Přemýšlecí program – piskvorky_ai

Miroslav Olšák

Obsah

1 Úvod

Hraní piškvorek na toru jsem rozdělil do dvou programů. Jeden je grafický napsaný v GTK, zobrazuje hrací plán, pozná, kdo vyhrál, atd. Pro přemýšlení počítače však volá program druhý – tento.

Grafický program takovouto pěknou dokumentaci nemá, protože je implementačně méně zajímavý, je hlavně o tom, které GTK funkce se mají zvolit, aby to běhalo a zobrazovalo to, co chceme. Za poznámku stojí, že grafický program používá i zde použitý modul `field`.

Přemýšlecí program `piskvorky_ai` je volán grafickým programem, ale je možné jej spustit i samostatně. Implicitně hraje za křížky, parametr `-O` říká, aby hrál za kolečka. Dostane-li ještě další parametr, chápe to jako název souboru, do kterého má zahrát. Program tento soubor nejprve přečte, vymyslí tah a následně do něj ve stejném formátu zapíše novou pozici.

Formát souboru předpokládá na prvním řádku dvě čísla udávající nejprve výšku a pak šířku a pak symboly `.` (prázdné políčko), `X` nebo `#` (Křížek), `O` (Ó, nikoli nula) nebo `8` (kolečko). Prázdný hrací plán tohoto formátu naleznete v souboru `pistart`.

Nedostane-li program název souboru, očekává na standardním vstupu data výše uvedeného formátu. Výstupem je pak číslo udávající index políčka, kam chce hrát.

2 Modul base

Cílem hry je získat 5 svých symbolů vedle sebe:

base.h

```
1: #define WINLEN        5
```

3 Modul field

Nejprve musíme přečíst hrací plán, uložit ho a následně s ním umět pracovat. Hrací plán implementujeme jako jednorozměrné pole `field` znaků, kde křížek je zastoupen číslem 1, kolečko číslem −1 a prázdné políčko nulou.

WINLEN: 4–5 `char field`: 2–3, 9–11

field.h
```
1: char *field;
```

Rozměry hracího plánu udávají čísla `width` (šířka), `height` (výška) a `size` (délka pole, tedy `width*height`). Kromě toho si pamatuji počet volných políček v proměnné `freesquares`.

field.h
```
2: int width, height, size;
3: int freesquares;
```

Jelikož je pole `field` jednorozměrné, mám funkce na převádění souřadnic na index (souřadnice mohou být mimo meze, torusoidně se přepočítají)

field.c
```
7: int coor2index (int x, int y)
8: {
9:   while(x < 0) x += width;
10:  while(x >= width) x -= width;
11:  while(y < 0) y += height;
12:  while(y >= height) y -= height;
13:
14:  return y*width + x;
15: }
```

a stejně tak zpátky z indexu na souřadnice.

field.c
```
17: void index2coor (int index, int *x, int *y)
18: {
19:   *x = index%width;
20:   *y = index/width;
21: }
```

Pak se ještě ukázalo být praktickým mít funkci, která ze souřadnic vrátí prvek `field` daného indexu.

field.c
```
23: char getsquare (int x, int y)
24: {
25:   while(x < 0) x += width;
26:   while(x >= width) x -= width;
27:   while(y < 0) y += height;
28:   while(y >= height) y -= height;
29:
30:   return field[y*width + x];
31: }
```

Samotné pole `field` je možné alokovat buď prázdné o daných rozměrech

field.c
```
33: void initfield (int w, int h)
34: {
35:   int i;
36:
37:   width = w;
38:   height = h;
39:   size = w*h;
40:   freesquares = size;
41:   field = (char *)malloc(size);
42:   for(i=0; i<size; i++) field[i] = 0;
43: }
```

nebo ho nechat načíst ze souboru o formátu: výška, šířka, políčka (`X`, `O`, `.`).

field.c
```
45: void loadfield (FILE *f)
46: {
47:   int ch, i;
48:
49:   fscanf(f, "%d %d", &height, &width);
50:   size = width*height;
51:   field = (char *)malloc(sizeof(char)*size);
```

