zda potřebujeme dvakrát nebo třikrát víc paměti nebo instrukcí procesoru, než je množství vstupních dat. Absolutní konstanty a koeficienty úměrnosti prostě zanedbáváme. Ba ještě víc: má-li výpočtová složitost tvar polynomu (např. $5 \cdot N^3 + 4 \cdot N^2 + 7$), zajímá nás jenom nejrychleji rostoucí člen (tj. N^3), ostatní členy zanedbáme. To má ovšem za následek, že takto vyjádřená výpočtová složitost platí až pro dostatečně velké hodnoty N.

Na obr. 1.2 vidíme, že např. lineární funkce $10 \cdot N$ roste zpočátku rychleji než kvadratická funkce N^2, teprve až pro velká N roste rychleji funkce kvadratická. Proto se takto vyjádřená výpočtová složitost někdy také nazývá složitostí asymptotickou.[26]

V praxi nás zajímají zejména tyto výpočtové složitosti:

konstantní	C, 1 apod.	např. přístup k prvku pole
logaritmická	$\log N$	např. hledání prvku ve vyhledávacím stromě s N vrcholy
lineární	N	např. prohledání spojového seznamu s N členy
lineárně logaritmická	$N \cdot \log N$	např. seřazení pole heapsortem
kvadratická	N^2	např. cyklus v cyklu
exponenciální	k^N	např. hledání cesty dlouhé N hran ve stromě
superexponenciální	$\gg k^N$	např. Ackermannova funkce

Zhruba můžeme odhadnout, že algoritmy s menší složitostí než exponenciální můžeme běžně používat, exponenciální algoritmy používáme jen po dobré úvaze v omezeném rozsahu N, kdežto superexponenciální algoritmy bývají úplně nepoužitelné. Příklad takového nepoužitelně složitého algoritmu je tzv. *Ackermannova funkce,* kterou

[26] To znamená, že jsme vyjádřili hranici výpočtové složitosti, ke které se skutečná složitost přibližuje tím blíž, čím větší je N.

vymyslel Wilhelm Ackermann[27] v roce 1928, právě aby demonstroval nezvládnutelnou výpočtovou složitost. Formulaci Ackermannovy funkce pak ještě vylepšila Rózsa Péterová[28] takto:

$A(m, n) =$
- když $m = 0$, tak $n + 1$
- když $(m > 0) \wedge (n = 0)$, tak $A(m - 1, 1)$
- když $(m > 0) \wedge (n > 0)$, tak $A(m - 1, A(m, n - 1))$

Výpočet této funkce vyžaduje i pro malé hodnoty argumentů m, n obrovský počet operací a dosahuje nepředstavitelně velkých hodnot. Např. $A(4, 3)$ je větší než počet elementárních částic hmoty v kosmu. Zatímco výpočet pro hodnoty argumentů 1 a 2 dává výsledek téměř okamžitě i na pomalém počítači, výsledku pro hodnoty argumentů 4 nebo víc se zaručeně nedočkáme, přestože i tento výpočet musí nutně jednou skončit – je jen škoda, že se toho nedožijeme ani my, ani počítač, na kterém výpočet běží.

V praxi nás zpravidla nezajímá výpočtová složitost kteréhokoli nebo každého výpočtu, ale jen některých typických výpočtů. Zajímavý bývá výpočet náročný **nejméně, nejvíce** nebo **průměrně.** Zejména náročnost nejhoršího případu bývá kritická – může být důvodem, proč se některý pěkný algoritmus nedá prakticky použít.

1.2.2 Formalizace výpočtové složitosti

Vidíme, že výpočtová složitost může rozhodovat o použitelnosti nebo nepoužitelnosti algoritmu – je samozřejmě lákavé zavrhnout špatný algoritmus včas, dřív, než vložíme desítky nebo stovky hodin práce do jeho naprogramování. S výpočtovou složitostí je ovšem potíž.

