Outline

Cuprins

1 Paradigme de proiectare a algoritmilor

Conform DEX, *paradigmă* înseamnă *model* sau *exemplu*. O paradigmă de proiectare a algoritmilor este o metodă generală de a rezolva o clasă de probleme. Dintre cele mai importante paradigme de proiectare a algoritmilor fac parte divide et impera (vezi cursul Structuri de Date), greedy (acest curs), programare dinamică (cursurile următoare) și backtracking.

2 Probleme de optimizare

Paradigma greedy este folosită în special pentru a rezolva *probleme de optimizare*, adică probleme în care datele de intrare au orice formă, iar datele de ieșire presupun maximizarea sau minimizarea unei anumite cantități, respectând anumite constrângeri:

> Problemă de optimizare:
> `Input`: ...
> `Output`: cel mai mare/cel mai mic număr cu proprietatea că ...

Un exemplu de problemă de optimizare este găsirea celui mai scurt drum între două noduri într-un graf:

> Problema celui mai scurt drum într-un graf:
> `Input`: Un graf $G = (V, E)$ și două noduri $s, t \in V$.
> `Output`: Lungimea n a celui mai scurt drum de la s la t în graf.

Într-o problemă de optimizare, este posibil ca datele de ieșire să conțină informații suplimentare. De exemplu, pentru problema celui mai scurt drum într-un graf, putem cere și drumul de lungime minimă (nu doar lungimea acestuia:

Problema celui mai scurt drum într-un graf (cu output explicit):
`Input`: Un graf $G = (V, E)$ și două noduri $s, t \in V$.
`Output`: $v_1, \ldots, v_n$ - cel mai scurt drum între s și t.

În contextul acestei probleme, orice drum între s și t se numește *soluție*, iar un drum între s și t mai scurt decât toate celelalte drumuri între s și t este *soluția optimă*. În general, o problemă de optimizare cere să alegem din mulțimea de soluții posibile (în exemplul de mai sus, mulțimea de drumuri între s și t), soluția optimă (cea de cost minim, sau cea de câștig maxim – în funcție de cerința problemei).

O problemă de optimizare este rezolvată corect de un algoritm dacă aceasta produce în toate cazurile o soluție optimă.

Exerciţiul 1. Dați alte exemple de probleme de optimizare pe care le-ați întâlnit până acum.

3 Paradigma greedy

Greedy este una dintre cele mai simple paradigme de programare. Multe probleme de optimizare pot fi rezolvate eficient folosind algoritmi greedy. Algoritmii greedy sunt de obicei foarte simplu de implementat, dar nu neapărat simplu de demonstrat.

Pentru alte probleme de optimizare, algoritmii greedy nu produc soluția optimă. În aceste cazuri, există două strategii:

1. Dacă se dorește o soluție optimă, se caută o altă metodă de rezolvare decât greedy (folosind, e.g., backtracking sau programare dinamică);

2. Dacă diferența dintre costul optim și costul produs de algoritmul greedy este tolerabilă în practică, se poate folosi algoritmul greedy (algoritmul produce o soluție *aproximativă*, în acest caz se numește algoritm de aproximare).

4 Exemple de probleme care pot fi rezolvate prin strategii greedy

Algoritmii de tip *greedy* (lacom) sunt aplicați deseori în practică, inclusiv în viața de zi cu zi. De exemplu, dacă plecați într-o excursie și vă pregătiți bagajul, veți introduce întâi în bagaj lucrurile mari și apoi cele mici. Această strategie vă ajută să folosiți spațiul din bagaj în mod eficient.

4.1 Bin-packing

Problemele computaționale în care la intrare se dau un container și o mulțime de obiecte și la ieșire se cere o modalitate de a aranja obiectele în container se numesc probleme de tip *bin-packing* și apar în diferite domenii.

De exemplu, în contextul jocurilor 2D, mai multe *sprite*-uri (imagini mici (e.g. 128x128 sau 64x64) sau serii de imagini mici reprezentând un personaj sau un alt element grafic din joc) trebuie așezate fără să se suprapună într-un

număr cât mai mic de imagini de dimensiuni mai mari (e.g. 1024x1024) care pot fi încărcate de placa video ca texturi și apoi afișate pe ecran cât mai eficient. Folosirea unui număr cât mai mic de texturi crește performanța jocului și din acest motiv este de dorit să "înghesuim" cât mai multe sprite-uri într-o singură textură. Cele mai multe framework-uri pentru programarea jocurilor conțin implementarea unor algoritmi pentru sprite-packing (căutați "sprite packer" folosind un motor de căutare).

În cele mai multe cazuri, pentru problemele de tip bin-packing nu se cunoaște o rezolvare polinomială și din acest motiv se preferă un algoritm greedy care calculează o soluție aproximativă.

