Tehnici de programare - TP

Cursul 12 – Greedy. Divide et Impera. Backtracking

Ș.l. dr. ing. Cătălin Iapă

catalin.iapa@cs.upt.ro

De data trecută: Recursivitatea

Greedy

Divide et impera

Backtracking

Exerciții

Recursivitate

În C, recursivitatea se realizează prin intermediul funcțiilor, care se pot *autoapela*.

O funcție trebuie definită iar apoi se poate apela.

Recursivitatea constă în faptul că în definiția unei funcții apare apelul ei însăși. Acest apel, care apare în însăși definiția funcției, se numește autoapel.

Primul apel, făcut în altă funcție, se numește apel principal.

Recursivitate

Funcția factorial implementată iterativ (nerecursiv)

```
int fact(int n){
    int p = 1, i;
    for(i = 1 ; i <= n ; i ++)
        p = p * i;
    return p;
}
```

Funcția factorial implementată recursiv

```
int fact(int n){
    if(n == 0)
        return 1;
    else
        return n * fact(n-1);
}
```

De data trecută: Recursivitatea

Greedy

Divide et impera

Backtracking

Exerciții

Metode generale de programare

În teoria şi practica programării există un număr foarte mare de probleme pentru care trebuie găsite rezolvări. Din totalitatea acestor probleme se disting clase de probleme similare. Pentru problemele dintr-o asemenea clasă se poate aplica aceeaşi metodă generală de rezolvare, evident cu mici ajustări care depind de problema concretă.

În timp s-au cristalizat mai multe metode generale de rezolvare a problemelor. Programatorii experimentaţi stăpânesc foarte bine aceste metode generale şi le aplică în mod automat, ca nişte scheme de programare, atunci când au posibilitatea.

Greedy

În limba engleză cuvântul greedy înseamnă lacom.

Algoritmii de tip greedy sunt algoritmi “lacomi”: ei urmăresc să construiască într-un mod cât mai rapid soluţia problemei.

Algoritmii de tip greedy se caracterizează prin luarea unor decizii rapide care duc la găsirea unei soluţii a problemei.

Nu întotdeauna asemenea decizii rapide conduc la o soluţie optimă, dar vom vedea că există anumite tipuri de probleme unde se pot obţine soluţii optime sau foarte apropiate de optim.

Greedy - principii

Metoda **Greedy** se aplică la acele probleme la care, dându-se mulţimea *A* a datelor de intrare, se cere să se determine o submulţime *B* a sa, care trebuie să îndeplinească anumite condiţii pentru a fi acceptată ca **soluţie posibilă**.

În general, există mai multe soluţii posibile. Dintre acestea se pot selecta, conform unui anumit criteriu, nişte submulţimi B* care reprezintă **soluţii optime** ale problemei. Scopul este acela de a găsi una dintre mulţimile B*. Dacă acest lucru nu este posibil, atunci scopul este găsirea unei mulţimi B care să fie cât mai aproape de mulţimile B*, conform criteriului de optimalitate impus. Soluţiile posibile au următoarea proprietate:

- dacă B este o soluţie posibilă atunci orice submulţime a sa, inclusiv mulţimea vidă este, de asemenea, soluţie posibilă.

Greedy - principii

Din proprietatea de mai sus rezultă și modul practic, de construire, al mulţimii *B:* Iniţial se porneşte cu mulţimea vidă (*B* = Φ). Urmează un şir de *decizii*. Fiecare decizie constă în alegerea unui element din mulţimea *A*, analizarea lui şi eventual introducerea în mulţimea *B*. În funcţie de modul în care se iau aceste decizii, mulţimea *B* se va apropia mai mult sau mai puţin de soluţia optimă *B**. În cazul ideal vom avea *B = B**.

La fiecare pas se analizează câte un element din mulţimea A şi se decide dacă să fie sau nu inclus în mulţimea *B* care se construieşte. Astfel se progresează de la Φ cu un sir de mulţimi intermediare (Φ, *B0*, *B1*, *B2*, ...), până când se obţine o soluţie finală *B*.

