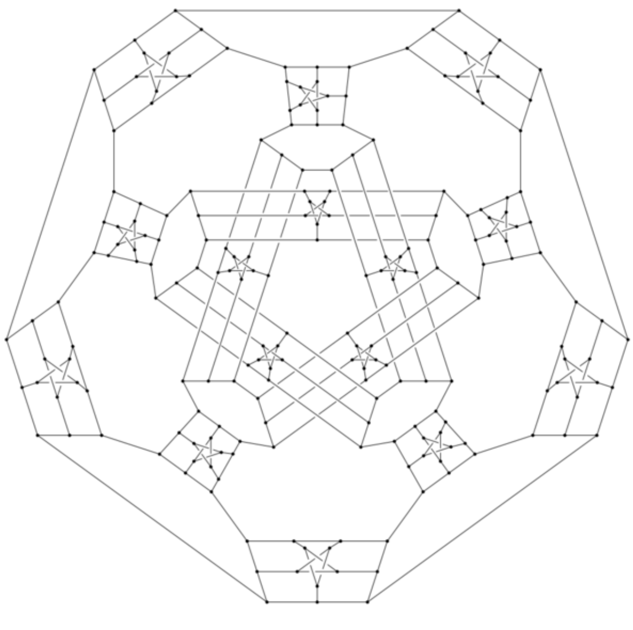
Matematicko-fyzikální fakulta
Univerzity Karlovy v Praze

Zápočtový program k NPRG031
LS 2010/2011

Knihovna GraphAlpha

Autor:
Duc Trung Ha

Cvičící:
RNDr. M. Pergel, Ph.D.



16. srpna 2011

Abstrakt

Dokumentace k zápočtovému programu. Program je implementací knihovny pro práci s grafy orientovanými i neorientovanými.

Obsah

I Úvodní slovo 3

II Programátorský manuál 4

1 Anotace 4

2 Přesné zadání 4

3 Algoritmy a datové struktury 5
- 3.1 Použité datové struktury . . . 5
- 3.2 Použité algoritmy . . . 6
  - 3.2.1 Napojení knihovny na program . . . 6
  - 3.2.2 Funkce `symMtrx` . . . 6
  - 3.2.3 Funkce `dijkstra` . . . 7
  - 3.2.4 Funkce `FloydWarshall` . . . 8
  - 3.2.5 Funkce `Tarjan` . . . 9
  - 3.2.6 Funkce `TSort` . . . 10
  - 3.2.7 Funkce `transitiveClosure` . . . 11
  - 3.2.8 Funkce `transitiveReduction` . . . 11
  - 3.2.9 Další funkce . . . 12

III Uživatelský manuál 13

4 Menu 13

5 Vstup a příkazy programu 14
- 5.1 Sym . . . 14
- 5.2 Dijkstra . . . 15
- 5.3 Floyd-Warshall . . . 15
- 5.4 Tarjan . . . 15
- 5.5 TSort . . . 16
- 5.6 Closure . . . 16
- 5.7 Reduction . . . 16
- 5.8 List-convert . . . 17
- 5.9 Matrix-convert . . . 18
- 5.10 Display . . . 18
- 5.11 Clear . . . 18
- 5.12 Quit . . . 19

Část I

Úvodní slovo

Když v roce 1926 slavný český matematik *Otakar Borůvka* publikoval svou práci *O jistém problému minimálním*—potýkající se s problémem elektrické sítě na Moravě—zřejmě netušil, že píše o problému z oblasti *teorie grafů* bez jejího jediného využití. Vskutku—jeho práce se značně opírala o aparát lineární algebry a co tehdy zabralo několik desítek stran, by dnes za pomoci grafových pojmů šlo popsat na počtu stránek spočitatelném na prstech jedné ruky.

Teorie grafů však neobnáší pouhou *sílu* terminologie, nýbrž dokáže neobyčejně zdárně popsat problémy všedního života—ať už se již jedná o elektrickou síť, dopravní infrastrukturu, finanční tok či vzájemné vztahy uživatelů na *Facebooku*.

A právě knihovna `GraphAlpha` se snaží algoritmicky *vydolovat* některé základní informace o grafech—neboli informaticky řečeno na nich provádět tzv. *grafové operace*. V drtivé většině případů k tomu využívá známé algoritmy slavných informatiků. Autor pevně doufá, že pročítáním jejich popisu a implementace příjemně strávíte svůj volný čas.

