Algoritmy a ich zložitosť

Stanislav Palúch

Fakulta riadenia a informatiky, Žilinská univerzita

11. apríla 2011

Algoritmy

Pod pojmom **algoritmus** rozumieme postupnosť krokov, ktorá nás dovedie k žiadanému riešeniu daného problému.
Žiada sa, aby algoritmus mal tieto vlastnosti:

- determinovanosť – má byť zadaný konečným počtom jednoznačných pravidiel
- efektívnosť – má zaručiť vyriešenie úlohy po konečnom počte krokov
- hromadnosť – má byť použiteľný na celú triedu prípadov úlohy svojho typu

Pravidlá v algoritme musia byť rozhodnuteľné v okamihu výpočtu.

Príklad nekorektného pravidla (Plesník):
„Ak existujú mimozemské civilizácie, tak choď domov, inak zostaň tu.“

Výpočtová náročnosť algoritmov

Ukázalo sa rozumné posudzovať algoritmy podľa počtu elementárnych krokov, ktoré potrebuje na vyriešenie daného problému. Tieto elementárne kroky môžu byť

- sčítanie
- odčítanie
- násobenie
- delenie
- porovnávanie s vetvením
- atď

Jednotlivé elementárne kroky považujeme za rovnako časovo náročné.

Budeme hovoriť, že

algoritmus vyrieši danú konkrétnu úlohu U v čase T,

ak na jej vyriešenie potrebuje T elementárnych krokov.

Výpočtová náročnosť algoritmov

Pre ohodnotenie výpočtovej zložitosti algoritmu nás však viac ako jeden konkrétny prípad zaujíma **závislosť počtu elementárnych krokov algoritmu $T(n)$ na veľkosti resp. rozsahu počítanej úlohy n.**

Dĺžka úlohy – množstvo vstupných dát príslušnej úlohy.

Príklad

Pre graf $G = (V, H)$ alebo digraf $\overrightarrow{G} = (V, H)$ s n vrcholmi a m hranami, t.j. kde $|V| = n$, $|V| = m$, bude dĺžka úlohy $(m + n)$.

Niekedy sa udáva len závislosť času výpočtu len na počte vrcholov n, pričom sa berie do úvahy, že $m \leq \dfrac{n(n-1)}{2} \leq n^2$ pre grafy, resp. $m \leq n(n-1) \leq n^2$ pre digrafy.

Počet krokov algoritmu môže závisieť nielen od množstva vstupných dát, ale aj od ich vzájomnej konfigurácie.

Preto funkcia $T(n)$ môže vyjadrovať iba **hornú hranicu počtu krokov algoritmu pre najhorší prípad úlohy s dĺžkou** n (worst case analysis).

Výpočtová náročnosť algoritmov

Pre ohodnotenie výpočtovej zložitosti algoritmu nás však viac ako jeden konkrétny prípad zaujíma **závislosť počtu elementárnych krokov algoritmu $T(n)$ na veľkosti resp. rozsahu počítanej úlohy n.**

Dĺžka úlohy – množstvo vstupných dát príslušnej úlohy.

Príklad

Pre graf $G = (V, H)$ alebo digraf $\overrightarrow{G} = (V, H)$ s n vrcholmi a m hranami, t.j. kde $|V| = n$, $|V| = m$, bude dĺžka úlohy $(m + n)$.

Niekedy sa udáva len závislosť času výpočtu len na počte vrcholov n, pričom sa berie do úvahy, že $m \leq \dfrac{n(n-1)}{2} \leq n^2$ pre grafy, resp. $m \leq n(n-1) \leq n^2$ pre digrafy.

Počet krokov algoritmu môže závisieť nielen od množstva vstupných dát, ale aj od ich vzájomnej konfigurácie.

Preto funkcia $T(n)$ môže vyjadrovať iba **hornú hranicu počtu krokov algoritmu pre najhorší prípad úlohy s dĺžkou n** (worst case analysis).

Výpočtová náročnosť algoritmov

Pre ohodnotenie výpočtovej zložitosti algoritmu nás však viac ako jeden konkrétny prípad zaujíma **závislosť počtu elementárnych krokov algoritmu $T(n)$ na veľkosti resp. rozsahu počítanej úlohy n.**

Dĺžka úlohy – množstvo vstupných dát príslušnej úlohy.

Príklad

Pre graf $G = (V, H)$ alebo digraf $\overrightarrow{G} = (V, H)$ s n vrcholmi a m hranami, t.j. kde $|V| = n$, $|V| = m$, bude dĺžka úlohy $(m + n)$.

Niekedy sa udáva len závislosť času výpočtu len na počte vrcholov n, pričom sa berie do úvahy, že $m \leq \dfrac{n(n-1)}{2} \leq n^2$ pre grafy, resp. $m \leq n(n-1) \leq n^2$ pre digrafy.

Počet krokov algoritmu môže závisieť nielen od množstva vstupných dát, ale aj od ich vzájomnej konfigurácie.

Preto funkcia $T(n)$ môže vyjadrovať iba **hornú hranicu počtu krokov algoritmu pre najhorší prípad úlohy s dĺžkou n** (worst case analysis).

Výpočtová náročnosť algoritmov

Pre ohodnotenie výpočtovej zložitosti algoritmu nás však viac ako jeden konkrétny prípad zaujíma **závislosť počtu elementárnych krokov algoritmu $T(n)$ na veľkosti resp. rozsahu počítanej úlohy n.**

Dĺžka úlohy – množstvo vstupných dát príslušnej úlohy.

Príklad

Pre graf $G = (V, H)$ alebo digraf $\overrightarrow{G} = (V, H)$ s n vrcholmi a m hranami, t.j. kde $|V| = n$, $|V| = m$, bude dĺžka úlohy $(m + n)$.

Niekedy sa udáva len závislosť času výpočtu len na počte vrcholov n, pričom sa berie do úvahy, že $m \leq \dfrac{n(n-1)}{2} \leq n^2$ pre grafy, resp. $m \leq n(n-1) \leq n^2$ pre digrafy.

Počet krokov algoritmu môže závisieť nielen od množstva vstupných dát, ale aj od ich vzájomnej konfigurácie.

Preto funkcia $T(n)$ môže vyjadrovať iba **hornú hranicu počtu krokov algoritmu pre najhorší prípad úlohy s dĺžkou n** (worst case analysis).

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1
6 7 5 4 3 2 1
6 5 7 4 3 2 1
6 5 4 7 3 2 1 $(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1
6 5 4 3 2 7 1
6 5 4 3 2 1 7

---

6 5 4 3 2 1 7
5 6 4 3 2 1 7
. . . . $(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7

---

5 4 3 2 1 6 7
. . . . $(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7

. . . .
. . . .

---

2 1 3 4 5 6 7
1 2 3 4 5 6 7 $(n-1)$ vykonaní príkazu **if**

---

1 2 3 4 5 6 7 $(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
....	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
....	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
....	
....	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7	6	5	4	3	2	1	
6	7	5	4	3	2	1	
6	5	7	4	3	2	1	
6	5	4	7	3	2	1	$(n-1)$ vykonaní príkazu **if**
6	5	4	3	7	2	1	
6	5	4	3	2	7	1	
6	5	4	3	2	1	7	
6	5	4	3	2	1	7	
5	6	4	3	2	1	7	
…							$(n-1)$ vykonaní príkazu **if**
5	4	3	2	1	6	7	
5	4	3	2	1	6	7	
…							$(n-1)$ vykonaní príkazu **if**
4	3	2	1	5	6	7	
…							
…							
2	1	3	4	5	6	7	
1	2	3	4	5	6	7	$(n-1)$ vykonaní príkazu **if**
1	2	3	4	5	6	7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7	6	5	4	3	2	1	
6	7	5	4	3	2	1	
6	5	7	4	3	2	1	
6	5	4	7	3	2	1	$(n-1)$ vykonaní príkazu **if**
6	5	4	3	7	2	1	
6	5	4	3	2	7	1	
6	5	4	3	2	1	7	
6	5	4	3	2	1	7	
5	6	4	3	2	1	7	
. . .							$(n-1)$ vykonaní príkazu **if**
5	4	3	2	1	6	7	
5	4	3	2	1	6	7	
. . .							$(n-1)$ vykonaní príkazu **if**
4	3	2	1	5	6	7	
. . .							
. . .							
2	1	3	4	5	6	7	
1	2	3	4	5	6	7	$(n-1)$ vykonaní príkazu **if**
1	2	3	4	5	6	7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7	6	5	4	3	2	1	
6	7	5	4	3	2	1	
6	5	7	4	3	2	1	
6	5	4	7	3	2	1	$(n-1)$ vykonaní príkazu **if**
6	5	4	3	7	2	1	
6	5	4	3	2	7	1	
6	5	4	3	2	1	7	
6	5	4	3	2	1	7	
5	6	4	3	2	1	7	
...							$(n-1)$ vykonaní príkazu **if**
5	4	3	2	1	6	7	
5	4	3	2	1	6	7	
...							$(n-1)$ vykonaní príkazu **if**
4	3	2	1	5	6	7	
...							
...							
2	1	3	4	5	6	7	
1	2	3	4	5	6	7	$(n-1)$ vykonaní príkazu **if**
1	2	3	4	5	6	7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
. . . .	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
. . . .	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
. . . .	
. . . .	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
. . . .	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
. . . .	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
. . . .	
. . . .	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
. . . .	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
. . . .	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
. . . .	
. . . .	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
. . . .	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
. . . .	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
. . . .	
. . . .	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1
6 7 5 4 3 2 1
6 5 7 4 3 2 1
6 5 4 7 3 2 1 $(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1
6 5 4 3 2 7 1
6 5 4 3 2 1 7

---

6 5 4 3 2 1 7
5 6 4 3 2 1 7
. . . . $(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7

---

5 4 3 2 1 6 7
. . . . $(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7

. . . .
. . . .