Greedy – Strategii

Metoda Greedy **nu** urmăreşte să determine toate soluţiile posibile ca să aleagă apoi pe cea optimă, conform criteriului de optimizare dat. O astfel de metodă, care ar face **căutare exhaustivă** în spaţiul soluţiilor și care ar determina, *cu exactitate*, soluția optimă necesită, de regulă, un efort de calcul foarte mare. Metoda Greedy, în schimb, nu necesită nici timp de calcul, nici spaţiu de memorie mare, comparativ cu metodele exacte.

Pe baza proprietăţii enunţate anterior, **la fiecare pas al metodei există o soluţie posibilă care se va îmbogăţi cu un nou element**, ales astfel încât **şirul soluţiilor posibile să conveargă spre soluţia optimă**.

Noua soluţie "înghite" elementul cel mai "promiţător".

Greedy – Strategia 1

Există mai multe strategii prin care se poate implementa metoda Greedy. În continuare se prezintă două astfel de strategii care diferă prin ordinea de efectuare a unor operaţii.

Se consideră că mulţimea datelor de intrare, *A*, are iniţial *n* elemente. *B* reprezintă, în orice moment, soluţia posibilă. Funcţia logică posibil are valoarea true dacă elementul selectat, transmis ca parametru, formează împreună cu mulţimea *B* o nouă *soluţie posibilă*. Verificările efectuate în această funcţie trebuie să rezulte din enunţul problemei.

Iniţial, *B* este mulţimea vidă. La fiecare pas al algoritmului se alege, într-un anumit fel, un element din *A* neales la paşii precedenţi (funcţia alege). Se adaugă acest nou element la soluţia posibilă anterioară, dacă prin această adăugare se obţine tot o soluţie posibilă

Greedy – Strategia 1

```
B = multimea vida;
for (i=0; i<n; i++) {
        x = alege( A);
        if (posibil( B ,x))
        * adauga elementul x la multimea B;
        }
```

Dificultatea la această primă variantă constă în scrierea funcţiei alege. Dacă funcţia alege este bine concepută, atunci putem fi siguri că soluţia *B* găsită este o soluţie foarte bună, apropiată de cea optimă sau chiar optimă. Dacă funcţia alege nu este foarte bine concepută, atunci soluţia *B* găsită va fi doar o soluţie posibilă şi nu va fi optimă. **=>** Criteriul de selecţie implementat în funcţia alege, o poate apropia mai mult sau mai puţin de soluţia optimă *B**.

Greedy – Strategia 2

În anumite cazuri, ordinea în care trebuie considerate elementele din mulţimea *A* se poate stabili de la început (funcţia prelucreaza), obţinându-se, pe baza mulţimii *A*, un vector *V* cu *n* componente:

```
B = multimea vida;
prelucreaza(A,V);
for (i=0; i<n; i++) {
        x = V[i];
        if (posibil(B,x))
                * adauga elementul x la multimea B; }
```

La a doua variantă, dificultatea funcţiei alege nu a dispărut, ci s-a transferat funcţiei prelucreaza. Dacă prelucrarea mulţimii *A* este bine făcută, atunci se poate ajunge la o soluţie optimă. Altfel se va obţine doar o soluţie posibilă, mai mult sau mai puţin apropiată de optim.

Greedy – probleme de optimizare

Un exemplu tipic de aplicare al metodei Greedy îl reprezintă problemele de optimizare. În acest tip de probleme, de regulă, se cere să se selecteze din datele de intrare acele elemente care maximizează o funcţie de cost.

Ideea generală a metodei este de a alege la fiecare pas acel element care determină cea mai mare creştere a acestei funcţii.

Neanalizând influenţa corelaţiei dintre elemente asupra funcţiei de cost, metoda nu poate garanta că aceste *maximizări locale*, succesive, conduc întotdeauna la *maximul global* aşteptat. Aceasta înseamnă că sunt situaţii în care metoda Greedy nu generează soluţia optimă, deşi aceasta există.

Greedy

Există o clasă de probleme pentru care metoda Greedy conduce la rezultate optimale, și anume atunci când *soluția optimă globală se compune întotdeauna din soluțiile optime locale*, de la fiecare pas.

