6 21. a 28. dubna – Grafové algoritmy a programování

Grafy jsou poměrně typickými představiteli diskrétních matematických struktur. Výhodné jsou značnou univerzalitou, ze které plyne široká použitelnost. Teorii grafů jste sice probrali částečně na Diskrétní matematice, částečně bude na Kombinatorice a grafech, na Algoritmech a datových strukturách, Složitosti, Automatech a mnoha dalších kurzech. S využitím grafů často zkoumáme algoritmické vlastnosti nějakých problémů. A algoritmy jsou určeny k naprogramování. Proto si i zde povíme o grafech.

6.1 Definice a reprezentace grafů

Definici grafu znáte (a pro jistotu ji ještě vidíme na slidu). Jako každá správná struktura jde uspořádanou několikatici. V tomto případě dvojici (vrcholy, hrany). Grafy rozlišujeme orientované a neorientované. Nad těmi druhými byl kdysi výzkum zahájen, grafy orientované jsou výrazně novější. S tím souvisí i přístup k těmto strukturám. Výzkum neorientovaných grafů byl motivován řešením praktických problémů. Orientované grafy se objevily časem a začaly být zkoumány již od začátku systematicky. Pro množství problémů sice potřebujeme orientované grafy, ale zobecnění z neorientovaného případu je zcela přirozené, takže většinu času strávíme nad neorientovanými grafy.

Graf je potřeba nějak reprezentovat. Některé možnosti znáte (matici sousednosti – vrcholy proti vrcholům, matici incidence – vrcholy proti hranám). Jiné si dovedete představit, například seznam vrcholů, v němž jednotlivé vrcholy odkazují na seznam sousedů. Povšimněte si, že seznam sousedů je vlastně jen komprimovaná matice sousednosti. Mezi těmito přirozenými reprezentacemi lze převádět. Popis reprezentací (definice) najdete ve slidech (anebo v Kapitolách z diskrétní matematiky).

Graf můžeme reprezentovat i objektově: Třída `vrchol` reprezentuje vrchol, třída `hrana` reprezentuje hranu. Hrana má dva koncové vrcholy, vrchol může mít například seznam přilehlých hran. Tato reprezentace má výhodu, že ji lze zobecňovat. Někdy totiž máme graf ohodnocený (či olabelovaný). V takovém případě vrcholům či hranám přiřazujeme nějaké atributy (čísla, názvy, barvy,...). Atributů může být i více (například u toků v sítích, kde každá hrana má kapacitu, která jí smí protékat, a ještě aktuální tok, tedy dvě čísla). Objektová reprezentace se tedy dá přirozeně rozšířit (tak, že od vrcholu nebo hrany zdědíme a v synech přidáme atributy).

Chceme-li pracovat s grafy, potřebujeme nějaký základní instrukční repertoir. Jaký si ho navrhneme, takový bude. A může významně ovlivnit složitost implementovaných algoritmů! Abychom dokázali pracovat s grafem, můžeme například implementovat proměnnou `vrcholy,` funkci `najdi_sousedy` (která pro daný vrchol najde jeho sousedy), funkci `hrana`, která nám řekne, zda mezi zadanými dvěma vrcholy existuje hrana. Máme-li graf s atributy (vrcholů či hran), potřebujeme i nějaký interface ke zjištění hodnot těchto atributů.

Při praktickém programování se poměrně často stane, že nepoužijete přesně to, co vás učili o hodinách Diskrétní matematiky, ale vymyslíte si vlastní vhodnou modifikaci. Například máte-li graf s nezáporně ohodnocenými hranami, můžete jeho reprezentaci provést maticí sousednosti, ke které přidáte ještě matici téhož rozměru, která bude obsahovat údaje o ohodnocení dotyčné hrany. Takováto reprezentace by jistě fungovala. Nicméně víme-li, že ohodnocení hran budou nezáporná, co kdybychom graf reprezentovali maticí, kde nehrany budou značeny -1 a hrany budou značeny vahou? Poměrně očividně ušetříme matici sousednosti (místo dvou budeme mít jednu).

Chceme-li reprezentovat orientované grafy, můžeme postupovat podobně (matice sousednosti, matice incidence, seznam sousedů). Jen je třeba reprezentaci přiměřeně modifikovat. V případě matice sousednosti je třeba dát pozor na to, odkud kam hrana vede (matice přestane být symetrická). U matice incidence je taktéž třeba označit, odkud kam hrana vede (například start -1, cíl 1). Seznam sousedů není třeba modifikovat. :-)

6.2 Prohledávání grafu

Při práci s grafy je jedním z typických problémů hledání cesty. Ve slidech najdete definice sledu, tahu a cesty. Tah je sled bez opakování hran, cesta je sled bez opakování vrcholů. Kružnici lze definovat jako tah bez opakujících se vrcholů, u kterého startovní a cílový vrchol splývá.