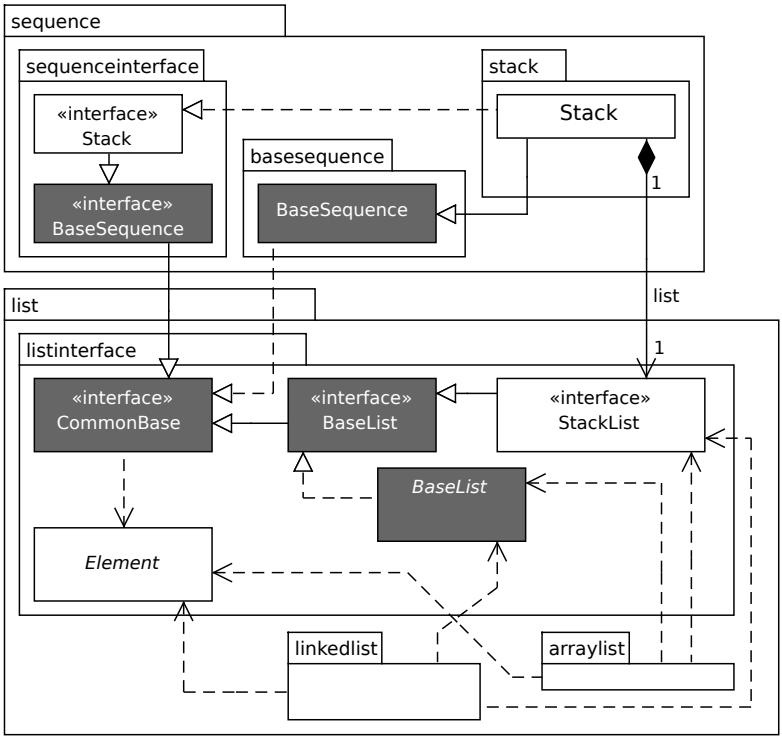
[27] Wilhelm Ackermann (1896–1962) je německý matematik, žák a spolupracovník Davida Hilberta

[28] Péter Rózsa (1905–1977) je maďarská matematička, která ve 30. letech 20. století vypracovala teorii rekurzivních funkcí a později se zabývala možnostmi jejich využití v programování. Proslavila se také populární knihou *Hra s nekonečnem* z roku 1957.

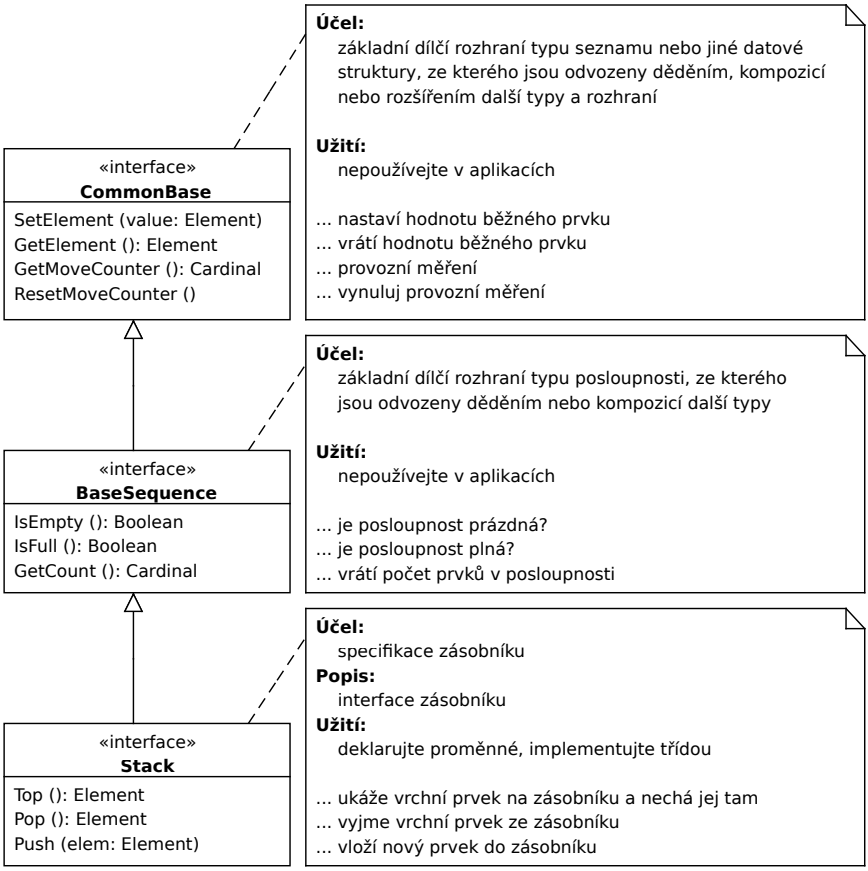
takhle definovat nedá, protože musí vyhovovat matematické definici funkce. A funkce může mít pro každou hodnotu argumentu právě jednu (v případě parciální funkce nejvýše jednu) funkční hodnotu, a to je v případě operace *Pop* nová hodnota změněného zásobníku. Při programování se však nemusíme přísně řídit matematickou definicí funkce a můžeme si dovolit praktičtější implementaci operace *Pop*.

