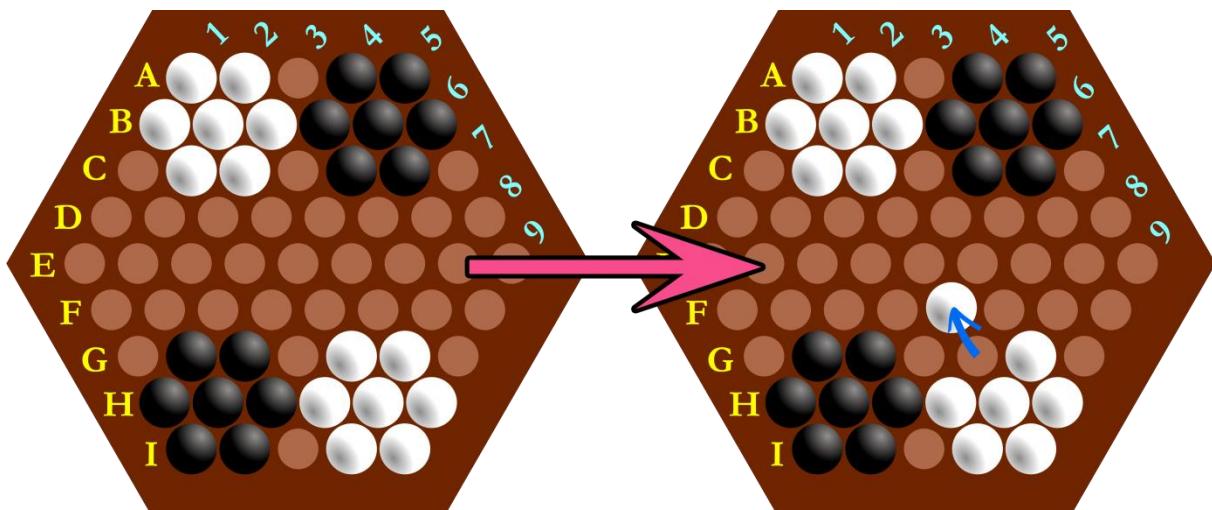
Implementace používá notací podle [12].

Každé pole se zapisuje ve tvaru RS, kde "R" je řádek a "S" je sloupec. Notace jednotlivých polí zobrazuje následující obrázek 11.



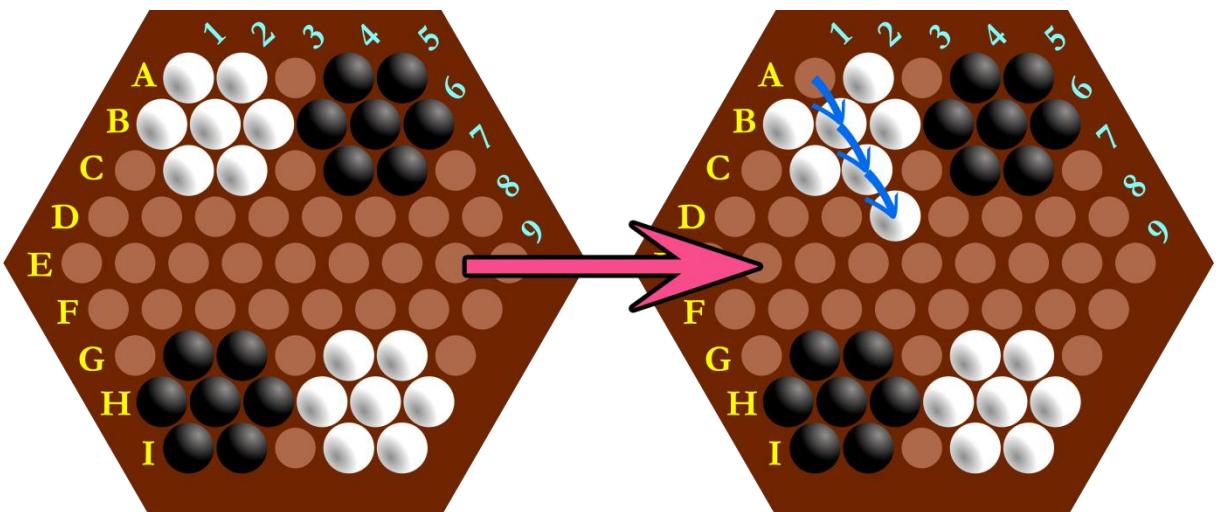
Obr. 11 - Souřadnice notací hry Abalone z [37].

Pokud hráč posouvá 1 kouli, tah se zapisuje ve formátu "R1S1, R2S2", kde "R1" a "S1" je původní řádek a sloupec koule, "R2" a "S2" je nový řádek a sloupec koule po provedení tahu.



Obr. 12 - Příklad notace pro posun 1 koule. Levý obrázek je stav před tahem, pravý po tahu. Notace vyobrazeného tahu je „g8,f7“. Úprava z [37].

Pokud hráč posouvá 2 nebo 3 koule, tak se tah zapisuje ve formátu "R1S1-R2S2,R3S3", kde "R1" a "S1" je původní poloha začátku řady vlastní koulí a "R2" a "S2" je poloha jejího konce. Pořadí zápisu "R1S1-R2S2" se určuje podle alfanumerického vzestupného pořadí. Poloha "R3" a "S3" je vždy přilehlá pozice ve směru tahu od "R1S1" nebo "R2S2". Demonstraci zápisu takových těchto tahů ukazuje následující obrázek 13.



Obr. 13 - Příklad notace pro více koulí. Levý obrázek je stav před tahem, pravý po tahu. Notace vyobrazeného tahu je „a1-c3,b2“ nebo „a1-c3,d4“. Úprava z [37].

## 2.1.6. Diskuze

### 2.1.6.1. Předcházení remíze

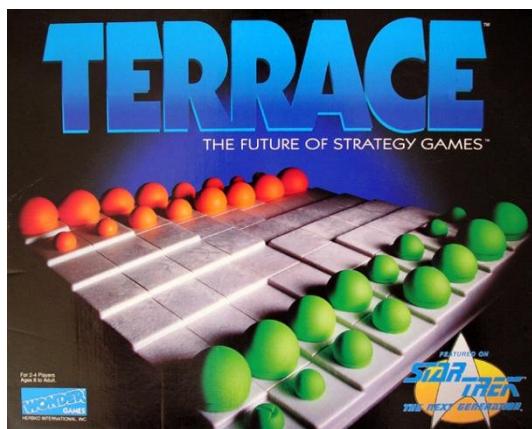
Podle postřehů ze hry a podle [6] dynamika hry trpí závažným problémem. Konzervativní hráč má možnost své koule seskupit do defenzivního postavení, kde může nekonečně odolávat útokům soupeřů. A vzhledem k tomu, že často útok je nebezpečnější pro útočníka než pro obránce, může se hra zablokovat v defensivním postavení všech hráčů. Tento fakt ještě zhoršuje nepřítomnost jakéhokoliv pravidla, které by tomu zabránilo ( např. limit opakování tahů ze šachů, aj. ). Jako nejlepší řešení tohoto problému defenzivní hry se jeví změna základního postavení. Ke změně základního postavení opravdu došlo a profesionální hráči se dohodli na turnajích hrát "daisy" postavení (viz obrázek 10).

## 2.2. Terrace

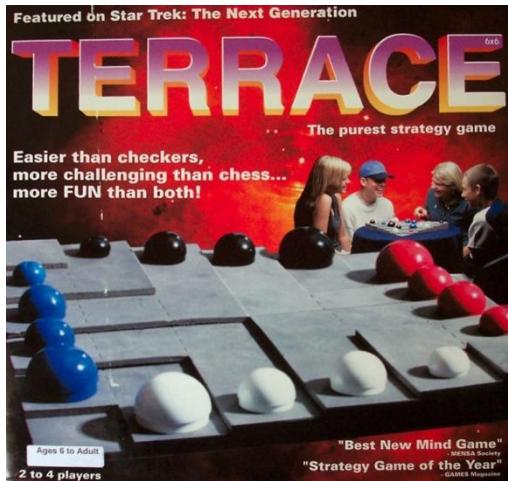
### 2.2.1. Původ

Terrace bylo vynalezeno v roce 1950 Antonem Dresdnem [13], který se snažil vytvořit myšlenkově náročnou hru, ale s jednoduchými pravidly. V roce 1988 objevil hru Buzz Siler, který ve spolupráci s Dresdnem vyladil pravidla a umožnil první vydání hry v roce 1991 [14]. V zálepí Terrace získala několik mezinárodních ocenění. Hra se stala známější také díky jejímu použití v seriálu Star Trek: New Generation. V roce 1997 byla uvedena nová upravená verze s velikostí 6x6 polí, hlavně kvůli právním problémům s původními vlastníky práv na hru.

## 2.2.2. Varianty a rozšíření



Obr. 15 - Terrace z [1].



Obr. 14 - Terrace 6x6 z [1].

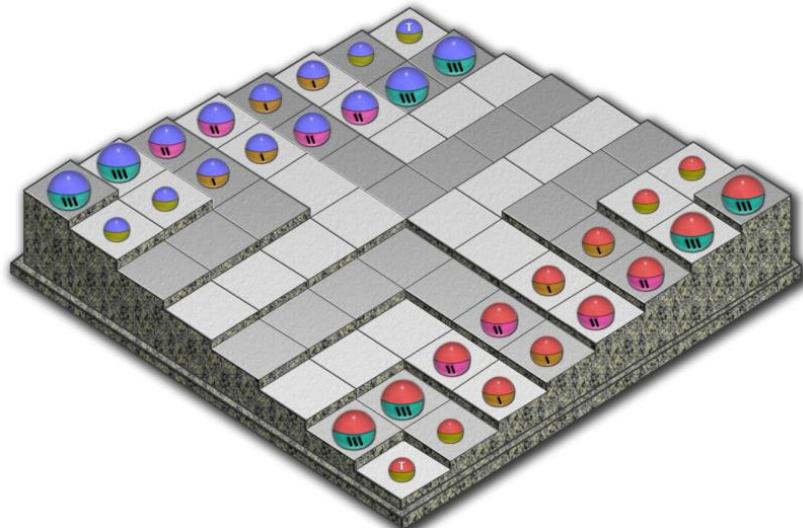
Původní varianta má desku velikosti 8x8 polí. Později byl vytvořena novější varianta, resp. revize původní hry s deskou velikost 6x6 polí a menším počtem kamenů v základním postavení. Tato varianta se nazývá Terrace 6x6. **V této práci se zaměříme na původní variantu Terrace.**

## 2.2.3. Specifikace hry

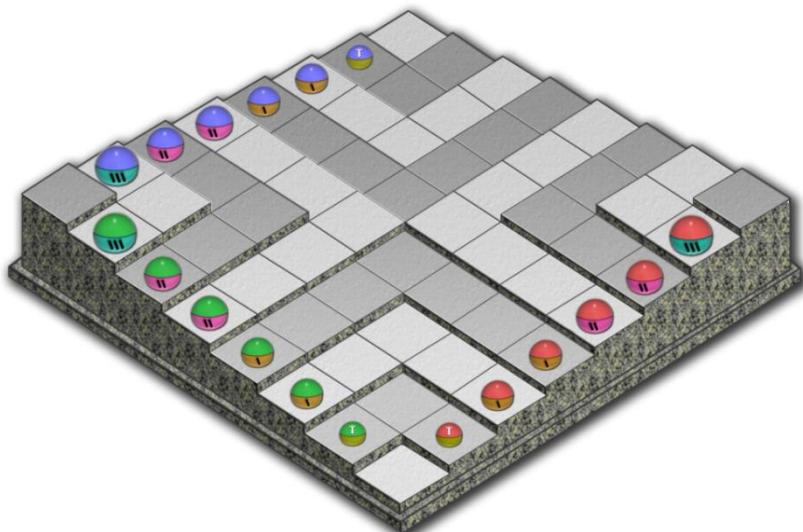
<i>Název hry</i>	Terrace
<i>Návrh</i>	Anton Dresden, Buzz Siler
<i>Publikováno</i>	1991
<i>Počet hráčů</i>	2-4
<i>Hrací čas</i>	40m
<i>Kategorie</i>	Abstraktní hra
<i>Herní mechaniky</i>	pohyb po ploše, obětování

## 2.2.4. Pravidla

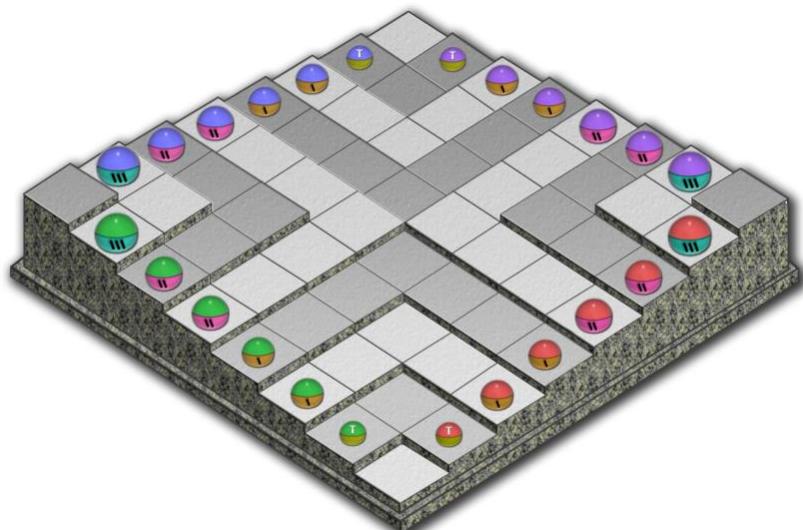
Terrace má desku o velikost  $8 \times 8$  polí stejné barvy. Deska je trojrozměrná, resp. jednotlivé pole mají různou výšku. Deska má 8 různých úrovní výšek, kde pole se stejnou výškou tvoří tvar písmene L v jednom kvadrantu (viz obrázek 16). Každý hráč má 16 kamenů (platí pro 2 hráče, jinak každý má 12 kamenů) stejného tvaru a bary, ale různé velikosti, která indikuje různou sílu kamene při vyhazování. Kameny mají 4 různé velikosti. Jeden z kamenů hráče je označen písmenem T (dále jako T kámen). Tento kámen má vždy nejmenší velikost. Základní postavení pro 2, 3, 4 hráče zobrazují následující obrázky.



Obr. 16 - Základní postavení pro 2 hráče.



Obr. 17 - Základní postavení pro 3 hráče.

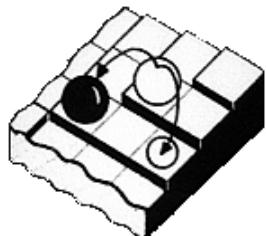


Obr. 18 - Základní postavení pro 4 hráče.

#### 2.2.4.1. Cíl hry

Cíle hry jsou dva. Prvním cílem hry je se dostat se s T kamenem do rohu desky s nejnižší výškou na protější straně. Druhým cílem je vyhodit soupeřům jejich T kameny (hráč pak vyhrává, když jeho T kámen zůstane jako poslední na desce).

#### 2.2.4.2. Pohyb



Obr. 19 - Vyhození,  
úprava z [13].



Obr. 20 - Pohyb na stejně  
úrovni, úprava z [13].



Obr. 21 - Pohyb  
nahoru, úprava z [13].



Obr. 22 - Pohyb dolů,  
úprava z [13].

Rozlišují se 4 základní tahy

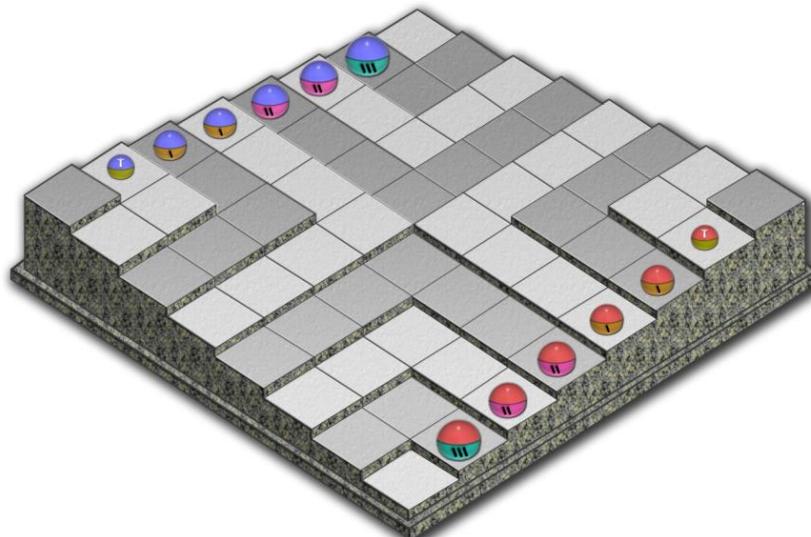
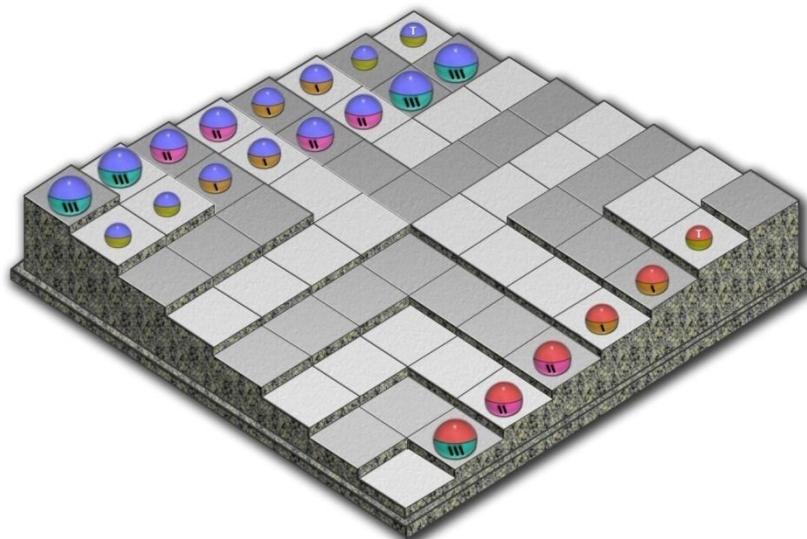
- Vyhození - Kámen může vyhodit jakýkoliv jiný kámen diagonálním pohybem dolů na sousední pole v nižší výšce. Vyhozený kámen je odstraněn ze hry. V případě vyhození T kamene, jeho vlastník prohrál a odstraní si z plochy všechny své kameny. Vyhodit lze i vlastní kámen.
- Pohyb na stejně úrovni - Kámen se může pohnout na jakékoli pole ve stejné výšce, které může dosáhnout bez přeskovení jakéhokoliv soupeřova kamene. Kámen se takto nemůže pohnout přes střed desky.
- Pohyb nahoru - Kámen se může pohnout rovně nebo diagonálně o úroveň výš na sousední pole ve vyšší úrovni.
- Pohyb dolů - Kámen se může pohnout rovně (ne diagonálně) o úroveň níž na sousední pole v nižší úrovni.

#### 2.2.4.3. Další pravidla a alternativní rozestavení

Je zvykem říci "Terrace" pokud hráč ohrozil soupeřův T kámen. A to takovým způsobem, že by mohl být vyhozen v dalším tahu. V případě, že soupeř se tomuto vyhození nemůže vyhnout, je možné říci "Terrace mate" (*Terrace Mate*). Toto pravidlo není povinné a v počítacové implementaci není zahrnuto.

V této hře je možné docílit remízy. To se stane v případě, že hráč na tahu nemůže udělat přípustný tah. Remíza také může nastat po dohodě hráčů.

Hra kromě standardních rozestavení nabízí i další počáteční rozestavení podle [15], které jsou zobrazeny na následujících obrázcích 23 a 24.

Obr. 23 - Alternativní rozestavení pro 2 hráče - *Short*.Obr. 24 - Alternativní rozestavení pro 2 hráče – *Second Player Master*.

## 2.2.5. Notace tahů

Oficiální notace tahů podle [15] je v následujícím formátu:

- *Normální tah:* K-R1S1-R2S2
- *Vyhození:* K-R1S1-R2S2:V

Kde "K" představuje kámen, se kterým se tállo. Obyčejné kameny se značí "A", "B", "C", "D", kde "A" je nejmenší kámen a "D" je největší kámen. T kameny se značí "T". "R1" a "S1" je řádek a sloupec pole, na kterém byl kámen před tahem a "R2" a "S2" je řádek a sloupec pole, kde je kámen po tahu. "V" je označení kamene, který byl vyhozen.

## 2.3. Yinsh

### 2.3.1. Původ

Yinsh byla vynalezená Krisem Burmem z Belgie v roce 2003 [18]. Jedná se tedy o poměrně novou hru. Hra je součástí skupiny her, která se nazývá GIPF projekt, který pochází od stejného autora. Gipf projekt tvoří 6 her ( Gipf, Tzaar, Zértz, Dvonn, Yinsh, Pünct), přičemž Yinsh je pátá hra ze souboru. Hra získala mnoho ocenění jako Mensa Select, Spiel Des Jahres ad. [19] a patří mezi nejlepších 50 her na serveru BoardGameGeek [1]. V roce 2008 k příležitosti 10. let existence Gipf projektu, se konalo v Praze mistrovství světa, které organizoval sám Kris Burm [17].

### 2.3.2. Varianty a rozšíření

Yinsh nemá žádné varianty. Yinsh je možno hrát jako samostatnou hru nebo jako hru v rámci do Gipf projektu, kde se hraje pak v kombinaci s ostatními hrami podle pravidel Gipf projektu [20]. Aby mohl být Yinsh použit v kombinaci s ostatními hrami byl v roce 2006 vydán Gipf Set 3, obsahující speciální kameny pro hru. **V této práci se budeme zabývat samostatnou hrou Yinsh s jejími samostatnými pravidly.**



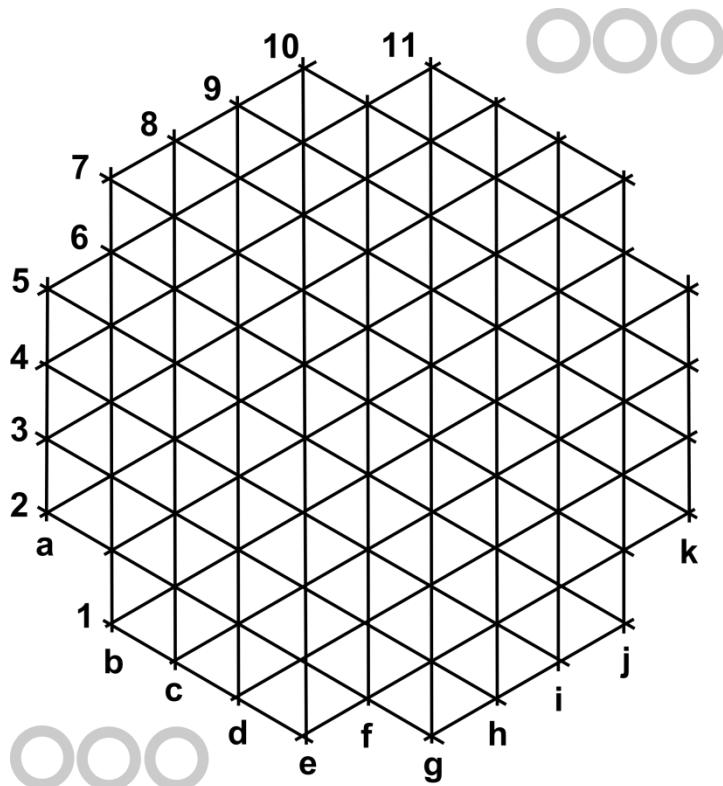
Obr. 25 - Yinsh z [1].

### 2.3.3. Specifikace hry

<i>Název hry</i>	Yinsh
<i>Návrh</i>	Kris Burm
<i>Publikováno</i>	2003
<i>Počet hráčů</i>	2
<i>Hrací čas</i>	30m
<i>Kategorie</i>	Abstraktní hra (Gipf Projekt)
<i>Herní mechaniky</i>	pohyb po ploše, umisťování kamenů, variabilní kameny

### 2.3.4. Pravidla hry

Hrací deska hry Y Cindy se skládá z 11 rovnoběžných přímek ve třech osách vzájemně otočených o 60 stupňů. Průsečíky přímek tvoří pole, kam se umisťují kameny a kroužky. Podrobněji tvar desky znázorňuje obrázek 26.



Obr. 26 - Prázdná deska hry Y Cindy.

Hru hrají 2 hráči - černý a bílý. Každý hráč má 5 kroužků. Oba hráči mají dohromady 51 kamenů, které z jedné strany jsou bílé, z druhé černé.

#### 2.3.4.1. Cíl hry

Cílem hry je jako první vytvořit 3 řady o 5 kamenech své barvy jdoucí za sebou v jakémkoli směru. Pokud se hraje "Blitz" varianta pravidel, tak pro vítězství stačí vytvořit pouze 1 řadu o 5 kamenech namísto 3.

#### 2.3.4.2. Začátek hry

Hra začíná s prázdnou hrací deskou. Hráči se střídají v tazích, přičemž bílý začíná.

#### 2.3.4.3. Fáze umisťování kroužků

Jako první hráč položí každý svých 5 kroužků na pole desky, která představující průsečíky čar. Kroužky se pokládají po jednom, přičemž hráči se střídají. Položením všech 10 kroužků je určena startovní pozice hry a končí tato fáze.

#### 2.3.4.4. Fáze tvorby řad

V každém tahu smí hráč pohnout s jedním ze svých kroužků následujícím způsobem.

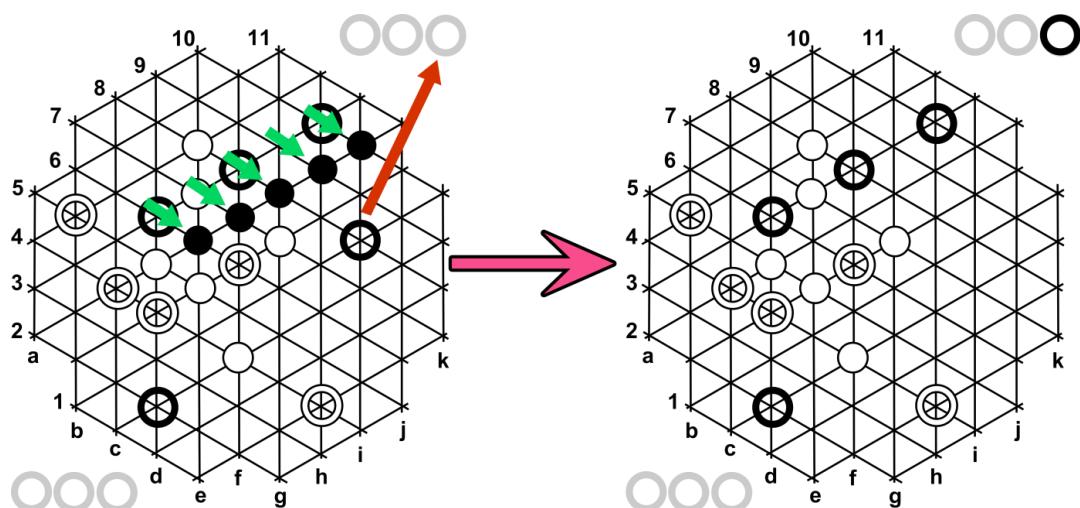
Do zvoleného kroužku umístí hráč kámen se svou barvou navrchu. Následně skočí kroužkem (bez kamenu) na přípustné pole. Přípustná pole jsou taková, která jsou prázdná a leží na přímce vycházející z tohoto pole na jeden z 6 směrů. Přičemž kroužek nesmí skákat přes jiný kroužek. Kroužek může přeskocit jeden či více kamenů jdoucích za sebou, ale pak musí skočit na první následující volné místo. Pokud kroužek přeskocil jakýkoliv kámen, tak tento kámen se převrátí, tzn. z černého kamene se stane bílý a naopak.

#### 2.3.4.5. Vytvoření řady

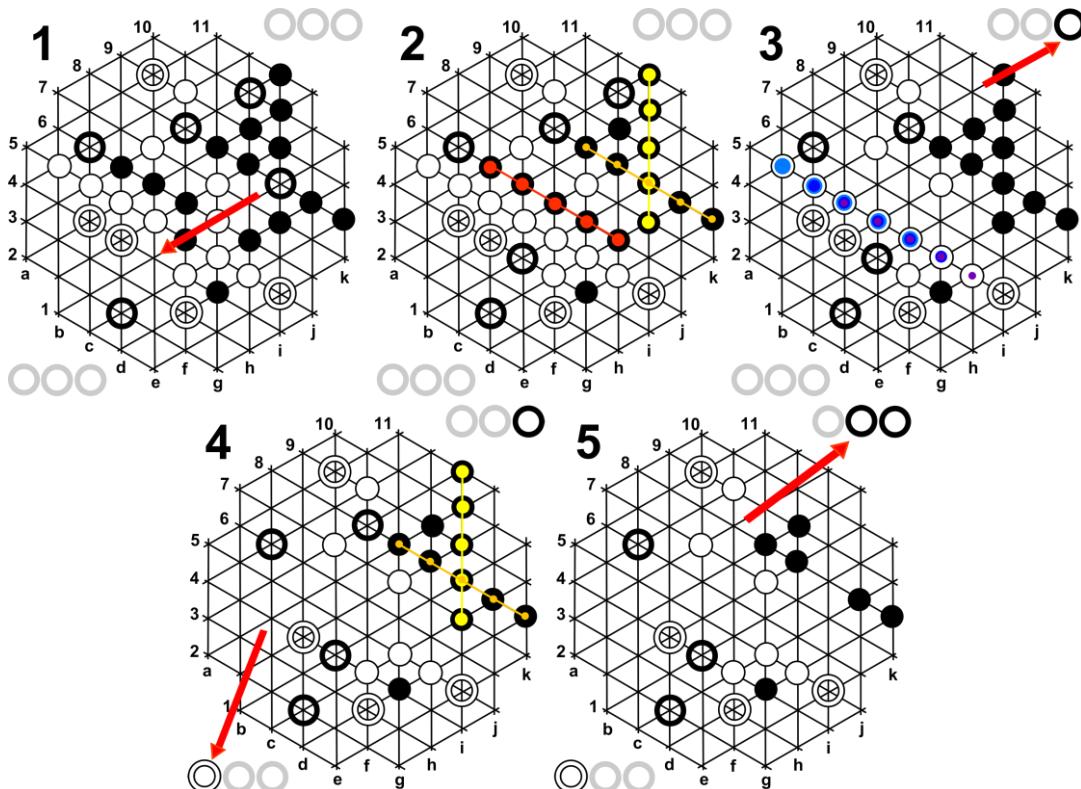
Pokud hráč po provedení tahu vytvořil řadu pěti po sobě jdoucích kamenů, tak vlastník tyto kameny odstraní ze hry a vybere si vlastní kroužek, který odstraní ze hry a položí ho na označená místo na kraji desky (viz obrázek 27). Z toho vyplývá, že hráč může vytvořit vlastní řadu 5 kamenů (dále bude uváděno také jako řada), nebo také soupeřovu řadu, což závisí na barvě kamenů.

V případě, že hráč vytvoří řadu více než 5 kamenů, jejich vlastník z nich vybere 5 po sobě jdoucích, které se odstraní.

Je také možné, že se v jednom tahu utvoří více řad, buď různé barvy, nebo jedné barvy spojené nebo nespojené. V tomto případě platí, že hráč, který provedl tah, odstraní řadu 5 kamenů, kterou vybere z možných variant pokud nějakou/é utvořil. Poté odeberá soupeř svojí řadu podobným způsobem. Následuje opět původní hráč, který odstraní jednu z možných vytvořených řad, pokud nějaká taková řada na desce zbyla. Tento postup se opakuje, dokud nejsou odebrány všechny řady (viz obrázek 28).



Obr. 27 - Ukázka odebrání samostatné řady. Černý vykonal tah, kde vytvořil vlastní řadu, vyznačenou zelenými šipkami na levé desce. Tyto kameny se odeberou a černý zvolí si kroužek, který odeberou z plochy a položí ho na kraj desky. Tento kroužek je zde vyznačen červenou šipkou. Na pravé desce je zobrazen stav desky po odebrání řady.



Obr. 28 - Ukázka odebírání více řad či vzájemně propojených. Na těchto obrázcích je vyobrazen složitější případ při odebírání řad. Na obrázku číslo 1, černý svým tahem vytvoří několik řad. Byly vytvořeny řady, jak pro bílého tak pro černého. Jelikož tah vykonal černý, bude odebírat svou řadu první. Vybírá se ze tří možností na obrázku 2 (označeny žlutě, oranžově a červeně). Černý odebírá jednu červeně označenou řadu s jedním svým kroužkem.

Hráči se střídají v odebírání řad, a proto nyní odebírá řadu bílý, který volí z možných variant označených tyrkysově, modře a fialově. To je zobrazeno na obrázku 3. V obrázku 4 bílý vybral řadu označenou tyrkysově a odebírá jí, současně také odebírá vlastní kroužek. Nyní opět odebírá řadu černý, který opět volí z variant označených žlutě a oranžově. Černý vybírá žlutou variantu a odebírá kameny a kroužek. To je vyobrazeno na obrázku 5. Jelikož už na desce není žádná zbývající řada na odebrání, hra dále pokračuje tahem bílého, protože poslední přesun kroužku provedl černý.

### 2.3.4.6. Remíza

Hra končí remízou v případě, že jakýkoli hráč nemůže položit kámen, protože žádné nezbyly (resp. všech 51 kamenů leží na desce).

### 2.3.5. Notace tahů

Oficiální nezkrácená notace Yinsh [21] se řídí podle následujících pravidel.

Zápis využívá souřadnicový systém, vyobrazený na obrázku 26 výše.

Zápis položení kroužku v fázi pokládání kroužků : S1R1

Kde "S1R1" je pole, na které byl kroužek položen, kde "S1" je písmeno sloupce a "R1" je číslo řádku.

Zápis položení kamenu a přemístění kroužku: S1R1-S2R2

Zde je "S1R1" je pole, na které byl položen kámen a "S2R2" pole, na které byl přemístěn kroužek

Pokud hráč odstraní 1 vlastní řadu, tak za zápis "S1R1-S2R2" přidá ";♣". V případě, že v tahu odstranil 2 vlastní řady, přidá se dvakrát ";♣". Podle podobného pravidla se zapíše tah v případě více odebraných řad.

Pokud oponent vytvořil hráči jeho řadu v předchozím tahu, tak zápis hráčova tahů obsahuje předponu "♣;" následovanou obvyklým "S1R1-S2R2". V případě, že šlo o 2 řady, před zápisem tahu bude dvakrát "♣;". Podle podobného pravidla se zapíše tah v případě, že šlo o více odebraných řad.