4.2 Plata unei sume folosind număr minim de bancnote

Un alt exemplu de problemă unde aplicăm un algoritm greedy este plata unei sume de bani folosind un număr minim de bancnote. De exemplu, pentru a plăti (exact) 86 de lei, folosim 6 bancnote: una de 50 de lei, trei de 10 lei, una de 5 lei și una de 1 leu. Orice altă metodă de a plăti suma de 86 de lei folosește mai multe bancnote. Formal, problema poate fi exprimată astfel:

> Problema plății cu număr minim de bancnote.
> `Input`: un număr natural n – suma ce trebuie plătită
> `Output`: numerele $n_{500}, n_{200}, n_{100}, n_{50}, n_{10}, n_5, n_1$
> (n_i - câte bancnote de i RON folosesc), astfel încât
> $\Sigma_{i \in \{500,200,100,50,10,5,1\}} n_i$ să fie minimă și
> $n = \Sigma_{i \in \{500,200,100,50,10,5,1\}} i \times n_i$.

Strategia greedy pe care o folosim este să plătim mereu cea mai mare bancnotă care este mai mică decât suma pe care o avem de achitat. De exemplu, pentru 86 de lei, cea mai mare bancnotă este de 50 de lei. Rămân de achitat 36 de lei. Cea mai mare bancnotă este de 10 lei. Rămân de achitat 26 de lei. (...).

Se poate demonstra că această strategie conduce la un număr minim de bancnote, pentru sistemul de bancnote pe care îl avem în țară (bancnote de 500, 200, 100, 50, 10, 5, 1). Pentru alte sisteme de bancnote, strategia greedy nu conduce tot timpul la o soluție optimă.

Exerciţiul 2. Imaginați-vă că trăiți într-o țară în care sunt disponibile bancnote de 1 leu, de 7 lei și de 8 lei. Dați exemplu de o sumă de bani pentru care strategia greedy descrisă mai sus nu produce soluția optimă.

4.3 Problema selecției activităților

Azi, un student poate participa la una sau mai multe activități. De exemplu:

curs de la 10 la 12 poate participa la cursul PA;

codecamp de la 9 la 17 poate participa la CodeCamp;

teatru de la 18 la 20 poate merge la teatru;

film de la 21 la 22 poate merge la film;

club de la 19 la 24 poate merge în club.

Scopul studentului este să participe la **cât mai multe activități** (dar trebuie să participe la fiecare de la început la sfârșit și nu poate fi în două locuri în același timp). Ce activități trebuie să aleagă? În exemplul de mai sus, studentul poate participa la maxim 3 activități: curs, teatru, film sau codecamp, teatru, film.

Formal, problema selecției activităților este următoarea:

`Input`: n - numărul de activități
$s[0..n-1]$ - un tablou care conține timpul de început al activităților
$f[0..n-1]$ - un tablou care conține timpul de final al fiecărei activități
a.î. $s[i] < f[i]$ pentru orice $0 \leq i \leq n-1$ și
f este în ordine crescătoare.
`Output`: $A \subseteq \{0, \ldots, n-1\}$
A este o mulțime de activități care nu se suprapun
A este de cardinal maxim.

Considerăm că activitatea i începe exact în momentul s_i și se termină puțin înainte de momentul f_i. Cu alte cuvinte, activitatea i durează de la s_i (inclusiv) până la f_i (exclusiv). Astfel, două activități i, j se suprapun dacă $[s_i, f_i) \cap [s_j, f_j) \neq \emptyset$ (remarcați faptul că intervalele sunt închise la stânga și deschise la dreapta).

Exerciţiul 3. Arătați că două activități i, j nu se suprapun (sunt compatibile între ele) dacă și numai dacă $s_i \geq t_j$ sau $s_j \geq t_i$.

De exemplu, pentru $n = 5$ și pentru tablourile s și f date mai jos:

i	0	1	2	3	4
$s[i]$	10	9	18	21	19
$f[i]$	12	17	20	22	24,

un răspuns corect este: $A = \{0, 2, 4\}$.

IDEE 1.

O primă idee de rezolvare a problemei printr-o strategie greedy este să aleg activitățile în ordinea duratei lor. De exemplu, în exemplul de mai sus, aleg activitatea 4 deoarece durează 1 oră, apoi activitatea 1 și activitatea 3, deoarece durează amândouă 2 ore. Totuși, această strategie nu conduce în general la o soluție optimă. De exemplu, pentru următoarele activități:

i	0	1	2
$s[i]$	9	15	17
$f[i]$	16	18	23,

strategia descrisă mai sus alege activitatea 1, care are durată de 3 ore. Dar activitatea 1 se suprapune atât cu 0 cât și cu 2 și deci nu mai putem alege nicio altă activitate. Soluția $\{1\}$ nu este optimă, deoarece soluția $\{0, 2\}$ conține mai multe activități.