---

2 1 3 4 5 6 7
1 2 3 4 5 6 7 $(n-1)$ vykonaní príkazu **if**

---

1 2 3 4 5 6 7 $(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1
6 7 5 4 3 2 1
6 5 7 4 3 2 1
6 5 4 7 3 2 1 $(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1
6 5 4 3 2 7 1
6 5 4 3 2 1 7

---

6 5 4 3 2 1 7
5 6 4 3 2 1 7
.... $(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7

---

5 4 3 2 1 6 7
.... $(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7

....
....

---

2 1 3 4 5 6 7
1 2 3 4 5 6 7 $(n-1)$ vykonaní príkazu **if**

---

1 2 3 4 5 6 7 $(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7	6	5	4	3	2	1	
6	7	5	4	3	2	1	
6	5	7	4	3	2	1	
6	5	4	7	3	2	1	$(n-1)$ vykonaní príkazu **if**
6	5	4	3	7	2	1	
6	5	4	3	2	7	1	
6	5	4	3	2	1	7	
6	5	4	3	2	1	7	
5	6	4	3	2	1	7	
...							$(n-1)$ vykonaní príkazu **if**
5	4	3	2	1	6	7	
5	4	3	2	1	6	7	
...							$(n-1)$ vykonaní príkazu **if**
4	3	2	1	5	6	7	
...							
...							
2	1	3	4	5	6	7	
1	2	3	4	5	6	7	$(n-1)$ vykonaní príkazu **if**
1	2	3	4	5	6	7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
     {sorted=0;
      swap(a[i],a[i+1]);
      }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
....	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
....	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
....	
....	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
     {sorted=0;
      swap(a[i],a[i+1]);
      }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
....	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
....	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
....	
....	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
     {sorted=0;
      swap(a[i],a[i+1]);
      }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
....	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
....	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
....	
....	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
     {sorted=0;
      swap(a[i],a[i+1]);
      }
   }
}
```

7 6 5 4 3 2 1
6 7 5 4 3 2 1
6 5 7 4 3 2 1
6 5 4 7 3 2 1 $(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1
6 5 4 3 2 7 1
6 5 4 3 2 1 7

---

6 5 4 3 2 1 7
5 6 4 3 2 1 7
. . . . $(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7

---

5 4 3 2 1 6 7
. . . . $(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7

. . . .
. . . .

---

2 1 3 4 5 6 7
1 2 3 4 5 6 7 $(n-1)$ vykonaní príkazu **if**

---

1 2 3 4 5 6 7 $(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
     {sorted=0;
      swap(a[i],a[i+1]);
      }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
....	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
....	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
....	
....	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Príklad – usporadanie postupnosti n prvkov vzostupne

```
/* algoritmus bsort */
sorted=0;
while(sorted==0)
{sorted=1;
for (i=1; i<n; i++)
   {if(a[i]> a[i+1])
      {sorted=0;
       swap(a[i],a[i+1]);
       }
   }
}
```

7 6 5 4 3 2 1	
6 7 5 4 3 2 1	
6 5 7 4 3 2 1	
6 5 4 7 3 2 1	$(n-1)$ vykonaní príkazu **if**
6 5 4 3 7 2 1	
6 5 4 3 2 7 1	
6 5 4 3 2 1 7	
6 5 4 3 2 1 7	
5 6 4 3 2 1 7	
....	$(n-1)$ vykonaní príkazu **if**
5 4 3 2 1 6 7	
5 4 3 2 1 6 7	
....	$(n-1)$ vykonaní príkazu **if**
4 3 2 1 5 6 7	
....	
....	
2 1 3 4 5 6 7	
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**
1 2 3 4 5 6 7	$(n-1)$ vykonaní príkazu **if**

V prípade zostupne zotriedenej postupnosti na vstupe algoritmus vykoná $n(n-1)$ krát príkaz **if**, v prípade vzostupne zotriedenej postupnosti na vstupe algoritmus vykoná tento príkaz len $(n-1)$ krát.

Definícia

Nech $g(n)$, $h(n)$ sú dve kladné funkcie definované na množine prirodzených čísel. Budeme písať $g(n) = O(h(n))$ a hovoriť, že **funkcia $h(n)$ asymptoticky dominuje funkcii** *$g(n)$, ak existuje konštanta K a prirodzené číslo n_0 také, že*

$$g(n) \leq K.h(n) \quad \forall n \geq n_0.$$

Definícia

Hovoríme, že **algoritmus $\mathcal{A}$ má zložitosť** *$O(f(n))$, ak pre horný odhad $T(n)$ počtu krokov algoritmu $\mathcal{A}$ pre úlohu dĺžky n platí*

$$T(n) = O(f(n)).$$

Skrátene hovoríme o $O(f(n))$ algoritme.

Špeciálne ak $f(n) \leq n^k$ pre nejaké konštantné k, hovoríme, že $\mathcal{A}$ je **polynomiálny algoritmus**.

Definícia

Nech $g(n)$, $h(n)$ sú dve kladné funkcie definované na množine prirodzených čísel. Budeme písať $g(n) = O(h(n))$ a hovoriť, že **funkcia $h(n)$ asymptoticky dominuje funkcii** *$g(n)$, ak existuje konštanta K a prirodzené číslo n_0 také, že*

$$g(n) \leq K.h(n) \quad \forall n \geq n_0.$$

Definícia

Hovoríme, že **algoritmus $\mathcal{A}$ má zložitosť** *$O(f(n))$, ak pre horný odhad $T(n)$ počtu krokov algoritmu $\mathcal{A}$ pre úlohu dĺžky n platí*

$$T(n) = O(f(n)).$$

Skrátene hovoríme o $O(f(n))$ algoritme.

Špeciálne ak $f(n) \leq n^k$ pre nejaké konštantné k, hovoríme, že $\mathcal{A}$ je **polynomiálny algoritmus**.

Príklady polynomiálnych algolritmov

Príklad

Triediaci algoritmus bsort vykoná v najhoršom prípade $n(n-1)$ príkazov **if**, *príkaz* **if** *vykoná nanajvýš tri elementárne operácie (porovnanie, priradenie a swap) takže počet všetkých operácií možno zhora ohraničiť číslom*
$n(n-1)K \leq K.n^2$ *pre* $K = 3$.
Algoritmus bsort je $O(n^2)$ *algoritmus.*

Príklad

Floydov algoritmus:

for`(k=1; k <= n; k++)`
  for`(i=1; i <= n; i++)`
    for`(j=1; j <= n; j++)`
      if$(c[i,j] > c[i,k] + c[k,j])$
      $\{\ c[i,j] = c[i,k] + c[k,j];$
        $x[i,j] = x[k,j];\ \}$

Počet elementárnych operácií za príkazom **if** *najvnútornejšieho cyklu možno zhora ohraničiť istým číslom* K. *Tento príkaz sa vykoná* n^3 *krát. Počet všetkých elementárnych krokov Floydovho algoritmu možno zhora ohraničiť číslom* Kn^3.
Floydov algoritmus je $O(n^3)$ *algoritmus.*

Príklad

Triediaci algoritmus bsort vykoná v najhoršom prípade $n(n-1)$ príkazov **if**, *príkaz* **if** *vykoná nanajvýš tri elementárne operácie (porovnanie, priradenie a swap) takže počet všetkých operácií možno zhora ohraničiť číslom* $n(n-1)K \leq K.n^2$ *pre* $K = 3$.
Algoritmus bsort je $O(n^2)$ *algoritmus.*

Príklad

Floydov algoritmus:

```
for(k=1; k <= n; k++)
   for(i=1; i <= n; i++)
      for(j=1; j <= n; j++)
         if(c[i,j] > c[i,k] + c[k,j])
         {  c[i,j] = c[i,k] + c[k,j];
            x[i,j] = x[k,j]; }
```

Počet elementárnych operácií za príkazom **if** *najvnútornejšieho cyklu možno zhora ohraničiť istým číslom* K. *Tento príkaz sa vykoná* n^3 *krát. Počet všetkých elementárnych krokov Floydovho algoritmu možno zhora ohraničiť číslom* Kn^3.
Floydov algoritmus je $O(n^3)$ *algoritmus.*

Zložitosť problému

Definícia

Hovoríme, že **problém má zložitosť nanajvýš** $O(f(n))$, *ak preň existuje* $O(f(n))$ *algoritmus.*

Príklad

Pre problém triedenia n-prvkovej postupnosti sme uviedli algoritmus bsort so zložitosťou $O(n^2)$.
Na základe toho bu bolo možné konštatovať, že problém triedenia má zložitosť nanajvýš $O(n^2)$.

Existujú však lepšie algoritmy so zložitosťou $O(n.\log n)$.

Dá sa ukázať, že pre problém triedenia algoritmy s menšou zložitosťou než $O(n.\log n)$ *neexistujú.*

Problém triedenia n čísel má teda zložitosť $O(n.\log n)$.

Zložitosť problému

Definícia

Hovoríme, že **problém má zložitosť nanajvýš** $O(f(n))$*, ak preň existuje* $O(f(n))$ *algoritmus.*

Príklad

Pre problém triedenia n-prvkovej postupnosti sme uviedli algoritmus bsort so zložitosťou $O(n^2)$*.*
Na základe toho bu bolo možné konštatovať, že problém triedenia má zložitosť nanajvýš $O(n^2)$*.*

Existujú však lepšie algoritmy so zložitosťou $O(n.\log n)$*.*

Dá sa ukázať, že pre problém triedenia algoritmy s menšou zložitosťou než $O(n.\log n)$ *neexistujú.*

Problém triedenia n čísel má teda zložitosť $O(n.\log n)$*.*

Zložitosť problému

Definícia

Hovoríme, že **problém má zložitosť nanajvýš** $O(f(n))$*, ak preň existuje* $O(f(n))$ *algoritmus.*

Príklad

Pre problém triedenia n-prvkovej postupnosti sme uviedli algoritmus bsort so zložitosťou $O(n^2)$.
Na základe toho bu bolo možné konštatovať, že problém triedenia má zložitosť nanajvýš $O(n^2)$.

Existujú však lepšie algoritmy so zložitosťou $O(n. \log n)$.

Dá sa ukázať, že pre problém triedenia algoritmy s menšou zložitosťou než $O(n. \log n)$ *neexistujú.*

Problém triedenia n čísel má teda zložitosť $O(n. \log n)$.

Sústava lineárnych rovníc

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned} \tag{1}$$

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots\dots\dots & & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} . \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{pmatrix} \tag{2}$$

$$\mathbf{A}.\mathbf{x} = \mathbf{b}$$

$\mathbf{A} = (a_{ij})$ je matica ľavej strany sústavy (1),
$\mathbf{b}$ je stĺpcový vektor pravých strán $\mathbf{b} = (b_1, b_2, \dots, b_m)^T$ a
$\mathbf{x} = (x_1, x_2, \dots, x_n)^T$ stĺpcový vektor neznámych, $\mathbf{x} \in \mathbb{R}^n$.

Sústava lineárnych rovníc

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots & \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m \end{aligned} \tag{1}$$

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots\dots\dots & & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} . \begin{pmatrix} x_1 \\ x_2 \\ .. \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{pmatrix} \tag{2}$$

$$\mathbf{A}.\mathbf{x} = \mathbf{b}$$

$\mathbf{A} = (a_{ij})$ je matica ľavej strany sústavy (1),
$\mathbf{b}$ je stĺpcový vektor pravých strán $\mathbf{b} = (b_1, b_2, \dots, b_m)^T$ a
$\mathbf{x} = (x_1, x_2, \dots, x_n)^T$ stĺpcový vektor neznámych, $\mathbf{x} \in \mathbb{R}^n$.

Sústava lineárnych rovníc

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots & \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned} \tag{1}$$

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots\dots\dots & & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} . \begin{pmatrix} x_1 \\ x_2 \\ .. \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{pmatrix} \tag{2}$$

$$\mathbf{A}.\mathbf{x} = \mathbf{b}$$

$\mathbf{A} = (a_{ij})$ je matica ľavej strany sústavy (1),
$\mathbf{b}$ je stĺpcový vektor pravých strán $\mathbf{b} = (b_1, b_2, \dots, b_m)^T$ a
$\mathbf{x} = (x_1, x_2, \dots, x_n)^T$ stĺpcový vektor neznámych, $\mathbf{x} \in \mathbb{R}^n$.

Sústava lineárnych rovníc

Platí

- Sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie práve vtedy, keď hodnosť matice $\mathbf{A}$ ľavej strany sa rovná hodnosti rozšírenej matice $(\mathbf{A}|\mathbf{b})$ sústavy.
- každé riešenie sústavy $\mathbf{A}.\mathbf{x} = \mathbf{b}$ dostaneme ako súčet jedného pevne vzatého riešenia tejto sústavy a niektorého riešenia sústavy s nulovou pravou stranou – t.j sústavy $\mathbf{A}.\mathbf{x} = \mathbf{o}$.
- Ak $m < n$ a sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie, potom má nekonečne veľa riešení.
- V mnohých praktických úlohách premenné $x_1, x_2, \dots, x_n$ predstavujú množstvá reálnych látok, substrátov alebo objektov a sústava rovníc $\mathbf{A}.\mathbf{x} = \mathbf{b}$ predstavuje ohraničenia pre tieto množstvá, preto nás zaujímajú len také riešenia tejto sústavy, pre ktoré je $x_1 \geq 0, \ x_2 \geq 0, \ \dots, \ x_n \geq 0$, t. j. pre ktoré sú množstvá nezáporné.
- Každému nezápornému riešeniu je priradená hodnota lineárnej kriteriálnej funkcie
$$f(\mathbf{x}) = c_1x_1 + c_2x_2 + \dots + c_nx_n = \mathbf{c}^T.\mathbf{x}$$
predstavujúca náklady na riešenie $\mathbf{x}$.
- Zo všetkých nezáporných riešení nás zaujíma to, ktoré nám prináša čo najmenšie náklady.

Sústava lineárnych rovníc

Platí

- Sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie práve vtedy, keď hodnosť matice $\mathbf{A}$ ľavej strany sa rovná hodnosti rozšírenej matice $(\mathbf{A}|\mathbf{b})$ sústavy.
- každé riešenie sústavy $\mathbf{A}.\mathbf{x} = \mathbf{b}$ dostaneme ako súčet jedného pevne vzatého riešenia tejto sústavy a niektorého riešenia sústavy s nulovou pravou stranou – t.j sústavy $\mathbf{A}.\mathbf{x} = \mathbf{o}$.
- Ak $m < n$ a sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie, potom má nekonečne veľa riešení.
- V mnohých praktických úlohách premenné $x_1, x_2, \dots, x_n$ predstavujú množstvá reálnych látok, substrátov alebo objektov a sústava rovníc $\mathbf{A}.\mathbf{x} = \mathbf{b}$ predstavuje ohraničenia pre tieto množstvá, preto nás zaujímajú len také riešenia tejto sústavy, pre ktoré je $x_1 \geq 0,\ x_2 \geq 0,\ \dots,\ x_n \geq 0$, t. j. pre ktoré sú množstvá nezáporné.
- Každému nezápornému riešeniu je priradená hodnota lineárnej kriteriálnej funkcie

$$f(\mathbf{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n = \mathbf{c}^T.\mathbf{x}$$

  predstavujúca náklady na riešenie $\mathbf{x}$.
- Zo všetkých nezáporných riešení nás zaujíma to, ktoré nám prináša čo najmenšie náklady.

Sústava lineárnych rovníc

Platí

- Sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie práve vtedy, keď hodnosť matice $\mathbf{A}$ ľavej strany sa rovná hodnosti rozšírenej matice $(\mathbf{A}|\mathbf{b})$ sústavy.
- každé riešenie sústavy $\mathbf{A}.\mathbf{x} = \mathbf{b}$ dostaneme ako súčet jedného pevne vzatého riešenia tejto sústavy a niektorého riešenia sústavy s nulovou pravou stranou – t.j sústavy $\mathbf{A}.\mathbf{x} = \mathbf{o}$.