Část II
Programátorský manuál

1 Anotace

Knihovna pro práci s grafy orientovanými a neorientovanými (a třeba také s multigrafy nebo hypergrafy) — definuje si nějaké rozumné uložení grafu v paměti (třeba pole vrcholů a seznamy hran z vrcholů vycházejících) a nabízí funkce pro běžné grafové operace (např. hledání nejkratší cesty, minimální kostry, komponent souvislosti, transitivní redukce či uzávěru, topologické třídění grafu).

2 Přesné zadání

Implementujte knihovnu pro práci s váženými digrafy a operacemi nad nimi. Graf bude representován následující datovou strukturou:

- ♠ **Pole vrcholů** obsahující informace o svých následovnících.
- ♠ **Spoják následovníků** u každého vrcholu, obsahující navíc informaci o váze (resp. ceně) hrany.

Dále bude nad digrafy možné provádět následující operace:

- ♣ **Sym** orientující každou hranu digrafu *do obou směrů*.
- ♣ **Dijkstra** hledající vzdálenosti (tj. délky nejkratších cest) od zadaného vrcholu ke všem ostatním vrcholům digrafu.
- ♣ **Floyd-Warshall** hledající vzdálenosti mezi každými dvěma vrcholy digrafu.
- ♣ **Tarjan** hledající komponenty silné souvislosti za pomoci Tarjanova algoritmu.
- ♣ **TSort** hledající topologické uspořádání a v případě neexistence výpis o přítomnosti cyklu v digrafu.[1]
- ♣ **Closure** hledající transitivní uzávěr digrafu.
- ♣ **Reduction** hledající transitivní redukci digrafu.[2]

[1] Což je ekvivalentní s neexistencí topologického uspořádání v digrafu.
[2] Pozn. autora: Tato implementace zatím zvládá jen transitivní redukci v acyklických grafech.

3 Algoritmy a datové struktury

3.1 Použité datové struktury

V této sekci si přiblížíme datové části použité v programu.

Hlavní struktura	**Datový typ položky**	**Název položky**	**Popis**
arc	unsigned int	index	*index vrcholu, do něhož vede hrana*
	double	weight	*váha (potažmo cena) dané hrany*
	arc *	next	*ukazatel na další hranu (resp. dalšího následovníka) v pořadí*
node	arc *	successors	*spoják sousedů*
graph	unsigned int	size	*počet vrcholů*
	unsigned int	order	*počet hran*
	node *	nodes	*pole vrcholů*
heap	unsigned int	regularity	*max. počet potomků 1 uzlu v haldě*
	int *	indices	*pole s umístěním každého vrcholu grafu vzhledem haldě*
	int *	heap_arr	*samotná halda s indexy náležejících vrcholů*
	int	bottom	*konec haldy*
c_bit	unsigned int : 1	bit	*bit na matici příznaků*
llist_node	unsigned int	val	*hodnota uzlu spojáku*
	llist_node *	next	*ukazatel na další uzel ve spojáku*
queue	llist_node *	head	*začátek fronty (odtud se odchází)*
	llist_node*	tail	*konec fronty (sem se přichází)*

Životně důležitou strukturou je `graph`, v němž je vstupní graf uložen jako seznam následovníků.

Na druhou stranu `c_bit` slouží pro jednoduché (neboli šetřící pamět, poněvadž se jedná o pouhé samostatné bity) ukládání matic sousednosti.

Dijkstrův algoritmus pro hledání vzdáleností využívá k-regulární haldy, kde k je určeno v položce `regularity`.

Pro *Tarjanův algoritmus* na hledání silně souvislých komponent je využita zásobníková struktura `queue`.

3.2 Použité algoritmy

V této sekci si popíšeme algoritmy použité pro operace nad grafy. Jedná se *de facto* o jádro `GraphAlpha`, jež tvoří slavné algoritmy velikánů informatiky jakými jsou např. *Tarjan* či *Dijkstra*. Půjde spíše o „nošení dříví do lesa“, neboť tyto algoritmy jsou velmi známé a dobře prozkoumané.