În acest gen de probleme se încadrează găsirea *arborelui de acoperire minimă a unui graf*, pentru care avem algoritmii *Prim* și *Kruskal*, de tip Greedy.

Greedy

La modul general, Greedy nu găsește soluția optimă și nici măcar nu garantează că se va găsi o soluție, chiar dacă aceasta există.

De exemplu, să considerăm că ne aflăm în Paris și dorim să vizităm turnul Eiffel. Vedem turnul pe deasupra clădirilor, dar, tot din cauza clădirilor, nu putem vedea în totalitate niciun drum până la el.

Dacă aplicăm metoda Greedy, vom proceda în felul următor: alegem întotdeauna strada care se îndreaptă cel mai mult în direcția turnului și mergem pe ea. Când ajungem la o intersecție, din nou alegem strada care se îndreaptă cel mai mult către turn.

Greedy

Conform acestei abordări, există următoarele posibilități:

- drumul ales de noi este într-adevăr cel mai scurt și ajungem cel mai repede la destinație - în acest caz am găsit soluția optimă
- drumul găsit este mai lung decât alte drumuri (de exemplu dacă strada care părea că duce chiar către destinație face ulterior un ocol larg, ceea ce ne abate de la direcția noastră), dar în final tot reușim să ajungem la destinație - am găsit o soluție, dar ea nu este optimă
- drumul ales se oprește într-o fundătură din care nu mai putem ieși, deoarece Greedy nu face reveniri la pașii anteriori - nu am găsit nicio soluție, deși ea există

Greedy

Exemple de abordări Greedy:

- dorim să umplem o cutie cu niște obiecte: alegem întotdeauna obiectul cel mai mare care trebuie pus în cutie (deci cel care ne duce cel mai aproape de umplerea cutiei), până când nu mai avem obiecte sau cutia se umple - vom ajunge întotdeauna la o soluție și în plus soluția necesită un număr minim de operații, dar este posibil ca spațiul din cutie să nu fie folosit în mod optim

Greedy

Exemple de abordări Greedy:

- avem de sortat crescător un vector: iterăm de la stânga la dreapta, și pentru fiecare poziție alegem elementul minim din vectorul care începe de la iterator (deci elementul care ne apropie cel mai mult de soluție), element pe care îl vom pune la poziția curentă - acesta este algoritmul de sortare cu două bucle *for*, destul de des folosit. Chiar dacă este unul dintre cei mai ineficienți algoritmi de sortare, implementarea sa este simplă.

De data trecută: Recursivitatea

Greedy

Divide et impera

Backtracking

Exerciții

Divide et Impera - Principii

Este o metodă fundamentală de proiectare a algoritmilor care poate să conducă la soluţii deosebit de eficiente.

Principiul de bază al acestei tehnici este acela de a descompune în mod repetat o problemă complexă în două sau mai multe subprobleme de acelaşi tip, urmată de combinarea soluţiilor acestor subprobleme pentru a obţine soluţia problemei iniţiale.

Întrucât subproblemele rezultate din descompunere sunt de acelaşi tip cu problema iniţială, metoda se exprimă în mod natural printr-o funcţie recursivă.

Apelul recursiv se continuă până în momentul în care subproblemele devin banale şi soluţiile lor evidente.

Divide et Impera - Exemplu

Exemplu: o localitate este despărțită în două de un râu peste care există un singur pod. Noi ne aflăm la o locație dintr-o parte a râului și dorim să ajungem la o altă locație din cealaltă parte a râului.

Din cauză că există doar un singur pod, este evident că drumul pe care îl alegem ca să străbatem prima parte a localității trebuie neapărat să ne ducă la acel pod.

Analogic, orice drum pe care trebuie să-l parcurgem în cealaltă jumătate a localității, trebuie neapărat să înceapă de la pod. Pe baza acestor observații simple, putem descompune problema găsirii drumului în două subprobleme independente: găsirea unui drum până la pod și găsirea unui drum de la pod la destinație. Rezolvările acestor subprobleme nu depind în niciun fel una de cealaltă.

