1 MATLAB a lineární programování

1.1 Solver `linprog`

MATLAB je vybaven toolboxem *Optimization*, kde je k dispozici solver `linprog` pro lineární programování (zdůrazněme, že zde hovoříme výhradně o lineárním programování se spojitými proměnnými; o celočíselném či smíšeném programování budeme hovořit až v kapitole 1.11). V základní podobě se solver volá ve tvaru

$$x^* = \texttt{linprog(c,A,b)}; \tag{1}$$

řeší se tak lineární program (LP) tvaru

$$\min_{x\in\mathbb{R}^n}\{c^{\mathrm{T}}x \mid Ax \le b\} \tag{2}$$

a návratovou hodnotou je optimální řešení x^*. Konvencí je, že vektory c a b se rozumí jako sloupcové; nalezené optimum x^* je rovněž sloupcový vektor.

Poznámka. Jak poznat nepřípustnost (*infeasibility*) či neomezenost (*unboundedness*) LP, to vysvětlíme v (22) na straně 15.

Převod obecného lineárního programu do tvaru (2): několik triků. Připomeňme, že (2) je zcela obecný tvar LP v tom smyslu, že libovolný LP lze do tvaru (2) převést. Vystačíme s několika málo triky:

(a) Je-li cílem řešit LP tvaru $\max\{c^{\mathrm{T}}x \mid Ax \le b\}$, stačí užít pozorování $\max\{c^{\mathrm{T}}x \mid Ax \le b\} = -\min\{-c^{\mathrm{T}}x \mid Ax \le b\}$. Maximalizace funkce $f(x)$ je totiž ekvivalentní minimalizaci funkce $-f(x)$. Stačí proto volat `linprog(-c,A,b)`.

(b) Máme-li LP zapsaný ve tvaru s nerovnostmi „$\le$“ i „$\ge$“, stačí nerovnosti druhého typu přenásobit konstantou -1. Obecně: máme-li řešit lineární program tvaru $\min\{c^{\mathrm{T}}x \mid Ax \le b,\ Dx \ge h\}$, stačí jej přepsat do tvaru

$$\min\left\{c^{\mathrm{T}}x \;\middle|\; \begin{pmatrix} A \\ -D \end{pmatrix} x \le \begin{pmatrix} b \\ -h \end{pmatrix}\right\}$$

a volat `linprog(c, [A;-D], [b;-h])`.

(c) Jsou-li mezi omezujícími podmínkami rovnosti, lze postupovat dvojím způsobem. **První způsob** přepíše rovnost $\alpha = \beta$ jako ekvivalentní dvojici nerovností $\alpha \le \beta$, $-\alpha \le -\beta$. To znamená: máme-li řešit LP tvaru

$$\min\{c^{\mathrm{T}}x \mid Ax \le b,\ Ux = v\}, \tag{3}$$

přepíšeme jej do ekvivalentního tvaru

$$\min\left\{c^{\mathrm{T}}x \;\middle|\; \begin{pmatrix} A \\ U \\ -U \end{pmatrix} x \le \begin{pmatrix} b \\ v \\ -v \end{pmatrix}\right\}$$

a voláme `linprog(c, [A;U;-U], [b;v;-v])`. **Druhý způsob** využívá možnosti volat `linprog` v rozšířené syntaxi oproti základnímu tvaru (2): příkaz

```
linprog(c, A, b, U, v)
```  (4)

přímo řeší LP tvaru (3). Oba způsoby jsou ekvivalentní a záleží jen na preferenci uživatele, který způsob zvolí.

Poznámka. Syntaxe příkazu `linprog` je ještě obecnější (pro detaily je vhodné se podívat do dokumentace `doc linprog`). Například je možné pomocí dalších parametrů zadávat horní a/nebo dolní meze proměnných (což se hodí například při práci s nezápornými proměnnými), je možné zadat počáteční nástřel a je možné nastavovat řadu parametrů solveru, například lze volit algoritmus k řešení LP, numerickou citlivost atd. Za zmínku stojí varianta

$$\texttt{linprog(}c,\ A,\ b,\ U,\ v,\ \underline{x},\ \overline{x}\texttt{)}, \tag{5}$$

která řeší LP tvaru $\min_x\{c^{\mathrm{T}}x \mid Ax \le b,\ Ux = v,\ \underline{x} \le x \le \overline{x}\}$. Nicméně my se zde nebudeme těmito možnostmi dále zabývat a vesměs vystačíme jen se základním tvarem (1) a (4). Budeme-li například potřebovat omezit proměnné shora/zdola, prostě tato omezení zařadíme mezi omezení systému $Ax \le b$ (ale upozorňujeme, že to není jediná možnost). Variantu (5) použijeme — kvůli stručnosti zápisu — až v kapitole 1.11, byť i tam lze vše udělat jen s (1) a/nebo (4).

1.2 Příklad 1: jednoduchý LP

Řešme LP

$$\max 3x_1 + 2x_2 \;\text{ s.t. }\; x_1 + 2x_2 \le 2;\; 4x_1 + 2x_2 \le 3;\; x_1 + x_2 \ge 1;\; x_1, x_2 \ge 0$$

(zde „s.t.“ značí „subject to“, „za omezujících podmínek“; je to běžný žargon). Dle triku (b) přenásobíme nerovnosti typu „$\ge$“ konstantou -1 a a dle triku (a) převedeme maximalizaci na minimalizaci:

$$\min \underbrace{(-3,-2)}_{c^{\mathrm{T}}} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \;\text{ s.t. } \underbrace{\begin{pmatrix} 1 & 2 \\ 4 & 2 \\ -1 & -1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}}_{A} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \le \underbrace{\begin{pmatrix} 2 \\ 3 \\ -1 \\ 0 \\ 0 \end{pmatrix}}_{b}. \tag{6}$$

Všimněme si, že podmínkám $x_1, x_2 \ge 0$ odpovídá blok $-Ix \le 0$ v systému nerovností $Ax \le b$. (Zde I je jednotková matice.) Můžeme proto psát například

```
c = [-3;-2];
A = [1,2; 4,2; -1,-1; -eye(2)];
b = [2;3;-1;zeros(2,1)];
linprog(c,A,b)
```

a zjistíme, že optimum jest $x^* = \binom{0.333}{0.833}$.

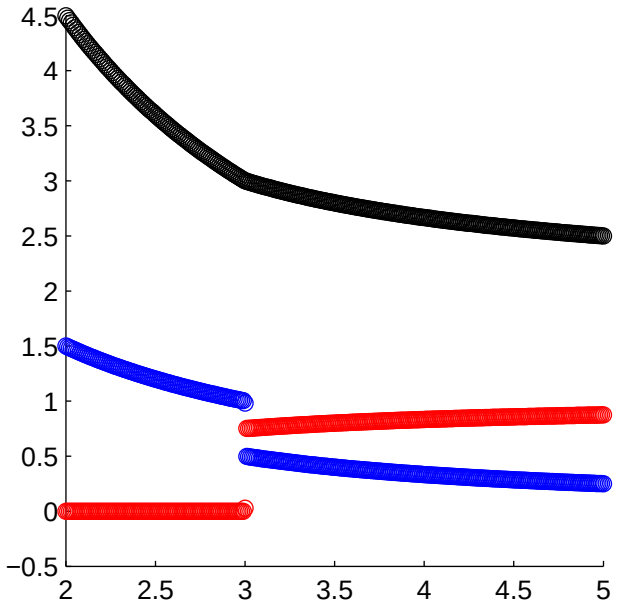
1.3 Příklad 2: citlivost na změnu dat

Je poučné si rovněž vyzkoušet jednoduchou analýzu citlivosti na změny dat lineárních programů. V lineárním programování totiž data A, b, c mívají věcný význam a bývá relevantní otázka, jak by se změnilo optimální řešení a/nebo optimální hodnota účelové funkce, kdyby se některý koeficient změnil. Například v případě tzv. dietního problému koeficienty matice A říkají, jaký je obsah živin v potravinách. A často je na místě se ptát, jak by se změnilo optimum, kdyby dodávka potravin měla (mírně) odlišný obsah živin, než je deklarováno.

V tomto příkladu vyjděme z LP (6) a položme si otázku, jak by se měnilo optimum x_1^*, x_2^* a optimální hodnota účelové funkce, kdybychom namísto pevné hodnoty $A_{21} = 4$ ji považovali za parametr $A_{21} = \alpha$, kde α probíhá interval (například) $[2, 5]$. Ptáme se vlastně na chování parametrického lineárního programu

$$\left\{ \min_{x \in \mathbb{R}^2} (-3,-2) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \;\text{ s.t. } \begin{pmatrix} 1 & 2 \\ \alpha & 2 \\ -1 & -1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \le \begin{pmatrix} 2 \\ 3 \\ -1 \\ 0 \\ 0 \end{pmatrix} \right\}, \qquad \alpha \in [2,5].$$

Pak optima $x_1^*(\alpha), x_2^*(\alpha)$ i optimální hodnota účelové funkce $-c^{\mathrm{T}} x^*(\alpha)$ jsou funkce α a můžeme si nakreslit graf jejich závislosti na α (α je na horizontální ose). Hodnoty $x_1^*(\alpha)$ kreslíme modře, hodnoty $x_2^*(\alpha)$ kreslíme červeně a optimální hodnotu účelové funkce kreslíme černě:



Všimněme si, že z obrázku je patrná změna báze (bod nespojistosti $x_1^*(\alpha)$ a $x_2^*(\alpha)$) pro $\alpha = 3$. Interpretujeme-li hodnoty α jako *možné scénáře* (například: až nám přivezou potraviny, uvidíme, jaký je skutečný obsah živin α, nyní to ovšem nevíme) a interpretujeme-li účelovou funkci jako ziskovou funkci, z obrázku například zjistíme: budeme-li mít štěstí a nastane-li nejlepší možný scénář $\alpha = 2$, budeme mít nejvyšší možný zisk 4.5. Zatímco nastane-li nejhorší možný scénář $\alpha = 5$, musíme počítat pouze se ziskem ≈ 2.7. Vlastně jsme „vyřešili“ dva nelineární optimalizační problémy

$$\min_{\alpha\in[2,5]} \min_{x\in\mathbb{R}^2} (-3,-2)\begin{pmatrix}x_1\\x_2\end{pmatrix} \quad \text{s.t.} \begin{pmatrix}1 & 2\\ \alpha & 2\\ -1 & -1\\ -1 & 0\\ 0 & -1\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix} \le \begin{pmatrix}2\\3\\-1\\0\\0\end{pmatrix},$$

$$\max_{\alpha\in[2,5]} \min_{x\in\mathbb{R}^2} (-3,-2)\begin{pmatrix}x_1\\x_2\end{pmatrix} \quad \text{s.t.} \begin{pmatrix}1 & 2\\ \alpha & 2\\ -1 & -1\\ -1 & 0\\ 0 & -1\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix} \le \begin{pmatrix}2\\3\\-1\\0\\0\end{pmatrix}.$$

Problémy takového typu se v rámci parametrického programování zkoumají často.

Obrázek snadno vygenerujeme skriptem, kde uvnitř `for`-cyklu postupně měníme hodnotu A_{21} v intervalu $[2, 5]$ s rozumně malým krokem (zde volíme krok $1/100$), v každé iteraci voláme solver `linprog` a výsledky vykreslíme do grafu.

```
c = [-3;-2];
A = [1,2; 4,2; -1,-1; -eye(2)];
b = [2;3;-1;zeros(2,1)];
figure; hold on;
for alpha=2:0.01:5
   A(2,1)=alpha;
   x=linprog(c,A,b);
   plot(alpha,x(1),'bo', alpha,x(2),'ro', alpha,-c'*x,'ko');
end
```

1.4 Příklad 3: maticové hry

V teorii maticových her je třeba řešit lineární programy tvaru

$$\max_{\substack{\gamma\in\mathbb{R}\\ \xi\in\mathbb{R}^n}} \gamma \ \text{ s.t. } \ P^{\mathrm{T}}\xi \ge \gamma e,\ e^{\mathrm{T}}\xi = 1,\ \xi \ge 0, \tag{7}$$

kde P je zadaná matice rozměru $(n \times m)$ a $e = (1, \ldots, 1)^{\mathrm{T}}$. (Abychom předešli nejasnostem: γ je skalár — proměnná, která může nabývat libovolných reálných hodnot (i záporných) — a γe je vektor $(\gamma, \ldots, \gamma)^{\mathrm{T}}$.) Bližší interpretací tohoto lineárního programu se budeme zabývat v kapitole ??; zde jen krátce shrňme, že získáme-li optimum (γ^*, ξ^*), pak ξ^* je nashovská strategie pro prvního hráče ve smíšeném rozšíření maticové hry s výplatní maticí P („payoff" — odtud symbol P) a γ^* je cena této hry. (*Poznámka.* Vyřešíme-li duál k (7), získáme nashovskou strategii pro druhého hráče; to zde ovšem nebudeme činit, konstrukci duálu a jeho řešení ponecháváme jako cvičení.)

S pomocí triků (a) – (c) přepišme LP (7) do ekvivalentního tvaru

$$\min_{\gamma,\xi} -\gamma \ \text{ s.t. } \ \gamma e - P^{\mathrm{T}}\xi \leq 0,\ e^{\mathrm{T}}\xi \leq 1,\ -e^{\mathrm{T}}\xi \leq -1,\ -I\xi \leq 0 \tag{8}$$

a uspořádejme proměnné γ, ξ do společného vektoru x například v pořadí $x = \binom{\gamma}{\xi}$. Získáme maticový tvar

$$\min_{x=\binom{\gamma}{\xi}} \underbrace{(-1, 0_{1\times n}^{\mathrm{T}})}_{c^{\mathrm{T}}} \underbrace{\begin{pmatrix}\gamma\\ \xi\end{pmatrix}}_{x} \ \text{ s.t. } \ \underbrace{\begin{pmatrix} e_{m\times 1} & -P_{m\times n}^{\mathrm{T}} \\ 0_{1\times 1} & e_{1\times n}^{\mathrm{T}} \\ 0_{1\times 1} & -e_{1\times n}^{\mathrm{T}} \\ 0_{n\times 1} & -I_{n\times n}\end{pmatrix}}_{A} \underbrace{\begin{pmatrix}\gamma\\ \xi\end{pmatrix}}_{x} \leq \underbrace{\begin{pmatrix} 0_{m\times 1} \\ 1_{1\times 1} \\ -1_{1\times 1} \\ 0_{n\times 1}\end{pmatrix}}_{b};$$

pro přehlednost jsme explicitně uvedli rozměry vektorů a matic. V tomto tvaru je možné volat solver `linprog`.

Uvažme pro příklad hru Kámen-nůžky-papír s výplatní maticí

$$P = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix}.$$

Skript napíšeme tak, aby počínaje druhým řádkem fungoval pro libovolnou výplatní matici P (tedy: na prvním řádku může uživatel zadat libovolnou matici libovolných rozměrů).

```
P = [0,1,-1; -1,0,1; 1,-1,0] % uživatel může zadat libovolnou matici
[n,m] = size(P);
c = [-1; zeros(n,1)];
A = [ones(m,1),  -P';        ...
     0,           ones(1,n); ...
     0,          -ones(1,n); ...
     zeros(n,1), -eye(n)    ];
b = [zeros(m,1); 1; -1; zeros(n,1)];
x = linprog(c,A,b);
gamma = x(1);
xi = x(2:n+1);
disp(['Cena hry  = ', num2str(gamma)]);
disp(['Strategie = ', num2str(xi')]);
```

Výstupem je:

```
Cena hry  = 1.4211e-014
Strategie = 0.33333   0.33333   0.33333
```

Číslo řádu 10^{-14} je numerická nula; skutečně, Kámen-nůžky-papír je symetrická hra, jež má nutně nulovou cenu. Nashovská strategie pro prvního hráče (a ze symetrie hry také pro druhého) je $\xi^* = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})^{\mathrm{T}}$; všechny tři strategie — Kámen, nůžky, papír — se mají hrát se stejnou pravděpodobností.