3.2.1 Napojení knihovny na program

Knihovna `GraphAlpha` je psána v jazyku C a její integrování do programu je velice jednoduché. Soubory knihovny `GraphAlpha.h` a `GraphAlpha.c` stačí zkopírovat do složky s programem využívající této knihovny a do programu vložit následující řádek:

```
#include "GraphAlpha.h"
```

To nalinkuje hlavičky funkcí a datové struktury knihovny. Dál je nutné při kompilaci zkompilovat soubory knihovny a připojit je ke kompilovanému programu.[3]

3.2.2 Funkce `symMtrx`

Funkce má následující hlavičku:

```
double** symMtrx(graph* p_gr);
```

Ve stručnosti si popíšeme, co funkce dělá:

- ♯ Převede representaci se seznamem následovníků na „matici sousednosti“, kde však místo 1 a 0 jsou váhy hran a hodnota `DBL_MAX`.[4]
- ♯ Porovnává po složkách hodnoty této matice a hodnoty její transposice[5] a, pokud jsou na obou místech platné hodnoty,[6] vezme se v absolutní hodnotě ta větší. Je-li přítomna hrana jen jedním směrem, vytvoří se i identická hrana druhým směrem.
- ♯ Výsledek se vrací jako ukazatel na 2-D pole.

Dále hlavní program `main.c` zpracová toto pole tak, že ho za pomocí funkce `listConvert` převede na representaci se seznamem následovníků. Ten se následovně zobrazí procedurou `displayGraph` umístěné v `main.c`.

Časová složitost Převod na matici sice probere celý graf, ale vzhledem k nutnosti inicializace matice a její následné projetí kvůli symetrisaci musí

[3] Jako příklad tohoto může posloužit přiložený `Makefile`
[4] pro vrcholy nespojené hranou.
[5] Jinak řečeno u každé dvojice vrcholů zkoumá tam i zpět.
[6] tedy existují hrany tam i zpět

algoritmus vždy zpracovat celou matici. Nechť ve zbytku textu n značí počet vrcholů, m počet hran, $T(n)$ časovou a $S(n)$ paměťovou nárořnost. Pak $T(n) \in \Theta(n^2)$

Prostorová složitost Analogicky je vždy zapotřebí celá *matice sousednosti*. Tudíž podobně $S(n) \in \Theta(n^2)$

3.2.3 Funkce `dijkstra`

Funkce má následující hlavičku:

```
double* Dijkstra (int v, graph* p_gr);
```

Idea *Dijkstrova algoritmu* je následující:

Představme si, že u každého vrcholu máme budík. Na začátku jsou všechny nenastavené (tj. nastaveny na „∞“[7]), až na zadaný výchozí vrchol, který je nastavený na 0 (tj. zazvoní hned).

Dále algoritmus pracuje v cyklech.

Na začátku každého cyklu se probere vrchol, jemuž zazvoní budík nyní jako první. Jeho čas probuzení určí kýženou vzdálenost od startovního vrcholu. Dále může tento vrchol tzv. *zrelaxovat* hrany k sousedům, tedy p ařenastavit jim časy probuzení, pokud by příslušné budíky za doby určené vahami příslušných hran zazvonily dříve než mají aktuálně nastaveno.

Takto se pokračuje v cyklech dál, dokud stále nějaký budík tiká.

Je jasné, že *Dijkstrův algoritmus* není nic jiného než-li obyčejná *diskrétní simulace*.[8]

Časová složitost Klíčový je zde způsob výběru vrcholu s minimálním časem probuzení. K tomu je zapotřebí vybrat vhodnou datovou strukturu, která dokáže rychle provádět operaci *Insert*, *Decrease* a *ExtractMin*.

V řeči matematické, jsou-li T_I, T_D a T_E po řadě doby trvání operací *Insert*, *Decrease* a *ExtractMin*, pak časová složitost celého algoritmu činí $T(n) \in \mathcal{O}(n \times (T_I + T_E) + m \times T_D)$.

Pro tento účel je jako stvořená tzv. *halda* nebo ještě lépe její zobecněná verse tzv. *k-regulární haldy* s následujícími časy operací:

- ◇ $T_I \in \mathcal{O}(\log_k n)$
- ◇ $T_D \in \mathcal{O}(\log_k n)$
- ◇ $T_E \in \mathcal{O}(k \log_k n)$

[7] zde řešeno pomocí `DBL_MAX`

[8] Z tohoto důvodu *D. A.* zkolabuje na grafech se zápornými hranami, neb by se přes ně dalo vracet v čase a simulace by již nebyla simulací.

Po menších úpravách vyjde $T(n) \in \mathcal{O}((kn+m)\log_k n)$.