Divide et Impera – Aplicații

Abordările de tip D&C sunt în special importante în contextul microprocesoarelor cu mai multe nuclee (cores). Dacă avem un algoritm D&C, putem rezolva simultan subprobleme folosind toate nucleele disponibile (multithreading).

Chiar dacă rulat într-un singur fir de execuție (thread), algoritmul D&C este mai lent decât alte abordări, atunci când este executat în paralel, el le va depăși ca performanțe.

Există și posibilitatea de a distribui într-o rețea de calculatoare rezolvarea subproblemelor (distributed computing), mărind astfel și mai mult capacitățile computaționale.

Divide et Impera - Pseudocod

```
void Divide(* parametri care definesc o problema) {
if (* problema este una triviala) {
        * rezolva problema in mod direct;
        * returneaza rezultatele;}
else{
        * imparte problema in subprobleme;
        for( fiecare subproblema )
                * apeleaza Divide(subproblema);
        * combina rezultatele subproblemelor;
        * returneaza rezultatele pentru problema;
        }
}
```

De data trecută: Recursivitatea

Greedy

Divide et impera

Backtracking

Exerciții

Metoda Backtracking

Pentru a înţelege semnificaţia cuvântului *backtracking* vom reda definiţia din *Cambridge Online Dictionary*, http://dictionary.cambridge.org/:

to go back along a path you have just followed

Ideea de bază este aceea de *revenire pe calea parcursă*. Algoritmii de tip backtracking încep să exploreze spaţiul soluţiilor în mod exhaustiv, pe toate căile posibile. Atunci când pe calea curentă de explorare se constată că nu mai sunt şanse să se ajungă la o soluţie validă, se revine cu un pas înapoi şi se abordează o altă cale de explorare.

În concluzie, metoda Backtracking constă în efectuarea unor **încercări repetate**, în vederea găsirii soluţiilor, **cu posibilitatea revenirii** în caz de eşec.

Metoda Backtracking - principii

Soluţia se poate reprezenta sub forma unui vector *X=(x0, x1,…, xn-1), X*∈ *S = S0* x*S1*x … x *Sn-1*, unde mulţimile *S0 , …, Sn-1* sunt mulţimi finite. Pentru fiecare problemă concretă sunt date anumite relaţii între componentele *x0, x1, …, xn-1* ale vectorului *X*, numite condiţii interne.

Mulţimea *S* reprezintă spaţiul soluţiilor posibile. Soluţiile posibile care satisfac condiţiile interne se numesc soluţii rezultat.

În continuare, exprimăm condiţiile care trebuie satisfăcute sub forma unei funcţii logice notată: *Solutie(x0,x1,...,xn-1)*. Un element *X*=(*x0, x1, ..., xn-1*) ∈ *S* este soluţie a problemei dacă funcţia *Solutie* aplicată componentelor lui *X* va returna valoarea *true*.

Metoda Backtracking - principii

Scopul algoritmului concret poate să fie determinarea unei soluţii rezultat sau a tuturor soluţiilor rezultat, fie în scopul afişării lor, fie pentru a alege una optimă din punctul de vedere al unor criterii de optimizare (minimizare sau maximizare).

O metodă simplă de selectare a soluţiilor rezultat este aceea de a genera toate soluţiile posibile şi de a verifica satisfacerea condiţiilor interne *(căutare exhaustivă* în întregul spațiu al soluțiilor posibile*).* Această metodă necesită însă un timp de execuţie foarte mare şi nu se aplică decât rar în practică.

Metoda Backtracking - principii

Un algoritm backtracking performant, ca şi în cazul Greedy de altfel, evită generarea tuturor soluţiilor posibile. În acest scop, elementele vectorului X primesc, pe rând şi în ordine, valori. După asocierea unei valori lui *x[k]* se verifică îndeplinirea unor **condiţii de continuare** pentru secvenţa (*x0, ..., xk*): funcţia *Validare(x0, x1, ..., xk)* şi numai apoi se trece la încercarea de asociere a unei valori pentru *x[k+1]*.