Poznámka. Lze postupovat i takto: namísto tvaru (8) zvlášť napíšeme nerovnosti a zvlášť rovnosti (viz (3)), čímž získáme

$$\min_{\gamma,\xi} -\gamma \ \text{ s.t. } \ \underbrace{\gamma e - P^{\mathrm{T}}\xi \leq 0,\ -I\xi \leq 0}_{\text{nerovnosti}},\ \underbrace{e^{\mathrm{T}}\xi = 1}_{\text{rovnosti}},$$

anebo totéž v maticovém tvaru

$$\min_{x=\binom{\gamma}{\xi}} \underbrace{(-1, 0_{1\times n}^{\mathrm{T}})}_{c^{\mathrm{T}}} \underbrace{\begin{pmatrix}\gamma\\ \xi\end{pmatrix}}_{x} \ \text{ s.t. } \ \underbrace{\begin{pmatrix} e_{m\times 1} & -P_{m\times n}^{\mathrm{T}} \\ 0_{n\times 1} & -I_{n\times n}\end{pmatrix}}_{A} \underbrace{\begin{pmatrix}\gamma\\ \xi\end{pmatrix}}_{x} \leq \underbrace{\begin{pmatrix} 0_{m\times 1} \\ 0_{n\times 1}\end{pmatrix}}_{b}, \quad \underbrace{(0, e_{1\times n}^{\mathrm{T}})}_{U} \underbrace{\begin{pmatrix}\gamma\\ \xi\end{pmatrix}}_{x} = \underbrace{(1)}_{v}.$$

Voláme `linprog` ve tvaru (4):

```
P = [0,1,-1; -1,0,1; 1,-1,0]; % stále hrajeme Kámen-nůžky-papír
[n,m] = size(P);
c = [-1; zeros(n,1)];
A = [ones(m,1), -P'; zeros(n,1), -eye(n)];
b = zeros(m+n,1);
U = [0, ones(1,n)];
v = 1;
x = linprog(c,A,b,U,v);
gamma = x(1);
xi = x(2:n+1);
disp(['Cena hry  = ', num2str(gamma)]);
disp(['Strategie = ', num2str(xi')]);
```

Dodatek 1. I s maticovou hrou si lze „hrát“ podobně jako v kapitole 1.3 a obvykle takové hraní pomáhá při porozumění chování studovaného problému. Zde si položme otázku, jak by se změnily nashovské strategie ve hře Kámen-nůžky-papír, kdyby různé situace byly nestejně oceněné. Uvažme například výplatní matici

$$P(\alpha) = \begin{pmatrix} 0 & \alpha & -1 \\ -\alpha & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix} \tag{9}$$

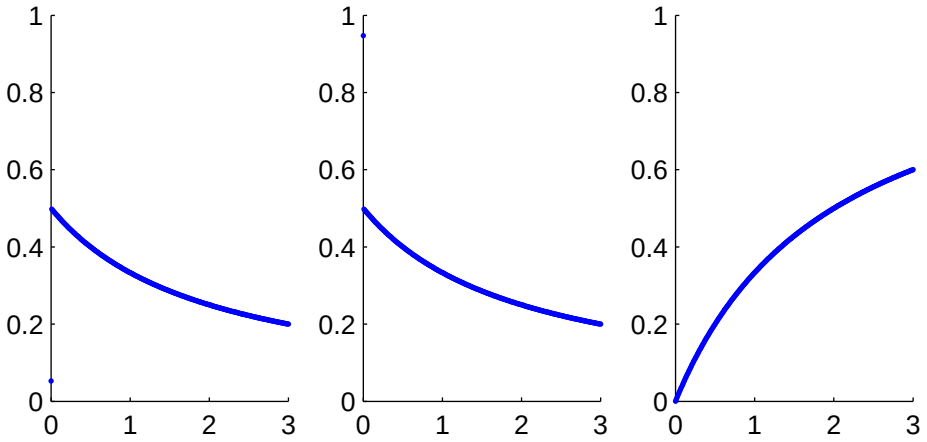
závisející na parametru $\alpha > 0$; řekněme pro příklad, že α probíhá interval $[0, 3]$. Jedná se o modifikaci hry Kámen-nůžky-papír pro situaci, kdy dvojice strategií kámen-nůžky má výplatu α, obecně odlišnou od 1. Jedná se stále o symetrickou hru, takže cena hry je nulová; nicméně nashovské strategie $\xi_1^*(\alpha), \xi_2^*(\alpha), \xi_3^*(\alpha)$ jistě na parametru α závisejí. Jak? Bylo by možné se pokusit odvodit tuto závislost analyticky; my si zde jen numericky nakreslíme obrázek. Nejprve si skript z předchozího textu napišme jako funkci, která dostane na vstup výplatní matici P a vrátí cenu hry γ a vektor nashovských strategií ξ; bude to náš „univerzální game-solver“.

```
function [gamma, xi] = GameSolver(P)
 [n,m] = size(P);
 c = [-1; zeros(n,1)];
 A = [ones(m,1), -P'; zeros(n,1), -eye(n)];
 b = zeros(m+n,1);
 U = [0, ones(1,n)];
 v = 1;
 x = linprog(c,A,b,U,v);
 gamma = x(1);
 xi = x(2:n+1);
end
```

Nyní stačí volat `GameSolver` uvnitř `for`-cyklu probíhajícího $\alpha \in [0, 3]$ s dostatečně malým krokem (zde zvolíme například $1/100$) a vykreslit výstup.

```
P = [0,1,-1; -1,0,1; 1,-1,0]; % výplatní matice
figure;
for alpha = 0:0.01:3
  P(1,2)=alpha;               % změníme hodnoty P_12 a P_21 podle (9)
  P(2,1)=-alpha;
  [gamma, xi] = GameSolver(P);
  for i=1:3                   % vizualizace strategie i = 1 (=K), 2 (=N), 3 (=P)
    subplot(1,3,i); hold on;
    plot(alpha,xi(i),'.');     % vykresli i-tou strategii proti α
    axis([0,3,0,1]);          % nastavení os
  end
end
```

Výsledkem je následující obrázek:

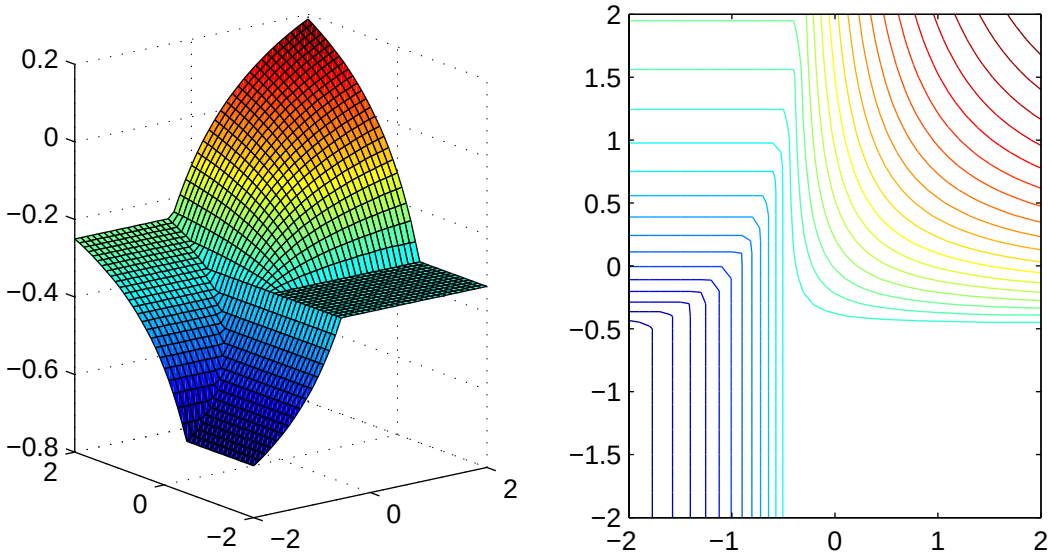


Na vodorovné ose je ve všech třech případech $\alpha \in [0,3]$. Na prvním obrázku jsou hodnoty $\xi_1^*(\alpha)$ — tedy pravděpodobnosti, s nimiž se má volit strategie Kámen, v závislosti na α. (A analogicky druhý a třetí obrázek zobrazuje Nůžky $\xi_2^*(\alpha)$ a Papír $\xi_3^*(\alpha)$). Je vidět, že při $\alpha = 0$ se strategie Papír nemá hrát vůbec; její váha (pravděpodobnost) roste s hodnotou α, zatímco váha strategií Kámen a Nůžky klesá, a to stejným tempem. A nyní, po přehlédnutí techto obrázků, je vhodné se pokusit tuto závislost zdůvodnit analyticky: jednak kvalitativně vysvětlit trend, a jednak se pokusit najít explicitní formuli pro funkce $\xi_i^*(\alpha)$, $i = 1, 2, 3$.

Dodatek 2. Úvahy z předchozího textu můžeme samozřejmě zkombinovat i s dalšími nástroji MATLABu: například s 3D vizualizací. Uvažme modifikaci hry Kámen-nůžky papír s dvěma parametry

$$P(\alpha,\beta) = \begin{pmatrix} 0 & \alpha & -1 \\ -1 & 0 & 1 \\ \beta & -1 & 0 \end{pmatrix}, \quad \alpha \in [-2,2],\ \beta \in [-2,2]. \tag{10}$$

Toto již není symetrická hra; má proto smysl se ptát, jak závisí cena hry γ^* na hodnotách α, β. V levém obrázku vykreslíme cenu hry na svislé ose v závislosti na α, β (horizontální osy) jako trojrozměný graf pomocí `surf`; v pravém obrázku vykreslíme tutéž funkci v prostoru (α, β) pomocí 30 vrstevnic funkcí `contour`.



Připravíme si `meshgrid`: pokryjeme čtverec $[-2,2] \times [-2,2]$ v (α, β)-prostoru sítí bodů s krokem (například) 0.1; pak v každém bodě sítě spočteme pomocí již vytvořeného `GameSolver`u cenu hry γ. Tu si uložíme v tabulce g. Nakonec vykreslíme hodnoty v g proti bodům sítě.

```
P = [0,1,-1; -1,0,1; 1,-1,0];  % výplatní matice Kámen-nůžky-papír
[alpha,beta] = meshgrid(-2:0.1:2, -2:0.1:2);
[k,l] = size(alpha);           % rozměr meshgridu
for i = 1:k
  for j = 1:l                  % i,j probíhají všechny body meshgridu
    P(1,2) = alpha(i,j);       % ve výplatní matici nastavíme hodnoty parametrů α,β
    P(3,1) = beta(i,j);
    [gamma,xi] = GameSolver(P); % gamma je cena hry
    g(i,j) = gamma;
  end
end
figure;                        % vizualizace
subplot(1,2,1);
surf(alpha,beta,g);
subplot(1,2,2);
contour(alpha,beta,g,30);
```

Poznámka. Často se stává, že výplatní matice P není zadána explicitně; namísto toho je dána sada pravidel, která umožňují prvek P_{ij} dopočítat na základě (i,j). Tak se činí především tehdy, je-li tento popis kratší a transparentnější než explicitní výčet všech prvků (často hodně velké) výplatní matice. Pak je třeba napsat skript, který na základě pravidel matici P sestaví. Dva takové příklady si ukážeme v kapitole ?? — předešleme, že půjde o hry známé jako *n-prstá Morra* (jde o starobylou hazardní hru) a tzv. *(m,n)-hra plukovníka Blotto*, v níž se modeluje jistý konflikt dvou armád o m a n plucích.

1.5 Příklad 4: toky v sítích

Buď dána síť s množinou vrcholů $V = \{1,\dots,n\}$ a množinou (orientovaných) hran E s nezáporným ohodnocením (tzv. *kapacitou*) k_e ($e \in E$). Připomeňme, že v síti je rozlišena dvojice speciálních vrcholů, tzv. *zdroj* (vrchol s nulovým vstupním stupněm) a *cíl* (vrchol s nulovým výstupním stupněm). Řekněme, že zdrojem je vrchol 1 a cílem je vrchol n. Cílem je najít maximální tok mezi zdrojem a cílem: jedná se o ohodnocení hran x_e ($e \in E$) takové, že

(i) pro každou hranu e jest $0 \le x_e \le k_e$ („kapacity nelze překročit“, tzv. *kapacitní podmínky*);

(ii) pro každý vrchol v různý od zdroje a cíle platí $\sum_{e\in\text{in}(v)} x_e = \sum_{e\in\text{out}(v)} x_e$ („součet přítoků je stejný jako součet odtoků — ve vrcholu v se nic neztrácí“, tzv. *tokové podmínky*);

(iii) veličina $\sum_{e\in\text{in}(n)} x_e$ je maximální („celkový přítok do cíle je maximální možný“).

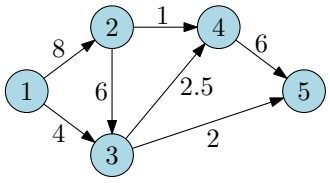
Symbolem in(v) jsme označili množinu hran vstupujících do vrcholu v a symbolem out(v) jsme označili množinu hran vystupujících z vrcholu v.

Podmínky (i) – (iii) jsou lineární v x_e, a tak je snadné tento problém zformulovat jako lineární program. Za každou hranu $e \in E$ zařadíme proměnnou x_e a můžeme psát

$$\max_{x_e, e\in E} \sum_{e\in\text{in}(n)} x_e \text{ s.t. } \underbrace{0 \le x_e \le k_e \ (\forall e \in E)}_{(\star)}, \underbrace{\sum_{e\in\text{in}(v)} x_e = \sum_{e\in\text{out}(v)} x_e \ (\forall v \in \{2,\dots,n-1\})}_{(\dagger)}. \tag{11}$$

Podmínky $(\star)$ jsou kapacitní omezení (i) a podmínky $(\dagger)$ jsou tokové podmínky (ii).

Uvažme příklad sítě na obrázku (vrchol 1 je zdroj, vrchol $n = 5$ je cíl):



(12)

Proměnné x_e ($e \in E$) uspořádejme do vektoru x například v pořadí $(x_{12}, x_{13}, x_{23}, x_{24}, x_{45}, x_{34}, x_{35})^{\mathrm{T}}$; zde x_{ij} značí proměnnou odpovídající hraně (i,j). Analogicky zaveďme vektor kapacit $k = (8,4,6,1,6,2.5,2)^{\mathrm{T}}$. Lineární program (11) pak

získá tvar vhodný pro volání solveru `linprog` v syntaxi (4):

$$\max \overbrace{(0\,0\,0\,0\,1\,0\,1)}^{c^{\mathrm{T}}} \begin{pmatrix} x_{12} \\ x_{13} \\ x_{23} \\ x_{24} \\ x_{45} \\ x_{34} \\ x_{35} \end{pmatrix} \quad \text{s.t.} \quad \underbrace{\overbrace{\begin{pmatrix} I_{7\times 7} \\ -I_{7\times 7} \end{pmatrix}}^{A} \begin{pmatrix} x_{12} \\ x_{13} \\ x_{23} \\ x_{24} \\ x_{45} \\ x_{34} \\ x_{35} \end{pmatrix} \leq \overbrace{\begin{pmatrix} k_{7\times 1} \\ 0_{7\times 1} \end{pmatrix}}^{b}}_{(\star)}, \quad \underbrace{\overbrace{\begin{pmatrix} -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & -1 & 1 & -1 & 0 \end{pmatrix}}^{U} \begin{pmatrix} x_{12} \\ x_{13} \\ x_{23} \\ x_{24} \\ x_{45} \\ x_{34} \\ x_{35} \end{pmatrix} = \overbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}}^{v}}_{(\dagger)}. \tag{13}$$

Opět, podmínky $(\star)$ jsou kapacitní omezení (i) a podmínky $(\dagger)$ jsou tokové podmínky (ii).

Matice a vektory A, b, v sestavíme snadno. Nechť $m = |E|$ značí počet hran. Matice U vypadá velmi podobně jako incidenční matice grafu G. Připomeňme, že *incidenční matice* Z je matice rozměru $n \times m$, řádky jsou indexovány vrcholy $v = 1, \ldots, n$ a sloupce jsou indexovány hranami $e \in E$ a platí

$$Z_{ve} = \begin{cases} 1, & \text{vystupuje-li hrana } e \text{ z vrcholu } v, \\ -1, & \text{vstupuje-li hrana } e \text{ do vrcholu } v, \\ 0 & \text{jinak.} \end{cases} \tag{14}$$

V našem příkladu jest

$$Z = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & -1 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 \end{pmatrix}.$$

Matice U vznikne z matice Z smazáním prvního a posledního řádku; skutečně, tokové podmínky (ii) platí pro všechny vrcholy *kromě* zdroje (vrchol 1) a cíle (poslední vrchol v pořadí). A není náhodou, že vektor c^{T} je (až na znaménko) roven poslednímu řádku matice Z; skutečně, poslední řádek matice Z je charakteristický vektor hran vstupujících do cíle. Sestrojíme-li tedy incidenční matici Z, snadno z ní získáme U a c.

Je třeba zvolit vhodnou reprezentaci dat: jedna z přirozených reprezentací je uspořádat hrany do matice rozměru $m \times 2$, kde řádky odpovídají hranám a v řádku je uvedeno číslo počátečního a koncového vrcholu. Ve stejném pořadí zapišme kapacity ve vektoru k:

```
n = 5;          % počet vrcholů
e = [1,2; 1,3; 2,3; 2,4; 4,5; 3,4; 3,5];  % výčet hran
k = [8;   4;   6;   1;   6;   2.5; 2  ];  % kapacity
m = length(k); % délka vektoru kapacit = počet hran
```

Incidenční matici Z vytvoříme ve dvou krocích: nejprve si připravíme nulovou matici $0_{n\times m}$ a poté `for`-cyklem přes hrany projdeme postupně sloupce Z a vyplníme do nich ± 1 podle předpisu (14):

```
Z = zeros(n,m);
for i = 1:m
  Z(e(i,1),i) =  1;
  Z(e(i,2),i) = -1;
end
```

Nyní již můžeme snadno definovat A, b, c, U, v dle (13) a volat solver `linprog` v syntaxi (4):

```
A = [eye(m); -eye(m)];  % viz (13)
b = [k; zeros(m,1)];    % viz (13)
c = -(Z(n,:))';         % poslední řádek incidenční matice
U = Z(2:n-1,:);         % incidenční matice bez prvního a posledního řádku
v = zeros(n-2,1);       % viz (13)
x = linprog(-c,A,b,U,v) % píšeme −c, úloha je totiž maximalizační
```

Optimální x^* je nalezený maximální tok. (Upozorňujeme, že obecně nemusí být jednoznačný.)