Je očividné, že se zvětšujícím se k logaritmus snižuje asymptotickou složitost. Zároveň se tím ale i zvyšuje multiplikativní konstanta. Ovšem asymptoticky nám nebude vadit, pokud k budeme zvyšovat až do takové míry, kdy $kn = m$,[9] neboť člen m nám stále dokáže přebít člen kn. Tedy stačí nastavit $k := \max\{2, \lfloor \frac{m}{n} \rfloor\}$[10]

Celkově vychází $T(n) \in \mathcal{O}(m \log_{\frac{m}{n}} n) = \mathcal{O}(\frac{m \log n}{\log m - \log n})$

Prostorová složitost Krom paměti pro uložení grafu ještě využíváme pole pro zaznamení vzdáleností a k-regulární haldy, které však zaberou každý vrchol nejvýše jednou, takže $S(n) \in \mathcal{O}(n)$, což je ovšem k nutnosti ukládat celý graf i s hranami pořád asymptoticky *okay*.

3.2.4 Funkce `FloydWarshall`

Tato funkce má následující podobu:

```
double** FloydWarshall(graph* p_gr);
```

Na tomto algoritmu je nádherně názorná síla *dynamického programování*. Označme si vrcholy libovolně indexy $1, 2, \ldots n$. Teď uvažme veličinu D^k_{ij} jako délku nejkratší cesty z i do j využívající uvnitř pouze vrcholy $1, 2, \ldots k$.

Nelze si nepovšimnout následujících pozorování:

$$D^0_{ij} = \begin{cases} w(i,j) & \text{iff } (i,j) \in E(G); \\ \infty & \text{jinak,} \end{cases}$$

kde $w(i,j)$ označuje váhu hrany ij,

$$D^n_{ij} = d(i,j),$$

kde $d(i,j)$ označuje vzdálenost od i do j, a nakonec důležitý rekursivní vztah

$$D^{k+1}_{ij} = \min\{D^k_{ij}, D^k_{i,k+1} + D^k_{k+1,i}\}.$$

Nyní přichází takový malý trik! Platí zároveň i tyto 2 vztahy:

$$D^k_{i,k+1} = D^{k+1}_{i,k+1},$$
$$D^k_{k+1,i} = D^{k+1}_{k+1,i}.$$

To je z toho důvodu, že vrchol $k+1$ je krajní, tudíž je použit takjakotak a je tedy zbytečné ho používat uprostřed cesty.[11] To nám ovšem značně

[9] Dokonce stačí $kn \in \mathcal{O}(m)$

[10] Musí být $k \geq 2$, aby se stále jednalo o haldu.

[11] Tedy nadvakrát!!

zjednoduší práci i paměť, páč stačí použít jedinou matici. Ač jsou v této matici smíchány v jediném cyklu hodnoty jak pro prvních k, tak pro prvních $k + 1$ vrcholů, můžeme si dovolit používat obě hodnoty, páč je to jedno dle dvou předchozích pozorování. Pro objasnění si lze algoritmus prohlédnout ve zdrojovém kódu.

Časová složitost Pro celkově n horních indexů vyplňujeme matici $n \times n$. Složitost vychází $T(n) \in \Theta(n^3)$.

Prostorová složitost Vždy si vystačíme s jedinou maticí $n \times n$. Složitost je $S(n) \in \Theta(n^2)$.

3.2.5 Funkce `Tarjan`

Tato funkce má tuto hlavičku:

```
void Tarjan(graph* p_gr);
```

Popisovat *Tarjanův algortimus* pro hledání *silně souvislých komponent* je v jistém smyslu zbytečné—je již podrobně (a doufám si tvrdit, že i mnohem srozumitelněji) popsán na mnoha jiných místech.[12] Takže jen opravdu stručně:

Algoritmus je ve své podstatě modifikací *DFS*[13] *algoritmu*. Kromě hodnoty *in*—času příchodu—řeší ještě tzv. hodnotu *low*. Ta je ve značné míře svázána s tzv. *kořeny SSK*.[14] To je vždy ten vrchol, jenž byl vzhledem ke své *SSK* navštíven jako první[15] a *low* je jeho hodnota, která je stejná v celé příslušné *SSK*.

Při průchodu grafem do hloubky se `Tarjan` snaží minimalisovat hodnotu *low* u každého vrcholu. Dojde-li se do již navštíveného vrcholu,[16] musí ležet v *SSK* společné se svým předchůdcem a může mu tím pádem snížit *low*, neb onen následovník může být sám kořenem *SSK*.