3.2.2 Implementace obecného zásobníku

Rozhraní `Stack` je v knihovně implementováno třídou `Stack` (viz obr. 3.11, 3.12 a 3.13). Třída `Stack` vznikla specializací (tj. děděním) z třídy `BaseSequence` – to je společný základ všech posloupností:



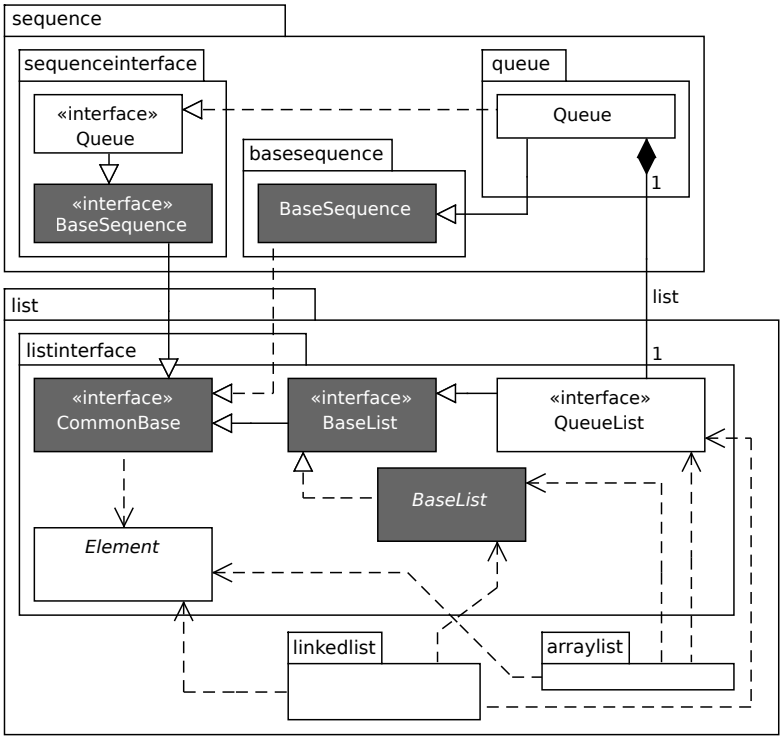
Obr. 3.11: Design obecného zásobníku `Stack`



Obr. 3.12: Rozhraní `Stack` a jeho předci

definuje metody pro nastavení a zjištění hodnoty běžného prvku, zjištění počtu prvků, příp. zda je posloupnost prázdná nebo plná a také provozní měření výpočtové složitosti.