Grafy mají jisté základní vlastnosti. Některé jsou souvislé (tedy z jakéhokoliv vrcholu se lze dostat do jakéhokoliv jiného), některé obsahují kružnice (tedy se lze z jednoho vrcholu dostat tamtéž, aniž bychom se vraceli). Dodatečné informace o grafech jsou užitečné (mohou různé problémy usnadnit). Různá tvrzení o grafech byla probrána (a dokázána) na Diskrétní matematice. Jedno takové nám umožňuje testovat souvislost grafu prohledáním do šířky. Které? Ano, to, které říká, že graf je souvislý, právě tehdy, když existuje vrchol, ze kterého se dostaneme do všech ostatních. Povšimněte si, že tentokrát je řeč jen o grafu. Implicitním grafem bude graf neorientovaný (a jednoduchý, tedy bez násobných hran a smyček).

Tvrzení vyučovaná v Diskrétní matematice sice typicky nebudou zkoušena, ale bude zkoušena jejich aplikace (tedy zda rozumíte, jak je lze použít a jak ne). Ačkoliv se tak nejspíše nestane, že bychom chtěli vidět důkaz lemmatu o trhání listů, ušatého lemmatu či tvrzení o souvislosti, je k praktické aplikaci nutné jejich pochopení (včetně důkladného chápání důkazů). Nyní stočíme svou pozornost k některým konkrétním problémům a algoritmům je řešícím:

Chceme-li tedy například vyšetřit souvislost grafu, můžeme použít prohledání do šířky. Anebo prohledání do hloubky (naznačeno na slidech). Tyto algoritmy fungují tak, jak jsme si říkali v zimním semestru. Tedy založíme frontu resp. zásobník, v každém případě datovou strukturu. Všechny vrcholy označíme jako dosud nenavštívené. Jak? *Nápověda: Vyrobíme si pole typu boolean indexované vrcholy – anebo aspoň tímtéž, čím indexujeme vrcholy a hodnota v poli nám bude říkat, zda je vrchol navštívený, nebo ne.* Startovní vrchol označíme jako navštívený a zařadíme do fronty (prostředky pro implementaci fronty v C#,

se už cvičily). Dokud je datová struktura neprázdná, opakujeme následující: Vytáhni první prvek, najdi dosud nenavštívené sousedy (cyklus přes řádek matice sousednosti a seznam navštívenosti vrcholů), přidej je do datové struktury a označ je jako navštívené.

Tyto algoritmy byly demonstrovány již v zimě pro speciální třídy grafů (vrcholy tvořila políčka šachovnice, hrany tvořily schopnosti pohybu zadané figurky). Všimněte si, že takto nám vznikaly grafy neorientované, protože všechny námi zkoumané figurky se pohybovaly symetricky (tedy když se dostaly odněkud, mohly se tam zase stejným způsobem vrátit). Kdybychom zkoumali pohyb pěšce, vznikl by graf orientovaný. O těchto algoritmech jste si (na Algoritmizaci) říkali, že umějí vyšetřit komponentu souvislosti a BFS umí v grafu najít nejkratší cestu.

Jak tyto algoritmy využijeme k vyšetřování souvislosti? Využijeme vlastní tvořivost. Víme, že algoritmy vyšetří komponentu souvislosti. Známe-li tedy velikost grafu (počet vrcholů), můžeme spočítat navštívené vrcholy. Bude-li jich tolik, jako všech vrcholů, graf je souvislý. Alternativou je inspekce seznamu navštívenosti (uvidíme-li tam false, graf souvislý není).

Mezi grafy mají prominentní postavení stromy (jak jste si všimli). Pro strom existuje mnoho ekvivalentních definic (a při programování je důležité vybrat tu správnou). Chcete-li vyšetřit, zda graf je strom, můžete využít toho, že strom je minimální souvislý, maximální bez kružnic, souvislý bez kružnic, souvislý se správným počtem hran (Eulerova formule), bez kružnic se správným počtem hran,...

Cvičení: Jak byste co nejsnáze vyšetřili souvislost grafu? Kterou z ekvivalentních definic byste využili a proč?