Přičemž notace odstraněné řady "♣" = "xS3R3-S4R4xS5R5"

"S1R1" je pole, na které byl položen kámen a "S2R2" je pole, na které byl přemístěn kroužek. Pole mezi "S3R3" a "S4R4" jsou pole, ze kterých byly odebrány kameny řady. Pole "S5R5" je pole, ze kterého byl odebrán kroužek.

Příklady notace při odstraňování řad z [21]:

- Vytvoření 2 řad ve vlastním tahu:

*i8-f5;xg5-g9xa5;xa5-e5xg2*

- Oponent vytvořil řadu pro hráče a hráč sám vytvořil svou druhou řadu:

*xg5-g9xa5;i8-f5;xa5-e5xg2*

- Oponent vytvořil pro hráče 2 řady:

*xg5-g9xa5;xa5-e5xg2;i8-f5*

Více detailní popis notací této hry se dá nalézt na oficiálních stránkách [21].

## Část II - Principy umělé inteligence

V této části se budeme zabývat teoretickým popisem algoritmů a myšlenek, které se využívají při implementování umělé inteligence. Tyto algoritmy nejsou přímo závislé na konkrétní hře, proto je můžeme použít jako základ pro umělou inteligenci všech implementovaných her. Popíšeme si postupně jednotlivé algoritmy a způsob, jak je možné je implementovat. Rovněž také si uvedeme jejich možné vylepšující techniky.

### 3. Optimální rozhodnutí ve hrách

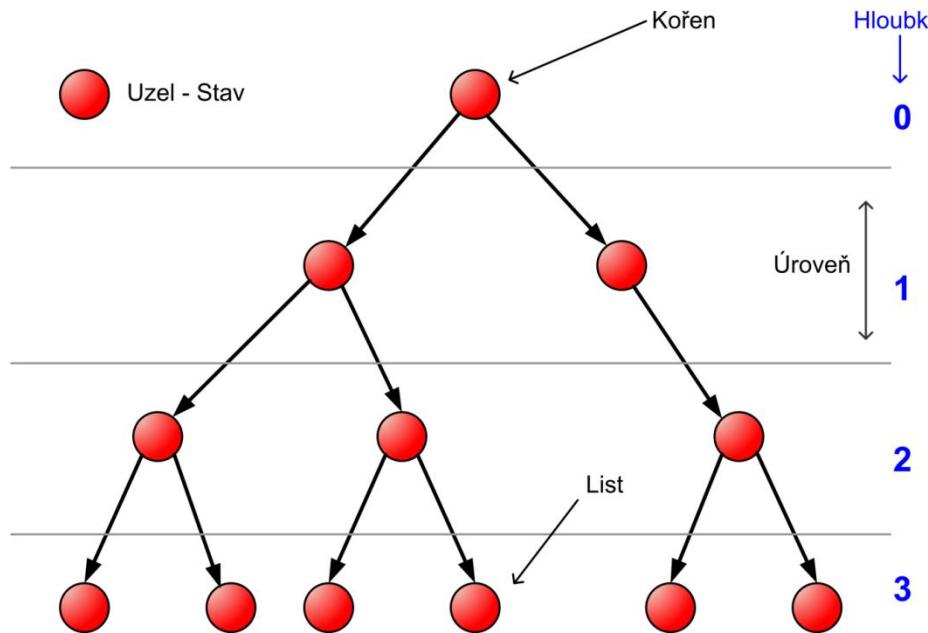
Podle [22] uvažujme tahovou hru s dvěma hráči s nulovým součtem a perfektní informací (právě takové budeme později implementovat – viz úvodní část). Taková hra muže být formulována jako vyhledávací problém (*search problem*). V takové hře, každý ví výsledek tahů, které mohou být zahrány. Abychom mohli vyhodnotit tyto výsledky, musíme také myslet za protihráče a zjistit jeho výsledky tahů. V takovéto hře, zisk jednoho hráče představuje ztrátu druhého. Takovou hru můžeme popsat následujícími částmi [23]:

- Výchozí stav, který popisuje hrací desku a základní postavení kamenů. Zároveň také popisuje, který hráč bude první tahnout.
- Funkce následníků, která vrací seznam páru tahů a stavů, které popisují legální tahy a jejich výsledné stavy.
- Terminální test (*Terminal test*), který rozpozná, zda hra skončila. Stavy, kde hra skončila, se nazývají terminální.
- Užitková funkce (*Utility function*), která vrací číselnou hodnotu pro terminální stav. Např. v šachách je výsledek výhra, remíza, prohra s hodnotami +1, 0, -1. Některé hry mají rozsah této hodnoty větší, jako např. Backgammon, kde je rozpětí od +192 do -192.

Pro nalezení správného tahu je nutné prověřit následující budoucí tahy a zjistit tak, které tahy vedou k vítězství. K tomu se používá herní strom (*game tree*) [24].

### 4. Herní stromy

Herní strom slouží jako vizuální pomůcka, jak vyjádřit možné posloupnosti tahu z výchozího (nebo aktuálního) do terminálních stavů (v případě, že takové existují). Herní strom je vyobrazen na obrázku 29.



Obr. 29 - Herní Strom z [54].

Herní strom je acyklický orientovaný graf, jehož vrcholy představují jednotlivé stavy hry a orientované hrany představují jednotlivé tahy (vždy tah jednoho hráče). Kořen stromu představuje aktuální stav hry. Skupina uzlů, které mají stejný počet předchůdců (nebo mají stejný počet tahů) ke kořenu stromu budeme nazývat úrovní (*ply*) stromu. Stupeň uzlu (počet následníků uzlu) je definován počtem validních možných tahů v této pozici. Listy stromu budou terminální stavy. Takto popsaný herní strom se nazývá úplný herní strom (*complete game tree*). Pokud se nám podaří sestavit takový strom od výchozí pozice, znamená to, že hra byla "vyřešena". To znamená, že v takové hře je možné najít posloupnost tahů pro jednoho nebo druhého hráče, kterou když se daný hráč bude řídit, tak mu zajistí výhru nebo remízu (viz [24]). Takové hry existují a jsou to například piškvorky, Connect 4 nebo dokonce i dáma. Podrobnější informace je možné najít na [25].

Avšak u většiny her takový strom sestavit nelze a nejsou vyřešené, protože nejsou konečné nebo má tento strom příliš velký faktor větvění (vysvětlení v kapitole 7). To je případ našich her, které jsou buď příliš komplexní na vytvoření takového stromu nebo by tvorba stromu vyžadovala neúměrné množství výpočetních prostředků. Navíc hry Abalone ani Terrace nejsou konečné. U těchto her můžeme opakovat do nekonečna určitou posloupnost tahů a nezabrání nám v tom žádné pravidlo. U hry Yinsh díky inkrementálnímu přidávání kamenů a jejich odebírání pouze při vytvoření řady, vždy hra končí vítězstvím nějakého hráče nebo vyčerpáním kamenů a remízou.

## 5. Nekompletní herní strom

V případech, kdy není možno sestavit herní strom, sestavíme částečný herní strom (*partial game tree*), kde omezíme hloubku prohledávání, neboli omezíme počet tahů od aktuálního tahu (kořene stromu). Díky tomuto omezení už nenajdeme všechny terminální stavy, z čehož vyplývá, že nemůžeme rozhodnout, které stavy vedou na terminální stavy výhry, prohry či remízy. A tudíž nemůžeme ani vybrat, jaký nejlepší tah vede k vítězství. V této situaci

nahradíme užitkovou funkci heuristickou funkcí , s jejíž principem se seznámíme v další kapitole.

## 6. Heuristická funkce

Heuristická funkce<sup>2</sup> (*heuristic function*) nebo také ohodnocovací funkce, nám vrací číselnou hodnotou, která popisuje, jak daný stav je výhodný či nevýhodný pro hráče. Zpravidla rozlišuje více hodnot než užitková funkce. Na rozdíl od užitkové funkce, stanovení jejích hodnot není absolutní. To znamená, že není žádný obecný návod, jak vypočítat hodnotu této funkce. Heuristická funkce se proto stanovuje na základě pozorování hry a zkušeností. Protože funkce není přesná, tak vždy představuje určitý kompromis mezi jednotlivými vlastnostmi ovlivňující hru. Muže to být například počet kamenů, mobilita kamenů (úroveň možnosti pohybu kamenů), určité kombinace rozložení kamenů či jejich vlastnosti. Detaily o vlastnostech a samotných heuristických funkcích pro implementované hry se dozvíte v kapitole o implementaci. Je důležité zmínit, že kvalitní heuristická funkce podstatnou měrou ovlivňuje výsledek hledání nejvýhodnějších tahů hráče. Rozsah výstupních hodnot z heuristické funkce ani jejich rozložení není pevně definováno. Avšak u her dvou hráčů (hráče nazveme třeba hráč A a hráč B) se ustálila následující pravidla:

- $+\infty$  - vítězství hráče A
- *kladná hodnota* - výhodnější pozice pro hráče A
- 0 - neutrální pozice nebo remíza
- *záporná hodnota* - výhodnější pozice pro hráče B
- $-\infty$  - vítězství hráče B

## 7. Složitost her

Než se však pustíme do přímého uplatnění heuristické funkce ve vyhledávácích algoritmech (viz kapitola 8), tak napřed podle [26] probereme samotnou náročnost her (*game complexity*), jejíž parametry ovlivňují výrazně vyhledávání tahu.

Na měření náročnosti her používáme 2 hlavní ukazatele. Počet přípustných pozic (*space-state complexity*) a složitost herního stromu (*game-tree complexity*). Navíc k těmto dvou parametrům nás bude zajímat průměrný faktor větvení (*average branching factor*) a jeho problematika.

- Počet přípustných pozic je počet všech dosažitelných pozic ve hře z jejího počátečního stavu.
- Složitost herního stromu je počet možných odlišných her, které je možno hrát. V podstatě je to tedy počet terminálních uzlů (listů) úplného herního stromu z počátečního stavu hry. U her, které nejsou konečné, představuje tento ukazatel nekonečno.

---

<sup>2</sup> V některých materiálech je alternativně nazývána jako užitková funkce, avšak zde rozlišujeme mezi užitkovou a heuristickou funkcí.

- Průměrný faktor větvení je průměrný počet možných následních stavů z každého neterminálního stavu. Např. pokud je možno ve hře v pozici zahrát 3 různé tahy vedoucí do 3 různých stavů, je faktor větvení tohoto stavu 3.

Často je obtížné určit počet přípustných pozic nebo složitost herního stromu, pak zjišťujeme horní mez těchto hodnot i se započtením nedosažitelných pozic. Pokud se nám nedaří určit složitost herního stromu přímo, můžeme ji aproximativně získat umocněním průměrného faktoru větvení na průměrný počet tahů hry. Nyní přejděme ke konkrétním hodnotám těchto parametrů pro zvolené hry. U jednotlivých her budeme určovat složitost pro hru 2 hráčů v základním postavení hry.

## 7.1. Složitost hry Abalone

### 7.1.1. Počet přípustných pozic

Pomocí vzorce popsaného v [27] a [28] vychází aproximace horní meze:

$$\begin{aligned}
 & \left( \sum_{\substack{\text{kamenů hráče A} \\ a=\min.\text{možný počet kamenů hráče A}}}^{\max.\text{počet kamenů hráče A}} \sum_{\substack{\text{kamenů hráče B} \\ b=\min.\text{možný počet kamenů hráče B}}}^{\max.\text{počet kamenů hráče B}} \binom{\text{počet polí desky}}{a} \binom{\text{počet polí desky} - a}{b} \right) \\
 & * \frac{1}{\text{počet odlišných symetrií a rotací}} = \\
 & = \frac{\sum_{a=8}^{14} \sum_{b=9}^{61} \binom{61}{a} \binom{61-a}{b}}{12} \cong 1.70 \times 10^{24} \tag{1}
 \end{aligned}$$

### 7.1.2. Složitost herního stromu

Jak bylo řečeno hra Abalone není konečná. Avšak pokud bychom pro naše potřeby zakázali opakování pozic tak můžeme tuto složitost spočítat (vycházejíc z předpokladu, že lidští hráči nebudou neustále opakovat pozice, ale budou mít snahu hru ukončit). Složitost pak zde je obtížné stanovit přímo z herního stromu, ale můžeme využít výpočet na základě průměrného faktoru větvení a průměrného počtu tahů (úrovní). Podle [27] je průměrné větvení 60. Průměrný počet tahů hry je 87, který byl zjištěn pomocí statistik z online hraní na PBEM serveru [28]. Přibližná složitost herního stromu tedy je:

$$\text{faktor větvení počet tahů} = 60^{87} \cong 5.00 \times 10^{154} \tag{2}$$

## 7.2. Složitost hry Terrace

### 7.2.1. Počet přípustných pozic

Uvedený vzorec na počet přípustných pozic (viz složitost Abalone) lze použít i u Terrace. Zde počet přípustných pozic přibližně vychází:

$$\frac{\sum_{a=1}^{16} \sum_{b=1}^{16} \binom{64}{a} \binom{64-a}{b}}{2} \cong 1.81 \times 10^{27} \quad (3)$$

### 7.2.2. Složitost herního stromu

Terrace stejně jako Abalone není konečná hra, ale můžeme využít zákaz opakování, a tím počítat složitost podle podobného postupu jako u Abalone, protože i zde neznáme přesně celý strom.

$$\text{faktor větvení počet tahů} = 64.7^{79.2} \cong 2.65 \times 10^{143} \quad (4)$$

Faktor větvení a počet tahů na hru byl zjištěn testováním v implementaci.

## 7.3. Složitost hry Yinsh

### 7.3.1. Počet přípustných pozic

I zde můžeme použít vzorec použitý u hry Abalone, ale musíme ho přizpůsobit. U této hry máme v podstatě 4 druhy "prvků" hry, jde o kameny a kroužky v černé a bílé barvě. Přizpůsobený vzorec pro horní mez tedy vypadá následovně:

$$\left( \begin{array}{cccc} \sum_{\substack{\text{max.počet} \\ \text{kamenů} \\ \text{hráče } A}} & \sum_{\substack{\text{max.počet} \\ \text{kamenů} \\ \text{hráče } B}} & \sum_{\substack{\text{max.počet} \\ \text{kroužků} \\ \text{hráče } A}} & \sum_{\substack{\text{max.počet} \\ \text{kroužků} \\ \text{hráče } B}} \\ \sum_{\substack{a=\min. \\ \text{možný} \\ \text{počet} \\ \text{kamenů} \\ \text{hráče } A}} & \sum_{\substack{b=\min. \\ \text{možný} \\ \text{počet} \\ \text{kamenů} \\ \text{hráče } B}} & \sum_{\substack{c=\min. \\ \text{možný} \\ \text{počet} \\ \text{kroužků} \\ \text{hráče } A}} & \sum_{\substack{d=\min. \\ \text{možný} \\ \text{počet} \\ \text{kroužků} \\ \text{hráče } B}} \end{array} \right) \left( \begin{array}{c} \left( \begin{array}{c} \text{počet} \\ \text{polí} \\ \text{desky} \\ a \end{array} \right) \\ \left( \begin{array}{c} \text{počet} \\ \text{polí} - a \\ \text{desky} \\ b \end{array} \right) \\ \left( \begin{array}{c} \text{počet} \\ \text{polí} - a - b \\ \text{desky} \\ c \end{array} \right) \\ \left( \begin{array}{c} \text{počet} \\ \text{polí} - a - b - c \\ \text{desky} \\ d \end{array} \right) \end{array} \right) \\ * \frac{1}{\text{počet odlišných symetrií a rotací}} = \\ = \frac{\sum_{a=0}^{51} \sum_{b=0}^{51-a} \sum_{c=3}^5 \sum_{d=3}^5 \binom{85}{a} \binom{85-a}{b} \binom{85-a-b}{c} \binom{85-a-b-c}{d}}{12} \cong 3.69 \times 10^{49} \quad (5)$$

Na rozdíl od ostatních implementovaných her, je Yinsh zajímavý v tom, že nemá standardní rozestavení, nýbrž rozestavení je určené položením kroužků. Počet těchto výchozích pozic po položení kroužků je:

$$\frac{\sum_{a=5}^{85} \sum_{b=5}^{85} \binom{85}{a} \binom{85-a}{b}}{12} \cong 7.51 \times 10^{13} \quad (6)$$

### 7.3.2. Složitost herního stromu

Složitost herního stromu opět určíme nepřímo pomocí průměrného faktoru větvení a průměrného počtu tahů jako u hry Terrace a Abalone.

$$\text{faktor větvení } ^{\text{počet tahů}} = 45.5^{53.4} \cong 3.45 \times 10^{88} \quad (7)$$

Faktor větvení a počet tahů na hru byl zjištěn testováním v implementaci.

## 8. Vyhledávací algoritmy umělé inteligence

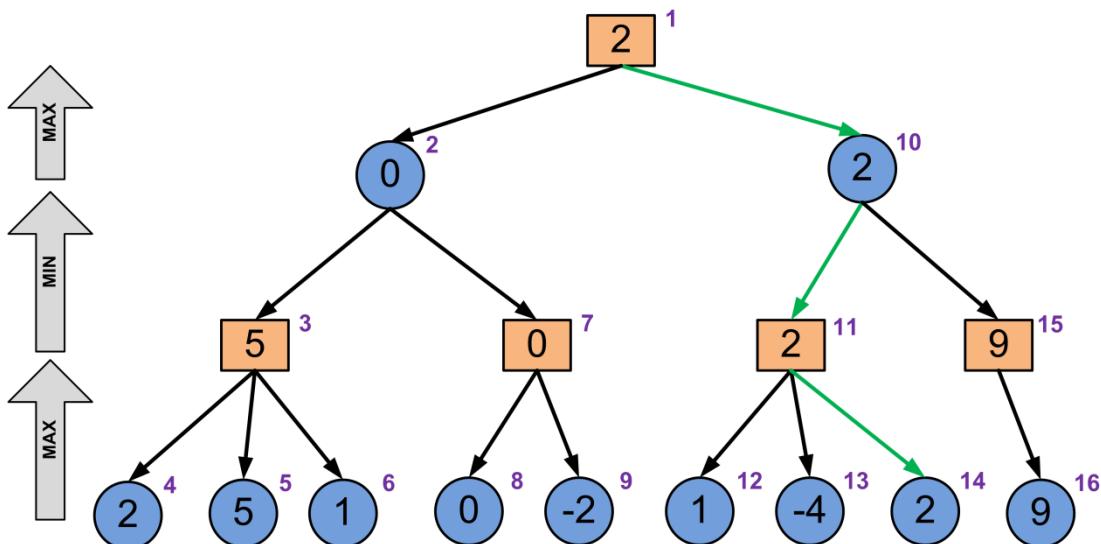
Když už víme, jak náročné jsou zvolené hry, a jak posuzovat výhodnost stavu pro určitého hráče, tak už nám jen schází způsob, jak vybereme podle těchto informací nejvýhodnější tah pro hráče, který má hrát. Tím se zabývají herní vyhledávací algoritmy (*game search algorithms*) a nejdůležitější z nich si popíšeme.

### 8.1. Algoritmus Minimax

Algoritmus minimax je určen pro 2 hráče a je založen na prohledávání herního stromu do určité hloubky. Herní strom se prohledává do hloubky (podle nákresu jakoby zleva doprava). Algoritmus se řídí podle předpokladu, že každý hráč si vždy zvolí nejlepší tah, přičemž kvalitu tahu určuje heuristická funkce. Při prohledávání se v tazích střídají dva hráči. Hráč, který je na tahu, je tzv. MAX hráč. To z důvodu toho, že tento hráč ve svých tazích bude podle předpokladu maximalizovat svůj zisk (ohodnocení heuristickou funkcí). A druhý hráč je tzv. MIN hráč. To proto, že v tazích tohoto hráče bude hráč na tahu, MAX hráč, minimalizovat svůj zisk. To opět vyplývá z předpokladu, kde MIN hráč se snažil ze svého pohledu maximalizovat svůj zisk, a tím minimalizoval zisk z pohledu MAX hráče. Tato úvaha platí pouze v hrách s nulovým součtem (viz kapitola 1), kde zisk jednoho hráče představuje ztrátu druhého. Z toho celého vyplývá, že ve hře se střídá MAX a MIN hráč - od toho pochází název algoritmu. V kontextu herního stromu to znamená, že se střídají MAX a MIN úrovně, kde z pohledu hráče na tahu (MAX hráče) se budou heuristická ohodnocení střídavě maximalizovat a minimalizovat.

### **8.1.1. Použití**

Cílem algoritmu je pochopitelně určit nejlepší tah, resp. nejlepší následující stav. Toho docílíme tak, že zjistíme ohodnocení heuristické funkce pro všechny listy (koncové stavy) herního stromu a budeme směrem ke kořenu stromu vyhodnocovat maxima či minima ohodnocení podle toho, zda jde o MAX či MIN úroveň. Takto zpětně dojdeme ke kořenu herního stromu, kde zjistíme, který tah je pro aktuálního hráče nejvýhodnější. Tento proces je zobrazen na následujícím příkladě herního stromu:



Obr. 30 - Procházení algoritmu Minimax herním stromem. Na obrázku jsou ve čtvercích stav, kde se zjišťuje maximální následný stav. A v kruzích jsou stav, kde se následný stav minimalizuje. Číslo v čtverci nebo kruhu je výsledek ohodnocovací heuristické funkce (v listech je hodnota zjišťována, v ostatních stavech se předává). Fialová čísla označují pořadí, v jakém jsou jednotlivé stav objevovány. Nejlepší tah pro hráče je vyznačen zelenými šípkami.

## **8.1.2. Pseudokód algoritmu**

Algoritmus Minimax vypadá v pseudokódu následovně:

(Následující kód není v této podobě funkční, ani nemusí odpovídat implementaci, zde je kladen důraz na přehlednost a jednoduchost.)

```

class HodnotaTah{
    int hodnota;
    Tah tah;
    HodnotaTah(int h,Tah t){
        hodnota=h;
        tah=t;
    }
}
HodnotaTah minimax(Pozice p,int hloubka){
    if (hracNaTahu()== prvníHrac())
        return maxHrac(p,hloubka);
    else return minHrac(p,hloubka);
}

HodnotaTah maxHrac(Pozice p, int hloubka) {
    if ((konecHry(p) or hloubka<=0){
        return new HodnotaTah(Ohodnot(p),null);
    }
    int max = -NEKONECNO;
    Tah maxTah = null;
    foreach tah of GenerujTahy(p){
        Pozice novaPozice= udelejTah(tah,p);
        int hodnota = minHrac(novaPozice,hloubka-1).hodnota;
        vratTah(tah, novaPozice);
        if (hodnota > max){
            max = hodnota;
            maxTah = tah;
        }
    }
    return new HodnotaTah(max,maxTah);
}
HodnotaTah minHrac(Pozice p, int hloubka) {
    if ((KonecHry(p) or hloubka<=0){
        return new HodnotaTah(Ohodnot(p),null);
    }
    int min = NEKONECNO;
    Tah minTah = null;
    foreach tah of GenerujTahy(p){
        Pozice novaPozice=udelejTah(tah,p);
        int hodnota = maxHrac(novaPozice,hloubka-1).hodnota;
        vratTah(tah,novaPozice);
        if (hodnota < min){
            min = hodnota;
            minTah = tah;
        }
    }
    return new HodnotaTah(min,minTah);
}

```

### 8.1.3. Popis práce pseudokódu

Funkce metod s neuvedeným kódem:

- GenerujTahy(Pozice p) - vrací seznam legálních tahů z této pozice. Jednotlivé tahy vedou na různé pozice
- udelejTah(Tah t,Pozice p) - vrací novou pozici, kde byl na pozici p vykonán tah t
- vratTah(Tah t,Pozice p) - vrací novou pozici, kde byl vrácen tah t z pozice p

- `Ohodot(Pozice p)` - vrací číselné ohodnocení heuristické funkce
- `konecHry(Pozice p)` - vrací logickou hodnotu, zda daná pozice představuje konec hry, resp. je terminální
- `hracNaTahu()` - vrací hráče, který má táhnout
- `prvniHrac()` - vrací hráče, který začíná hru. Např. při hře bílého a černého hráče, to bude bílý

Kód se volá např. takto:

```
HodnotaTah nejlepsiTah = minimax(aktualniPozice(), 3)
```

Toto volání prohledá algoritmem minimax herní strom do hloubky 3 (3 tahy dopředu) a vrátí výsledek.

Algoritmus na začátku volá střídavě `maxHrac(...)` a `minHrac(...)`, a tím se zanořuje do větší hloubky herního stromu a klesá hodnota hloubky. U listů stromu, kde je hloubka rovná nule, se toto zanořování zastaví díky podmínce na začátku těchto metod. Pomocí rekurze se začne výsledné ohodnocení vracet zpět, kde se hledá tah s maximálním nebo minimálním ohodnocením u metody `maxHrac(...)`, resp. `minHrac(...)`. Při rekursivním návratu ke kořenu herního stromu je důležité hlavně ohodnocení, tah se ukládá pouze proto, aby se nám po zavolání algoritmu vrátil i nejlepší tah. Návrat nejlepšího tahu nebo práce s pozicí se v literatuře popisující Minimax neuvádí, ale pro implementaci algoritmu jsou tyto operace nezbytné.

### 8.1.4. Složitost algoritmu Minimax

Algoritmus prochází všechny stavy herního stromu, tudíž s větší hloubkou herního stromu se nám zvětšuje exponenciálně počet uzlů podle faktoru větvení neboli počtu následných tahů v pozici. Složitost pak závisí na tomto faktoru větvení ( $b$ ) a hloubce ( $h$ ), do jaké budeme prohledávat. Časová složitost algoritmu tedy je:

- Nejlepší případ  $O(b^h)$
- Průměrný případ  $O(b^h)$
- Nejhorší případ  $O(b^h)$

Časová složitost je u všech případu asymptoticky stejná, protože algoritmus není datově sensitivní. Paměťová složitost algoritmu odpovídá hledání do hloubky, kde je nutné ukládat pouze stav zásobníku. Paměťová složitost je tudíž:

$$O(h)$$

## 8.2. Algoritmus Negamax

Algoritmus Negamax není plnohodnotným algoritmem, spíše jde o elegantnější variantu Minimaxu s kratším kódem. Jak z pseudokódu Minimaxu vidíme, metody `minHrac(...)` a `maxHrac(...)` jsou velice podobné a v podstatě duplikují stejný kód. Negamax spočívá v tom, že s úpravami spojíme tyto metody do jedné a tím zaručíme, že kód bude

možno lépe spravovat, a tím bude spolehlivější. Tento algoritmus nám mimo jiné umožní přechod k pokročilejším algoritmům a postupům popsaným dále v této kapitole.

Princip tohoto algoritmu spočívá v tom, že místo střídání maximalizace a minimalizace ohodnocení budeme pouze maximalizovat. Minimalizaci hodnot u MIN hráče, docílíme změnou znaménka. To znamená, že při přechodu na nižší úroveň herního stromu vždy změníme znaménko ohodnocení, "negujeme" ohodnocení. Od tohoto negování pochází název algoritmu.

Jinak je nutno připomenout, že algoritmus prochází herní strom totožným způsobem jako Minimax a vrací stejný tah jako Minimax. Proto rovnou přejdeme na pseudokód tohoto algoritmu.

### 8.2.1. Pseudokód algoritmu

Následující kód představuje pseudokód algoritmu Negamax, kde změny oproti Minimaxu jsou vyznačeny tučně.

(Následující kód není v této podobě funkční ani nemusí odpovídat implementaci, zde je kladen důraz na přehlednost a jednoduchost.)

```
class HodnotaTah{
    int hodnota;
    Tah tah;
    HodnotaTah(int h,Tah t) {
        hodnota=h;
        tah=t;
    }
}

HodnotaTah negaMax(Pozice p, int hloubka) {
    if ((konecHry(p) or hloubka<=0) {
        return new HodnotaTah(Ohodnot(p),null);
    }
    int max = -NEKONECNO;
    Tah maxTah = null;
    foreach tah of GenerujTahy(p) {
        Pozice novaPozice= udelejTah(tah,p);
        int hodnota = -negaMax(novaPozice,hloubka-1).hodnota;
        vratTah(tah, novaPozice);
        if (hodnota > max) {
            max = hodnota;
            maxTah = tah;
        }
    }
    return new HodnotaTah(max,maxTah);
}
```

### 8.2.2. Popis práce pseudokódu

Význam nepopsaných metod a způsob volání algoritmů je stejný jako u Minimaxu. Práce pseudokódu je analogická s algoritmem Minimax.

### 8.2.3. Složitost algoritmu Negamax

Vzhledem k tomu, že Negamax je pouze elegantnější úpravou Minimaxu a pracuje stejným způsobem, tak časová i paměťová složitost je stejná jako u Minimaxu.

## 8.3. Algoritmus Alfa-Beta ořezávání

Jak bylo popsáno, časová složitost algoritmu Minimax exponenciálně roste, což znamená, že jen malé zvětšení hloubky vede k velkému počtu prohledávaných stavů. To omezuje použití Minimaxu. Tento nedostatek částečně řeší algoritmus Alfa-Beta ořezávání (*Alpha-Beta pruning*), který nám umožňuje určité stavy neprohledávat - ořezat (*prune*) a snížit tak počet prohledávaných stavů. Současně nám zaručuje, že výsledný bude identický s výsledkem algoritmu Minimax. Alfa-Beta je velice podobná Minimaxu a i prohledávání herního stromu probíhá podobně.

Princip algoritmu je snaha ořezat stavy, které nevedou k řešení a současně zanechat stavy, které k řešení vedou, aniž by to ovlivnilo výsledek prohledávání. Základní myšlenkou ořezávání je takové pravidlo, že hráč na tahu neudělá tah, který se po dalším tahu protihráče ukáže jako nevýhodný (více ziskový pro oponenta). Proto všechny další tahy, které by byly vykonány po takovém nevýhodném tahu (tahy patřící do větve herního stromu s kořenem, který představuje nevýhodný tah) nebudou prohledávány a brány v potaz – budou „ořezány“.

Tyto myšlenky je možno ukázat na následujícím "výrazu" (inspirováno [23]). Tento výraz se vyhodnocuje stejným způsobem, jakoby se vyhodnocoval průchod Minimaxu herním stromem.

$$\begin{aligned}
 & \text{Ohodnocení kořene} = \\
 & = \max(\min(-1, 12, 4), \min(-3, x, y), \min(14, 5, -10)) = \\
 & = \max(-1, \min(-3, x, y), -10) = \\
 & = \max(-1, z, -10) = -1 \quad , \text{kde } z \leq -3
 \end{aligned} \tag{8}$$

Zde jsme dva stavy označili neznámou hodnotou  $x$  a  $y$ . Vidíme, že výsledek není závislý na hodnotách těchto stavů, což jsou právě takové stavy, které můžeme pomocí tohoto algoritmu ořezat.

### 8.3.1. Použití

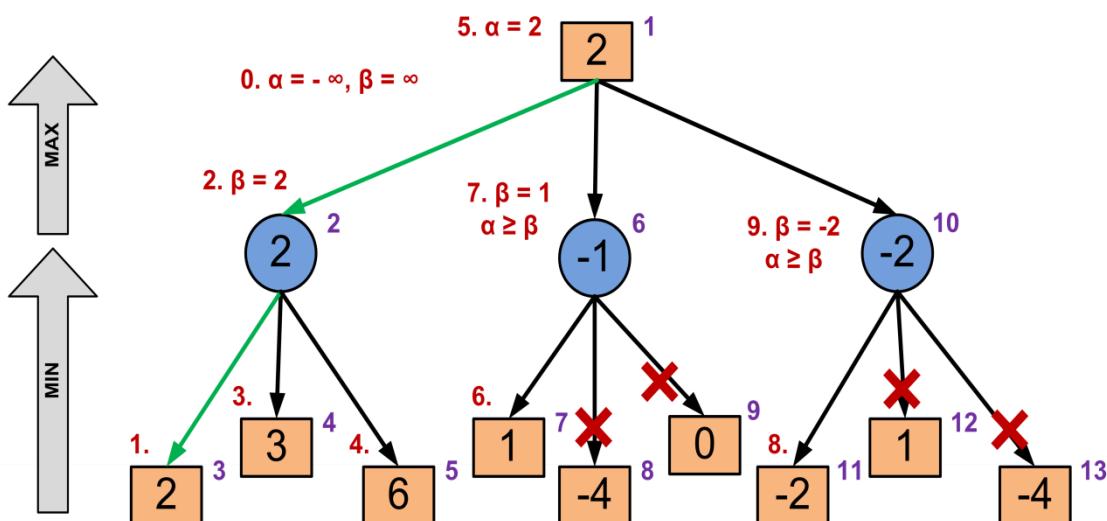
K ořezávání při průchodu herním stromem algoritmu využívá dvě hodnoty, a to  $\alpha$  a  $\beta$ . Tyto hodnoty definují tzv. vyhledávací okno (*search window*), kam patří všechny budoucí stavy s lepším ohodnocením pro daného hráče. Alfa představuje spodní hranici ohodnocení stavu, které byly dosud prohledány. Pokud tedy alfa v průběhu procházení stromem stoupá, znamená to, že hráč si vede lépe. Beta naopak značí horní hranici ohodnocení stavu, které bylo dosud

dosaženo. Pokud beta klesá, znamená to, že oponentovi se daří lépe. Pokud se tyto hodnoty dostanou do stavu  $\alpha \geq \beta$ , okno zaniká. V následujících stavech, vedoucích z takové pozice, nemůžeme najít takovou hodnotu, aby patřila do vyhledávacího okna, proto tyto stavy ořežeme (odpovídá nalezení -3 ve výrazu výše). Pokud hráč nebo oponent nalezne pozici, která zvětší alfa, tak se alfa nastaví na tuto hodnotu. Naopak pokud oponent nalezne pozici, která zmenší betu, tak se beta také nastaví na tuto hodnotu. Při každém začátku prohledávání nastavíme výchozí hodnoty na:

$$\alpha = -\infty$$

$$\beta = \infty$$

Jak vypadá průchod algoritmu alfa-beta prořezávání na herním stromu demonstруje následující obrázek 31.



Obr. 31 - Průchod algoritmu Alfa-Beta herním stromem. Jednotlivé prvky mají stejný význam jako na obrázku 30. Přidány jsou červená čísla, která značí jednotlivé kroky, při průchodu algoritmem Alfa-beta. Pokud se v daném kroku mění hodnota  $\alpha$  nebo  $\beta$ , tak je to v obrázku vyznačeno červeně. Ořezané stavy jsou červeně přeskrtnutý.