IDEE 2.

În continuare, vom vedea o strategie greedy pentru problema selecției activităților care conduce în toate cazurile la soluția optimă.

Strategia pe care o vom folosi:

1. începem cu mulțimea $A = \emptyset$;

2. dintre activitățile care au rămas, alegem activitatea i care se **termină cel mai devreme** (intuitiv, deoarece îmi lasă timp mai mult pentru următoarele activități);

3. adăugăm i la A;

4. ștergem i și toate celelalte activități care se suprapun cu i din lista de activități disponibile;

5. repetăm procesul dacă mai sunt activități disponibile.

În pseudocod, strategia de mai sus poate fi descrisă astfel:

- $A = \emptyset$ (mulțimea de activități selectate)
- $time = 0$ (timpul începând cu care sunt disponibil)
- `for` i $= 0$ `to` $n - 1$ (activitățile sunt deja în ordine descrescătoare a timpului de final)
  - if $s[i] >= time$ (sunt disponibil pentru activitatea i)
    - $A = A \cup \{i\}$ (selectez activitatea i)
    - $time = f[i]$ (marchez că nu pot accepta activități mai devreme de $f[i]$)

Întrebare. Strategia de rezolvare IDEE 1 nu produce tot timpul soluția optimă. Cum pot fi sigur ca strategia descrisă în IDEE 2 produce tot timpul soluția optimă? **Răspuns**. Putem demonstra acest lucru.

În primul rând, vom arăta că, pentru orice mulțime S de activități, există o submulțime S' de activități care:

- conține doar activități compatibile între ele (care nu se suprapun),
- este de cardinal maxim și
- conține activitatea care se termină cel mai devreme din S.

Cu alte cuvinte, fără a pierde din generalitate, într-o mulțime de cardinal maxim de activități compatibile între ele se poate alege activitatea care se termină cel mai devreme:

Lemă 1 (Lema de alegere lacomă)**.** *Fie $S \subseteq \{0, 1, \ldots, n-1\}$ o mulțime nevidă de activități (nu neapărat compatibile între ele).*

Fie x activitatea din S care se termină cel mai devreme.

Există o submulțime $S' \subseteq S$ a mulțimii S care:

- *conține doar activități compatibile între ele;*
- *este de cardinal maxim (dintre toate submulțimile lui S de activități compatibile între ele);*
- *conține activitatea x: $x \in S'$.*

Proof. Fie S'' o submulțime oarecare a lui S de activități compatibile între ele de cardinal maxim. Nu știm dacă $x \in S''$.

Fie y activitatea din S'' care se termina cel mai devreme.

Alegem $S' = (S'' \setminus \{y\}) \cup \{x\}$.

Observăm că:

- S' are același cardinal cu S'';
- S' conține x.

Mai avem de arătat că S'' conține doar activități compatibile între ele. Cum y este activitatea care se termină cel mai devreme din S' și cum S' conține doar activități compatibile între ele, rezultă că $\forall z \in S' \setminus \{y\}, t[y] \leq s[z]$. Dar $y \in S'$, $S' \subseteq S$ și $x \in S$ este activitatea din S care se termină cel mai devreme. Deci x se termină înaintea lui y: $t[x] \leq t[y]$. Rezultă că $\forall z \in S' \setminus \{y\}, t[x] \leq s[z]$, ceea ce înseamnă că x este compatibilă cu toate activitățile din $S' \setminus \{y\}$, deci S' conține doar activități compatibile între ele.

□

Am arătat că, fără a pierde optimalitatea, putem alege activitatea care se termină cel mai devreme.

Pentru a arăta că algoritmul, în întregimea sa, produce o soluție optimă, a mai rămas de arătat următorul aspect: combinarea activității care se termină cel mai devreme cu restul activităților alese de algoritm este o soluție optimă. Pentru a formaliza acest lucru, avem nevoie de noțiunea de subproblemă:

Definiţie 1. *Notăm cu $S_t = \{i \in S \mid s[i] \geq t\}$ submulțimea de activități care încep după momentul t.*

Vom spune că mulțimile S_t sunt *subproblemele* problemei inițiale. Subproblema S_0 coincide cu problema inițială. Putem înțelege strategia greedy prezentată mai sus ca rezolvând, rând pe rând, mai multe subprobleme de forma S_t.