Chceme-li vyšetřovat komponenty, můžeme postupovat i úplně jiným algoritmem, a to takto: Každý vrchol obarvíme jinou barvou. Postupně bereme hrany. Pro každou hranu zjistíme, zda vede mezi vrcholy téže barvy (což u té první nebude splněno, protože každé dva vrcholy mají různé barvy). Vede-li hrana mezi vrcholy různých barev, v celém grafu jednu z těch barev změníme v tu druhou (důležité je, že ne jen u zkoumaného vrcholu, ale všude). Toto opakujeme, dokud nevyzkoušíme všechny hrany. Po proběhnutí tohoto algoritmu nám zbývající barvy označují komponenty souvislosti.

Jak při programování reprezentovat barvy? Nejlepší barvou je přirozené číslo. Jak reprezentovat obarvení vrcholů? Přeci polem (integerů) indexovaným stejně jako vrcholy. Údaj v poli bude určovat barvu dotyčného vrcholu. Složitost algoritmu je zřejmě O(mn). Pro každou hranu provedeme přebarvování. Vrcholů je n, hran m. Přebarvování provedeme průchodem všech vrcholů. Získáme-li další hranu v konstantním čase (tedy například máme-li seznam hran), náš odhad platí. V opačném případě je třeba odhad náležitě upravit!

Poznámka: Tento algoritmus jste možná již potkali pod krycím označením Union Find, kde se řeší, jak vybírat přebarvované vrcholy. Přebarvujeme-li menší komponentu, získáme výrazně lepší složitost (Algoritmy a datové struktury).

Souvislost (či komponenty) grafu můžete ovšem vyšetřovat mnoha způsoby. Výhodnost se bude lišit podle toho, co o grafu víme.

6.3 Kružnice

Hledáme-li kružnici, můžeme postupovat podobně. Tedy můžeme buďto opět provést BFS či DFS a sledovat, zda se někdy vrátíme vrcholu, ve kterém jsme již byli. Tento algoritmus má nepříjemnou vlastnost, že je nutné spustit jej pro každou komponentu zvlášť. Další nepříjemnou vlastností je, že hrany, po kterých jsme již prošli, musíme zrušit (jinak bychom se po již jednou prošlé hraně vrátili zpátky a triviálně bychom hlásili nalezenou kružnici).

Tento problém lze však opět řešit přebarvovacím algoritmem vyšetřujícím komponenty. Opět přiřadíme každému vrcholu pro začátek jinou barvu. Opět budeme brát hranu za hranou a sledovat, zda vede mezi komponentami. Spojuje-li hrana dvě komponenty, opět barvu nové komponenty sjednotíme (tedy jednu přebarvíme na druhou). Tentokrát se ovšem zastavíme (kromě varianty, že dojdou hrany) také v případě, že hrana vede mezi vrcholy též barvy. Tato hrana totiž uzavírá kružnici. Oproti minulému algoritmu se tak rovnou dozvíme, ve které komponentě kružnice je (nemusíme algoritmus spouštět opakovaně pro různé komponenty).

6.4 Topologické uspořádání

Problém si napřed takto motivujeme: Vlastníme továrnu vyrábějící náročný výrobek. Jelikož přišla krize (kvůli SARS verze 2.0), všechny pracovníky jsme propustili a výrobek chceme vyrobit sami. Jelikož jsme nedůvěřiví, musíme u každého úkonu, který při výrobě děláme, být osobně přítomni. Výrobu máme popsánu orientovaným grafem. Vrcholy odpovídají jednotlivým úkonům (které je nutné udělat) a orientované hrany říkají, které dvojice úkonů na sobě navzájem závisejí (například nemůžeme šaty vyžehlit dřív, než je ušijeme). Topologické uspořádání je takové uspořádání vrcholů orientovaného grafu, ve kterém jsou všechny hrany orientovány zleva doprava (tedy vrcholy rozmístíme na vodorovnou osu tak, aby žádná šipka nevedla zprava doleva). Problém tedy očividně odpovídá motivaci (vykonáme-li ve správném pořadí všechny úkony, vyrobíme kýžený výrobek).

Tento problém je zvláštní tím, že je řešitelný nápadně snadným algoritmem (s mírně netriviálním důkazem):

Topologické uspořádání vrcholů generujeme postupně takto: Dokud to jde, odebírej vrcholy se vstupním stupněm 0 (a dávej je v tomtéž pořadí na výstup). Tento algoritmus najde topologické uspořádání, kdykoliv existuje. To dokážeme asi takto: Pokud se podaří takto rozebrat celý graf, našli jsme topologické uspořádání. Nyní zbývá ukázat, že kdykoliv algoritmus selže, topologické uspořádání neexistuje. Právě to nyní uděláme. Ukážeme, že pokud algoritmus selže, graf obsahuje orientovanou kružnici. Orientovaná kružnice je očividně překážkou nalezení topologického uspořádání.