Dodatek: jak si nakreslit obrázek. MATLAB disponuje knihovnou pro práci s grafy a jejich pokročilou vizualizaci. Její popis přesahuje rámec tohoto textu (nicméně je užitečné se podívat např. na `doc graph` pro základní informace o neorientovaných grafech a `doc digraph` pro základní informace o orientovaných grafech). Naším cílem je ukázat jednoduchý skript bez použití

grafových nástrojů, kterým si lze síť a nalezený maximální tok nakreslit jen s pomocí vizualizace hran šipkami. Šipku bychom si mohli nakreslit sami pomocí funkce `plot` (šipka je koneckonců jen trojice úseček); nicméně elegantnější bude použít funkci

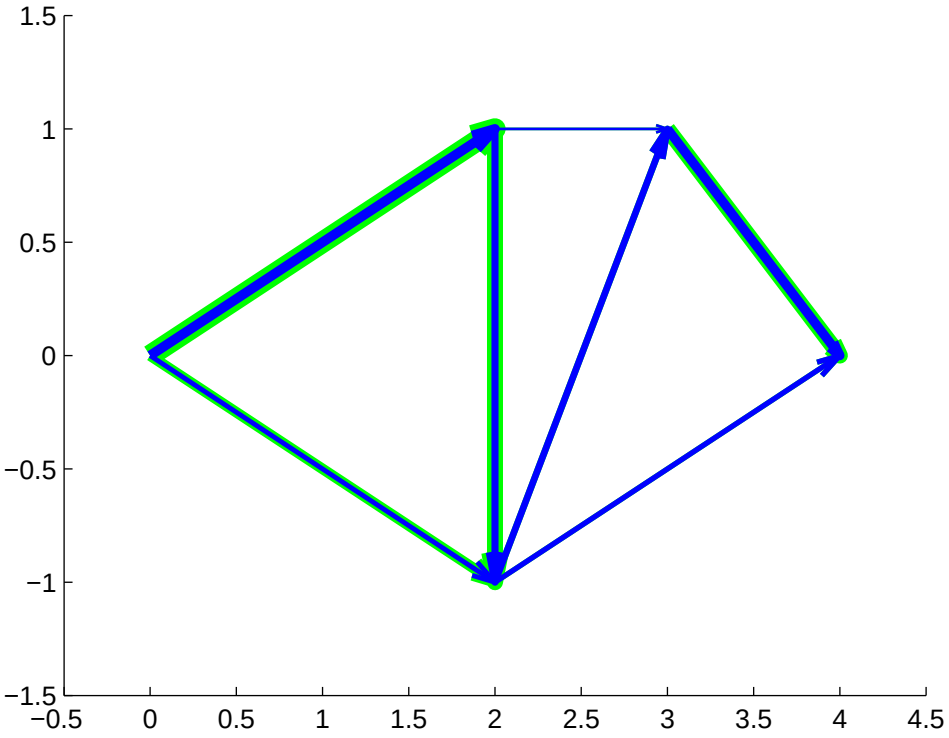
$$\texttt{quiver}(x, y, u, v, 0), \tag{15}$$

která nakreslí šipku z bodu (x, y) do bodu $(x + u, y + v)$. Pátý argument bude u nás vždy nula; funkce `quiver` totiž ještě pracuje s „natahováním“ šipek (což se v některých aplikacích hodí), ale zde tuto funkci vypínáme. Funkce `quiver` používá stejné formátovací konvence jako `plot`, pročež můžeme analogicky nastavit barvu či sílu šipky. Například

```
quiver(1,1,2,3,0,'r','LineWidth',5)
```

nakreslí silnou červenou šipku z bodu $(1, 1)$ do bodu $(3, 4)$; síla šipky je 5 bodů.

Naším cílem bude nakreslit následující obrázek:



Je zde nakreslená síť (12). Zelená barva odpovídá kapacitám hran; síla zelené šipky je úměrná kapacitě. Modrými šipkami kreslíme nalezený maximální tok. Síla modré šipky odpovídá velikosti optimálního toku hranou. Například hrana $(1, 2)$ má kapacitu 8, což kreslíme zelenou šipkou síly 8 bodů; optmiální tok touto hranou je jen 5, a tak silnou zelenou šipku částečně (ale ne zcela) překryjeme modrou šipkou o síle 5 bodů. Odtud je patrné, že kapacita této hrany není vyčerpána (zelená šipka je silnější než modrá). Jiným příkladem je hrana $(2, 4)$; ta má kapacitu 1. Zelená šipka síly 1 ovšem není viditelná, neboť je plně překryta modrou šipkou síly 1. To znamená, že kapacita této hrany je při optimálním toku zcela využita. Totéž platí pro hrany $(3, 4)$ a $(3, 5)$.

Je třeba říci, na jakých souřadnicích (ξ_i, y_i) v rovině mají být umístěny vrcholy $i = 1, \ldots, n\,(= 5)$. (Horizontální souřadnici raději značíme ξ, neboť symbol x už jsme použili pro optimální tok.) Tyto souřadnice pak použijeme jako počáteční a koncové body šipek. Aby byl náš obrázek opticky podobný obrázku (12), zvolme například

$$(\xi_1, y_1) = (0, 0),\ \ (\xi_2, y_2) = (2, 1),\ \ (\xi_3, y_3) = (2, -1),\ \ (\xi_4, y_4) = (3, 1),\ \ (\xi_5, y_5) = (4, 0).$$

Do skriptu pišme

```
xi = [0; 2;  2; 3; 4];  % vektor (ξ_1,...,ξ_n)^T
y  = [0; 1; -1; 1; 0];  % vektor (y_1,...,y_n)^T
```

Nyní se hodí vytvořit pomocnou funkci `PlotNet`, která pomocí `for`-cyklu postupně vykreslí hrany jako šipky, a to v síle dané vektorem `weights` a barvou `color`. Tuto funkci pak zavoláme dvakrát; nejprve pro vykreslení zelených šipek, kde `weights`

jsou kapacity, a podruhé pro vykreslení modrých šipek, kde `weights` jsou velikosti optimálního toku hranami.

```
function PlotNet(weights,color)
 for i=1:m         % i probíhá množinu hran
  quiver(xi(e(i,1)), y(e(i,1)), xi(e(i,2))-xi(e(i,1)), y(e(i,2))-y(e(i,1)), ...
         0,color,'Linewidth',weights(i));  % nakreslí šipku
 end
end
```

Všimněme si, že `xi(e(i,1))` a `y(e(i,1))` jsou rovinné souřadnice počátečního vrcholu i-té hrany a `xi(e(i,2))` a `y(e(i,2))` jsou rovinné souřadnice koncového vrcholu i-té hrany. Kreslíme tedy šipku z bodu `xi(e(i,1))` a `y(e(i,1))` a v (15) klademe $u =$ `xi(e(i,2))` $-$ `xi(e(i,1))` a $v =$ `y(e(i,2))` $-$ `y(e(i,1))`.

Nyní stačí volat `PlotNet(k,'g')` pro vykreslení zelených šipek a `PlotNet(x,'b')` pro vykreslení modrých šipek (zde `x` je nalezený optimální tok pomocí solveru `linprog`). Celý skript si napišme jako samostatnou funkci; říkejme jí například `MyNetFlow`.

```
function MyNetFlow

   function PlotNet(weights,color)  % vnořená funkce - vykreslí síť
    % color je barva, např. 'r'; weights je vektor pro tloušťky šipek
    for i=1:m
     quiver(xi(e(i,1)), y(e(i,1)), xi(e(i,2))-xi(e(i,1)), y(e(i,2))-y(e(i,1)), ...
            0,color,'LineWidth',weights(i));
    end
   end

  % data: počet vrcholů n, seznam hran e, kapacity k, počet hran m
   n = 5;
   e = [1,2; 1,3; 2,3; 2,4; 4,5; 3,4; 3,5];
   k = [8;   4;   6;   1;   6;   2.5; 2  ];
   m = length(k);

  % data: rovinné souřadnice vrcholů
   xi= [0; 2;  2; 3; 4];
   y = [0; 1; -1; 1; 0];

  % konstrukce incidenční matice
   Z = zeros(n,m);
   for i = 1:m
    Z(e(i,1),i)=1;
    Z(e(i,2),i)=-1;
   end

  % volání linprog
   c = -(Z(n,:))';
   A = [eye(m); -eye(m)];
   b = [k; zeros(m,1)];
   U = Z(2:n-1,:);
   v = zeros(n-2,1);
   x = linprog(-c,A,b,U,v);

  % vizualizace: kapacity k zeleně ('g'), optimální tok x modře ('b')
   figure; hold on;
   PlotNet(k,'g');
   PlotNet(x,'b');
end
```

Poznámka. S modelem si lze „hrát“ obvyklým způsobem. Kdybychom například připustili, že tok hranou může být oběma směry (a model má vybrat ten směr, který povede k maximalizaci přítoku do cíle), stačí v (11) nahradit podmínku $0 \le x_e \le k_e$ podmínkou $-k_e \le x_e \le k_e$. Pak v (13) bude $b = \binom{k}{-k}$, a tedy stačí ve skriptu psát `b = [k; -k]`. Vše ostatní zůstává beze

změny (pochopitelně by bylo třeba upravit vizualizační skript, aby kreslil modré šipky správným směrem podle znamének ve vektoru optimálního toku x^*; to ponecháváme jako cvičení).

Analogicky lze uvážit tzv. multikomoditní toky, toky s dvojitě oceněnými hranami — kapacitou a přepravními náklady —, toky s více zdroji a/nebo více cíli, toky s úbytky ve vnitřních vrcholech atd. Je poučné coby cvičení tyto lineární programy zformulovat a napsat je jako skripty.

1.6 Příklad 5: Metoda CPM

Metoda CPM (metoda kritické cesty, *Critical Path Method*) je jednoduchý postup v řízení projektů. Projekt sestává z dílčích úkolů, indexovaných $1, \ldots, n$, a pro úkol i je zadána doba trvání d_i. Dále je zadána *precedence*: pro některé dvojice úkolů (i, j) je řečeno, že úkol j nemůže začít před dokončením úkolu i; jinak mohou úkoly běžet paralelně. Je přirozené tuto situaci reprezentovat pomocí orientovaného grafu $G = (V, E)$: vrcholy $V = \{1, \ldots, n\}$ nechť představují úkoly a nechť dvojice (i, j) tvoří hranu ($(i, j) \in E$), právě když úkol j nemůže začít před dokončením úkolu i. Cílem je zjistit dobu trvání celého projektu. Předpokládejme, že graf G je acyklický (jinak by projekt nešlo dokončit nikdy); nutně tudíž má vrchol nulového vstupního stupně a vrchol nulového výstupního stupně. Budeme bez újmy na obecnosti dále předpokládat, že vrchol (úkol) s nulovým výstupním stupněm je jediný, že je to vrchol n a že platí $d_n = 0$; snadno se nahlédne, že toho lze dosáhnout vždy. Úkol s pořadovým číslem n pak vlastně představuje pouhý formální úkol, totiž ukončení projektu, který trvá nulovou dobu.

Nechť x_i představuje časový okamžik, kdy nejdříve může úkol x_i začít. Dobu trvání projektu lze zjistit pomocí lineárního programu

$$\min_{x_1,\ldots,x_n} x_n \;\text{ s.t. }\; \underbrace{x_j \geq x_i + d_i \;(\forall (i,j) \in E)}_{(\star)},\;\; x_i \geq 0 \;(\forall i \in V). \tag{16}$$

(Jde to spočítat mnohem jednodušeji než pomocí LP — po krátké úvaze se snadno přijde na přímočarý algoritmus, ovšem nám zde jde o cvičení na LP v MATLABu, a je to takto navíc zřejmě nejjednodušší k programování.) Podmínky $(\star)$ říkají: je-li (i, j) hrana, pak začátek x_j úkolu j může nastat nejdříve poté, co je dokončen úkol i (to je okamžik $x_i + d_i$). Minimalizujeme x_n — to je okamžik, kdy projekt vyvrcholí posledním úkolem, a tedy také doba trvání celého projektu (připomeňme konvenci $d_n = 0$).

Přepišme (16) do tvaru

$$\min_{x_1,\ldots,x_n} x_n \;\text{ s.t. }\; x_i - x_j \leq -d_i \;(\forall (i,j) \in E),\;\; -x_i \leq 0 \;(\forall i \in V);$$

odtud je již přímočarý krok k tvaru

$$\min_{x=(x_1,\ldots,x_n)^{\mathrm{T}}} (0_{1\times(n-1)}\; 1)^{\mathrm{T}} x \;\text{ s.t. }\; Z^{\mathrm{T}} x \leq -\delta,\; -I_{n\times n} x \leq 0_{n\times 1},$$

kde Z je incidenční matice grafu G daná předpisem (14), m označuje počet hran a δ je vektor dob trvání úkolů definovaný předpisem

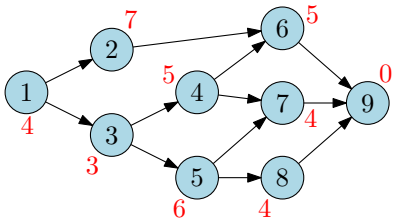
$$\delta_e = d_i, \;\; \text{jestliže hrana } e \text{ začíná ve vrcholu } i \;\; (e \in E). \tag{17}$$

Složky vektoru δ jsou indexovány hranami ve stejném pořadí jako sloupce incidenční matice Z. Pišme konečně

$$\min_x \underbrace{(0_{1\times(n-1)} 1)^{\mathrm{T}}}_{c^{\mathrm{T}}} x \;\text{ s.t. }\; \underbrace{\begin{pmatrix} Z^{\mathrm{T}} \\ -I_{n\times n} \end{pmatrix}}_{A} x \leq \underbrace{\begin{pmatrix} -\delta \\ 0_{n\times 1} \end{pmatrix}}_{b}; \tag{18}$$

toto je již vhodný tvar pro solver `linprog`.

Pro účely skriptu zvolíme stejnou reprezentaci grafu G jako v kapitole 1.5, totiž výčet hran v matici rozměru $m \times 2$, kde v k-tém řádku je zapsán počáteční a koncový vrchol k-té hrany. Pro příklad uvažme graf (projekt) z následujícího obrázku; černě jsou uvedeny indexy vrcholů (úkolů) i a červeně jejich doby trvání d_i.



(19)

Nejprve reprezetujme data stejně jako v kapitole 1.5:

```
d = [4; 7; 3; 5; 6; 5; 4; 4; 0]; % doby trvání
e = [1,2; 1,3; 2,6; 3,4; 3,5; 4,6; 4,7; 5,7; 5,8; 6,9; 7,9; 8,9]; % výčet hran
n = length(d);          % počet vrcholů = počet dob trvání
m = length(e);          % počet hran = počet řádků matice e
```

Sestrojme incidenční matici Z (opět, stejně jako v kapitole 1.5):

```
Z = zeros(n,m);
for i = 1:m
  Z(e(i,1),i) =  1;
  Z(e(i,2),i) = -1;
end
```

Nyní jsme připraveni vyřešit LP (18):

```
c = [zeros(n-1,1);1];
A = [Z'; -eye(n)];
delta = d(e(:,1));
b = [-delta; zeros(n)];
x = linprog(c,A,b);
disp(x(n));  % x_n je poslední úkol, a tedy doba trvání celého projektu
```

Skript ohlásí, že doba trvání projektu je 17; prostým pohledem na graf G se snadno ověří, že je tomu skutečně tak.

Je vhodné si podrobně rozmyslet, proč instrukce `delta = d(e(:,1))` skutečně vytvoří vektor δ splňující (17) — připomeňme, že `e(:,1)` je vektor indexů počátečních vrcholů jednotlivých hran.

Simulace je lepší než PERT. Řekněme, že doby trvání úkolů d_i nejsou přesně známé; modelujme je jako náhodné veličiny, jejichž rozdělení (jak se zde předpokládá) známe. Pak i doba trvání projektu je náhodná veličina; pochopitelně nás zajímá její rozdělení a jeho charakteristiky (střední hodnota, medián, rozptyl, případně vyšší momenty, kvantily atd.). V padesátých létech byla vyvniuta tzv. metoda PERT (*Program Evaluation and Review Technique*), která se snaží rozdělení doby trvání projektu aproximovat pomocí normálního rozdělení, ovšem za velmi restriktivních předpokladů a s vážným rizikem velké chyby aproximace. (Snadno se nahlédne, že obecně lze těžko očekávat, že by doba trvání projektu měla mít například symetrické rozdělení; už z tohoto základního náhledu je patrné, že aproximace normálním — a tedy symetrickým — rozdělením může být silně zavádějící, může třeba podhodnocovat riziko výrazného prodloužení projektu.) Metoda PERT je dosti hrubá heuristika, jež v podstatě nemá význam v okamžiku, kdy je snadné rozdělení doby trvání projektu nasimulovat. Simulaci si nyní ukažme.

Vytvořme si pomocnou funkci `time = SolveCPM(d,e)`, kde shrneme, co jsme dosud vytvořili — funkce dostane na vstup vektor d dob trvání úkolů a seznam e hran grafu G a vrátím nám `time`, dobu trvání projektu. Tato funkce poslouží coby CPM-solver.

```
function time = SolveCPM(d, e)
  % time = doba trvání projektu zadaného hranami e a dobami úkolů d
  n = length(d);   % počet vrcholů (úkolů)
  m = length(e);   % počet hran
  Z = zeros(n,m);  % příprava incidenční matice Z
  for i = 1:m
    Z(e(i,1),i) = 1;
    Z(e(i,2),i) =-1;
  end
  c = [zeros(n-1,1);1];  % příprava na volání linprog
  A = [Z'; -eye(n)];
  b = [-d(e(:,1)); zeros(n,1)];
  x = linprog(c,A,b);
  time = x(n);     % x_n je poslední úkol, a tedy doba trvání celého projektu
end
```

Nyní budeme simulovat doby trvání úkolů z jejich distribucí a opakovaně volat `SolveCPM`. Simulaci zopakujeme (řekněme) 10^5-krát; doby trvání zaznamenáme do pomocného vektoru h, z nějž vykreslíme histogram (to je simulací získaný odhad skutečné distribuce doby trvání projektu) a pro ilustraci napočteme některé charakteristiky. Protože simulace chvíli trvá, je vhodné skript animovat — postupně vykreslovat empirickou distribuci (histogram) a nechat uživatele sledovat, jak proces konverguje.

V našem příkladu řekněme, že doby trvání jsou nezávislé, rovnoměrně rozdělené náhodné veličiny na intervalu $[d_i-2, d_i+3]$, kde $i = 1,\ldots,8$ jsou úkoly z (19) a d_i jsou v (19) červeně uvedné hodnoty. (Připomínáme, že podle přijaté konvence vždy jest $d_9 = 0$.)