### 8.3.2. Pseudokód algoritmu

Nyní si ukážeme, jak tento algoritmus pracuje na kódu. Tento kód se mírně liší od teoretického popisu, protože byl upraven do "negamaxového" provádění<sup>3</sup>. Tato úprava eliminuje minimalizování, díky čemuž se v minimalizačních hloubkách změní znaménko. Provádění algoritmu a výsledek to však nezmění. Změny oproti algoritmu Negamax jsou vyznačeny tučně

(Následující kód není v této podobě funkční ani nemusí odpovídat implementaci, zde je kladen důraz na přehlednost a jednoduchost.)

<sup>3</sup> Algoritmus je možné vyjádřit pomocí dvou metod jako u Minimaxu, ale tento způsob implementace se v praxi nepoužívá.

```

class HodnotaTah{
    int hodnota;
    Tah tah;
    HodnotaTah(int h,Tah t) {
        hodnota=h;
        tah=t;
    }
}
HodnotaTah alfaBeta(Pozice p, int hloubka,int alfa,int beta) {
    if ((konecHry(p) or hloubka<=0){
        return new HodnotaTah(Ohodnot(p),null);
    }
    int max = -NEKONECNO;
    Tah maxTah = null;
    foreach tah of GenerujTahy(p) {
        Pozice novaPozice= udelejTah(tah,p);
        int hodnota = -alfaBeta(novaPozice,
                    hloubka-1,-beta,-alfa).hodnota; // (A)
        vratTah(tah, novaPozice);
        if (hodnota > max){
            max = hodnota;
            maxTah = tah;
        }
        if (max > alfa) alfa=max; // (B)
        if (alfa>=beta) return new HodnotaTah(alfa, maxTah); // (C)
    }
    return new HodnotaTah(max,maxTah);
}

```

### 8.3.3. Popis práce pseudokódu

Význam nepopsaných metod a způsob volání algoritmů je stejný jako u Minimaxu.

Příklad volání algoritmu:

```
HodnotaTah nejlepsiTah =
    alfaBeta(aktualniPozice(),3,-NEKONECNO, NEKONECNO)
```

Toto volání prohledá algoritmem alfa-beta herní strom do hloubky 3 (3 tahy dopředu) a vrátí výsledek.

V pseudokódu je důležité objasnit označené řádky. Na řádku A si všimněme mínsu výsledku volání `alfaBeta`, což je důsledek negamaxové úpravy, kde se mezi úrovněmi neguje ohodnocení. Toto negování, se projeví na i parametry alfa a beta, které je nutno rovněž znegovat abyhom zachovali velikost okna do další úrovně herního stromu. Na řádku B se přenastavuje hodnota alfy při nalezení lepšího tahu. Ač to není zřejmé, tento řádek přenastavuje i hodnotu beta v minimalizujících úrovních, a to tak, že volání na řádku A zamění alfu za betu. Řádek C je důležitý, protože je zde podmínka, jejíž splnění způsobí ořezání zbylých stavů v cyklu `foreach`.

### 8.3.4. Složitost algoritmu Alfa-beta ořezávání

Zjišťování časové složitosti vychází z popisu u Minimaxu (viz podkapitola 8.1.4). Avšak u tohoto algoritmu dochází k úspoře díky ořezaným stavům. Efektivitu algoritmu nejvíce ovlivňuje pořadí samotných prohledávaných stavů.

- Pokud bychom měli stavy seřazeny tak, že nejlepší tah z pohledu každého hráče se vybíral první, tak dojde k největšímu ořezu. Konkrétně by se na maximalizačních úrovních procházelo b stavů, kde b je faktor větvení. Na minimalizačních by se prošel pouze 1 stav. Nejlepší případ pak má složitost  $O(b^{\frac{h}{2}})$ , kde h je hloubka prohledávání.
- Nejhorší případ nastane, když bychom seřazení z nejlepšího případu řadili pozpátku. Nedojde potom k žádným ořezům a algoritmus degraduje na Minimax se složitostí  $O(b^h)$ .
- Průměrná složitost by odpovídala náhodnému seřazení a složitost je pak  $O(b^{\frac{3h}{4}})$ .

Díky této časové úspoře může Alfa-beta prohledat herní strom do dvojnásobné hloubky oproti Minimaxu. Neboli pokud máme u Minimaxu faktor větvení  $b_{minimax}$ , tak u Alfa-bety se nám tento faktor zredukuje na  $b_{alfabeta} \sim \sqrt{b}_{minimax}$ .

Paměťová složitost algoritmu Alfa-Beta je stejná jako u Minimaxu.

## 9. Zlepšující techniky vyhledávacích algoritmů

Popsané algoritmy tvoří základní způsob, jak najít nejvýhodnější tah pro určitého hráče. Mimo tyto algoritmy, existuje mnoho technik, které různými způsoby vylepšují efektivitu těchto algoritmů. Některé z těchto technik si popíšeme v následující kapitole.

### 9.1. Řazení tahů (*Move ordering*)

U Alfa-beta prořezání v podkapitole o složitosti, bylo zmiňováno, jak moc tento algoritmus závisí na pořadí prohledávaných tahů. Tato technika se přímo zaměřuje na tento aspekt seřazením tahů.

Řazení tahů se uplatňuje pouze u Alfa-bety, kde má vliv na efektivitu algoritmu, na rozdíl od algoritmu Negamax či Minimax. Začlenění této techniky do pseudokódu by vypadalo takto:

```
foreach tah of GenerujTahy(p) {  
    se nahradí za  
  
    foreach tah of SeradTahy(GenerujTahy(p)) {
```

Kde metoda "SeradTahy" seřadí tahy. Zde narázíme na problém, jak tahy seřadit. Vzhledem k tomu, že nevíme optimální pořadí tahů (pozn. pokud bychom to věděli, tak bychom se pak nemuseli zabývat prohledáváním herního stromu), tak jejich pořadí vyhodnotíme podle funkce, kterou nazveme jako heuristickou řadící funkci. Tuto funkci je nutné stejně jako ohodnocovací funkci určit na základě znalostí hry a její vlastností, je tedy individuální pro danou hru. Současně také musí být dostatečně rychlá, protože se bude vyhodnocovat v každém vnitřním stavu herního stromu. Konkrétní implementaci řadící funkce si popíšeme v implementační části v podkapitole 18.3.

## 9.2. Iterativní prohlubování (*Iterative Deepening*)

Prohledávání vždy je časově náročné. Iterativní prohledávání nám právě pomůže lépe kontrolovat potřebný čas na prohledávání. Často totiž potřebujeme omezit čas prohledávání, v čemž nám tato technika pomůže. Technika je založená na jednoduchém principu, že ve stanoveném časovém limitu prohledáváme do hloubky 1,2,3... Když nám dojde čas, ukončíme hledání a použijeme výsledek posledního prohledávání, které se stihlo dokončit. Pokud nastane nepravděpodobný případ, že v časovém limitu nestihneme prohledat do základní hloubky 1, tak prohledáme do této hloubky znova bez časového limitu. To je proto, abychom získali alespoň nějaký výsledek z prohledávání. Tuto techniku lze využít na algoritmu Minimax či Negamax, tak také u alfa-beta prořezávání.

### 9.2.1. Pseudokód techniky

Pseudokód pro tuto techniku na algoritmus Alfa-Beta vypadá následovně:

```
HodnotaTah prohledejIterativne(Pozice p, int casLimit) {
    HodnotaTah vysledek=null;
    Uloha(casLimit) {
        for (hloubka=1;;hloubka++) {
            vysledek=alfaBeta(p,hloubka,-NEKONECNO,NEKONECNO);
        }
    }catch(casVyprsel) {
        if (vysledek==null)
            vysledek= alfaBeta(p,1,-NEKONECNO,NEKONECNO);
    }
    return vysledek;
}
```

### 9.2.2. Popis práce pseudokódu

Funkce konstruktů neuvedeným kódem:

- `Uloha(int caslimit)`, `catch(int casLimit)` - Fiktivní konstrukt, který měří, zda neuplynul daný čas. Pokud ano, přeruší provádění a přeskočí na kód v `catch{}`. V reálné implementaci se toto řeší složitěji s vlákny a synchronizací.

Příklad volání:

```
HodnotaTah nejlepsiTah = prohledejIterativne(aktualniPozice(),4000)
```

Tímto voláním zavoláme iterativní prohledávání s časovým limitem 4s (parametr je uveden v milisekundách).

### 9.2.3. Navýšení počtu stavů

Můžeme si všimnout, že s použitím této techniky prohledávání do stejné hloubky vyžaduje prohledat celkově více stavů, a tím i trvá déle, protože musíme prohledávat i do nižších hloubek. Pokud bychom měli konstantní faktor větvení  $b$ , tak prohledáme navíc následující počet stavů, kde  $h$  je hloubka a  $x$  je maximální hloubka.

$$\text{navýšení počtu stavů} = 100 \times \lim_{x \rightarrow \infty} \left( \sum_{h=1}^x \frac{1}{b^h} \right) \cong \frac{100}{b-1} \% \quad (9)$$

I přes tento fakt tato metoda ve výsledku přinese zlepšení a to hlavně v kombinaci s dalšími technikami jako transpozičními tabulkami, které budou popsány v další podkapitole. Zde pravě využijeme tento iterativní průchod herního stromu pomocí této techniky.

## 9.3. Transpoziční tabulka (*Transposition Table*)

Při prohledávání herního stromu nemusí být všechny jeho stavů vzájemně odlišné. Dochází k tzv. transpozicím, což jsou stejné stavů hry na více místech v herním stromu. Základní myšlenka této techniky je uložení výsledku prohledávání pozic, které jsme prošli a v případě, že na ně narazíme jinde znovu, už je nemusíme znova prohledávat.

Ukládat všechny pozice není možné, protože počet stavů v herním stromu roste exponenciálně s hloubkou a už při malých hloubkách bychom vyčerpali operační paměť počítače. Tento problém se řeší hashovací tabulkou, která pro každý stav vyrobí tzv. hash kód (více viz kapitola 9.3.4), podle kterého určí (mapuje) index v tabulce, kam se stav uloží. Vzhledem k tomu, že tabulka bývá menší než počet všech možných stavů, může se stát, že více stavů je mapováno na stejný index, kde je vždy uložena nejnovější prohledávaná pozice. Transpoziční tabulka nám urychluje vyhledávání několika způsoby. Jak už bylo zmíněno, umožňuje nám neprohledávat znova prohledané stavů. U algoritmu Alfa-beta pomocí této tabulky můžeme přesněji nastavit hodnoty alfa a beta, čímž můžeme zefektivnit ořezávání. Rovněž také můžeme zefektivnit řazení tahů, pokud podle tabulky víme, které tahy byly kvalitní.

### 9.3.1. Pseudokód techniky

V následujícím pseudokódu upraveného algoritmu Alfa-beta podle [30], si popíšeme detailněji všechna tato zlepšení. Změny oproti Alfa-beta prořezávání jsou vyznačeny tučně.

```
class HodnotaTah{
    int hodnota;
    Tah tah;
    HodnotaTah(int h,Tah t) {
        hodnota=h;
        tah=t;
    }
}
```

```

    }
Tah[] seradTahyTT(Tah[] tahy,ZaznamPozice zaznam) {
    Tah[] serazeneTahy=seradTahy(tahy)
    if (zaznam!=null){
        foreach tah of serazeneTahy{
            if (tah==zaznam.nejlepsiTah){
                serazeneTahy=vloz(serazeneTahy, 0,tah);
                break;
            }
        }
    }
    return serazeneTahy;
}
HodnotaTah alfaBetaTT(Pozice p, int hloubka,int alfa,int beta) {
    if ((konecHry(p) or hloubka<=0){ // (A)
        return new HodnotaTah(Ohodnot(p),null);
    }
    int max = -NEKONECNO;
    Tah maxTah = null;
    int zaznamHodnotaUlozeni=alfa;
    ZaznamPoziceTyp zaznamTyp=ALFA;
    ZaznamPozice zaznam=transpozicniTab.nacti(p)
    if (zaznam!=null and zaznam.hloubka>=hloubka){
        if (zaznam.typ==PRESNE){
            return new HodnotaTah(
                zaznam.hodnota,zaznam.nejlepsiTah); // (B)
        }else if (zaznam.typ==ALFA
            and zaznam.hodnota>alfa and zaznam.hodnota <beta){
            alfa=zaznam.hodnota;
            zaznamTyp=PRESNE;
            zaznamHodnotaUlozeni=alfa;
        }else if (zaznam.typ==BETA
            and zaznam.hodnota>alfa and zaznam.hodnota<beta){
            beta=zaznam.hodnota;
        }
        foreach tah of SeradTahyTT(GenerujTahy(p) ,zaznam) {
            Pozice novaPozice= udelejTah(tah,p);
            int hodnota = -alfaBetaTT(novaPozice,
                hloubka-1,-beta,-alfa).hodnota; // (A)
            vratTah(tah, novaPozice);
            if (hodnota > max){
                max = hodnota;
                maxTah = tah;
            }
            if (max > alfa){
                alfa=max; // (B)
                zaznamTyp=PRESNE;
                zaznamHodnotaUlozeni=max;
            }
            if (alfa>=beta){
                transpozicniTab.uloz(p,hloubka,BETA ,beta,maxTah);
                return new HodnotaTah(alfa, maxTah); // (C)
            }
        }
        transpozicniTab.uloz(p,hloubka,zaznamTyp,
            zaznamHodnotaUlozeni,maxTah);
    }
    return new HodnotaTah(max,maxTah);
}

```

### 9.3.2. Popis práce pseudokódu

Funkce metod s neuvedeným kódem, které dosud nebyly popsány:

- Tah[] vloz(Tah[] tahy, int index, Tah tah) – Metoda vloží daný tah do pole na daný index a prvky od indexu v poli posune. Vrací výsledné pole.
- ZaznamPozice transpozicniTab.nacti(Pozice p) - Metoda vrací uloženou pozici s transpoziční tabulkou. Pokud pozice v tabulce není, vrací null.
- transpozicniTab.uloz(Pozice p, int hloubka, ZaznamPoziceTyp typ, int hodnotaUlozeni, Tah nejlepsiTah) – Metoda uloží pozici s parametry do transpoziční tabulky.

Příklad volání pro hledání do hloubky 4:

```
HodnotaTah nejlepsiTah = alfaBetaTT(aktualniPozice(),
                                         4, -NEKONECNO, NEKONECNO)
```

Při každé pozici, kterou projdeme v herním stromu, uložíme do transpoziční tabulky záznam o dané pozici (metoda `uloz(...)`). Ukládání neprovádíme u listů herního stromu (řádek A), protože u takových pozic při opětovném nalezení nedocílíme úspory procházených stavů (listy nemají následníky). Maximálně bychom dosáhli předpočítaného ohodnocení od heuristické funkce. Ta však musí být rychlá, a proto předpočítání ohodnocení převažuje výhoda použití menší tabulky při nepřítomnosti těchto stavů.

Při každém ukládání je důležité uložit ohodnocení z heuristické funkce. V případě, že bychom pracovali s Negamaxem nebo Minimaxem, tak u těchto algoritmů je aktuální ohodnocení vždy "přesné". Narozdíl o toho algoritmus Alfa-beta při ořezávání vrací hraniční hodnoty vyhledávání alfa a beta, z nichž naopak právě nevíme přesné původní ohodnocení. Proto při každém ukládání se specifikuje `ZaznamPoziceTyp` parametr, který nám říká povahu ukládané hodnoty. Tzn. `PRESNE` u "přesné" hodnoty a `ALFA` nebo `BETA` pro "nepřesné" mezní hodnoty. V každém stavu prohledáváme tabulkou, zda neobsahuje už procházený záznam pomocí metody `nacti(...)`. Pokud se nám vrátí platný stav, tak musíme ověřit, zda uložená hloubka záznamu je alespoň stejná jako aktuální. Nemůžeme použít stavy, kde tato hloubka je menší, protože nám nevrátí dostatečně přesné výsledky. Nevidíme totiž, co se stane v následujících tazích, které by jinak byly prohledány. Může se pak stát, že tah s dobrým ohodnocením by se v následujícím tahu ukázal jako špatný, aniž bychom to věděli. Pokud tedy máme platný záznam s odpovídající hloubkou, rozeznáme typ záznamu. Přesné záznamy můžeme okamžitě vrátit jako výsledek (řádku B). U typu `ALFA` a `BETA` díky neznámé přesné hodnotě ohodnocení toto udělat nemůžeme. Ale můžeme zpřesnit meze vyhledávací okna ( $\alpha, \beta$ ) pomocí uložených záznamů, pokud mají takovou hodnotu, která do okna patří.

Jak bylo uvedeno, transpoziční tabulka nám zlepší i řazení tahů. V pseudokódu je nahrazeno řazení tahů za metodu `seradTahyTT(...)`, která řadí tahy podobně jako původní algoritmus Alfa-beta s řazením. Navíc ale vyhledává uložený nejlepší tah (resp. tah vedoucí na stav s nejlepším ohodnocením) záznamu z tabulky, který v případě nalezení vloží jako první do pořadí. To znamená, že pokud byl v minulosti uložen stav  $S$ , kde byl nejlepší tah  $T$ , je velmi pravděpodobné, že při nalezení stavu  $S$  na jiném místě herního stromu, bude opět tah  $T$ .

nejlepší. Proto zkusíme tento tah  $T$  jako první, což může výrazně zefektivnit ořezávání a zmenšit celkový počet prohledaných uzlů, pokud půjde o nejlepší tah.

Samotná transpoziční tabulka (`transpozicniTab`) je sdílená hledáním v rámci celé hry mezi všemi hráči. Což znamená, že můžeme využívat i uložených výsledků z předchozích tahů a hledání. To má velký vliv na účinnost v kombinaci s iterativním prohlubováním, kde z jedné pozice se prohledává několikrát s různou limitní hloubkou. Zde především dochází k úspoře času, díky ukládání uložených nejlepších tahů při předchozích vyhledáváních na nižších hloubkách (hodnotu uzlů přímo vrátit většinou nelze, protože máme uložené uzly nižší hloubky, které neposkytují dostatečnou přesnost, viz předchozí odstavec). Od transpoziční tabulky se celkově požaduje rychlosť jejich operací, protože téměř v každém stavu se jí dotazujeme na výsledky nebo je ukládáme.

Transpozice na vizualizaci herního stromu se projevují tak, že stejné stavy se spojují, a tím se ze stromu stává graf (ve smyslu teorie grafů). Obecný vliv použití transpozičních tabulek na časovou složitost nelze vyjádřit, protože složitost je zde závislá na konkrétní hře, heuristické funkci a na předešlých hodnotách uložených v tabulce samotné. Nyní se podíváme na detaily ukládání a vybíraní prvků z tabulky.

### 9.3.3. Práce s transpoziční tabulkou

Nyní se zaměříme na to, co budeme do transpoziční tabulky ukládat, a jak budeme vyhledávat daný záznam. Následující položky tvoří nutný základ, který je potřeba ukládat:

```
class Zaznam{
    long hash;
    int hloubka;
    ZaznamTyp typ;
    int hodnota;
    Tah nejlepsiTah
}
```

Význam většiny parametrů známe z předchozí kapitoly, s výjimkou položky `hash`. Do tabulky se neukládá `Pozice`, ta je používána jen pro hledání indexu v tabulce. Jak bylo uvedeno, pro zjištění indexu v tabulce potřebujeme vygenerovat `hash` z dané vstupní `Pozice`. Hash generujeme podle Zobrist hashe, který je popsán v následující podkapitole 9.3.4. Tento nagenerovaný hash má délku 64 bitů. Avšak pro indexování v tabulce ho celý použít nemůžeme, protože 64 bitová délka přestavuje  $2^{64} = 1.84 \times 10^{19}$  hodnot, resp. uložených záznamů, což přesahuje paměťové možnosti jakéhokoliv počítače. Proto se používají na index pouze některé bity hashe. To se provádí přes operaci modulo s operandem, který je mocnina dvou. Protože nedokážeme z indexu zjistit celý hash, ukládáme ho právě do položky `hash` (celý jelikož datový typ `long` je 64 bitový). S tímto musíme počítat, když hledáme záznam v tabulce. I když takový záznam nalezneme, tak platný je pouze pokud se shodují vzájemně i 64 bitové hashe hledané a uložené pozice. V opačném případě je zde uložená jiná pozice.

### 9.3.4. Zobrist Hash

Samotné generování číselné hodnoty `hash` se provádí pomocí Zobrist hashování. Tato metoda se nazývá po jejím vynálezci Albertu Zobristovi, který ji vynalezl už v roce 1969.

Hashování se snaží, aby generované hodnoty byly co nejvíce různé i pro malé rozdíly mezi pozicemi.

Postup hashování si vysvětlíme na jednodušším příkladu a to u hry Abalone<sup>4</sup>. Zde máme na jednotlivých polích pouze jeden druh kamenů a ty se liší pouze podle hráče, kterému patří. Proto můžeme celou pozici popsat pomocí čísla hráče kamenu (a navíc jeden stav pro prázdné pole) na každé ze souřadnic osy x a y. V Zobrist hashování pro každou takovou možnost určíme náhodné číslo. To znamená, že bude mít trojrozměrné pole naplněné různými náhodnými hodnotami:

```
zobristcisla [xSouradnice] [ySouradnice] [hracCislo]
```

Tyto náhodné hodnoty zůstávají stejné po celou dobu existence transpoziční tabulky. Z těchto hodnot určíme hash pro pozici tak, že pro každé pole získáme hodnotu a vzájemně mezi hodnotami uplatníme operaci XOR. V pseudokódu bychom to zapsali následovně:

```
long generujHash() {
    long hash=0;
    for (int x = 0; x < maxX; x++) {
        for (int y = 0; y < maxY; y++) {
            hash ^= zobristcisla[x][y][hracNaPozici(x,y)]
        }
    }
    return hash;
}
```

Popis jednotlivých příkazů je následující:

- `maxX` - maximální pozice na ose X
- `maxY` - maximální pozice na ose Y
- `hracNaPozici (x, y)` - funkce, která vrátí číslo hráče na dané pozici nebo vrátí 0, pokud tam není žádný hráč

Pokud bychom měli více vlastností, tak hash generujeme obdobně, jen pole `zobristcisla` by mělo více dimenzi. Při celém generování pracujeme s 64 bitovými čísly `long`, abychom minimalizovali hashovací kolize. To je případ, kdy dva různé stavy mají stejný hash. Pokud nastane kolize, tak se to může projevit v chybách při vyhledávání. Avšak s takto dlouhým klíčem je pravděpodobnost kolize velmi nepravděpodobná.

### 9.3.5. Inkrementální hashování

U hashování použití operace XOR ( v zápisu pseudokódu to je znak „`^`“ ) má výhodu v tom, že můžeme hashovat inkrementálně. Výhoda operace XOR je taková, že pokud použijeme XOR dvakrát, tak máme opět původní hodnotu, tj.  $a \text{ XOR } b \text{ XOR } b = a$ . Inkrementální hashování znamená, že pokud bychom v pozici změnili jen hráče na jednom poli tak, že stačí na výsledný hash použít XOR původního stavu pole, čímž pozici jakoby "vymažeme". A poté ještě jednou použít XOR s novým stavem pole. A tak máme rychleji nový hash, který bude stejný, jako kdybychom zpracovali všechna pole (jako v metodě `generujHash (...)` z kapitoly 9.3.4).

---

<sup>4</sup> Pro demonstraci tento příklad je zjednodušen a plně neodpovídá implementaci.

## 10. Problematické aspekty vyhledávání

### 10.1. Nestabilita hledání (*Search Instability*)

Pokud začleníme do vyhledávání transpoziční tabulky, vyhledávání se nám stane nestabilní [30]. To znamená, že jednotlivá hledání nejsou opakovatelná. Není jistota, že se při opakování vyhledávání vrátí stejný tah, nebo že průchod herním stromem bude stejný. Tento jev způsobuje transpoziční tabulka tím, že si zachovává svůj stav v rámci celé hry. To znamená např., že při opakování stejného hledání už situace není stejná, protože už máme uložené výsledky z minulého hledání, které ovlivňují průchod herního stromu, ohodnocení i hodnoty alfa i beta. Tudíž není předvídatelné, zda výsledek bude stejný nebo ne. S velmi malou pravděpodobností mohou výsledek hledání ovlivnit také chyby v získávání stavů z tabulky způsobené hashovacími kolizemi. I přes tuto nevýhodu se techniky jako transpoziční tabulky a jiné vyplatí implementovat, protože jejich přínos v rychlosti převažuje nevýhody nestability hledání. Každopádně je nutné mít tento tyto důsledky při implementaci na paměti, aby se předešlo chybám ve vyhledávání.

### 10.2. Horizon efekt (*Horizon Effect*)

Často při hraní her uvažujeme o vzájemných výměnách (vyhozeních) kamenů. Toto je velice znatelné např. u dámy. Jeden hráč vyhodí soupeři kámen. Z této situace pak vyplývá ztráta vlastního kamenů. Takže ač v počátku tato pozice byla zisková, ve výsledku je neutrální. Pokud vyhledáváme do omezené konstantní hloubky (což využívají všechny herní vyhledávací algoritmy), tak je možné, že narazí hráč na zisk soupeřova kamene, ale už nevidí svojí ztrátu v dalším tahu, jelikož už tento tah přesahuje stanovenou hloubku. Tento jev je nazýván efekt horizontu (hledání) [31]. Tuto nevýhodu lze částečně eliminovat zavedením proměnlivé hloubky prohledávání, která se bude řídit podle toho, jak je daný tah "akční".

## Část III - Návrh a realizace

V této části se budeme věnovat tomu, jak byla napsána samotná aplikace, a jak využila poznatky z teoretické části. Avšak samotný program a jeho kompletní vysvětlení přesahuje rozsah této práce, proto se budeme zabývat jen jeho některými důležitými částmi. Hlavně se zaměříme na popis tříd a jejich propojení, heuristické funkce a na implementaci samotných algoritmů.

### 11. Použité technologie

Celá aplikace je napsána v programovacím jazyku *Java J2SE* [34] využívající aktuální verzi 1.6. Grafické rozhraní využívá grafického frameworku *Swing*, který je standardní součástí Javy. Kromě standardních knihoven aplikace používá i knihovny 3. stran. Jsou to:

- *TableLayout, TableLayoutExtension* – Knihovny poskytující podporu pro grafické tabulkové rozvržení prvků (*layout component*). Více na [35].
- *Apache Commons* - Balíček s nejběžnějšími užitečnými podpůrnými metodami. Více na [36].

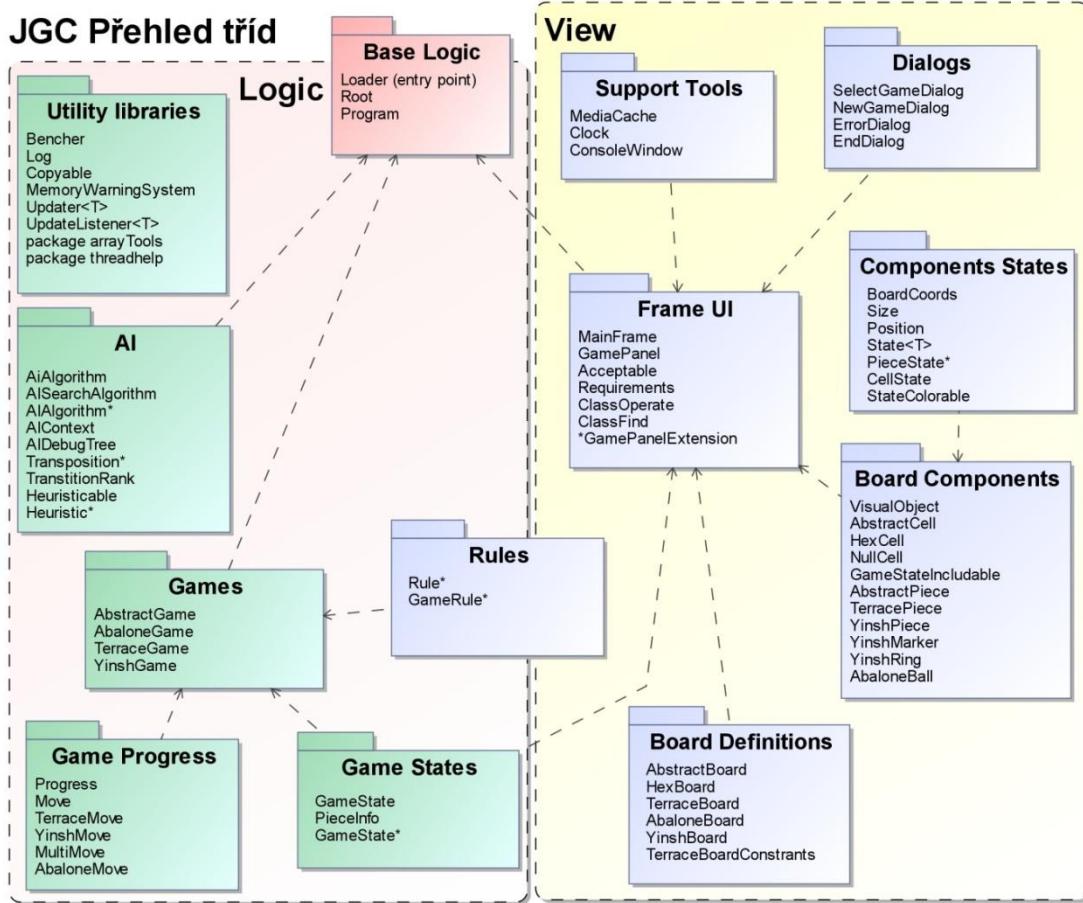
K samotnému vývoji bylo využito IDE Eclipse 3.5 Galileo s použitím většího množství různých pluginů. V aplikaci jsou dále použity vlastní samostatně vyvýjené knihovny *threadhelp* a *arrayTools*.

### 12. Objektový model

Aplikace se skládá z mnoha tříd, které mají různé závislosti mezi sebou a obsluhují různé funkce. Jelikož jde o aplikaci psanou v Javě, byl při návrhu struktury tříd kladen důraz na to, aby aplikace byla co nejvíce založena na principech objektového programování. Rovněž také byla snaha o použití vhodných návrhových vzorů (*design patterns*) na místech, kde je to vhodné.

#### 12.1. Přehled tříd aplikace

Přehled jednotlivých celků aplikace s jejich závislostmi nám popisuje následující UML diagram.



Obr. 32 - Přehled tříd.

Diagram zobrazuje skupiny tříd v jednotlivých blocích. V každém bloku je napsáno, jaké třídy sem patří. Znak "\*" zastupuje libovolnou skupinu znaků. Pokud blok obsahuje balíček tříd, je to označeno jako "package". Pokud blok obsahuje nějakou třídu, znamená to, že obsahuje i její vnitřní třídy, ač nejsou vypsány. Celá aplikace je tvořena 2 hlavními částmi. První část "Logic" je logika aplikace, kde je všechnen výkonný kód, který používá jádro aplikace. Druhá část "View", je část starající se o grafické rozhraní a úkoly s tím spojené. Barvy jednotlivých celků odpovídají jednotlivým částem. Zelené jsou pro logickou část, modré pro grafické rozhraní a červená barva značí výchozí blok aplikace. Je nutno upozornit, že seznam tříd zde uvedený, není kompletní. Chybí zde několik málo využívaných tříd, které nejsou důležité pro pochopení modelu aplikace.

Tento přehled diagramu je zjednodušenou verzí kompletního diagramu tříd, který se nachází mezi fyzickými přílohami. V kompletním diagramu jsou znázorněny důležité vlastnosti a vzájemné vztahy mezi jednotlivými třídami. Rovněž jsou zde krátké popisy funkcí těchto tříd. Avšak nenajdeme tam jednotlivé popisy metod a to proto, abychom zachovali přehlednost, jelikož celá aplikace obsahuje přes 1000 metod, aniž bychom do toho započítávali použité knihovny.

## 12.2. Popis tříd a jejich funkcí

Nyní si ve stručnosti popíšeme přehled tříd na obrázku 32 s přihlédnutím ke kompletnímu diagramu v příloze.

### 12.2.1. Třídy části *Logic*

Základ aplikace tvoří skupina *Base Logic*, kde v třídě *Loader* je vstupní bod aplikace (metoda *main*). Zbylé třídy je možné získat globálně a poskytují tak spojení mezi částí *Logic* a *View*.

Nejdůležitější část části *Logic* tvoří skupina *Games*, kde se nachází výkonný kód pro hry. Jde hlavně o kód zabývající se problematikou hráčů, pravidel her a dalších individuálních vlastností her. O obecnou funkcionality pro všechny hry se stará nadřazená třída *AbstractGame*. Třída nám poskytuje prostor pro přidání nové hry, která zdědíme vlastnosti *AbstractGame* a dodefinujeme specifické vlastnosti pro danou hru.

Ke skupině *Games* má nejbližší vztah skupina *Game Progress*. Tato skupina se zabývá průběhem hry, resp. tím, jak se hráči mají střídat ve hře. Řeší rovněž také vracení/znovu vykonávání tahů a také kupř. ukládá informace o tazích, které byly provedeny. Tyto tahy se ukládají jako třída *Move*, která je obecná. Jednotlivé třídy poté mají vlastní implementace podle toho, co potřebují ukládat.

Skupina tříd *GameStates* se věnuje ukládání stavů/pozic jednotlivých her. Pro uložení stavů jsou zde 2 obecné třídy *GameStateMutable* a *GameStateImmutable*, které se liší podle způsobu, jak chceme se stavem pracovat. Třída *PieceInfo* je prostředník mezi *GameState* a grafickou částí.

Největší je skupina *AI* zabývající se umělou inteligencí. Zde je použit návrhový vzor *strategy*, který nám umožňuje zvolit si libovolnou kombinaci algoritmu a heuristiky pro naši hru. Jednotlivé algoritmy rozšiřují standardní rozhraní *AIAlgorithm*. U heuristik je to podobné s rozhraním *Heuristicable*. Dále se v této skupině nachází třídy, které se zabývají implementací transpozičních tabulek.

Další skupina, skupina *Rule* se nachází na pomezí mezi *Logic* a *View*. *Rule* ukládá možnosti, které se zobrazí při výběru nové hry nebo v menu aplikace. Poskytuje jejich vizuální podobu, proto patří do *View*. Ale rovněž také obstarává akce při zvolení určitých možností, které zasahují do části *Logic*, proto patří i sem.