Neîndeplinirea acestor condiţii de continuare ne arată, încă din această fază, că soluţia finală nu poate fi o soluţie rezultat. În cazul unui eşec la această verificare, se alege o altă valoare pentru *x[k]* ∈ *Sk* sau, dacă *Sk* a fost epuizat, se micşorează *k* cu o unitate şi procesul de alegere se repetă.

Metoda Backtracking - concluzii

Numele metodei (algoritmi cu revenire) provine de la revenirile care se efectuează în caz de eşec.

Condiţiile de *continuare (validare)* derivă din condiţiile *interne* ale problemei.

Alegerea optimă a condiţiilor de continuare (validare) poate determina reducerea numărului de calcule (încercări) care urmează să fie efectuate (viteza programului).

Acest proces de încercări repetate şi revenire în caz de eşec (cu reluarea altei valori şi continuare) se exprimă în mod natural şi în manieră recursivă.

Metoda Backtracking - concluzii

Algoritmii backtracking se pot implementa atât în variantă recursivă cât și nerecursivă.

Varianta recursivă ne scutește de implementarea unor structuri de date în care să memorăm progresul din pașii intermediari.

Metoda Backtracking - pseudocod

În varianta recursivă, algoritmul backtracking se implementează printr-o funcție, conform următorului pseudocod:

funcție back(poziție_curentă)

dacă poziție_curentă nu este validă sau face deja parte din calea curentă, revenire

adaugă poziție_curentă la calea curentă

dacă poziție_curentă determină o soluție

atunci folosește soluția găsită

altfel

pentru fiecare poziție_următoare la care se poate ajunge din poziție_curentă

back(poziție_următoare)

șterge poziție_curentă din calea curentă

Metoda Backtracking - algoritm

Mai simplu spus, algoritmii de tip backtracking funcționează în felul următor:

- soluția problemei se construiește succesiv, pas cu pas
- dacă la un pas există mai multe posibilități de continuare, se vor încerca pe rând fiecare dintre ele
- dacă s-a ajuns într-un punct în care nu se mai poate continua, se revine la pasul anterior pentru a se încerca următoarea variantă posibilă
- după ce s-au epuizat toate posibilitățile de la pasul anterior, se revine cu încă un pas mai înainte și tot așa, în mod recursiv, până când au fost încercate toate posibilitățile din toți pașii

Metoda Backtracking – implementare

O implementare recursivă simplă a algoritmului backtracking în limbajul de programare C:

```
void back(int k){
 for(int i=0;i<n;i++)
      {
       v[k]=i;
       if (valid(k))
              if(solutie(k))
                     afisare();
              else
                      back(k+1);
        }
}
```

Metoda Backtracking - optimizări

Din cauză că algoritmii de tip backtracking au o complexitate exponențială, timpul computațional necesar poate depăși destul de repede posibilitățile existente.

De exemplu, după unele estimări, numărul de mutări posibile într-un joc de șah este mai mare decât numărul de electroni din univers. Astfel, un algoritm de backtracking neoptimizat, care să joace șah, nu va putea testa decât foarte puține mutări în avans.

Din acest motiv, sunt foarte importante modalitățile prin care putem reduce numărul de cazuri luat în considerare.

Metoda Backtracking - optimizări

Câteva optimizări importante:

- limitarea la prima soluție găsită, chiar dacă ea nu este optimă
- renunțarea la a căuta mai departe pe o anumită cale, din momentul în care ea duce la un rezultat mai slab decât cel mai bun rezultat găsit anterior
- selectarea variantelor posibile la fiecare pas într-o manieră *Greedy*, astfel încât primele variante încercate să fie cele care ne apropie cât mai mult de destinație
- dacă există mai multe variante posibile, testarea a doar câtorva dintre ele, cele mai promițătoare

Metoda Backtracking - optimizări

Câteva optimizări importante:

- implementarea unui contor de timp sau de număr de pași, iar atunci când contorul depășește o limită maximă, returnarea celui mai bun rezultat găsit până la acel moment
- identificarea unor *puncte nodale* prin care trebuie să treacă toate soluțiile și apoi împărțirea căutării în două: prima căutare înainte de punctul nodal și a doua după punctul nodal.
  - De exemplu, dacă între două orașe există un singur drum și vrem să ajungem dintr-un oraș în altul, se poate face o căutare de la adresa de pornire până la drumul de legătură și o altă căutare de la drum până la adresa de destinație.