Rovnoměrné rozdělení jsme zvolili jen pro příklad; při simulaci lze použít kterékoliv rozdělení, třeba β-rozdělení (jak je zvykem v PERTu) či třeba empirické rozdělení dob trvání úkolů získané z historických dat. Stejně tak lze užívat i závislé náhodné veličiny; to by se hodilo např. tehdy, provádějí-li různé úkoly titíž pracovníci, anebo více úkolů je ovlivněno společným faktorem, třeba nepříznivým počasím při stavebních pracích.

```
function MyProjectSimul

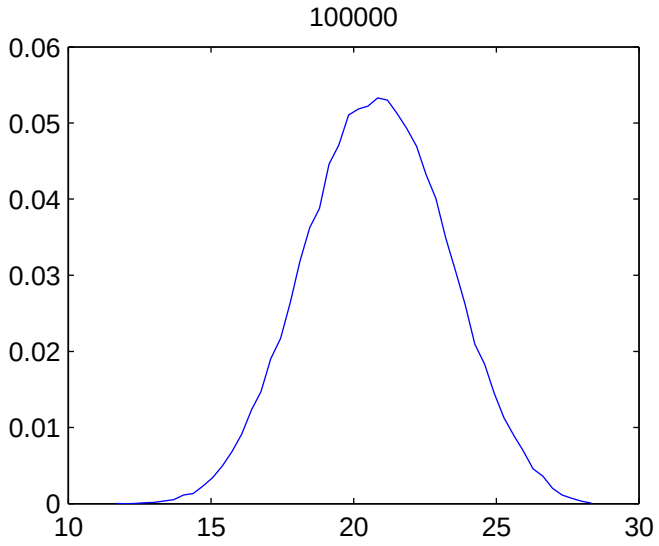
 % ZADÁNÍ VSTUPNÍCH DAT
 d = [4; 7; 3; 5; 6; 5; 4; 4; 0];   % doby trvání úkolů
 e = [1,2; 1,3; 2,6; 3,4; 3,5; 4,6; 4,7; 5,7; 5,8; 6,9; 7,9; 8,9];   % výčet hran

 % POMOCNÉ PRÁCE
 n = length(d);   % počet vrcholů (úkolů)
 h = [];          % zde budeme uchovávat simulované doby trvání projektů
 figure;

 % VLASTNÍ SIMULACE
 for j=1:10^5      % počet simulací
   d1 = d + [random('unif',-2,3, [n-1,1]); 0];
       % d1 jsou původní doby d plus náhodná chyba z Unif(-2,3) [kromě posledního
       % úkolu, ta je vždy 0]
   time = SolveCPM(d1,e);   % spočti dobu trvání projektu při dobách úkolů d1
   h = [h; time];           % do vektoru h ulož hodnotu time
   [freq, bins] = hist(h,50);
       % z dosud spočtených hodnot h spočti histogram s padesáti přihrádkami
       % freq jsou absolutní četnosti a bins jsou středy přihrádek
   plot(bins, freq./length(h));
       % vykresli (relativní) četnosti proti středům přihrádek
   title(j);
       % v hlavičce obrázku vypiš číslo iterace (ať víme, jak jsme daleko)
   pause(0.0001);   % formální pauza --- překreslí obrázek (refresh)
 end

 % NA ZÁVĚR: PÁR CHARAKTERISTIK
 disp(['Prumer = ',      num2str(mean(h))]);
 disp(['Max = ',         num2str(max(h))]);
 disp(['Min = ',         num2str(min(h))]);
 disp(['Smerodatna odchylka = ', num2str(var(h)^0.5)]);
 disp(['Median = ',      num2str(median(h))]);
 disp(['90% kvantil = ', num2str(quantile(h,0.9))]);
 disp(['95% kvantil = ', num2str(quantile(h,0.95))]);
 disp(['99% kvantil = ', num2str(quantile(h,0.99))]);
end
```

Výsledkem skriptu je simulované rozdělení doby trvání projektu a několik jeho základních charakteristik:



```
Prumer = 20.8116
Max = 28.5031
Min = 12.0171
Smerodatna odchylka = 2.4402
Median = 20.8179
90% kvantil = 23.9856
95% kvantil = 24.8517
99% kvantil = 26.2794
```

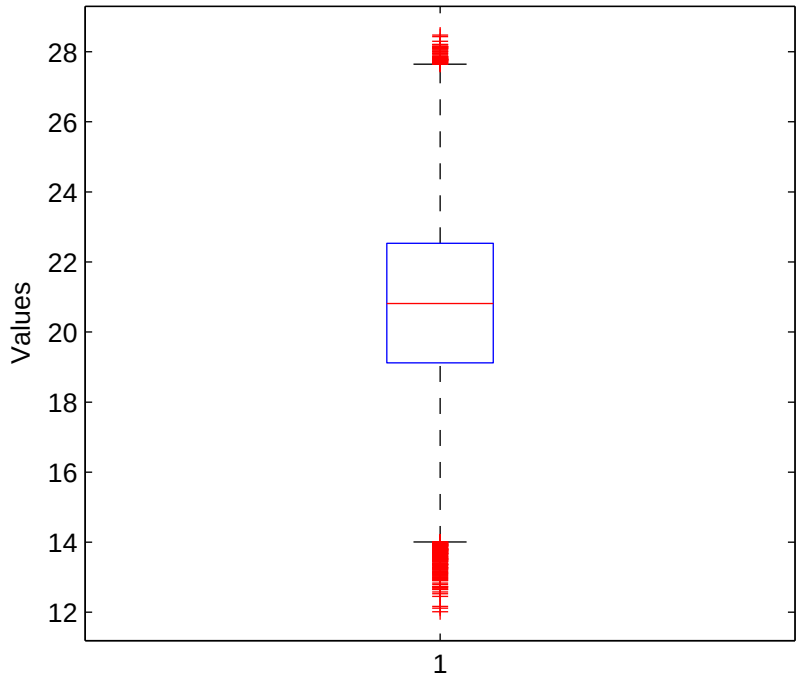
Poznámka. Ačkoliv simulovaná distribuce se může na první pohled jevit podobná normálnímu rozdělení (jak ji aproximuje metoda PERT), není tomu tak; již na druhý pohled je patrné sešikmení (jinými slovy: je zde větší tendence k prodlužování doby trvání projektu než k jeho rychlému dokončení). Můžeme zkusit otestovat shodu s normálním rozdělením např. Jarque-Berovým testem.[1] Stačí na konec skriptu přidat instrukci

```
[rozhodnuti, pvalue] = jbtest(h).
```

Výsledná hodnota `rozhodnuti` je 1, jestliže test na 5% hladině zamítá hypotézu o normalitě (a je 0, nezamítá-li), a `pvalue` je dosažená hladina testu. V našem případě je `pvalue` nerozlišitelně blízko nule; J.-B. test tedy hypotézu o shodě simulací získané distribuce s normálním rozdělením dokonce zamítá „zcela radikálně“.

Konečně je možné užít i další běžné nástroje pro vizualizaci dat, například nakreslit boxplot; stačí přidat instukci

```
figure;  boxplot(h).
```



1.7 Příklad 6: von Neumannův růstový model

Uvažme von Neumannův růstový model s nezápornou maticí vstupů $A \in \mathbb{R}^{n\times m}$ (input matrix) a nezápornou maticí výstupů $B \in \mathbb{R}^{n\times m}$ (output matrix). Víme, že von Neumannovo optimální tempo růstu γ^* a von Neumannovy optimální intenzity x^* producentů (aktivit) se získají jako řešení optimalizačního problému

$$\max_{\substack{\gamma\in\mathbb{R}\\ x\in\mathbb{R}^n}} \gamma \ \text{ s.t. } \ B^{\mathrm{T}}x \geq \gamma A^{\mathrm{T}}x, \ \ x \geq 0, \ \ x \neq 0. \tag{20}$$

[1] Připomeňme, že kouzlo J.-B. testu na shodu distribucí spočívá v následujícím. Další běžné testy — např. χ^2-test či Kolmogorov-Smirnovův test (v MATLABu implementovaný jako `kstest`) — dokáží testovat shodu empirické (v našem případě: simulované) distribuce s $N(\mu, \sigma^2)$ *při daných hodnotách* μ *a* σ^2. Ty ale nikdo nezná a jejich odhady pomocí výběrového průměru a rozptylu jsou zatíženy další chybou. Naproti tomu J.-B. test má za nulovou hypotézu, že pozorovaná empirická (simulovaná) distribuce patří *do třídy rozdělení* $\{N(\mu, \sigma^2) \mid \mu \in \mathbb{R},\ \sigma > 0\}$, a nevyžaduje tudíž znalost μ, σ^2. J.-B. test má vlastně za nulovou hypotézu „*existuje* μ a $\sigma > 0$ takové, že empirická (simulovaná) data tvoří náhodný výběr z $N(\mu, \sigma^2)$.“

Připomeňme, že (20) se také nazývá *primární von Neumannův problém* či *von Neumannův problém technologické expanze*.

Zřejmě je (20) nelineární optimalizační problém: jednak se zde vyskytuje součin proměnných $\gamma \cdot x_i$ a jednak zde máme podmínku $x \neq 0$. Součiny proměnných a omezující podmínky typu „$\neq$“ jsou v LP zakázány (a obecně jsou takové problémy **NP**-těžké).

Naším cílem je sestavit funkci `[gamma, x] = vonNeumann(A,B)`, které uživatel na vstup zadá dvojici input-output matic (A, B) a funkce spočte optimální tempo růstu `gamma` a optimální vektor intenzit `x`. Musíme si ovšem poradit právě s nelinearitou optimalizačního problému (20). Lze na to jít různě; jednou z možností je reformulovat (20) jako tzv. zobecněný lineárně-frakcionální program (*generalized linear-fractional programming problem*, GLFP); jde o typ nelineárních optimalizačních problémů, které jsou pomocí metod vnitřního bodu řešitelné zhruba stejně efektivně jako lineární programy. Touto cestou se nevydáme (vyžadovalo by to totiž příliš mnoho teorie) a ukážeme, jak problém (20) řešit s užitím LP.

Snadno se nahlédne, že systém nerovností $B^{\mathrm{T}}x \geq \gamma A^{\mathrm{T}}x$ je homogenní v x (to znamená: je-li x řešením, pak je také αx řešením pro libovolné $\alpha > 0$). A díky podmínkám $x \geq 0$, $x \neq 0$ můžeme bez újmy na obecnosti zafixovat z nekonečně mnoha řešení jedno konkrétní, například to, jehož složky se nasčítají na 1. Pišme proto

$$\max_{\substack{\gamma\in\mathbb{R}\\ x\in\mathbb{R}^n}} \gamma \ \text{ s.t. } \ B^{\mathrm{T}}x \geq \gamma A^{\mathrm{T}}x, \ \ x \geq 0, \ \ e^{\mathrm{T}}x = 1, \tag{21}$$

kde $e = (1, \ldots, 1)^{\mathrm{T}}$.

Z teorie víme, že problém (20) má vždy optimum s $\gamma^* > 0$ (za velmi mírných a přirozených předpokladů na input-output matice (A, B)). Pohleďme nyní na „lineární program“

$$(LP_\gamma) \quad \max_{x\in\mathbb{R}^n} 0^{\mathrm{T}}x \ \text{ s.t. } \ \underbrace{B^{\mathrm{T}}x \geq \gamma A^{\mathrm{T}}x, \ \ x \geq 0, \ \ e^{\mathrm{T}}x = 1}_{(\star)},$$

kde $\gamma > 0$ je *parametr*, nikoliv proměnná modelu. Pak se skutečně jedná o lineární program! Jen účelová funkce je zde identická nula — ale nám nepůjde o její maximalizaci či minimalizaci, to by bylo absurdní. Půjde nám o následující: pomocí solveru `linprog` chceme jen rozhodnout, při dané hodnotě γ, o *přípustnosti či nepřípustnosti* systému lineárních omezení $(\star)$. Proto je účelová funkce irelevantní a můžeme ji zvolit jakkoliv, třeba jako identickou nulu. Idea pro následující postup je tato: pro velké hodnoty γ je systém $(\star)$ jistě nepřípustný, zatímco pro malé hodnoty γ je přípustný; optimální hodnotu γ^* (a jí odpovídající optimální intenzity x^*) budeme hledat „někde mezi“. Učiníme tak zanedlouho metodou půlení intervalu.

Testování (ne)přípustnosti. Užijeme syntaxi

```
[x, optval, flag] = linprog(c,A,b,U,v);
```
(22)

zde solver `linprog` řeší lineární program tvaru $\min_x\{c^{\mathrm{T}}x \mid Ax \leq b, Ux = v\}$ a vrací optimální řešení `x`, optimální hodnotu účelové funkce `optval` a — pro nás nyní důležitou — hodnotu `flag`, která informuje o způsobu ukončení výpočtu solveru. Hodnota $\texttt{flag} = 1$ znamená, že bylo nalezeno optimum, a $\texttt{flag} = -2$ znamená, že problém je nepřípustný. (Pro úplnost dodejme, že $\texttt{flag} = -3$ znamená neomezenost, která ovšem v našem konkrétním případě nemůže nastat; další hodnoty `flag` značí vesměs numerické problémy při řešení rozsáhlých úloh LP či překročení povoleného počtu iterací. Detaily podá `help linprog`.)

Chceme-li otestovat přípustnost (LP_γ) (při zadané hodnotě parametru γ), pišme (LP_γ) v ekvivalentním tvaru

$$\max_{x\in\mathbb{R}^n} 0^{\mathrm{T}}x \ \text{ s.t. } \ (\gamma A^{\mathrm{T}} - B^{\mathrm{T}})x \leq 0, \ \ -x \leq 0, \ \ e^{\mathrm{T}}x = 1;$$

pak už okamžitě vidíme

$$\max_{x\in\mathbb{R}^n} \underbrace{0^{\mathrm{T}}}_{c^{\mathrm{T}}}x \ \text{ s.t. } \ \underbrace{\begin{pmatrix}\gamma A^{\mathrm{T}} - B^{\mathrm{T}}\\ -I\end{pmatrix}}_{D} x \leq \underbrace{\begin{pmatrix}0\\0\end{pmatrix}}_{b}, \ \ \underbrace{e^{\mathrm{T}}}_{U}x = \underbrace{1}_{v} \tag{23}$$

a voláme `[x, optval, flag] = linprog(c,D,b,U,v)`. Je-li $\texttt{flag} = 1$, pak je problém přípustný; jiank je nepřípustný (pomiňme zde možné další hodnoty `flag` z titulu numerických problémů). Pro stručnost volejme solver `linprog` v kompaktní formě

```
[x, optval, flag] = ...
      linprog( zeros(n,1) , [gamma.*A' - B'; -eye(n)] , zeros(m+n,1) , ones(1,n) , 1 ).
```
(c, D, b, U, v)

Binární vyhledávání (půlení intervalu). *Krok 1.* Zvolme velkou hodnotu $\overline{\gamma}$, a to tak, aby $(LP_{\overline{\gamma}})$ byl jistě nepřípustný; pro naše účely jistě postačí zvolit například $\overline{\gamma} = 10^4$. Zvolme také malou hodnotu $\underline{\gamma}$ (například $\underline{\gamma} = 10^{-8}$) tak, aby $(LP_{\underline{\gamma}})$ byl jistě přípustný.

Krok 2. Pohleďme doprostřed intervalu $[\underline{\gamma}, \overline{\gamma}]$: položme

$$\gamma' = \tfrac{1}{2}(\underline{\gamma} + \overline{\gamma}). \tag{24}$$

Krok 3. Otestujme přípustnost $(LP_{\gamma'})$.
Krok 4. Nyní:

(i) je-li $(LP_{\gamma'})$ přípustný, musí optimální γ^* ležet v intervalu $[\gamma', \overline{\gamma}]$. Položíme proto $\underline{\gamma} := \gamma'$ a pokačujeme Krokem 2 s novým, nyní již na polovinu zúženým intervalem $[\underline{\gamma}, \overline{\gamma}]$;

(ii) je-li ovšem $(LP_{\gamma'})$ nepřípustný, musí optimální γ^* ležet v intervalu $[\underline{\gamma}, \gamma']$. Položíme proto $\overline{\gamma} := \gamma'$ a pokračujeme Krokem 2 s novým, nyní již na polovinu zúženým intervalem $[\underline{\gamma}, \overline{\gamma}]$.

Postup opakujeme, dokud se nedosáhne zanedbatelně malé chyby, řekněme dokud není $\overline{\gamma} - \underline{\gamma} < 10^{-8}$. Takto jsme (sice jen přibližně, ale s libovolně malou chybou) nalezli optimální tempo růstu γ^* (a samozřejmě také jemu odpovídající vektor optimálních intenzit x^*). Všimněme si, že interval $[\underline{\gamma}, \overline{\gamma}]$ se zužuje geometrickou řadou; požadované přesnosti tudíž dosáhneme velice rychle (přibližně za $\log_2 \frac{10^4}{10^{-8}} \approx 40$ iterací).

Celý skript může vypadat například následujícím způsobem.

```
function [gamma, x] = vonNeumann(A,B)

  % INICIALIZACE POČÁTEČNÍCH HODNOT
  gammaUp  = 10^4;      % počáteční hodnota γ̄
  gammaLow = 10^-8;     % počáteční hodnota γ̲

  % HLAVNÍ CYKLUS BINÁRNÍHO VYHLEDÁVÁNÍ (PŮLENÍ INTERVALU)
  while (gammaUp - gammaLow >= 10^-8)        % dokud je interval příliš široký
    gammaPrime = mean([gammaUp, gammaLow]);  % prostředek intervalu, viz (24)
    [x0, optval, flag] = linprog(zeros(n,1),...
            [gammaPrime.*A' - B'; -eye(n)], ...
            zeros(m+n,1), ones(1,n), 1);     % test přípustnosti
    if flag == 1                             % problém je přípustný
      gammaLow = gammaPrime;                 % posuň dolní mez nahoru
      x = x0;                                % pamatuj si přípustné intenzity
      gamma = gammaPrime;                    % pamatuj si přípustné tempo růstu
    else                                     % problém je nepřípustný
      gammaUp = gammaPrime;                  % posuň horní mez dolů
    end
  end
  % VÝSTUPNÍ HODNOTOU JE POSLEDNÍ ZAPAMATOVANÉ PŘÍPUSTNÉ (gamma, x)
end
```

Poznámka. Půlením intervalu lze také analogicky řešit duální von Neumannův problém

$$\min_{\substack{\beta\in\mathbb{R}\\ y\in\mathbb{R}^m}} \ \beta \ \text{ s.t. } By \le \beta Ay, \ \ y \ge 0, \ \ y \ne 0,$$

jehož optimum (β^*, y^*) dává stínové ceny y^* produktů a optimální profit-faktor (optimální úrokovou sazbu) β^*. Detaily ponecháme k vypracování jako cvičení.