Pokud algoritmus nezafunguje, víme, že do každého vrcholu vede aspoň jedna hrana. Vypravíme se tedy proti směru těchto hran (vede-li jich více, vyjdeme po kterékoliv z nich). Jelikož všechny naše grafy jsou konečné, po konečně mnoha krocích se vrátíme do vrcholu, v němž jsme už byli (a našli jsme orientovaný

cyklus).

Graf mající topologické uspořádání označujeme jako DAG (directed acyclic graph).

6.5 Nejkratší cesta

Jedním z typických problémů je hledání nejkratší cesty (například z přednášky domů – nebo z domova do nejbližšího obchodu s potravinami, abyste věděli, kam máte povoleno jít nakoupit). Tento problém již částečně umíme řešit. Podmínkou je, aby graf byl neohodnocený (tedy aby všechny hrany měly délku 1). Povšimněte si, že nyní přirozeným způsobem začínáme pracovat s atributem hrany (v podobě délky).

Teoreticky užitečný algoritmus se opírá o to, že mocníme-li matici sousednosti, hodnoty k-té mocnině nám určují pro každou dvojici vrcholů počet sledů délky k. Jednou možností by tedy bylo mocnění matice sousednosti a sledování, kdy se na správném místě objeví nenula. Tento algoritmus je však pro typické využití poměrně těžkopádný a funguje jen v případě, že graf má všechny hrany jednotkové délky).

Pokud hrany nemají stejnou délku, zkusíme provést následující úvahu: Již od časů úlohy o Théseovi v Labyrintu (který vyplavoval Minotaura) se nám osvědčilo (při návrhu BFS) zalévat graf vodou (protože voda teče, kam chce sama – všemi směry včetně směru nejkratší cesty). Tuto úvahu můžeme diskretizovat (voda se těžko implementuje). Ze startu vypravíme mravence, kteří se valí jak velká voda. Mravenec je zvíře pracovité (pohybuje se stále stejnou rychlostí) a umí se neomezeně množit (dojde-li na křižovatku, najednou se objeví dostatek brabenců, aby bylo možno pokračovat všemi možnými směry). Navíc jsme před časem měli diskrétní simulaci. Tak co kdybychom zkusili problém hledání nejkratší cesty vyřešit diskrétní simulací mravenců lezoucích ze startu? Nápad je to tak výborný, že nám z něj vyjde Dijkstrův algoritmus:

Dijkstrův algoritmus vypadá stejně jako DFS či BFS, jen místo zásobníku, resp. fronty použijeme prioritní frontu. Prioritní fronta je datová struktura, ve které se nám prvky řadí podle nějakého atributu. V našem případě je budeme do prioritní fronty dávat vrcholy a řadit je podle času, kdy do nich doleze první brabenec. Všimněte si, že provedeme-li tento algoritmus, budeme diskrétně simulovat lezení mravenců (tedy událost nastane, doleze-li brabenec do nějakého vrcholu). Dojde-li mravenec do nějakého vrcholu, pomnoží se dostatečně, aby bylo možno pokračovat ke všem sousedním vrcholům. Technicky se algoritmus implementuje tak, že sledujeme, kteří mravenci do vrcholu dolezou jako první. Stane-li se, že do již rozvrženého vrcholu přijdou rychleji mravenci odjinud, událost "příchod mravenců"se dotyčnému vrcholu přerozvrhne.

Všimněte si, že Dijkstrův algoritmus je váženou verzí BFS (nastavíme-li délky všech hran na 1, implementací Dijkstrova algoritmu získáme BFS). Důkaz korektnosti algoritmu je na Algoritmech a datových strukturách. Pro účely důkazu vrcholy roztřídíme na dosud nezjištěné, rozvržené a navštívené. Nezjištěné jsou vrcholy, do kterých ještě mravenci nevyrazili. Rozvržené jsou ty, do kterých mravenci ještě nepřišli, ale již jsou na cestě. Navštívené vrcholy jsou

ty, do kterých se již dostali. Důkaz se opírá o invariant, že po každé fázi (kde jedna fáze zahrnuje přichod mravenců do jednoho vrcholu) máme nalezenu nejkratší cestu ze startu do všech vrcholů, přičemž tyto cesty smějí využívat pouze již navštívené vrcholy. Jsou-li všechny vrcholy navštívené, jako důsledek tohoto invariantu máme nejkratší cestu. Invariant se dokazuje indukcí (po prvním kroku očividně platí, přidáme-li další navštívený vrchol, nejkratší cesta buď to tento vrchol nepoužívala $\Rightarrow$ již jsme ji měli, nebo použila a pak nahlédneme, že do cílového vrcholu vedla již přímo přes hranu (protože do ostatních již navštívených vrcholů, přes které by mohla jít, bychom se dokázali dostat rychleji jinudy).

