Laborator 5: Backtracking

Obiective laborator

- Întelegerea noțiunilor de bază despre backtracking;
- Însușirea abilităților de implementare a algoritmilor bazați pe backtracking;
- Rezolvarea unor probleme NP-complete în timp exponențial.

Precizări inițiale

Toate exemplele de cod se găsesc pe pagina pa-lab::demo/lab05 [https://github.com/acs-pa/pa-lab/tree/main/demo/lab05].

Exemplele de cod apar încorporate și în textul laboratorului pentru a facilita parcurgerea cursivă a acestuia. ATENȚIE! Varianta actualizată a acestor exemple se găsește întotdeauna pe GitHub.

- Toate bucățile de cod prezentate în partea introductivă a laboratorului (înainte de exerciții) au fost testate. Cu toate acestea, este posibil ca din cauza mai multor factori (formatare, caractere invizibile puse de browser etc.) un simplu copy-paste să nu fie de ajuns pentru a compila codul.
- Vă rugam să compilați **DOAR** codul de pe GitHub. Pentru raportarea problemelor, contactați unul dintre maintaineri.
- Pentru orice problemă legată de conținutul acestei pagini, vă rugăm să dați e-mail unuia dintre responsabili.

Ce este Backtracking?

Backtracking este un algoritm care caută **una sau mai multe soluții** pentru o problema, printr-o căutare exhaustiva, mai eficientă însă în general decât o abordare „generează si testează”, de tip „forță brută”, deoarece un candidat parțial care nu duce la o soluție este abandonat. Poate fi folosit pentru orice problemă care presupune o căutare în **spațiul stărilor**. În general, în timp ce căutăm o soluție e posibil să dăm de un deadend în urma unei alegeri greșite sau să găsim o soluție, dar să dorim să căutăm în continuare alte soluții. În acel moment trebuie să ne întoarcem pe pașii făcuți (**backtrack**) și la un moment dat să luăm altă decizie. Este relativ simplu din punct de vedere conceptual, dar complexitatea algoritmului este exponentială.

Importanța – aplicaţii practice

Există foarte multe probleme (de exemplu, problemele NP-complete sau NP-dificile) care pot fi rezolvate prin algoritmi de tip backtracking mai eficient decât prin „forta bruta” (adică generarea tuturor alternativelor și selectarea soluțiilor). Atenție însă, complexitatea computațională este de cele mai multe ori **exponențială**. O eficientizare se poate face prin combinarea cu tehnici de propagare a restricțiilor. Orice problemă care are nevoie de parcurgerea spațiului de stări se poate rezolva cu backtracking.

Descrierea problemei și a rezolvărilor

Pornind de la strategiile clasice de parcurgere a **spațiului de stări**, algoritmii de tip backtracking practic enumeră un set de candidați parțiali, care, după completarea definitivă, pot deveni soluții potențiale ale problemei inițiale. Exact ca strategiile de **parcurgere în lățime/adâncime** și backtracking-ul are la bază expandarea unui nod curent, iar determinarea soluției se face într-o manieră incrementală. Prin natura sa, backtracking-ul este recursiv, iar în arborele expandat top-down se aplică operații de tipul pruning (tăiere) dacă soluția parțială nu este validă.

Algoritm de baza

```
/* Domain - domeniul curent (cu un cardinal mai mic decat la pasul trecut)
Solution - solutia curenta pe care o extindem catre cea finala */
back(Domain, Solution):
    if check(Solution):
        print(Solution)
        return

    for value in Domain:
        NextSolution = Solution.push(value)
        NextDomain = Domain.erase(value)
        back(NextDomain, NextSolution)
```

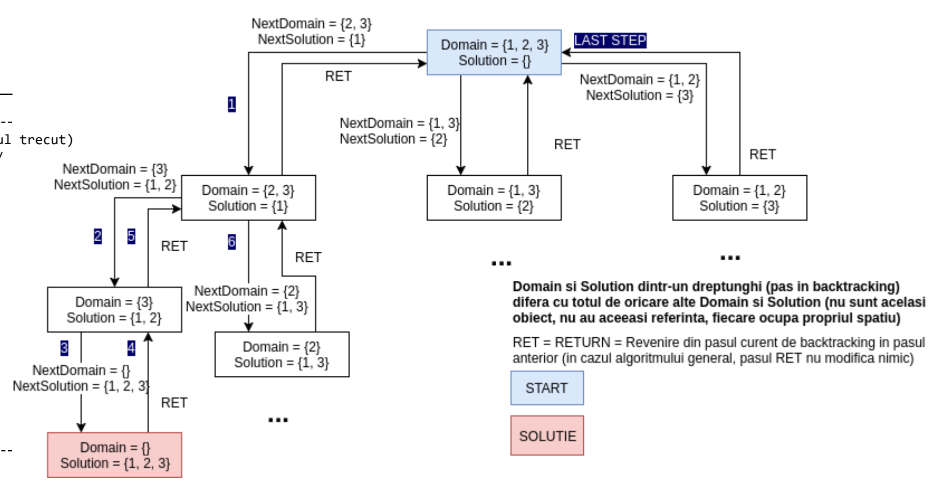
Algoritm de baza (modificat pentru transmitere prin referinta)