1.8 Příklad 7: Support Vector Machines (SVM)

Geometricky se podstata metody SVM dá vystihnout tak, že jsou dány dvě množiny bodů v $\mathbb{R}^n$ a cílem je najít nadrovinu, která je „dobře“ odděluje (tzv. *separátor*), anebo konstatovat, že je oddělit nelze. V praxi se metoda často užívá v data miningu, a to v situacích, které jsou analogické kapitole ?? — připomeňme, že tam jsme konstruovali logistický scoringový model pro klienty bank. Tuto aplikaci použijeme jako základní příklad i zde. Sestrojíme jednoduchý SVM-klasifikátor klientů; lze říci, že SVM-klasifikace je alternativní metodou ke scoringu založeném na statistických metodách.

Mějme dány dvě množiny klientů bank, říkejme jim *dobří* a *špatní* (např. na základě minulé zkušenosti, zdali řádně spláceli úvěry). Klient je charakterizován dvojicí údajů (x, y); řekněme, že jde výši majektu (x) a pravidelnou měsíční mzdu (y).

(V praktických aplikacích bývají takových údajů, tzv. *atributů*, desítky či stovky; zde volíme jen dva, aby bylo snadné kreslit obrázky.) Mějme n dobrých a m špatných klientů; jejich data uspořádejme do vektorů

$$\begin{aligned} \text{dobří klienti:} \quad & x = (x_1, \dots, x_n)^{\mathrm{T}}, \quad y = (y_1, \dots, y_n)^{\mathrm{T}}, \\ \text{špatní klienti:} \quad & \widetilde{x} = (\widetilde{x}_1, \dots, \widetilde{x}_m)^{\mathrm{T}}, \quad \widetilde{y} = (\widetilde{y}_1, \dots, \widetilde{y}_m)^{\mathrm{T}}. \end{aligned}$$

Najít separátor dobrých a špatných klientů obnáší najít koeficienty θ_1, θ_2 přímky

$$y = \theta_1 + \theta_2 x$$

takové, že

$$y_i \geq \theta_1 + \theta_2 x_i \ \ (\forall i = 1, \dots, n); \quad \widetilde{y}_j \leq \theta_1 + \theta_2 \widetilde{x}_j \ \ (\forall j = 1, \dots, m). \tag{25}$$

Znalost separátoru pak umožňuje klasifikovat nové klienty: přijde-li nový klient s majetkem x^0 a mzdou y^0, SVM-klasifikátor jej prohlásí za *dobrého*, jestliže $y^0 > \theta_1 + \theta_2 x^0$; a prohlásí jej za *špatného*, jestliže $y^0 < \theta_1 + \theta_2 x^0$. (Případ $y^0 = \theta_1 + \theta_2 x^0$, který teoreticky rovněž může nastat, můžeme například ztotožnit s odpovědí „nevím“).

Podmínky (25) jsou lineární v θ_1, θ_2, a tak se θ_1, θ_2 snadno najdou pomocí lineárního programování. Bude výhodné zvolit obecnější tvar: předpokládejme, že jsou namísto vektorů $x, \widetilde{x}$ dány matice $X \in \mathbb{R}^{n \times p}$ a $\widetilde{X} \in \mathbb{R}^{m \times p}$ a namísto (25) pišme

$$y \geq X\theta, \quad \widetilde{y} \leq \widetilde{X}\theta, \tag{26}$$

kde $\theta \in \mathbb{R}^p$ je vektor parametrů. Zvolíme-li nyní

$$p = 2, \quad X = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix}, \quad \widetilde{X} = \begin{pmatrix} 1 & \widetilde{x}_1 \\ 1 & \widetilde{x}_2 \\ \vdots & \vdots \\ 1 & \widetilde{x}_m \end{pmatrix}, \tag{27}$$

obdržíme přesně (25). Zvolíme-li ovšem například

$$p = 3, \quad X = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{pmatrix}, \quad \widetilde{X} = \begin{pmatrix} 1 & \widetilde{x}_1 & \widetilde{x}_1^2 \\ 1 & \widetilde{x}_2 & \widetilde{x}_2^2 \\ \vdots & \vdots & \vdots \\ 1 & \widetilde{x}_m & \widetilde{x}_m^2 \end{pmatrix}, \tag{28}$$

vyřešením systému (26) získáme vektor $\theta = (\theta_1, \theta_2, \theta_3)^{\mathrm{T}}$ takový, že dobré a špatné klienty odděluje parabola

$$y = \theta_1 + \theta_2 x + \theta_3 x^2;$$

SVM-klasifikátor pak nového klienta (x^0, y^0) klasifikuje podle nerovnosti $y^0 \lessgtr \theta_1 + \theta_2 x^0 + \theta_3 (x^0)^2$.

Obecně může být separátorů (tj. vektorů θ) splňujících (26) mnoho, nebo také nemusí existovat žádný.

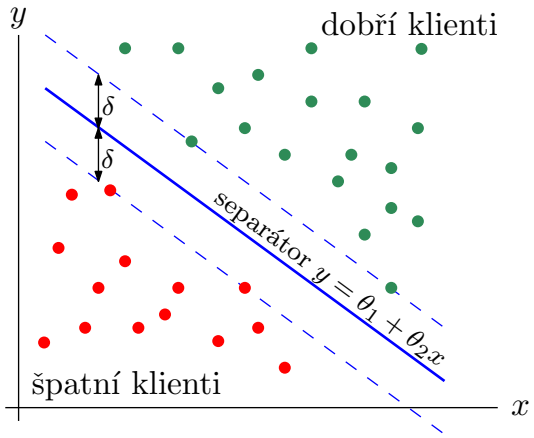
Případ, kdy je separátorů mnoho. V takovém případě bývá cílem najít separátor nikoliv coby přímku, ale coby *pás co největší šířky* oddělující body (x_i, y_i) od bodů $(\widetilde{x}_j, \widetilde{y}_j)$. Šířku pásu označme δ. Vyjdeme-li z (25), cílem je pak najít co největší δ takové, že

$$y_i \geq \delta + \theta_1 + \theta_2 x_i \ \ (\forall i = 1, \dots, n); \quad \widetilde{y}_j \leq -\delta + \theta_1 + \theta_2 \widetilde{x}_j \ \ (\forall j = 1, \dots, m).$$

Užijeme-li maticovou notaci z (26), cílem je najít co největší δ takové, že

$$y \geq \delta e + X\theta, \quad \widetilde{y} \leq -\delta e + \widetilde{X}\theta, \tag{29}$$

kde e je vektor samých jedniček příslušného rozměru. Situaci ilustruje následující obrázek, kde dobří klienti $(x_i, y_i)_{i=1,\dots,n}$ jsou zobrazeni jako zelené body v rovině a špatní klienti $(\widetilde{x}_j, \widetilde{y}_j)_{j=1,\dots,m}$ jsou zobrazeni jako červené body v rovině.



Vzhledem k (29) můžeme okamžitě psát lineární program

$$\max_{\delta,\theta} \delta \text{ s.t. } y \geq \delta e + X\theta, \;\; \widetilde{y} \leq -\delta e + \widetilde{X}\theta,$$

který snadno reformulujeme do verze vhodné pro solver `linprog`:

$$\min_{\binom{\delta}{\theta}} \begin{pmatrix} -1 & 0_{1\times p} \end{pmatrix} \begin{pmatrix} \delta \\ \theta \end{pmatrix} \text{ s.t. } \begin{pmatrix} e_{n\times 1} & X \\ e_{m\times 1} & -\widetilde{X} \end{pmatrix} \begin{pmatrix} \delta \\ \theta \end{pmatrix} \leq \begin{pmatrix} y \\ -\widetilde{y} \end{pmatrix}. \tag{30}$$

Nyní můžeme volat

```
xi = linprog([-1; zeros(p,1)], [ones(n+m,1), [X; -Xtilde]], [y; -ytilde]),
```

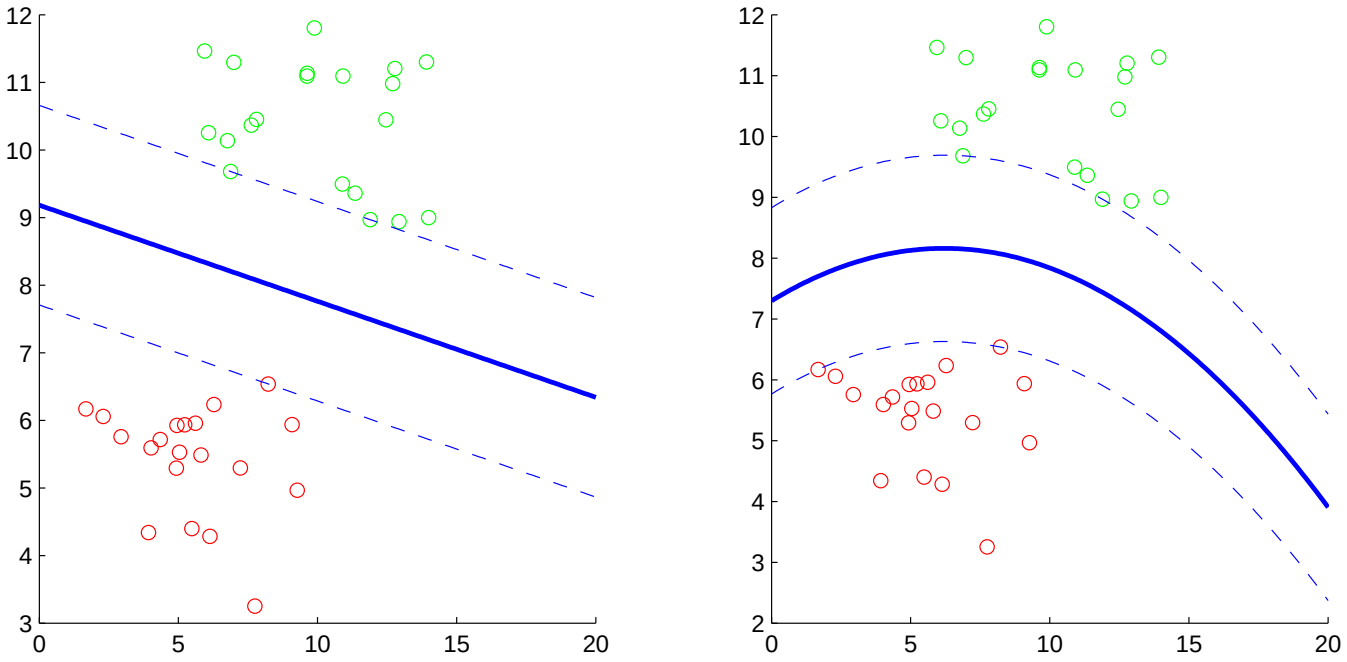
kde data špatných klientů $(\widetilde{X}, \widetilde{y})$ jsme označili jako (`Xtilde`, `ytilde`). Výsledkem je optimální vektor `xi`; jeho první složka `xi(1)` je optimální šíře oddělujícího pásu δ a `theta = xi(2:p+1)` je nalezený separátor θ, při němž je šíře oddělujícího pásu největší. Solver pro SVM je velice jednoduchý a nepotřebuje další komentář:

```
function [delta, theta] = SolveSVM(X, y, Xtilde, ytilde)
  [n,p] = size(X);
  [m,p] = size(Xtilde);
  xi = linprog([-1; zeros(p,1)], [ones(n+m,1), [X; -Xtilde]], [y; -ytilde]);
  delta = xi(1);
  theta = xi(2:p+1);
end
```

Můžeme zkusit testovat solver `SolveSVM` na simulovaných datech a nakreslit si obrázek. Řekněme, že náhodně vygenerujeme $n = 20$ dobrých klientů (x_i, y_i) a $m = 20$ špatných klientů $(\widetilde{x}_j, \widetilde{y}_j)$ například pomocí

$$x_i \sim N(10, 3^2), \quad y_i \sim N(10, 1), \quad \widetilde{x}_i \sim N(5, 2^2), \quad \widetilde{y}_i \sim N(5, 1). \tag{31}$$

Napišme skript, který s využitím již hotové funkce `SolveSVM` najde a vykreslí separátor ve tvaru přímky a paraboly (dobří klienti jako zelené body, špatní jako červené body):



Nejprve vygenerujeme náhodná data podle (31):

```
n = 20; m = 20;
x      = random('norm', 10, 3, [n,1]);
y      = random('norm', 10, 1, [n,1]);
xtilde = random('norm',  5, 2, [m,1]);
ytilde = random('norm',  5, 1, [m,1]);
```

Vytvoříme obrázek; do levé části budeme kreslit lineární separátor. Začneme tím, že vykreslíme vygenerované datové body.

```
figure; subplot(1,2,1); hold on;
plot(x, y, 'og',  xtilde, ytilde, 'or');
```

Vytvoříme matice X a $\widetilde{X} =$ `Xtilde` podle (27) a voláme `SolveSVM`.

```
X = [ones(n,1), x];
Xtilde = [ones(m,1), xtilde];
[delta, theta] = SolveSVM(X, y, Xtilde, ytilde);
```

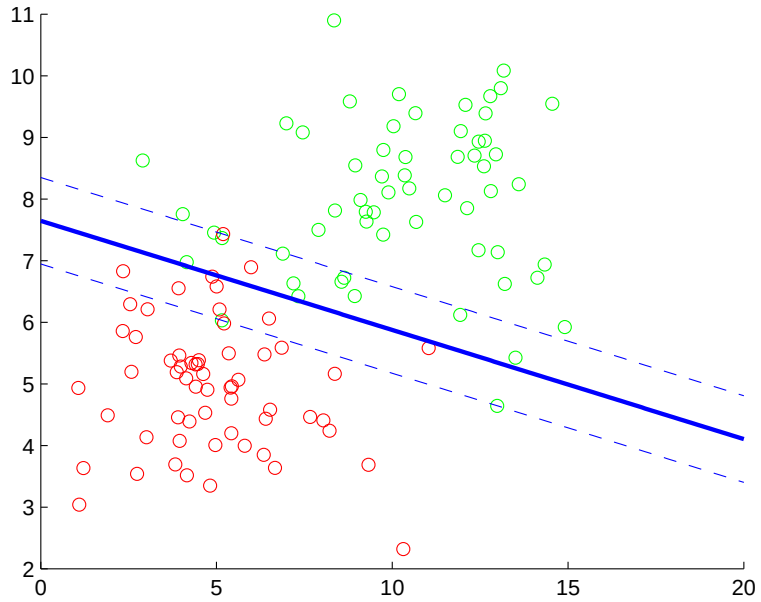
Konečně zbývá vykreslit trojici přímek — silněji vlastní separátor $y = \theta_1 + \theta_2 x$ a čárkovaně kraje oddělujícího pásu $y = \theta_1 + \theta_2 x \pm \delta$.

```
t = [0:0.1:20];
plot(t, theta(1) + theta(2)*t,         'b', 'LineWidth', 2);
plot(t, theta(1) + theta(2)*t - delta, 'b--');
plot(t, theta(1) + theta(2)*t + delta, 'b--');
```

Nyní celý postup zopakujeme (už bez komentářů); do pravého obrázku vykreslíme kvadratický separátor. Užijeme proto matice $X, \widetilde{X}$ ve tvaru (28).

```
subplot(1,2,2); hold on;
plot(x,y,'og',  xtilde, ytilde,'or');
X = [ones(n,1), x, x.^2];
Xtilde = [ones(m,1), xtilde, xtilde.^2];
[delta, theta] = SolveSVMx(X, y, Xtilde, ytilde);
t = [0:0.1:20];
plot(t,theta(1) + theta(2)*t + theta(3)*t.^2,         'b', 'LineWidth',2);
plot(t,theta(1) + theta(2)*t + theta(3)*t.^2 - delta, 'b--');
plot(t,theta(1) + theta(2)*t + theta(3)*t.^2 + delta, 'b--');
```

Případ, kdy separátor neexistuje. Samozřejmě se může stát, že body oddělit nejde; dobří a špatní klienti mohou být „promícháni“. To ovšem snadno poznáme; optimální hodnota δ lineárního programu (30) je pak záporná. Potom dolní a horní mez oddělujícího pásu jen vymění své role a výsledkem optimalizace bude *co nejužší pás obsahující oba typy klientů*; mimo pás už je separace jednoznačná. Klasifikátor pak typicky nově příchozího klienta spadajícího do oddělujícího pásu označí odpovědí „nevím“. Situace pak může vypadat jako na následujícím obrázku.



Očištění o odlehlé body. V praxi se často připouští, že některá data mohou být v jistém smyslu netypická, nereprezentativní, extrémní, či dokonce chybná; říkává se jim *odlehlé body* či *outliers*. Takové body mohou způsobit, že model má slabou klasifikační schopnost; optimální hodnota δ lineárního programu (30) je hodně záporná a pás odpovědí „nevím“ je hodně široký. Bývá proto povoleno z datového souboru několik bodů vynechat; cílem je vynechat právě ony odlehlé body (pokud v datech existují), které způsobují špatnou oddělitelnost.

