3. Implementarea aplicației

În acest capitol vor fi discutate aspectele ce țin de implementarea aplicației prin prezentarea unor secvențe de cod a algoritmilor, structurilor de date și claselor implicate în alcătuirea sistemului.

3.1. Motorul de șah

Considerări generale pentru reprezentarea unei poziții* de șah[1]:

a. reținerea pozițiilor pentru fiecare piesă de pe tablă;
b. reținerea culorii ce trebuie să mute;
c. monitorizarea mișcării regilor și turelor, utilă pentru rocadă;
d. reținerea ultimei mutări efectuate, determină existența unei capturări de tip *en passant*;
e. contorizarea numărului de mutări de la ultima mutare a unui pion sau ultima capturare a unei piese, utilizată pentru regula celor 50 de mutări;
f. o cheie unică de identificare a poziției de joc.

Pentru reprezentarea acestor date s-a folosit o structură de date *S_BOARD* ce reține informațiile necesare unei poziții de șah(figura 21).

BRD_SQ_NUM reprezintă numărul total de pătrate ce se reprezintă în memorie.

```
typedef struct {

     int pieces[BRD_SQ_NUM];
     int side;
     int castlePerm;
     int enPas;
     int fiftyMove;
     U64 posKey;
} S_BOARD;
```

Figura 21
Strutura de date S_BOARD

* modul în care este organizată tabla de șah la un moment dat

3. Implementarea aplicației

Scopul cheii unice de identificare a poziției de joc este de oferi posibilitatea de verificare a repetiției unei poziții de joc[7]. Fiecare piesă se poate afla pe oricare dintre cele 64 de pătrate ale tablei de șah. Având în vedere că sunt 12 piese putem genera 64 x 12 = 768 de numere aleatoare corespunzătoare fiecărei piese de șah pentru fiecare pătrat de pe tablă. Astfel, folosind operația logică binară „sau exclusiv”[1], putem include sau exclude din cheia tablei numărul aleatoriu corespunzător piesei de pe pătratul pe care se află. Pentru a genera cheia unică de identificare trebuie să includem, folosind operația precizată, pentru toate piesele de pe tablă numărul aleatoriu corespunzător în cheia poziției și astfel vom obține o cheie unică de identificare.

Mutările ce sunt generate pe parcursul căutării și evaluării lor trebuie reținute într-o structură de date pentru a fi analizate ulterior. În acest scop se folosește structura de date *S_MOVELIST*(figura 22), care este o structură simplă ce conține un șir de mutări. O mutare trebuie salvată împreună cu un scor pe care aceasta îl are, de aceea se folosește structura de date *S_MOVE*(figura 23) pentru stocarea unei mutări.

```
typedef struct {

     int move;
     int score;

} S_MOVE;
```

Figura 23
Structura de date S_MOVE

```
typedef struct {

     S_MOVE moves[MAXPOSITIONMOVES];
     int count;

} S_MOVELIST;
```

Figura 22
Structura de date S_MOVELIST

Structura *S_MOVELIST*, pe lângă mulțimea de mutări, conține și un indicator al numărului de mutări prezente în mulțime. După cum observăm în figura de mai sus o mutare este stocată ca și un număr întreg pe 32 de biți. Se folosesc 25 de biți din cei 32 disponibili pentru a stoca informațiile necesare despre o mutare după cum urmează[7]:

i. primii 7 biți reprezintă numărul pătratului de unde se mută piesa; numărul

de pătrate disponibile(incluzând cele folosite pentru a detecta dacă piesa este încă pe tablă) este 120, având în vedere că 7 biți permit o valoare maximă de 127 atunci dimensiunea rezervată este suficientă;

ii. următorii 7 biți reprezintă numărul pătratului unde se mută piesa;
iii. tipul piesei capturate se stochează în următorii 4 biți, reprezentând o valoare maximă de 15 care este suficientă pentru a reține unul din cele 12 tipuri de piese de șah;
iv. bitul cu numărul 19 este folosit pentru a stoca dacă s-a produs o capturare de tip en passant;
v. următorul bit este fi folosit pentru a verifica dacă un pion a avansat cu două câmpuri;
vi. următorii 4 biți sunt folosiți pentru a identifica, în cazul în care s-a produs o promovare, tipul piesei la care pionul a fost promovat;
vii. iar bitul 25, ultimul folosit din cei 32 disponibili, este folosit pentru a semnala dacă s-a produs o rocadă.