### 12.2.2. Třídy části *View*

V části *View* hlavní skupinu tvoří *Frame UI* skupina. Zde najedeme základní komponentu *MainFrame*, představující okno aplikace. Dále se zde nachází komponenta *GamePanel* a *GamePanelExtension*, starající se o vykreslování desky a jednotlivých hracích prvků. Tyto třídy také zpracovávají veškerou interakci s uživatelem. Do hlavní skupiny patří také pluginovací systém, tj. třídy *ClassOperate* a *ClassFind*, které dokáží vyhledat vhodné třídy v aplikaci, aniž by byly definovány přímo. Třída *GamePanel* hojně využívá definice desek,

potomky *AbstractBoard* ze skupiny *Board Definitions*. Tyto třídy se zabývají definováním startovních konfigurací a umožněním je převést na třídy ze skupiny *Board Components*.

Tato skupina se stará o vykreslování, stav a reakce herních komponent - polí (od třídy *AbstractCell*) a kamenů (od třídy *AbstractPiece*). Společným předkem těchto hracích komponent je třída *VisualObject*, která využívá ještě další informace ze skupiny *Components States*.

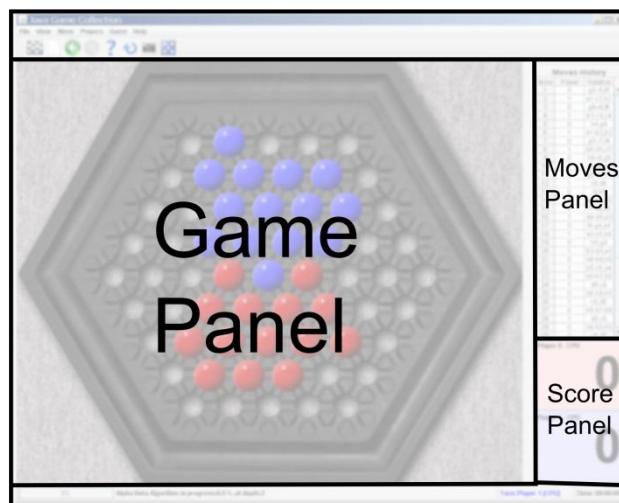
Tyto třídy uchovávají důležité informace o jednotlivých "vizuálních objektech", jako je pozice na obrazovce (*Position*), pozice na desce (*BoardCoords*), velikost (*Size*) nebo stav tohoto pole (*State<T>* - tím je myšleno např. jakému hráči objekt patří apod.).

V části *View* také najedeme třídy vykreslující různé dialogy. Tyto dialogy jsou ve skupině *DIALOGS*. Mimo jiné také ve skupině *Support Tools* najdeme různé podpůrné komponenty. Nejdůležitější z nich představuje globální třída *MediaCache*, která globálně cachuje obrázky, které se mají vykreslit, a tím pomáhá k větší výkonnosti grafického rozhraní.

## 13. Grafická reprezentace herní desky

### 13.1. Hrací panel

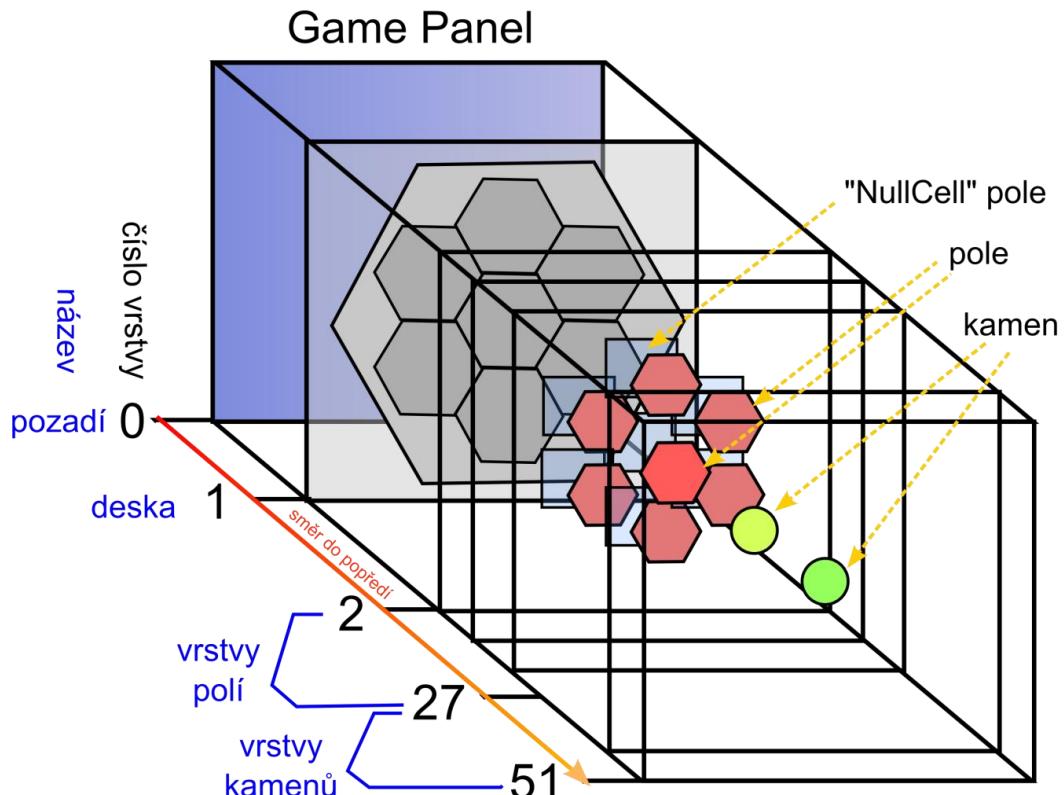
Aplikace obsahuje celkem komplexní systém, jak se zobrazují desky nebo jednotlivé prvky na desce. O vykreslování hrací desky se stará vizuální komponenta *GamePanel* její poloha je vyobrazena na obrázku 33.



Obr. 33 - Poloha komponenty GamePanel v aplikaci.

### 13.1.1. Model vrstev hracího panelu

Vizuální komponenta se skládá z (grafických) vrstev, které leží na sobě (podobně jako u pokročilejších grafických editorů). Pokud je nějaká část vrstvy prázdná, je průhledná, což nám umožní vidět vrstvy pod ní. Rozložení jednotlivých vrstev komponenty demonstruje obrázek 34.

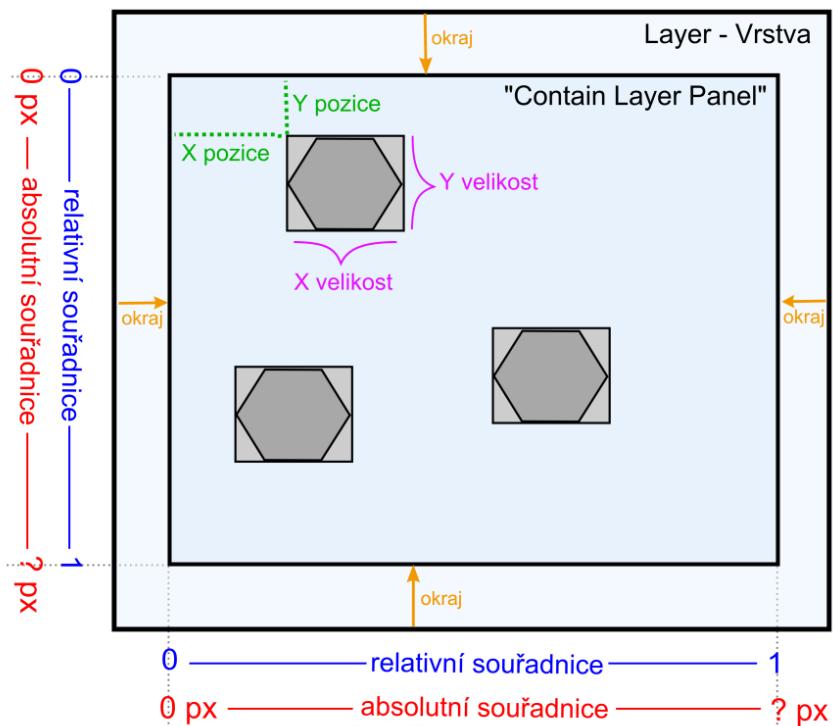


Obr. 34 - Rozložení vrstev komponenty GamePanel.

Ve vrstvě 0 je výchozí pozadí při startu. Nad tímto pozadím je obrázek desky. Poté následuje vrstva polí, tyto pole mohou být buď neviditelná (*NullCell*) nebo viditelná (např. *HexCell*). Neviditelná pole mají význam u her, kde není potřeba pole vykreslovat (protože jsou např. obsažena v obrázku desky). Slouží pro popis grafických souřadnic, kam se budou umisťovat nové hrací kameny (resp. kameny se umisťují podle těchto souřadnic). Neviditelná pole se umisťují do nejnižší vrstvy polí, u viditelných polí není umístění omezeno. V popředí je vrstva, kam se umisťují jednotlivé kameny. Vrstev pro pole a kameny je více, aby bylo možné je různě překrývat, pokud to hra vyžaduje.

### 13.1.2. Vrstva hracího panelu

Libovolná vrstva pole nebo kamenu, se skládá z více podprvků. Souřadnicovou oblast vrstvy určuje vnitřní panel (*containLayerPanel*), zbytek vrstvy vyplňuje okraje, které jsou dynamicky nastavitelné. To je zobrazeno na obrázku 35.



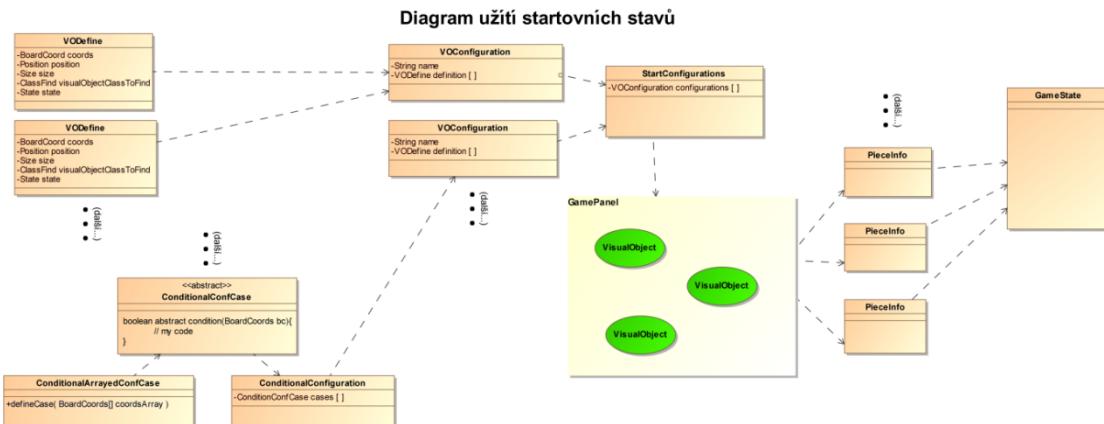
Obr. 35 - Detailní pohled na jednu vrstvu komponenty GamePanel. Znak „?“ představuje konkrétní velikost pro dané okno (např. 550 px).

Vnitřní panel má souřadnicový systém, který má nastaven absolutní Java - Layout. Do tohoto systému můžeme umístit libovolné hrací prvky (reprezentované třídou *VisualObject*). Každý takový hrací prvek má svojí souřadnici pozice (třída *Position*). Ta může být absolutní v pixelech nebo relativní v rozsahu 0-1, která je poté podle celkové velikosti přepočítána na absolutní. Každý hrací objekt má krom polohy také svou velikost (třída *Size*), která je rovněž absolutní či relativní ve stejném smyslu jako u souřadnice pozice.

## 14. Reprezentace startovních stavů

Startovní pozice lze definovat různými způsoby. Potřebné údaje o hracích prvcích můžeme vložit pomocí objektu *VODefine* a spojením těchto objektů pak máme *VOConfiguration* objekt. Pokud chceme nadefinovat více stejných prvků na různé pozice, jde to pomocí třídy *ConditionalCase*, kde bud' pomocí pole specifikujeme, o jaké souřadnice půjde (*ConditionArrayedConfCase*), nebo specifikujeme souřadnice pomocí kódu - anonymní třídy, který nadefinujeme v třídě *ConditionalConfCase*. Tyto případy (*Cases*) spojíme a pomocí *ConditionalConfiguration* vygenerujeme rovněž *VOConfiguration*. Tuto cestu využívají všechny naimplementované hry. Každá hra může mít libovolný počet *VOConfiguration*, které pak dohromady utvoří startovní pozice (objekt *StartConfigurations*). Při startu hry a zvolení jedné konfigurace desky, se podle definic najdou vhodné třídy *VisualObject* a nastaví požadované proměnné stavu, pozice, velikost, atd. Tyto vizuální objekty jsou vázány přímo na hrací panel a konkrétní jeho vrstvu (viz podkapitola 13.1.2 - Vrstva hracího panelu). Informace z těchto objektů se v případě potřeby redukují a vytvoří přepravní objekty *PieceInfo*. Objekt *PieceInfo*

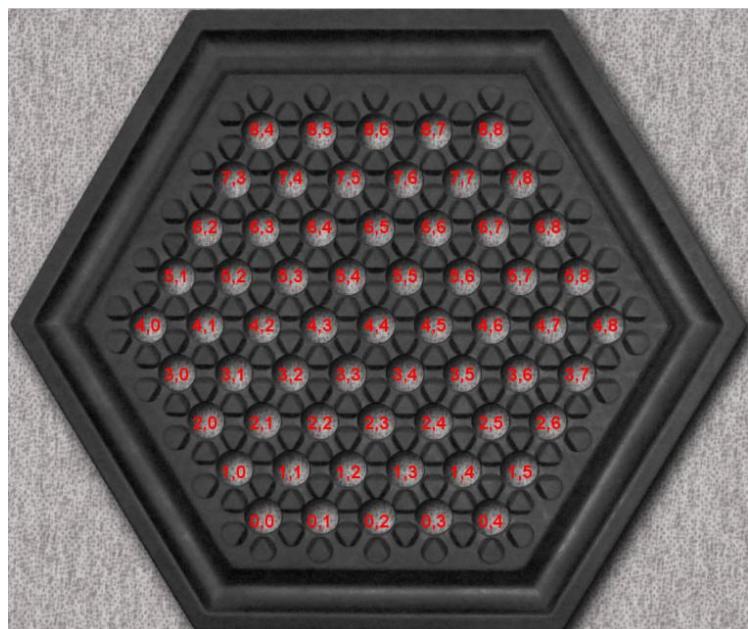
obsahuje pouze primitivní a neměnitelné prvky a představuje přechod v aplikaci z části pohled k logické části (viz podkapitola 12.2.1). Skupina objektů *PieceInfo* nakonec vytvoří herní stav *GameState*, kde cesta těchto dat ze startovní pozice končí. *GameState* se používá zejména při ukládání stavu hry nebo v umělé inteligenci. Vizuálně tento průběh je demonstrován na obrázku 36.



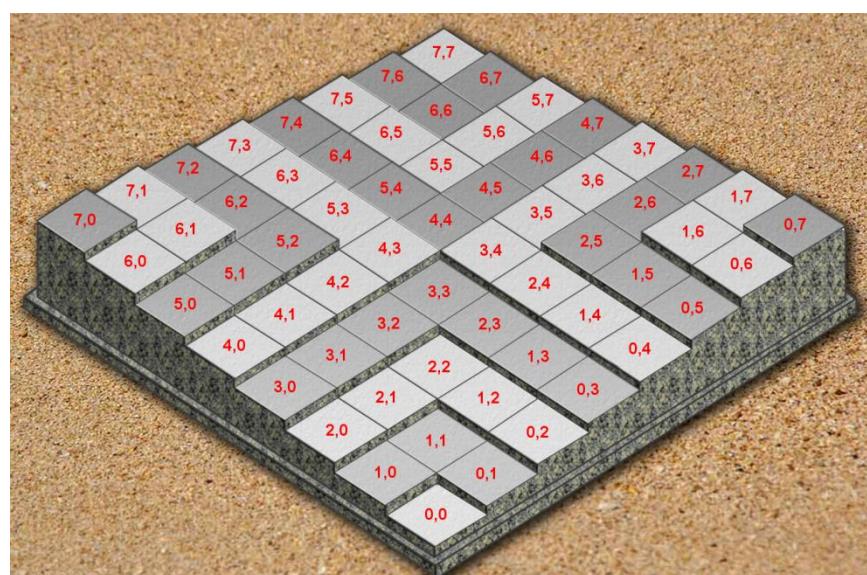
Obr. 36 - Diagram užití startovních stavů. Ilustruje způsoby definování startovních stavů a využití těchto informací skrz celou aplikaci.

## **15. Logická a datová reprezentace herní desky**

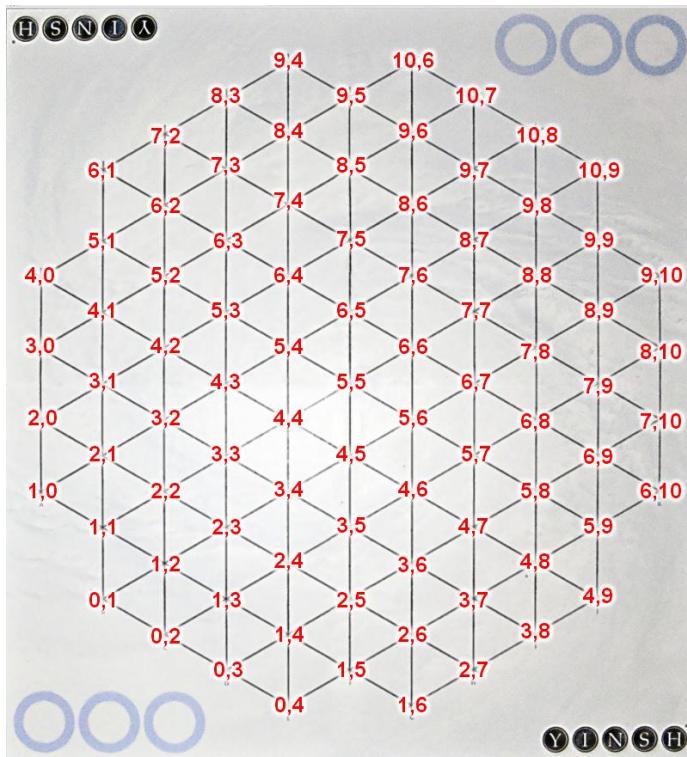
Jelikož musíme mít stále na paměti, že můžeme aplikaci rozšířit o jakoukoli hru, je reprezentace desky obecná, tudíž méně efektivní, ale více přizpůsobitelná. Desku obecně tvoří souřadný systém souřadnic  $x$ ,  $y$ , který je popsán třídou *BoardCoords*. Tyto souřadnice nemusí odpovídat horizontální a vertikální ose. Souřadnice rozhodně nijak nereflektují grafické souřadnice na obrazovce (zobrazený objekt může být např. v různé rotaci nezávisle). Rovněž není definován jejich rozsah, ač se předpokládá, že budou nezáporné. Výjimky tvoří 2 speciální případy a to souřadnice identifikující hrací prvek mimo plochu *OUT\_OF\_BOARD\_COORDS* (souřadnice  $-2^{31}, -2^{31}$ ) a nezobrazitelné neznámé souřadnice *NONEEXIST\_BOARD\_COORDS* (souřadnice  $-2^{31}+1, -2^{31}+1$ ). Jakým způsobem se namapují souřadnice na jednotlivá pole není definováno, pro implementované hry jsou souřadnice pro jednotlivé pole vyobrazeny níže :



Obr. 37 - Souřadnice polí hry Abalone v implementaci. Uprostřed polí jsou souřadnice vypsány ve formátu x, y.



Obr. 38 - Souřadnice polí hry Terrace v implementaci. Uprostřed polí jsou souřadnice vypsány ve formátu x, y.



Obr. 39 - Souřadnice polí hry Y Cindy v implementaci. Uprostřed polí jsou souřadnice vypsány ve formátu x, y.

## 16. Reprezentace hracích prvků

Každý hrací prvek (*VisualObject*), ať jde o pole či kámen, má vnitřní stav popsaný třídou *State*. Třídu *State* lze objektově rozšířit a přidat tím různé informace či funkce vztahující se ke stavu konkrétního prvku. Aby bylo možno s takovým stavem pracovat, poskytuje navenek pole hodnot (v aplikaci pojmenováno jako *intValues*), které ho plně definuje. U tohoto pole je pravidlo, že první prvek představuje vždy index hráče, na ostatní prvky požadavky nejsou. Následující seznam obsahuje nejpoužívanější nadtřídy používající stav pro implementované hry.

- *PieceState* - Prvky, které používají tento stav si pamatují přiřazeného hráče z výčtu všech hráčů. Hráč je reprezentován číslem, kde -1 znamená, že tento prvek nepatří žádnému hráči. Při hodnotě 0 a větší patří prvek hráči s odpovídajícím indexem. Např. pokud hrají 2 hráči, jsou jejich indexy 0 a 1, atd. Pole hodnot je [-1... počet hráčů].
- *PieceStateTerrace* – Stav je rozšířením *PieceState* pro Terrace. Zde se navíc pamatuje velikost kamenu ("lines" - 4 velikosti možné s hodnotami 0-3) a zda jde o T kámen (ano/ne s hodnotami 1, 0). Pole hodnot je [-1... počet hráčů, 0... 3, 0... 1].
- *CellState* - Rozlišují se zde typy polí (např. u šachů by to mohly být černé a bílé pole). Obecně tyto typy jsou definovány pomocí hodnot od nuly. Pole hodnot je [-1, 0... počet typu polí]
- *EmptyState* - Standardní prázdný stav. Pole hodnot je [-1].

Použití u jednotlivých her:

- Abalone
  - *NullCell - EmptyState*
  - *AbaloneBall - PieceState*
- Terrace
  - *NullCell - EmptyState*
  - *TerracePiece - PieceStateTerrace*
- Yinsh
  - *NullCell - EmptyState*
  - *YinshMarker - PieceState*
  - *YinshRing - PieceState*

## 17. Implementace heuristických funkcí

Pro každou hru bylo implementováno několik různých heuristických funkcí. V následující podkapitole si rozebereme jednotlivé heuristické funkce u každé hry. Všechny heuristické funkce implementují rozhraní *Heuristicable*. Společně s funkcemi také budeme popisovat i ohodnocování výhry, ohodnocování remízy a heuristiku řazení tahů. A to proto, že každá heuristika implementující třídu *Heuristicable* definuje všechny tyto aspekty.

### 17.1. Obecné vlastnosti heuristických funkcí

Obecně všechny heuristiky dodržují následující kontrakt týkající se rozmezí vrácených hodnot definovaný algoritmy.

Heuristická funkce pozice (*getStateRank*) - Heuristika musí vracet vždy hodnoty mezi 0 a *POSITIVE\_INF* ( $2^{31}-1$ ). Čím pozice je lepší z pohledu hráče, který byl použit jako parametr metody, tím hodnota by měla být větší. Na rozdíl od popisu Minimaxu (v podkapitole 8.1), zde heuristika nezohledňuje znaménko podle hráče. A to proto, že může být použita i ve hře s více než 2 hráči, kde je nutné, aby správné znaménko rozhodoval samotný algoritmus.

- *Ohodnocení vítězství (getFinalWinnerStateRank)* - Pro správnou funkci algoritmu by měla tato funkce vracet nezápornou hodnotu, která by měla být vyšší než všechny možné ohodnocení pozice stejného hráče.
- *Ohodnocení remízy (getFinalDrawStateRank)* – U toho ohodnocení není kladeno omezení na hodnotu, ale mělo by toto ohodnocení být blízko neutrálnímu zisku hráče (většinou 0). Někdy toto ohodnocení je menší než neutrální zisk hráče (např. heuristika Abalone - *Standard*, viz níže), což má za následek větší snahu hráče usilovat o výhru, ač to může vést ke ztrátám.
- *Heuristická funkce řazení tahu (movesOrderCompareRank)* - Heuristika nemá omezení na hodnotu. Tato hodnota se nepoužívá přímo v algoritmech, ale jen při řazení samotném.

Podle doporučení z [55] byl celkový vypočet podle jednotlivých vlastností ohodnocován standardizován na **váženou lineární funkci**, kde v představuje vlastnost a  $w$  váhu dané vlastnosti pro pozici  $p$ :

$$\text{ohodnocení} = \sum_i v_i w_i(p) \quad (10)$$

Tím docílíme toho, že jednotlivé vlastnosti jsou vzájemně na sobě nezávislé a jejich funkční průběh je monotónní.

## 17.2. Abalone

### 17.2.1. Seznam vlastností

Abalone je z vybraných her nejzkoumanější hra, a proto existuje více myšlenek, podle kterých jsem se inspiroval, jak heuristikou funkci sestavit. Souhrnně si uvedeme vlastnosti všech Abalone heuristik, podle kterých můžeme pozice hodnotit.

#### 17.2.1.1. Vlastnosti při ohodnocování pozic

A0 - Základní skóre - Výchozí hodnota ohodnocení

Jelikož cílem hry je vytlačovat koule z desky, důležitou vlastností je počet vytlačených kouli. Vzhledem k implementaci tuto vlastnost měříme přes počet koulí hráče.

A1 - Počet vlastních koulí - Počet koulí na desce vlastního hráče (hráče, z jehož pohledu se ohodnocuje)

A2 - Počet koulí soupeřů - Součet koulí soupeřů

Každý hráč by se měl snažit obsadit střed desky, jelikož středové pozice jsou silné. Ve středu nehrozí vytlačení a zároveň je možné vytlačovat soupeře z všech stran ven z desky. Proto je důležité, aby koule hráče byly co nejbliže středu desky. Vzdálenost od středu měříme pomocí Manhattanové vzdálenosti [40]. Je to počet polí, přes které bychom museli projít mezi měřenými body. U Abalone nemůžeme použít obvyklý výpočet  $|x_1 - x_2| + |y_1 - y_2|$ , ale musíme použít následující výpočet pro šestiúhelníkovou desku:

$$\frac{|x_1 - x_2| + |y_1 - y_2| + ||x_1 - x_2| - |y_1 - y_2||}{2} \quad (11)$$

*A3 - Vzdálenost od středu vlastních koulí* - Součet vzdáleností od středu všech vlastních koulí.

*A4 - Vzdálenost od středu koulí soupeřů* - Součet vzdáleností od středu všech koulí soupeřů.

Další důležitou vlastnost představuje vzájemná vzdálenost koulí. Velkou výhodu mají hráči, kteří drží své koule pevně u sebe. Na takovéto uskupení se špatně útočí a sami mají velikou útočnou sílu. Proto heuristika měří seskupování koulí. Hodnota seskupování je počet sousedních koulí stejněho hráče sečtený v rámci všech koulí hráče.

*A5- Seskupování vlastního hráče* - Součet počtu sousedních vlastních kouli, u všech vlastních koulí.

*A6- Seskupování soupeřů* - Součet počtu sousedních nevlastních kouli, u všech nevlastních koulí.

Abychom mohli vytlačit soupeře z desky, je nutné provést přetlačení a poté vytlačení z desky.

*A7 - Útočení na soupeře* - Hodnota útočení na soupeře právě vyjadřuje počet koulí soupeře, které jdou nějakým tahem přetlačit. Pokud nějakou kouli lze přetlačit více různými tahy, počítá se pouze jednou.

*A8 - Útočení od soupeřů* - Hodnota útočení od soupeřů znamená, kolik vlastních kouli můžou v součtu soupeři přetlačit. Pokud nějakou kouli lze přetlačit více různými tahy, počítá se pouze jednou.

Další vlastnost, podle které můžeme vyhodnocovat je rozrušování skupin koulí. Vlastnost porovnává, jak moc byly silné seskupení koulí rozrušovány a oddělovány.

*A9 - Rozrušování skupin koulí soupeřů* - Počet výskytů situace, kdy u vlastní koule sousedí koule soupeře a současně na protější straně také sousedí koule soupeře.

*A10 - Rozrušování skupin vlastních koulí* - Počet výskytů situace, kdy u soupeřovy koule sousedí vlastní koule a současně na protější straně také sousedí vlastní koule.

Podobnou vlastnost představuje posilování skupin koulí. Vlastnost porovnává, jak je daný hráč schopen držet nepřátelské koule na kraji své skupiny.

*A11 - Posilování skupin vlastních koulí* - Počet výskytu situace, kdy u vlastní koule sousedí vlastní koule a současně na protější straně sousedí koule soupeře.

*A12 - Posilování skupin koulí soupeřů* - Počet výskytu situace, kdy u soupeřovy koule sousedí jeho koule a současně na protější straně sousedí vlastní koule.

Alternativní možnost, jak zhodnocovat pozice koulí je "skupinová vzdálenost". Vyjadřuje, jak vzájemně jsou k sobě a k středu desky postaveny skupiny koulí hráčů. Tato vzdálenost se počítá vždy od referenčního pole. Toto pole získáme tak, že vezmeme průměrnou souřadnici (průměry umístění na jednotlivých osách) vlastních koulí, průměrnou

souřadnici koulí soupeře a střed desky. Zprůměrováním těchto tří souřadnic a zaokrouhlením dostaneme výslednou souřadnici referenčního pole.

*A13 - Skupinová vzdálenost vlastních koulí* - Součet Manhattanských vzdáleností (viz výše) mezi referenčním bodem a každou vlastní koulí.

*A14 - Skupinová vzdálenost koulí soupeře* - Součet Manhattanských vzdáleností (viz výše) mezi referenčním bodem a každou koulí soupeře.

### 17.2.1.2. Vlastnosti při řazení tahů

*AR0 - Počet pohnutých koulí* - Počítá se kolika koulemi pohybujeme v tahu, tzn. počet 1,2 nebo 3 podle pravidel.

*AR1 - Typ tahu* - Hodnota určená typem tahu podle následujících možností:

- *-1* - dopředný potlačovací tah, kde byla z desky vytlačena vlastní koule
- *0* - tah do strany
- *1* - dopředný tah bez přetlačování
- *2* - dopředný tah s přetlačením nějaké koule
- *3* - dopředný tah s přetlačením a vytlačením koule soupeře

*AR2 - tah k středu* - Hodnota vyjadřuje, o kolik polí se posouváná skupina koulí v tahu posunula ke středu desky.

*AR3 - Ohodnocení heuristikou funkcí pozice* - Na výslednou pozici po provedení tahu, se zavolá ohodnocování pozice (*getStateRank*) ve stejném typu heuristiky.

### 17.2.2. Seznam heuristik hry Abalone

- *Analyzing* - Heuristika inspirována podle [27]. Váhy některých vlastností počítá váženým průměrem podle počtu koulí.
- *Experimental* - Heuristika inspirována podle [28].
- *Mass Based* - Heuristika inspirována podle [32].
- *Standard* - Heuristika lehce inspirována podle [22]
- *Simple* - Základní heuristika s minimem vlastností zaměřená na rychlosť. Heuristika řazení tahů v ní nepracuje (vrací vždy 0).

### 17.2.3. Tabulka vah jednotlivých heuristik

Následující tabulka zobrazuje hodnoty vah pro jednotlivé hodnoty vlastností. Pokud je pole tabulky prázdné, znamená to, že váha je 0 (vlastnost se nepodílí na výsledku). V případě, že u vlastnosti je uvedeno "(konst. 1x)", značí to, že výsledná hodnota vlastnosti je vždy 1 (na celkový součet má vliv tedy jen váha této vlastnosti). Výsledné ohodnocení se počítá podle vzorce (11). Vlastnosti začínající s kódem obsahujícím „R“ patří k ohodnocení při řazení tahů. Ostatní vlastnosti patří k ohodnocení pozice.

Vlastnost \ Heuristika název	Analyzing	Experimental	Mass Based	Standard	Simple
<b>Výhra (konst. 1x)</b>	1000000	400000	5000	10000	500
<b>Remíza (konst. 1x)</b>	100000	100000	1000	900	0
<b>A0 - Základní skóre (konst. 1x)</b>	100000	100000	1000	1000	
<b>A1 - Počet vlastních koulí</b>	2000	10000	120	25	2
<b>A2 - Počet koulí soupeřů</b>	-2000	-10000	-120	-25	1
<b>A3 - Vzdálenost od středu vlastních koulí</b>	-500 / A1	-280		-1	
<b>A4 - Vzdálenost od středu koulí soupeřů</b>	500 / A2	280			
<b>A5 - Seskupování vlastního hráče</b>	1500 / A1	215		1	
<b>A6 - Seskupování soupeřů</b>	-1500 / A2	-215			
<b>A7 - Útočení na soupeře</b>	40				
<b>A8 - Útočení od soupeřů</b>	-40				
<b>A9 - Rozrušování skupin koulí soupeřů</b>		200			
<b>A10 - Rozrušování skupin vlastních koulí</b>		-200			
<b>A11 - Posilování skupin vlastních koulí</b>		290			
<b>A12 - Posilování skupin koulí soupeřů</b>		-290			
<b>A13 - Skupinová vzdálenost vlastní koulí</b>			-10		
<b>A14 - Skupinová vzdálenost koulí soupeře</b>			10		
<b>AR0 - Počet pohnutých koulí</b>	1	1	1		
<b>AR1 - Typ tahu</b>	3	3			
<b>AR2 - Tah k středu</b>			4		
<b>AR3 - Ohodnocení heur. funkcí pozice</b>				1	

Tab. 1 - Váhy vlastností heuristik hry Abalone.

## 17.3. Terrace

### 17.3.1. Seznam vlastností

Terrace patří k nejméně prozkoumané hře, a proto veškeré heuristiky jsou převážně založené na vlastní zkušenosti ze hry.

#### 17.3.1.1. Vlastnosti při ohodnocování pozic

*T0 - Základní skóre* - Výchozí hodnota ohodnocení.

Jelikož jedním z cílů je vyhodit cizí T kámen, budeme zjišťovat počet T kamenů.

*T1 - Počet vlastních T kamenů* - Vlastnost nepřímo zjišťující, zda vlastní T kámen byl vyhozen nebo ne.

*T2 - Počet T kamenů soupeřů* - Vlastnost nepřímo zjišťující, zda T kámen soupeře byl vyhozen nebo ne.

Druhým cílem je dostat se s vlastním T kamenem do protějšího rohu s nejnižší výškou.

*T3 - Vzdálenost vlastního T kamenu k cílovému poli* - Spočítá počet tahů nutných k vykonání vlastním T kamenem vykonat, aby se dostal do cílového pole<sup>5</sup>. A tato hodnota se odečte od 10, jelikož maximální možná vzdálenost na desce 8x8 je 9. Abychom dostali kladné nenulové číslo použijeme tedy 10.

---

<sup>5</sup> Kvůli rychlosti je výpočet počítající počet tahů approximativní a neuvažuje ostatní kameny na desce.

*T4 - Vzdálenost T kamenu soupeře* - Hodnota se určuje podobně jako u vlastnosti T3, přičemž se počítá počet tahů nutných k vykonání všema T kameny soupeřů, aby se každý z nich dostal do cílové pozice.