Cu ajutorul lemei de mai sus, putem demonstra proprietatea de substructură optimă:

Lemă 2 (Proprietatea de substructură optimă)**.** *Fie S_t o subproblemă și $x \in S_t$ activitatea aleasă prin strategia greedy (cea care se termină cel mai devreme). Fie $A \subseteq S_t$ o submulțime de activități compabilite între ele de cardinal maxim care conține x ($x \in A$ – prin Lema de alegere greedy, această mulțime A există).*

Fie $B = A \setminus \{x\}$. Atunci B este o soluție optimă pentru $S_{f[x]}$.

Proof. Presupunem că există o soluție C mai bună decât B pentru subproblema $S_{f[x]}$. Atunci $C \cup \{x\}$ ar fi o soluție pentru S_t mai bună decât A, ceea ce contrazice optimalitatea lui A.

□

Demonstrația proprietății de substructură optimă este în general foarte simplă. Proprietatea de substructură optimă ne spune că subsoluțiile unei soluții optime sunt la rândul lor soluții optime pentru subprobleme.

Prin inducție, folosindu-ne de cele două leme de mai sus, este ușor de demonstrat corectitudinea algoritmului:

Teoremă 1. *Strategia greedy de mai sus produce o soluție optimă.*

Proof. Vom proceda prin inducție după mulțimea S de activități disponibile.

Prin lema de alegere greedy, există o soluție optimă A pentru S astfel încât $x \in A$. Prin ipoteza de inducție, algoritmul găsește o soluție optimă B pentru subproblema $T = \{i \in S \mid s[i] \geq f[x]\}$ (T este strict inclusă în S). Prin lema de substructură optimă, $\{x\} \cup A$ este o soluție optimă (cea găsită de algoritm).

□

O demonstrație alternativă, care poate fi mai simplu de înțeles fiindcă pasul inductiv este explicitat:

Proof. Fie $s_1, \ldots, s_k$ lista de activități selectată de algoritmul greedy. Presupunând că nu e optimă, căutăm cel mai mic număr $j \in \{1, 2, 3, \ldots\}$ astfel încât să existe o listă optimă de activități care conține primele $j-1$ activități din $s_1, \ldots, s_n$ și apoi alte activități:

$$s_1, \ldots, s_{j-1}, t_j, \ldots, t_m.(t_j \neq s_j)$$

Presupunem că activitățile din soluția optimă $s_1, \ldots, s_{j-1}, t_j, \ldots, t_m$ sunt ordonate crescător (fiind compatibile între ele, nu contează dacă după timpul de început sau după cel de sfârșit – obținem aceeași ordine).

Prin definiție, s_j este compatibilă cu $s_1, \ldots, s_{j-1}$ și începe mai devreme decât t_j. Deci $s_1, \ldots, s_{j-1}, s_j, t_{j+1}, \ldots, t_m$ ar fi și ea o soluție optimă, contrazicând minimalitatea lui j.

□

Exerciţiul 4. Ce se întâmplă dacă problema nu garantează că vectorul f este în ordine crescătoare?

Arătați că cele două probleme (în care f este ordonat și respectiv în care f nu este neapărat ordonat) se reduc una la cealaltă.

4.4 Algoritmii greedy

Algoritmii greedy

1. Algoritmii greedy = secvență de alegeri locale, alegeri care par a fi cele mai bune în momentul respectiv;
2. Odată făcută o alegere, nu ne putem răzgândi.
3. Câteodată această soluție conduce la un optim global (e.g. problema selecției activităților); alteori nu (e.g. problema discretă a rucsacului).

Algoritmii greedy - proces de proiectare

1. Identificăm subprobleme ale problemei de optimizare (e.g. $S_{time} = \{i \mid s[i] \geq time\}$ astfel încât o alegere greedy într-o subproblemă să conducă la o altă subproblemă).
2. (greedy-choice property) Arătăm că există o soluție optimă a problemei inițiale care folosește alegerea greedy.
3. (optimal substructure property) Arătăm că dacă facem o alegere greedy, combinația dintre alegerea greedy și o soluție optimă pentru subproblema rezultată este o soluție optimă pentru problema inițială.

4.5 Problema rucsacului - varianta continuă

Problema rucsacului - varianta continuă

Un hoț a spart un magazin și a găsit n bunuri. Al i-lea bun valorează v_i lei și cântărește w_i kilograme. Hoțul are un rucsac care poate să ducă cel mult W kilograme de bunuri. Orice bun poate fi secționat iar valoarea unei părți este proporțională cu dimensiunea acesteia (e.g. jumătate dintr-un obiect cu $v_i = 10$ și $w_i = 6$ cântărește 3 kg și valorează 5 lei). Hoțul vrea să maximizeze valorea obiectelor pe care le va pune în rucsac.