`int width`: 2–3, 5 `int height`: 2–3, 5 `int size`: 2–3, 9–11 `int freesquares`: 2–3, 9–10
`int coor2index()` `void index2coor()`: 5 `char getsquare()`: 4–5 `void initfield()`
`void loadfield()`: 11

```
52:
53:   i=0;
54:   freesquares = 0;
55:   while(i < size){
56:     ch = fgetc(f);
57:     if(isspace(ch)) continue;
58:     if(ch == 'X' || ch == '#') field[i] = 1;
59:     else if(ch == '0' || ch == '8') field[i] = -1;
60:     else{
61:       field[i] = 0;
62:       freesquares++;
63:     }
64:     i++;
65:   }
66: }
```

Při zpětném ukládání do souboru vyznačím zahraný křížek symbolem # případně zahrané kolečko symbolem 8. Index tohoto políčka je uložen v proměnné `lastmove`, při čtení souboru tyto znaky opět chápu jako obyčejný křížek nebo kolečko.

field.h

```
4: int lastmove;
```

field.c

```
68: void savefield (FILE *f)
69: {
70:   int i;
71:
72:   fprintf(f, "%d %d", height, width);
73:
74:   for(i=0; i<size; i++){
75:     if(i%width == 0) fprintf(f, "\n");
76:
77:     if(field[i] < 0){
78:       if(i == lastmove) fprintf(f, "8 ");
79:       else fprintf(f, "0 ");
80:     }
81:     else if(field[i] > 0){
82:       if(i == lastmove) fprintf(f, "# ");
83:       else fprintf(f, "X ");
84:     }
85:     else fprintf(f, ". ");
86:   }
87:   fprintf(f, "\n");
88: }
```

4 Modul mvalue

Jelikož počítač není schopen promyslet hru až do konce, je třeba přiřadit dané pozici určitou hodnotu, jak moc ji budeme považovat za výhodnou. To uděláme následujícím způsobem: uvážíme všechny možné pětičky (vodorovné, svislé, úhlopříčné). Pak z nich vybereme pouze ty, ve kterých je zastoupen právě jeden druh symbolů a rozdělíme je podle toho, který druh symbolů v nich je zastoupený a kolikrát. Napíšeme si postupně: počet mých pětiček, počet soupeřových čtveřiček, počet mých čtveřiček (již ne nutně souvislých), počet soupeřových trojiček, ..., počet mých jednojiček. Tyto devítice pak porovnáváme lexikograficky.

mvalue.h

```
3: typedef struct mvalue{
4:   int a[MVAL_LEN];
5:   char attack, defense;
6: } mvalue;
```

Výše uvedená definice hodnoty pozice je pěkná, ale o tu se vlastně zas tolik nezajímám (ostatně dalo by docela práci to pokaždé spočítat) místo toho počítám hodnotu tahu jakožto hodnotu nové pozice

minus hodnota staré pozice. Ke spočítání této hodnoty již stačí prozkoumat políčka blízko táhnutého políčka.

V položce `a` je výše zmíněné pole, ovšem setřízené opačně, tj. `a[0]` udává počet mých jednojiček. Booleanová položka `attack` říká, že tah je hrozbou a položka `defense` říká, že kryje hrozbu. Tyto položky jsou využity při rozhodování se, zda prozkoumávat i další vrstvy.

Funkce `countpentad` spočítá, kolik a jak dlouhých pětiček v dané sekvenci symbolů je (předpokládá již sekvenci plnou jen jednoho druhu symbolů), ta je volána funkcí `countsquare`, která spočítá, kolik je nebráněných pětiček hráče `player`, která zasahují do daného políčka. Konečně funkce `val_get` na základě `countsquare` spočítá hodnotu tahu daným hráčem na dané políčko. Návratové hodnoty funkcí `countpentad` a `countsquare` slouží k určení položek `attack` a `defense`. Bylo by možné tyto hodnoty určit přesněji, například tah křížku doprostřed `O X X . . . . . X X O` je považován za útok, ale vypilovávat mi to přijde zbytečné, zvlášť když by to bylo na úkor časové náročnosti výpočtu.