Jak takové body najít? Přímo se nabízí intuitivně zřejmá úvaha. S ohledem na (30) položme

$$N = n + m$$

a

$$U = \begin{pmatrix} X \\ -\widetilde{X}' \end{pmatrix}, \quad v = \begin{pmatrix} y \\ -\widetilde{y} \end{pmatrix}. \tag{32}$$

Nechť matice U^k vznikne z matice U vypuštěním k-tého řádku, kde $k \in \{1, \ldots, N\}$. Analogicky, vektor v^k nechť vznikne z vektoru v vypuštěním k-té složky. Vyřešíme nyní lineární programy $(LP_1), \ldots, (LP_N)$ tvaru

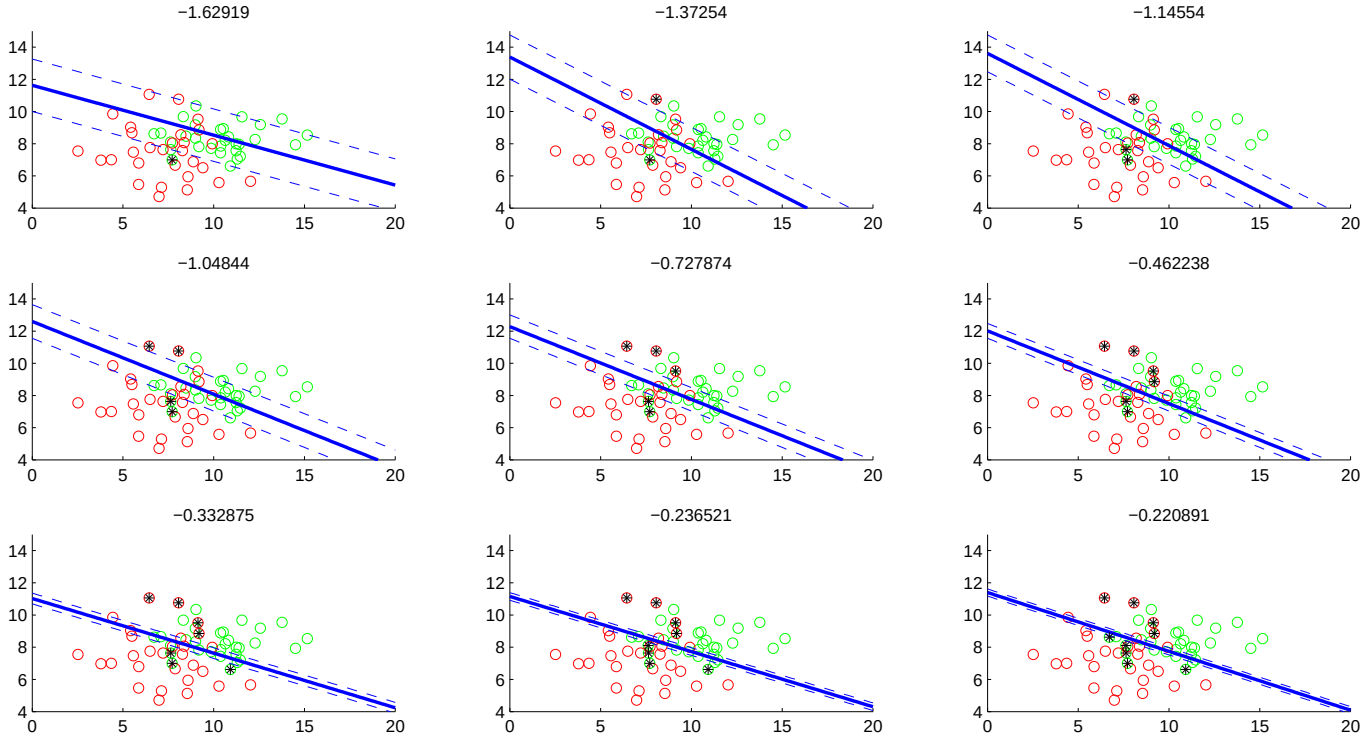
$$(LP_k) \quad \min_{\binom{\delta}{\theta}} \begin{pmatrix} -1 & 0_{1\times p} \end{pmatrix} \begin{pmatrix} \delta \\ \theta \end{pmatrix} \text{ s.t. } \begin{pmatrix} e_{(N-1)\times 1} & U^k \end{pmatrix} \begin{pmatrix} \delta \\ \theta \end{pmatrix} \leq v^k. \tag{33}$$

Jsou-li

$$\delta_1^*, \ldots, \delta_N^* \tag{34}$$

jejich optimální šíře dělících pásů, vybereme ten index i_0, pro nějž je $\delta_{i_0}^*$ největší. Tím jsme vlastně zjistili, že vypuštěním i_0-tého bodu dosáhneme nejlepší separace. Máme-li povoleno vypouštět více bodů, řekněme r, pak celou proceduru r-krát opakujeme.

Řekněme, že máme povoleno vypustit $r = 9$ bodů (číslo 9 volíme jen proto, abychom mohli vypouštění bod po bodu kreslit do obrázku 3×3; samozřejmě by bylo také možné skript animovat). Nakreslíme následující obrázek:



V prvním obrázku je výsledek separace po vyřazení prvního bodu (označen černou hvězdičkou); číslo nad obrázkem je šíře pásu δ. Ve druhém obrázku jsou vypuštěny dva body (označeny černě), a tak dále; v posledním obrázku je vypuštěno devět bodů a šíře oddělujícího pásu (byť stále záporná) je velmi blízko nuly. Body jsme generovali (jako příklad) pro $n = 30$ a $m = 30$ s $x_i \sim N(10, 3^2)$, $y_i \sim N(8, 1)$, $\widetilde{x}_i \sim N(7, 2^2)$ a $\widetilde{y}_i \sim N(7, 2^2)$.

```
% VYGENERUJEME NÁHODNÁ DATA
n = 30; m = 30; N = n+m;
x      = random('norm', 10, 3, [n,1]);
y      = random('norm',  8, 1, [n,1]);
xtilde = random('norm',  7, 2, [m,1]);
ytilde = random('norm',  7, 2, [m,1]);

% PŘÍPRAVA MATIC
X  = [ones(n,1), x];  Xtilde  = [ones(n,1), xtilde];  % X, X' podle (27)
U  = [X; -Xtilde]; v  = [y; -ytilde];                 % U, v podle (32)
xp = [x; xtilde];  yp = [y; ytilde];
       % uložíme si současnice bodů, použijeme je později pro vizualizaci
 r = 9; % počet povolených bodů k vypuštění

% PŘÍPRAVA VIZUALIZACE
figure;
for l = 1:r     % devět obrázků, do každého vykreslíme červené a zelené body
  subplot(3,3,l); hold on;
  plot(x,y,'og', xtilde,ytilde,'or');
  axis([0,20,4,15])  % ať je na každém obrázku stejné měřítko
  hold on;
end

% HLAVNÍ CYKLUS - VYPOUŠTÍME 9 BODŮ
for i = 1:r
  delta  = [];   % zde si ukládáme hodnoty delta (34)
  Thetas = [];   % zde si ukládáme jim odpovídající vektory theta

  % VYPUŠTĚNÍ JEDNOHO BODU
  for k = 1:N
    Uk = U([1:k-1, k+1:N], :);   % vypouštíme k-tý bod
    vk = v([1:k-1, k+1:N]);      % vypouštíme k-tý bod
    xi = linprog([-1; zeros(2,1)], [ones(N-1,1), Uk], vk);  % LP (33)
    delta(k) = xi(1);            % uložíme optimální delta
    Thetas(:,k)= xi(2:3);        % uložíme i nalezený separátor
  end

  [deltaopt, j] = max(delta);
      % j = index, kde je delta nejvyšší, tento bod budeme vypouštět z datasetu
  theta = Thetas(:,j);           % theta = separátor odpovídající delta(j)

  % VIZUALIZACE: VYKRESLENÍ SEPARÁTORU A ODDĚLUJÍCÍHO PÁSU
  subplot(3,3,i); hold on;
  t = [0:0.1:20];
  plot(t, theta(1) + theta(2)*t, 'b', 'LineWidth', 2);
  plot(t, theta(1) + theta(2)*t - deltaopt, 'b--');
  plot(t, theta(1) + theta(2)*t + deltaopt, 'b--');

  title(deltaopt)  % hodnotu delta vypíšeme nad obrázek

  % VIZUALIZACE: OZNAČENÍ VYPUŠTĚNÉHO BODU
  for l = i:r      % černou hvězdičkou vyznačíme vypuštěný bod
    subplot(3,3,l)
    plot(xp(j),yp(j),'*k','MarkerSize',6); % v xp,yp máme uloženy souřadnice bodů
  end

  % BOD S INDEXEM j VYMAŽEME Z DATASETU U, v, xp, yp
  U = U([1:j-1,j+1:N],:);
  v = v([1:j-1,j+1:N]);
  xp = xp([1:j-1,j+1:N],:);
  yp = yp([1:j-1,j+1:N]);
  N = N-1;                 % velikost datasetu se zmenšila o jedničku
end                        % konec hlavního for-cyklu
```

Poznámka. Úlohu vypustit r bodů tak, aby šířka pásu δ byla co největší, lze také psát jako *smíšený* lineární program

$$\max_{\substack{\delta\in\mathbb{R}\\ \theta\in\mathbb{R}^2\\ w\in\{0,1\}^n\\ z\in\{0,1\}^m}} \delta \text{ s.t. } \underbrace{y_i + Mw_i \geq \delta + \theta_1 + \theta_2 x_i \ (\forall i = 1,\dots n)}_{(\star)},\ \underbrace{\widetilde{y}_j - Mz_j \leq -\delta + \theta_1 + \theta_2 \widetilde{x}_j \ (\forall j = 1,\dots n)}_{(\dagger)},\ \underbrace{e^{\mathrm{T}}w + e^{\mathrm{T}}z = r}_{(\ddagger)}, \tag{35}$$

kde M je pevně zvolené velké číslo. Nula-jedničkové proměnné w, z jsou indikátory bodů, které jsou vypuštěny. Omezení $(\ddagger)$ říká, že jich musí být právě r. Je-li $w_i = 1$, pak i-té omezení v $(\star)$ je jistě splněno (levá strana je jistě větší než pravá, je-li M

dostatečně velké, bez ohledu na hodnoty x_i, y_i), a tedy model se chová, jako by i-tý bod mezi dobrými klienty nebyl v datech přítomen. Analogický je význam aditivního členu $-Mz_j$ v (†).

Zřejmě je (35) smíšený lineární program, některé proměnné jsou totiž diskrétní. Nelze jej tudíž řešit pomocí solveru `linprog`. Jak lze takové problémy řešit, ukážeme v kapitole 1.11.

1.9 Příklad 8: Dopravní problém

Dopravní problém je následující úloha: je dán seznam m dodavatelů jisté komodity a n odběratelů této komodity. Úkolem je přepravit komoditu od dodavatelů k odběratelům s tím, že je dán vektor $p = (p_1, \ldots, p_n)^{\mathrm{T}}$ požadavků (odběratel i chce odebrat právě množství p_i) a vektor $k = (k_1, \ldots, k_m)^{\mathrm{T}}$ kapacit dodavatelů (dodavatel j disponuje jen omezeným množstvím k_j). Dále je zadána nezáporná matice $C = (c_{ij}) \in \mathbb{R}^{n\times m}$ jednotkových cen přeprav — c_{ij} je cena přepravy jedné tuny komodity od dodavatele j k odběrateli i. Cílem je zajist přepravu, která uspokojí požadavky odběratelů, nepřekročí kapacity dodavatelů a bude co nejlevnější. (Všimnemě si, že úloha má řešení právě tehdy, když $e^{\mathrm{T}}k \geq e^{\mathrm{T}}p$; tuto podmínku je snadné ověřit.)

Problém se snadno zformuluje jako LP s proměnnými x_{ij} = objem přepravy mezi dodavatelem j a odběratelem i:

$$\min_{x_{ij}} \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} \quad \text{s.t.} \quad \underbrace{\sum_{j=1}^{m} x_{ij} = p_i \ (\forall i = 1,\ldots,n)}_{(\star)}, \quad \underbrace{\sum_{i=1}^{n} x_{ij} \leq k_j \ (\forall j = 1,\ldots,m)}_{(\dagger)}, \quad x_{ij} \geq 0 \ (\forall i,j). \tag{36}$$

Podmínky $(\star)$ uspokojí požadavky odběratelů a podmínky (†) zajistí, že se nepřekročí kapacity dodavatelů.

Naším cílem zde je napsat funkci `X = DopravniProblem(C, k, p)`, kde uživatel na vstup zadá data C, k, p a obdrží matici `X` $= (x^*_{ij})$ optimálních převozů. Zvolíme-li na chvíli jako příklad $m = 3$ dodavatele a $n = 4$ odběratele, můžeme (36) přepsat do podoby, kde „naskládáme“ proměnné a ceny do jednoho vektoru po sloupcích

$$x = (x_{11}, x_{21}, x_{31}, x_{41},\ x_{12}, x_{22}, x_{32}, x_{42},\ x_{13}, x_{23}, x_{33}, x_{43})^{\mathrm{T}}, \tag{37}$$

$$c = (c_{11}, c_{21}, c_{31}, c_{41},\quad c_{12}, c_{22}, c_{32}, c_{42},\quad c_{13}, c_{23}, c_{33}, c_{43})^{\mathrm{T}} \tag{38}$$

a obdržíme (36) ve tvaru

$$\min_x c^{\mathrm{T}}x \ \text{ s.t. } \underbrace{\begin{pmatrix} 1 & & & & 1 & & & & 1 & & & \\ & 1 & & & & 1 & & & & 1 & & \\ & & 1 & & & & 1 & & & & 1 & \\ & & & 1 & & & & 1 & & & & 1 \end{pmatrix}}_{A_1} x = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix},$$

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & 1 & & & & & & & & \\ & & & & 1 & 1 & 1 & 1 & & & & \\ & & & & & & & & 1 & 1 & 1 & 1 \end{pmatrix}}_{A_2} x \leq \begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix},$$

$$-x \leq 0.$$

(Prázdné prvky v maticích A_1, A_2 jsou nuly.) Pro obecné m, n uvážíme matice A_1, A_2 analogické struktury a získáme (36) v podobě

$$\min_x c^{\mathrm{T}}x \ \text{ s.t. } \underbrace{(I_{n\times n} \quad I_{n\times n} \quad \cdots \quad I_{n\times n})}_{A_1} x = p, \quad \underbrace{\begin{pmatrix} e^{\mathrm{T}}_{1\times n} & & & \\ & e^{\mathrm{T}}_{1\times n} & & \\ & & \ddots & \\ & & & e^{\mathrm{T}}_{1\times n} \end{pmatrix}}_{A_2} x \leq k, \quad -x \leq 0. \tag{39}$$

(Aby nedošlo k nedorozumění: $e^{\mathrm{T}}_{1\times m}$ je řádkový vektor m jedniček.) K vyřešení dopravního problému zřejmě stačí volat

`linprog(`c`, [`A_2`; -eye(`m`*`n`)], [`k`; zeros(`m`*`n`,1)],` A_1`,` p`).`

Jak lze vytvořit vektor c a matice A_1, A_2? Jedna možnost je pochopitelně pomocí `for`-cyklů, například

```
c  = [];  for j=1:m; C  = [C; c(:,j)]; end;
A1 = [];  for j=1:m; A1 = [A1, eye(n)]; end;
A2 = [];  for j=1:m; A2 = [A2, zeros(j-1,n); zeros(n*(j-1),1), ones(1,n)]; end;
```

Existují ale i elegantnější způsoby. Příkaz

$$c = \texttt{reshape(}C\texttt{, [}m\texttt{*}n\texttt{, 1])}$$

učiní přesně operaci „přeskládání“ matice C do vektoru rozměru $mn \times 1$ podle (38). Matice A_1 a A_2 lze jednoduše vytvořit pomocí tzv. Kroneckerova součinu. Jsou-li dány matice $U = (u_{ij}) \in \mathbb{R}^{k_1 \times \ell_1}$ a $V \in \mathbb{R}^{k_2 \times \ell_2}$, pak *Kroneckerův součin* $U \otimes V$ je matice rozměru $k_1 k_2 \times \ell_1 \ell_2$ s bloky

$$U \otimes V = \begin{pmatrix} u_{11}V & u_{12}V & \cdots & u_{1,\ell_1}V \\ u_{21}V & u_{22}V & \cdots & u_{2,\ell_1}V \\ \vdots & \vdots & \ddots & \vdots \\ u_{k_1,1}V & u_{k_1,2}V & \cdots & u_{k_1,\ell_1}V \end{pmatrix},$$

například

$$\begin{pmatrix} 1 & 2 & 3 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 2 & 0 & 3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 3 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

V MATLABu se Kroneckerův součin počítá pomocí funkce `kron`. Z (39) je patrno, že

$$A_1 = e^{\mathrm{T}}_{1\times m} \otimes I_{n\times n}, \quad A_2 = I_{m\times m} \otimes e^{\mathrm{T}}_{1\times n},$$

a tedy můžeme matice A_1, A_2 vytvořit stručným příkazem

$$A_1 = \texttt{kron(ones(1,}m\texttt{), eye(}n\texttt{))}, \quad A_2 = \texttt{kron(eye(}m\texttt{), ones(1,}n\texttt{))}.$$

Nyní již máme všechny ingredience a můžeme napsat jednoduchý solver pro dopravní problém. Uživatel zadá (sloupcové) vektory k, p a matici C a obdrží matici $X = (x^*_{ij})$ s optimální přepravou.

```
function X = SolveTP(k, p, C)
  [n,m] = size(C);
  c  = reshape(C,[m*n, 1]);
  A1 = kron(ones(1,m), eye(n));
  A2 = kron(eye(m), ones(1,n));
  x  = linprog(c, [A2; -eye(m*n)], [k; zeros(m*n,1)], A1, p);
  X  = reshape(x,[n,m]);
end
```

Poslední příkaz jen „přeskládá“ získané optimální řešení `x` tvaru (37) do matice `X` tvaru $n \times m$.