Dále se informace o *low* propagují zpět při *backtrace*, takže každý vrchol pozná, zda je či není kořenem.[17] Pokud ano, okamžitě se ze zásobníku odeberou vrcholy s vyšším *in*, poněvadž k těmto vrcholům lze dojít z kořene, a jelikož měli *low* nižší než *in*,[18] sami nemůžou být *kořenem*.

Navíc vrcholy nepřístupné z *kořene* se nám zde neobjeví, páč k těm jsme ani nemohli dojít aniž bychom nejprve neuzavřeli poslední *SSK*.

[12] Za všechny uveďme `http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm`

[13] *Depth-first search*—prohledávání do hloubky

[14] silně souvislých komponent

[15] Tedy má ve své *SSK* nejnižší hodnotu *in*.

[16] Tedy šlo se po zpětné hraně.

[17] Ten má očividně *low(root) == in(root)*

[18] *low* je nerostoucí funkce!

Časová složitost Jakožto modifikace *DFS* má algoritmus náročnost $T(n) \in \Theta(n+m)$. Problém by mohli činit testy navíc, které provádíme při ověření nenavštívenosti vrcholů či jejich přítomnosti v zásobníku. Toto vše lze však pomocí polí příznaků zvládat v konstantních časech, což se nám asymptoticky schová do Θ.

Prostorová složitost Co je potřeba...

- ∇ pole *low*
- ∇ pole *in*
- ∇ příznaková pole pro ověření navštívených vrcholů
- ∇ příznaková pole přítomnosti v zásobníku
- ∇ zásobník pro vrcholy
- ∇ zásobník pro rekursivní volání funkce *DFS*

To vše postačuje ve velikosti odpovídající počtu vrcholů, ergo paměti je třeba $S(n) \in \Theta(n)$.

3.2.6 Funkce `TSort`

Tato funkce má takovouto hlavičku:

```
int TSort(graph* p_gr, int* seq);
```

Principem tohoto algoritmu je postupné odtrhávání *výtoků*.[19] Pokud by totiž vedla hrana z aktuálního výtoku do nějakého již odtrženého vrcholu, ten by pak v době svého odtržení nemohl sám býti výtokem, poněvadž do něj vede hrana z původního uvažovaného výtoku. To je ovšem spor ↯

Nejprve se průchodem všemi vrcholy určí *vstupní stupně*[20] vrcholů. Poté umístíme do fronty všechny vrcholy s nulovým vstupním stupněm[21] a ty budeme postupně zařazovat do *topologického uspořádání*. U každého vrcholu budeme navíc jeho sousedům *dekrementovat*[22] vstupní stupně v nově vznikajícím grafu.

Dojdou-li všechny výtoky ještě před koncem, graf žádné *topologické uspořádání* nemá a není tudíž acyklický. Právě toto se vypíše.

Časová složitost Průchod grafem pro zjištění *vstupních stupňů* zabere (jako každý průchod grafem) $\Theta(n+m)$ času.

Každý vrchol se odtrhne nejvýše jednou a přes každou hranu se i nanejvýš jednou dekrementuje.

To činí $T(n) \in \Theta(n+m)$.

[19] vrcholů, do nichž nevedou žádné hrany
[20] počet hran končících ve vrcholu
[21] tj. *výtoky*
[22] snižovat o 1

Prostorová složitost Vždy pracujeme s datovými entitami pojímající informaci ke každému z vrcholu. K tomu ještě zásobník pro výtoky, který však maximálně bude pojímat všechny vrcholy.

Sumo sumárum $S(n) \in \Theta(n)$.

3.2.7 Funkce `transitiveClosure`

Funkce s následující hlavičkou:

```
c_bit** transitiveClosure(graph* p_gr);
```

Uzávěr lze získat mnoha způsoby. Je třeba jen spojit hranou vrcholy, mezi kterými existuje cesta.

Jedním způsobem je pustit na graf funkci `FloydWarshall` a spojit hranou vrcholy, které mají vzdálenost menší než ∞.

Ještě jednodušším způsobem jest menší úprava aktualizačního přiřazování uvnitř „trojcyklu“:

$$e_{ij}^{k+1} = e_{ij}^{k} \vee (e_{i,k+1}^{k} \wedge e_{k+1,i}^{k})$$

kde e_{ij}^{k} značí příznak přítomnost $i \to j$ cesty využívající prvních k vrcholů.