Složitost Dijkstrova algoritmu závisí na použitých datových strukturách. Vidíme, že algoritmus sestává z n fází (v každé fázi navštívíme jeden vrchol a vrcholů je n). Jak dlouho bude dotyčná fáze trvat, závisí na implementačních detailech. Budeme-li například sousedy hledat prohledáním matice sousednosti, potrvá nám vyšetření sousedů n. Každého souseda můžeme buď to přidat do fronty, nebo přerozvrhnout. Sousedů může být lineárně mnoho (pro lineárně mnoho vrcholů) a ještě ke všemu může být v prioritní frontě lineárně mnoho dalších vrcholů, které je potřeba přeskádat. Užijeme-li tedy naivní implementaci, může být délka jedné fáze kvadratická (vůči počtu vrcholů). Celkově by tak algoritmus byl kubický. Budeme-li mít seznam sousedů (pro každý vrchol), bez ohledu na počet vrcholů budeme prioritní frontou manipulovat $c'm$krát (protože každá hrana má jen dva konce). Ve frontě lze vždy předjíždět nejvýše $c''n$ vrcholů, složitost by tedy byla cmn pro nějakou konstantu c. Algoritmus lze výrazně vylepšit, reprezentujeme-li prioritní frontu něčím jiným nežli spojovým seznamem. Například haldou (rozpomeňte se na definici haldy, měli-li jste jich více, pak binární leftist haldu). Všimněte si, že s prioritní frontou potřebujeme provádět operace *Insert, ExtractMin* a *DecreaseKey*, které jste se (jako z udělání) učili právě nad haldami. Použijete-li haldu, přeskupování prioritní fronty pořídíte v logaritmickém čase (namísto lineárního) a složitost by tak byla $O(m \log n)$.

Nyní uděláme menší krok stranou. Podumejme nad úlohou o králi na šachovnici. Jedno řešení operovalo s tím, že si vytvoříme frontu a spustíme BFS (tak, jak jsme se učili). Někteří však úlohu řešili bez fronty tak, že šachovnici opakovaně prohlíželi a z každého již dosaženého políčka zkoušeli, zda nenajdou dalším tahem kratší (nebo aspoň nějakou) cestu do sousedních polí. I pro hledání nejkratší cesty v nezáporně ohodnoceném grafu (či v tomto případě dokonce v grafu bez uzavřeného záporného sledu - tedy sledu, který začíná a končí tamtéž a celkově má zápornou délku) můžeme provést podobnou úvahu. Tím získáme Bellman-Fordův algoritmus:

Tento algoritmus se také pokouší cestu hledat postupně (napřed první správnou hranu, potom druhou správnou hranu,...), ale oproti Dijkstrovu algoritmu nespekuluje na to, ze kterého vrcholu máme pokračovat nyní. Když nevíme, ze kterého vrcholu pokračovat, zkusíme to ze všech. Podobně jako jsme zkoušeli v případě krále na šachovnici. Algoritmus tedy bude postupovat takto:

Všem vrcholům nastav vzdálenost nekonečno. Startovnímu vrcholu nastav vzdálenost 0 a $n-1$krát opakuj: Pro každý vrchol v vezmi jeho sousedy a zjisti,

zda jejich vzdálenost není větší než vzdálenost do v + délka hrany z v do něj. Pokud ano (vzdálenost v + délka hrany z v je menší), vzdálenost dotyčného vrcholu nastav (na vzdálenost v + délku hrany z v).

Ačkoliv, když si zkusíme algoritmus představit (tedy vizualizovat) uvidíme něco velmi podobného jako u algoritmu Dijkstrova, myšlenky jsou nyní úplně jiné. Tento algoritmus nefunguje proto, že v i-té fázi najde cestu přes i vybraných vrcholů, ale proto, že v $i$té fázi najde všechny nejkratší cesty využívající nejvýše i hran (rozmyslete si, proč je to pravda). Jelikož cesta v grafu na n vrcholech může sestávat z nejvýše $n-1$ hran, po tomto počtu fází máme bezpečně nalezené nejkratší cesty ze startu do všech vrcholů. Složitost tohoto algoritmu je zjevně kubická (vůči počtu vrcholů).