mvalue.c

```
12: static char countpentad (char *src, int len, char *dest)
13: {
14:   int i, num;
15:
16:   num = 0;
17:   if(len < WINLEN) return 0;
18:   for(i=0; i<WINLEN; i++) if(src[i]) num++;
19:   dest[num]++;
20:   for(; i<len; i++){
21:     if(src[i-WINLEN]) num--;
22:     if(src[i]) num++;
23:     dest[num]++;
24:   }
25:   if(dest[WINLEN-1] || dest[WINLEN-2] > 1) return 3;
26:   if(dest[WINLEN-2] || dest[WINLEN-3] > 1) return 1;
27:
28:   return 0;
29: }
30:
31: static char countsquare (int x, int y, char player, char *dest)
32: {
33:   char row[2*WINLEN-1];
34:   int i, len, action;
35:
36:   player = -player;
37:   for(i=0; i < WINLEN; i++) dest[i] = 0;
38:
39:   action = 0;
40:
41:   if(row_e){
42:     for(i = -1; i > -WINLEN && player != getsquare(x+i, y); i--);
43:     i++;
44:     for(len = 0; i+len < WINLEN && player != (row[len] = getsquare(x+(i+len), y)); len++);
45:     action |= countpentad(row, len, dest);
46:   }
47:   if(col_e){
48:     for(i = -1; i > -WINLEN && player != getsquare(x, y+i); i--);
49:     i++;
50:     for(len = 0; i+len < WINLEN && player != (row[len] = getsquare(x, y+(i+len))); len++);
51:     action |= countpentad(row, len, dest);
52:   }
53:   if(diag_e){
54:     for(i = -1; i > -WINLEN && player != getsquare(x+i, y+i); i--);
55:     i++;
56:     for(len = 0; i+len < WINLEN && player != (row[len] = getsquare(x+(i+len), y+(i+len))); len++);
57:     action |= countpentad(row, len, dest);
58:
59:     for(i = -1; i > -WINLEN && player != getsquare(x-i, y+i); i--);
60:     i++;
```

char countpentad(): 4–5 char countsquare(): 4–5 char player: 4–5 void val_get(): 5–6, 8–9

```
 61:     for(len = 0; i+len < WINLEN && player != (row[len] = getsquare(x-(i+len), y+(i+len))); len++);
 62:     action |= countpentad(row, len, dest);
 63:   }
 64:
 65:   return action;
 66: }
 67:
 68: void val_get (int index, char player, mvalue *dest)
 69: {
 70:   char attack[WINLEN], defense[WINLEN];
 71:   int i, x, y;
 72:
 73:   index2coor(index, &x, &y);
 74:
 75:   dest->attack = countsquare(x, y, player, attack);
 76:   dest->defense = (countsquare(x, y, -player, defense) == 3);
 77:
 78:   for(i=0; i<WINLEN-1; i++) dest->a[2*i] = attack[i]-attack[i+1];
 79:   dest->a[2*i] = attack[i];
 80:   for(i=1; i<WINLEN; i++) dest->a[2*i-1] = defense[i];
 81: }
```

Na začátku programu je volána `val_init`. Ta

1) Inicializuje náhodný generátor
2) Inicializuje hodnoty `val_minf` ($-\infty$), `val_pinf` ($+\infty$) a `val_neut` (0).
3) Podívá se na velikost toru a podle toho nastaví booleany `row_e`, `col_e`, `diag_e`, které udávají, jestli má smysl zkoumat pětičky v řádcích / sloupcích / úhlopříčkách. (např. má-li torus výšku 4, není možné udělat svislou čtveřičku).

mvalue.c

```
155: void val_init ()
156: {
157:   int i;
158:
159:   row_e = col_e = diag_e = 0;
160:   if(width >= WINLEN) row_e = 1;
161:   if(height >= WINLEN) col_e = 1;
162:   if(width/gcd(width, height)*height >= WINLEN) diag_e = 1;
163:
164:   srand(time(NULL));
165:
166:   for(i=0; i<MVAL_LEN; i++){
167:     val_pinf.a[i] = INT_MAX/2;
168:     val_minf.a[i] = INT_MIN/2;
169:     val_neut.a[i] = 0;
170:   }
171: }
```

Pro práci se strukturou `mvalue` mám následující funkce:

Pomocí `val_cmp` porovnávám standardním způsobem dvě hodnoty, jinými slovy (ne)rovnítko mezi `val_cmp()` a `0` bude stejné jako mezi parametry `v1` a `v2`.