Časová složitost Jedná se jen o upravený *Floyd-Warshallův algoritmus* $\Rightarrow T(n) \in \Theta(n^3)$.

Prostorová složitost Stále jen upravený *Floyd-Warshallův algoritmus* $\Rightarrow S(n) \in \Theta(n^2)$.

3.2.8 Funkce `transitiveReduction`

Funkce s následující hlavičkou:

```
c_bit** transitiveReduction(graph* p_gr);
```

Algoritmus využívá následujícího pozorování—pro acyklický graf R platí

$$R^- = R \setminus (R \circ R^+),$$

kde R^-, R^+ značí po řadě *transitivní redukci* a *transitivní uzávěr*[23] uvažovaného grafu R.

To funguje z následujícího důvodu—$(R \circ R^+)$ jsou všechny dvojice vrcholů, mezi kterými vede cesta s alespoň jedním mezivrcholem. Po odebrání těchto „přebytečných hran“ zbydou jen dvojice vrcholů bezprostředně následující po sobě.

[23] bez smyček—hran do stejného vrcholu

To by však zkolabovalo u vrcholu „napojeného“ na cyklus[24] v grafu—odebrala by se totiž i hrana mezi vrcholem a cyklem, již potřebujeme.

Časová složitost Potřebujeme nejprve získat *transitivní uzávěr* za pomoci funkce `transitiveClosure`. Dále postupujeme dle výše uvedeného vzorce.

Zřejmě největší obtíže nám bude činit výpočet operace složení relací. Uvažme ale nejhorší případ—tedy úplný graf K_n. Jeho uzávěr má $\mathcal{O}(n^2)$ hran, každá lze složit s libovolným dalším z $\mathcal{O}(n)$ vrcholů. Tedy skládání se zvládne v čase $\mathcal{O}(n^3)$.

Množinový rozdíl už je *triviální* záležitost na $\Theta(n^2)$ času.

Celkově tedy $T(n) \in \Theta(n^3)$.

Prostorová složitost Pro vzorec využíváme jen 3 matice. Takže pro paměť platí $S(n) \in \Theta(n^2)$.

3.2.9 Další funkce ...

Mezi další funkce náleží takové „vychytávky“ jako

- $\bowtie$ převod mezi 2 používanými representacemi grafu
- $\bowtie$ zobrazení grafu
- $\bowtie$ vyčištění obrazovky
- $\bowtie$ ukončení programu

U těchto funkcí si autor dovoluje vynechat pasáže o časových a prostorových náročnostech, páč, jak se čtenář ze zdrojového textu sám přesvědčí, se jedná o pouhé triviality `:-)`

[24] Příklad takovéhoto grafu lze nalézt v `testing/reduction_cyclic.in`

Část III
Uživatelský manuál

Pro účely názorné demonstrace knihovny *GraphAlpha* byl sepsán obslužný program umístěný v souboru `main.c`. Ten je zcela separován od knihovních funkcí, tudíž operace nad grafy lze libovolně kombinovat (např. vytvořit neorientovanou versi transitivní redukce) či využívat je jakožto součást jiných programů.

4 Menu

Při spuštění obsluhy nás uvítá zpráva o aktuální versi ovládacího programu:

```
Welcome to GraphAlpha v1.3!
This is a demonstration program of GraphAlpha library.
```

Následuje dotaz na parametry vstupního grafu:

```
Enter count of nodes: 3
Enter count of arcs: 2
```

Program se poté zeptá na detaily jednotlivých hran:

```
1. node
 From:
 To:
 Weight:
2. node
 From:
 To:
 Weight:
```

Jedná se o následující údaje:

- $\propto$ *From:* výstupní vrchol, ze kterého zadávaná hrana vede
- $\propto$ *To:* vstupní vrchol, do kterého zadávaná hrana vede
- $\propto$ *Weight:* váha (popř. cena), zadávané hrany

Každá hrana obsahuje návěští s pořadovým číslem zadávané hrany.