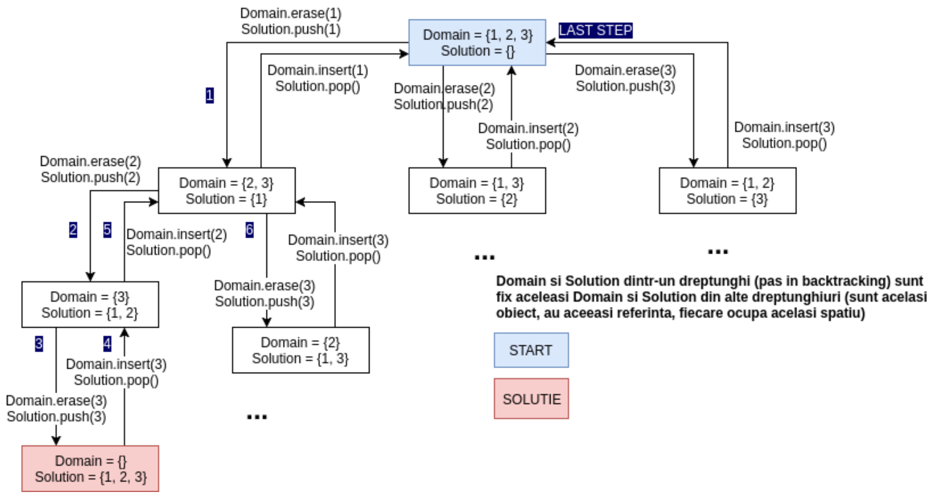
```
/* Domain - domeniul curent (cu un cardinal mai mic decat la pasul trecut)
Solution - solutia curenta pe care o extindem catre cea finala */
back(Domain, Solution):
    if check(Solution):
        print(Solution)
        return

    for value in Domain:
        /* DO */
        Solution = Solution.push(value)
        Domain = Domain.erase(value)

        /* RECURSION */
        back(Domain, Solution)

        /* UNDO */
        Solution = Solution.pop()
        Domain = Domain.insert(value)
```





Exemple clasice

Ne vom ocupa în continuare de următoarele probleme:

- Permutări
- Combinări
- Aranjamente
- Submulțimi
- Generare de șiruri
- Problema damelor
- Problema șoricelului
- Tic-Tac-Toe
- Sudoku
- Ultimate Tic-Tac-Toe

Sudoku și Ultimate Tic-Tac-Toe sunt probleme foarte grele. În general nu putem explora tot spațiul stărilor pentru un input arbitrar dat.

Permutări

Enunț

Se dă un număr N. Să se genereze toate permutările mulțimii formate din toate numerele de la 1 la N.

Exemple

N = 3 ⇒ M = {1, 2, 3}

Solutie:

- {1, 2, 3}
- {1, 3, 2}
- {2, 1, 3}
- {2, 3, 1}
- {3, 1, 2}
- {3, 2, 1}

Soluții

Backtracking (algoritmul în cazul general)

```
/* deoarece numerele sunt sterse din domeniu odata ce sunt folosite, soluția generata este garantata
sa nu contina duplicate. Astfel, atunci cand domeniul ajunge vid, soluția este intotdeauna corecta */
bool check(std::vector<int> solution) {
    return true;
}

void printSolution(std::vector<int> solution) {
    for (auto &s : solution) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}

void back(std::vector<int> domain, std::vector<int> solution) {
    /* dupa ce am folosit toate elementele din domeniu putem verifica daca
    am gasit o solutie */
    if (domain.size() == 0) {
        if(check(solution)) {
            printSolution(solution);
        }
        return;
```

```
    }

    /* incercam sa adaugam in solutie toate valorile din domeniu, pe rand */
    for (unsigned int i = 0; i < domain.size(); ++i) {
        /* cream o solutie noua si un domeniu nou care sunt identice cu cele
        de la pasul curent */
        std::vector<int> newSolution(solution), newDomain(domain);

        /* adaugam in noua solutie elementul ales din domeniu */
        newSolution.push_back(domain[i]);
        /* stergem elementul ales din noul domeniu */
        newDomain.erase(newDomain.begin() + i);

        /* apelam recursiv backtracking pe noul domeniu si noua solutie */
        back(newDomain, newSolution);
    }
}

int main() {
    /* dupa ce am citit n initializam domeniul cu n elemente, numerele de la 1 la n,
    iar solutia este vida initial */
    std::vector<int> domain(n), solution;
    for (int i = 0; i < n; ++i) {
        domain[i] = i + 1;
    }

    /* apelam backtracking pe domeniul nostru, cautand solutia in vectorul solution */
    back(domain, solution);
}
```