mvalue.c

```
131: int val_cmp (const mvalue *v1, const mvalue *v2)
132: {
133:   int i;
134:
135:   for(i = MVAL_LEN-1; i>=0; i--){
136:     if(v1->a[i] > v2->a[i]) return 1;
137:     if(v1->a[i] < v2->a[i]) return -1;
138:   }
139:
140:   return 0;
```

```
141: }
```

Funkce `val_invert` změní znaménko. Je používána v negamaxovém algoritmu, je proto důležité, aby platilo, že mezi hodnotami s opačným znaménkem je opačná nerovnost než mezi hodnotami se znaménkem původním.

mvalue.c

```
 83: void val_invert (mvalue *v)
 84: {
 85:   int i;
 86:
 87:   for(i=0; i<MVAL_LEN; i++) v->a[i] *= -1;
 88: }
```

Všechny tyto hodnoty chci počítat jen z pohledu jednoho hráče. Ovšem funkce `val_get` počítá vždy z hlediska hráče, který je na tahu. Funkce `val_swapplayers` přeuspořádá pole tak, že nová hodnota bude z pohledu druhého hráče, ale s opačným znaménkem (pro negamax se mi to hodí).

mvalue.c

```
 90: void val_swapplayers (mvalue *v)
 91: {
 92:   int i, tmp;
 93:
 94:   for(i=0; i<MVAL_LEN-1; i+=2){
 95:     tmp = v->a[i];
 96:     v->a[i] = v->a[i+1];
 97:     v->a[i+1] = tmp;
 98:   }
 99: }
```

Pomocí funkce `val_add` přičtu hodnotu `a` k hodnotě `v` a získám tak složenou hodnotu více tahů.

mvalue.c

```
106: void val_add (mvalue *v, mvalue *a)
107: {
108:   int i;
109:
110:   for(i=0; i<MVAL_LEN; i++)
111:     v->a[i] += a->a[i];
112: }
```

Funkce `val_isinf` odpoví, zda je daná hodnota výherní (pět symbolů v řadě). Porovnání s hodnotou `val_pinf` neposlouží, neboť to je nekonečno ještě mnohem vyšší.

mvalue.c

```
114: char val_isinf (mvalue *v)
115: {
116:   return v->a[MVAL_LEN-1] > 0;
117: }
```

Funkce `val_randomize` přičte k méně významným číslům hodnoty tahu náhodná čísla, čímž se snažím docílit ne zcela deterministické chování.

mvalue.c

```
173: void val_randomize (mvalue *v)
174: {
175:   v->a[0] += rand()%10;
176:   v->a[1] += rand()%6;
177:   v->a[2] += rand()%4;
178:   v->a[3] += rand()%2;
179: }
```

Pro ladící účely se občas hodí si hodnotu vytisknout.

mvalue.c

```
119: void val_print (mvalue *v)
120: {
121:   int i;
122:
123:   printf("(");
124:   for(i=0; i<MVAL_LEN; i++){
125:     if(i) printf(", ");
```

```
126:     printf("%d", v->a[i]);
127:   }
128:   printf(")\n");
129: }
```

Funkce `val_plusplus` převede hodnotu na nejmenší možnou vyšší hodnotu. V kódu ji nepoužívám. Hodila se, když jsem zkoušel negascout, ale ukázalo se, že klasická alphabeta je rychlejší.

mvalue.c

```
101: void val_plusplus (mvalue *v)
102: {
103:   v->a[0]++;
104: }
```

Nakonec mám (opět pro ladící účely) funkci, která mi jednoduše vytvoří hodnotu na přání.

mvalue.c

```
181: mvalue val_create (int a0, int a1, int a2, int a3, int a4,
182:            int a5, int a6, int a7, int a8)
183: {
184:   mvalue result;
185:
186:   result.a[0] = a0;
187:   result.a[1] = a1;
188:   result.a[2] = a2;
189:   result.a[3] = a3;
190:   result.a[4] = a4;
191:   result.a[5] = a5;
192:   result.a[6] = a6;
193:   result.a[7] = a7;
194:   result.a[8] = a8;
195:
196:   return result;
197: }
```

5 Modul alphabeta

Již ohodnocovací funkce (zahraji-li na nejlépe ohodnocené políčko) sama o sobě hraje vcelku dobře. Například:

- Pokud může, vyhraje.
- Zablokuje čtveřičku.
- Zahraje vidličku.
- Zablokuje vidličku.
- Vyrobí neblokovanou trojičku.