Ve hře je rovněž důležité si udržet kameny, aniž by je spoluhráči vyhodili.

*T5 - Celková velikost vlastních kamenů* - Součet velikostí vlastních kamenů, kde nejmenší kamen (kámen A) má hodnotu 1 a největší (kámen D) 4.

*T6 - Celková velikost kamenů soupeřů* - Součet velikosti kamenů soupeřů, kde nejmenší kamen (kámen A) má hodnotu 1 a největší (kámen D) 4.

Mezi další důležité vlastnosti patří mobility jednotlivých kamenů. Mobilita kamene je počet různých polí, kam lze pomocí kamene táhnout.

*T7 - Mobilita vlastních kamenů* - Součet mobility každého vlastního kamene vyjma vlastního T kamene.

*T8 - Mobilita kamenů soupeře* - Součet mobility každého kamene soupeře vyjma T kamenu soupeřů.

Alternativně lze určovat mobilitu vztaženou na terasu (pole na stejně úrovni v určitém kvadrantu) a na velikost kamene. Tím budeme motivovat heuristiku ke hře do středových teras, které mají nejvíce polí a je zde nejlepší mobilita.

*T9 - Alternativní mobilita vlastních kamenů* - Součet mobility vlastních kamenů, vzhledem k velikosti a terase. Tu spočítáme tak, že velikost kamene (hodnota 1-4 podle velikosti, viz T5, T6) vynásobíme počtem polí terasy, na které se kámen vyskytuje.

*T10 - Alternativní mobilita kamenů soupeřů* - Součet mobility kamenů soupeřů, vzhledem k velikosti a terase. Tu spočítáme tak, že velikost kamene (hodnota 1-4 podle velikosti, viz T5, T6) vynásobíme počtem polí terasy, na které se kamen vyskytuje.

*T11 - Obsazenost teras vlastními kameny* - Obsazenost určuje, jak se odlišuje počet vlastních kamenů na dané terase oproti optimálnímu. Umožňuje pak zjistit tahy, kde jsou kameny sice na správných terasách, ale na těchto terasách je jich velmi mnoho nebo málo. To pak může snižovat mobilitu nebo schopnost útoku na soupeře. Hodnota této vlastnosti se pro každý kámen vlastního hráče spočítá podle následujících vzorců:

$$\begin{aligned}
 & obsazenost_{\text{optimální}}(\text{výška terasy}) = \\
 & = \begin{cases} 8 - \text{výška terasy}, & \text{výška terasy} \geq 4 \\ \text{výška terasy} + 1, & \text{výška terasy} < 4 \end{cases} \quad (12)
 \end{aligned}$$

$$\begin{aligned}
 & obsazenost_{\text{kamene}}(\text{výška pole kamene}) = \\
 & = obsazenost_{\text{optimální}}(\text{výška pole kamene}) \\
 & - |obsazenost_{\text{optimální}}(\text{výška pole kamene}) \\
 & - obsazenost_{\text{vlastními kameny aktuální}}(\text{výška pole kamene})| \quad (13)
 \end{aligned}$$

### 17.3.1.2. Vlastnosti heuristik při řazení tahů

*TR0 - Tah provedený T kamenem* - Hodnota je 1, pokud byl tah proveden T kamenem.

*TR1 - Vyhozen kámen vlastního hráče* - Hodnota určuje počet vyhozených vlastních kamenů tahem (podle pravidel to může být 0 nebo 1 kamenů).

*TR2 - Vyhozen kámen soupeře* - Hodnota určuje počet vyhozených vlastních kamenů tahem (podle pravidel to může být 0 nebo 1 kamenů).

### 17.3.2. Seznam heuristik hry Terrace

- *Improved* - Vychází ze standardní heuristiky, ale výpočet některých vlastností je změněn a urychljen. Preferuje spíše defenzivní hru (hodnoty vlastností pro soupeře jsou v absolutní hodnotě větší)
- *Standard* - Prvotní heuristika.
- *Fast* - Mírně urychlěná Improved heuristika.

### 17.3.3. Tabulka vah jednotlivých heuristik

Vysvětlení významů jednotlivých polí je napsáno v stejně podkapitole u hry Abalone.

Vlastnost \ Heuristika název	Improved	Fast	Standard
<b>Výhra (konst. 1x)</b>	20000	20000	10000
<b>Remíza (konst. 1x)</b>	8500	8500	4000
<b>T0 - Základní skóre (konst. 1x)</b>	10000	10000	5000
<b>T1 - Počet vlastních T kamenů</b>	5000	5000	1600
<b>T2 - Počet T kamenu soupeřů</b>	-5000	-5000	-1600
<b>T3 - Vzdálenost vlastního T kamenu k cíli</b>	66	66	35
<b>T4 - Vzdálenost T kamenu soupeřů k cíli</b>	-88	-88	-40
<b>T5 - Celková velikost vlastních kamenů</b>	90	90	15
<b>T6 - Celková velikost kamenu soupeřů</b>	-120	-120	-20
<b>T7 - Mobilita vlastních kamenů</b>	3		
<b>T8 - Mobilita kamenu soupeřů</b>	4		
<b>T9 - Alternativní mobilita vlastních kamenů</b>			4
<b>T10 - Alternativní mobilita kamenu soupeřů</b>			-4
<b>T11 - Obsazenost teras vlastními kameny</b>			23
<b>TR0 - Tah provedený T kamenem</b>	1	1	1
<b>TR1 - Vyhozen kámen vlastního hráče</b>	-2	-2	-2
<b>TR2 - Vyhozen kamene soupeře</b>	2	2	2

Tab. 2 - Váhy vlastností heuristik hry Terrace.

## 17.4. Yinsh

### 17.4.1. Seznam vlastností

Heuristiky hry Yinsh se vyznačují relativní složitostí při získávání jejich vlastností. Je to z důvodu, že vždy je nutno posuzovat vztahy kamenů vůči sobě, někdy je určité uskupení výhodné, ale v jiné situaci to může být naopak.

#### 17.4.1.1. Vlastnosti ve fázi pokládání kroužků

##### 17.4.1.1.1. Vlastnosti při ohodnocování pozic

YP0 - Základní skóre - Výchozí hodnota ohodnocení.

YP1 - Počet kroužků na osách od vlastní kroužků - Spočítá součet kroužků (vlastních nebo soupeřů) na osách pro každý vlastní kroužek. Poloha na "ose" mezi kroužky znamená, že tyto kroužky leží na shodné přímce představující jednu z přímek desky. Tato vlastnost vyplývá ze zkušenosti, že není dobré pokládat kroužek do přímky s jiným kroužkem, protože je pak omezen při pohybu.

YP2 - Mobilita vlastních kroužků na prázdné desce - Za každý vlastní kroužek se určí počet polí, na které je možno přemístit tento kroužek s předpokladem, že se neuvažuje přítomnost jiných kamenů na desce. Pro standardní desku je tento počet polí v rozmezí 18 až 28. Výsledná hodnota této vlastnosti je součet

těchto počtů polí od všech vlastních kroužků. Vlastnost usměrňuje vkládání kroužku do míst s největší mobilitou, což jsou pole těsně vedle středu.

*YP3 - Okrajová mobilita vlastních kroužků na prázdné desce* - Odpovídá vlastností mobility jako u YP2, jen hodnota se určuje jako

$$hodnota = (28 - \text{minimální mobilita z vlastních kroužků}) \quad (14)$$

Vlastnost tím zajišťuje, aby se preferovaly pozice více k okrajům.

#### 17.4.1.1.2. Vlastnosti heuristik řazení tahů

*YRPO - Mobilita umístěného vlastního kroužku na desce* - Hodnota odpovídá mobilitě vlastnosti YP2. Počítá se pouze pro umístěný kroužek v daném tahu.

#### 17.4.1.2. Vlastnosti ve fázi tvorby řad

##### 17.4.1.2.1. Vlastnosti při ohodnocování pozic

*Y0 - Základní skóre* - Výchozí hodnota ohodnocení.

*Y1 - Rozdíl počtu kroužků* - Jelikož hra je založena na odebírání kroužků z desky při vytvoření řady, je počet kroužků na desce důležitý. Rozdíl kroužků se určuje podle vzorce: *soupeřovy kroužky – vlastní kroužky*

Následující vlastnosti se zaměřují na situace, kdy je vytvořena sekvence 4 kamenů stejné barvy u nějakého kroužku. To poskytuje šanci hráči vytvořit řadu jedním tahem.

*Y2 - Počet sekvencí 4 kamenů od kroužku vlastní barvy* - Počet všech sekvencí 4 kamenů od všech vlastních kroužků stejné barvy v jakémkoli směru.

*Y3 - Počet sekvencí 4 kamenů od kroužku barvy soupeře* - Počet všech sekvencí 4 kamenů stejné barvy od všech kroužků soupeře v jakémkoli směru.

*Y4 - Počet sekvencí 4 kamenů vlastní barvy od kroužku vlastní barvy* - Počet všech sekvencí 4 kamenů vlastní barvy od všech vlastních kroužků v jakémkoli směru. Zde mohu utvořit řadu přidáním kamenu na okraj sekvence

*Y5 - Počet sekvencí 4 kamenů soupeřovi barvy od kroužku barvy soupeře* - Odpovídá vlastnosti Y4 pro soupeře.

*Y6 - Počet sekvencí 4 kamenů soupeřovi barvy od kroužků vlastní barvy* - Počet všech sekvencí 4 kamenů barvy soupeře od všech vlastních kroužků v jakémkoli směru. Zde mohu utvořit řadu přidáním kamenu na okraj sekvence a otočením kamenů v sekvenci.

*Y7 - Počet sekvencí 4 kamenů vlastní barvy od kroužků barvy soupeře* - Odpovídá vlastnosti Y6 pro soupeře.

Následné vlastnosti zohledňují, jak moc mají hráči seskupené kroužky a kameny u středu desky. To je důležitá vlastnost pro mobilitu a dynamiku daného hráče.

*Y8 - Vzdálenost vlastních kroužků od středu desky* - Součet Manhattanských vzdáleností (viz vzorec (11)) všech vlastních kroužků od středu desky.

*Y9 - Vzdálenost kroužků soupeře od středu desky* - Součet Manhattanských vzdáleností (viz vzorec (11)) všech kroužků soupeře od středu desky.

*Y10 - Vzdálenost vlastních kamenů od středu desky* - Součet Manhattanských vzdáleností (viz vzorec (11)) všech vlastních kamenů od středu desky.

*Y11 - Vzdálenost kamenů soupeře od středu desky* - Součet Manhattanských vzdáleností (viz vzorec (11)) všech kamenů soupeře od středu desky.

Dále používáme také vlastnosti kvantifikující počty kamenu a kroužků.

*Y12 - Počet vlastních kroužků*

*Y13 - Počet kroužků soupeře*

*Y14 - Počet vlastních kamenů*

*Y15 - Počet kamenů soupeře*

*Y16 - Rozdíl počtu kamenů – Rozdíl kamenů podle vzorce:*

$$vlastní kameny - kameny soupeře \quad (15)$$

Další způsob, kterým je možné vytvořit řadu, vychází ze situace, kdy máme kroužek uprostřed 4 kamenů stejné barvy (resp. tyto kameny s tímto kroužkem tvoří sekvenci - leží za sebou na přímce). Řadu poté můžeme vytvořit tak, že tímto kroužkem odskočíme v jiném směru, než leží tyto kameny (uděláme tah, kterým neotočíme žádny z kamenů)

*Y17 - Vlastní kroužek uprostřed 4 vlastních kamenů* - Počet výskytů situací, kdy vlastní kroužek bude uprostřed 4 vlastních kamenů. Tato situace u jednoho kroužku může vzniknout až 3x, protože se vždy zkoumá protilehlá dvojice směrů (jedna z os). Situaci můžeme přesněji popsat tak, že na jedné ze stran máme x kamenů vlastní barvy a na protilehlé máme y kamenů vlastní barvy, přičemž  $x = \{x \in \mathbb{Z}; x \neq 1, 3\}$  a  $y = \{y \in \mathbb{Z}; y \neq 4 - x, 3\}$ .

*Y18 - Soupeřův kroužek uprostřed 4 soupeřových kamenů* - Odpovídá vlastnosti Y17 pro soupeřovy kameny a soupeřův kroužek.

Jiný způsob, jak vytvořit řadu vede přes situaci, kdy se při přesunu kroužku otočí kámen. Pokud tento kámen měl okolo sebe v přímce před otočením sekvenci opačné barvy, tak se vytvoří řada.

*Y19 - Otočení kamene soupeře přesunem vlastního kroužku* - Celkový počet situací, kdy mohu přesunout vlastní kroužek tak, že otočím soupeřův kámen, a tím bych vytvořil vlastní řadu.

*Y20 - Otočení vlastního kamene přesunem kroužku soupeře* - Celkový počet situací, kdy mohu přesunout kroužek soupeře tak, že otočím vlastní kámen, a tím bych vytvořil soupeřovu řadu.

*Y21 - Počet tahů na každou možnou utvořitelnou vlastní řadu* - Tato vlastnost zjistí počet tahů nutných pro vytvoření každé možné řady z tohoto postavení. Za každou řadu se udělí určité skóre a výsledek této vlastnosti je pak celkové skóre za všechny řady.

Přesněji tato vlastnost zjistí všechny pětice polí, které mohou tvořit řadu, a v kterých je alespoň jedno pole dosažitelné nebo ovlivnitelné pohybem nějaké kroužku vlastního hráče. To znamená, že pokud v pětici polí je soupeřův kroužek, nepovede se vytvořit řadu nikdy (pokud nepředpokládáme, že soupeř spolupracuje). Pokud je v pětici polí více vlastních kroužků než 1, potřebujeme tahy navíc, abychom odskočili zbývajícími kroužky. Pokud zde máme více sekvencí soupeřových kamenů, tak potřebujeme vždy tah na přeskočení této sekvence a obrácení, pokud je to možné. V opačném případě se počítá tah na každý soupeřův kámen, protože bychom je museli jakoby přeskakovat ze strany. Podle počtu tahů na jednu pětici přidělíme jí následující skóre:

$$\text{skóre (počet tahů)} = \begin{cases} 50, & \text{počet tahů} = 0 \\ 14, & \text{počet tahů} = 1 \\ 4, & \text{počet tahů} = 2 \\ 1, & \text{počet tahů} = 3 \\ 0, & \text{počet tahů} > 3 \end{cases} \quad (16)$$

přičemž, platí

$$\text{skóre (počet tahů)} \geq \sum_{i=1}^{3-\text{počet tahů}} \text{skóre (počet tahů} + i)(3 + i - \text{počet tahů}) \quad (17)$$

, kde  $3 > \text{počet tahů} \geq 0$

Celkový součet skóre pro každou pětici tvoří výslednou hodnotu vlastnosti.

#### 17.4.1.2.2. Vlastnosti heuristik řazení tahů

*YR0 - Utvoření sekvencí vlastního hráče ze změněných kamenů* - Vlastnost prozkoumá tvorbu nových sekvencí všech nově přidaných / otočených kamenů. Zjišťují se sekvence se změnou kamenů na jejích krajích, tak i uprostřed nich. Celková hodnota této vlastnosti je součet bodů za jednotlivé sekvence (tj. skupina po sobě jdoucích kamenů stejné barvy) podle jejich délky. Body za sekvence se počítají podle skóre:

$$\text{body za sekvencí} = \sum_{i=3}^{\begin{cases} \text{délka sekvence}, & \text{délka sekvence} < 5 \\ 5, & \text{délka sekvence} \geq 5 \end{cases}} (2i - 3) \quad (18)$$

*YR1 - Utvoření sekvencí soupeře ze změněných kamenů* - Počítá se podobně jako YR0, ale pro soupeře.

### 17.4.2. Seznam heuristik hry Yinsh

- *Storm* - heuristika inspirována podle [33]. Počítá různé vlastnosti a průměruje je vzhledem k počtu daných kamenů, či kroužků.
- *ByPosition* - Heuristika založená na vlastnostech pětic polí. Má složitější vlastnosti ohodnocování při fázi umisťování řad. Význam jednotlivých typů je:
  - *Edge* - vyhodnocuje se v situaci, kdy se umisťuje 3. kroužek hráče. Hodnoty se vztahují pouze na kroužek s minimální mobilitou (YP2), ostatní kroužky se počítají podle typu *Normal*.
  - *Block* - vyhodnocuje se v situaci, kdy se umisťuje 3. kroužek hráče. Hodnoty se vztahují pouze na kroužek s největším počtem ostatních kroužků na stejných osách (YP1), ostatní kroužky se počítají podle typu *Normal*.
  - *Normal* - vyhodnocuje se ve všech ostatních situacích.
- *Basic* - Základní heuristik s minimálním počtem vlastností, zaměřená na rychlosť. Heuristika řazení tahů v ní nepracuje (vrací 0 vždy).

### 17.4.3. Tabulka vah jednotlivých heuristik

Vysvětlení významů jednotlivých polí je opět napsáno v stejné podkapitole u hry Abalone. Jedinou výjimku, představuje písmeno „P“ v kódu vlastnosti, které značí, že jde o vlastnost ve fázi pokládání kroužku. V opačném případě jde o vlastnost ve fázi tvorby řad.

Vlastnost \ Heuristika název	Storm	ByPositions			Basic
		Normal	Block	Edge	
Výhra (konst. 1x)	1100000000	1000			1951
Remíza (konst. 1x)	1000000000	100			1
Y0 - Základní skóre (konst. 1x)	1000000000	0			1
Y1 - Rozdíl počtu kroužků	10000000	250			65
Y2 - Počet sekvencí 4 kamenů od kroužku vlastní barvy	9195 / Y12				
Y3 - Počet sekvencí 4 kamenů od kroužku barvy soupeře	-6066 / Y13				
Y4 - Počet sekvencí 4 kamenů vlastní barvy od kroužku vlastní barvy	8609 / Y12				
Y5 - Počet sekvencí 4 kamenů barvy soupeře od kroužku barvy soupeře	-3926 / Y13				
Y6 - Počet sekvencí 4 kamenů barvy soupeře od kroužku vlastní barvy	9335 / Y12				
Y7 - Počet sekvencí 4 kamenů vlastní barvy od kroužku barvy soupeře	-2469 / Y13				
Y8 - Vzdálenost vlastních kroužků od středu desky	-4025 / Y12				
Y9 - Vzdálenost kroužků soupeře od středu desky	4025 / Y13				
Y10 - Vzdálenost vlastních kamenů od středu desky	-1000 / Y14				-1
Y11 - Vzdálenost kamenů soupeře od středu desky	1000 / Y15				1
Y12 - Počet vlastních kroužků					
Y13 - Počet kroužků soupeře					
Y14 - Počet vlastních kamenů					
Y15 - Počet kamenů soupeře					
Y16 - Rozdíl počtu kamenů	294				
Y17 - Vlastní kroužek uprostřed 4 vlastních kamenů	5378 / Y12				
Y18 - Soupeřův kroužek uprostřed 4 kamenů soupeře	-6505 / Y13				
Y19 - Otočení kamene soupeře přesunem vlastního kroužku	1000 / Y12				
Y20 - Otočení vlastního kamenu přesunem kroužku soupeře	-8373 / Y13				
Y21 - Počet tahů na každou možnou utvořitelnou vlastní řadu		1			
YP0 - Základní skóre (konst. 1x)	2000	20000			2000
YP1 - Počet kroužků na osách od vlastních kroužků	-38	-380	45	-380	-38
YP2 - Mobilita vlastních kroužků na prázdné desce	1	10	10		1
YP3 - Okrajová mobilita vlastních kroužků na prázdné desce				2	
YR0 - Utvoření sekvencí vlastního hráče ze změněných kamenů	1	1			
YR1 - Utvoření sekvencí soupeře ze změněných kamenů	-1	-1			
YRPO - Mobilita umístěného vlastního kroužku na desce	1	1			

Tab. 3 - Váhy vlastností heuristik hry Yinsh.

## 18. Implementace algoritmů umělé inteligence

Z popsaných algoritmů jsem se rozhodl implementovat algoritmy Negamax a Alfa-Beta prořezávání. Algoritmus Minimax jsem neimplementoval, jelikož poskytuje stejné výsledky jako Negamax a vyžaduje duplicitní kód svou méně elegantnější úpravou. Také byl implementován další algoritmus nazvaný jako "Best Move". U každého algoritmu uvedu jeho zjednodušený kód implementace. Obecný nebo nepopsaný kód v teoretické části bude zde detailněji popsán.

### 18.1. Implementace algoritmu Best Move

Algoritmus Best Move pracuje stejně jako algoritmu Negamax s hloubkou 1 (prohledávající jeden tah dopředu). Poskytuje rovněž stejný výsledek. Jeho vlastnosti jsou stejné jako u algoritmu Negamax. S výjimkou jednodušší a rychlejší implementace oproti provádění hloubky 1 u Negamaxu. Tento algoritmus je základní a neposkytuje podporu pro zlepšující techniky a to hlavně z důvodu toho, že má vždy konstantní hloubku. Proto ho ani nemůžeme časové omezit, jelikož nepoužívá iterativní prohlubování. Avšak význam

prohlubování nebo ostatních technik je zde zanedbatelný, protože tento algoritmus prohledá pouze  $b$  stavů, kde  $b$  je faktor větvení, což probíhá velice rychle.

## 18.2. Implementace algoritmu Negamax

### 18.2.1. Pseudokód algoritmu

Na následujícím kódu samotného algoritmu Negamax si popíšeme některé vlastnosti specifické pro implementaci.

Kód není úplně přesný, protože je uváděn bez ladícího kódu a kódu pro informování GUI (procentuální průběh atd.) Rovněž některé konstrukty a organizace kódu neodpovídají realitě, aby se kód přiblížil pseudokódu uvedenému v teoretické části.

```

1 HodnotaTah negamaxImpl(Pozice p, int hloubka,int predchoziHrac,int
2 hrac, int[] skore) {
3
4     if ((konecHry(hrac) or hloubka<=0) {
5         if (jeRemiza(hrac)) {
6             int ohodnoceni =
7                 heuristika.OhodnoceniRemiza(p,hracNaTahu());
8             return new HodnotaTah(ohodnoceni,null);
9         }else if (jeVyhra(hrac)) {
10            int ohodnoceni =
11                heuristika.OhodnoceniVyhra(p,hracNaTahu());
12            HodnotaTah ht= new HodnotaTah(ohodnoceni,null);
13            if (!jeStejnyHrac(hrac,hracNaTahu()))
14                ht.hodnota=-ht.hodnota;
15            return ht;
16        }else{
17            int ohodnoceni= heuristika.Ohodnoceni(p,hracNaTahu());
18            return new HodnotaTah(ohodnoceni, null);
19        }
20    }
21    int max = -NEKONECNO;
22    Tah maxTah = null;
23    boolean minimalizujici!=jeStejnyHrac(hrac,hracNaTahu());
24
25    foreach tah of GenerujTahy(p,hrac) {
26
27        Pozice novaPozice = udelejTah(tah,p);
28        if (algoritmusPrerusen()) return PRERUSENO;
29        novaPozice.skore=pripoctiSkore(p,skore);
30        int dalsiHrac=dalsiHrac(hrac, novaPozice,novaPozice.skore);
31        HodnotaTah hodnTahAktualni;
32
33        ZaznamPozice zaznam=transpozicniTab.nacti(p,hracNaTahu());
34        if (zaznam!=null and zaznam.hloubka>=hloubka) {
35            hodnTahAktualni= new HodnotaTah(
36                zaznam.hodnota, zaznam.nejlepsiTah);
37
38        }else{
39            hodnTahAktualni =negamaxImpl(novaPozice,hloubka-1,hrac
40                                         ,dalsiHrac);
41        }
42        if (hodnTahAktualni==PRERUSENO) return PRERUSENO;
43
44        vratTah(tah, novaPozice);
45
46        if (minimalizujici) hodnTahAktualni.hodnota=
47                        -hodnTahAktualni.hodnota;
48        if (hodnTahAktualni.hodnota > max){
49            max = hodnTahAktualni.hodnota;
50            maxTah = tah;
51        }
52    }
53    if (minimalizujici) max =-max;
54    transpozicniTab.uloz(p,hloubka,PRESNE,max,maxTah,hracNaTahu());
55    return new HodnotaTah(max,maxTah);
56 }
```

### 18.2.2. Popis práce pseudokódu

Zde budou popsány pouze nové nebo změněné příkazy, popis ostatních se nachází v předchozích kapitolách o popisu pseudokódu.

Nepopsané metody:

- boolean jeRemiza(int hrac) - Metoda vrací logickou hodnotu, zda dané číslo hráče (z metody dalsiHrac(...)) představovalo remízu.
- boolean jeVyhra(int hrac) - Metoda vrací logickou hodnotu, zda dané číslo hráče (z metody dalsiHrac(..)) představovalo výhru.
- int Ohodnoceni(Pozice p,int hrac) – Metoda vrací ohodnocení heuristickou funkcí pro danou pozici  $p$  a hráče. V kódu odpovídá (*getStateRank*). Více podrobností v podkapitole o heuristické funkci 17.1.
- int OhodnoceniVyhra(Pozice p,int hrac) – Metoda vrací ohodnocení heuristickou funkcí pro danou pozici  $p$  a hráče ve stavu, kdy hráč vyhrál. V kódu odpovídá (*getFinalWinnerStateRank*). Více podrobností v podkapitole o heuristické funkci 17.1.
- int OhodnoceniRemiza(Pozice p,int hrac) – Metoda vrací ohodnocení heuristickou funkcí pro danou pozici  $p$  a hráče ve stavu, kdy nastala remíza. V kódu odpovídá (*getFinalDrawStateRank*). Více podrobností v podkapitole o heuristické funkci 17.1.
- boolean jeStejnyHrac(int hrac1,int hrac2) – Zjišťuje, zda jde o stejného hráče. Tato metoda neprovádí jen triviální  $hrac1==hrac2$ , ale také zohledňuje jiné hodnoty pro hráče z metody *dalsiHrac()*, pokud nějaký hráč např. vyhrál.
- int hracNaTahu() – Metoda vrací hráče, který má aktuálně táhnout.
- boolean algoritmusPrerusen() - Dotazuje se, zda algoritmus nebyl přerušen z nedostatku času na tah (viz iterativní prohlubování podkapitola 9.2) nebo zda uživatel neukončil hru z GUI. V kódu to představuje dotaz na Interrupt flag aktuálního vlákna.
- int[] pripoctiSkore(Pozice p ,int[] skore) - Ke uloženému skóre každého hráče v pozici  $p$  ( $p.skore$ ) přičte skóre z parametru a výsledek vrátí.
- int dalsiHrac(int hrac, Pozice p,int[] skore) - Zjistí, který hráč bude hrát v následujícím tahu v závislosti na pozici  $p$  a skóre. Ač může zdát, že se hráči pouze střídají, tak to rozhodně neplatí. Může nastat terminální stav - remíza nebo výhra některého hráče, pro takové situace jsou vyhrazeny speciální návratové hodnoty. V případě, že hráč nemůže táhnout, metoda vrací dalšího hráče, který je toho schopen (pouze v případě, že to pravidla hry definují).

Příklad volání algoritmu:

```
HodnotaTah nejlepsiTah = negamaxImpl(aktualniPozice(),3,0,1,[0,0])
```

Vrací nejlepší tah pro hloubku 3. Přičemž tento tah se hledá pro hráče s indexem 1 (hráč s indexem 0 hrál předtím). Aktuální skóre hráčů je nulové.

Na rozdíl od teoretické části, u implementace se navíc předávají parametry hráčů a skóre. Skóre je důležité pro určování dalšího hráče a detekci výhry v `dalsiHrac()` na řádku 27. Hráče je nutné si předávat, kvůli tomu, že není jistota střídání maximalizujícího a minimalizujícího hráče. To platí hlavně pro hry s více hráči než dvěma nebo v případě existence vyněchaných tahů nějakého hráče. Proto nenegujeme ohodnocení, ale necháváme ho nezáporné. Z toho plyne požadavek na heuristiku, aby výsledek z metod `Ohodonoceni()`, `OhodnoceniVyhra()`, `OhodnoceniRemiza()` nebyl záporný. Více o heuristických funkcích v kapitole 17.1. Algoritmus zjišťuje podle čísel hráčů, zda má minimalizovat či maximalizovat ohodnocení. Na řádku 14 se v případě vítězství soupeře mění znaménko, aby tento stav nebyl zvolen. Zda bude algoritmus minimalizovat, rozhoduje logická hodnota `minimalizujici` na řádce 23. Podle této hodnoty algoritmus převrátí hodnotu 46, aby ji převrátil po zjištění nejlepšího tahu na řádce 53 zpět. Důležité je také, že v implementaci se přesunul kód transpozičních tabulek dovnitř cyklu `for`. To je důsledek zefektivnění místa v transpoziční tabulce, kam nebudeme ukládat koncové stavy splňující podmínu na řádce 4. Co se týče tabulky, je volána s novým parametrem a to indexem hráče na tahu. Je to z důvodu nutnosti rozlišování jednotlivých hráčů na tahu v tabulce, protože i stejný stav, může mít obecně pro hráče různý význam.

## 18.3. Implementace algoritmu Alfa-Beta prořezávání

### 18.3.1. Pseudokód algoritmu

Podobně jako u Negamaxu i zde si popíšeme kód algoritmu Alfa-Beta. Budeme se zabývat hlavně odlišnostmi od implementace Negamaxu. Pro stručnost zde není uvedena volající metoda pro iterativní prohlubování, která se téměř shoduje s pseudokódem popsaným v podkapitole 9.2.1. Kód, který se liší od `negamaxImpl` je vyznačen tučně.



```

62         }else{
63             hodnTahAktualni=alfabetaImpl(novaPozice,hloubka-1,
64                                         -beta, alfa, hrac ,dalsiHrac,nejlepsiTah);
65         }
66     }
67 }
68
69     if (hodnTahAktualni==PRERUSENO) return PRERUSENO;
70
71     vratTah(tah, novaPozice);
72
73     if (minimalizujici)
74         hodnTahAktualni.hodnota=-hodnTahAktualni.hodnota;
75     if (hodnTahAktualni.hodnota > max){
76         max = hodnTahAktualni.hodnota;
77         maxTah = tah;
78     }
79     if (max > alfa){
80         alfa=max;
81         zaznamTyp=PRESNE;
82         zaznamHodnotaUlozeni=max;
83     }
84     if (alfa>=beta){
85         if (minimalizujici) alfa ==-alfa;
86         transpozicniTab.uloz(p,hloubka,BETA ,beta,maxTah
87                               ,hracNaTahu());
88         return new HodnotaTah(alfa, maxTah);
89     }
90 }
91
92 if (minimalizujici) max ==-max;
93 transpozicniTab.uloz(p,hloubka, zaznamTyp,zaznamHodnotaUlozeni,
94                         maxTah, hracNaTahu());
95 return new HodnotaTah(max,maxTah);
96 }

97

98 Tah[] seradTahyTT2(Tah[] tahy,Tah nejlepsiTah){
99     Tah[] serazeneTahy=seradTahy(tahy, heuristika)
100    if (zaznam!=null){
101        foreach tah of serazeneTahy{
102            if (tah== nejlepsiTah){
103                serazeneTahy=vloz(serazeneTahy, 0,tah);
104                break;
105            }
106        }
107    }
108    return serazeneTahy;
109 }

```

### 18.3.2. Popis práce pseudokódu

Zde budou popsány pouze nové nebo změněné příkazy, popis ostatních se nachází v předchozích kapitolách o popisu pseudokódu.

Nepopsané metody:

- Tah[] seradTahyTT2(Tah[] tahy, Tah nejlepsiTah) – Odpovídá funkci metody seradTahyTT(...). Metoda seřadí tahy přijaté v parametru a navíc tah - nejlepsiTah, upřednostní na první místo ve výsledném seznamu tahů, které metoda vrací.

Příklad volání algoritmu:

```
HodnotaTah nejlepsiTah =  
alfabetaImpl(aktualniPozice(), 4, -NEKONECNO, NEKONECNO, 0, 1, [0, 0], null)
```

Vrací nejlepší tah pro hloubku 4. Přičemž tento tah se hledá pro hráče s indexem 1 (hráč s indexem 0 hrál předtím). Aktuální skóre hráčů je nulové. Nejlepší tah seřazení při prvním volání není definován, a proto je nastaven na null.

U tohoto algoritmu rovněž upravujeme znamínka podle čísel hráčů jako u implementace Negamaxu. Navíc zde musíme hlídat správné otáčení mezí alfa a beta při rekurzivním volání na řádcích 60 a 63. Pokud totiž nebudeme přecházet z maximalizující na minimalizujícího hráče (nebo naopak), tak se meze neprevrací, což řeší podmínka a volání na řádcích 56-59. Znaménko měníme zpět kromě řádku 92, také u ořezu na řádku 85, abychom zachovali jeho správnou hodnotu. Co se týče transpozice, zde používáme rozšířenou verzi s opravováním alfa a beta mezi testováním nejlepšího uloženého tahu jako prvního. V implementaci se nepoužily "beta transpoziční opravy", jelikož měly negativní vliv na výsledek celého algoritmu, který pak neodpovídá výsledku Negamaxu. Řazení tahů bylo přepracováno novou metodou SeradTahyTT2, která se od metody SeradTahyTT z kapitoly 9.3.1 liší tím, že má jako parametr přímo Tah místo záznamu transpoziční tabulky. To je důsledkem přesunutí dotazování transpoziční tabulky do cyklu for až za řazení, a tudíž je nutno si předávat nejlepsiTah přes parametr samotné metody alfaBetaImpl.

## 19. Implementace zlepšujících technik algoritmů umělé inteligence

Zatímco ostatní zlepšující techniky se příliš neliší od teoretického popisu, nebo zde není nic implementačně specifického, tak u transpozičních tabulek a hashovací funkce rozdíly jsou.