- Ak $m < n$ a sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie, potom má nekonečne veľa riešení.
- V mnohých praktických úlohách premenné $x_1, x_2, \ldots, x_n$ predstavujú množstvá reálnych látok, substrátov alebo objektov a sústava rovníc $\mathbf{A}.\mathbf{x} = \mathbf{b}$ predstavuje ohraničenia pre tieto množstvá, preto nás zaujímajú len také riešenia tejto sústavy, pre ktoré je $x_1 \geq 0,\ x_2 \geq 0,\ \ldots,\ x_n \geq 0$, t. j. pre ktoré sú množstvá nezáporné.
- Každému nezápornému riešeniu je priradená hodnota lineárnej kriteriálnej funkcie
$$f(\mathbf{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n = \mathbf{c}^T.\mathbf{x}$$
predstavujúca náklady na riešenie $\mathbf{x}$.
- Zo všetkých nezáporných riešení nás zaujíma to, ktoré nám prináša čo najmenšie náklady.

Sústava lineárnych rovníc

Platí

- Sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie práve vtedy, keď hodnosť matice $\mathbf{A}$ ľavej strany sa rovná hodnosti rozšírenej matice $(\mathbf{A}|\mathbf{b})$ sústavy.
- každé riešenie sústavy $\mathbf{A}.\mathbf{x} = \mathbf{b}$ dostaneme ako súčet jedného pevne vzatého riešenia tejto sústavy a niektorého riešenia sústavy s nulovou pravou stranou – t.j sústavy $\mathbf{A}.\mathbf{x} = \mathbf{o}$.
- Ak $m < n$ a sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie, potom má nekonečne veľa riešení.
- V mnohých praktických úlohách premenné $x_1, x_2, \ldots, x_n$ predstavujú množstvá reálnych látok, substrátov alebo objektov a sústava rovníc $\mathbf{A}.\mathbf{x} = \mathbf{b}$ predstavuje ohraničenia pre tieto množstvá, preto nás zaujímajú len také riešenia tejto sústavy, pre ktoré je $x_1 \geq 0,\ x_2 \geq 0,\ \ldots,\ x_n \geq 0$, t. j. pre ktoré sú množstvá nezáporné.
- Každému nezápornému riešeniu je priradená hodnota lineárnej kriteriálnej funkcie
$$f(\mathbf{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n = \mathbf{c}^T.\mathbf{x}$$
predstavujúca náklady na riešenie $\mathbf{x}$.
- Zo všetkých nezáporných riešení nás zaujíma to, ktoré nám prináša čo najmenšie náklady.

Sústava lineárnych rovníc

Platí

- Sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie práve vtedy, keď hodnosť matice $\mathbf{A}$ ľavej strany sa rovná hodnosti rozšírenej matice $(\mathbf{A}|\mathbf{b})$ sústavy.
- každé riešenie sústavy $\mathbf{A}.\mathbf{x} = \mathbf{b}$ dostaneme ako súčet jedného pevne vzatého riešenia tejto sústavy a niektorého riešenia sústavy s nulovou pravou stranou – t.j sústavy $\mathbf{A}.\mathbf{x} = \mathbf{o}$.
- Ak $m < n$ a sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie, potom má nekonečne veľa riešení.
- V mnohých praktických úlohách premenné $x_1, x_2, \dots, x_n$ predstavujú množstvá reálnych látok, substrátov alebo objektov a sústava rovníc $\mathbf{A}.\mathbf{x} = \mathbf{b}$ predstavuje ohraničenia pre tieto množstvá, preto nás zaujímajú len také riešenia tejto sústavy, pre ktoré je $x_1 \geq 0,\ x_2 \geq 0,\ \dots,\ x_n \geq 0$, t. j. pre ktoré sú množstvá nezáporné.
- Každému nezápornému riešeniu je priradená hodnota lineárnej kriteriálnej funkcie
$$f(\mathbf{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n = \mathbf{c}^T.\mathbf{x}$$
predstavujúca náklady na riešenie $\mathbf{x}$.
- Zo všetkých nezáporných riešení nás zaujíma to, ktoré nám prináša čo najmenšie náklady.

Sústava lineárnych rovníc

Platí

- Sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie práve vtedy, keď hodnosť matice $\mathbf{A}$ ľavej strany sa rovná hodnosti rozšírenej matice $(\mathbf{A}|\mathbf{b})$ sústavy.
- každé riešenie sústavy $\mathbf{A}.\mathbf{x} = \mathbf{b}$ dostaneme ako súčet jedného pevne vzatého riešenia tejto sústavy a niektorého riešenia sústavy s nulovou pravou stranou – t.j sústavy $\mathbf{A}.\mathbf{x} = \mathbf{o}$.
- Ak $m < n$ a sústava $\mathbf{A}.\mathbf{x} = \mathbf{b}$ má aspoň jedno riešenie, potom má nekonečne veľa riešení.
- V mnohých praktických úlohách premenné $x_1, x_2, \dots, x_n$ predstavujú množstvá reálnych látok, substrátov alebo objektov a sústava rovníc $\mathbf{A}.\mathbf{x} = \mathbf{b}$ predstavuje ohraničenia pre tieto množstvá, preto nás zaujímajú len také riešenia tejto sústavy, pre ktoré je $x_1 \geq 0,\ x_2 \geq 0,\ \dots,\ x_n \geq 0$, t. j. pre ktoré sú množstvá nezáporné.
- Každému nezápornému riešeniu je priradená hodnota lineárnej kriteriálnej funkcie
$$f(\mathbf{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n = \mathbf{c}^T.\mathbf{x}$$
predstavujúca náklady na riešenie $\mathbf{x}$.
- Zo všetkých nezáporných riešení nás zaujíma to, ktoré nám prináša čo najmenšie náklady.

Definícia

Úloha lineárneho programovania *je nájsť také reálne čísla* $x_1, x_2, \dots, x_n$*, pre ktoré je*

$$f(\mathbf{x}) = \mathbf{c}^T.\mathbf{x} = c_1x_1 + c_2x_2 + \dots + c_nx_n \quad \textit{minimálne}$$

za predpokladov:

$$\begin{array}{rl}
a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n & = b_1 \\
a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n & = b_2 \\
\dots\dots\dots\dots\dots\dots\dots\dots & \dots \\
a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n & = b_m \\
 & \\
x_1, x_2, \dots, x_n & \geq 0
\end{array}$$

Poznámka

Skrátene pomocou maticových zápisov možno úlohu lineárneho programovania formulovať ako:
Minimalizovať $\mathbf{c}^T.\mathbf{x}$ *za predpokladov* $\mathbf{A}.\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq 0$.

Úloha lineárneho programovania

Definícia

Úloha lineárneho programovania *je nájsť také reálne čísla* $x_1, x_2, \ldots, x_n$*, pre ktoré je*

$$f(\mathbf{x}) = \mathbf{c}^T.\mathbf{x} = c_1x_1 + c_2x_2 + \cdots + c_nx_n \quad \textit{minimálne}$$

za predpokladov:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots &\cdots\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \\ \\ x_1, x_2, \ldots, x_n &\geq 0 \end{aligned}$$

Poznámka

Skrátene pomocou maticových zápisov možno úlohu lineárneho programovania formulovať ako:
Minimalizovať $\mathbf{c}^T.\mathbf{x}$ *za predpokladov* $\mathbf{A}.\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq 0$.

Začiatky teórie lineárneho programovania

Poznámka

Na riešenie problému lineárneho programovania je niekoľko efektívnych algoritmov - simplexová metóda, revidovaná simplexová metóda, atď.
Na riešenie takýchto úloh máme k dispozícii komerčné i voľne dostupné výpočtové systémy, ktoré zvládajú inštancie s tisíckami ohraničení i premenných.

Poznámka

- *Lineárne programovanie vzniklo za druhej svetovej vojny ako matematický model na plánovanie výdavkov a tržieb na zníženie výdavkov vlastnej armády a zvýšenie strát nepriateľa. Bolo držané v tajnosti až do roku 1947.*
- *Zakladateli metódy boli Leonid Kantorovič (1912 - 1986) (Rus) (1939) a George B. Dantzig (1914 - 2005) – r. 1947 publikoval slávnu simplexovú metódu.*
- *R. 1975 Tjalling C. Koompans a Leonid V. Kantorovič dostali Nobelovu cenu za rozvoj lineárneho programovania ("for their contributions to the theory of optimal allocation of resources.")*

Začiatky teórie lineárneho programovania

Poznámka

Na riešenie problému lineárneho programovania je niekoľko efektívnych algoritmov - simplexová metóda, revidovaná simplexová metóda, atď.