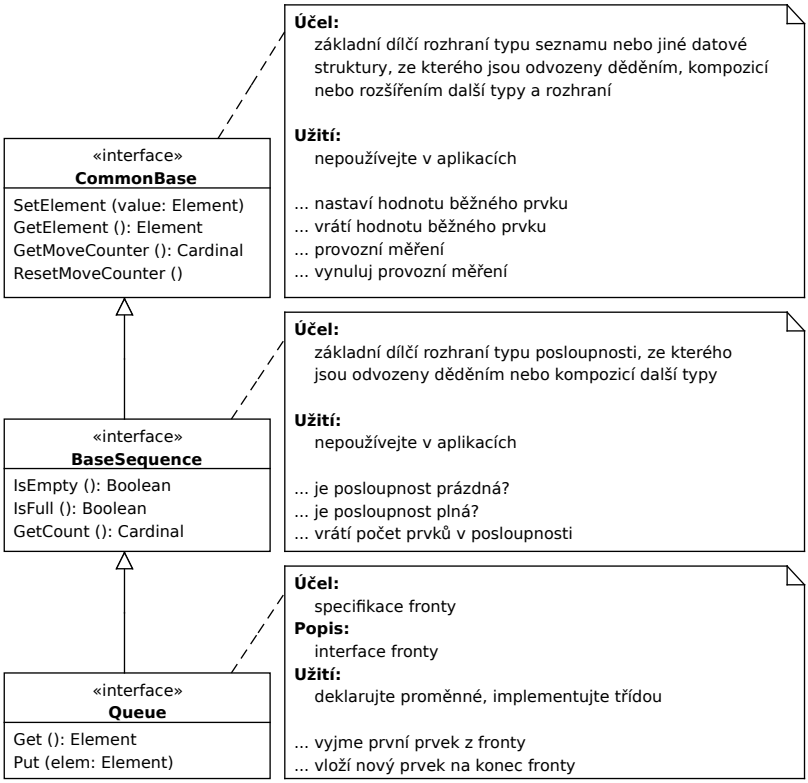
Na první pohled se může zdát, že zásobník bychom měli implementovat děděním z posloupnosti, protože zásobník je zvláštním (speciálním) případem posloupnosti a protože zrovna dědičnost je ta vazba, která vyjadřuje vztah mezi obecným a zvláštním (neboli generalizace – specializace). Skutečně: každý zásobník je posloupnost a ne každá posloupnost musí být zrovna zásobník. To je správný postřeh



Obr. 3.28: Design obecné fronty `Queue`

vit novou nerozkolísanou hodnotu otáček vždycky, když nám přijde nový pulz ze snímače. Kdybychom počítali aritmetický průměr třeba ze šedesáti hodnot, ty pak zahodili a čekali, až naměříme dalších šedesát hodnot, měřili bychom aktuální hodnoty otáček šedesátkrát méně často – a považte, co vám motor někdy dokáže udělat za jedinou sekundu, když pořádně šlápnete na plyn (nebo taky když prudce sundáte nohu z plynu)!