Nakonec nabídne aplikace nabídku poskytovaných příkazů. Tyto lze spatřit na následující stránce ...

```
----------------------------------------------------------------
Choose option (enter the part in brackets)
(S)ym             - symmetrization
(D)ijkstra        - shortest path between node & other nodes
(F)loyd-Warshall - shortest path between every 2 nodes
tar(J)an          - strongly connected components
(T)sort           - topological ordering
(C)losure         - transitive closure
(R)eduction       - transitive reduction
l(I)st-convert    - from adjacency matrix to list of successors
(M)atrix-convert - from list of successors to adjacency matrix
c(L)ear           - clear screen (only for in *nix like OS)
Displa(Y)         - display the graph
(H)elp            - display this menu:)
(Q)uit            - quit the program
----------------------------------------------------------------
>>
```

Jak vidno, tuto nabídku lze kdykolit znova vyvolat pomocí příkazu `h`:[25]

```
>> h
```

5 Vstup a příkazy programu

5.1 Sym

Příkaz, jenž „zorientuje každou hranu na obě strany“, dokáže i zajistit převod orientovaného grafu na neorientovaný.

```
>> s
List of successor representation:
1. node -> 3(3.140000)
3. node -> 1(3.140000)
Matrix representation:
----    ----    3.14
----    ----    ----
3.14    ----    ----
```

Pozn. Vstup z `testing/sym2.in`

[25] Příkazy lze zadávat velkými i malými písmeny abecedy. Dokonce ani nevadí, když jsou za prvním písmenem další znaky, ty se až do konce řádku ignorují.

5.2 Dijkstra

Příkaz spuštění *Dijkstrova algoritmu* ze zadaného vrcholu.

```
>> d
Enter starting node of graph: 1
From 1.node to all nodes respectively:
0.000000 7.000000 9.000000 20.000000 20.000000 11.000000
```

Pozn. Vstup z `testing/dijkstra2.in`

5.3 Floyd-Warshall

Příkaz spuštění *Floyd-Warshallova algoritmu.*[26]

```
>> f
List of successor representation (i.e., transitive closure):
1. node -> 4(3.600000) 2(1.200000) 1(1.100000)
2. node -> 4(2.400000)
3. node -> 4(6.700000) 2(4.300000) 1(3.100000)
Matrix representation:
1.10    1.20    ----    3.60
----    0.00    ----    2.40
3.10    4.30    0.00    6.70
----    ----    ----    0.00
```

Pozn. Vstup z `testing/floyd_warshall.in`

5.4 Tarjan

Příkaz spuštění *Tarjanova algoritmu.*[27][28]

```
>> j
( 7 6 )
( 8 4 3 )
( 5 2 1 )
```

Pozn. Vstup z `testing/scc2.in`

[26] Mimochodem tento algoritmus je použit pro vytvoření grafu transitivního uzávěru v representaci se seznamem následovníků. Autor tak učinil z dvou prostých důvodů: zaprvé Floyd-Warshall podává lepší informaci o vahách mezi vrcholy jakožto jejich vzdálenosti; zadruhé algoritmus transitivního uzávěru je jen upravená verse Floyd-Warshallova algoritmu *:-)*

[27] Počínaje od vrcholu s indexem 1, neb hlavní procedura zkoumá nezpracované vrcholy v rostoucím pořadí indexů.

[28] Tarjanův algoritmus zároveň určuje reversní topologické uspořádání grafu komponent zadaného grafu.

5.5 TSort

Příkaz spuštění algoritmu pro nalezení *topologického uspořádání.*[29]

```
>> t
Topological ordering: 1 4 2 3 5
```

Pozn. Vstup z `testing/tsort.in`

5.6 Closure

Příkaz spuštění algoritmu pro vytvoření *transitivního uzávěru.*[30]

```
>> c
List of successor representation:
1. node -> 5(1.000000) 4(1.000000) 3(1.000000) 2(1.000000) 1(1.000000)
2. node -> 5(1.000000) 3(1.000000) 2(1.000000)
3. node -> 5(1.000000) 3(1.000000)
4. node -> 5(1.000000) 4(1.000000) 3(1.000000)
5. node -> 5(1.000000)
Matrix representation:
1 1 1 1 1
0 1 1 0 1
0 0 1 0 1
0 0 1 1 1
0 0 0 0 1
```

Pozn. Vstup z `testing/closure.in`

5.7 Reduction

Příkaz spuštění algoritmu pro vytvoření *transitivního redukce.*

Pozn. Tato implementace zvládá nalézt redukci **pouze** *v acyklických grafech. Autor z důvodů časové tísně a značné vyspělosti obecného algoritmu nebyl schopen vyřešit případ pro grafy s cykly a slibuje, že se do příště napraví :-)*

Výstup toho příkazu je k nalezení na další stránce ...