Apelarea inițială (din “main”) se face astfel: “back(domain, solution);”, unde domain reprezintă un vector cu elementele de la 1 la N, iar solution este un vector gol.

Nu este indicată implementarea backtracking-ului astfel deoarece este foarte costisitor din punct de vedere al memoriei(se creează noi domenii și soluții la fiecare pas).

Complexitate

Soluția va avea următoarele complexitati:

- complexitate temporala : $T(n) = O(n * n!)$
  - explicație : Complexitatea generarii permutarilor, $O(n!)$, se înmulteşte cu complexitatea copierii vectorilor soluție si domeniu si a stergerii elementelor din domeniu, $O(n)$
- complexitate spatiala : $S(n) = O(n^2)$
  - explicație : Fiecare nivel de recursivitate are propria lui copie a soluției și a domeniului. Sunt n nivele de recursivitate, deci complexitatea spatială este $O(n * n) = O(n^2)$

Backtracking (date transmise prin referinta)

```
/* deoarece numerele sunt sterse din domeniu odata ce sunt folosite, soluția generata este garantata sa nu contina duplicate. Astfel, atunci cand domeniul ajunge vid, soluția este intotde
bool check(std::vector<int> solution) {
    return true;
}

void printSolution(std::vector<int> &solution) {
    for (int s : solution) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}

void back(std::vector<int> &domain, std::vector<int> &solution) {
    /* dupa ce am folosit toate elementele din domeniu putem verifica daca
    am gasit o solutie */
    if (domain.size() == 0) {
        if(check(solution)) {
            printSolution(solution);
        }
        return;
    }

    /* incercam sa adaugam in solutie toate valorile din domeniu, pe rand */
    for (unsigned int i = 0; i < domain.size(); ++i) {
        /* retinem valoarea pe care o scoatem din domeniu ca sa o readaugam dupa
        apelarea recursiva a backtracking-ului */
        int tmp = domain[i];

        /* adaug elementul curent la potentiala solutie */
        solution.push_back(domain[i]);
        /* sterg elementul curent din domeniu ca sa il pot pasa prin referinta
        si sa nu fie nevoie sa creez alt domeniu */
        domain.erase(domain.begin() + i);

        /* apelez recursiv backtracking pe domeniul si solutia modificate */
        back(domain, solution);

        /* refac domeniul si solutia la modul in care aratau inainte de apelarea
        recursiva a backtracking-ului, adica readaug elementul eliminat in
        domeniu si il sterg din solutie */
        domain.insert(domain.begin() + i, tmp);
        solution.pop_back();
    }
}

int main() {
    /* dupa ce am citit n initializam domeniul cu n elemente, numerele de la 1 la n,
    iar solutia este vida initial */
    std::vector<int> domain(n), solution;
    for (int i = 0; i < n; ++i) {
        domain[i] = i + 1;
    }

    /* apelam backtracking pe domeniul nostru, cautand solutia in vectorul solution */
```

```
    back(domain, solution);
}
```

Apelarea initiala (din “int main”) se face astfel: “back(domain, solution);”, unde domain reprezinta un vector cu elementele de la 1 la N, iar solution este un vector gol.

Complexitate

Soluția va avea următoarele complexități:

- complexitate temporală : $T(n) = O(n * n!)$
  - explicație : Complexitatea generării permutărilor, $O(n!)$, se înmulțește cu complexitatea ștergerii elementelor din domeniu, $O(n)$
- complexitate spatială : $S(n) = O(n)$
  - explicație : Spre deosebire de solutia anterioară, toate nivelele de recursivitate folosesc aceeași soluție și același domeniu. Complexitatea spatială este astfel redusă la $O(n)$

Această abordare este mai eficientă decât cea generală, deoarece se evită folosirea memoriei auxiliare.