Na riešenie takýchto úloh máme k dispozícii komerčné i voľne dostupné výpočtové systémy, ktoré zvládajú inštancie s tisíckami ohraničení i premenných.

Poznámka

- *Lineárne programovanie vzniklo za druhej svetovej vojny ako matematický model na plánovanie výdavkov a tržieb na zníženie výdavkov vlastnej armády a zvýšenie strát nepriateľa. Bolo držané v tajnosti až do roku 1947.*
- *Zakladateli metódy boli Leonid Kantorovič (1912 - 1986) (Rus) (1939) a George B. Dantzig (1914 - 2005) – r. 1947 publikoval slávnu simplexovú metódu.*
- *R. 1975 Tjalling C. Koompans a Leonid V. Kantorovič dostali Nobelovu cenu za rozvoj lineárneho programovania ("for their contributions to the theory of optimal allocation of resources.")*

Začiatky teórie lineárneho programovania

Poznámka

Na riešenie problému lineárneho programovania je niekoľko efektívnych algoritmov - simplexová metóda, revidovaná simplexová metóda, atď.
Na riešenie takýchto úloh máme k dispozícii komerčné i voľne dostupné výpočtové systémy, ktoré zvládajú inštancie s tisíckami ohraničení i premenných.

Poznámka

- *Lineárne programovanie vzniklo za druhej svetovej vojny ako matematický model na plánovanie výdavkov a tržieb na zníženie výdavkov vlastnej armády a zvýšenie strát nepriateľa. Bolo držané v tajnosti až do roku 1947.*
- *Zakladateli metódy boli Leonid Kantorovič (1912 - 1986) (Rus) (1939) a George B. Dantzig (1914 - 2005) – r. 1947 publikoval slávnu simplexovú metódu.*
- *R. 1975 Tjalling C. Koompans a Leonid V. Kantorovič dostali Nobelovu cenu za rozvoj lineárneho programovania ("for their contributions to the theory of optimal allocation of resources.")*

Začiatky teórie lineárneho programovania

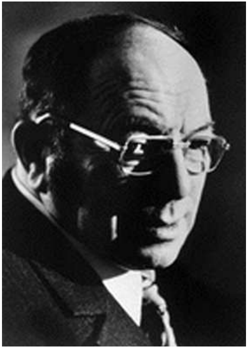
Poznámka

Na riešenie problému lineárneho programovania je niekoľko efektívnych algoritmov - simplexová metóda, revidovaná simplexová metóda, atď.
Na riešenie takýchto úloh máme k dispozícii komerčné i voľne dostupné výpočtové systémy, ktoré zvládajú inštancie s tisíckami ohraničení i premenných.

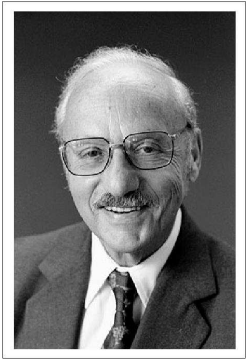
Poznámka

- *Lineárne programovanie vzniklo za druhej svetovej vojny ako matematický model na plánovanie výdavkov a tržieb na zníženie výdavkov vlastnej armády a zvýšenie strát nepriateľa. Bolo držané v tajnosti až do roku 1947.*
- *Zakladateli metódy boli Leonid Kantorovič (1912 - 1986) (Rus) (1939) a George B. Dantzig (1914 - 2005) – r. 1947 publikoval slávnu simplexovú metódu.*
- *R. 1975 Tjalling C. Koompans a Leonid V. Kantorovič dostali Nobelovu cenu za rozvoj lineárneho programovania ("for their contributions to the theory of optimal allocation of resources.")*

Zakladatelia teórie lineárneho programovania



Leonid V. Kantorovič
(1912 - 1986)



George B. Dantzig
(1914 - 2005)

Tjalling C. Koompans
(1910 - 1985)

Úloha celočíselného lineárneho programovania

Pri riešení ďalších praktických úloh požadujeme, aby premenné $x_1, x_2, \ldots, x_n$ predstavovali počty reálnych objektov.
V tomto prípade musia byť všetky premenné $x_1, x_2, \ldots, x_n$ celé čísla.

Definícia

Úloha celočíselného lineárneho programovania *(úloha CLP) je nájsť takú n–ticu celých čísel* $\mathbf{x}$*, pre ktorú je* $\mathbf{c}^T.\mathbf{x}$ *minimálne za predpokladov* $\mathbf{A}.\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq 0$.

V niektorých prípadoch premenné x_i modelujú rozhodnutia, či nejakú akciu vykonať – vtedy $x_i = 1$, alebo nie – vtedy $x_i = 0$.
Špeciálnym prípadom celočíselného lineárneho programovania je prípad, keď požadujeme, aby všetky premenné nadobúdali len hodnoty 0 alebo 1.

Definícia

Úloha bivalentného (alebo binárneho) lineárneho programovania *(úloha BLP) je nájsť takú n–ticu celých čísel* $\mathbf{x}$*, pre ktorú je* $\mathbf{c}^T.\mathbf{x}$ *minimálne za predpokladov* $\mathbf{A}.\mathbf{x} = \mathbf{b}$, $x_i \in \{0, 1\}$ *pre* $i = 1, 2, \ldots, n$.

Úloha celočíselného lineárneho programovania

Pri riešení ďalších praktických úloh požadujeme, aby premenné $x_1, x_2, \dots, x_n$ predstavovali počty reálnych objektov.
V tomto prípade musia byť všetky premenné $x_1, x_2, \dots, x_n$ celé čísla.

Definícia

Úloha celočíselného lineárneho programovania *(úloha CLP) je nájsť takú n–ticu celých čísel* $\mathbf{x}$*, pre ktorú je* $\mathbf{c}^T.\mathbf{x}$ *minimálne za predpokladov* $\mathbf{A}.\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq 0$*.*

V niektorých prípadoch premenné x_i modelujú rozhodnutia, či nejakú akciu vykonať – vtedy $x_i = 1$, alebo nie – vtedy $x_i = 0$.
Špeciálnym prípadom celočíselného lineárneho programovania je prípad, keď požadujeme, aby všetky premenné nadobúdali len hodnoty 0 alebo 1.

Definícia

Úloha bivalentného (alebo binárneho) lineárneho programovania *(úloha BLP) je nájsť takú n–ticu celých čísel* $\mathbf{x}$*, pre ktorú je* $\mathbf{c}^T.\mathbf{x}$ *minimálne za predpokladov* $\mathbf{A}.\mathbf{x} = \mathbf{b}$, $x_i \in \{0, 1\}$ *pre* $i = 1, 2, \dots, n$*.*

Príklad – najkratšia cesta ako úloha BLP

Príklad, ako možno úlohu hľadania najkratšej u–v cesty v hranovo ohodnotenom digrafe $\overrightarrow{G} = (V, H, c)$, kde $c(h) > 0$, previesť na úlohu bivalentného programovania.

Označme:
$$c_{ij} = \begin{cases} c(i,j) & \text{ak } (i,j) \in H \\ \infty & \text{ak } (i,j) \notin H \end{cases} \tag{3}$$

Nech x_{ij} je bivalentná premenná, ktorá nadobúda hodnoty nasledovne:

$$x_{ij} = \begin{cases} 1 & \text{ak } u\text{–}v \text{ cesta } \mu(u,v) \text{ obsahuje hranu } (i,j) \\ 0 & \text{inak} \end{cases} \tag{4}$$

Predstavme si, že štvorcová matica typu n (x_{ij}) zložená iba z núl a jedničiek zodpovedá nejakej ceste $\mu(u,v)$ v digrafe $\overrightarrow{G}$.
Potom pre ľubovoľné $(i,j) \in V \times V$ je $c_{ij}.x_{ij}$ rovné dĺžke hrany (i,j) ak hrana (i,j) leží na ceste $\mu(u,v)$, alebo nula inak.

Z toho môžeme usúdiť, že
$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{5}$$

predstavuje dĺžku cesty $\mu(u,v)$.

Príklad – najkratšia cesta ako úloha BLP

Príklad, ako možno úlohu hľadania najkratšej u–v cesty v hranovo ohodnotenom digrafe $\overrightarrow{G} = (V, H, c)$, kde $c(h) > 0$, previesť na úlohu bivalentného programovania.

Označme:
$$c_{ij} = \begin{cases} c(i,j) & \text{ak } (i,j) \in H \\ \infty & \text{ak } (i,j) \notin H \end{cases} \tag{3}$$

Nech x_{ij} je bivalentná premenná, ktorá nadobúda hodnoty nasledovne:
$$x_{ij} = \begin{cases} 1 & \text{ak } u\text{–}v \text{ cesta } \boldsymbol{\mu}(u,v) \text{ obsahuje hranu } (i,j) \\ 0 & \text{inak} \end{cases} \tag{4}$$

Predstavme si, že štvorcová matica typu n (x_{ij}) zložená iba z núl a jedničiek zodpovedá nejakej ceste $\mu(u,v)$ v digrafe $\overrightarrow{G}$.
Potom pre ľubovoľné $(i,j) \in V \times V$ je $c_{ij}.x_{ij}$ rovné dĺžke hrany (i,j) ak hrana (i,j) leží na ceste $\mu(u,v)$, alebo nula inak.

Z toho môžeme usúdiť, že
$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{5}$$

predstavuje dĺžku cesty $\mu(u,v)$.

Príklad – najkratšia cesta ako úloha BLP

Príklad, ako možno úlohu hľadania najkratšej u–v cesty v hranovo ohodnotenom digrafe $\overrightarrow{G} = (V, H, c)$, kde $c(h) > 0$, previesť na úlohu bivalentného programovania.