6.5.1 Floyd – Warshallův algoritmus

Chceme-li vyšetřit nejkratší cesty mezi všemi dvojicemi vrcholů grafu (alias all-pairs-shortest-paths čili APSP), byly by výše uvedené algoritmy těžkopádné. V tomto případě můžeme použít algoritmus jiný (opírající se opět o úplně jiné myšlenky). Abychom na to nezapomněli, stále předpokládáme, že hrany mají nezápornou délku.

Tentokrát se vrátíme k poznatku z počátku kapitoly o mocnění matice sousednosti a zkusíme jej vhodně opracovat. Taktéž použijeme myšlenku z Bellman-Fordova algoritmu, že cesta délky $k+1$ vznikne z cesty délky k přidáním poslední hrany. Tento poznatek ještě vylepšíme zjištěním, že cesta o nejvýše $2k$ hranách vznikne slepením dvou cest délky nejvýše k.

Začneme s maticí sousednosti. V té máme délky nejkratších cest využívajících nejvýše 1 hranu. Z této matice spočítáme matici délek nejkratších cest využívajících nejvýše 2 hrany a výsledky zapíšeme do matice (zápis algoritmu najdete ve slidech). Máme-li matici délek nejkratších cest využívajících nejvýše 2 hrany, jak zjistíme délky nejkratších cest využívajících nejvýše 4 hrany? Stejně. Tento postup opakujeme, dokud nezjistíme délky nejkratších cest využívajících $n-1$ čí více hran. Přepočítávat matice tedy budeme logaritmicky-krát. Jeden přepočet proběhne v kubickém čase (pro startovní a cílový vrchol zkusíme všechny možné vrcholy mezilehlé).

Všimněte si, že tento algoritmus je představitelem algoritmů z famílie dynamického programování. My jsme si popsali jen výsledek, ke kterému se lze dobrat rekurzí. Tato rekurze by se pokoušela nalézt nejkratší cestu (pro každý pár vrcholů) o maximální délce $n-1$ tak, že by zkoušela slepit dvě nejkratší cesty délky $\lceil \frac{n-1}{2} \rceil$ přes všechny možné mezilehlé vrcholy. Jelikož cesta z u do v délky nejvýše k je stále stejná, můžeme výsledky výpočtu nacachovat (do třírozměrné cache). Pak si ovšem všimneme, že tato cache se vyplňuje postupně pro rostoucí k a že v následujícím kroku potřebujeme pouze výsledky z minulého kroku (takže můžeme třírozměrnou cache redukovat na dvě cache dvourozměrné, tedy na dvě matice).

6.6 Minimální kostra

Motivace: Chceme zavést elektrický proud do vybraných obcí (takovéto problémy se řešily přibližně před 100 lety, dnes chceme zavést například drátové připojení k Internetu). Chceme, aby celková délka drátů byla co nejkratší. O elektrickém proudu všichni víme, že připojíme-li vodič k vodiči, máme v novém vodiči (přibližně) totéž napětí (jako bylo v původním vodiči). Není tedy třeba vést elektrický proud od elektrárny co nejkratší cestou.

Z tohoto problému vznikne graf, jehož vrcholy jsou obce a hrany určují možnosti, kudy vést elektrické vedení. Poměrně očividně hrany nebudou všechny stejně dlouhé (to by problém trivializovalo). Tento problém znáte jako problém minimální kostry. Zabývali se jím i významní čeští matematici, jako prof. Borůvka (který jím řešil elektrifikaci na jižní Moravě) či náš někdejší děkan, prof. Jarník (a oba problém vyřešili k optimu polynomiálním algoritmem). My si ovšem ukážeme jiný algoritmus, a to Primův:

Hrany setřídíme vzestupně podle délky a opakujeme: Začni s prázdným grafem (se všemi vrcholy a žádnou hranou). Každou hranu (v pořadí rostoucí délky) zkus do grafu přidat. Způsobí-li cyklus, odstraň ji. Jinak ji tam ponechej.