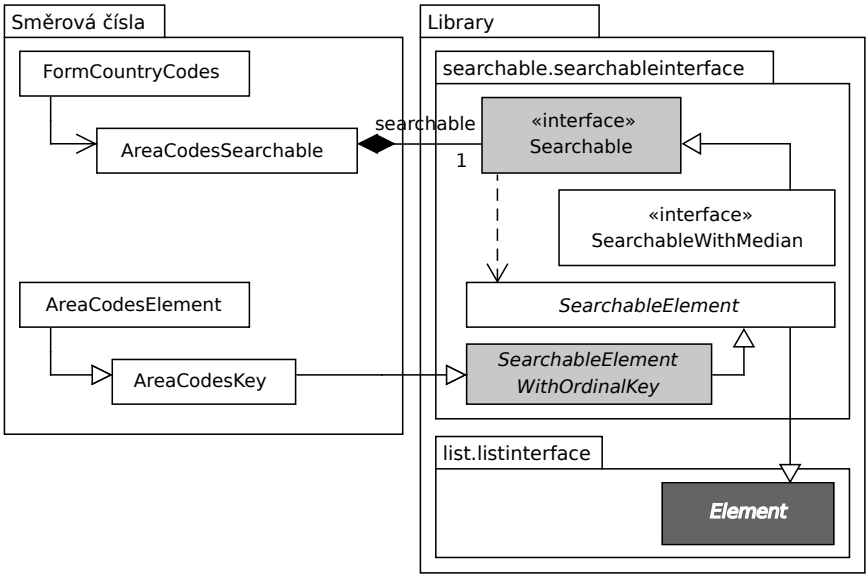
Pro naše potřeby si úlohu zjednodušíme: místo snímače zubů umístíme do formuláře tlačítko a tím budeme simulovat „cvakání“ zubů procházejících okolo snímače. Frekvenci otáček budeme vypisovat do textových políček – jednak frekvenci okamžitou (nezprůměrovanou), jednak klouzavý průměr z několika posledních hodnot. Tento formulář vidíte na obrázku 3.25.



Obr. 3.29: Rozhraní `Queue` a jeho předci

Naše aplikace vypočítává klouzavý průměr a k tomu potřebuje frontu. Obecná fronta je součástí objektové knihovny a my ji potřebujeme jenom specializovat na frontu racionálních čísel. Uživatelské rozhraní aplikace tvoří jediný formulář typu `FormClicking`. Aplikace specializuje abstraktní třídu `Element` – obecný prvek posloupnosti na prvek obsahující racionální číslo `DoubleElement`. Dále definuje rozhraní racionální fronty `DoubleQueue`.[12] Třída `DoubleQueue` je naprogramována jako *obal* (návrhový vzor, anglicky *wrapper*), který

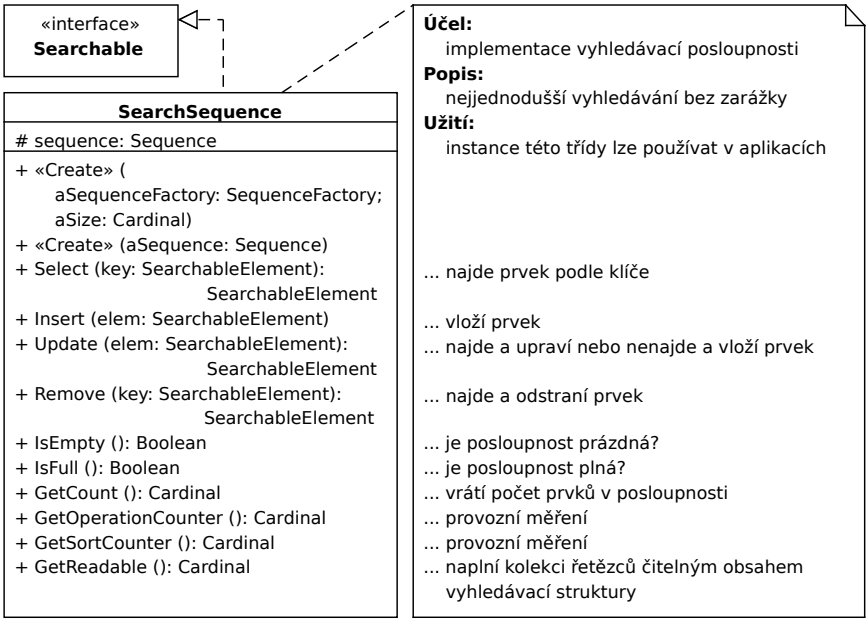
[12] rozhraní `DoubleQueue` je potřebné v Delphi k tomu, aby fungovala automatická správa paměti



Obr. 5.2: Design aplikace *„Směrová čísla“*

nosti dokážeme hledat rychleji než v posloupnosti neuspořádané). To si podrobně ukážeme v následujících podkapitolách.