Označme:
$$c_{ij} = \begin{cases} c(i,j) & \text{ak } (i,j) \in H \\ \infty & \text{ak } (i,j) \notin H \end{cases} \tag{3}$$

Nech x_{ij} je bivalentná premenná, ktorá nadobúda hodnoty nasledovne:

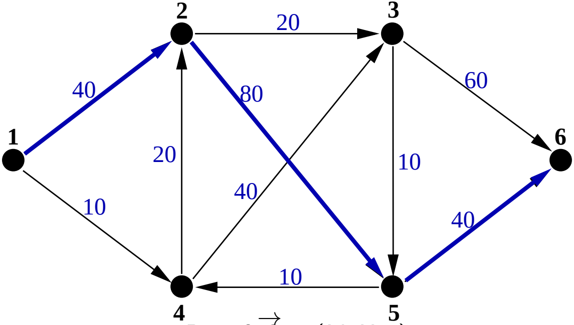
$$x_{ij} = \begin{cases} 1 & \text{ak } u\text{–}v \text{ cesta } \boldsymbol{\mu}(u,v) \text{ obsahuje hranu } (i,j) \\ 0 & \text{inak} \end{cases} \tag{4}$$

Predstavme si, že štvorcová matica typu n (x_{ij}) zložená iba z núl a jedničiek zodpovedá nejakej ceste $\boldsymbol{\mu}(u,v)$ v digrafe $\overrightarrow{G}$.
Potom pre ľubovoľné $(i,j) \in V \times V$ je $c_{ij}.x_{ij}$ rovné dĺžke hrany (i,j) ak hrana (i,j) leží na ceste $\boldsymbol{\mu}(u,v)$, alebo nula inak.

Z toho môžeme usúdiť, že
$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{5}$$

predstavuje dĺžku cesty $\boldsymbol{\mu}(u,v)$.

Príklad – najkratšia cesta ako úloha BLP



Digraf $\overrightarrow{G} = (V, H, c)$.

	1	2	3	4	5	6
1	∞	40	∞	10	∞	∞
2	∞	∞	20	∞	80	∞
3	∞	∞	∞	∞	10	60
4	∞	20	50	∞	∞	∞
5	∞	∞	∞	10	∞	40
6	∞	∞	∞	∞	∞	∞

Matica $\mathbf{c} = (c_{ij})$
príslušná k digrafu $\overrightarrow{G}$.

	1	2	3	4	5	6
1	-	1	-	-	-	-
2	-	-	-	-	1	-
3	-	-	-	-	-	-
4	-	-	-	-	-	-
5	-	-	-	-	-	1
6	-	-	-	-	-	-

Matica $\mathbf{x} = (x_{ij})$ pre
vyznačenú cestu v $\overrightarrow{G}$.

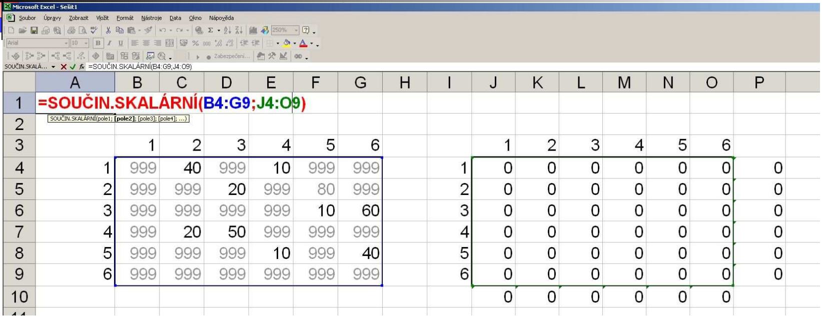
Príklad – najkratšia cesta ako úloha BLP

Model bivalentného programovania
pre problém hľadania najkratšej u-v cesty v digrafe.

$$\text{Minimalizovať} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

$$\begin{aligned}
\text{za predpokladov:} \quad & \sum_{j=1}^{n} x_{uj} && = 1 \\
& \sum_{i=1}^{n} x_{iv} && = 1 \\
& \sum_{j=1}^{n} x_{kj} && = \sum_{i=1}^{n} x_{ik} \quad \forall k \in V,\ k \neq u,\ k \neq v \\
& \quad x_{ij} && \in \{0,1\}
\end{aligned}$$

Príklad – najkratšia cesta ako úloha BLP

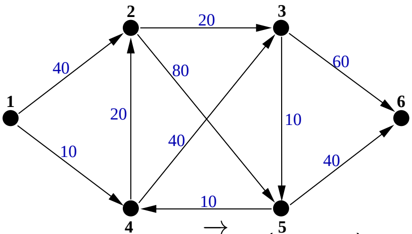


Microsoft Excel - Sešit1

Soubor Úpravy Zobrazit Vložit Formát Nástroje Data Okno Nápověda

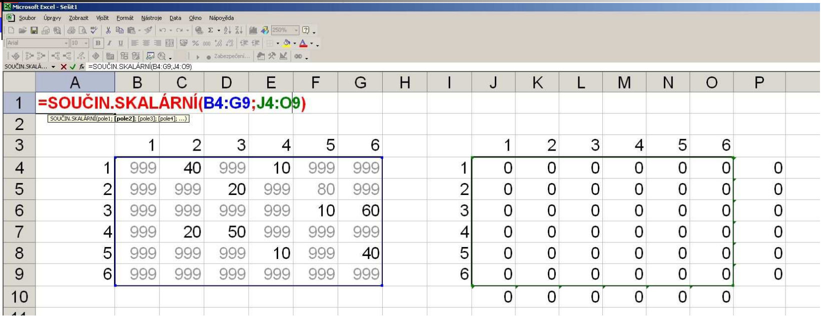
SOUČIN.SKALÁ... =SOUČIN.SKALÁRNÍ(B4:G9;J4:O9)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	=SOUČIN.SKALÁRNÍ(B4:G9;J4:O9)															
2	SOUČIN.SKALÁRNÍ(pole1; [pole2]; [pole3]; [pole4]; ...)															
3		1	2	3	4	5	6			1	2	3	4	5	6	
4	1	999	40	999	10	999	999		1	0	0	0	0	0	0	0
5	2	999	999	20	999	80	999		2	0	0	0	0	0	0	0
6	3	999	999	999	999	10	60		3	0	0	0	0	0	0	0
7	4	999	20	50	999	999	999		4	0	0	0	0	0	0	0
8	5	999	999	999	10	999	40		5	0	0	0	0	0	0	0
9	6	999	999	999	999	999	999		6	0	0	0	0	0	0	0
10										0	0	0	0	0	0	

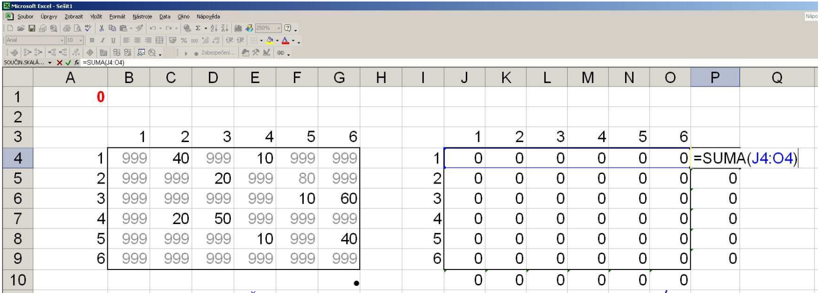


Digraf $\overrightarrow{G} = (V, H, c)$.

Príklad – najkratšia cesta ako úloha BLP

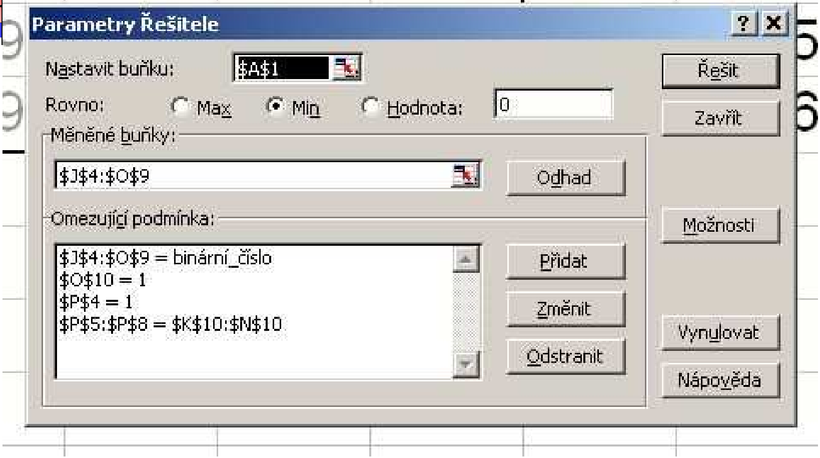


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	=SOUČIN.SKALÁRNÍ(B4:G9;J4:O9)															
2																
3		1	2	3	4	5	6			1	2	3	4	5	6	
4	1	999	40	999	10	999	999		1	0	0	0	0	0	0	0
5	2	999	999	20	999	80	999		2	0	0	0	0	0	0	0
6	3	999	999	999	999	10	60		3	0	0	0	0	0	0	0
7	4	999	20	50	999	999	999		4	0	0	0	0	0	0	0
8	5	999	999	999	10	999	40		5	0	0	0	0	0	0	0
9	6	999	999	999	999	999	999		6	0	0	0	0	0	0	0
10										0	0	0	0	0	0	

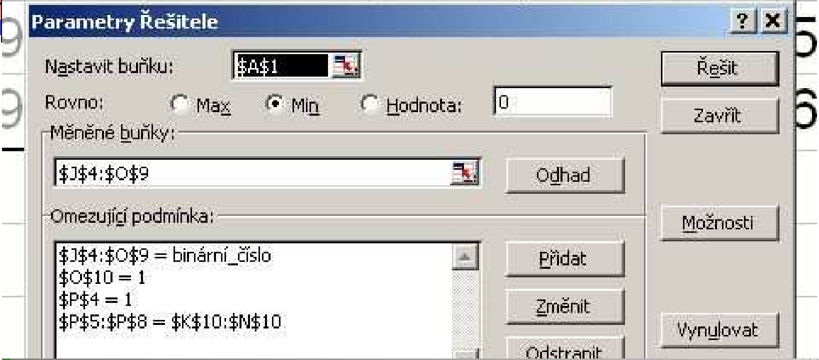


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	0																
2																	
3		1	2	3	4	5	6			1	2	3	4	5	6		
4	1	999	40	999	10	999	999		1	0	0	0	0	0	0	=SUMA(J4:O4)	
5	2	999	999	20	999	80	999		2	0	0	0	0	0	0	0	
6	3	999	999	999	999	10	60		3	0	0	0	0	0	0	0	
7	4	999	20	50	999	999	999		4	0	0	0	0	0	0	0	
8	5	999	999	999	10	999	40		5	0	0	0	0	0	0	0	
9	6	999	999	999	999	999	999		6	0	0	0	0	0	0	0	
10										0	0	0	0	0	0		

Príklad – naikratšia cesta ako úloha BLP

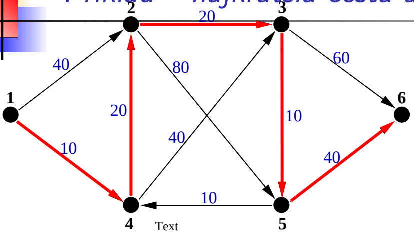


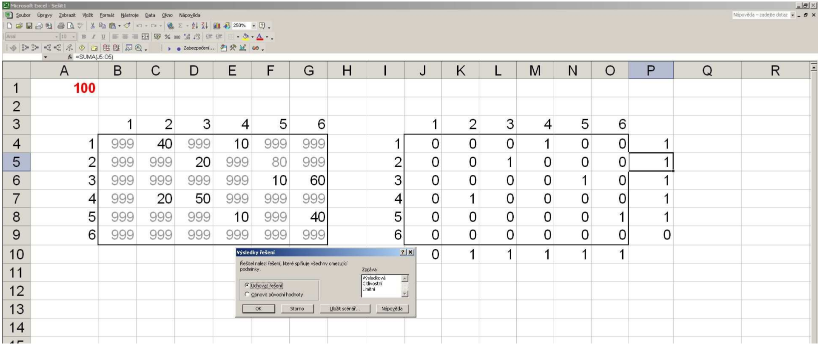
Príklad – naikratšia cesta ako úloha BLP



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	100															
2																
3		1	2	3	4	5	6			1	2	3	4	5	6	
4	1	999	40	999	10	999	999		1	0	0	0	1	0	0	1
5	2	999	999	20	999	80	999		2	0	0	1	0	0	0	1
6	3	999	999	999	999	10	60		3	0	0	0	0	1	0	1
7	4	999	20	50	999	999	999		4	0	1	0	0	0	0	1
8	5	999	999	999	10	999	40		5	0	0	0	0	0	1	1
9	6	999	999	999	999	999	999		6	0	0	0	0	0	0	0
10										0	1	1	1	1	1	

Príklad – najkratšia cesta ako úloha BLP





	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	100															
2																
3		1	2	3	4	5	6			1	2	3	4	5	6	
4	1	999	40	999	10	999	999		1	0	0	0	1	0	0	1
5	2	999	999	20	999	80	999		2	0	0	1	0	0	0	1
6	3	999	999	999	999	10	60		3	0	0	0	0	1	0	1
7	4	999	20	50	999	999	999		4	0	1	0	0	0	0	1
8	5	999	999	999	10	999	40		5	0	0	0	0	0	1	1
9	6	999	999	999	999	999	999		6	0	0	0	0	0	0	0
10										0	1	1	1	1	1	

Definícia

Nech je daný problém $\mathcal{P}_1$ so vstupnými dátami D_1.

Problém $\mathcal{P}_1$ môžeme riešiť aj tak, že ho pretransformujeme na iný problém $\mathcal{P}_2$ tak, že dáta D_1 prerobíme na dáta D_2 problému $\mathcal{P}_2$.

Ak riešenie R_2 problému $\mathcal{P}_2$ vieme prerobiť na riešenie R_1 problému $\mathcal{P}_1$, transformácia je hotová.

Ak prerobenie dát D_1 na D_2 a riešenia R_2 na R_1 vyžaduje len polynomiálny počet elementárnych operácií, nazveme túto transformáciu **polynomiálnou transformáciou**.

Ak problém $\mathcal{P}_1$ možno polynomiálne transformovať na problém $\mathcal{P}_2$ a problém $\mathcal{P}_2$ má polynomiálny algoritmus riešenia, potom aj problém $\mathcal{P}_1$ má polynomiálny algoritmus riešenia

Polynomiálne transformácie

Definícia

Nech je daný problém $\mathcal{P}_1$ so vstupnými dátami D_1.

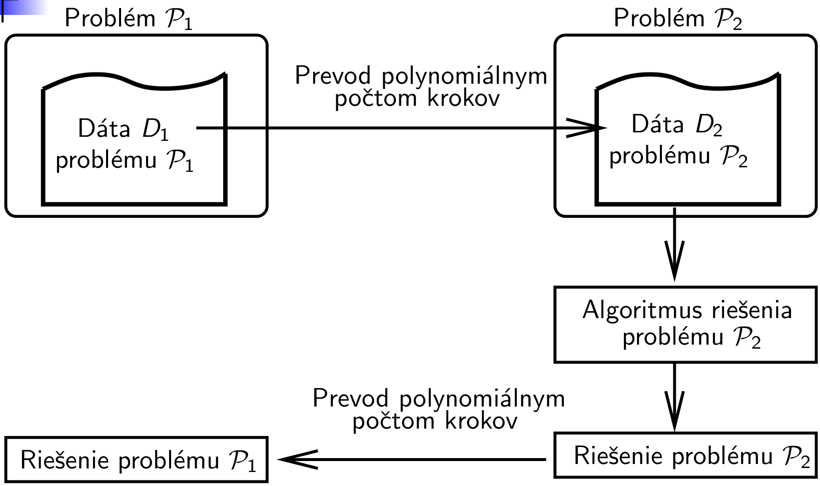
Problém $\mathcal{P}_1$ môžeme riešiť aj tak, že ho pretransformujeme na iný problém $\mathcal{P}_2$ tak, že dáta D_1 prerobíme na dáta D_2 problému $\mathcal{P}_2$.

Ak riešenie R_2 problému $\mathcal{P}_2$ vieme prerobiť na riešenie R_1 problému $\mathcal{P}_1$, transformácia je hotová.

Ak prerobenie dát D_1 na D_2 a riešenia R_2 na R_1 vyžaduje len polynomiálny počet elementárnych operácií, nazveme túto transformáciu **polynomiálnou transformáciou**.

Ak problém $\mathcal{P}_1$ možno polynomiálne transformovať na problém $\mathcal{P}_2$ a problém $\mathcal{P}_2$ má polynomiálny algoritmus riešenia, potom aj problém $\mathcal{P}_1$ má polynomiálny algoritmus riešenia

Polynomiálne transformácie



Obr.: Polynomiálna transformácia problému $\mathcal{P}_1$ na problém $\mathcal{P}_2$.

Polynomiálna redukcia

Definícia

V niektorých prípadoch vyriešenie problému $\mathcal{P}_1$ vyžaduje vyriešiť niekoľko prípadov problému $\mathcal{P}_2$ (napr. násobenie prevádzame na niekoľko sčítaní).

Ak prevody dát a riešení vyžadujú len polynomiálny počet elementárnych operácií a výpočet vyžaduje polynomiálny počet výpočtov problému $\mathcal{P}_2$, hovoríme, že sme problém $\mathcal{P}_1$ **polynomiálne redukovali na problém** $\mathcal{P}_2$.

Ak problém $\mathcal{P}_1$ možno polynomiálne redukovať na problém $\mathcal{P}_2$ a naopak, problém $\mathcal{P}_2$ možno polynomiálne redukovať na $\mathcal{P}_1$ hovoríme, že problémy $\mathcal{P}_1$, $\mathcal{P}_2$ sú **polynomiálne ekvivalentné**.