Tento algoritmus nalezne minimální kostru. Důkaz je lehce trikový a postupuje sporem. Předpokládejme, že existuje ještě optimálnější kostra.[1] Na tuto kostru budeme mít subtilní požadavek, že mezi všemi optimálními kostrami vybereme takovou, která se co nejvíce podobá Kruskalovské kostře (co do počtu hran). Hrany kostry nalezené Kruskalovým algoritmem označme $e_1, e_2, ...e_{n-1}$ v pořadí rostoucích vah, hrany ještě optimálnější kostry označme $f_1, f_2, ...f_{n-1}$ taktéž v pořadí rostoucích vah a ještě k tomu tak, aby se hrany u začátku co nejdéle shodovali s Kruskalovskými hranami. Nyní ukážeme, že liší-li se tyto kostry, najdeme s jejich pomocí už úplně nejoptimálnější kostru (což bude spor). Nechť i je minimální index takový, že $\forall_{j<i} e_j = f_j$ a současně $e_i \neq f_i$. Vezměme tu ještě optimálnější kostru a přidejme k ní e_i. Tím získáme cyklus (dle jedné z definic stromu, kterým kostra je). Tento cyklus odstraníme odebráním některé hrany f_k pro $k \geq i$. Takto (očividně) získáme buďto kostru ještě optimálnější (nežli ta ještě optimálnější, která měla být optimální, což je spor), anebo kostru stejné váhy, která se ale shoduje s Kruskalovskou na více prvcích (což je spor s výše uvedeným subtilním předpokladem). Zbývá drobnost: Proč se kružnice účastní i hrana f_k pro $k \geq i$? Protože v Kruskalovské kostře žádná kružnice není a začátky obou koster se shodují.

Nyní přichází obvyklá otázka: Jak algoritmus implementovat? Zatímco popis algoritmu je až magický, implementace je až směšně jednoduchá: Setřídíme hrany (vzestupně podle délky). Ve druhé fázi na tyto hrany spustíme algoritmus vyšetřování komponent modifikovaný takto: Všem vrcholům přiřaď různé barvy. Opakuj pro každou hranu: Vede-li hrana mezi vrcholy různých barev, přidej ji do kostry a sluč příslušné komponenty. To je vše (pro jistotu můžeme ještě doplnit: "Vede-li hrana mezi vrcholy téže barvy, nepřidávej ji").

Ostatní algoritmy hledání minimální kostry taktéž zkoušejí v nějakém pořadí

[1] Slovo *optimus* je třetím stupněm přídavného jména *bonus*. Výraz *ještě optimálnější* by tedy byl již čtvrtým stupněm a v tomto kontextu odkazuje k tomu, že je to nesmysl.

přidávat hrany mezi komponentami a působí zajímavým dojmem, že všechny fungují. Skutečností však je, že kostry tvoří *matroid*, o čemž se budete učit ve vyšších ročnících. Stručně řečeno, matroidy jsou struktury, na kterých fungují hladové algoritmy. Hladový algoritmus (jak již jistě víte z jiných kurzů) je takový, který se pokouší v každém okamžiku "urafnout co nejvíce", v tomto případě přidat co nejméně. Jak hledání nejkratší cesty Dijkstrovým algoritmem tak minimální kostra (tam konkrétně všechny algoritmy) patří do familie hladových algoritmů.

6.7 Zobecnění

V sekci pojednávající o cestách jsme vždy měli předpoklad, že graf musí být ohodnocen nezáporně. Tento předpoklad nám u koster chyběl. Je to tak správně, nebo je to tak špatně? A dá se s tím něco dělat?

Takto vypadají přirozené otázky týkající se problémů, které zkoumáme. Odpovědi jsou asi tyto: Pro hledání koster nepotřebujeme omezení na nezápornou délku hran. Algoritmus funguje taktéž (rozmyslete si proč – tedy projděte si důkaz). Chceme-li hledat nejkratší cestu v obecně (tedy potenciálně záporně) ohodnoceném grafu, problém je NP-těžký (což má důsledek, že dosud není znám polynomiální algoritmus a jeden z problémů milénia, P- a NP-hypotéza operuje s tím, zda existuje). Za nejkratší cestou v obecně ohodnoceném grafu se totiž schovává problém nejdelší cesty v (neohodnoceném) grafu (který je těžký). Nastavíme-li délku všech hran na -1 a hledáme-li nejkratší cestu, nejkratší cesta bude přesně ta, která vede přes co nejvíce hran.

Jaký je tedy rozdíl mezi cestami a kostrami? Proč pro kostry algoritmus funguje i při záporném ohodnocení, kdežto pro cesty ne? Odpověď již padla, protože kostry (potažmo podkostry) tvoří matroid. To pro cesty neplatí. Jednou z viditelných charakteristik je, že všechny kostry (pevně daného grafu - a dokonce jakéhokoliv grafu o n vrcholech) mají stejný počet hran $(n-1)$. Přes kolik hran vede nejkratší cesta, bez bližší znalosti grafu říct nemůžeme. Proto není problém najít i nejtěžší kostru. Najít nejdelší cestu však problém je. Krátkých cest je málo, dlouhých cest je hodně. Krátkost se způsobuje využitím pár krátkých hran, dlouhost se dá vyvolat buďto prolezením mnoha kratších hran, nebo prolezením několika dlouhých. A pro tu první variantu je možností mnoho.