Uživatelské rozhraní aplikace tvoří formulář typu `FormCountryCodes`. Aplikace je oddělená od knihovny opakovaně použitelných tříd pomocí rozhraní `Searchable` a hierarchie abstraktních tříd `Element` – `SearchableElement` – `SearchableElementWithOrdinalKey`. Aplikace dědí abstraktní třídu `SearchableElementWithOrdinalKey`, která slouží jako obecný prvek posloupnosti, a specializuje ji na prvek obsahující celočíselný klíč `AreaCodesKey`. Z této třídy dále odvozuje prvek s klíčem i hodnotou typu `AreaCodesElement`. Ve formuláři se pak využívá speciální vyhledávací datová struktura typu `AreaCodesSearchable`, která je navržena jako obal okolo obecné vyhledávací struktury představované rozhraním `Searchable`. Rozhraní `Searchable` je pak implementováno v knihovně různým způsobem: jako různé druhy vyhledávacích posloupností, jako strom nebo jako tabulka.



Obr. 5.3: Třída `SearchSequence`

Třída `SearchSequence` je obyčejná vyhledávací posloupnost. Je to jedna z možných implementací rozhraní `Searchable.` Z této základní třídy jsou děděním postupně odvozeny další čtyři třídy vyhledávacích posloupností uspořádaných a reorganizovaných a vyhledávacích posloupností se zarážkou, které vyhledávají efektivněji než obyčejná vyhledávací posloupnost. Postupně si ukážeme těchto pět druhů vyhledávacích posloupností:

- obyčejná vyhledávací posloupnost
- vyhledávací posloupnost se zarážkou – *with sentinel*
- reorganizovaná (pořadí prvků je určeno heuristikou)
- uspořádaná – *ordered* (průběžně řazená, nové prvky se správně zařazují hned při vložení)
- seřazená – *sorted* (vždy před prohledáváním)

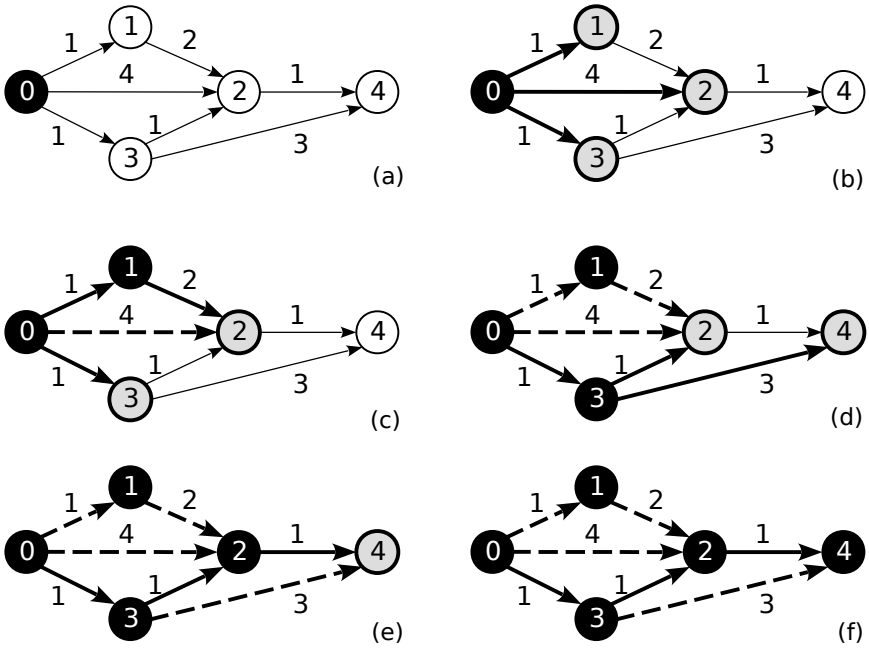
Ať se jedná o základní obyčejnou vyhledávací posloupnost, nebo o některou třídu odvozenou, každá z nich vždy obsahuje jako