Backtracking (tăierea ramurilor nefolositoare)

```
bool check(std::vector<int> &solution) {
    return true;
}

void printSolution(std::vector<int> &solution) {
    for (auto s : solution) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}

void back(int step, int stop, std::vector<int> &domain,
        std::vector<int> &solution, std::unordered_set<int> &visited) {

    /* vom verifica o solutie atunci cand am adaugat deja N elemente in solutie,
    adica step == stop */
    if (step == stop) {
        /* deoarece am avut grija sa nu se adauge duplicate, "check()" va returna
        intotdeauna "true" */
        if(check(solution)) {
            printSolution(solution);
        }
        return;
    }

    /* Adaugam in solutie fiecare element din domeniu care *NU* a fost vizitat
    deja renuntand astfel la nevoia de a verifica duplicatele la final prin
    functia "check()" */
    for (unsigned int i = 0; i < domain.size(); ++i) {
        /* folosim elementul doar daca nu e vizitat inca */
        if (visited.find(domain[i]) == visited.end()) {
            /* il marcam ca vizitat si taiem eventuale expansiuni nefolositoare
            viitoare (ex: daca il adaug in solutie pe 3 nu voi mai avea
            niciodata nevoie sa il mai adaug pe 3 in continuare) */
            visited.insert(domain[i]);

            /* adaugam elementul curent in solutie pe pozitia pasului curent
            (step) */
            solution[step] = domain[i];

            /* apelam recursiv backtracking pentru pasul urmator */
            back(step + 1, stop, domain, solution, visited);

            /* stergem vizitarea elementului curent (ex: pentru N = 3, dupa ce
            la pasul "step = 0" l-am pus pe 1 pe prima pozitie in solutie si
            am continuat recursiv pana am ajuns la solutiile {1, 2, 3} si
            {1, 3, 2}, ne dorim sa il punem pe 2 pe prima pozitie in solutie si
            sa continuam recursiv pentru a ajunge la solutiile {2, 1, 3} etc.) */
            visited.erase(domain[i]);
        }
    }
}

int main() {
    /* dupa ce am citit n initializam domeniul cu n elemente, numerele de la 1 la n,
    iar solutia este initializata cu un vector de n elemente (deoarece o permutare
    contine n elemente) */
    std::vector<int> domain(n), solution(n);
    std::unordered_set<int> visited;
    for (int i = 0; i < n; ++i) {
        domain[i] = i + 1;
    }

    /* apelam back cu step = 0 (atatea elemente avem adaugate in solutie),
    stop = n (stim ca vrem sa adaugam n elemente in solutie pentru ca o
    permutare e alcatuita din n elemente), domain este vectorul de valori
    posibile, solution este vectorul care simuleaza stiva pe care o vom
    umple, visited este un unordered_set (initial gol) in care retinem daca
    un element din domeniu se afla deja in solutia curenta la un anumit pas */
    back(0, n, domain, solution, visited);
}
```

Apelarea inițială (din “main”) se face astfel: “back(0, n, domain, solution, visited);”, unde domain reprezintă un vector cu elementele de la 1 la N, iar solution este un vector de n elemente, 0 este pasul curent, n este pasul la care dorim să ne oprim, iar visited este map-ul care ne permite să ținem cont de ce elemente au fost vizitate sau nu.

Complexitate

Soluția va avea următoarele complexitați:

- complexitate temporală : $T(n) = O(n * n!) = O(n!)$

- explicație : Complexitatea generării permutărilor, O(n!), se înmulțește cu complexitatea iterării prin domeniu, $O(n)$
- complexitate spatială : $S(n) = O(n)$
  - explicație : Toate nivelele de recursivitate folosesc aceeași soluție și același domeniu.

Această soluție este optimă și are complexitatea temporală $T(n) = O(n!)$. Nu putem să obținem o soluție mai bună, întrucât trebuie să generăm n! permutări.

De asemenea, este optimă și din punct de vedere spatial, întrucât trebuie să avem $S(n) = O(n)$, din cauza stocării permutării generate.

Combinări

Enunț

Se dau numerele N si K. Să se genereze toate combinările mulțimii formate din toate numerele de la 1 la N, luate câte K.