Pentru a prelua datele din mutare se folosesc operații pe biți de tip shift, „sau” logic, „și” logic. Se vor crea fragmente de cod(MACRO) ce execută aceste operații pentru a ușura folosirea lor.

O structură importantă ce trebuie menționată este structura de date de tip enumerare ce conține tipurile de piese existente pe tabla de șah plus un element care poate identifica dacă un pătrat nu conține nicio piesă, adică este gol. Această structură este descrisă în figura de mai jos(figura 24).

```
enum {

      WHITE,
      BLACK,
      BOTH

};
```

Figura 25
Enumerare tipuri de jucători

```
enum {

      EMPTY,
      wP, wN, wB, wR, wQ, wK,
      bP, bN, bB, bR, bQ, bK

};
```

Figura 24
Enumerare tipuri de piese

Tipurile de piese sunt enumerate pentru fiecare jucător în parte, astfel ajungând la 12 piese. Pentru identificare culorii jucătorului este folosită o enumerare(figura 25) ce include și un element folosit în scopul de a salva informații ce sunt necesare pentru ambii jucători.

În timpul generării mutărilor de șah există necesitatea anulării unui număr de mutări pentru a reveni la o poziție anterioară. Pentru a îndeplini această necesitate implementarea algoritmului de căutare folosește structura *S_UNDO*(figura 26) care conține informațiile necesare pentru revenirea la o poziție anterioară: mutarea făcută, permisiunile de rocadă, numărul pătratului en passant, numărul de mutări efectuate de la ultima capturare sau mutare a unui pion și o cheie unică de identificare a unei table. Pentru a permite anularea mai multor mutări structurii *S_BOARD* i se mai adaugă o componentă: o mulțime de structuri de date tip *S_UNDO*. Astfel motorul de căutare va putea anula un număr de mutări necesare pentru a reveni la o poziție anterioară din care poate genera alte mutări.

```
typedef struct {

     int move;
     int castlePerm;
     int enPas;
     int fiftyMove;
     U64 posKey;

} S_UNDO;
```

Figura 26

Structura de date S_UNDO

Având structurile de date necesare pentru căutare definite și inițializate se poate porni o căutare a unei mutări de șah pentru o anumită poziție. Funcția principală ce reprezintă punctul de intrare în căutare este funcție de căutare în adâncime progresivă[7](figura 27). Rolul acestei funcții este de a găsi cea mai bună mutare pe baza funcțiilor de evaluare existente.

```
void SearchPositions(S_BOARD *pos) {
     int bestMove = NOMOVE;
     int bestScore = -INFINITE;
     // iterare progresivă a adâncimii
     for (currentDepth = 1; currentDepth <= max_depth; +
+currentDepth) {
          bestScore = NegaMax(-INFINITE, INFINITE, currentDepth,
pos, info, TRUE);
          pvMoves = GetPvLine(currentDepth, pos);
          bestMove = pos->PvArray[0];
     }
```