Problema rucsacului - formalizare varianta continuă

`Input`: n, $v[0..n-1]$, $w[0..n-1]$, W, toate numere naturale
`Output`: $p[0..n-1]$, $p[i] \in [0,1]$ astfel încât:

1. $\Sigma_i p[i]w[i] \leq W$ (părțile alese ale obiectele încap în rucsac)
2. $\Sigma_i p[i]v[i]$ este maxim (valoarea părților este maximă)

Problema rucsacului - exemplu instanță

Avem $n = 3$ obiecte:

i	0	1	2
$w[i]$	1	2	3
$v[i]$	10	15	20

Rucsacul are capacitate de $W = 5$ kg.
O soluție optimă pentru varianta continuă:

1. iau 100% din primul obiect (deci $p[0] = 1$). Câștig: $1 \times 10 = 10$, capacitate rămasă: $5 - 1 = 4$ kg.
2. iau 100% din al doilea obiect (deci $p[1] = 1$). Câștig: $10 + 1 \times 15 = 25$, capacitate rămasă: $4 - 2 = 2$ kg.
3. iau 2/3 din al treilea obiect (deci $p[2] = 0.66\ldots$). Câștig: $25 + 2/3 \times 20 = 38.33\ldots$.

O abordare greedy care conduce la soluția optimă este să alegem cât mai mult din obiectul cu câștig unitar (câștig / kilogram) cel mai mare.

Exerciţiul 5. Identificați subproblemele pentru problema rucsacului - varianta continuă.

Enunțați și demonstrați lema de alegere greedy.

Enunțați proprietatea de substructură optimă (și demonstrați-o, dar demonstrația va fi surprinzător de simplă).

Problema rucsacului - varianta discretă

Un hoț a spart un magazin și a găsit n bunuri. Al i-lea bun valorează v_i lei și cântărește w_i kilograme. Hoțul are un rucsac care poate să ducă cel mult W kilograme de bunuri. **Niciun bun nu poate fi secționat - obiectul i trebuie furat integral sau deloc**. Hoțul vrea să maximizeze valorea obiectelor pe care le va pune în rucsac.

Problema rucsacului - formalizare varianta discretă

`Input`: n, $v[0..n-1]$, $w[0..n-1]$, W, toate numere naturale
`Output`: $p[0..n-1]$, $p[i] \in \{0,1\}$ astfel încât:

1. $\Sigma_i p[i]w[i] \leq W$ (obiectele încap în rucsac)
2. $\Sigma_i p[i]v[i]$ este maxim (valoarea obiectelor este maximă)

Problema rucsacului - exemplu instanță

Avem $n = 3$ obiecte:

i	0	1	2
$w[i]$	1	2	3
$v[i]$	10	15	20

Rucsacul are capacitate de $W = 5$ kg.
O soluție optimă pentru varianta discretă:

1. nu iau primul obiect ($p[0] = 0$);
2. iau al doilea obiect ($p[1] = 1$);
3. iau al treilea obiect ($p[2] = 2$).

Abordările greedy nu produc soluții optime pentru varianta discretă (vezi capitolele următoare: programare dinamică, backtracking).

4.6 Problema codurilor Huffman

Motivație

Să presupunem că avem un fișier care conține 100 de caractere. Caracterul a apare de 45 de ori, b de 30 de ori, c de 10 ori și d de 15 ori.

Fișierul inițial ocupă $100 * 8 = 800$ de biți.

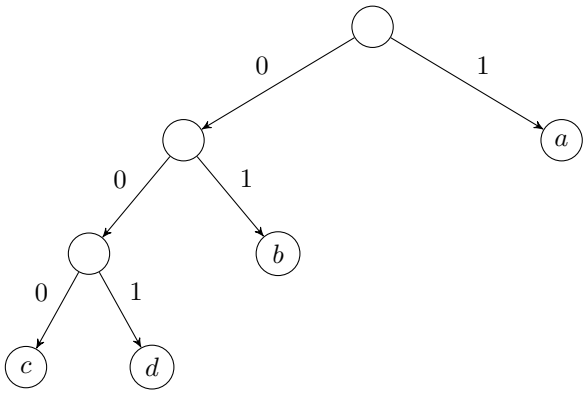
Deoarece în fișier apar doar caracterele a, b, c, d, putem comprima fișierul înlocuind fiecare apariție a lui a cu secvența de biți 00, b cu 01, c cu 10 și d cu 11.

Obținem în acest fel $100 * 2 = 200$ de biți.

Se poate obține o compresie mai bună? Da, folosind coduri de lungime variabilă: $a = 1$, $b = 01$, $c = 000$, $d = 001$. Fișierul comprimat este reprezentat acum prin $1 * 45 + 2 * 30 + 3 * 10 + 3 * 15 = 180$ de biți (90% din dimensiunea obținută folosind prima codare).