Exemple

N = 4, K = 2 ⇒ M = {1, 2, 3, 4}

Soluție:

- {1, 2}
- {1, 3}
- {1, 4}
- {2, 3}
- {2, 4}
- {3, 4}

Soluții

Backtracking (tăierea ramurilor nefolositoare)

```
bool check(std::vector<int> &solution) {
    return true;
}

void printSolution(std::vector<int> &solution, std::vector<int> &domain, int stop) {
    for (unsigned i = 0; i < stop; ++i) {
        std::cout << domain[solution[i]] << " ";
    }
    std::cout << "\n";
}

void back(int step, int stop, std::vector<int> &domain,
        std::vector<int> &solution) {
    /* vom verifica o solutie atunci cand am adaugat deja K elemente in solutie,
    adica step == stop */
    if (step == stop) {
        /* deoarece am avut grija sa se adauge elementele doar in ordine
        crescatoare, "check()" va returna intotdeauna "true" */
        if(check(solution)) {
            printSolution(solution, domain, stop);
        }
        return;
    }

    /* daca este primul pas, alegem fiecare element din domeniu ca potential
    candidat pentru prima pozitie in solutie; altfel, pentru a elimina ramurile
    in care de exemplu {2, 1} se va genera dupa ce s-a generat {1, 2} (adica
    ar fi duplicat), vom folosi doar elementele din domeniu care sunt mai mari
    decat ultimul element adaugat in solutie (solution[step - 1]) */
    unsigned i = step > 0 ? solution[step - 1] + 1 : 0;
    for (; i < domain.size(); ++i) {
        solution[step] = i;
        back(step + 1, stop, domain, solution);
    }
}

int main() {
    /* dupa ce citim n si k initializam domeniul cu valorile de la 1 la n,
    iar solutia este initializata cu un vector de k elemente (fiindca o
    combinare de "n luate cate k" are k elemente) */
    std::vector<int> domain(n), solution(k);
    for (int i = 0; i < n; ++i) {
        domain[i] = i + 1;
    }

    back(0, k, domain, solution);
}
```

În această soluție ne bazăm pe faptul că toate combinările pot fi generate în ordine crescătoare, adică soluția {1, 3, 4} e echivalentă cu {4, 1, 3}.

Această soluție este optimă întrucât toate soluțiile generate sunt corecte (de aceea funcția check întoarce true). Deoarece problema cere obținerea tuturor combinărilor, aceasta complexitate nu poate fi mai mică de Combinări(n, k).

Complexitate

Soluția va avea următoarele complexități:

- complexitate temporală : $T(n) = O(Combinari(n, k))$
- complexitate spatială : $S(n) = O(n + k) = O(n)$
  - explicație : $k <= n$, deci $O(n + k) = O(n)$

Problema șoricelului

Enunț

Se dă un număr N și o matrice pătratică de dimensiuni N x N în care elementele egale cu 1 reprezintă ziduri (locuri prin care nu se poate trece), iar cele egale cu 0 reprezintă spații goale. Această matrice are un șoricel în celula (0, 0) și o bucată de brânză în celula (N - 1, N - 1). Scopul șoricelului e să ajungă la bucata de brânză. Afișați toate modurile în care poate face asta știind că acesta poate merge doar în dreapta sau în jos cu câte o celulă la fiecare pas.

Exemple

- 2
- 0 1
- 0 0

Există 1 drum posibil:

- (0, 0)→(1, 0)→(1, 1)

- 3
- 0 0 0
- 0 1 0
- 0 0 0

Există 2 drumuri posibile:

- (0,0)→(0,1)→(0,2)→(1,2)→(2,2)
- (0,0)→(1,0)→(2,0)→(2,1)→(2,2)

- 4
- 0 0 0 1
- 0 1 1 0
- 0 0 0 0
- 0 0 0 0

Există 4 drumuri posibile:

- (0,0)→(1,0)→(2,0)→(2,1)→(2,2)→(2,3)→(3,3)
- (0,0)→(1,0)→(2,0)→(2,1)→(2,2)→(3,2)→(3,3)
- (0,0)→(1,0)→(2,0)→(2,1)→(3,1)→(3,2)→(3,3)
- (0,0)→(1,0)→(2,0)→(3,0)→(3,1)→(3,2)→(3,3)