### 19.1. Detaily implementace transpoziční tabulky

V podkapitole 9.3.4 o hashovaní byl postup generování utvářen na trojrozměrném poli odpovídající souřadnici x, souřadnici y a indexu hráče. Ve skutečnosti je těchto parametrů více, jen díky tomu, že některé hry (např. Terrace) vyžadují více informací o poli, než jen index hráče. Proto budeme potřebovat více dimenzí, konkrétně tyto dimenze jsou:

- *Typ pole* - Pole mimo plochu (*OUT\_OF\_BOARD\_COORDS*), neexistující pole (*NONEXIST\_BOARD\_COORDS*), normální (viz kapitola 15). To znamená, že velikost této dimenze je 3.
- *Souřadnice x pole* – „X“ souřadnice na desce, kde rozsah je dán hrou. Platí pouze pro normální typ pozice, pro ostatní je 0.
- *Souřadnice y pole* – „Y“ souřadnice na desce, kde rozsah je dán hrou. Platí pouze pro normální typ pozice, pro ostatní je 0.
- *Typ hracího prvku* - Typ hracího prvku (*VisualObject*), že všech možných pro danou hru. Velikost dimenze je dána celkovým počtem těchto typů. Např. pro hru Yinch máme tři typy *NullCell* - bezstavové pole, *YinchRing* - kroužek, *YinchMarker* - kámen. Více viz reprezentace hracích prvků kapitola 16.
- *Index hráče* – Index odpovídá číslu hráče na tahu. Velikost odpovídá celkovému počtu hráčů pro danou hru. Více informací, proč se rozlišují hráči při ukládání do tabulky, najdete v implementaci algoritmu Negamax v podkapitole 18.2.
- D dimenzí stavu hracího prvku - Jak bylo popsáno v kapitole 16 o reprezentaci hracích prvků, tak každý prvek lze popsat polem s čísly (*intValues* pole). Každé toto číslo představuje část stavu daného prvku (např. u *PieceTerrace* index 2 představuje, zda kámen je T kámen). Zde tedy každý index tohoto pole představuje dimenzi navíc podle hodnot tohoto prvku. Vzhledem k tomu, že u různých hracích prvků je pole *intValues* jinak dlouhé, bere se počet těchto dimenzi podle největšího z nich. Celkový rozsah každého indexu pole *intValues* je průnikem rozsahu tohoto indexu za každý typ hracího prvku.

Jak vidíme, můžeme mít velký počet dimenzi. S takovýmto mnohorozměrným polem, by se špatně pracovalo, protože Java je paměťově neefektivní při práci s multidimensionálními poli [41]. Proto je použito mapování do jednoho rozměru, resp. nemapuje se, ale už od začátku se pracuje s 1 dimenzí, kde se zjišťuje cíl stejně jako u mapování. To spočívá v tom, že vynásobíme velikosti dimenzi mezi sebou, což nám určí velikost jednodimenzionálního pole, kam postupně ukládáme jednotlivé hodnoty pro každou kombinaci možných hodnot v "mapovaných" dimenzích.

Jinak co se týče samotné transpoziční tabulky, tak v implementaci používáme pole, které indexujeme pomocí 19 bitů (což představuje část 64 bitového hashe, jak bylo popsáno v teoretické části). To znamená, že naše tabulka má velikost 524288 záznamů.

## 20. Repetiční remíza

Ač žádná hra nemá pravidlo, které by zabraňovalo nekonečnému opakování tahů či pozic, může dojít při hře počítače proti počítači k nekonečné smyčce tahů. Aby se tomu předešlo, aplikace obsahuje následující globální pravidlo repetiční remízy:

Pokud se opakuje 5 stejných pozic v rámci jedné hry, hra končí jako (repetiční) remíza. (*Repetition draw*).

## 21. Plugin systém

Pro jednodušší přidávání nových her, seznam her v aplikaci není statický. To se rovněž týká i třeba hracích prvků (třída *VisualObject*) a vůbec všech tříd, kde je společná abstraktní nadtída a jednotlivé hry jí vlastní implementací rozšiřují. Aby takováto funkce byla možná, byl implementován pluginovací systém nacházející se v třídě *ClassOperate*. Tento systém využívá funkce reflexe [42] v Javě. Pokud například hledáme všechny hrací prvky hry Yinsh (což jsou kroužky a kameny), tak napřed specifikujeme požadavky pomocí třídy *Requirements*. To jsou hlavně požadavky pro jakou hru budeme hledat prvky, a jaký by měly mít obecně požadovaný tvar (např. čtvercový apod.). A poté necháme pomocí plugin systému vyhledat třídy, jednak splňující požadavky, a jednak rozšiřující určenou třídu. V našem případě *VisualObject*, což nám zajistí, že opravdu jde o vizuální hrací prvek. Při vyhledávání plugin systém prohledá tzv. Classpath Javy [43] na výskyt třídy se zvolenými požadavky. Pomocí reflexe pak vytvoří instance vhodných tříd a výsledek se nám vrátí. Samozřejmě jsou zde odlišnosti v závislosti, co přesně hledáme, nebo co všechno požadujeme, které si můžeme volit, ale to však představuje jen malé rozdíly od popsaného postupu. Díky reflexi dokonce v extrému můžeme přidávat nové třídy, resp. hry, ve kterých chceme hledat i uprostřed běhu aplikace.

## 22. Skórování her

V aplikaci není jednotné skórování, resp. hodnota vyjadřující, jak blízko je hráč k vítězství. U každé hry je skórování individuální a může být různě definované. Například u dárky by to mohl být počet vyhozených kamenů spoluhráče nebo také počet vlastních kamenů, záleží na definici. Toto skóre by pokud možno mělo mít buď nezvyšující, nebo nesnižující průběh. V jistém smyslu reflektuje heuristickou funkci, protože také popisuje, jak si hráč dobře vede, sice ne v kontextu pozice, ale v kontextu celé hry. Každá hra aplikace má vlastní skórování, které je definováno několika pevnými hodnotami popsanými v následující tabulce

Hra	Minimální skóre	Maximální skóre	Vítězné skóre	Skóre prohry	Výchozí skóre	Maximální změna skóre za tah
<b>Abalone</b>	0	6	6	-	0	1
<b>Terrace</b>	0	80	-	39	80	41
<b>Yinsh</b>	0	3 (blitz: 1)	3 (blitz: 1)	-	0	11

Tab. 4 - Definované skórování pro jednotlivé hry.

Další tabulka popisuje akce, které mění skórování pro hráče v implementovaných hrách

Hra	Změna skóre	Akce
<b>Abalone</b>	1	vytlačení 1 koule soupeře
<b>Terrace</b>	-1	ztráta vlastního kamenu A (nejmenšího)
<b>Terrace</b>	-2	ztráta vlastního kamenu B
<b>Terrace</b>	-3	ztráta vlastního kamenu C
<b>Terrace</b>	-4	ztráta vlastního kamenu D (největšího)
<b>Terrace</b>	-41	ztráta vlastního kamenu T
<b>Yinsh</b>	1	utvoření vlastní řady

Tab. 5 - Akce měnící skórování

## 23. Cache (vyrovnávací paměť)

Aby se urychlil výkon grafického rozhraní je implementovaná vyrovnávací paměť - cache na obrázky. Do cache se ukládají obrázky, se kterými bylo pracováno. Pokud je později nutno použít stejný obrázek, který je v ní uložen, vykreslí se přímo. Tím se šetří IO operace z disku a také se šetří čas nutný na úpravu obrázků (hlavně změna velikosti). V cache jsou obrázky dvou typů - původní a změněné s jinou velikostí. Původní obrázky se do ní načtou při prvním načtení souboru s obrázkem, zatímco změněné jsou až výsledkem po změně velikosti původního obrázku. Cache je limitována celkovým počtem obrázků (standardně 60) a počtem duplicitních obrázků (stejných obrázků s různou velikostí - standardně 4). Pokud tedy nastane situace, že daný obrázek se vyskytuje v cache, ale s jinou velikostí, tak použijeme původní obrázek a změníme mu velikost, jinak použijeme rovnou uložený obrázek. Cache je řízena algoritmem LRU [44], který zaznamenává, které položky cache byly nejméně používané a ty pak při nedostatku místa v cache maže. Vychází tak z předpokladu, že záznam, s kterým se nedávno pracovalo, bude s větší pravděpodobností použit znova.

## 24. Přidání vlastní hry

K aplikaci je možno doprogramovat vlastní hru. Doprogramování vyžaduje znalosti kódu aplikace, a tím může být přidání vlastní hry komplikovanější. Jelikož kód v aplikaci je hodně a je komplexní, tak samotné přidání si zde popíšeme velice stručně.

Základ tvoří vytvoření vlastních potomků od tříd *AbstractGame*, *AbstractBoard*, *AbstractGamePanelExtension*. V těchto třídách je nutné dopsat povinné metody (abstraktní) a rovněž přetížit některé další metody podle chování, kterého chceme dosáhnout. Do projektu je nutné začlenit obrázky pro danou hru. Pravděpodobně pokud chceme mít nové hrací prvky, je nutné vytvořit vlastní potomky *VisualObject*, kde nadefinujeme jejich stav a vzhled. Pro funkčnost umělé inteligence potřebujeme mít generátor tahů, jehož dopsání je součástí přetížení *AbstractGame*, a heuristiku pro danou hru, což představuje implementaci podle rozhraní *Heuristicable*. Pokud nastane případ, že hra bude příliš specifická mimo rozsah naprogramovaných funkcí, pak je nutné poupravit i tyto metody.

## 25. Parametry příkazové řádky

Aplikace využívá parametry příkazové řádky na nastavení méně využívaných funkcí. Většina z možností se týká testování a nebyly určeny pro uživatele, a proto při jejich zapnutí může aplikace vykazovat chyby. Následující seznam popisuje jednotlivé možnosti:

(„[]“ popisují skupinu možností, z nichž jedna je povinná, „{}“ popisuje povinnou celočíselnou hodnotu, „()“ popisuje výchozí hodnotu)

-*help* - () - Vyvolá nápovědu.

-*statistics=[true,false]* - (*statistics=false*) - Do podadresáře „/log“ začne vypisovat statistické údaje o tazích.

-*debugglobal=[true,false]* - (*debugglobal=false*) - Zapne/vypne globální ladění, aplikace začne vypisovat různé ladící výpisy, ad.

-*debugcache=[true,false]* - (*debugcache=false*) - Zapne/vypne ladění cache paměti obrázků.

-*cachesize={hodnota}* - (*cachesize=60*) - Nastavením se určí počet obrázků, který se vejde do cache.

-*cacheduplicate={hodnota}* - (*cacheduplicate=4*) - Nastaví maximální počet duplicitních obrázků v cache (vzájemně pak mají různé velikosti).

-*cacheenable=[true,false]* - (*cacheenable=true*) - Zapne/vypne cache pro obrázky (vyrovnávací paměť).

-*console* -() - Zobrazí po startu grafické okno vypisující informace do konzole. Tuto možnost lze později vyvolat z GUI.

-*consolelog =[true,false]* - (*consolelog=false*) - Explicitně zapne/vypne ladící výpisy. Tato volba dokáže potlačit ladící výpisy, i když byla nastavena volba *debugglobal=true*.

-*nocatcher* -() - Vypne odchytávání výjimek jazyku Java. Tím potom nebude zobrazen grafický dialog výjimky.

-*debugai=[true,false]* - (*debugai=false*) - Zapne/vypne ladění umělé inteligence. Při ladění jsou veškeré informace z průchodu herního stromu ukládány a na konci jsou zobrazeny. Při této volbě není doporučeno prohledávat do hloubky větší než 3, protože pak ukládané informace zaplní veškerou paměť, čímž dochází k výraznému snížení rychlosti aplikace.

-*aiid=[true,false]* - (*aiid=true*) - Zapne/vypne iterativní prohlubování. Pokud je vypnuto iterativní prohlubování tak je rovněž neaktivní časový limit na dokončení tahu.

-*aitranspositions=[true,false]* - (*aitranspositions=true*) - Zapne/vypne transpoziční tabulky v umělé inteligenci.

-*aimoveordering=[true,false]* - (*aimoveordering=true*) - Zapne/vypne řazení tahů pro umělou inteligenci. V algoritmech a heuristikách, kde není řazení používáno, se změna neprojeví.

## Část IV - Testování umělé inteligence

### 26. Testovací sestava

Obecně měření času je relativní a je ovlivněno mnoha faktory. Je ovlivněno, jak samotným počítáčem, tak rychlostí samotného jazyka (zde Javy), operačním systémem s okolními programy a také prostředky, jakými je čas měřen. Kvůli relativitě měření času bude kladen důraz při testování na parametry, které nejsou ovlivněny časem, pokud to bude možné. V každém případě, aby časové údaje měly svůj kontext, je následně uvedena konfigurace, na které bylo prováděno testování.

MS Windows XP SP2 32bit

Intel Core 2 Duo E6600 @ 3,3 Ghz (2 jádra)

2048 MB RAM DDR2 @ 950 Mhz

Java SE 1.6 Update 17

### 27. Porovnání výkonnosti heuristických funkcí

Toto měření bylo prováděno, ke zjištění, která heuristická funkce poskytuje nejlepší výsledky. Pro eliminaci lidského faktoru, hrály jednotlivé heuristické funkce proti sobě jako počítačový hráč. Pro odstínění vlivu ostatních parametrů, jsme zvolili stejný algoritmus a stejné nastavení hloubky a času na prohledávání. Vždy v rámci každé hry se začínalo ze stejné - dané pozice. Každá kombinace heuristik se utkala vždy jednou. Opakování není nutné, jelikož veškeré algoritmy neobsahují žádný náhodný element. Ovlivnění nestabilním vyhledáváním při použití transpozičních tabulky (viz podkapitola 9.3) bylo eliminováno vždy jejich vyčištěním mezi jednotlivými hrami (resp. znova spuštění aplikace).

U každého tahu byl omezen čas na jeho provedení s použitím iterativního prohlubování. Tato technika odliší výsledky jednotlivých heuristických funkcí podle jejich rychlosti v limitovaném čase. To takovým způsobem, že rychlejší funkce stihne prohledat herní strom do větší hloubky, zatímco pomalejší do hloubky menší, naopak ale může získat výhodu díky přesnějšímu ohodnocení, na které má více času.

U každé hry byly měřeny následující parametry:

- *Vzájemné skóre* - Dosažené skóre každého hráče na konci hry (viz kapitola skórování 22).
- *Bodové ohodnocení výsledku hry* - Vychází přímo ze skóre. Pokud daná heuristika vyhrála hru, získává 10 bodů, při remíze 5 bodů a při prohře 0 bodů. V případě repetiční remízy je výsledek brán podle dosaženého skóre. "Vítěz" u repetiční remízy je ten, kdo má větší skóre. Vítěz pak obdrží 6 bodů, při stejném skóre oba hráči získají 4 body a při prohře 2 body.
- *Parametry hry* - Následující parametry jsou průměrné v rámci dané heuristiky ze všech tahů ve všech vzájemných soubojích u dané hry.

- *Průměrná hloubka*
- *Průměrný faktor větvení* - Faktor větvení hry v aplikaci ovlivněný zlepšujícími technikami algoritmů a určený podle vzorce:

$$\begin{aligned} & \text{průměrný faktor větvení} \\ & = \frac{\text{průměrná hloubka}}{\sqrt{\text{průměrný počet prozk. stavů}}} \end{aligned} \quad (19)$$

- *Průměrný počet prozkoumaných stavů*
- *Časová náročnost ohodnocení k nejnáročnější heuristice* - Porovnává, kolik relativně času jednotlivé heuristiky potřebují na ohodnocení stavu. Nejnáročnější heuristika má vždy náročnost 100%, ostatní jsou vztahovány k ní.

V následujících tabulkách můžeme vidět výsledky měření, přičemž jednotlivé parametry pro měření jsou uvedeny u příslušných tabulek. Všimněme si např., že u hry Abalone se jako výchozí pozice nepoužívá standardní pozice, ale pozice Belgian Diasy, která nevede k tolka repetičním remízám a zablokováním celé hry.

## 27.1. Výsledky u hry Abalone

Skóre		Heuristická funkce soupeře					Celkem
Heur. Funkce	Standard	Experimental	Mass-Based	Simple	Analyzing		
<b>Standard</b>		6	4	6	4	<b>20</b>	
<b>Experimental</b>	0		6	6	4	<b>16</b>	
<b>Mass-Based</b>	6	3		6	3	<b>18</b>	
<b>Simple</b>	1	1	1		1	<b>4</b>	
<b>Analyzing</b>	6	6	6	6		<b>24</b>	

Tab. 6 - Skóre získané v jednotlivých vzájemných hrách heuristik u Abalone. Parametry obou soupeřících hráčů: Abalone - Belgian Diasy 2 Players - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5 s

Body		Heuristická funkce soupeře					Celkem
Heur. Funkce	Standard	Experimental	Mass-Based	Simple	Analyzing		
<b>Standard</b>		10	0	10	0	<b>20</b>	
<b>Experimental</b>	0		10	10	0	<b>20</b>	
<b>Mass-Based</b>	10	0		10	0	<b>20</b>	
<b>Simple</b>	0	0	0		0	<b>0</b>	
<b>Analyzing</b>	10	10	10	10		<b>40</b>	

Tab. 7 - Body udělené za jednotlivé vzájemné hry heuristik u Abalone. Parametry obou soupeřících hráčů: Abalone - Belgian Diasy 2 Players - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5 s

Heuristika	Průměrná hloubka	Průměrný faktor větvení	Průměrný počet prozkoumaných stavů	Čas. náročnost ohodnocení k nejnáročnější heur. [%]
<b>Standard</b>	4,43	7,76	8680,29	76,17
<b>Experimental</b>	4,82	8,12	24121,53	100,00
<b>Mass-Based</b>	4,80	8,30	25986,01	58,83
<b>Simple</b>	5,25	7,57	41398,11	3,12
<b>Analyzing</b>	4,76	8,32	24035,25	59,53

Tab. 8 - Průměrné parametry naměřené během všech vzájemných her heuristik. Parametry obou soupeřících hráčů: Abalone - Belgian Diasy 2 Players - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5 s

Z tabulky skóre hry *Abalone* vyplývá, že nejlepší heuristická funkce je *Analyzing*, která vyhrála ve všech případech. Pravděpodobně díky většímu počtu sledovaných vlastností, se jí podařilo překonat stejně rychlou heuristikou *Mass-Based*, které se pravě nezdařilo překonat komplexnější heuristiky jako *Analyzing* nebo *Experimental*. Heuristika *Experimental* byla díky velkému počtu hodnocených vlastností nadějný kandidát na nejlepší heuristiku, ale skončila na předposledním místě. Ze všech heuristik byla časově nejnáročnější, což není její problém, jelikož její průměrná hloubka v daném čase nebyla větší než u heuristiky *Analyzing*. Spíše půjde tedy o špatné nastavení vah a vlastností. Zajímavá je heuristika *Simple*, jelikož díky její jednoduchosti dokázala prohledat herní strom s větší hloubkou. Bohužel, její neschopnost všimnout si některých důležitých vlastností jako seskupování či orientaci na střed, jí odsoudila na poslední místo bez jediného zisku bodu. Průměrný počet stavů jednotlivých heuristik odpovídá hloubce, která byla v časovém limitu prohledána, jejíž souhrnnou hodnotu představuje průměrná hloubka.

## 27.2. Výsledky u hry Terrace

Skóre		Heuristická funkce soupeř		
Heur. Funkce		Simple	Improved	Fast
Simple		78	0	<b>78</b>
Improved	63		0	<b>63</b>
Fast	71	67		<b>138</b>

Tab. 9 - Skóre získané v jednotlivých vzájemných hrách heuristik u Terrace. Parametry obou soupeřících hráčů: Terrace - Default 2 Players - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5 s.

Body		Heuristická funkce soupeř		
Heur. Funkce		Simple	Improved	Fast
Simple		6	0	<b>6</b>
Improved	2		0	<b>2</b>
Fast	10	10		<b>20</b>

Tab. 10 - Body udělené za jednotlivé vzájemné hry heuristik u Terrace. Parametry obou soupeřících hráčů: Terrace - Default 2 Players - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5 s.

Heuristika	Průměrná hloubka	Průměrný faktor větvení	Průměrný počet prozkoumaných stavů	Čas. náročnost ohodnocení k nejnáročnější heur. [%]
Simple	4,57	8,97	22546,19	56,04
Improved	4,54	9,21	23647,85	100,00
Fast	4,96	7,67	24407,48	52,30

Tab. 11 - Průměrné parametry naměřené během všech vzájemných her heuristik u Terrace. Parametry obou soupeřících hráčů: Terrace - Default 2 Players - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5s.

Na rozdíl od hry Abalone zvítězila heuristika orientovaná na rychlosť, *Fast*. Té se podařilo snížit nejvíce průměrný faktor větvení a zvýšit nejvíce průměrnou hloubku. Ostatní heuristiky, ač se moc neliší ve vlastnostech, s heuristikou *Fast* prohrály. Mezi sebou tyto heuristiky - *Improved* a *Standard* vedly vyrovnanou hru, kterou ukončila repetiční remíza ve prospěch heuristiky *Improved*, jíž se podařilo sebrat několik cenných kamenů soupeře. Podle časové náročnosti heuristik si můžeme všimnout, že mobilita kamenů, kterou vykonává heuristika *Improved* navíc oproti *Fast*, zabírá téměř o polovinu času navíc.

### 27.3. Výsledky u hry Yinsh

Skóre		Heuristická funkce soupeř		
Heur. Funkce		Basic	Storm	By Positions
Basic		0	1	1
Storm	3		1	4
By Positions	3	3		6

Tab. 12 - Skóre získané v jednotlivých vzájemných hrách heuristik u Yinsh.  
Parametry obou soupeřících hráčů: Yinsh - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5 s.

Body		Heuristická funkce soupeř		
Heur. Funkce		Basic	Storm	By Positions
Basic		0	0	0
Storm	10		0	10
By Positions	10	10		20

Tab. 13 - Body udělené za jednotlivé vzájemné hry heuristik u Yinsh. Parametry obou soupeřících hráčů: Yinsh - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5 s.

Heuristika	Průměrná hloubka	Průměrný faktor větvení	Průměrný počet prozkoumaných stavů	Čas. náročnost ohodnocení k nejnáročnější heur. [%]
Basic	4,14	10,46	16521,13	14,08
Storm	4,20	9,11	10786,11	37,02
By Positions	4,17	8,45	7383,43	100,00

Tab. 14 - Průměrné parametry naměřené během všech vzájemných her heuristik u Yinsh. Parametry obou soupeřících hráčů: Yinsh - Alpha-Beta + MO + TT, Unlimited Depth, Max Time 5 s.

U hry Yinsh jsme byli svědky, jak jediná, ač komplexní, vlastnost může být lepší než více jednoduchých (časově) vlastností. Právě heuristice *By Positions* se podařilo díky tomu vyhrát všechny partie. Nevýhodou této heuristiky, však byla rychlosť, která byla téměř 3x nižší než druhá nejnáročnější heuristika *Storm*. Tato nevýhoda se neprojevila tak výrazně, protože ostatní heuristiky nebyly rychlé natolik, aby v limitu stihly dokončit prohledávání větší hloubky

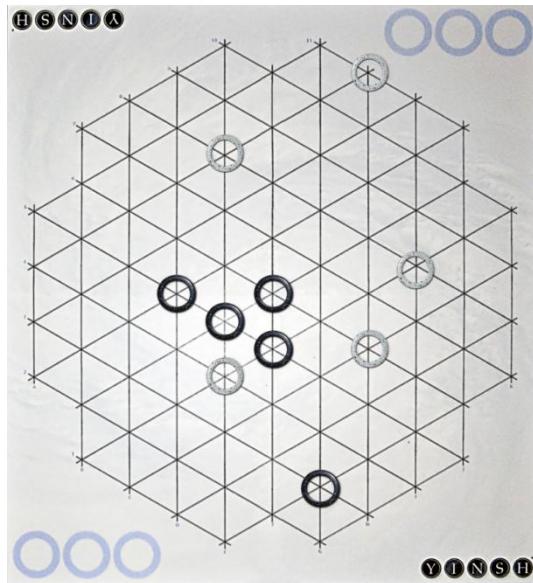
rozsáhlého herního stromu. V souboru heuristik *Storm* a *Simple* se ukázala síla řazení tahů. Heuristika *Simple* totiž toto řazení tahů neobsahovala. I když byla 3x rychlejší, tak jí tento čas navíc nestačil na vyrovnaní zvýšeného počtu prohledaných stavů vlivem většího průměrného faktoru větvění.

## 28. Měření efektivity algoritmů umělé inteligence a jejich zlepšujících technik

Abychom zjistili, jak dobře jednotlivé algoritmy a jejich vylepšení prohledávají herní strom, bylo provedeno měření jejich efektivity. Toto měření spočívalo v měření počtu prohledaných pozic pro jednotlivé algoritmy a jejich hloubky. Algoritmus *Best Move* nebyl zahrnut do měření, protože jak již bylo řečeno, jde o shodný algoritmus jako *Negamax* na hloubce 1. Počty těchto pozic byly měřeny na začátku, uprostřed hry a průměrný počet pozic byl měřen za celou hru. Každé toto měření bylo prováděno pro každou implementovanou hru. Aby jednotlivé hry probíhaly stejně, hráli vždy dva počítačoví hráči proti sobě. Protože se jedná o měření algoritmů, byla zvolena jedna heuristika a stejná výchozí konfigurace. Jednotlivé hloubky byly určovány pomocí iterativního prohlubování. Jednotlivé nastavení parametrů hry pro soupeře, je vždy uvedeno pod danou tabulkou.

### 28.1. Měření na začátku hry

V tomto měření se vždy měřil počet prozkoumaných stavů v prvním tahu z pozice uvedené pod příslušnou tabulkou s výsledky. U hry *Yinsh*, kde nejsou startovní pozice, byla zvolena konfigurace kroužku vyobrazená na obrázku 40.



Obr. 40 - Základní pozice položení kroužků u hry *Yinsh* pro měření.

V následujících tabulkách jsou počty prozkoumaných stavů pro jednotlivé hloubky a algoritmy pro každou hru:

Algoritmus / hloubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	52	52	52	52	52
2	2692	2688	568	192	192
3	149322	144911	11544	3295	3262
4	8033300	7612405	54477	13275	13166

Tab. 15 - Počet prozkoumaných stavů k získání výsledného tahu na začátku hry. Parametry hráče, který hraje první ze startovní pozice: *Abalone - Belgian Diasy 2 Player - Unlimited Time - Standard* ( soupeř: stejný algoritmus AI - Unlimited Time - Experimental ).

Algoritmus / hloubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	50	50	50	50	50
2	2450	2450	791	650	650
3	127166	123932	18960	15649	15583
4	6626315	6552173	205236	170246	160231

Tab. 16 - Počet prozkoumaných stavů k získání výsledného tahu na začátku hry. Parametry hráče, který hraje první ze startovní pozice: *Terrace - Default 2 Player - Unlimited Time - Simple* ( soupeř: stejný algoritmus AI - Unlimited Time - Improved ).

Algoritmus / hloubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	72	72	72	72	72
2	4864	4864	485	485	485
3	326327	324291	17019	16107	16028
4	21848604	21411573	99698	88195	82977

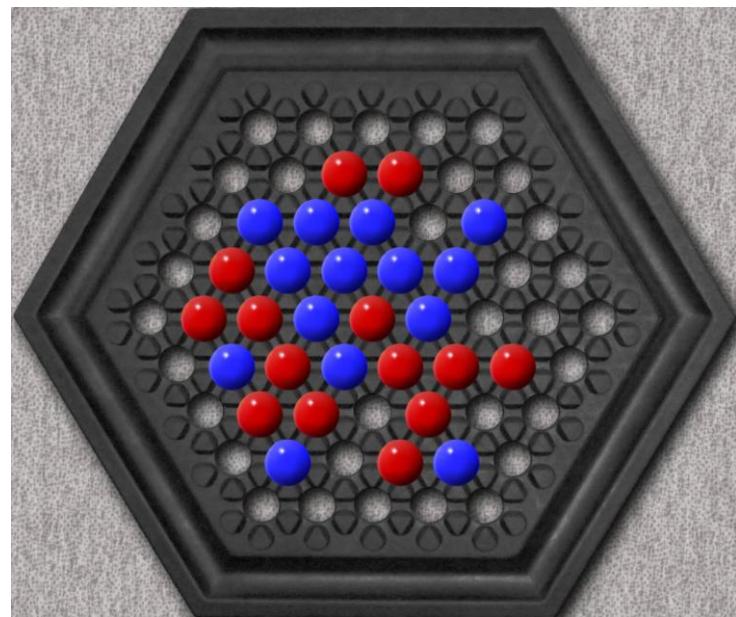
Tab. 17 - Počet prozkoumaných stavů k získání výsledného tahu na začátku hry. Parametry hráče, který hraje první ze startovní pozice: *Yinsh - Unlimited Time - Storm* ( soupeř: stejný algoritmus AI - Unlimited Time - By Positions ).

Globálně můžeme z tabulek pozorovat, že pořadí podle počtu jednotlivých stavů odpovídá teoretické části, resp. Alfa-beta prořezávání je efektivnější než Negamax a použití jednotlivých technik vylepšuje efektivitu. Čím je efektivnější algoritmus, tím prohledá méně stavů na získání stejné výsledné pozice. Podle očekávání se nasazení Alfa-Beta prořezávání projevilo výrazným snížením počtu stavů. To zhruba odpovídá průměrnému snížení stavů na pětinu u hry Abalone, na třetinu u Terrace a dokonce na sedminu u hry Yinsh na každou nižší hloubku než jedna. U větších hloubek se tento rozdíl samozřejmě zvyšuje. Zlepšení algoritmu Alfa-Beta řazením tahů odpovídá kvalitě implementace řazení tahů v dané heuristice, proto může být různé. U hry Abalone můžeme pozorovat zhruba 3 násobně menší počet stavů díky vhodné zvolenému řazení. Transpoziční tabulka se neprojevila tak výrazně, jak se očekávalo. V nižších hloubkách je její význam zanedbatelný. Avšak ve větších hloubkách (3 a více) je dostatečný počet uspořených stavů, aby se tabulka uplatnila. Toto zlepšení není tak razantní jako v případě řazení tahů. Transpoziční tabulka se projevovala lépe v kombinaci s méně

efektivním algoritmem Negamax, kde dokázala uspořit 3 - 6 % prohledávaných stavů v závislosti na hře.

## 28.2. Měření uprostřed hry

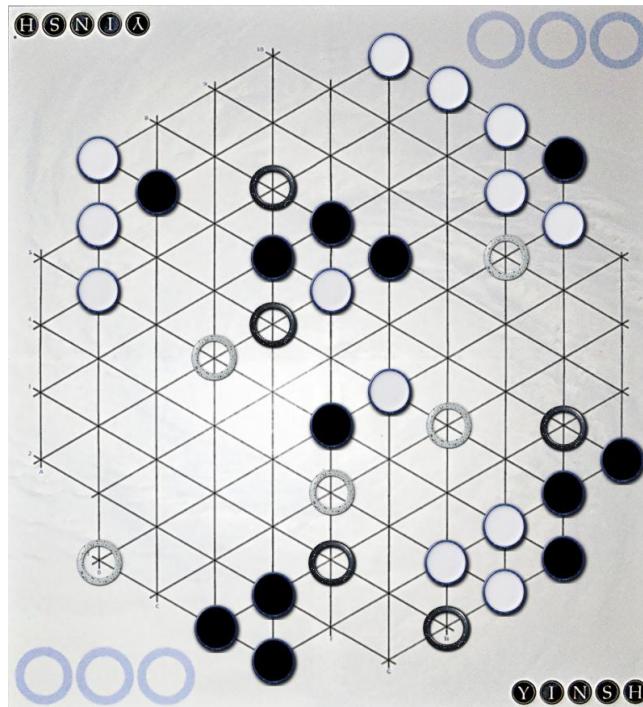
Měření bylo prováděno z pevně určených pozic, které odpovídají konfiguracím stavu desky uprostřed hry. Jednotlivé určené pozice jsou vidět na obrázcích 41, 42 a 43.



Obr. 41 - Výchozí rozložení kamenů u hry Abalone pro měření uprostřed hry.



Obr. 42 - Výchozí rozložení kamenů u hry Terrace pro měření uprostřed hry.



Obr. 43 - Výchozí rozpoložení u hry Yinsh pro měření uprostřed hry.

V následujících tabulkách jsou počty prozkoumaných stavů pro jednotlivé hloubky a algoritmy pro každou hru:

Algoritmus / hloubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	58	58	58	58	58
2	3357	3343	705	399	399
3	197383	191784	11693	6573	6502
4	11616799	10955776	127386	42286	40966

Tab. 18 - Počet prozkoumaných stavů k získání výsledného tahu uprostřed hry. Parametry hráče, který hraje první z vyobrazené pozice uprostřed hry: *Abalone - Unlimited Time - Standard* ( soupeř: stejný algoritmus AI - Unlimited Time - Experimental ).

Algoritmus / hloubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	66	66	66	66	66
2	4153	4153	981	977	977
3	272982	268234	26245	18632	18579
4	18384137	17447871	547279	125160	114673

Tab. 19 - Počet prozkoumaných stavů k získání výsledného tahu uprostřed hry. Parametry hráče, který hraje první z vyobrazené pozice uprostřed hry: *Terrace - Unlimited Time - Simple* ( soupeř: stejný algoritmus AI - Unlimited Time - Improved ).

Algoritmus / hloubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	61	61	61	61	61
2	2385	2385	354	538	538
3	128203	120583	9203	8675	8619
4	5866390	4880669	48449	59243	56778

Tab. 20 - Počet prozkoumaných stavů k získání výsledného tahu uprostřed hry. Parametry hráče, který hraje první z vyobrazené pozice uprostřed hry: *Yinsh - Unlimited Time - Storm* ( soupeř: *stejný algoritmus AI - Unlimited Time - By Positions* ).

V tomto měření opět můžeme pozorovat výraznou úsporu stavů při použití Alfa-Beta ořezávání. Podle tabulek si můžeme všimnout, že tyto stavy desky jsou složitější a vyžadují prohledat více stavů, což znamená, že se zvětšil faktor větvení. To platí s výjimkou hry Yinsh, kde větší počet kamenů na desce snižuje faktor větvení z toho důvodu, že vždy můžeme táhnout pouze kroužky, kterých je stejný počet, ale cílových polí je méně díky většímu počtu kamenů na desce. Opět se zde také potvrzuje menší význam použití transpoziční tabulky než techniky řazení tahů u algoritmu Alfa-Beta. Jinak si můžeme z tabulek také všimnout, že algoritmus Alfa-Beta proti Negamaxu, bez vylepšujících technik, prohledá stejný počet stavů s hloubkou o 0.9 - 1.4 menší (podle vzorce

$$4 - \log_{faktor\ větvení\ Negamaxu}(počet\ stavů\ alphabeta\ v\ hlouce\ 4) \quad (20)$$

) v závislosti na dané hře. Takové zlepšení hloubky podle teoretické části odpovídá průměrnému případu pro algoritmus Alfa - Beta.

### 28.3. Měření za celou hru

Měření bylo prováděno, tak že byla odehrána celá hra až do konce. Jednotlivé následující tabulky pak odpovídají průměrným hodnotám ze všech tahů hráče, který hrál s parametry uvedenými vždy pod tabulkou. Jednotlivé hry začínaly se stejných pozic jako u měření na začátku hry.