Coduri prefix

Orice astfel de cod se poate reprezenta în mod unic printr-un arbore binar cu n frunze:



Formalizarea problemei

`Input`: n - numărul de caractere distincte; $c[0..n-1]$ - numărul de apariții ale fiecărui caracter.

`Output`: un arbore binar care reprezintă codul prefix optim.

Exemplu: dacă $n = 4$ și $c = [45, 30, 15, 10]$, atunci arborele precedent este un răspuns corect ($a = 1$, $b = 01$, $c = 000$, $d = 001$).

Ideea: combină caracterele cu frecvențele cele mai reduse (pe tablă).

Algoritmul greedy pentru coduri prefix

1. $Q = \emptyset$ (coadă cu prioritate)
2. `for` $i = 0$ `to` $n - 1$
   $Q.insert(info : i, frec : c[i], left : null, right : null)$
3. `for` $i = 0$ `to` $n - 2$
   $x = new()$
   $x.info = -1$
   $x.left = Q.extractmin()$
   $x.right = Q.extractmin()$
   $x.frec = x.left.frec + x.right.frec$
   $Q.insert(x)$
4. `return` $Q.extractmin()$

Corectitudinea și analiza algoritmului

1. (Proprietatea de alegere greedy) Există un cod optim în care ultimele două caractere dpdv al frecvenței au aceeași lungime și diferă doar în ultimul bit.
2. (Proprietatea de substructură optimă) Fie x, y cele două caractere cu frecvența cea mai mică. Fie z un nou caracter astfel încât $c[z] = c[x] + c[y]$. Fie T arborele binar care determină codul optim pentru $\Sigma \setminus \{x, y\} \cup \{z\}$ și T' arborele binar obținut din T prin înlocuirea lui z cu un nou nod având copiii x și y. Atunci T' este optim pentru Σ.

3. Timp de rulare: $O(nlogn)$, dacă coada cu prioritate este implementată printr-un heap binar.

4.7 Matroizi

Matroid

Mulți algoritmi de tip greedy pot fi demonstrați folosind teoria matroizilor.

Definiţie 2. *Un matroid este o pereche $M = (S, I)$ cu proprietățile:*

1. *S este o mulțime finită*
2. *(ereditate) I este o mulțime nevidă de submulțimi ale lui S (numite mulțimi independente) și*

$$\text{dacă } B \in I \text{ și } A \subseteq B, \text{ atunci } A \in I.$$

3. *(interschimbare) dacă $A \in I, B \in I$ și $|A| < |B|$, atunci există $x \in B \setminus A$ astfel încât $A \cup \{x\} \in I$.*

Exerciţiul 6. Demonstrați că $\emptyset \in I$ (Hint: folosiți proprietatea de ereditate).

Matroid - exemplu

Fie $G = (V, E)$ un graf și $M_G = (S_G, I_G)$ definit astfel:

1. $S_G = E$, muchiile grafului
2. $A \subseteq E \in I$ ddacă A nu conține ciclu.

Exerciţiul 7. Arătați că M_G este matroid.

Exerciţiul 8. Arătați că, dacă la itemul 2 cerem ca A să fie arbore, M_G nu este matroid.

Exerciţiul 9. Arătați că orice mulțime maximală (dpdv al incluziunii) din I are același cardinal.

Matroid ponderat

Fie $M = (S, I)$.

Fie $w : S \to \mathbb{N}$ o funcție care asociază fiecărui element x din S o pondere $w(x)$.

Funcția w se extinde la mulțimi $A \subseteq S$ astfel: $w(A) = \Sigma_{x \in A} w(x)$.

Teoremă 2. *Dacă M este matroid, atunci există un algoritm greedy pentru găsirea unei mulțimi independente de pondere maximă.*

Algoritmul greedy (general pentru matroizi)

1. $A = \emptyset$ (încep cu mulțimea vidă)
2. sortează $M.S$ în ordine descrescătoare a ponderilor
3. `for each` $x \in M.S$ (în ordine descrescătoare)
   - `if` $A \cup \{x\} \in I$
     $A = A \cup \{x\}$

Teoremă 3. *Algoritmul găsește o mulțime independentă de pondere maximă.*

Pentru a demonstra teorema de mai sus, este suficient să arătăm că matroizii dețin cele două proprietăți prin care am demonstrat corectitudinea algoritmilor greedy:

Lemă 3 (Proprietatea de alegere greedy pentru matroizi ponderați)**.** *Fie $M = (S, I)$ un matroid. Fie $w : S \to \mathbb{N}$ funcția de ponderare. Fie x cel mai mare (dpdv al funcției de ponderare) element din S astfel încât $\{x\} \in I$ (mulțimea singleton alcătuită doar din x este independentă).*

Atunci există o submulțime optimă (dpdv al funcției de ponderare) $A \subseteq S$ astfel încât $x \in A$.