Definícia

V niektorých prípadoch vyriešenie problému $\mathcal{P}_1$ vyžaduje vyriešiť niekoľko prípadov problému $\mathcal{P}_2$ (napr. násobenie prevádzame na niekoľko sčítaní).

Ak prevody dát a riešení vyžadujú len polynomiálny počet elementárnych operácií a výpočet vyžaduje polynomiálny počet výpočtov problému $\mathcal{P}_2$, hovoríme, že sme problém $\mathcal{P}_1$ **polynomiálne redukovali na problém** $\mathcal{P}_2$.

Ak problém $\mathcal{P}_1$ možno polynomiálne redukovať na problém $\mathcal{P}_2$ a naopak, problém $\mathcal{P}_2$ možno polynomiálne redukovať na $\mathcal{P}_1$ hovoríme, že problémy $\mathcal{P}_1$, $\mathcal{P}_2$ sú **polynomiálne ekvivalentné**.

Polynomiálna redukcia

Ak úlohu $\mathcal{P}_1$ možno polynomiálne redukovať na úlohu $\mathcal{P}_2$, znamená to nasledovné:

- Ak existuje polynomiálny algoritmus riešenia úlohy $\mathcal{P}_2$, potom existuje aj polynomiálny algoritmus pre úlohu $\mathcal{P}_1$.
- Naopak to však nemusí platiť – ak existuje polynomiálny algoritmus pre $\mathcal{P}_1$, polynomiálna redukovateľnosť $\mathcal{P}_1$ na $\mathcal{P}_2$ ešte nič nehovorí o zložitosti problému $\mathcal{P}_2$.
- Ak sú však problémy $\mathcal{P}_1$, $\mathcal{P}_2$ polynomiálne ekvivalentné, existencia polynomiálneho algoritmu pre jeden implikuje existenciu polynomiálneho algoritmu pre druhý.

Polynomiálna redukcia

Ak úlohu $\mathcal{P}_1$ možno polynomiálne redukovať na úlohu $\mathcal{P}_2$, znamená to nasledovné:

- Ak existuje polynomiálny algoritmus riešenia úlohy $\mathcal{P}_2$, potom existuje aj polynomiálny algoritmus pre úlohu $\mathcal{P}_1$.
- Naopak to však nemusí platiť – ak existuje polynomiálny algoritmus pre $\mathcal{P}_1$, polynomiálna redukovateľnosť $\mathcal{P}_1$ na $\mathcal{P}_2$ ešte nič nehovorí o zložitosti problému $\mathcal{P}_2$.
- Ak sú však problémy $\mathcal{P}_1$, $\mathcal{P}_2$ polynomiálne ekvivalentné, existencia polynomiálneho algoritmu pre jeden implikuje existenciu polynomiálneho algoritmu pre druhý.

Polynomiálna redukcia

Ak úlohu $\mathcal{P}_1$ možno polynomiálne redukovať na úlohu $\mathcal{P}_2$, znamená to nasledovné:

- Ak existuje polynomiálny algoritmus riešenia úlohy $\mathcal{P}_2$, potom existuje aj polynomiálny algoritmus pre úlohu $\mathcal{P}_1$.
- Naopak to však nemusí platiť – ak existuje polynomiálny algoritmus pre $\mathcal{P}_1$, polynomiálna redukovateľnosť $\mathcal{P}_1$ na $\mathcal{P}_2$ ešte nič nehovorí o zložitosti problému $\mathcal{P}_2$.
- Ak sú však problémy $\mathcal{P}_1$, $\mathcal{P}_2$ polynomiálne ekvivalentné, existencia polynomiálneho algoritmu pre jeden implikuje existenciu polynomiálneho algoritmu pre druhý.

NP–ťažké úlohy

Kombinatorické úlohy, ku ktorým patria aj mnohé úlohy teórie grafov, spočívajú vo výbere jedného optimálneho z konečného počtu kombinatorických objektov.
Najjednoduchším spôsobom riešenia takýchto úloh je úplné prehľadanie všetkých objektov (full search, brute force).

		$n = 10$	$n = 50$	$n = 100$
Počet štvorcových matíc $n \times n$	n^2	100	2500	1.0e+4
Počet podmnožín n–prvkovej množiny	2^n	1024	1.1e+15	1.2e+30
Počet permutácií z n prvkov	$n!$	3.6e+6	3.0e+64	9.3e+157
Počet všetkých zobrazení n–prvkovej množiny do seba	n^n	1.0e+10	8.9e+84	1.0e+200

Poznámka

Edmonds (1965) bol prvý, ktorý si uvedomil rozdiel medzi polynomiálnymi algoritmami (t. j. algoritmami so zložitosťou typu $O(n^k)$) a nepolynomiálnymi algoritmami, ktorých počet krokov nevieme ohraničiť polynómom. Tie prvé nazýva „dobré“.

NP–ťažké úlohy

Kombinatorické úlohy, ku ktorým patria aj mnohé úlohy teórie grafov, spočívajú vo výbere jedného optimálneho z konečného počtu kombinatorických objektov.
Najjednoduchším spôsobom riešenia takýchto úloh je úplné prehľadanie všetkých objektov (full search, brute force).

		$n = 10$	$n = 50$	$n = 100$
Počet štvorcových matíc $n \times n$	n^2	100	2500	1.0e+4
Počet podmnožín n–prvkovej množiny	2^n	1024	1.1e+15	1.2e+30
Počet permutácií z n prvkov	$n!$	3.6e+6	3.0e+64	9.3e+157
Počet všetkých zobrazení n–prvkovej množiny do seba	n^n	1.0e+10	8.9e+84	1.0e+200

Poznámka

Edmonds (1965) bol prvý, ktorý si uvedomil rozdiel medzi polynomiálnymi algoritmami (t. j. algoritmami so zložitosťou typu $O(n^k)$) a nepolynomiálnymi algoritmami, ktorých počet krokov nevieme ohraničiť polynómom. Tie prvé nazýva „dobré“.

NP–ťažké úlohy

- Ako etalón ťažkých úloh bola vybratá úloha bivalentného lineárneho programovania (BLP).
- Problém, ktorý možno polynomiálne redukovať na úlohu BLP, je ľahší alebo rovnako ťažký ako úloha BLP.
- Problém, na ktorý možno polynomiálne redukovať úlohu BLP, je ťažší alebo rovnako ťažký ako úloha BLP.

Definícia

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ťažký**, *ak úlohu bivalentného lineárneho programovania možno polynomiálne redukovať na* $\mathcal{P}$.

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ľahký**, *ak problém* $\mathcal{P}$ *možno polynomiálne redukovať na úlohu bivalentného lineárneho programovania.*

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ekvivalentný**, *ak je problém* $\mathcal{P}$ *polynomiálne ekvivalentný s úlohou bivalentného lineárneho programovania.*

NP–ťažké úlohy

- Ako etalón ťažkých úloh bola vybratá úloha bivalentného lineárneho programovania (BLP).
- Problém, ktorý možno polynomiálne redukovať na úlohu BLP, je ľahší alebo rovnako ťažký ako úloha BLP.
- Problém, na ktorý možno polynomiálne redukovať úlohu BLP, je ťažší alebo rovnako ťažký ako úloha BLP.

Definícia

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ťažký**, *ak úlohu bivalentného lineárneho programovania možno polynomiálne redukovať na* $\mathcal{P}$.

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ľahký**, *ak problém* $\mathcal{P}$ *možno polynomiálne redukovať na úlohu bivalentného lineárneho programovania.*

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ekvivalentný**, *ak je problém* $\mathcal{P}$ *polynomiálne ekvivalentný s úlohou bivalentného lineárneho programovania.*

NP–ťažké úlohy

- Ako etalón ťažkých úloh bola vybratá úloha bivalentného lineárneho programovania (BLP).
- Problém, ktorý možno polynomiálne redukovať na úlohu BLP, je ľahší alebo rovnako ťažký ako úloha BLP.
- Problém, na ktorý možno polynomiálne redukovať úlohu BLP, je ťažší alebo rovnako ťažký ako úloha BLP.

Definícia

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ťažký**, *ak úlohu bivalentného lineárneho programovania možno polynomiálne redukovať na* $\mathcal{P}$.

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ľahký**, *ak problém* $\mathcal{P}$ *možno polynomiálne redukovať na úlohu bivalentného lineárneho programovania.*

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ekvivalentný**, *ak je problém* $\mathcal{P}$ *polynomiálne ekvivalentný s úlohou bivalentného lineárneho programovania.*

NP–ťažké úlohy

- Ako etalón ťažkých úloh bola vybratá úloha bivalentného lineárneho programovania (BLP).
- Problém, ktorý možno polynomiálne redukovať na úlohu BLP, je ľahší alebo rovnako ťažký ako úloha BLP.
- Problém, na ktorý možno polynomiálne redukovať úlohu BLP, je ťažší alebo rovnako ťažký ako úloha BLP.