Figura 27

Funcția de căutare în adâncime progresivă

3. Implementarea aplicației

Iterarea progresivă a adâncimii este realizată pornind de la o adâncime 1 până la limita maximă de căutare[1], care pentru o căutare în timp util este mai mică sau egală cu 7. Pentru o căutare cu o configurare a poziției de șah egală cu „r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 1” o căutare cu o adâncime maximă de 6 se generează 8031647685 de mutări posibile după ce algoritmul a suferit modificările necesare de optimizare. Algoritmul ce urmează a fi detaliat este *NegaMax* cu tăieri alfa beta(figura 28). Rolul tăierilor alfa beta este de a reduce numărul de noduri ce sunt evaluate de negamax în arborele de căutare generat. Aceste tăieri funcționează prin oprirea completă a evaluării unui nod în momentul în care când a fost găsită cel puțin o posibilitate în care mutarea se dovedește a fi mai slabă față de mutarea anterioară.

```
int NegaMax(int alpha, int beta, int depth, S_BOARD *pos) {
      int legal;
      S_MOVELIST list[1];
      GenerateAllMoves(pos, list);
      for (moveNum = 0; moveNum < list->count; ++moveNum) {
            PickNextMove(moveNum, list);
            if (!MakeMove(pos, list->moves[moveNum].move)) {
                  continue;
            }
            legal++;
            score = -NegaMax(-beta, -alpha, depth - 1, pos);
            TakeMove(pos);
            if (score > alpha) {
                  if (score >= beta) {
                        return beta;
                  }
                  alpha = score;
```

```
                bestMove = list->moves[moveNum].move;
            }
        }
        if (legal == 0) {
            if (SqAttacked(pos->KingSq[pos->side], pos->side ^ 1,
pos)) {
                return -MATE + pos->ply;
            } else {
                return 0;
            }
        }
        if (alpha != oldAlpha) {
            StorePvMove(pos, bestMove);
        }
        return alpha;
}
```

Figura 28

Funcția NegaMax cu tăieri alfa beta

Primul pas în algoritm este generarea tuturor mutărilor disponibile din poziția ce este evaluată, urmat de parcurgerea mutărilor una câte una. Dacă o mutare este considerată legală atunci valorile utilizate pentru tăiere alfa și beta sunt actualizate. După ce au fost iterate toate mutările există posibilitatea ca să nu se fi găsit nicio mutare legală, în acest moment se folosește indicatorul *pos→ply* și scorul poziției de șah mat pentru a indica numărul de mutări rămase până la șah mat. Dacă jucătorul alege să facă această mutare atunci i se va returna scorul pentru șah mah din care se scade numărul de mutări în care se ajunge la mat. În concluzie, se poate constata că jucătorul a pierdut în momentul în care scorul returnat este scorul maxim șah mat(mai sunt 0 mutări până la șah mat).

Generarea mutărilor constă în parcurgerea tuturor pieselor de pe tabla de șah și adăugarea fiecărei mutări posibile în mulțimea din S_MOVELIST. Se observă că mutările nu sunt parcurse în ordinea lor din mulțime, alegerea următoarei mutări ce trebuie analizată se face cu ajutorul unei funcției *PickNextMove*. Această funcție are rolul de a selecta din mulțimea de mutări generate mutarea cu cel mai mare scor.

Funcția ce generează mutările folosește un concept simplu pentru a asocia fiecărei mutări câte un scor, principiul este „cea mai valoroasă captură cu cel mai puțin valoros atacator”[7](MvvLva – most valuable victim least valuable attacker). Acest principiu este implementat folosind un vector care conține câte un scor pentru fiecare victimă și atacator. Astfel avem vectorul *MvvLva*(figura 29) care conține elemente de forma:

```
// primul indice reprezintă victima, iar al doilea reprezintă atacatorul
MvvLva[regină][pion] = 505;
MvvLva[regină][cal] = 504;
…
MvvLva[regină][regină] = 500;
```

Figura 29
Vectorul MvvLva

Folosind acest vector ne asigurăm ca atunci când inserăm în lista de mutări o mutare ce reprezintă o captură ea va avea scorul corespunzător astfel încât în momentul în care ea trebuie evaluată să se prefere mutarea care reprezintă „cea mai bună” captură. Următoarea optimizare ce trebuie discutată este acordarea de prioritate pentru evaluarea mutărilor ce reprezintă capturări, aceasta se realizează adăugând pentru fiecare tip de mutare ce reprezintă o captură, pe lângă scorul din vectorul MvvLva, un punctaj foarte mare. Astfel ne asigurăm că mutările ce reprezintă capturi vor fi evaluate primele.