délku do `pathLength` – viz obr. 10.19c. Stav následovníka nastaví na navštívený (`Visited`). Jestliže vrchol už byl navštíven a nyní se jen zkrátila jeho vzdálenost od výchozího uzlu, je již také vložen do pomocné haldy. Tím, že se mu změnila vzdálenost, mohlo být ovšem narušeno správné uspořádání haldy, proto se halda musí přerovnat metodou `Justify`. Vrchol nově navštívený stačí prostě jen přidat do haldy. Po prozkoumání a případné opravě všech následovníků algoritmus přejde z dosud běžného vrcholu do některého z vrcholů navštívených, ale dosud nedokončených. Do kterého? Ze všech navštívených vrcholů musí vybrat ten, do kterého vede nejkratší cesta z výchozího vrcholu. Nejkratší dosud známá cesta se totiž dalším prohledáváním už nemůže zkrátit, zatímco do vrcholů s cestou delší by se možná dala najít nějaká jiná, kratší cesta. Všechny vrcholy, ze kterých vybíráme, máme uložené v pomocné haldě. A právě halda má tu vlastnost, že nám vždy dodá nejmenší ze svých prvků, tj. vrchol s nejkratší cestou. Napřed ovšem ještě musíme zjistit, zda halda vůbec nějaký vrchol obsahuje. Je-li totiž prázdná, znamená to, že algoritmus již prozkoumal všechny dostupné vrcholy v grafu, ale cílový vrchol nenašel. To znamená, že v grafu neexistuje cesta z výchozího do cílového vrcholu. V tom případě musí algoritmus skončit a jako výsledek vrátí prázdnou cestu, která neobsahuje žádný vrchol. Jestliže halda není prázdná, dodá vrchol s nejkratší cestou z vrcholu výchozího, algoritmus jej nastaví jako běžný a jeho stav na dokončený (`Completed`) – viz obr. 10.19c–f. Celý cyklus prohledávání grafu se pak opakuje pro další a další běžné vrcholy. Poté, co algoritmus ukončí cyklický průzkum cest v grafu, naplní výslednou posloupnost vrcholy, které leží na nejkratší cestě: začne z cílového vrcholu a ten uloží na konec posloupnosti. V cílovém vrcholu je uložen odkaz na předchůdce, tam je odkaz na předchůdce předchůdce a tak se algoritmus postupně dostane k vrcholu výchozímu, který uloží do čela posloupnosti. Tak dospěje k výsledku.

Nyní si ukažme postup Dijkstrova algoritmu na příkladu:

1. Obr. 10.19a: Výchozí vrchol se nastaví do stavu dokončený.

2. Obr. 10.19b: Vrcholy 1, 2 a 3 jsou navštívené, ale nedokončené.



Obr. 10.19: Ukázka Dijkstrova algoritmu

3. Obr. 10.19c: Vrchol 1 se nastaví do stavu dokončený, do vrcholu 2 byla nalezena kratší cesta.

4. Obr. 10.19d: Vrchol 3 se nastaví do stavu dokončený, do vrcholu 2 byla nalezena ještě kratší cesta, bylo dosaženo cílového vrcholu 4, ale ještě není dokončený.

5. Obr. 10.19e: Vrchol 2 se nastaví do stavu dokončený, do vrcholu 4 byla nalezena kratší cesta.

6. Obr. 10.19f: Cílový vrchol se nastaví do stavu dokončený.

Tento příklad Dijkstrova algoritmu v 6 krocích je stručně shrnut do tabulky 10.1. Kurzívou je v každém kroku vyznačen navštívený vrchol nejblíž od vrcholu výchozího.