Poznámka. Stejně — jako v dalších kapitolách — i zde je vhodné si jako cvičení rozmyslet, jak by vypadalo řešení rozličných modifikací dopravního problému. Například: řekneme, že instance (k, p, C) dopravního problému vykazuje tzv. *more-for-less paradox*, jestliže je možné zvýšením kapacit (některých) dodavatelů a adekvátním zvýšením požadavků (některých) odběratelů dosáhnout přepravy celkově většího množství *a přitom snížit celkové přepravní náklady*. Zdali more-for-less paradox nastává, lze zjistit pomocí vhodného LP — jako cvičení je možné rozšířit skript `SolveTP`, aby uživateli podal tuto informaci. Podobně je možné rozšířit skript o vizualizaci výsledné optimální přepravy jako v kapitole 1.5; umístění dodavatelů a odběratelů lze vykreslit coby body v rovině a mezi nimi zobrazit šipky, jejichž síla odpovídá transportovanému množství. A konečně zájemce se může podívat na toolbox *mapping* — pomocí funkcí `worldmap` a `geoshow` lze vykreslit (například) mapu konkrétního státu, a pak je možné toky přepravy zobrazovat na reálné mapě. (Kreslení map ovšem přesahuje rámec tohoto textu.)

1.10 Příklad 9: Data Envelopment Analysis (DEA)

DEA je oblíbená rankovací metoda. Existuje v nepřeberném množství variant a modifikací; my zde ukážeme jen jednoduchý základní model. Je dán systém p jednotek (tzv. *decision making unit, DMU*), z nichž každá spotřebovává jisté vstupy a generuje jisté výstupy, a jednotky jsou homogenní v tom smyslu, že generují stejné typy výstupů při stejných typech vstupů. Jednotkou může být v konkrétní aplikaci takřka cokoliv, o čem lze rozumně prohlásit, že spotřebovává vstupy a transformuje je na výstupy. Může jít o podniky ve stejném odvětví (vstupy jsou například vybavenost pracovní silou a kapitálem, výstupy jsou výrobky několika druhů); může jít o nemocnice (vstupem jsou např. počty lékařů, sester a vybavenost technikou; výstupy jsou objemy hospitalizací, počty operací, ambulatní výkony atd.), může jít o lidi (u učitele může být vstupem mzda, úroveň kvalifikace a délka praxe, výstupy mohou být objem vědeckých výkonů a objem pedagogických výkonů); může jít o státy či regiony (vstupem je např. struktura pracovní síly, vybavenost infrastrukturou apod., výstupem je HDP).

Cílem DEA je stanovit *relativní srovnání* (ranking) daného systému jednotek, u nichž známe vstupy a výstupy. Potíž je, že vstupy i výstupy jsou různého druhu; je-li jednotkou telekomunikační společnost, jejími výstupy může být objem datových služeb

a objem hlasových služeb *a není jasné, podle kterého kritéria jednotky srovnávat* — jedna jednotka může být „lepší“ v datových službách, jiná v hlasových službách, a není jasné, jak nesrovnatelná kritéria sloučit a udělat jednoznačné porovnání.

V rámci DEA metodiky se definuje *efektivita* jednotky jako *vážený součet výstupů ku váženému součtu vstupů*. Označme $a_k \geq 0$ vektor výstupů a $b_k \geq 0$ vektor vstupů k-té jednotky ($k = 1, \ldots, p$); ty jsou uživatelem zadány. Efektivita k-té jednotky je podle definice

$$\mathrm{ef}_k = \frac{a_k^{\mathrm{T}} v}{b_k^{\mathrm{T}} w}, \tag{40}$$

kde $v \geq 0$ a $w \geq 0$ jsou vektory vah (později váhy omezíme podmínkou zajišťující $\mathrm{ef}_k \in [0, 1]$). Vše by bylo jednoduché, kdyby váhy v, w byly známé — kdyby byl někdo třeba schopen říci, že jeden gigabyte přenesených dat je ekvivalentní deseti hlasovým spojením; kdyby byl někdo schopen říci, že jedna transplantace je ekvivalentní deseti ambulatním vyšetřením; nebo kdyby byl někdo schopen říci, že učitel vychovávající tři doktorandy je ekvivalentní učiteli publikujícímu jednu vědeckou monografii. Pak by nebylo co řešit: prostě bychom jednotky mohli srovat podle efektivit (40) spočtených se známými váhami. Zde přichází hlavní idea DEA metodiky. Váhy jsou neznámé; dovolme tedy jednotce $k \in \{1, \ldots, p\}$, ať si *zvolí váhy v, w sama, a to tak, aby to pro ni bylo co nejvýhodnější* (aby dosáhla co nejvyšší efektivity). Nicméně poté musí své váhy předložit ostatním jednotkám; pokud by některá jiná jednotka dosáhla při těchto vahách lepšího efektivitního poměru, prohlásíme jednotku k za *DEA-neefektivní* ($\mathrm{ef}_k < 1$); jinak ji prohlásíme za *DEA-efektivní* ($\mathrm{ef}_k = 1$).

Vektory zadaných výstupů a vstupů uspořádejme do matic

$$A = \begin{pmatrix} a_1^{\mathrm{T}} \\ a_2^{\mathrm{T}} \\ \vdots \\ a_p^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{p \times n}, \quad B = \begin{pmatrix} b_1^{\mathrm{T}} \\ b_2^{\mathrm{T}} \\ \vdots \\ b_p^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{p \times m}$$

(dvojici B, A se také říká *input-output matice*). Symbolem n jsme označili počet výstupů a symbolem m jsme označili počet vstupů. Ideu DEA — ať si k-tá jednotka zvolí své váhy, jak nejlépe dokáže — snadno napíšeme jako (nelineární) optimalizační problém

$$\max_{\substack{v \in \mathbb{R}^n \\ w \in \mathbb{R}^m}} \frac{a_k^{\mathrm{T}} v}{b_k^{\mathrm{T}} w} \text{ s.t. } 0 \overset{(\star)}{\leq} \frac{a_\ell^{\mathrm{T}} v}{b_\ell^{\mathrm{T}} w} \overset{(\dagger)}{\leq} 1 \ (\forall \ell = 1, \ldots, p), \ \ v, w \geq 0. \tag{41}$$

Omezující podmínky $(\dagger)$ vlastně formalizují test, kdy se váhy v, w volené jednotkou k předkládají jednotkám $\ell = 1, \ldots, p$ a všechny efektivity se normalizují na škálu $[0, 1]$. Výsledný DEA-ranking se získá tak, že každá jednotka $k = 1, \ldots, p$ vyřeší problém (41) a jednotky se pak seřadí podle dosažených efektivit.

Problém (41) se snadno zlinearizuje. Omezení $(\star)$ jsou redundantní, čísla $a_\ell^{\mathrm{T}} v$, $b_\ell^{\mathrm{T}} w$ jsou totiž nezáporná. Díky nezápornosti $b_\ell^{\mathrm{T}} w$ můžeme $(\dagger)$ přepsat do lineárního tvaru

$$a_\ell^{\mathrm{T}} v \leq b_\ell^{\mathrm{T}} w.$$

Všimněme si, že váhy nejsou nikdy jednoznačné — hodnota výrazu $\frac{a_k^{\mathrm{T}} v}{b_k^{\mathrm{T}} w}$ se nezmění, nahradíme-li váhy v, w váhami $\alpha v, \alpha w$ pro libovolné $\alpha > 0$. Proto můžeme bez újmy na obecnosti váhy standardizovat; hodí se omezit se jen na váhy splňující $b_k^{\mathrm{T}} w = 1$. Tím získáme lineární program

$$(\mathrm{DEA}_k) \quad \max_{\substack{v \in \mathbb{R}^n \\ w \in \mathbb{R}^m}} a_k^{\mathrm{T}} v \text{ s.t. } a_\ell^{\mathrm{T}} v \leq b_\ell^{\mathrm{T}} w \ (\forall \ell = 1, \ldots, p), \ \ b_k^{\mathrm{T}} w = 1, \ \ v, w \geq 0. \tag{42}$$

Nyní se vyřeší řada LP $(\mathrm{DEA}_1), \ldots, (\mathrm{DEA}_p)$ a jejich optimální účelové hodnoty (= efektivity) dají výsledný ranking.

Přepišme (42) do maticového tvaru vhodného pro solver `linprog(`c`, `D`, `b`, `U`, `z`)`:

$$\min_{x = \binom{v}{w}} \underbrace{\begin{pmatrix} -a_k^{\mathrm{T}} & 0_{1\times m} \end{pmatrix}}_{c^{\mathrm{T}}} \underbrace{\begin{pmatrix} v \\ w \end{pmatrix}}_{x} \text{ s.t. } \underbrace{\begin{pmatrix} A & -B \\ -I_{n\times n} & 0_{n\times m} \\ 0_{m\times n} & -I_{m\times m} \end{pmatrix}}_{D} \underbrace{\begin{pmatrix} v \\ w \end{pmatrix}}_{x} \leq \underbrace{\begin{pmatrix} 0_{p\times 1} \\ 0_{n\times 1} \\ 0_{m\times 1} \end{pmatrix}}_{b}, \quad \underbrace{\begin{pmatrix} 0_{1\times n} & b_k^{\mathrm{T}} \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} v \\ w \end{pmatrix}}_{x} = \underbrace{1}_{z}. \tag{43}$$

Napišme skript, kde uživatel na vstup zadá matice výstupů a vstupů (A, B) a výsledkem bude vektor `ef` efektivit jednotek. Ve `for`-cyklu budeme řešit lineární program (DEA_k) pro $k = 1, \ldots, p$. Všimněme si v (43), že D, b nezávisí na k; můžeme si

tedy D, b připravit ještě před `for`-cyklem.

```
function ef = SolveDEA(A, B)
  [p,n] = size(A);
  [p,m] = size(B);
  D = [A, -B; -eye(n+m)];       % viz (43)
  b = zeros(p+n+m,1);           % viz (43)
  for k=1:p
    c = [-A(k,:), zeros(1,m)];  % viz (43)
    U = [zeros(1,n), B(k,:)];   % viz (43)
    x = linprog(c, D, b, U, 1);
    ef(k) = A(k,:)*x(1:n);
      % uložili jsme optimální hodnotu účelové funkce (=efektivitu),
      % A(k,:)*x(1:n) je hodnota účel. funkce, x(1:n) jsou optimální váhy v
  end
end
```

Poznámka. Často je rozumné předpokládat, že data (A, B) pro DEA analýzu jsou náhodné veličiny: počty telefonních hovorů, počty vyšetření pacientů či počty vychovaných studentů totiž často lze chápat jako výsledek jistého náhodného procesu (například to, kolik přijde pacientů do nemocnice, je do jisté míry náhoda). Formálně: předpokládejme, že data (A, B) jsou náhodné veličiny se známou distribucí (např. empirickou distribucí získanou měřením počtu příchozích pacientů v několika obdobích). Pak i efektivity jsou náhodné veličiny a je rozumné nasimulovat jejich rozdělení podobně jako v kapitolách 1.3 a 1.6. Tato simulace nám podá informaci například o *robustnosti* či *stabilitě* DEA-rankingu. Zůstane-li totiž jednotka DEA-efektivní i poté, co (náhodně) perturbujeme data — a modeluje-li tato perturbace, co se v realitě skutečně může stát —, ukazuje to, že její DEA-efektivita zůstává stabilní, i když se vnější podmínky mění. To je kvalitativní rozdíl oproti jednotce, která je sice při daných datech (A, B) DEA-efektivní, ovšem při náhodných perturbacích dat vykazuje její DEA-efektivita velký rozptyl.

1.11 Příklad 10: Celočíselné lineární programování (ILP) a Branch-and-Bound (B&B)

Až dosud jsme se zabývali lineárním programováním se spojitými proměnnými ($x \in \mathbb{R}^n$). MATLAB disponuje i solverem pro celočíselné a smíšené lineární programování (*ILP*, Integer Linear Programming; *MILP*, Mixed Integer Linear Programming). Solver má základní syntaxi analogickou solveru `linprog`, totiž

$$\texttt{intlinprog(c, I, A, b, U, v)},$$

která řeší problém

$$\min_x \; c^{\mathrm{T}}x \;\text{ s.t. }\; Ax \le b,\; Ux = v,\; x_i \in \begin{cases} \mathbb{Z}, & i \in I, \\ \mathbb{R}, & i \notin I. \end{cases}$$

Tedy: I je množina (vektor) indexů proměnných, které mají nabývat celočíselných hodnot. Mají-li být všechny proměnné celočíselné, stačí volit `I = [1:n]'`, kde n je počet proměnných.

Jako cvičení nyní doporučujeme vyzkoušet `intlinprog` na běžných problémech formulovatelných jako ILP, například problém batohu, problém obchodního cestujícího, job scheduling či různé další typy přiřazovacích problémů.

Připomeňme, že narozdíl od (spojitého) lineárního programování je celočíselné lineární programování obecně **NP**-těžké; není tedy snadné řešit rozsáhlejší instance. Byť je solver v MATLABu dobrý, patrně se nemůže srovnávat s CPLEXem a dalším podobným software (ovšem toto tvrzení by by bylo třeba potvrdit řádnou srovnávací studií). Potřebuje-li uživatel řešit reálné problémy ILP, vždy se doporučuje využít raději CPLEX či další speciální software.

My si v této kapitole zkusíme vytvořit vlastní jednoduchý solver pro ILP založený na metodě Branch-and-Bound (B&B). Pro řešení praktických problémů je to čirý nerozum: nikdy se nám totiž nepodaří naprogramovat něco, co by alespoň vzdáleně dokázalo konkurovat profesionálnímu software, a rozhodně se nic takového nedoporučuje. Psát vlastní solver má smysl jen tehdy, chce-li si uživatel vyzkoušet vlastnosti algoritmu a „hrát“ si s ním — chce-li jeho práci vizualizovat, nebo chce-li například zkoušet porovnávat efektivitu různých strategií volby indexu pro B&B-branching a tak podobně.

Úvod k B&B algoritmu. Připomeňme, že B&B algoritmus pro ILP je následující procedura, která instanci ILP redukuje na výpočet řady spojitých lineárních programů. Z technických důvodů bude na tomto místě pro nás jednodušší řešit ILP ve tvaru

$$\min_{x \in \mathbb{Z}^n} c^{\mathrm{T}}x \;\text{ s.t. }\; Ax \le b,\;\; Ux = v,\;\; \underline{x} \le x \le \overline{x}. \tag{44}$$

Napíšeme skript

$$\texttt{x = MyBaB(}c,\; A,\; b,\; U,\; v,\; \underline{x},\; \overline{x}\texttt{)},$$

který tuto úlohu řeší a vrací optimální řešení jako vektor `x`. Všimněme si, že díky existenci dolních a horních mezí $\underline{x}, \overline{x}$ se nemusíme zabývat neomezeností (to je jen proto, aby náš skript byl co nejjednodušší — nicméně jako cvičení nechť čtenář rozpracuje problém v plné obecnosti).

Jako podprogram budeme užívat standardní solver `linprog`, tentokrát v syntaxi

$$\texttt{[x, ov, flag] = linprog(}c\texttt{, }A\texttt{, }b\texttt{, }U\texttt{, }v\texttt{, }\underline{x}\texttt{, }\overline{x}\texttt{)},$$

který řeší LP

$$\min_{x\in\mathbb{R}^n} c^{\mathrm{T}}x \text{ s.t. } Ax \le b,\ Ux = v,\ \underline{x} \le x \le \overline{x} \tag{45}$$

(to je tzv. *relaxace* problému (44)). Výsledkem výpočtu je optimální řešení `x`, optimální hodnota účelové funkce `ov` a informace o způsobu ukončení výpočtu `flag`. Připomeňme, že hodnota `flag` $= 1$ znamená, že bylo nalezeno optimum. Protože v našem případě nemůže dojít k neomezenému případu, hodnoty `flag` různé od jedné ztotožníme s nepřípustností (abstrahujeme zde od toho, že některé hodnoty `flag` mohou signalizovat numerické problémy).