6.8 Další grafově optimalizační problémy

Grafy jsou velmi významnou abstraktní (kombinatorickou) strukturou, která umožňuje zkoumat plno zajímavých věcí. Například vrcholy grafu mohou reprezentovat jednotlivé lidi. Dva lidé se vždy buďto znají, nebo ne. V takovémto grafu můžeme zkusit najít co nejvíce lidí, kteří se navzájem zná. Tomuto problému říkáme Problém maximální kliky (a terminologie je opravdu motivována co největším spolčením). Nebo nás naopak může zajímat, jaký je maximální počet lidí takových, že se žádná dvojice navzájem nezná. A máme problém maximální

nezávislé množiny. Očividně tyto problémy jsou komplementární (tedy řešení jednoho je řešením druhého v komplementu grafu).

Dalšími problémy s grafy na pohled nesouvisejícími, ale které si můžeme předvést na figurkách na šachovnici, jsou dominance a nezávislost. Máme-li šachovnici a zadané figurky, dominancí těchto figur na zadané šachovnici nazveme takové rozmístění, kdy figurky v minimálním počtu ohrožují celou šachovnici. Tedy na každém políčku buďto dotyčná figura stojí, nebo se tam nějaká může (jedním tahem) pohnout. Duálním problémem je nezávislost. V tomto případě hledáme maximální počet figurek, které lze rozmístit (na šachovnici) tak, aby se navzájem neohrožovaly.

Jak jsme již zjistili, úlohy o figurkách na šachovnici jsou jen zamaskované grafově optimalizační problémy. Co se tedy stane, zkusíme-li úlohy z minulého odstavce interpretovat v grafovém kontextu? V prvním případě budeme chtít najít minimální počet vrcholů, od které je jakýkoliv vrchol ve vzdálenosti nejvýše 1 (tedy každý vrchol je buďto v dominující množině, nebo s vrcholem dominující množiny sousedí). Tomuto problému říkáme (grafová) dominance. A ta druhá úloha (kdy chceme rozmístit co nejvíce figurek na šachovnici)? Ta už tu přeci byla. To je maximální nezávislá množina alias nezávislost.

Dalším problémem, který bychom mohli zkoumat, je, zda graf obsahuje kružnici obsahující všechny vrcholy (zvanou Hamiltonská kružnice). Anebo můžeme chtít vrcholy (či hrany) grafu obarvit tak, aby žádná dvojice sousedících vrcholů (či k sobě přilehlých hran) nebyla jednobarevná (tedy aby dva příslušné prvky měly dvě různé barvy). Tomu říkáme problém barevnosti, resp. hranové barevnosti.

Všechny v této sekci popsané problémy mají zvláštní vlastnost, že jsou těžké. Konkrétně NP-těžké, tedy v obecném případě nevíme, zda existuje polynomiální algoritmus je řešící. Podotkněme ještě, že některé omezené varianty těchto problémů jsou řešitelné v polynomiálním čase. Například pro bipartitní grafy je problém maximální kliky řešitelný dokonce v konstantním čase umíme-li v konstantním čase přistoupit k nějaké hraně nebo zjistit jejich počet (je-li v bipartitním grafu aspoň jedna hrana, má kliku velikosti 2, jinak 1). Stejně můžeme posoudit barevnost bipartitního grafu (ta je nejvýše 2 a graf je obarvitelný jednou barvou právě tehdy, když nemá žádnou hranu). Snadné mohou být i některé varianty těchto problémů. Například barevnost je těžká od tří barev. Ptáme-li se, zda lze graf obarvit dvěma barvami, jde o problém ekvivalentní bipartitnosti. To už je ale jiný příběh. Pro nás je podstatné, že jde o grafově optimalizační problémy, které mají reálné aplikace a může se nám stát, že budeme potřebovat naprogramovat jejich řešení. U těchto problémů však (pokud o grafu skutečně nic nevíme) nepomáhá příliš jiných věcí, nežli stará dobrá rekurze. Postačí-li nám však přibližná hodnota, může se stát, že problém je aproximovatelný. Například barevnost grafu je evidentně shora omezena maximálním stupněm + 1. Máme-li $k+1$ barev a každý vrchol má nejvýše k sousedů, můžeme jej obarvit takto: Procházíme postupně vrcholy a sledujeme, která barva na jeho sousedech chybí. Z těch chybějících vybereme libovolnou. Tento algoritmus pro daný graf vždy nalezne obarvení.