Při odhadu časové složitosti vyjdeme z toho, že v krajně nepříznivém případě musíme navštívit všechny vrcholy grafu, některé vrcholy

Tab. 10.1: Příklad Dijkstrova algoritmu v 6 krocích

Krok \ vrchol	0	1	2	3	4
1	0, Cpl.	Unr.	Unr.	Unr.	Unr.
2		*1, Vis.*	4, Vis.	1, Vis.	
3		1, Cpl.	3, Vis.	*1, Vis.*	
4			*2, Vis.*	1, Cpl.	4, Vis.
5			2, Cpl.		*3, Vis.*
6					3, Cpl.

pak navštívíme dokonce víckrát. Každý vrchol grafu, který leží na nejkratší cestě (a může se tedy stát součástí řešení), musíme dokončit. V krajním případě musíme dokončit všech V vrcholů. Dokončený vrchol vybíráme z prioritní fronty – na to potřebujeme $\log V$ kroků výpočtu (v případě, že případě, že prioritní frontu implementujeme haldou). Z každého z nejvýše V dokončených vrcholů pak musíme prohlédnout i všechny jeho následovníky, kterých může být také až V. V tom případě vyjde odhad časové složitosti úměrný $V^2 \cdot \log_2 V$ vzhledem k počtu vrcholů. Vezmeme-li však v úvahu i počet hran E, můžeme využít i toho, že každou hranou projdeme nejvýše jednou. Pak vyjde časová složitost nanejvýš přímo úměrná $(V + E) \cdot \log_2 V$, kde V je počet vrcholů a E je počet hran.

Jednoduchou úpravou můžeme změnit Dijkstrův algoritmus tak, aby počítal nejkratší cestu i v grafech se záporným ohodnocením hran, ale bez cyklů. Nejkratší cestu v grafu bez ohodnocení můžeme hledat Dijkstrovým algoritmem tak, že všechny hrany fiktivně ohodnotíme jedničkou. Doc. Töpfer v knize [60] popisuje i další zajímavé modifikace Dijkstrova algoritmu, např. hledání nejkratší cesty v grafu, kde hrany jsou ohodnoceny několika hodnotami (např. doba jízdy a cena). Dejme tomu, že hledáme nejrychlejší spojení a z nich nás pak zajímá to nejlacinější. Jde vlastně jen o to, abychom uměli správně porovnat dvě dvojice hodnot: doba jízdy má přednost, a když je stejná, potom rozhoduje cena.

10.6.4 Cvičení 27

Najděte a opravte chybu v projektu *Telefonní síť* v balíčku `Library.Graph.Tasks` ve třídě `DijkstraShortestPath` v metodě `ShortestPath`. V `ConfigurationModule` aktivujte příkaz `exercise27`.

10.7 Shrnutí

- **Vnitřní reprezentaci** grafu volíme na míru algoritmu, kterým hodláme řešit úlohu. Např. *k Floydovu algoritmu* se hodí **matice sousednosti,** *k Dijkstrovu algoritmu* **seznam následníků** a k uložení grafu do souboru zase spíš **posloupnost hran.**
- Je třeba rozlišovat mezi grafy **orientovanými** a **neorientovanými** (nejen proto, že se definují různě). Např. *topologicky uspořádáváme* graf orientovaný, *minimální kostru* hledáme pro graf **neorientovaný** a *nejkratší cestu* můžeme hledat jak na grafech orientovaných, tak na neorientovaných.
- Úlohy, které se řeší na grafech, jsou různorodé. Může jít o zjišťování různých vlastností grafu – např.: Je graf strom? Jsou v grafu cykly? Existuje cesta mezi zadanými vrcholy? Je graf souvislý? Typické jsou také **optimalizační úlohy,** např. hledání nejkratší cesty nebo řešení dopravního problému (tím jsme se zde ovšem nezabývali). Na rozdíl od optimalizací, které bývají časově nebo paměťově velmi náročné, tzv. **hladové (greedy) algoritmy jsou mimořádně efektivní.** Příkladem hladového algoritmu je *Kruskalův algoritmus* na hledání *minimální kostry grafu.*