Jsou však situace, kdy je takový tah vyloženě špatný (hraje za kolečko tam, kde je nakreslena osmička):

```
. . . . . . . . . .
. . . . . . . 8 . .
. . . . . . . . . .
. . . . . . . . . .
. . . . O . . . . .
. . . O X . . . . .
. . X . X . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

Mám tedy implementované přemýšlení dopředu. Základ je alphabeta minimax (popsáno na Wikipedii). Začínám v hloubce 0, ponořuji se do vyšších hloubek. Počítám v průběhu relativní hodnoty pozic, ale výsledek je stejný, jako bych v poslední vrstvě vždy spočítal hodnotu příslušné pozice. Počítám to

vzhledem k hráči, který hraje v předposlední vrstvě, snažím se, aby, i když ukončuji v dřívější vrstvě, byla parita vrstvy stejná (na základě toho vracím nejlepší tah soupeře nebo `val_neut`). Navíc k tomu mám další svá heuristická vylepšení:

- Zkouším jen tahy, co nejsou přehnaně daleko od současných symbolů. Konkrétně hodnota tahu musí být vyšší než taková, co má na pozici 1 hodnotu 3.
- Jsem-li za hloubkou 3, zkoumám jen tahy, kde se útočí nebo brání. Bylo by ovšem hloupé nutit hráče útočit, pokud se mu to nevyplatí. Proto, pokud hráč nemusí bránit, dovoluji mu ukončit větev.
- Tahy zkouším v pořadí od nejlepšího, jak mi je ohodnotila `val_get`. (Alphabetě to pomáhá.)
- V nulové hloubce náhodně trochu upravím hodnoty tahů.
- Hrozí-li mi soupeř čtveřičkou, odpovídám rychle a nepočítám hodnotu mého tahu.
- U programu prohledávajícího do hloubky nemůžu předpokládat, že bude přemýšlet rychle. Proto každých pět sekund snižuji hloubku propočtu o 2 (abych se na situaci díval stále stejným hráčem). Počáteční hloubka propočtu (definovaná jako `MAXDEPTH`) je 11, tedy program by neměl přemýšlet déle než půl minuty.

alphabeta.c
```
12: #define MAX_DEPTH     11
```

Funkce `alphabeta` vrací negamaxovou hodnotu pro hráče hrajícího za `player`, oříznutý hodnotami `alpha`, `beta`. Do pole `bestbranch` uloží nejlepší posloupnost tahů, co vymyslel ukončenou minus jedničkou (tohle by šlo optimalizovat, aby vracel jen nejlepší políčko v hloubce 0, ale pro ladící účely se to hodí).

alphabeta.c
```
39: static mvalue alphabeta (int depth, char player, mvalue alpha, mvalue beta, int *bestbranch)
40: {
41:   int i, j, s;
42:   time_t t;
43:   int mnum;  // pocet prozkoumavanych policek
44:   mvalue v, nalpha, nbeta;
45:   mvalue *vals;
46:   int *squares;
47:   int branch[MAX_DEPTH+1];
48:   char defense; // je treba branit?
```

Nejprve otestuji, jak na tom jsem s časem. čas kontroluji každých 100 zavolání `alphabeta`. Definice `COUNTER_LEN` udává právě onu stovku, definice `DEGRAD_TIME` říká, že mám snižovat hloubku každých 5 sekund.

alphabeta.c
```
13: #define COUNTER_LEN  100
14: #define DEGRAD_TIME    5
```

Proměnná `counter` počítá spuštění `alphabeta`, aktuální maximální hloubka propočtu je uložena v proměnné `max_depth`. Poslední čas, kdy jsem začínal nebo snižoval hloubku, je uložen v `last_time`.