Lemă 4 (Proprietatea de substructură optimă)**.** *Fie x primul element ales de algoritm pentru matroidul $M = (S, I)$.*

Fie matroidul $M' = (S', I')$, definit astfel:

- $S' = \{y \in S \mid \{x, y\} \in I\}$;
- $I' = \{B \subseteq S \setminus \{x\} \mid B \cup \{x\} \in I\}$.

Fie A soluția optimă găsită de algoritm pentru matroidul M ($x \in A$). Atunci $A \setminus \{x\}$ este soluție optimă pentru M'.

4.8 Problema APM

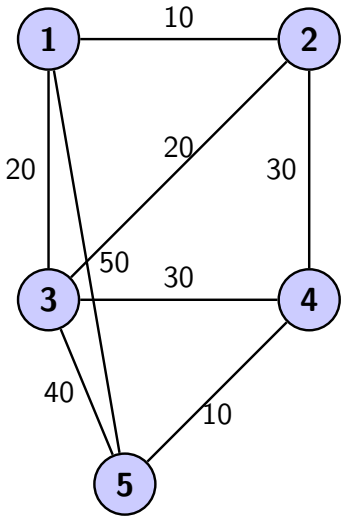
Problema arborelui parțial de cost minim

`Input`: un graf $G = (V, E)$ conex, fiecare muchie $e \in E$ având un cost $l(e)$

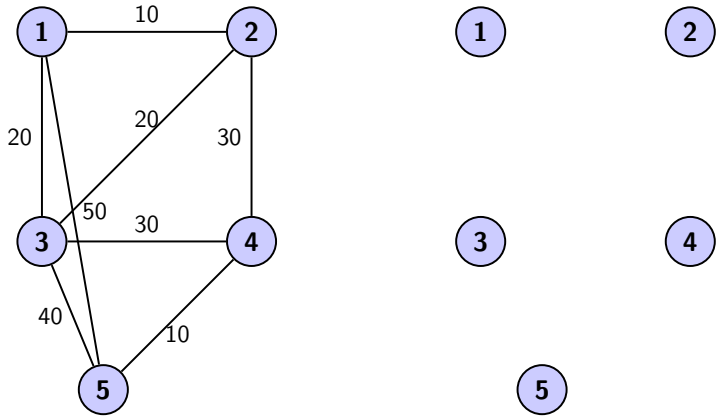
`Output`: un arbore $G = (V, A)$, astfel încât suma muchiilor arborelui să fie de cost minim.

Algoritmul lui Kruskal, pe care l-ați discutat la liceu, începe cu mulțimea vidă de muchii și procesează muchiile în ordin crescătoare a costurilor. Dacă adăugarea unei muchii nu produce ciclu, atunci aceasta este adăugată la soluție. La final, mulțimea de muchii selectate va forma un arbore. Iată o posibilă execuție a algoritmului lui Kruskal pentru graful ce urmează:

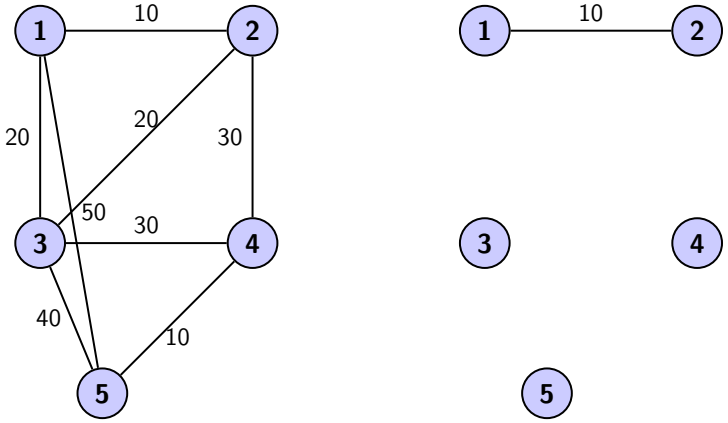
Algoritmul lui Kruskal - instanța



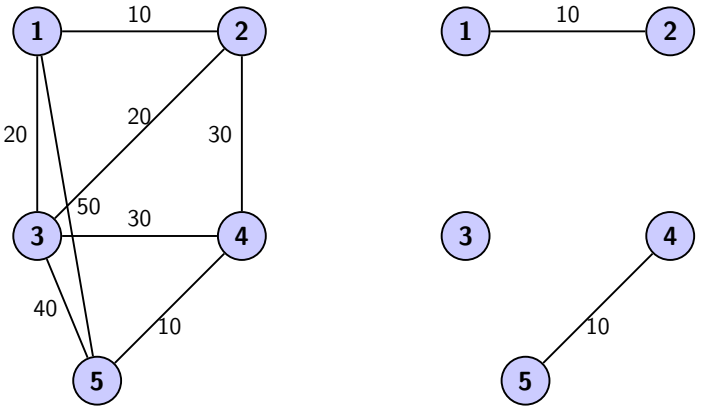
Algoritmul lui Kruskal - pasul 1



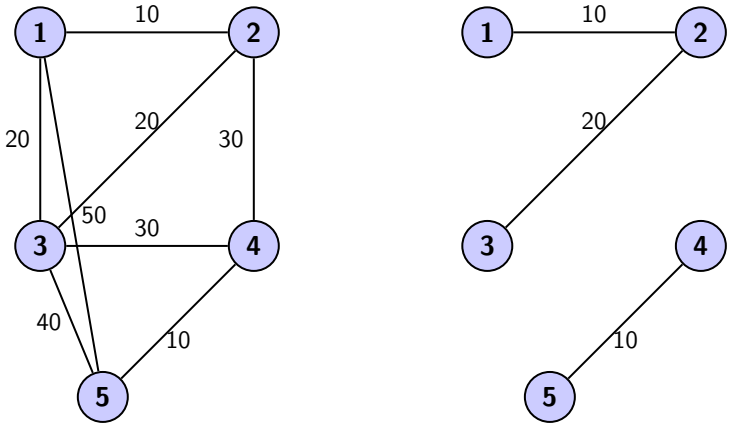
Algoritmul lui Kruskal - pasul 2



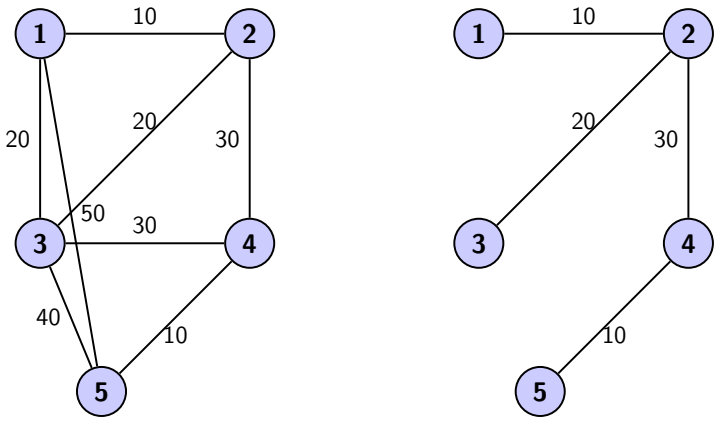
Algoritmul lui Kruskal - pasul 3



Algoritmul lui Kruskal - pasul 4



Algoritmul lui Kruskal - pasul 5



Algoritmul lui Kruskal ca instanță a unui matroid

Fie $G = (V, E)$ graful pentru care aplicăm algoritmul lui Kruskal.
Definim $M_G = (S_G, I_G)$ ca mai sus (am arătat că este matroid):

1. $S_G = E$, muchiile grafului
2. $A \subseteq E \in I$ ddacă A nu conține ciclu.

Exerciţiul 10. Arătați că mulțimile maximale din I sunt arbori.

Dacă definim $w(e) = M - l(e)$ (unde M este un număr suficient de mare), atunci algoritmul lui Kruskal este o instanță a algoritmului general pentru matroizi.

Concluzii

1. Greedy este o paradigmă importantă de proiectare a algoritmilor.
2. De obicei algoritmii greedy sunt ușor de implementat (nu neapărat și de demonstrat). Pentru a demonstra corectitudinea unui algoritm greedy, arătăm că problema are proprietatea de alegere greedy și proprietatea de substructură optimă (trebuie identificate convenabil subproblemele).
3. Multe probleme de optimizare au soluții optime ce pot fi găsite cu greedy (e.g. problema continuă a rucsacului). Pentru alte probleme (e.g. problema discretă a rucsacului), algoritmii greedy nu produc soluția optimă.
4. Matroidul este structură matematică cu ajutorul căreia putem modela diverse probleme de optimizare. Dacă problema se poate modela cu ajutorul unui matroid (ponderat), atunci algoritmul greedy produce soluția optimă. Avantajul este că nu mai avem de demonstrat algoritmul, deoarece am arătat deja că este corect pentru orice matroid. Este suficient să arătăm că putem modela problema ca un matroid.