Definícia

Hovoríme, že **problém $\mathcal{P}$ je NP–ťažký**, *ak úlohu bivalentného lineárneho programovania možno polynomiálne redukovať na $\mathcal{P}$.*

Hovoríme, že **problém $\mathcal{P}$ je NP–ľahký**, *ak problém $\mathcal{P}$ možno polynomiálne redukovať na úlohu bivalentného lineárneho programovania.*

Hovoríme, že **problém $\mathcal{P}$ je NP–ekvivalentný**, *ak je problém $\mathcal{P}$ polynomiálne ekvivalentný s úlohou bivalentného lineárneho programovania.*

NP–ťažké úlohy

- Ako etalón ťažkých úloh bola vybratá úloha bivalentného lineárneho programovania (BLP).
- Problém, ktorý možno polynomiálne redukovať na úlohu BLP, je ľahší alebo rovnako ťažký ako úloha BLP.
- Problém, na ktorý možno polynomiálne redukovať úlohu BLP, je ťažší alebo rovnako ťažký ako úloha BLP.

Definícia

Hovoríme, že **problém $\mathcal{P}$ je NP–ťažký**, *ak úlohu bivalentného lineárneho programovania možno polynomiálne redukovať na $\mathcal{P}$.*

Hovoríme, že **problém $\mathcal{P}$ je NP–ľahký**, *ak problém $\mathcal{P}$ možno polynomiálne redukovať na úlohu bivalentného lineárneho programovania.*

Hovoríme, že **problém $\mathcal{P}$ je NP–ekvivalentný**, *ak je problém $\mathcal{P}$ polynomiálne ekvivalentný s úlohou bivalentného lineárneho programovania.*

NP–ťažké úlohy

- Ako etalón ťažkých úloh bola vybratá úloha bivalentného lineárneho programovania (BLP).
- Problém, ktorý možno polynomiálne redukovať na úlohu BLP, je ľahší alebo rovnako ťažký ako úloha BLP.
- Problém, na ktorý možno polynomiálne redukovať úlohu BLP, je ťažší alebo rovnako ťažký ako úloha BLP.

Definícia

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ťažký**, *ak úlohu bivalentného lineárneho programovania možno polynomiálne redukovať na* $\mathcal{P}$.

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ľahký**, *ak problém* $\mathcal{P}$ *možno polynomiálne redukovať na úlohu bivalentného lineárneho programovania.*

Hovoríme, že **problém** $\mathcal{P}$ **je NP–ekvivalentný**, *ak je problém* $\mathcal{P}$ *polynomiálne ekvivalentný s úlohou bivalentného lineárneho programovania.*

Poznámka

- *Úloha obchodného cestujúceho je NP–ekvivalentná.*
- *Doteraz sa podarilo pre stovky úloh dokázať, že sú NP–ekvivalentné. Pre ďalšie sa podarilo dokázať, že sú NP-ťažké.*
- *Pre žiadnu NP–ťažkú úlohu sa doteraz nepodarilo nájsť polynomiálny algoritmus.*
- *Nepodarilo sa však ani dokázať, že polynomiálny algoritmus pre ne neexistuje.*
- *Nájdenie polynomiálneho algoritmu pre jedinú zo stoviek NP–ekvivalentných úloh by znamenalo existenciu polynomiálneho algoritmu pre všetky ostatné. Všeobecne sa neverí, že by sa to mohlo niekomu podariť, preto sú na NP–zložitosti niektorých úloh založené napr. aj niektoré kryptografické systémy.*

NP–ťažké úlohy

Poznámka

- *Úloha obchodného cestujúceho je NP–ekvivalentná.*
- *Doteraz sa podarilo pre stovky úloh dokázať, že sú NP–ekvivalentné. Pre ďalšie sa podarilo dokázať, že sú NP-ťažké.*
- *Pre žiadnu NP–ťažkú úlohu sa doteraz nepodarilo nájsť polynomiálny algoritmus.*
- *Nepodarilo sa však ani dokázať, že polynomiálny algoritmus pre ne neexistuje.*
- *Nájdenie polynomiálneho algoritmu pre jedinú zo stoviek NP–ekvivalentných úloh by znamenalo existenciu polynomiálneho algoritmu pre všetky ostatné. Všeobecne sa neverí, že by sa to mohlo niekomu podariť, preto sú na NP–zložitosti niektorých úloh založené napr. aj niektoré kryptografické systémy.*

NP–ťažké úlohy

Poznámka

- *Úloha obchodného cestujúceho je NP–ekvivalentná.*
- *Doteraz sa podarilo pre stovky úloh dokázať, že sú NP–ekvivalentné. Pre ďalšie sa podarilo dokázať, že sú NP-ťažké.*
- *Pre žiadnu NP–ťažkú úlohu sa doteraz nepodarilo nájsť polynomiálny algoritmus.*
- *Nepodarilo sa však ani dokázať, že polynomiálny algoritmus pre ne neexistuje.*
- *Nájdenie polynomiálneho algoritmu pre jedinú zo stoviek NP–ekvivalentných úloh by znamenalo existenciu polynomiálneho algoritmu pre všetky ostatné. Všeobecne sa neverí, že by sa to mohlo niekomu podariť, preto sú na NP–zložitosti niektorých úloh založené napr. aj niektoré kryptografické systémy.*

NP–ťažké úlohy

Poznámka

- *Úloha obchodného cestujúceho je NP–ekvivalentná.*
- *Doteraz sa podarilo pre stovky úloh dokázať, že sú NP–ekvivalentné. Pre ďalšie sa podarilo dokázať, že sú NP-ťažké.*
- *Pre žiadnu NP–ťažkú úlohu sa doteraz nepodarilo nájsť polynomiálny algoritmus.*
- *Nepodarilo sa však ani dokázať, že polynomiálny algoritmus pre ne neexistuje.*
- *Nájdenie polynomiálneho algoritmu pre jedinú zo stoviek NP–ekvivalentných úloh by znamenalo existenciu polynomiálneho algoritmu pre všetky ostatné. Všeobecne sa neverí, že by sa to mohlo niekomu podariť, preto sú na NP–zložitosti niektorých úloh založené napr. aj niektoré kryptografické systémy.*

NP–ťažké úlohy

Poznámka

- *Úloha obchodného cestujúceho je NP–ekvivalentná.*
- *Doteraz sa podarilo pre stovky úloh dokázať, že sú NP–ekvivalentné. Pre ďalšie sa podarilo dokázať, že sú NP-ťažké.*
- *Pre žiadnu NP–ťažkú úlohu sa doteraz nepodarilo nájsť polynomiálny algoritmus.*
- *Nepodarilo sa však ani dokázať, že polynomiálny algoritmus pre ne neexistuje.*
- *Nájdenie polynomiálneho algoritmu pre jedinú zo stoviek NP–ekvivalentných úloh by znamenalo existenciu polynomiálneho algoritmu pre všetky ostatné. Všeobecne sa neverí, že by sa to mohlo niekomu podariť, preto sú na NP–zložitosti niektorých úloh založené napr. aj niektoré kryptografické systémy.*

Heuristiky

Heuristiky – postupy či algoritmy, ktoré dávajú riešenie s hodnotou kriteriálnej funkcie blízkou k optimálnej hodnote (presnejšie k hodnote kriteriálnej funkcie optimálneho riešenia).

$$\text{Heuristiky} \quad \begin{cases} \text{vytvárajúce} \\ \text{zlepšujúce} \end{cases}$$

- Vytvárajúce heuristiky – napr. pažravá metóda
- Zlepšujúce heuristiky
  - Metóda prehľadávania okolia
  - Mataheuristika Tabu search
  - Metaheuristika Simulated anealing
  - Metódy vetví a hraníc
  - Genetické algoritmy
  - Metódy kolónia mravcov
  - .....

Heuristiky

Heuristiky – postupy či algoritmy, ktoré dávajú riešenie s hodnotou kriteriálnej funkcie blízkou k optimálnej hodnote (presnejšie k hodnote kriteriálnej funkcie optimálneho riešenia).

$$\text{Heuristiky} \quad \begin{cases} \text{vytvárajúce} \\ \text{zlepšujúce} \end{cases}$$

- Vytvárajúce heuristiky – napr. pažravá metóda
- Zlepšujúce heuristiky
  - Metóda prehľadávania okolia
  - Mataheuristika Tabu search
  - Metaheuristika Simulated anealing
  - Metódy vetví a hraníc
  - Genetické algoritmy
  - Metódy kolónia mravcov
  - .....

Heuristiky

Heuristiky – postupy či algoritmy, ktoré dávajú riešenie s hodnotou kriteriálnej funkcie blízkou k optimálnej hodnote (presnejšie k hodnote kriteriálnej funkcie optimálneho riešenia).

$$\text{Heuristiky} \quad \left\{ \begin{array}{l} \text{vytvárajúce} \\ \text{zlepšujúce} \end{array} \right.$$

- Vytvárajúce heuristiky – napr. pažravá metóda
- Zlepšujúce heuristiky
    - Metóda prehľadávania okolia
    - Mataheuristika Tabu search
    - Metaheuristika Simulated anealing
    - Metódy vetví a hraníc
    - Genetické algoritmy
    - Metódy kolónia mravcov
    - .....