alphabeta.c
```
16: static int max_depth, counter;
```

Samotná funkce `alphabeta` pokračuje takto:

alphabeta.c
```
50:   counter++;
51:   if(counter == COUNTER_LEN){  // snizuji hloubku v pripade dlouheho vypoctu
52:     counter = 0;
53:     t = time(NULL);
54:     if(t >= last_time + DEGRAD_TIME){
55:       last_time = t;
56:       if(max_depth >= 2) max_depth -= 2;
57:     }
58:   }
```

Dále pro jistotu ukončím `bestbranch`.

alphabeta.c
```
60:   bestbranch[depth] = -1;
```

MAXDEPTH mvalue alphabeta(): 8–10 int* bestbranch: 8–10 COUNTER_LEN: 8 DEGRAD_TIME: 8
int counter: 8, 10 int max_depth: 8–10 time_t last_time: 8, 10

A otestuji, jestli mi nedošla hloubka nebo políčka.

alphabeta.c
```
63:   if(depth == max_depth || !freesquares) return val_neut;
```

Pokud ne, tak si alokuji pole `squares` indexů políček, kam má smysl hrát a pole `vals` hodnot příslušných tahů.

alphabeta.c
```
65:   vals = alloca(size * sizeof(mvalue)); // hodnoty policek
66:   squares = alloca(size * sizeof(int)); // indexy policek
```

Z dobrých důvodů budu indexy políček sypat na začátek pole, zatímco hodnoty budu umisťovat na místa indexů daných políček. Projdu všechna políčka, počet zajímavých políček mám v proměnné `mnum`, nejlépe ohodnoceného políčka v proměnné `v`. V proměnné `defense` mám uloženo, zda jsem někdy musel bránit.

alphabeta.c
```
68:   defense = 0;
69:   v = val_minf;
70:   for(i=mnum=0; i < size; i++){
71:     if(field[i]) continue;  // neprazdne policko
72:
73:     val_get(i, player, &vals[i]);
74:
75:     defense |= vals[i].defense;
76:     if(val_isinf(&vals[i])){   // vyhravam
77:       bestbranch[depth] = i;
78:       bestbranch[depth+1] = -1;
79:       return vals[i];
80:     }
```

Přitom funkce `ok_move`, která vrací, zda je daný tah vhodné zkoumat, je definovaná takto (podle prvních dvou puntíků na stránce 8):

alphabeta.c
```
11: #define AI_ACTION      3
```

alphabeta.c
```
19: static char ok_move (int depth, mvalue *v)
20: {
21:   int i;
22:
23:   if(depth >= AI_ACTION){
24:     if(v->attack || v->defense) return 1;
25:     return 0;
26:   }
27:   if(v->a[1] >= 3) return 1;
28:   for(i=2; i<MVAL_LEN; i++) if(v->a[i]) return 1;
29:   return 0;
30: }
```

Zpět ale k funkci `alphabeta`, může být ještě pár důvodů, proč funkci ukončit.

alphabeta.c
```
94:   if(!mnum || depth > max_depth){     // nic zajimaveho, koncime
95:     if((MAX_DEPTH - depth)%2) return v;
96:     else return val_neut;
97:   }
98:
99:   if(depth == 0 && v.a[MVAL_LEN-2] > 0) return v; // reakce na ctvericku
100:
101:  if(depth >= AI_ACTION && !defense && val_cmp(&val_neut, &alpha) > 0){
102:    // nechci vynucovat utok
103:    bestbranch[depth] = -1;
104:    alpha = val_neut;
105:    if(val_cmp(&alpha, &beta) >= 0) return alpha;  // odstřihnutí betou
106:  }
```

Ale pak už musím setřídit políčka

---

alphabeta.c