B&B algoritmus. Během výpočtu budeme udržovat seznam $\mathfrak{L}$ lineárních programů. Seznam $\mathfrak{L}$ je na začátku prázdný.
Vstup: data $c, A, b, U, v, \underline{x}, \overline{x}$ problému (44).
Krok 1. Je-li LP (45) nepřípustný, skončíme — pak je totiž nepřípustný také ILP (44). Je-li LP (45) přípustný, vložíme jej do seznamu $\mathfrak{L}$.
Krok 2. Je-li seznam $\mathfrak{L}$ prázdný, skončíme — problém (44) je nepřípustný.
Krok 3. Je-li seznam $\mathfrak{L}$ neprázdný, nalezneme v něm ten lineární program $\{\min_{x\in\mathbb{R}} c^{\mathrm{T}}x \mid Ax \le b,\ Ux = v,\ \underline{x}^0 \le x \le \overline{x}^0\}$, který má mezi všemi LP v seznamu $\mathfrak{L}$ nejmenší optimální hodnotu účelové funkce. Říkejme mu (LP^0).
Krok 4. Odstraňme (LP^0) ze seznamu $\mathfrak{L}$.
Krok 5. Nechť x^* je optimální řešení (LP^0). Je-li x^* celočíselné, skončíme — nalezli jsme optimum (44).
Krok 6. Nechť i^0 je libovolný index takový, že $x^*_{i^0}$ není celočíselné. Zaveďme lineární programy

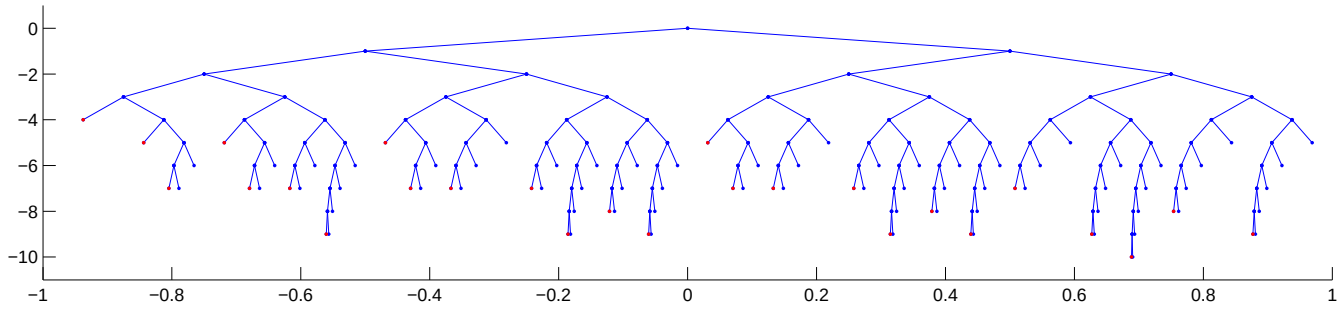
$$(LP') \quad \min_{x\in\mathbb{R}^n} c^{\mathrm{T}}x \text{ s.t. } Ax \le b,\ Ux = v,\ \underline{x}^0 \le x \le \overline{x}^0,\ x_{i^0} \ge \lceil x^*_{i^0} \rceil, \tag{46}$$

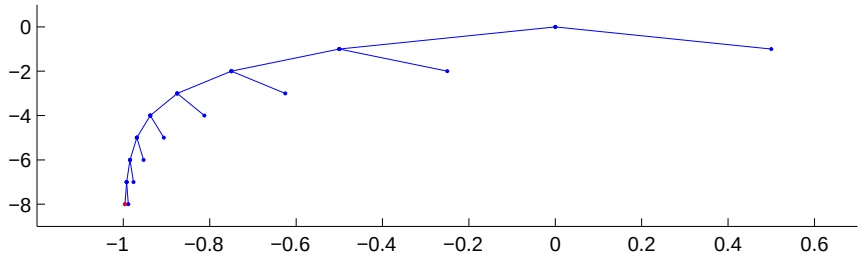
$$(LP'') \quad \min_{x\in\mathbb{R}^n} c^{\mathrm{T}}x \text{ s.t. } Ax \le b,\ Ux = v,\ \underline{x}^0 \le x \le \overline{x}^0,\ x_{i^0} \le \lfloor x^*_{i^0} \rfloor. \tag{47}$$

Zde $\lfloor \xi \rfloor = \max\{z \in \mathbb{Z} \mid z \le \xi\}$ značí dolní celou část čísla ξ (funkce `floor`) a $\lceil \xi \rceil = \min\{z \in \mathbb{Z} \mid z \ge \xi\}$ značí horní celou část čísla ξ (funkce `ceil`).
Krok 7. Jestliže je (LP') přípustný, přidejme jej do seznamu $\mathfrak{L}$.
Krok 8. Jestliže je (LP'') přípustný, přidejme jej do seznamu $\mathfrak{L}$.
Krok 9. Pokračujeme Krokem 2.

Poznámka 1. Konstrukci (46) a (47) lze udělat tak, že se jen zvýší dolní mez $\underline{x}^0_{i^0}$ na hodnotu $\lceil x^*_{i^0} \rceil$ (v případě (46)) a sníží se horní mez $\overline{x}^0_{i^0}$ na hodnotu $\lfloor x^*_{i^0} \rfloor$ (v případě (47)). To znamená, že všechny lineární programy v seznamu $\mathfrak{L}$ mají shodná data c, A, b, U, v a liší se jen v mezích $\underline{x}^0, \overline{x}^0$ (a pochopitelně také v optimálních řešeních x^* a optimálních hodnotách účelové funkce $c^{\mathrm{T}}x^*$). Proto můžeme reprezentovat LP v seznamu $\mathfrak{L}$ jen pomocí čtveřice údajů $(\underline{x}^0, \overline{x}^0, x^*, c^{\mathrm{T}}x^*)$. Konkrétně: uděláme to tak, že $\underline{x}^0$ budeme uchovávat jako sloupce speciální matice `LLB` („**L**ist of **L**ower **B**ounds“), $\overline{x}^0$ budeme uchovávat jako sloupce speciální matice `LUB` („**L**ist of **U**pper **B**ounds“), x^* budeme uchovávat jako sloupce speciální matice `Lx` a hodnoty účelové funkce $c^{\mathrm{T}}x^*$ budeme uchovávat jako prvky vektoru `Lov` („**L**ist of **o**ptimal **v**alues“).

Poznámka 2. Kroky 4 a 7–8 se nazývají *branching* (*větvení*) — nahrazujeme totiž (LP^0) dvojicí (LP'), (LP''). Protože v Kroku 1 začínáme s jediným LP v seznamu $\mathfrak{L}$, lze postupné větvení kreslit jako strom, tzv. *branching tree*. Ten si později ve skriptu také nakreslíme. Vrcholem ve stromě je (LP^0). Vrchol má dva syny, levého a pravého; je-li (LP') přípustný, je levým synem s modrou barvou, a je-li (LP') nepřípustný, je levým synem (a listem) s červenou barvou. Je-li (LP'') přípustný, je pravým synem s modrou barvou, a je-li (LP'') nepřípustný, je pravým synem (a listem) s červenou barvou. Kořenem je LP (45). Strom jistě může mít i modré listy — to jsou přípustné LP, které se ovšem dále nevětví (a jejichž zpracování tudíž B&B algoritmus „ušetřil“). Následující obrázky ilustrují dva příklady, jak konkrétně mohou stromy vypadat; druhý obrázek ukazuje příklad výpočtu, který je „velice úsporný“.





Poznámka 3. Výběr indexu i^0 (tzv. *branching index*) v Kroku 6 nemusí být jednoznačný. Skutečně, vektor x^* má často více neceločíselných složek. V principu můžeme vybrat libovolnou z nich. Metoda výběru i^0, je-li více možností, se nazývá *branching strategy* a různými volbami lze získat různé varianty B&B; dokonce stojí za to empiricky testovat, které strategie fungují „lépe“ a „hůře“ (to ponecháme jako cvičení). My vybereme i^0 takto: spočteme vektor $x' = x^* - \texttt{round}(x^*)$ (funkce `round` zaokrouhluje složky vektoru na nejbližší celé číslo). Vektor x' má tedy složky v intervalu $[\pm\frac{1}{2}]$ a nulové jsou právě ty složky, kde je x^* celočíselné. Pak spočteme $z = \max_i |x'_i|$. Je-li $z = 0$, je vektor x^* celočíselný; a je-li $z > 0$, za i^0 vezmeme ten index, pro nějž se maxima z nabývá. To je jistě legitimní branching strategie; nemá ovšem žádnou konkrétní výhodu, snad jen tu, že se snadno naprogramuje.

Poznámka 4. V numerických algoritmech bývají často výpočty přesné jen na určitý počet desetinných míst. Je-li správným výsledkem např. číslo 1, někdy můžeme od numerického algoritmu obdržet mírně odlišné číslo, např. 1 ± 10^{-15}. Abychom se nemuseli zabývat numerickými problémy, učiníme následující zjednodušení: zvolíme dostatečně malou chybu, řekněme 10^{-8}, a numericky spočtené číslo prohlásíme za celé, jestliže se od skutečně celého čísla liší nanejvýš o $\pm 10^{-8}$.

Ve skriptu budeme v proměnné `q` sledovat počet řešených LP — to jen kvůli informaci, kolik práce výpočet obnáší.

```
function xopt = MyBaB(c,A,b,U,v,LB,UB)
  % argumenty mají stejný význam jako u linprog

  % KROK 1: test přípustnosti LP (45)
  [x0, ov, flag] = linprog(c,A,b,U,v,LB,UB);
  if flag ~= 1             % LP (45) je nepřípustný
    disp('infeasible!');
    return;                % skončit
  end

  % KROK 1: do seznamu L vkládáme LP (45), viz také Poznámku 1
  LLB = LB; LUB = UB; Lx = x0; Lov = ov;

  n = 1;  % n = počet prvků seznamu L
  q = 1;  % počitadlo LP --- zatím jsme vyřešili jeden LP

  % HLAVNÍ CYKLUS
  while n >= 1             % dokud je seznam L neprázdný
    [fmin, j] = min(Lov);  % KROK 3: j je index LP v seznamu L s nejmenší
                           % účelovou hodnotou --- to je LP^0
    LB0 = LLB(:,j); UB0 = LUB(:,j); xstar = Lx(:,j);
                           % LB0, UB0, xstar jsou údaje o LP^0
    [z, i0] = max(abs(xstar - round(xstar)));
                           % KROK 6: i0 = branching index (viz též Poznámku 3)
    % KROK 5
    if z <= 10^-8          % Je xstar celočíselné? (viz též Poznámku 4)
      xopt = round(xstar); % výstupem je xopt
      disp(['Optimal objective value: ',num2str(fmin)]);
      disp(['LPs solved: ',num2str(q)]);
      return;              % skončit
    end;

    % KROK 4: Vyřadíme LP^0 ze seznamu L
    LLB = LLB(:,[1:j-1,j+1:n]);
    LUB = LUB(:,[1:j-1,j+1:n]);
    Lx  = Lx(:,[1:j-1,j+1:n]);
    Lov = Lov([1:j-1,j+1:n]);

    % KROK 6: Zkonstruujeme a vyřešíme (LP'), (LP''):
    UB1 = UB0; UB1(i0) = floor(xstar(i0));
    LB1 = LB0; LB1(i0) = ceil(xstar(i0));
    [x1, ov1, flag1] = linprog(c,A,b,U,v,LB1,UB0);
    [x2, ov2, flag2] = linprog(c,A,b,U,v,LB0,UB1);

    q = q+2;               % počitadlo --- vyřešili jsme dva LP

    % KROK 7
    if flag1 == 1                          % (LP') je přípustný
      LLB = [LLB, LB1]; LUB = [LUB, UB0];  % (LP') vkládáme do seznamu L
      Lx  = [Lx, x1];   Lov = [Lov; ov1];  % (LP') vkládáme do seznamu L
    end

    % KROK 8
    if flag2 == 1                          % (LP'') je přípustný
      LLB = [LLB, LB0]; LUB = [LUB, UB1];  % (LP'') vkládáme do seznamu L
      Lx  = [Lx, x2];   Lov = [Lov; ov2];  % (LP'') vkládáme do seznamu L
    end

    n = length(Lov);  % n = počet prvků seznamu L
  end                 % konec hlavního while-cyklu

  % KROK 2: seznam L je prázdný
  disp('infeasible!');
end
```

Použijme jednoduchý testovací celočíselný program — tzv. *problém batohu* s daty $c \in \mathbb{R}^n$ a $w \in \mathbb{R}$

$$\max_{x \in \{0,1\}^n} \; c^{\mathrm{T}} x \;\text{ s.t. }\; c^{\mathrm{T}} x \leq w. \tag{48}$$

Jde o tento problém: je dáno n kontejnerů s hmotnostmi $c_1, \ldots, c_n$ a loď s nosností w (v tomto kontextu se lodi někdy říká „batoh“ — odtud název problému). Cílem je určit, které kontejnery máme na loď naložit, aby naložená hmotnost byla co nejvyšší, ale přitom nepřesáhla nosnost w.

Vezměme pro příklad $n = 10$. Nechť $c_1, \ldots, c_{10}$ jsou náhodné hmotnosti z rovnoměrného rozdělení na intervalu $[0, 300]$

```
n = 10;  c = random('unif',0,300,[n,1]);
```

a zvolme $w = 1000$. Vyřešíme (48) pomocí našeho B&B-solveru:

```
x = MyBaB(-c, c', 1000, [], [], zeros(n,1), ones(n,1))
```

Za dolní a horní meze vkládáme vektory ze samých nul a jedniček. Systém omezení neobsahuje žádné rovnosti; proto je čtvrtý a pátý argument prázdný. Účelovou funkci píšeme s minusem, neboť jde o maximalizační problém, zatímco solver `MyBaB` pracuje s minimalizací.

Konkrétní výsledek samozřejmě závisí na hodnotách náhodných hmotností c; výsledek může vypadat například takto (jen výsledné optimální řešení `x` zde pro úsporu místa uvádíme jako řádkový vektor, byť skript vrací vektor sloupcový):

```
Optimal objective value: -999.9542
LPs solved: 185
x = 1 0 1 0 1 1 1 0 0 0
```

Abychom ilustrovali, že náš solver skutečně *není* konkurenceschopný vůči profesionálnímu softwaru typu CPLEX, změřme výpočetní čas: funkce `tic` zapíná stopky a funkce `toc` vypíná stopky. Zadáme-li

```
tic;  x = MyBaB(-c, c', 1000, [], [], zeros(n,1), ones(n,1));  toc;
```

obdržíme informaci typu

```
Elapsed time is 1.988427 seconds.
```

To je dosti špatný výsledek pro problém batohu s deseti proměnnými!

Vizualizace B&B stromu. Nyní funkci `MyBaB` rozšíříme o dodatečné instrukce, které nakreslí strom větvení výpočtu ze strany 26. Kořen stromu nakreslíme jako bod v rovině se souřadnicemi $(x, y) = (0, 0)$. K němu si také připojíme údaj $d = 1$; údaj d řekne, že synové mají být nakresleni na souřadnicích $(x - \frac{d}{2}, y - 1)$ (levý syn) a $(x + \frac{d}{2}, y - 1)$ (pravý syn). Sestoupíme-li ve stromě o patro níže, bude třeba vzdálenost synů d zkrátit na polovinu (jak je patrné z obrázku ze strany 26).

Trojici údajů (x, y, d) budeme chápat jako dodatečnou informaci o lineárním programu v seznamu $\mathfrak{L}$. Budeme tuto trojici ukládat jako sloupcový vektor v pomocné matici `Lpl` („`pl`“ znamená: „data pro `plot`“).

Funkci `MyBaB` rozšíříme o dodatečné instrukce uvedené červeně. Zbytek programu zůstává nezměněn; jen pro stručnost neopakujeme všechny komentáře. Říkejme této rozšířené verzi solveru `MyBaBPlotTree`. Zavoláme-li jej opět na problém batohu

```
n = 10;  c = random('unif', 0, 300, [n,1]);
x = MyBaBPlotTree(-c, c', 1000, [], [], zeros(n,1), ones(n,1))
```

obdržíme navíc obrázek podobného typu jako na straně 26.

```
function xopt = MyBaBPlotTree(c,A,b,U,v,LB,UB)

  % KROK 1: test přípustnosti LP (45)
  [x0, ov, flag] = linprog(c,A,b,U,v,LB,UB);
  if flag ~= 1
    disp('infeasible!');
    return;
  end

  % KROK 1: do seznamu L vkládáme LP (45)
  LLB = LB; LUB = UB; Lx = x0; Lov = ov;
  Lpl = [0;0;1];  % kořen vykreslíme na souřadnice (0,0) a vzdálenost
                  % jeho synů má být d = 1;
  n = 1;
  q = 1;

  % HLAVNÍ CYKLUS
  while n >= 1
    [fmin, j] = min(Lov);                              % KROK 3: výběr (LP^0)
    LB0 = LLB(:,j); UB0 = LUB(:,j); xstar = Lx(:,j);
    [z, i0] = max(abs(xstar - round(xstar)));          % KROK 6: volba branching indexu i0

    % KROK 5
    if z <= 10^-8
      xopt = round(xstar);
      disp(['Optimal objective value: ',num2str(fmin)]);
      disp(['LPs solved: ',num2str(q)]);
      return;
    end;

    px = Lpl(1,j); py = Lpl(2,j); pd = Lpl(3,j);
       % ze seznamu Lpl čteme rovinné souřadnice aktuálního vrcholu a hodnotu d

    % KROK 4: Vyřadíme LP^0 ze seznamu L
    Lpl = Lpl(:,[1:j-1,j+1:n]);   % aktuální vrchol vyřadit ze seznamu Lpl
    LLB = LLB(:,[1:j-1,j+1:n]);
    LUB = LUB(:,[1:j-1,j+1:n]);
    Lx  = Lx(:,[1:j-1,j+1:n]);
    Lov = Lov([1:j-1,j+1:n]);

    % KROK 6: Zkonstruujeme a vyřešíme (LP'), (LP''):
    UB1 = UB0; UB1(i0) = floor(xstar(i0));
    LB1 = LB0; LB1(i0) = ceil(xstar(i0));
    [x1, ov1, flag1] = linprog(c,A,b,U,v,LB1,UB0);
    [x2, ov2, flag2] = linprog(c,A,b,U,v,LB0,UB1);
    q = q+2;

    plot([px,px-pd/2],[py,py-1],'b.-', [px,px+pd/2],[py,py-1],'b.-');
       % k bodu se souřadnicemi (x,y) vykreslíme oba syny (x-d/2, y-1) a (x+d/2, y-1)

    % KROK 7
    if flag1 == 1
      LLB = [LLB, LB1]; LUB = [LUB, UB0];
      Lx  = [Lx, x1];   Lov = [Lov; ov1];
      Lpl = [Lpl, [px-pd/2; py-1; pd/2]];
         % vlož (x,y,d) pro (LP') do seznamu LPl; vzdálenost synů se zmenší na polovinu
    else
      plot(px-pd/2,py-1,'.r');   % je-li (LP') nepřípustný, vykresli vrchol červeně
    end

    % KROK 8
    if flag2 == 1
      LLB = [LLB, LB0]; LUB = [LUB, UB1];
      Lx  = [Lx, x2];   Lov = [Lov; ov2];
      Lpl = [Lpl, [px+pd/2; py-1; pd/2]];
         % vlož (x,y,d) pro (LP'') do seznamu LPl; vzdálenost synů se zmenší na polovinu
    else
      plot(px+pd/2,py-1,'.r');   % je-li (LP'') nepřípustný, vykresli vrchol červeně
    end

    n = length(Lov);
  end

  % KROK 2: seznam L je prázdný
  disp('infeasible!');
end
```