Algoritmus / hloubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	64,41	62,48	77,98	71,39	70,13
2	4172,82	3470,78	771,06	191,47	187,95
3	271034,41	210584,00	32352,50	5917,75	5742,65
4	17616559,18	11472300,44	95289,56	13507,36	15042,45

Tab. 21 - Průměrný počet prozkoumaných stavů k získání výsledného tahu v průběhu celé hry. Parametry prvního hráče, jehož stavy jsou měřeny: *Abalone - Belgian Diasy 2 Player - Unlimited Time - Standard* ( soupeř: *stejný algoritmus AI - Unlimited Time - Experimental* ).

Algoritmus / hłoubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	63,72	55,03	58,40	64,62	60,39
2	3335,00	3074,90	612,40	363,52	362,42
3	212497,06	166142,66	20746,87	9290,83	9182,24
4	12804605,07	8840400,84	380313,67	33268,97	31698,33

Tab. 22 - Průměrný počet prozkoumaných stavů k získání výsledného tahu v průběhu celé hry.

Parametry prvního hráče, jehož stavy jsou měřeny: *Terrace - Default 2 Player - Unlimited Time - Simple* ( soupeř: *stejný algoritmus AI - Unlimited Time - Improved* ).

Algoritmus / hłoubka	Negamax	Negamax + TT	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT
1	44,67	40,50	42,73	33,13	33,02
2	1845,05	1744,73	365,18	235,68	229,68
3	93625,09	81419,12	8358,73	4483,90	4334,00
4	4940165,37	3483516,85	59960,45	32875,23	30134,74

Tab. 23 - Průměrný počet prozkoumaných stavů k získání výsledného tahu v průběhu celé hry.

Parametry prvního hráče, jehož stavy jsou měřeny: *Yinsh - Unlimited Time - Storm* ( soupeř: *stejný algoritmus AI - Unlimited Time - By Positions* ).

I zde můžeme pozorovat podobné úkazy jako u měření na začátku a uprostřed hry. Největší rozdíly jsou vidět ve snížení počtu stavů vlivem transpoziční tabulky u Negamaxu, které je kolem 30 % oproti předchozím měřením, kde toto snížení dosahovalo 6 %. To je proto, že účinek transpoziční tabulky vzniká s jejím větším zaplněním v pozdějších tazích. Také můžeme pozorovat větší účinnost řazení tahů v průměru oproti jednotlivým pozicím v předchozích měřeních.

## 29. Porovnání výkonnosti algoritmů umělé inteligence a jejich zlepšujících technik

Abychom ověřili, zda doopravdy snížení počtu uzlů pro daný algoritmus vede k lepší hře a potažmo k výhře, nechali jsme hrát jednotlivé algoritmy proti sobě. Souboje probíhaly za situace, kdy byla zvolena stejná hra, heuristika a startovní pozice. Tím zajistíme, že se nebudou do měření promítat rozdíly v těchto parametrech. Díky tomu, že vyhledávání pokaždé probíhá stejně (viz kapitola 27), nám stačilo, aby se každý algoritmus umělé inteligence utkal s ostatními právě jednou.

U některých algoritmů byly měřeny partie s různým časovým limitem, abychom zjistili, zda čas navíc napomáhá úspěšnosti daného algoritmu. Měřené souboje často končily repetiční remízou (výsledky s repetiční remízou jsou vyznačeny červeně) a to hlavně z důvodu malého rozdílu mezi bojujícími algoritmy (zvláště pokud byly stejně rychlé). Výsledky těchto měření ukazují následující tabulky. U každé tabulky jsou vždy uvedeny parametry měření.

Vysvětlení jednotlivých tabulek:

- Skóre* - Výsledné skóre hry pro daného hráče (viz skórování kapitola 22).
- Body* - Bodové ohodnocení pro daného hráče. Je udělováno stejně jako v kapitole 27.
- Průměrná hloubka* - Průměrná hloubka ze hry za každý tah hráče, kterou algoritmus daného hráče stihl prohledat v časovém limitu.

Skóre		Heuristická funkce soupeř							
Čas		5 s	10 s	5 s	5 s	5 s	10 s		
	Algoritmus	Negamax + TT	Negamax + TT2	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT	Negamax	Celkem	
5 s	<b>Negamax + TT</b>		4	0	0	0	1	<b>5</b>	
10 s	<b>Negamax + TT</b>	6		3	0	1	3	<b>13</b>	
5 s	<b>Alpha-Beta</b>	5	6		0	0	5	<b>16</b>	
5 s	<b>Alpha-beta + MO</b>	6	6	4		0	5	<b>21</b>	
5 s	<b>Alpha-beta + MO + TT</b>	6	6	0	0		6	<b>18</b>	
10 s	<b>Negamax</b>	6	5	4	3	4		<b>22</b>	

Tab. 24 - Skóre získané v jednotlivých vzájemných hrách různých algoritmů. Červené jsou označeny partie ukončené repetiční remízou. Parametry obou soupeřících hráčů: *Abalone - Belgian Diasy 2 Player - Experimental - Unlimited depth*.

Body		Heuristická funkce soupeř							
Čas		5 s	10 s	5 s	5 s	5 s	10 s		
	Algoritmus	Negamax + TT	Negamax + TT2	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT	Negamax	Celkem	
5 s	<b>Negamax + TT</b>		0	2	0	0	0	<b>2</b>	
10 s	<b>Negamax + TT</b>	10		0	0	0	2	<b>12</b>	
5 s	<b>Alpha-Beta</b>	6	10		2	4	6	<b>28</b>	
5 s	<b>Alpha-beta + MO</b>	10	10	6		4	6	<b>36</b>	
5 s	<b>Alpha-beta + MO + TT</b>	10	10	4	4		10	<b>38</b>	
10 s	<b>Negamax</b>	10	6	2	2	0		<b>20</b>	

Tab. 25 - Body udelené v jednotlivých vzájemných hrách různých algoritmů. Červené jsou označeny partie ukončené repetiční remízou. Parametry obou soupeřících hráčů: *Abalone - Belgian Diasy 2 Player - Experimental - Unlimited depth*.

Průměrná hloubka		Heuristická funkce soupeř							
Čas		5 s	10 s	5 s	5 s	5 s	10 s		
	Algoritmus	Negamax + TT	Negamax + TT2	Alpha-Beta	Alpha-beta + MO	Alpha-beta + MO + TT	Negamax	Celkem	
5 s	<b>Negamax + TT</b>		3,889	3,700	3,720	3,467	3,700	<b>18,476</b>	
10 s	<b>Negamax + TT</b>	4,086		4,000	4,000	4,000	4,000	<b>20,086</b>	
5 s	<b>Alpha-Beta</b>	4,600	4,258		4,053	4,048	4,743	<b>21,701</b>	
5 s	<b>Alpha-beta + MO</b>	4,800	4,821	4,789		4,882	4,938	<b>24,230</b>	
5 s	<b>Alpha-beta + MO + TT</b>	5,000	5,000	4,857	5,000		5,016	<b>24,873</b>	
10 s	<b>Negamax</b>	4,000	4,000	4,000	4,000	4,000		<b>20,000</b>	

Tab. 26 - Naměřená průměrná hloubka tahů v rámci jednotlivých vzájemných her různých algoritmů. Parametry obou soupeřících hráčů: *Abalone - Belgian Diasy 2 Player - Experimental - Unlimited depth*.

Podle očekávání zvítězil algoritmus Alfa - Beta s použitím transpoziční tabulky a řazením tahů. Vyhrál hlavně díky bezchybnému výsledku proti různým nastavením algoritmu Negamax. Nejvíce mu dělal starost algoritmus Negamax bez transpoziční tabulky s dvojnásobným časem, kde podle skóre musely být síly relativně vyvážené, jelikož z předchozího měření v tabulce 21 se ukazuje, že Alfa-Beta v tomto případě prozkoumá stejný počet stavů s o 43% větší prohledávanou hloubkou než Negamax. V tomto případě je tato hodnota téměř vyvážená dvojnásobným časovým limitem pro Negamax.

Dále z těchto tabulek můžeme vypozorovat, že obecně hráči s algoritmem Alfa-Beta poráželi hráče s algoritmem Negamax nezávisle na parametrech. Avšak hry, kdy tito hráči hráli proti sobě, končily repetičními remízami. Výsledné skóre těchto remíz ovlivňovaly pak rozdíly v prohledaných hloubkách. Překvapivě nečekaně dopadl algoritmus Negamax s transpozičními tabulkami s časem 10 s, který sice porazil sám sebe s časovým limitem 5 s, ale prohrál s veškerými algoritmy Alfa-Bety, na rozdíl od verze bez transpozičních tabulek, která s nimi sehrála alespoň transpoziční remízy. Při sledování tabulky s průměrnou hloubkou si můžeme všimnout, že celkové výsledky prohledávaných hloubek odpovídají celkovým výsledkům z tabulky ukazující výsledky bodů. Je tedy zřejmé, že algoritmus Alfa-Beta poskytuje výrazně lepší výsledky. Co se týče zlepšujících technik, tak řazení tahů i transpoziční tabulky mají také vliv na zlepšení výsledku hry, ačkoli ne příliš výrazný (hlavně u transpozičních tabulek).

## Část V - Závěr

### 30. Zhodnocení

Na začátku práce byly zvoleny abstraktní hry Abalone, Terrace a Yinsh. Byla zkoumána jejich klasifikace v rámci teorie her a rovněž jejich vlastnosti a zákonitosti.

Tyto hry byly implementovány do grafického uživatelského rozhraní, které bylo podstatnou součástí této práce. Uživatelské rozhraní bylo navrženo obecně tak, aby bylo pro každou hru jednotné a umožňovalo vytvoření dalších her do budoucna. Hry v tomto rozhraní lze hrát a testovat v libovolné konfiguraci lidských/počítačových hráčů v rámci pravidel každé hry.

Principy počítačového hraní strategických her a umělé inteligence byly popsány v teoretické části práce. Tato část se zabývala obecným principem vyhledávacích algoritmů přes herní stromy, algoritmy Minimax, Negamax a algoritmem Alfa-Beta prořezávání. Také zde byly popsány vylepšující techniky používané u těchto algoritmů, které jsou iterativní prohlubování, řazení tahů, transpoziční tabulky se Zobrist hashováním.

Nasazení těchto algoritmů a technik v aplikaci bylo popsáno v implementační části práce. Velký díl této části byl věnován popisu různých heuristických funkcí pro jednotlivé hry a jejich vlastnosti. V této části jsou rovněž popsány implementační detaily aplikace.

Završením této práce bylo testování jednotlivých algoritmů, zlepšujících technik a heuristických funkcí proti sobě. Zde se ukázalo, že Alfa-Beta prořezávání poráží algoritmus Negamax díky menšímu množství pozic nutných k prohledání. Všechny zlepšující techniky mají kladný význam na kvalitu výsledného prohledávání. Nejvíce se tento vliv projevuje u řazení tahů a použití transpoziční tabulky na algoritmus Negamax. U vzájemného měření heuristických funkcí u hry Abalone dosahuje nejlepších výsledků funkce *Analyzing*, která zvítězila díky velkému počtu zkoumaných parametrů, aniž by byla nejpomalejší. U hry Terrace zvítězila funkce *Fast*, která hodnotila nejméně parametrů, díky čemuž byla nejrychlejší a dokázala prozkoumat nejvíce tahů dopředu. U poslední hry Yinsh zvítězila komplexní funkce *By Position*, která dokázala najít velmi kvalitní tahy navzdory jejímu trojnásobně pomalejšímu prohledávání oproti druhé nejpomalejší funkci *Storm*.

Aplikace poskytuje další alternativu pro hráče těchto nových a méně známých her s možností soupeřit a experimentovat s umělou inteligencí.

## 31. Náměty pro budoucí práci

Následující návrhy poskytují možnost, jak v budoucnu vylepšit tuto diplomovou práci, resp. aplikaci:

- Použití dalších zlepšujících technik umělé inteligence – MTD(f), Furtility pruning, hledání klidu, aj.
- Prozkoumání možnosti paralelizace algoritmů pro umělou inteligenci.
- Možnost hry po internetu.
- Naimplementování více her do aplikace.
- Lokalizace do češtiny, příp. dalších jazyků.

# Reference

## Internetové odkazy

- [1] BoardGameGeek  
<http://www.boardgamegeek.com/>
- [2] BoardGameGeek – Game Categories  
<http://www.boardgamegeek.com/browse/boardgamecategory>
- [3] BoardGameGeek – Game Mechanics  
<http://www.boardgamegeek.com/browse/boardgamenemechanic>
- [4] Desková hra  
[http://cs.wikipedia.org/wiki/Deskov%C3%A1\\_hra](http://cs.wikipedia.org/wiki/Deskov%C3%A1_hra)
- [5] Game Theory  
[http://en.wikipedia.org/wiki/Game\\_theory](http://en.wikipedia.org/wiki/Game_theory)
- [6] Abalone - Wikipedia  
[http://en.wikipedia.org/wiki/Abalone\\_\(board\\_game\)](http://en.wikipedia.org/wiki/Abalone_(board_game))
- [7] Abalone - MSO  
<http://www.boardability.com/games/abalone.html>
- [8] Mensa Select Award  
[http://www.boardgamegeek.com/wiki/page/Mensa\\_Select](http://www.boardgamegeek.com/wiki/page/Mensa_Select)
- [9] Abalone - BoardGameGeek  
<http://www.boardgamegeek.com/boardgame/526/abalone>
- [10] Abalone Extra Player Marbles - BoardGameGeek  
<http://www.boardgamegeek.com/boardgame/11679/abalone-extra-player-marbles>
- [11] Abalone Quattro - BoardGameGeek  
<http://www.boardgamegeek.com/boardgame/31968/abalone-quattro>
- [12] Abalone – PBM Server  
<http://www.gamerz.net/~pbmserv/abalone.html>
- [13] Terrace – Official Page ( Archive )  
<http://web.archive.org/web/20060430134153/www.terracegames.com/>
- [14] Terrace - Wikipedia  
[http://en.wikipedia.org/wiki/Terrace\\_\(board\\_game\)](http://en.wikipedia.org/wiki/Terrace_(board_game))
- [15] Terrace – PBM Server  
<http://www.gamerz.net/~pbmserv/terrace.html>
- [16] Terrace – Memory Alpha  
<http://memory-alpha.org/en/wiki/Terrace>
- [17] GIPF projekt - Wikipedia

- [http://cs.wikipedia.org/wiki/GIPF\\_projekt](http://cs.wikipedia.org/wiki/GIPF_projekt)
- [18] Yinsh - Wikipedia  
<http://en.wikipedia.org/wiki/Yinsh>
- [19] Yinsh – Official Page  
<http://www.gipf.com/yinsh/>
- [20] GIPF projekt – Official Page  
[http://www.gipf.com/project\\_gipf/index.html](http://www.gipf.com/project_gipf/index.html)
- [21] Yinsh Notations  
<http://www.gipf.com/yinsh/notations/notation.html>
- [22] E. Ozcan a B. Hulagu : A Simple Intelligent Agent for Playing Abalone Game: ABLA, 2004  
[http://cse.yeditepe.edu.tr/~eozcan/research/papers/ABLA\\_id136final.pdf](http://cse.yeditepe.edu.tr/~eozcan/research/papers/ABLA_id136final.pdf)
- [23] Java Practices  
<http://www.javapractices.com/>
- [24] Game tree  
[http://en.wikipedia.org/wiki/Game\\_tree](http://en.wikipedia.org/wiki/Game_tree)
- [25] Solved Games  
[http://en.wikipedia.org/wiki/Solved\\_game](http://en.wikipedia.org/wiki/Solved_game)
- [26] Game Complexity  
[http://en.wikipedia.org/wiki/Game\\_complexity](http://en.wikipedia.org/wiki/Game_complexity)
- [27] Pascal Chorus: Master Thesis - Implementing a Computer Player for Abalone using Alpha-Beta and Monte-Carlo Search, Maastricht University 2009  
[www.unimaas.nl/games/files/msc/pcreport.pdf](http://www.unimaas.nl/games/files/msc/pcreport.pdf)
- [28] N. P. P. M. Lemmens: Constructing an Abalone Game-Playing Agent, Maastricht University 2005  
[www.unimaas.nl/games/files/msc/Lemmens\\_BSc-paper.pdf](http://www.unimaas.nl/games/files/msc/Lemmens_BSc-paper.pdf)
- [29] PBM Server  
<http://www.gamerz.net/pbmserv/>
- [30] Bruce Moreland: Programming Topics (Archive)  
<http://web.archive.org/web/20040403211728/brucemo.com/compchess/programming/index.htm>
- [31] Hamed Ahmadi: An Introduction To Game Tree Algorithms  
<http://www.hamedahmadi.com/gametree/>
- [32] Oswin Aichholzer, Franz Aurenhammer, Tino Werner: Algorithmic Fun – Abalone, 2002  
[http://www.ist.tugraz.at/staff/aichholzer/research/rp/abalone/tele1-02\\_aich-abalone.pdf](http://www.ist.tugraz.at/staff/aichholzer/research/rp/abalone/tele1-02_aich-abalone.pdf)
- [33] Thomas Debray: Intelligent Search Techniques Project  
<http://www.netstorm.be/content.php?contentid=2>
- [34] Java SE  
<http://java.sun.com/javase/>

- [35] Java Table Layout  
<https://tablelayout.dev.java.net/>
- [36] Apache Commons Library  
<http://commons.apache.org/>
- [37] Abalone – Wikipedia (fr)  
[http://fr.wikipedia.org/wiki/Abalone\\_\(jeu\)](http://fr.wikipedia.org/wiki/Abalone_(jeu))
- [38] Adversarial Search  
[http://en.wikipedia.org/wiki/Adversarial\\_search#Adversarial\\_search](http://en.wikipedia.org/wiki/Adversarial_search#Adversarial_search)
- [39] Abalone Board Image  
<http://www.brettspiele-in-ol.de>
- [40] Manhattanská vzdálenost  
[http://en.wikipedia.org/wiki/Taxicab\\_geometry](http://en.wikipedia.org/wiki/Taxicab_geometry)
- [41] Java - Velikost multidimensionálních polí v paměti  
[http://www.javamex.com/tutorials/memory/array\\_memory\\_usage.shtml](http://www.javamex.com/tutorials/memory/array_memory_usage.shtml)
- [42] Java Reflexion API  
<http://java.sun.com/javase/6/docs/technotes/guides/reflection/index.html>
- [43] Java Classpath  
[http://en.wikipedia.org/wiki/Classpath\\_\(Java\)](http://en.wikipedia.org/wiki/Classpath_(Java))
- [44] LRU Algorithm  
[http://en.wikipedia.org/wiki/Cache\\_algorithms#Least\\_Recently\\_Used](http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used)
- [45] Game Tree Searching and Pruning  
[http://www.cs.unm.edu/~aaron/downloads/qian\\_search.pdf](http://www.cs.unm.edu/~aaron/downloads/qian_search.pdf)
- [46] Computer Chess Programming Theory  
<http://www.frayn.net/beowulf/theory.html>
- [47] Chess Wiki  
<http://chessprogramming.wikispaces.com/>
- [48] Strategy Game Programming  
<http://www.fierz.ch/strategy.htm>
- [49] Benson Lee, Hyun Joo Noh: Abalone – Final Project Report  
[www.cs.cornell.edu/~hn57/pdf/AbaloneFinalReport.pdf](http://www.cs.cornell.edu/~hn57/pdf/AbaloneFinalReport.pdf)
- [50] Nathan Sturtevant: How computers play games with you  
[www.cs.ucla.edu/classes/spring03/cs161/I1/CS161Games03.ppt](http://www.cs.ucla.edu/classes/spring03/cs161/I1/CS161Games03.ppt)
- [51] Two Player Perfect-Information Games  
<http://www.ise.bgu.ac.il/faculty/felner/teaching/new7.ppt>
- [52] Java Forums  
<http://forum.java.sun.com/>
- [53] Sun Development Network  
<http://java.sun.com/>

## Literatura

- [54] Petr Houdek: Bakalářská práce – Hexxagon  
České Vysoké Učení Technické – Fakulta Elektrotechnická, Praha 2007
- [55] Russel Norwig: Artificial Intelligence: Modern Approach - Second Edition, Prentice Hall 2002
- [56] Piotr Wróblewski: Algoritmy – Datové struktury a programovací techniky, 2003  
Přel. Marek Michálek, Bogdan Kiszka, 1. vyd. Computer Press, Brno 2004
- [57] M. Tim Jones: Artificial Intelligence: A Systems Approach, Jones and Bartlett Publishers 2009
- [58] George T. Heineman, Gary Pollice, Stanley Selkow: Algorithms In Nutshell, O'Reilly Media 2008
- [59] Rudolf Pecinovský: Návrhové vzory  
vyd. Computer Press, Brno 2007
- [60] Ian F. Darwin: Java kuchařka programátora 2004  
Přel. J. Gregor. 1. vyd. Computer Press, Brno 2006
- [61] Brett Spell: Java Programujeme profesionálně, 2000  
Přel. B. Kiszka. 1. vyd. Computer Press, Praha

## A Seznam zkratek

( České ekvivalenty se nachází pouze u zkratek, které mají český ekvivalent. )

AI - Artificial Intelligence - umělá inteligence

UI - User Interface - uživatelské rozhraní

GUI - Graphical User Interface - grafické uživatelské rozhraní

MSO - Mind Sports Organization

TT - Transposition Table - transpoziční tabulka

IDE - Integrated Development Environment – vývojové prostředí

J2SE - Java 2 Standard Edition

IO - Input / Output - vstupně / výstupní

UML - Unified Modelling Language

LRU - Least Recently Used

JRE – Java Runtime Environment

JDK – Java Development Kit

SDK – Software Development Kit

CD – Compact Disc – kompaktní disk

RAM – Random-Access Memory – operační paměť

API – Application Programming Interface - rozhraní pro programování aplikací

MO - Move Ordering - řazení tahů

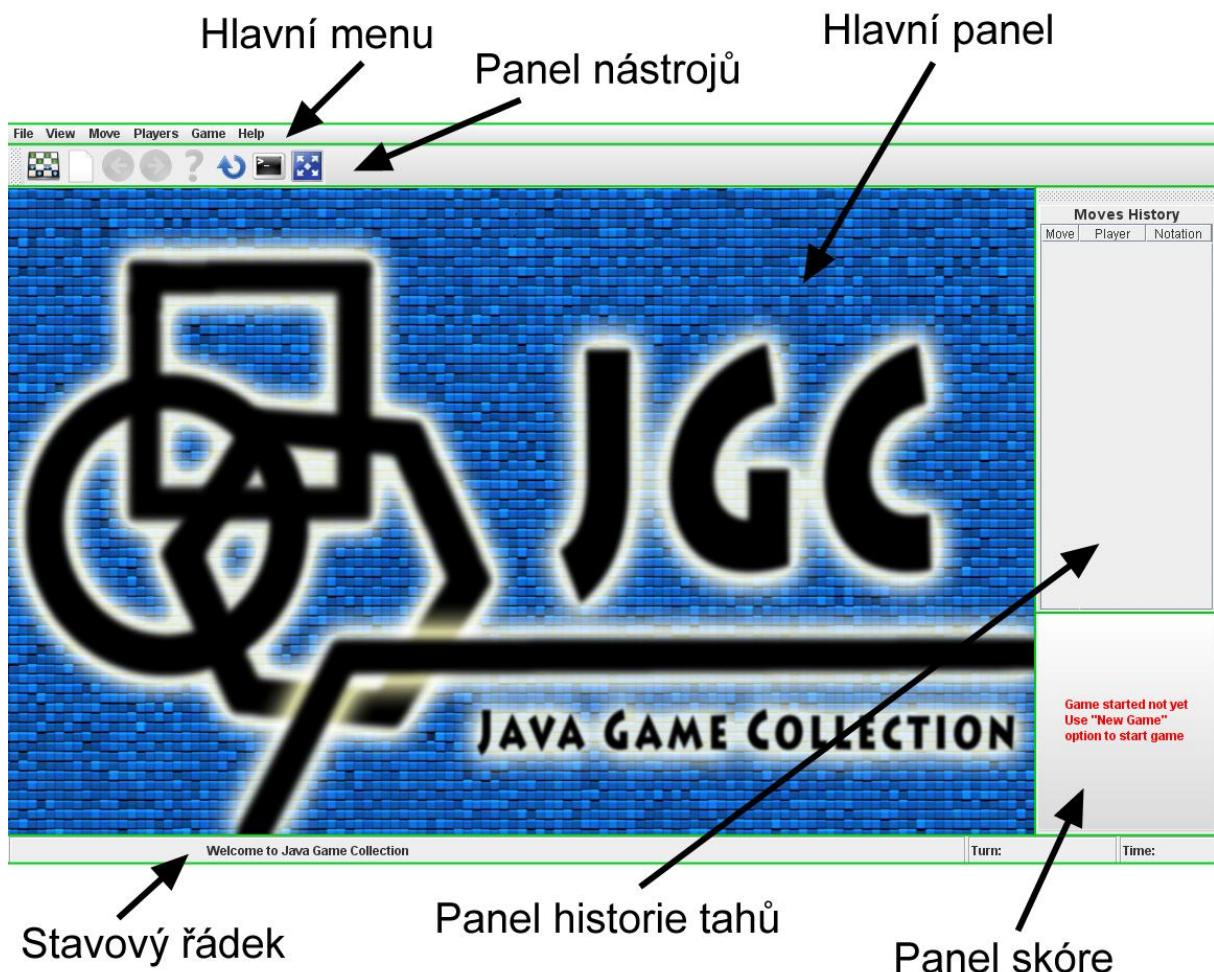
## B Uživatelská příručka

Uživatelská příručka popisuje funkce dostupné v aplikaci a jejich ovládání.

### B.1 Obecné

Aplikace *Java Game Collection* umožňuje hraní libovolných her, které jsou v aplikaci naprogramované. V aplikaci jsou naprogramované hry Abalone, Terrace a Yinsh. Další hry je možno doprogramovat (viz kapitola 24). Aplikace umožňuje zadávat některá méně obvyklá nastavení pomocí parametrů příkazové řádky (viz kapitola 25).

### B.2 Hlavní okno aplikace



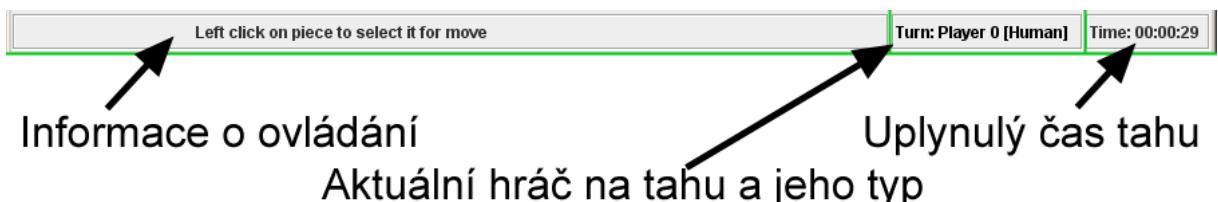
Obr. 44 - Popis úvodní obrazovky.

Prvky hlavního okna aplikace (obrázek 44) jsou následující:

- *Hlavní menu* – obsahuje možnosti a funkce k ovládání aplikace
  - *Hrací panel* – oblast využívaná pro hru
  - *Panel historie tahů* – oblast, kde se ukazují jednotlivé tahy
  - *Panel skóre* – oblast, kde pro každého hráče se ukazuje jeho skóre ve hře (viz podkapitola B.2.3)
  - *Panel nástrojů* - umožňuje rychlý přístup k nejpoužívanějším volbám
  - *Stavový řádek* – obsahuje informace k zjištění stavu hry

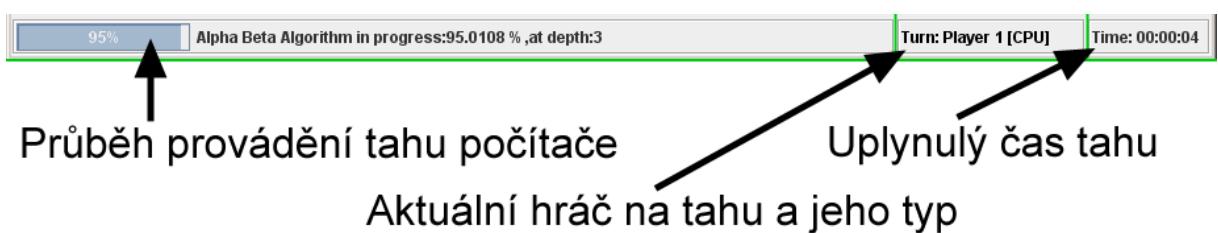
## B.2.1 Stavový řádek

Stavový řádek (obrázek 45) ukazuje, který hráč je na tahu, zda jde o počítač, a jaká je jeho barva. Rovněž také ukazuje čas uplynulý od provedení předchozího tahu. V hlavní části ukazuje informace, které se hlavně týkají ovládání v aktuálním kontextu.



Obr. 45 - Popis stavového řádku v tahu lidského hráče

Stavový řádek při provádění tahu počítače (obrázek 46) v hlavní části ukazuje jeho procentuální průběh. Dále ukazuje algoritmus, který se provádí a aktuální hloubku provádění.



#### Obr. 46 - Popis stavového řádku v tahu počítače

### B.2.2 Panel historie tahú

Panel historie tahů zobrazuje provedené tahy. U každého tahu se zobrazuje jeho pořadí, index hráče, který tak provedl a jeho notaci. Pokud se notace nevejde do pole, je možné ji vidět celou v „tooltipu“. Aktuální tah je zvýrazněn červeně.

Moves History		
Move	Player	Notation
1	0	f6
2	1	e4
3	0	d5
4	1	i7
5	0	g9
6	1	

Obr. 47 - Panel historie tahů.

### B.2.3 Panel skóre

Panel skóre zobrazuje skóre pro všechny hráče. U každého hráče je vidět jeho typ a číselná hodnota skóre. Barva pozadí odpovídá barvě kamenů hráče.

Player 0 - Human	80
Player 1 - Human	80
Player 2 - Human	80
Player 3 - Human	80

Obr. 48 - Panel skóre.

### B.2.4 Hlavní menu

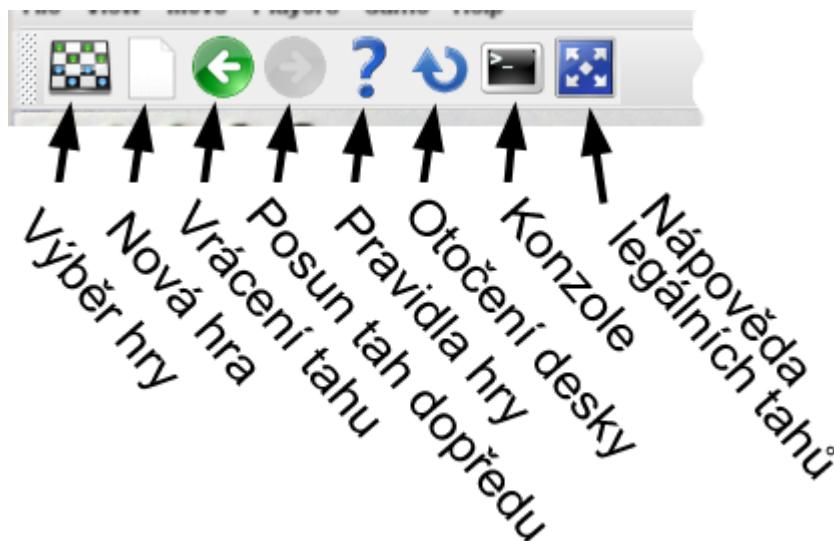


Obr. 49 - Hlavní menu a jeho funkce.

Hlavní menu obsahuje následující volby:

- *File*
  - *Select Game* – Vybere hru, která se bude hrát.
  - *New Game* – Začne novou hru, která byla vybrána předem pomocí *Select Game*.
  - *Exit* – Ukončí aplikaci.
- *View*
  - *Tool Bar* – Pokud je zaškrtnuto, zobrazí panel nástrojů (viz podkapitola B.2.5).
  - *Status Bar* – Pokud je zaškrtnuto, zobrazí stavový řádek (viz podkapitola B.2.1).
  - *Score Panel* – Pokud je zaškrtnuto, zobrazí panel skóre.
  - *Moves Panel* – Pokud je zaškrtnuto, zobrazí panel historie tahů (viz podkapitola B.2.2).
  - *Rotate Board* – Otočí pohled desky.
  - *Console* – Zobrazí obrazovku s konzolí (viz podkapitola B.4.1).
  - *Set Look & Feel* – Nastaví grafické téma aplikace z dostupných vzhledů.
    - *Metal Steel* – Standardní vzhled Javy 1.5
    - *Metal Ocean* – Standardní vzhled Javy 1.6
    - *Nimbus* – Dostupné od verze Javy 1.6 update 10. Podpora vzhledu je experimentální, proto může způsobovat grafické chyby.
    - *CDE/Motif* – Standardní vzhled operačního systému Solaris.
    - *Windows* – Standardní vzhled Windows XP, pokud je aktuálně používán.
    - *Windows Classic* – Standardní vzhled Windows 2000 a Windows 95, pokud je aktuálně používán.
- *Move*
  - *Undo Move* – Vrátí tah nazpět, pokud není první. Hrací prvky na ploše se obnoví. Volba je dostupná po začátku hry a to pouze pokud je proveditelná.
  - *Redo Move* – Posune tah dopředu, pokud není poslední. Hrací prvky na ploše se obnoví. Volba je dostupná po začátku hry a to pouze pokud je proveditelná.
  - *Legal Move Hints* – Když je zapnuto, zobrazují se graficky legální tahy.
- *Players* – Zobrazuje všechny aktivní hráče, jejich typ a vlastnosti. Volby jsou viditelné až po začátku hry.
- *Game* – Zobrazuje informace o nastavení hry a o zvolené startovní pozici. Volby jsou viditelné až po začátku hry.
- *Help*
  - *Game Rules* – Zobrazí panel s pravidly hry (podkapitola B.4.2). Volba je dostupná až po zvolení hry.
  - *About* – Zobrazí informace o autorovi aplikace a její verzi.

### B.2.5 Panel nástrojů



Obr. 50 - Popis panelu nástrojů.

Panel nástrojů umožňuje využít některé volby z hlavního menu přímo. Nedostupné jsou volby vyšedlé. Jednotlivé ikony z panelu nástrojů jsou popsány níže. V závorce je vždy uvedena odpovídající volba z hlavního menu.