Soluții

Backtracking (transmitere prin referință)

```
bool check(std::vector<std::pair<int, int> > &solution, int walls[100][100]) {
    for (unsigned i = 0; i < solution.size() - 1; ++i) {
        /* line_prev si col_prev reprezinta celula in care se afla soricelul la
        pasul i; line_next si col_next reprezinta celula in care se afla
        la pasul i + 1; trebuie sa fim siguri ca soricelul nu a ajuns pe zid
        si ca urmatoarea celula este sub sau in dreapta celulei curente */
        int line_prev = solution[i].first;
        int line_next = solution[i + 1].first;
        int col_prev = solution[i].second;
        int col_next = solution[i + 1].second;

        /* walls[x][y] == 1 inseamna ca este zid pe linia x, coloana y */
        if (walls[line_prev][col_prev] == 1 ||
                !((line_next == line_prev + 1 && col_next == col_prev) ||
                (line_next == line_prev && col_next == col_prev + 1))) {
            return false;
        }
    }

    return true;
}

void printSolution(std::vector<std::pair<int, int> > &solution) {
    for (std::pair<int, int> s : solution) {
        std::cout << "(" << s.first << "," << s.second << ")->";
    }
    std::cout << "\n";
}

void back(std::vector<std::pair<int, int> > &domain, int walls[100][100],
        std::vector<std::pair<int, int> > &solution, int max_iter) {
    /* daca am facut "max_iter" pasi ma opresc si verific daca este corecta
    solutia */
    if (solution.size() == max_iter) {
        if(check(solution, walls)) {
            printSolution(solution);
        }
        return;
    }

    /* avand domeniul initializat cu toate celulele din matrice, incercam sa
    adaugam oricare dintre aceste celule la solutie, verificand la final daca
    solutia este buna */
    for (unsigned int i = 0; i < domain.size(); ++i) {
        /* pastram elementul curent pentru a-l readauga in domeniu dupa
        apelarea recursiva */
```

```
        std::pair<int, int> tmp = domain[i];

        /* adaugam elementul curent la solutia candidat */
        solution.push_back(domain[i]);
        /* stergem elementul curent din domeniu */
        domain.erase(domain.begin() + i);

        /* apelam recursiv backtracking */
        back(domain, walls, solution, max_iter);

        /* adaugam elementul sters din domeniu inapoi */
        domain.insert(domain.begin() + i, tmp);
        /* stergem elementul curent din solutia candidat pentru a o forma pe
        urmatoarea */
        solution.pop_back();
    }
}

int main() {
    /* initializam domeniul si solutia ca vectori de perechi de int-uri;
    domeniul va contine initial toate perechile de indici posibile din
    matrice ((0, 0), (0, 1) ... (n - 1, n - 1)), iar solutia va fi initial
    vida */
    std::vector<std::pair<int, int> > domain, solution;

    fin >> n;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            /* walls[i][j] == 1 daca pe pozitia (i, j) este zid; 0 altfel */
            fin >> walls[i][j];
            domain.push_back({i, j});
        }
    }

    /* apelam back cu domeniul format initial, cu matricea de ziduri, cu
    solutia vida si cu numarul maxim de iteratii = 2 * n - 1 pentru ca
    mergand doar in dreapta si in jos, in 2 * n - 1 pasi va ajunge din
    (0, 0) in (n - 1, n - 1) */
    back(domain, walls, solution, 2 * n - 1);
}
```

Apelarea initiala (din “int main”) se face astfel: “back(domain, walls, solution, 2 * n - 1);”, unde domain reprezinta un vector cu perechi in care sunt toate celulele matricii, solution este un vector gol, walls este matricea care ne arata daca este zid sau nu pe o anumita pozitie, iar 2 * n - 1 e numarul e pasi in care ar trebui ca soricelul sa ajunga la branza pe oriunde ar merge.

Complexitate

Soluția va avea următoarele complexități:

- complexitate temporală : $T(n) = O(Aranjamente(n^2, 2n-1))$
  - explicație: Initial in domeniu avem n^2 valori. Noi dorim sa generam toate submultimile ordonate de cate $2n-1$ elemente. Acestea sunt tocmai aranjamentele de n^2 luate cate $2n-1$.
- complexitate spatială : $S(n) = O(n^2)$
  - explicație: Trebuie să stocăm informație despre drum, care are $2n-1$ celule; stocăm domeniul care are n^2 elemente