De data trecută: Recursivitatea

Greedy

Divide et impera

Backtracking

Exerciții

Metoda Backtracking – exerciții

Exercițiul 1:

Se primește un cuvânt din lina de comandă. Să se afișeze toate anagramele sale.(DEX anagramă: schimbare a ordinii literelor unui cuvânt, pentru a obține alt cuvânt; cuvânt obținut prin această schimbare.)

Metoda Backtracking – exerciții

```
int main(int argc, char* argv[])
{
char *cuv=strdup(argv[1]);
int n=strlen(cuv);
back(1, n, cuv);
return 0;
}
```

Metoda Backtracking – exerciții

```
void back(int k, int n, char* cuv)
{
for (int i=1; i<=n; i++){
        st[k]=i;
        if (valid(st, k)){
                if (solutie(st, k, n)){
                        afisare(st, k, cuv);          }
                else{
                        back(k+1, n, cuv);            }
                                }
        }
}
```

Metoda Backtracking – exerciții

```
int valid(int st[], int k){
        for (int i=1; i<k; i++){
                if (st[i]==st[k]){
                        return 0;       }
                }
        return 1;}

int solutie(int st[], int k, int n){
        return (k==n);}

void afisare(int st[], int k, char *cuv){
        for (int i=1; i<=k; i++){
                printf("%c", cuv[st[i]-1]);    }
        printf("\n");}
```

Metoda Backtracking – exerciții

Exercițiul 2:

Să se genereze toate șirurile de cifre distincte a căror sumă este egală cu n citit de la tastatură.

Metoda Backtracking – exerciții

```
int main(){
scanf("%d", &x);
back(0);
return 0;
}
```

Metoda Backtracking – exerciții

```
void back(int k){
for (int i = 1; i < n; i++)    {
        v[k] = i;
        if (valid(k))        {
                if (solutie(k))          {
                        afisare(k);          }
                else           {
                        back(k + 1);          }
                }
        }
}
```

Metoda Backtracking – exerciții

```
int valid(int k){
int suma = 0;
for (int i = 0; i < k; i++)    {
        if (v[i] == v[k])        {
                return 0;        }
        suma = suma + v[i];
        }
suma = suma + v[k];
        if (suma > x)    {
                return 0;    }
        else    {
                return 1;    }
}
```

Metoda Backtracking – exerciții

```
int solutie(int k){
int suma = 0;
for (int i = 0; i <= k; i++)    {
        suma = suma + v[i];    }
if (x == suma)    {
        return 1;    }
else    {
        return 0;    }
}
void afisare(int k){
for (int i = 0; i <= k; i++)    {
        printf("%d ", v[i]);    }
printf("\n");
}
```

Metoda Backtracking – exerciții

Problema celor 8 regine (Eight Queens)

Se cere să se realizeze programul care să plaseze opt regine pe o tablă de şah, astfel încât nici una dintre ele să nu le ameninţe pe celelalte. La jocul de şah, o regină "ameninţă" pe linii, coloane şi diagonale, pe orice distanţă.

Metoda Backtracking – exerciții

Problema celor 8 regine (Eight Queens)

Această problemă a fost investigată de Carl Friedrich Gauss în 1850 (care însă nu a rezolvat-o complet). Nici până în prezent problema nu are o soluţie analitică satisfăcătoare. În schimb ea poate fi rezolvată prin încercări, necesitând o mare cantitate de muncă, răbdare şi acurateţe (condiţii în care calculatorul se descurcă excelent).

Problema are 92 de soluţii din care, din motive de simetrie a tablei de șah, doar 12 sunt diferite.

Problema poate fi uşor extinsă pentru *n* regine plasate pe o tablă pătrată cu *n* linii şi *n* coloane.

Metoda Backtracking – exerciții

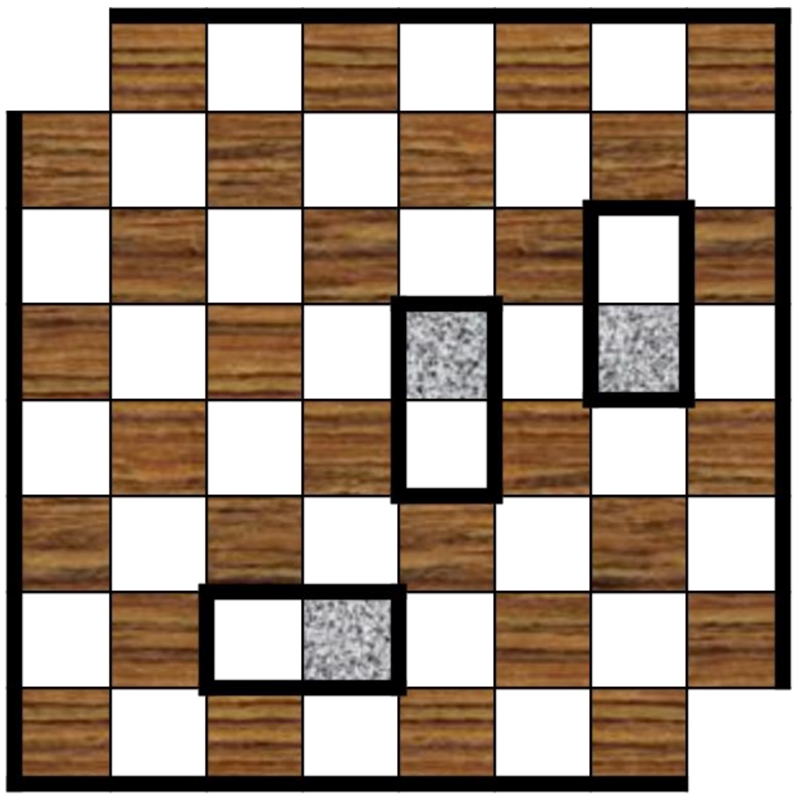
Pe fiecare linie sau coloană de pe tablă se va afla o singură regină. Se va parcurge tabla de şah linie cu linie (k=0..7), iar în cadrul unei linii coloană cu coloană (i=0..7) şi se vor plasa reginele în acele pătrate care nu sunt în “priza reginelor” plasate anterior. Pentru parcurgerea tablei se va utiliza tehnica backtracking.

Deoarece pe fiecare linie a tablei de şah se poate găsi exact o regină, o soluţie rezultat se poate reprezenta sub forma unui vector *C* = (*c*0,..., *c*7) unde *c[k]* reprezintă coloana pe care se află regina de pe linia *k* (c[k] aparține intervalului 0-7).

Metoda Backtracking – exerciții

- *Spaţiul soluţiilor posibile* este produsul cartezian $S = C \times C \times C \times C \times C \times C \times C \times C$
- *Condiţiile interne*, rezultă din regulile şahului şi sunt reprezentate de faptul că două dame nu se pot afla pe o aceeaşi coloană sau pe o aceeaşi diagonală.

Metoda Backtracking – exerciții



Metoda Backtracking – exerciții

Funcţia Solutie trebuie să verifice dacă nu există regine care se află pe aceeaşi coloană sau dacă nu cumva există regine care se atacă pe diagonală.

Verificarea este simplă. Trebuie să verificăm că intre elementele (c0, c1, c2, c3, c4, c5, c6, c7) nu există două care au aceeaşi valoare. Aceasta ar insemna că avem două regine pe aceeaşi coloană.

Apoi mai trebuie să verificăm că orice i,k din {0, 1, 2, 3, 4, 5, 6, 7}, $|i-k| \neq |ci-ck|$. Aceasta este condiţia ca să nu existe două regine care se atacă pe diagonală. Verificări similare vor fi efectuate pe parcurs de funcţia Valid.

Vă mulțumesc!