- *Výběr hry* – (*File – Select Game*) - Vybere hru, která se bude hrát.
- *Nová hra* – (*File – New Game*) - Začne novou hru, která byla vybrána předem pomocí *Select Game*.
- *Vracení tahu* – (*Move – Undo Move*) - Vrátí tah nazpět, pokud není první. Hrací prvky na ploše se obnoví. Volba je dostupná po začátku hry a to pouze pokud je proveditelná.
- *Posun tah dopředu* – (*Move – Redo Move*) - posune tah dopředu, pokud není poslední. Hrací prvky na ploše se obnoví. Volba je dostupná po začátku hry a to pouze pokud je proveditelná.
- *Pravidla hry* – (*Help – Game Rules*) - Zobrazí panel s pravidly hry (podkapitola B.4.2). Volba je dostupná až po zvolení hry.
- *Otočení desky* – (*View – Rotate Board*) - Otočí pohled desky.
- *Zobrazení konzole* – (*View – Console*) - Zobrazí obrazovku s konzolí (viz podkapitola B.4.1).
- *Zobrazení nápoovědy legálních tahů* – (*Move – Legal Move Hints*) - Když je zapnuto, zobrazují se graficky legální tahy.

## B.3 Přeuspořádání panelů grafického rozhraní

Jednotlivé panely grafického rozhraní lze přeuspořádat, a tím změnit jejich polohu. Změnit polohu lze u následujících panelů:

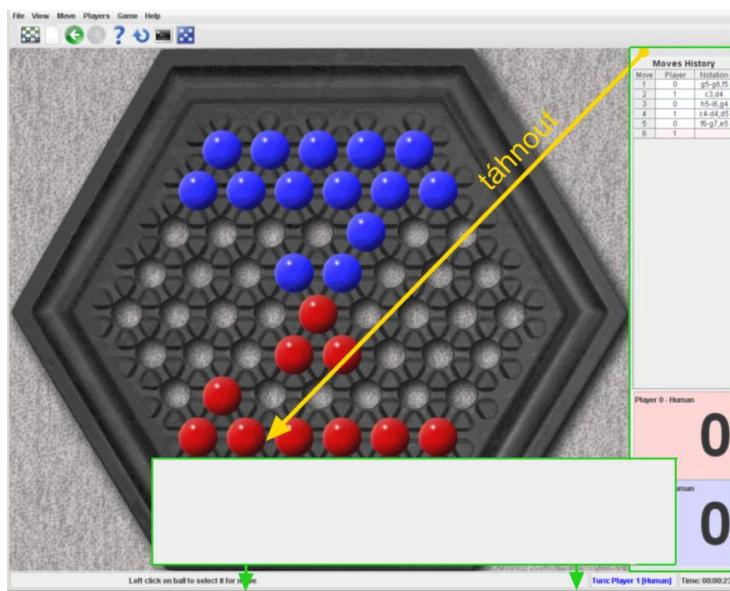
- Panel skóre
- Panel historie tahů
- Panel nástrojů

Tyto panely lze přemístit vždy po jednom okolo hracího panelu na jednotlivé strany: nahoru, dolů, napravo, nalevo nebo také mimo okno aplikace. Přetažení panelu na jinou stranu se provádí pomocí kliknutí myši na kraj panelu (operace *drag & drop*), který je zobrazen na následujícím obrázku 51, a jeho přetažením mimo okno aplikace nebo k jiné straně hracího panelu, která je neobsazena (přetažení je zobrazeno na obrázku 53).

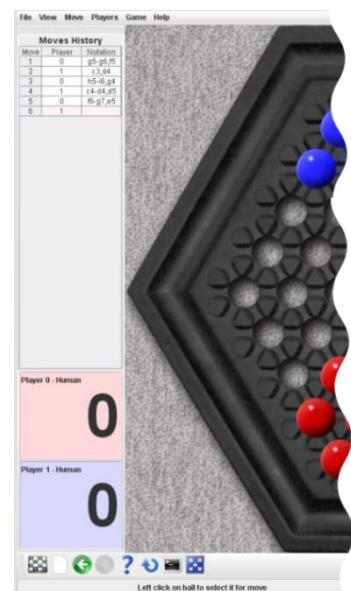


Obr. 51 - Oblast panelu, určená na vyvolání přetažení.

Akce se dokončí uvolněním tlačítka myši. Panely přemístěné na jiné než výchozí strany jsou zobrazeny na obrázku 52.



Obr. 53 - Demonstrace přetažení panelu k jiné straně aplikace. Žlutě označen směr tažení myší. Zeleně je označen přemísťovaný panel.



Obr. 52 - Demonstrace změny polohy panelu skóre, historie tahů a nástrojů po přetažení.

Panel přemístěný mimo okno aplikace je zobrazen na obrázku 54. Přemístěné panely lze také samozřejmě vrátit na původní polohy.



Obr. 54 - Panel  
přemístěný mimo  
okno aplikace.

## B.4 Dialogy aplikace

### B.4.1 Dialog konzole

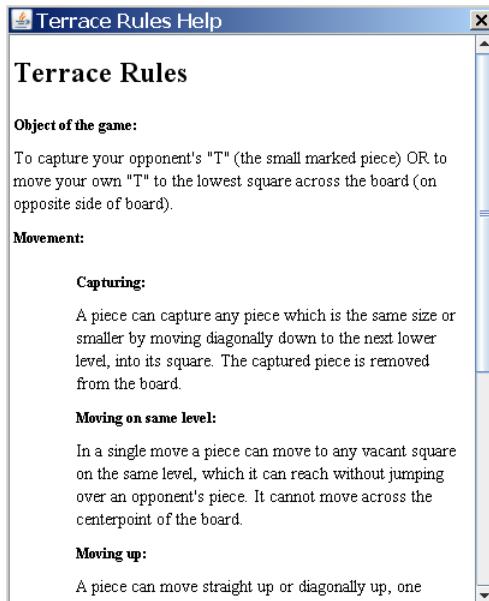
Dialog konzole ukazuje v grafickém okně výstup z konzole aplikace.

A screenshot of a window titled "Console". The window contains a large amount of text output, which appears to be a log of moves from a game. The text is in a monospaced font and is too long to reproduce here in full. It includes entries like "i36;x:3,y:3,cidx:27,layer:9", "i37;x:3,y:2,cidx:26,layer:8", and many other similar coordinates and layer numbers.

Obr. 55 - Dialog konzole.

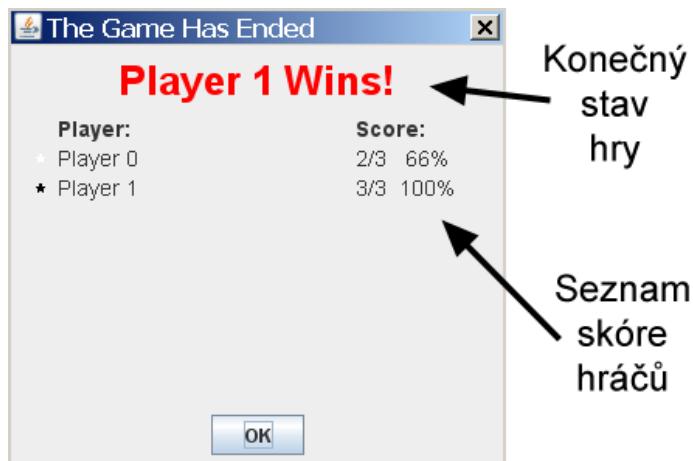
## B.4.2 Dialog pravidel

Dialog pravidel zobrazuje pravidla hry.



Obr. 56 - Dialog pravidel.

## B.4.3 Dialog konce hry



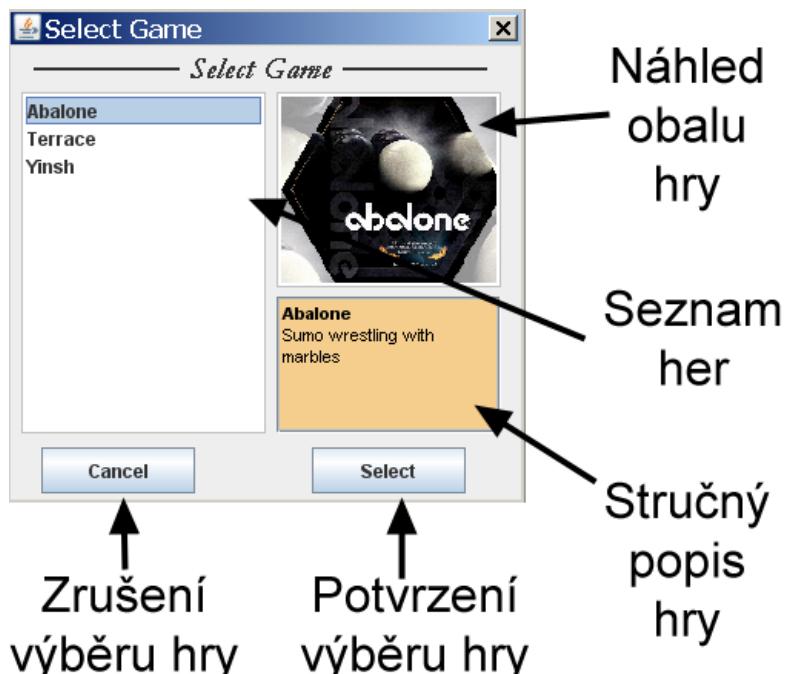
Obr. 57 - Popis dialogu konce hry.

Tento dialog se zobrazí po skončení hry. Informuje o tom, jak skončila hra a s jakým výsledkem. Jednotlivé části dialogu popisují následující:

- *Konečný stav hry* – Informuje, jak hra skončila. Jsou možné následující možnosti:
  - *Player X Wins!* – Hru vyhrál hráč s číslem X.
  - *Draw!* – Hra skončila jako remíza.
  - *Repetition Draw!* – Hra skončila jako repetiční remíza (detailněji popsáno v kapitole 20).
  - *Nobody wins?* – Nikdo hru nevyhrál.
- *Seznam skóre hráčů* – Popisuje dosažené skóre od každého hráče z maximálního možného a jeho procentuální vyjádření.

#### B.4.4 Dialog výběru hry

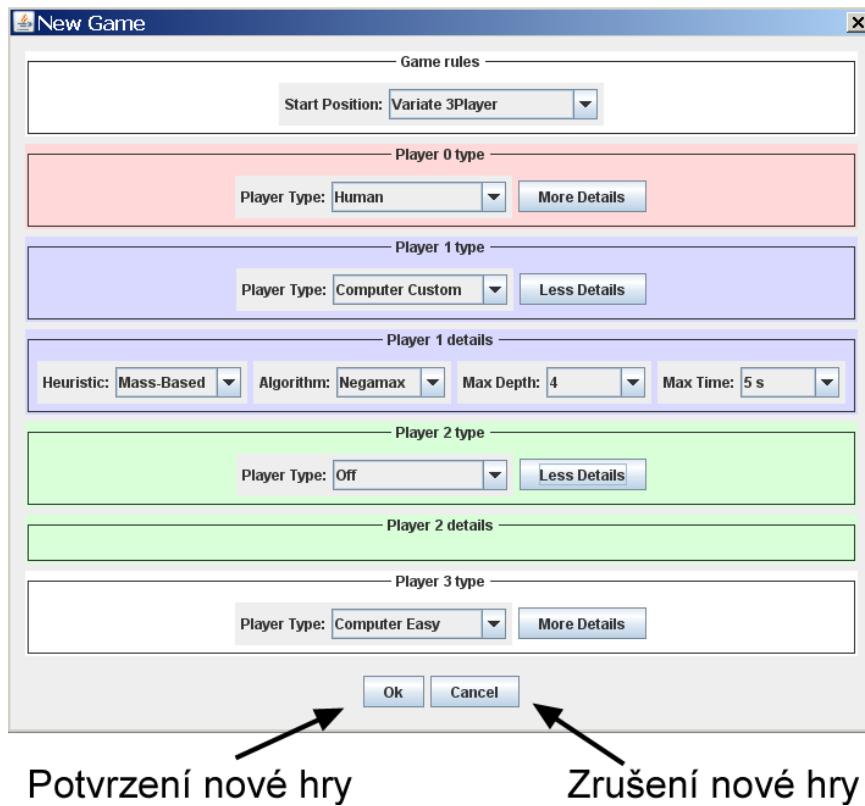
Tento dialog se zobrazuje při výběru hry.



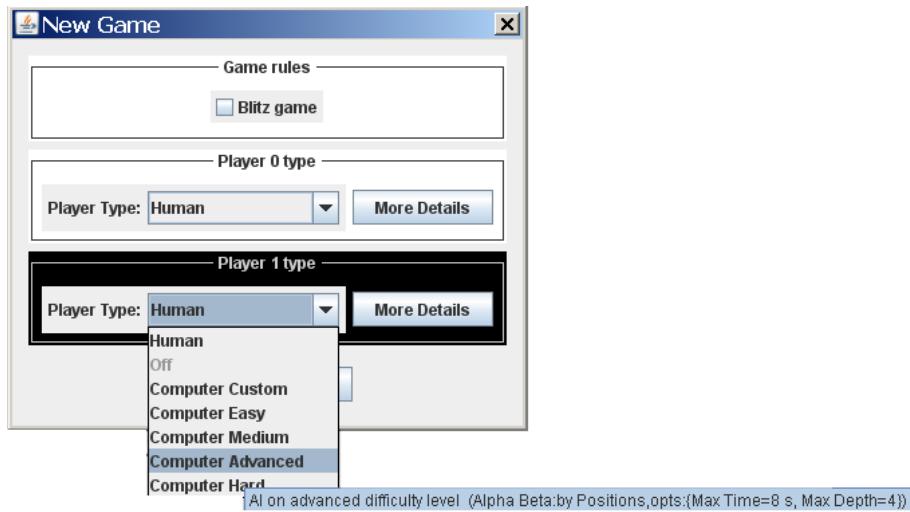
Obr. 58 - Popis dialogu výběru hry.

- *Zrušení výběru hru* – Zruší výběr nové hry a zavře dialog.
- *Potvrzení výběru hry* – Vybraná hra se zvolí. Dialog se zavře a automaticky se otevře dialog nové hry.
- *Náhled obalu hry* – Zobrazuje náhled obalu hry.
- *Stručný popis hry* – Zobrazuje stručný popis hry.
- *Seznam her* - Seznam nalezených her s možností jejich volby.

### B.4.5 Dialog nové hry



Obr. 59 - Popis dialogu nové hry.



Obr. 60 - Dialog nové hry se zobrazením dalších specifických vlastností.

Dialog nové hry umožňuje zadání parametrů hry a každého hráče. Jednotlivé prvky dialogu jsou následující:

- *Tlačítka More Details* – Zobrazí pole *Player details* pro nastavení detailů hráče, v jehož panelu se nachází toto tlačítko.
- *Tlačítka Less Details* – Skryje pole *Player details* pro nastavení detailů hráče, v jehož panelu se nachází toto tlačítko.
- *Zrušení nové hry* – Zruší nastavování nové hry a zavře dialog.
- *Potvrzení nové hry* – Vytvoří novou hru podle nastavených nastavení. Tato volba je dostupná, pouze pokud je nastaven alespoň minimální počet pro danou hru.

Vlastnosti hry, které je možno v rámci dialogu nastavit jsou následující:

- *Game Rules* – umožňuje nastavit počáteční specifické vlastnosti hry. Implementované hry umožňují nastavit následující vlastnosti:
  - *Start Positions* – (pro hry Abalone, Terrace) – Umožňuje nastavit počáteční pozici hry. Pozice jsou dostupné podle počtu zvolených hráčů.
  - *Blitz game* – (pro hru Y Cindy) – Volba pro zvolení bleskové hry. Hraje se na 1 vítěznou řadu místo 3.
- *Player type* – Volba typu hráče.
  - *Human* – Hráče bude ovládat člověk.
  - *Off* – Hráč nebude hrát. Tato volba je dostupná v závislosti na minimálním počtu hráčů pro danou hru (pro Abalone, Terrace, Y Cindy jsou to 2 hráčů).
  - *Computer Custom* - Hráče bude ovládat počítač s umělou inteligencí. Umožňuje nastavit detailně úroveň obtížnosti pomocí voleb v polí *Player details*.
  - *Computer Easy* - Hráče bude ovládat počítač s umělou inteligencí s lehkou obtížností. Tooltip ukazuje popis volby a odpovídající nastavení u volby *Computer Custom*.
  - *Computer Medium* - Hráče bude ovládat počítač s umělou inteligencí se střední obtížností. Tooltip ukazuje popis volby a odpovídající nastavení u volby *Computer Custom*.
  - *Computer Advanced* - Hráče bude ovládat počítač s umělou inteligencí s pokročilou obtížností. Tooltip ukazuje popis volby a odpovídající nastavení u volby *Computer Custom*.
  - *Computer Hard* - Hráče bude ovládat počítač s umělou inteligencí s těžkou obtížností. Tooltip ukazuje popis volby a odpovídající nastavení u volby *Computer Custom*.
- *Player details* – Umožňuje nastavit rozšiřující možnosti pro hráče zvolené v poli *Player Type*. Tyto možnosti jsou dostupné pouze pro volbu *Computer Custom*.
  - *Heuristic* – Volba typu heuristické funkce. Volba heuristik závisí na hře.
  - *Algorithm* – Volba typu algoritmu pro umělou inteligenci.
    - *Best Move* – Algoritmus Best Move (viz podkapitola 18.1).

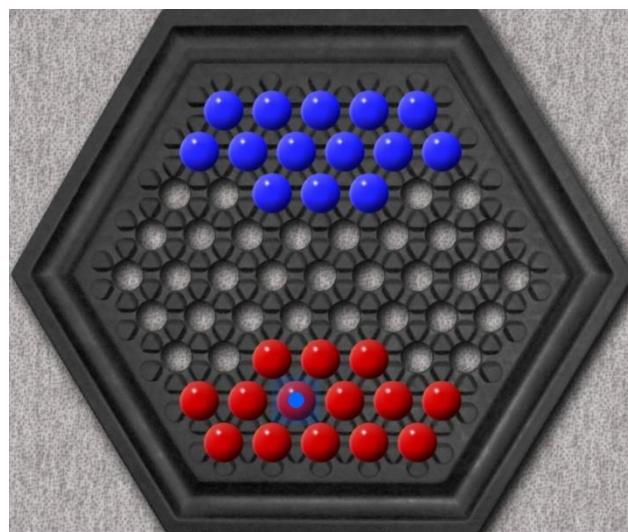
- *Negamax* – Algoritmus Negamax (viz podkapitola 18.2).
- *Alpha Beta* – Algoritmus Alfa Beta (viz podkapitola 18.3).
- *Možnosti algoritmu* – Volba doplňujících možností pro algoritmus, následující možnosti jsou dostupné pro algoritmy Negamax a Alfa Beta.
  - *Max Depth* – Volba maximální cílové hloubky.
    - *Unlimited* – Maximální hloubka bude „nekonečná“, resp. 99. Prohledávání bude omezené maximálním časem, pro který není povoleno být nekonečný.
    - *Ostatní volby* – Číslo určující maximální hloubku.
  - *Max Time* – Volba maximální možného času, který bude použit na tah umělé inteligence.
    - *Unlimited* – Maximální čas není omezen. Čas bude omezen maximální hloubkou, pro kterou není povoleno být nekonečnou.
    - *Ostatní volby* – Maximální čas bude odpovídat zvolené volbě.

## B.5 Ovládání aplikace v průběhu hry

Po začátku nové hry a potvrzení dialogu nové hry se průběh hry ovládá přes hrací panel s pomocí myši. O ovládání v aktuálním kontextu nás informuje vždy stavový panel. Ovládání bude popisováno na obrázcích hracího panelu.

### B.5.1 Hra Abalone

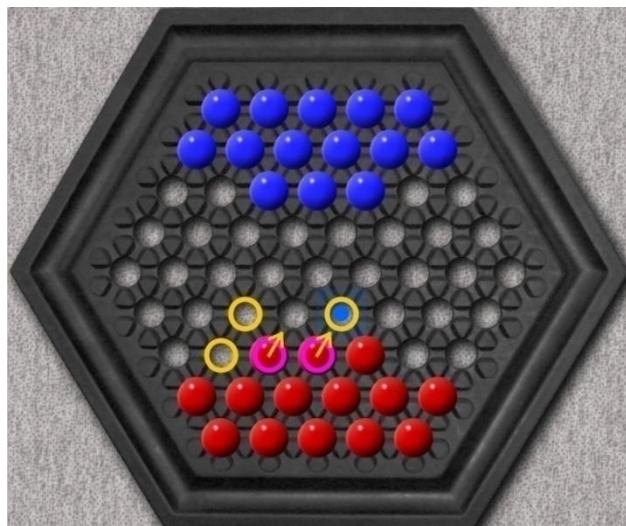
Pro pohyb kamenu, najedeme nad vlastní kámen, se kterým chceme hrát (viz obrázek 61) a označíme ho levým tlačítkem myši.



Obr. 61 - Abalone - výběr kamenu.

Když kámen označíme, zvýrazní se fialovou barvou. Další kámen, se kterým je možno hrát se současně označeným, je možné označit stejným způsobem opět levým tlačítkem myši. Označený kámen, lze odznačit najetím kurzoru na něj a stiskem opět levého tlačítka myši.

Výběr směru, na který budeme s kameny táhnout, provedeme tak, že najedeme na okolní pozici označenou oranžově (pokud je zapnuté zobrazování legálních tahů). Směr tahu je vyznačen oranžovými šipkami (viz obrázek 62).



Obr. 62 - Abalone – výběr cílové pozice.

Posun kamenů na cílovou pozici provedeme klepnutím pravým tlačítkem myši na jedno z oranžově označených polí.

### B.5.2 Hra Terrace

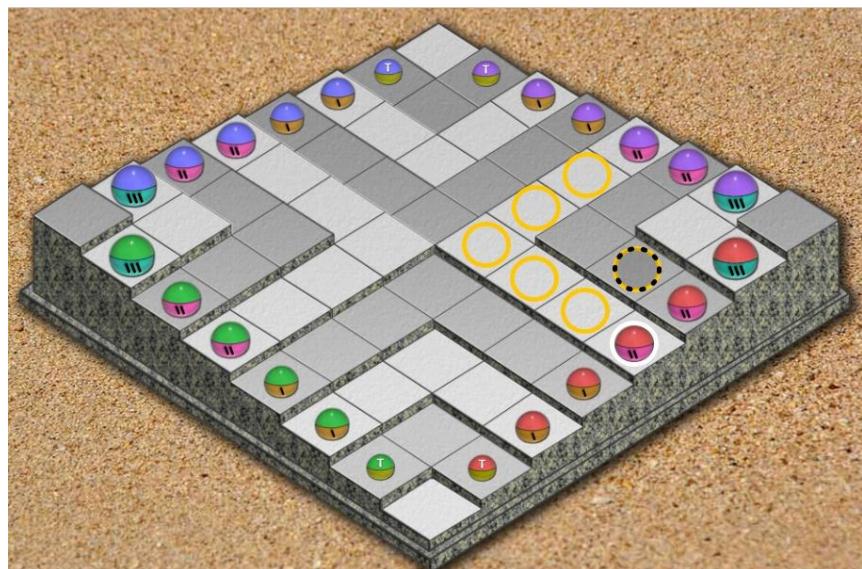
Pro pohyb kamenu, najedeme nad vlastní kámen, se kterým chceme hrát (viz obrázek 63) a označíme ho levým tlačítkem myši. Kámen, nad kterým se nachází kurzor myši, je označen černou přerušovanou čarou.



Obr. 63 - Terrace – výběr kamenu.

Pokud chceme vybrat jiný kámen, stačí levým tlačítkem myši označit jiný vlastní kámen. Nebo je také možno klepnutím levým tlačítkem myši označit aktuální označený kámen.

Aktuálně možné cílové pole pro tah jsou zobrazeny oranžově (pokud se zobrazují legální tahy), viz obrázek 64). Vybraný kámen je označen bíle. Pro provedení tahů stačí kliknout levým tlačítkem myši na jedno z cílových polí.

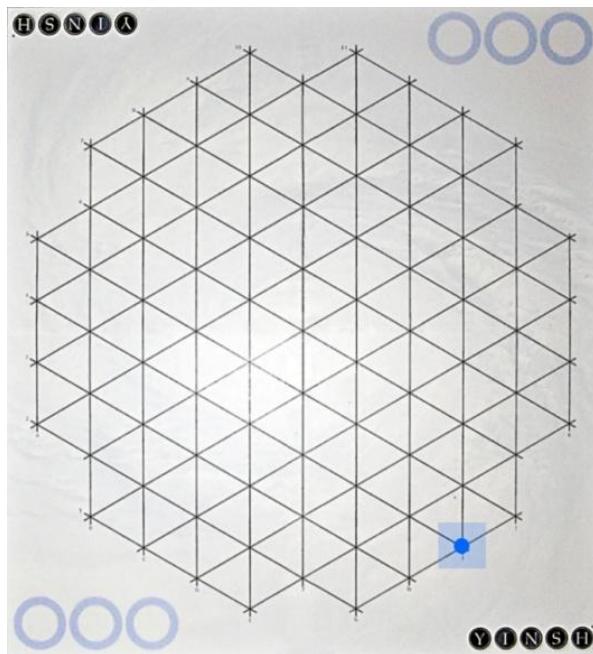


Obr. 64 - Terrace – výběr cílové pozice.

### B.5.3 Hra Yinsh

#### B.3.5.1 Fáze umisťování kroužků

Pro umístění vlastního kroužku, klepneme levým tlačítkem myši na aktuálně zvolené pole vyznačené modře, přičemž toto pole musí být prázdné (zobrazuje obrázek 65).

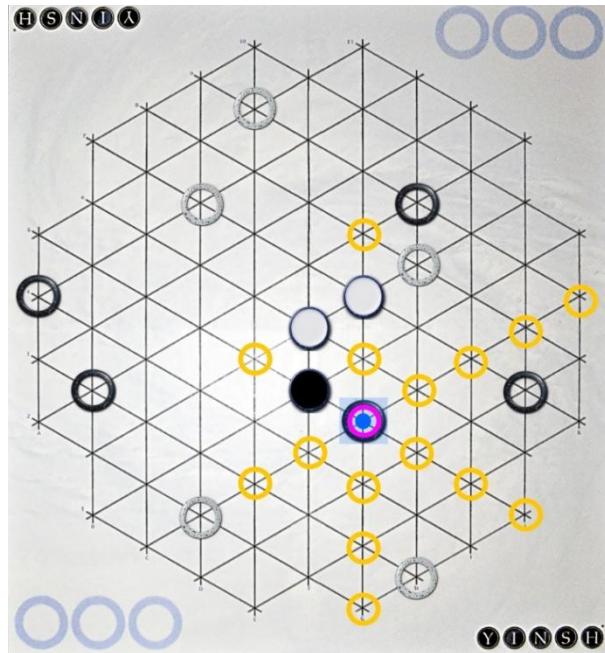


Obr. 65 - Yinsh – výběr pole pro kroužek.

#### B.3.5.2 Fáze tvorby řad

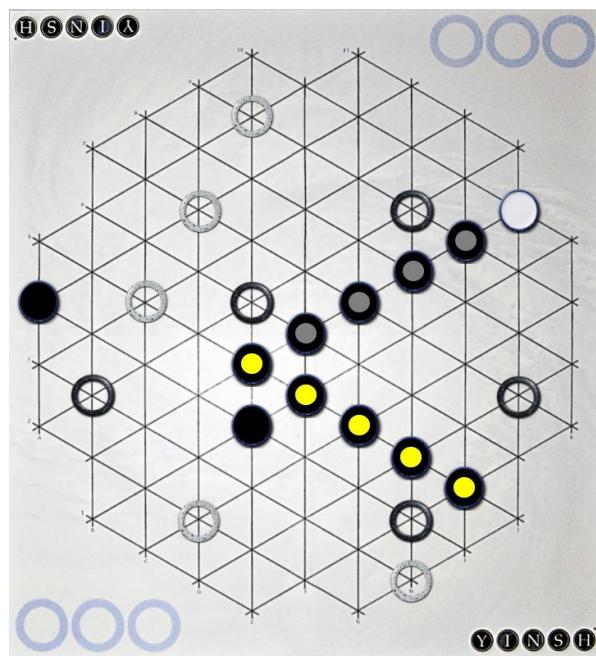
Pro umístění kamenu je nutné klepnout levým tlačítkem myši na vlastní kroužek. Ten se pak zbarví fialovou barvou. Pokud chceme označit jiný vlastní kroužek, musíme odznačit původní levým tlačítkem myši a poté označit nový opět levým tlačítkem myši.

Pokud se zobrazují legální tahy, jsou pole, na které lze následně přesunout kroužek, označeny oranžově (viz obrázek 66). Přesunutí kroužku vykonáme zvolením legálního cílového pole levým tlačítkem myši.



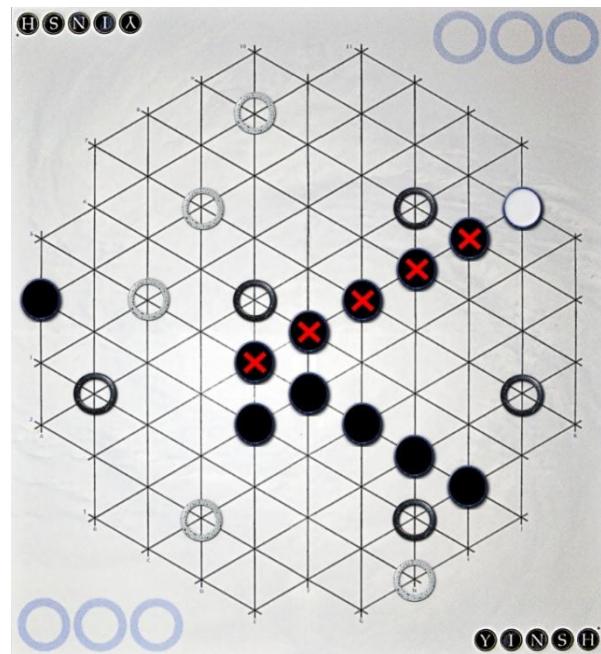
Obr. 66 - YINSH – výběr cílové pozice kroužku.

Při odebírání řad může nastat situace, že lze volit z více možností, jak odebrat kameny řady. V tom případě každý kámen, který je součástí alespoň jedné z těchto variant, je označen šedivě. Aktuální zvolená možnost na odebrání řady je označena žlutě, jako na obrázku 67. Poté pomocí pravého tlačítka lze volit mezi jednotlivými možnostmi a pomocí levého odebereme aktuálně zvolenou řadu.



Obr. 67 - YINSH – výběr řady na odebrání z více možností.

Když se odebírá řada, označí se červenými křížky, jako na obrázku 68. V této situaci volíme kroužek na odebrání v barvě řady. Tento kroužek zvolíme tak, že nad něj najedeme myší a klikneme na něj levým tlačítkem myši.



Obr. 68 - Yinsh – výběr kroužku na odebrání.

## C Instalační příručka

### C.1 Instalační požadavky aplikace

Minimální konfigurace:

- Operační systém Mac OS, Windows, Linux, Solaris 32bit/64bit
- Procesor 500 MHz nebo rychlejší
- 64 MB RAM
- 20 MB volného místa na disku
- Rozlišení VGA (640 x 480)
- Java JRE 1.6 nebo novější

Doporučená konfigurace

- Operační systém Mac OS, Windows, Linux, Sparc, Solaris 32bit/64bit
- Procesor 1,5 GHz nebo rychlejší
- 512 MB RAM
- 50 MB volného místa na disku
- Rozlišení SXGA (1280 x 1024)
- Java JRE 1.6 nebo novější

Aplikace byla testována v prostředí MS Windows, avšak díky multiplatformnosti Javy by měla aplikace fungovat stejně dobře i na dalších operačních systémech.

### C.2 Instalace prostředí Java

Pro spuštění aplikace je nutné mít v počítači nainstalovánu Javu JRE verze 1.6 a novější. Přítomnost Javy a její verzi v počítači lze ověřit pomocí příkazu v příkazové řádce / terminálu „*java - version*“. V případě, že v počítači Java chybí, je nutné nainstalovat její odpovídající verzi pro daný operační systém pomocí instalačního souboru z adresáře „*Application/Java SDK*“. Samotná instalace se pak provádí podle kroků instalačního průvodce z tohoto souboru.

### C.3 Spuštění aplikace

Aplikaci lze přímo spustit z adresáře „*Application/Bin*“ na CD<sup>6</sup>:

- Na systémech s operačním systémem MS Windows se spouští pomocí „*jgc – run.bat*“, popř. „*jgc – run – lowram.bat*“
- Na systémech s operačními systémy Unix (Linux) se spouští pomocí „*jgc – run.sh*“, popř. „*jgc – run – lowram.sh*“

---

<sup>6</sup> Při spuštění aplikace na CD nebudou fungovat volby využívající zapisování na disk.

Tento způsob je preferován, pokud počítač má alespoň 512 MB RAM, protože pak je možno tuto paměť aplikací využít. V případě nedostatku RAM lze použít verzi s příponou *lowram*, která vyžaduje pouze 64 MB RAM.

Aplikaci lze také alternativně spustit ze stejného adresáře pomocí příkazu příkazové řádky / terminálu:

*java –jar jgc.jar*

U některých operačních systémů (např. u MS Windows) lze také spustit aplikaci pomocí poklepání na soubor *jgc.jar* v grafickém prostředí.

Při spuštění se mohou zadávat parametry aplikace (viz kapitola 25). Jednotlivé parametry lze zapisovat do příkazové řádky / terminálu přímo za spouštěný soubor, jako například:

*jgc –run –console*

nebo

*java –jar jgc.jar –console*

## D Obsah přiloženého CD

Obsah CD:

- *Application* – adresář s aplikací
  - *Bin* - obsahuje spustitelnou aplikaci
    - *Lib* – přiložené knihovny nutné ke spuštění k aplikaci.
  - *Src* – obsahuje zdrojové kódy k aplikaci
  - *Javadoc* – dokumentace ke zdrojovému kódu
  - *Java SDK* - obsahuje instalaci virtuálního stroje Javy (Java SE JDK 6), jehož instalace je vyžadována pro spuštění aplikace (pokud už na daný počítač nebyla Java nainstalována dříve)
- *Text*
  - *Pdf* – elektronická podoba diplomové práce ve formátu PDF
  - *Doc* – elektronická podoba diplomové práce ve formátu pro Microsoft Office Word
  - *Src* – materiály součástí elektronické podoby diplomové práce, převážně obrázky
    - *UML* – UML diagramy užité v diplomové práci

## E Fyzické přílohy diplomové práce

- CD disk s aplikací, zdrojovými kódami a textem této diplomové práce
- A3 list s UML diagramem tříd aplikace