Backtracking (tăierea ramurilor nefolositoare)

```
bool check(std::vector<std::pair<int, int> > &solution) {
    return true;
}

void printSolution(std::vector<std::pair<int, int> > &solution) {
    for (std::pair<int, int> s : solution) {
        std::cout << "(" << s.first << "," << s.second << ")->";
    }
    std::cout << "\n";
}

void back(int step, int stop, int walls[100][100],
        std::vector<std::pair<int, int> > &solution, int line_moves[2],
        int col_moves[2]) {
    /* ne oprim dupa ce am ajuns la pasul "stop" si verificam daca solutia este
    corecta */
    if (step == stop) {
        /* deoarece am eliminat ramurile nefolositoare am ajuns la o solutie care
        sigur este corecta */
        if(check(solution)) {
            printSolution(solution);
        }
        return;
    }

    /* daca este primul pas stiu ca soricelul este in pozitia (0, 0) */
    if (step == 0) {
        /* adaugam (0, 0) la solutia candidat */
        solution.push_back({0, 0});

        /* apelam backtracking recursiv la pasul urmator */
        back(step + 1, stop, walls, solution, line_moves, col_moves);

        /* scoatem (0, 0) din solutie */
        solution.pop_back();
        return;
    }

    /* sunt doar doua mutari pe care le pot face intr-un pas: dreapta si jos;
    acestea sunt encodate prin vectorii de directii line_moves[2] = {0, 1} si
    col_moves[2] = {1, 0} care reprezinta la indicele 0 miscarea in dreapta, iar
    la indicele 1 miscarea in jos */
    for (unsigned int i = 0; i < 2; ++i) {
        /* cream noua linie si noua coloana cu ajutorul vectorilor de directii */
        int new_line = solution.back().first + line_moves[i];
        int new_col = solution.back().second + col_moves[i];
        int n = (stop + 1) / 2;
```

```
        /* daca linia si coloana sunt valide (nu ies din matrice) si nu este
        zid pe pozitia lor, putem continua pe acea celula */
        if (new_line < n && new_col < n && walls[new_line][new_col] == 0) {
            /* adaugam noua celula in solutia candidat;
            NOTE: {new_line, new_col} este echivalent cu
            std::pair<int, int>(new_line, new_col) si se numeste "initializer
            list", feature in C++11 */
            solution.push_back({new_line, new_col});

            /* apelam backtracking recursiv la pasul urmator */
            back(step + 1, stop, walls, solution, line_moves, col_moves);

            /* scoatem celula adaugata din solutie */
            solution.pop_back();
        }
    }
}

int main() {
    /* initializam solutia ca vector de perechi de int-uri */
    std::vector<std::pair<int, int> > solution;

    fin >> n;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            /* citim matricea zidurilor; 1 pentru zid, 0 altfel */
            fin >> walls[i][j];
        }
    }

    /* apelam back cu step = 0, stop = 2 * n - 1 deoarece in 2 * n - 1
    pasi soricelul va ajunge la branza, vectorul de ziduri, vectorul in
    care vom stoca solutia, vectorii de directii line_moves[2] = {0, 1} si
    col_moves[2] = {1, 0}; nu avem nevoie de domeniu deoarece folosind
    vectorii de directii vom sti din ultima pozitie pusa in solutie cele
    doua solutii in care putem merge, astfel domeniul nostru va fi alcatuit
    din doua solutii la fiecare pas (daca ultima pozitie din solutie a fost
    (5, 7) => domeniul pasului curent = {(5 + 0, 7 + 1) si (5 + 1, 7 + 0)}
    care este egal cu {(5, 8), (6, 7)}. */
    back(0, 2 * n - 1, walls, solution, line_moves, col_moves);
}
```

Pentru aceasta abordare la fiecare pas de backtracking vom merge doar in doua directii, in loc sa mergem in oricare celula din matrice, lucru care imbunatateste semnificativ complexitatea temporala.

Complexitate

Soluția va avea urmatoarele complexitati:

- complexitate temporală : $T(n) = O(2^{2n})$
  - explicație: avem de urmat un șir de $2n - 1$ mutari, iar la fiecare pas avem 2 variante posibile
- complexitate spatiala : $S(n) = O(n)$
  - explicație: stocam maximum $2n - 1$ căsuțe

Exerciții

Scheletul de laborator se găsește pe pagina pa-lab::skel/lab05 [https://github.com/acs-pa/pa-lab/tree/main/skel/lab05].

Aranjamente

Fie N și K două **numere naturale strict pozitive**. Se cere afișarea tuturor aranjamentelor de N elemente luate cate K din mulțimea {1, 2, …, N}.

Fie N = 3, K = 2 ⇒ M = {1, 2, 3}

Soluție:

- {1, 2}
- {1, 3}
- {2, 1}
- {2, 3}
- {3, 1}
- {3, 2}

Se dorește o complexitate $T(n, k) = A(n, k)$.

Folosiți-vă de problema **Permutări**.

Soluțiile se vor genera în ordine lexico-grafica!

Checkerul așteaptă să le stocați în această ordine.

Submulțimi

Fie N un **număr natural strict pozitiv**. Se cere afișarea tuturor submulțimilor mulțimii {1, 2, …, N}.