```
108:   cur_vals = vals; // predavka globalni promenne kvuli qsortu
109:   qsort(squares, mnum, sizeof(int), squarecmp);
```

a jít se rekurzit.

alphabeta.c

```
111:   for(i=0; i < mnum; i++){
112:     s = squares[i];
113:
114:     field[s] = player;  // hraju na policko
115:     freesquares--;
116:
117:     if((MAX_DEPTH - depth) % 2 == 0) val_swapplayers(&vals[s]);
118:
119:     nalpha = beta;   // negamaxove sarady
120:     nbeta = alpha;
121:     val_invert(&nbeta);
122:     val_invert(&nalpha);
123:     val_add(&nalpha, &vals[s]);
124:     val_add(&nbeta, &vals[s]);
125:
126:     v = alphabeta(depth+1, -player, nalpha, nbeta, branch); // rekurzim se
127:     val_invert(&v);
128:     val_add(&v, &vals[s]);
129:
130:     freesquares++;  // vracim tah
131:     field[s] = 0;
132:
133:     if(val_cmp(&v, &alpha) > 0){  // zatim nejlepsi tah
134:       for(j = depth+1; branch[j] >= 0; j++)
135:         bestbranch[j] = branch[j];           // zkopiruji vetev
136:       bestbranch[j] = -1;
137:       bestbranch[depth] = s;
138:
139:       alpha = v;
140:       if(val_cmp(&alpha, &beta) >= 0 || val_isinf(&alpha)) return alpha; // odstřihnutí betou
141:     }
142:   }
```

A nakonec jako každá správná alphabeta vracím `alpha`.

alphabeta.c

```
144:   return alpha;
145: }
```

Pro porovnání políček (abych měl `qsortu`, co předat) mám funkci `squarecmp`.

alphabeta.c

```
 32: static mvalue *cur_vals;
 33:
 34: static int squarecmp (const void *s1, const void *s2)
 35: {
 36:   return -val_cmp(&cur_vals[*(int*)s1], &cur_vals[*(int*)s2]);
 37: }
```

Funkci `alphabeta` pro lepší komfort obaluje (volá) funkce `ai_move`, která vrací index tahu, kam chce počítač zahrát.

alphabeta.c

```
147: int ai_move (char player)
148: {
149:   int i;
150:   int branch[MAX_DEPTH+1];
151:
152:   counter = 0;
153:
154:   max_depth = MAX_DEPTH;
155:   last_time = time(NULL);
156:
157:   alphabeta(0, player, val_minf, val_pinf, branch);
```

```
158:
159:   return branch[0];
```

Zakomentuje-li se tento return, spustí se následující kód, který způsobí, že počítač nezahraje, ale vypíše, jak si představuje ideální posloupnost tahů (kde oba hráči hrají nejlépe).

alphabeta.c

```
161:   for(i = 0; branch[i] >= 0; i++){
162:     if(i%2) field[branch[i]] = -player;
163:     else field[branch[i]] = player;
164:     savefield(stdout);
165:     printf("%d\n", branch[i]);
166:     printf("\n");
167:   }
168:
169:   return -1;
170: }
```

6 Modul piskvorky_ai

Nakonec to všechno obalím ve funkci `main`. Ta na základě parametrů příkazové řádky určí, který hráč má hrát a zdali je vstupem a výstupem soubor nebo má být vstup čten ze `stdin` a výslednému tahu má být jen vypsán index.

piskvorky_ai.c

```
  9: int main (int argc, char **argv)
 10: {
 11:   FILE *f;
 12:   char *filename;
 13:   char player;
 14:   int i;
 15:
 16:   player = 1;
 17:   filename = NULL;
 18:
 19:   for(i=1; i < argc; i++){
 20:     if(argc >= 2 && !strcmp(argv[i], "-O") && player == 1) player = -1;
 21:     else if(i == argc-1) filename = argv[i];
 22:     else{
 23:       fprintf(stderr, "Usage: piskvorky_ai [-O] [filename]\n");
 24:       return 1;
 25:     }
 26:   }
 27:
 28:   if(filename){
 29:     if(!(f = fopen(filename, "r"))){
 30:       fprintf(stderr, "Reading file '%s' failed.\n", filename);
 31:       return 1;
 32:     }
 33:   }
 34:   else f = stdin;
 35:
 36:   loadfield(f);
 37:   fclose(f);
 38:
 39:   val_init();
 40:
 41:   lastmove = ai_move(player);
 42:
 43:   if(lastmove >= 0 && lastmove < size){
 44:     if(filename){
 45:       field[lastmove] = player;
 46:       if(!(f = fopen("hra", "w"))){
 47:         fprintf(stderr, "Writing file '%s' failed.\n", filename);
 48:         return 1;
 49:       }
```

```
50:       savefield(f);
51:       fclose(f);
52:     }
53:     else printf("%d\n", lastmove);
54:   }
55:
56:   return 0;
57: }
```

7 Rejstřík