[29] V případě neexistence takového uspořádání (nastane právě v grafech s cykly) se vypíše `NOT AN ACYCLIC GRAPH!`

[30] Zde použitá implementace funkce vrací matici sousednosti nalezeného uzávěru. Ovšem knihovna poskytuje funkci `listConvertBit`, jež matici dokáže převést na representaci se seznamem následovníků. To je náležitě využito v `main.c` pro zobrazení obou možných representací, jak si lze prohlédnout v přiloženém výstupu.

```
>> r
List of successor representation:
1. node -> 4(1.000000) 2(1.000000)
2. node -> 3(1.000000)
3. node -> 5(1.000000)
4. node -> 3(1.000000)
Matrix representation:
0 1 0 1 0
0 0 1 0 0
0 0 0 0 1
0 0 1 0 0
0 0 0 0 0
```

Pozn. Vstup z `testing/reduction2.in`

```
>> r
Warning! Given graph is NOT acyclic.
The result of transitive reduction is ungranted!
List of successor representation:
1. node -> 2(1.000000)
2. node -> 3(1.000000)
3. node -> 1(1.000000)
4. node -> 5(1.000000)
5. node -> 6(1.000000)
6. node -> 4(1.000000)
Matrix representation:
0 1 0 0 0 0
0 0 1 0 0 0
1 0 0 0 0 0
0 0 0 0 1 0
0 0 0 0 0 1
0 0 0 1 0 0
```

Pozn. Vstup z `testing/reduction_cyclic.in`

5.8 List-convert

Příkaz spuštění pro převod *maticové* representace na representaci se *seznamem následovníků*.[31]

[31] Příkaz dokáže fungovat obecně. Pro účely jednoduchosti však *main.c* využívá funkci tak, že seznam následovníků nejprve převede na matici a ta se pomocí *list-convert* převede zpět. Samozřejmě je list-convert využit ještě u funkcí vracející maticové representací jakými jsou například `closure` či `reduction`.

```
>> i
Converting the already matrix-converted graph
  to list-of-sucessors representation...
1. node -> 4(0.200000) 3(0.600000) 2(0.200000)
2. node -> 3(0.300000)
3. node -> 5(0.500000)
4. node -> 5(0.800000) 3(0.200000)
```

Pozn. Vstup z `testing/list_convert.in`

5.9 Matrix-convert

Příkaz spuštění pro převod se *seznamem následovníků* representace na representaci *maticovou*.[32]

```
>> m

----    0.20    0.60    0.20    ----
----    ----    0.30    ----    ----
----    ----    ----    ----    0.50
----    ----    0.20    ----    0.80
----    ----    ----    ----    ----
```

Pozn. Vstup z `testing/matrix_convert.in`

5.10 Display

Příkaz pro zobrazení grafu v representaci se seznamem následovníků.
Formát:
<*index výchozího uzlu*>`.node ->` <*index příchozího uzlu*>`(`<*váha*>`)`

```
>> y
1. node -> 4(0.200000) 3(0.600000) 2(0.200000)
2. node -> 3(0.300000)
3. node -> 5(0.500000)
4. node -> 5(0.800000) 3(0.200000)
```

5.11 Clear

Příkaz na vyčištění obrazovky (v abstraktním slova smyslu, samozřejmě, jinak použijte suchý hadřík či látku `:-)` Funguje převážně na systémech *nixového typu, páč využívá systémové procedury `clear`.

```
>> l
```

[32]Příkaz dokáže fungovat obecně. Pro účely jednoduchosti však `main.c` využívá funkci tak, že seznam následovníků nejprve převede na matici a ta se pomocí `list-convert` převede zpět. Samozřejmě je list-convert využit ještě u funkcí vracející maticové representací jakými jsou například `closure či reduction`.

5.12 Quit

A tady končí naše cesta...

```
>> q
Bye bye...
```

Reference

[1] http://en.wikipedia.org/wiki/Dijkstra_algorithm

[2] http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm

[3] http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

[4] http://en.wikipedia.org/wiki/Topological_sorting

[5] http://en.wikipedia.org/wiki/Transitive_reduction#Graph_algorithms_for_transitive_reduction

[6] Doc. RNDr. Töpfer, Pavel. *Algoritmy a programovací techniky*, 2. vydání, Prometheus, Praha (2007). ISBN 978-80-7196-350-9.