Fie N = 4 ⇒ M = {1, 2, 3, 4}

Soluție: {} - mulțimea vidă {1} {1, 2} {1, 2, 3} {1, 2, 3, 4} {1, 2, 4} {1, 3} {1, 3, 4} {1, 4} {2} {2, 3} {2, 3, 4} {2, 4} {3} {3, 4} {4}

Se dorește o complexitate $T(n) = O(2^n)$.

Folosiți-vă de problema **Combinari**.

Soluțiile se vor genera în ordine lexico-grafica!

Checkerul așteaptă să le stocați în această ordine.

Problema damelor

Problema damelor (sau problema reginelor) tratează plasarea a 8 regine de sah pe o tablă de șah de dimensiuni 8 x 8 astfel încat să nu existe două regine care se amenință reciproc. Astfel, se caută **o soluție** astfel încât nicio pereche de doua regine să nu fie pe același rând, pe aceeași coloană, sau pe aceeași diagonală. Problema cu opt regine este doar un caz particular pentru problema generală, care presupune plasarea a N regine pe o tablă de șah N x N în aceleasi condiții. Pentru această problemă, există soluții pentru toate numerele naturale N cu excepția lui N = 2 si N = 3.

Fie N = 5

Soluție:

X	-	-	-	-
-	-	X	-	-
-	-	-	-	X
-	X	-	-	-
-	-	-	X	-

X reprezintă o damă, - reprezintă spațiu gol.

E nevoie să facem backtracking pe matrice sau e suficient pe vector?

Se va caută o singură soluție (**oricare** soluție corectă), care va fi returnată sub forma unui vector cu $n + 1$ elemente.

Soluția este $sol[0], sol[1], \ldots, sol[n]$, unde $sol[i]$ = coloana unde vom plasa regina de pe linia i.

Elementul 0 este nefolosit, dorim să păstrăm convenția cu indexare de la 1.

Generare de șiruri

Vi se dă o listă de caractere și o lista de frecvențe (pentru caracterul de pe poziția i, frecvența de pe poziția i). Vi se cere să generați toate șirurile care se pot forma cu aceste caractere și aceste frecvențe știind că nu pot fi mai mult de K apariții consecutive ale aceluiași caracter.

Fie caractere[] = {'a', 'b', 'c'}, freq[] = {1, 1, 2}, K = 5

Soluție:

- abcc
- acbc
- accb
- bacc
- bcac
- bcca
- cabc
- cacb
- cbac
- cbca
- ccab
- ccba

Fie caractere[] = {'b', 'c'}, freq[] = {3, 2}, K = 2

Solutie:

- bbcbc
- bbccb
- bcbbc
- bcbcb
- bccbb
- cbbcb
- cbcbb

Soluțiile se vor genera în ordine lexico-grafica!

Checkerul așteaptă să le stocați în această ordine.

Bonus

Problema damelor (AC3)

Aplicați AC3 pe problema damelor.

Algoritmul AC-3 (Arc Consistency Algorithm) este de obicei folosit în probleme de satisfacere a constrângerilor (CSP). Acesta șterge arcele din arborele de stări care sigur nu se vor folosi niciodata.

AC-3 lucrează cu:

- constrângeri
- variabile
- domenii de variabile

O variabilă poate lua orice valoare din domeniul său la orice pas. O constrângere este o relație sau o limitare a unor variabile.

Exemplu AC-3

Considerăm A, o variabilă ce are domeniul D(A) = {0, 1, 2, 3, 4, 5, 6} și B o variabila ce are domeniul D(B) = {0, 1, 2, 3, 4}. Cunoaștem constrângerile: C1 = “A trebuie să fie impar” și C2 = “A + B trebuie să fie egal cu 5”.

Algoritmul AC-3 va elimina în primul rând toate valorile pare ale lui A pentru a respecta C1 ⇒ D(A) = {1, 3, 5}. Apoi, va încerca să satisfacă C2, așa că va păstra în domeniul lui B toate valorile care adunate cu valori din D(A) pot da 5 ⇒ D(B) = {0, 2, 4}.

AC-3 a redus astfel domeniile lui A si B, reducând semnificativ timpul folosit de algoritmul backtracking.

Extra

Immortal

Enunt [https://www.infoarena.ro/problema/immortal]

OJI 2010 - clasele 11-12 [http://olimpiada.info/oji2010/index.php?cid=arhiva]
Articolul de pe leetcode [https://leetcode.com/tag/backtracking/] conține o listă cu diverse tipuri de probleme de programare dinamică, din toate categoriile discutate la PA (plus multe altele).

Referințe

[0] Chapter **Backtracking**, “Introduction to Algorithms”, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein