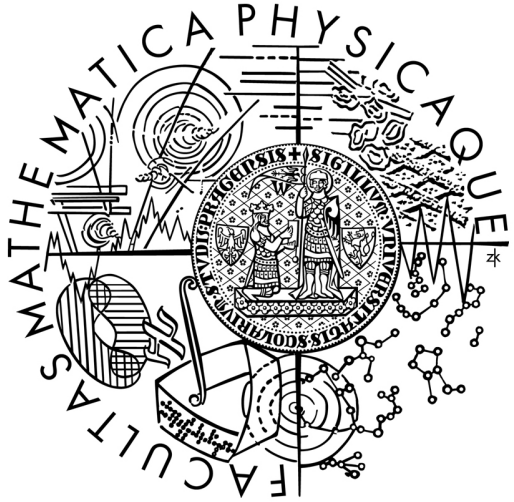
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jiří Šebesta

Vizualizace algoritmů pomocí uživatelských skriptů

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Pavel Ježek

Studijní program: Informatika, programování

2007

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne Jiří Šebesta

Obsah

Název práce: Vizualizace algoritmů pomocí uživatelských skriptů

Autor: Jiří Šebesta

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Pavel Ježek

e-mail vedoucího: pavel.jezek@mff.cuni.cz

Abstrakt: Cílem této práce bylo vytvoření programu AlgoShow sloužícího pro názornou ukázku fungování algoritmů. Algoritmy se zapisují jako skripty, ve kterých je možno využívat různých funkcí pro grafické znázornění běhu algoritmu. Práce se zabývá programem AlgoShow jak z uživatelského, tak i z programátorského hlediska. Součástí je krátký úvod do problematiky vizualizace algoritmů, srovnání programu AlgoShow s podobnými existujícími nástroji, popis návrhu a implementace programu včetně způsobu interpretace skriptu. Dále práce obsahuje charakteristiku základních funkcí programu a zejména podrobný popis skriptovacího jazyka.

Klíčová slova: AlgoShow, algoritmus, vizualizace, skript

Title: Algorithm visualization using user-defined scripts

Author: Jiří Šebesta

Department: Department of Software Engineering

Supervisor: Mgr. Pavel Ježek

Supervisor's e-mail address: pavel.jezek@mff.cuni.cz

Abstract: The goal of this thesis was to create a program (AlgoShow) for explanatory chart of algorithms' process. Algorithms are written in the form of user-defined scripts that offers various functions for graphic presentation of algorithm's process. The thesis describes the program from both user's and programmer's points of view. The text includes short introduction into algorithm visualization, comparison of the AlgoShow program with another similar tools, description of design and implementation of the program including the method of script's interpretation. The thesis also includes characterization of essential program's features and detailed description of scripting language.

Keywords: AlgoShow, algorithm, visualization, script

1. Úvod

1.1. Cíl práce

Cílem práce je vytvořit program AlgoShow sloužící pro názornou ukázku fungování algoritmů. Algoritmy nejsou v programu zabudovány „napevno“, ale zapisují se jako skripty (v jazyce podobném C), ve kterých je možné využít různých funkcí pro grafické znázornění běhu algoritmu. Program má dvě základní části - editor pro psaní a odladění skriptů a prohlížeč pro předvedení zapsaného algoritmu. Díky řadě wizardů a pomůcek je možné napsat skript i bez hlubší znalosti grafických funkcí skriptovacího jazyka. Součástí je i ukázková sada skriptů se základními algoritmy (třídění v poli, práce s binárními stromy, algoritmy na grafech).

2. Vizualizace algoritmů a existující nástroje

2.1. Krátký úvod do vizualizace algoritmů a její motivace

Algoritmus je *„přesný návod či postup, kterým lze vyřešit daný typ úlohy“ [3]*. Jeho vizualizací pak rozumíme přehledné grafické znázornění běhu algoritmu. Toto znázornění může mít mnoho podob – od několika statických obrázků, přes ztvárnění pomocí šipek a popisků až po propracované animace. Vizualizace algoritmů je způsob, jak studentům (případně jiným zájemcům) umožnit snadné a rychlé pochopení fungování algoritmu.

Vizualizace přináší pro zájemce o poznání algoritmu celou řadu výhod. Narozdíl od běžného výpisu zdrojového kódu, popřípadě hodnot proměnných v jednotlivých okamžicích běhu, nabízí vizualizace pohodlné grafické znázornění objektů, se kterými algoritmus pracuje (jako jsou vrcholy a hrany grafu, uzly stromu, položky pole...). Tyto objekty pak mohou být i logicky pospojovány například šipkami či animacemi. Grafické znázornění navíc umožňuje pochopení algoritmu i lidem, kteří neovládají žádný programovací jazyk.

Tento způsob výuky může být aplikován jak ve vyšších ročnících středních škol při vysvětlování základních algoritmů (Bubblesort, Quicksort, ...), tak zejména na školách vysokých, kde jsou probírány na pochopení náročnější algoritmy a datové struktury – jako je například práce s grafy a různé druhy binárních stromů. Jedinou možností, jak tuto názornou výukovou metodu nahradit, je klasický způsob křída-tabule, resp. tužka-papír. Čas většiny přednášek a cvičení je však výrazně omezen a proto je počet „odkrokovaných“ běhů algoritmu často minimální. Studenti pak mají před zkouškou problém pochopit princip daného postupu jen ze dvou malých příkladů. Mohou si sice vytvořit sadu vlastních příkladů, ale jen stěží ověří správnost svého řešení. Kvalitní vizualizace jim ale nabízí možnost testovat své schopnosti, dokud nebudou se svými výsledky plně spokojeni. Navíc je zde takřka vyloučena jakákoliv chyba, která se na cvičení či přednášce občas objevit může a studenta často zmate.

Je však nutné podotknout, že vizualizace je pouze prostředkem usnadňujícím pochopení algoritmu, ale je prakticky nemožné naučit se algoritmus pouze na jejím základě. Potřebné je zejména již předem znát základní principy daného algoritmu a jeho přibližný zápis, ať už ve zdrojovém kódu a nebo ještě lépe v přirozeném jazyce. Ideálním způsobem výuky algoritmů se tedy zdá být vysvětlení jeho základů při přednášce, odkrokování několika

málo příkladů na cvičení a následné domácí samostudium za pomoci kvalitního nástroje na vizualizaci algoritmů.

2.2. Existující nástroje a porovnání s programem AlgoShow

S rozvojem informačních technologií na konci 20. století nastal i rozmach škol zaměřených na výuku informatiky. Pak bylo již jen otázkou času, kdy se objeví první nástroje umožňující názornou výuku algoritmů. První vlaštovky se začaly objevovat na začátku devadesátých let minulého století, ale opravdový rozkvět těchto nástrojů nastal až s nástupem internetu.

I přes mnohahodinové hledání se mi nepodařilo najít program postavený na principu obdobném programu AlgoShow. Jediným produktem alespoň vzdáleně podobným je německý software j-Algo, o kterém podrobněji pohovořím později. V žádném z ostatních nalezených programů totiž není možné přidávat vlastnoručně vytvořené vizualizace a jsme v nich odkázáni pouze na vestavěné algoritmy.

Nejčastějším způsobem zpracování nástrojů na vizualizaci algoritmů je technologie Java appletů. Jejich nespornou výhodou je možnost integrace appletu do webové stránky, díky čemuž odpadá nutnost instalace. Značným mínusem všech těchto nástrojů je ale omezení na jeden či několik málo vestavěných algoritmů. Velké rozdíly pak existují v kvalitě zpracování, a to jak po grafické, tak funkční stránce. Na internetu lze nalézt také kvalitní katalog všech možných vizualizací (viz [5]) - nachází se zde i hodnocení kvality některých nástrojů. V následujících odstavcích popíši několik appletů, které slouží k předvádění algoritmů, jež jsou i součástí základní sady skriptů programu AlgoShow.

Nejprve se podívejme na algoritmy pro třídění v poli. Kromě mnoha prográmků simulujících jen jeden z mnoha algoritmů, existuje i několik nástrojů nabízejících porovnání nejznámějších třídících metod. Zmínit můžeme například applet [6], který nabízí setřídění pole pomocí šesti různých metod. Výhodou je ukázání zdrojového kódu algoritmu a aktuální polohy v něm, což zvládá jen malá část nástrojů (včetně AlgoShow). Další zajímavou možností je javová aplikace xSortLab, která nabízí celkem pět třídících metod. Je o něco uživatelsky příjemnější než předchozí applet, ale nenabízí pohled do zdrojového kódu třídícího algoritmu. Má však více možností pro porovnání efektivity jednotlivých metod –

například umožňuje zobrazení údajů o náročnosti setřídění rozsáhlejších polí, a to čas v milisekundách, počet porovnání a počet kopírování.

Nyní si proberme některé applety specializované na práci s binárními vyhledávacími stromy – zejména pak s červeno-černými a AVL stromy. Spíše odstrašujícím příkladem je nástroj [7], který sice neposkytuje funkční simulaci AVL stromů, ale ta je provázena naprosto nepřehledným a nevzhledným uživatelským prostředím. O trochu příjemnější je simulace červeno-černých stromů [8]. Ta ovšem nenabízí žádnou interakci s uživatelem a je tedy k výuce takřka nepoužitelná. Jednoznačně nejlepší applet na práci s binárními stromy lze ale najít na internetové adrese [9]. Ten nabízí uživatelsky velice příjemnou práci celkem se třemi druhy binárních stromů, a to se splay stromy, AVL stromy a červeno-černými stromy. U těchto tří datových struktur je navíc možné provádět všechny běžné operace (vkládání, mazání, hledání, atd.), které jsou navíc doprovázeny názornými animacemi.

Co se týče appletů na práci s grafy, je už výběr daleko menší. Například na demonstraci Dijkstrova algoritmu se mi podařilo najít pouze dva. Jeden byl však takřka nefunkční a druhý naprosto nepřehledný. V případě práce s grafy je tedy lepší použít některý z obecnějších programů popsaných v následujících odstavcích.

Na konec přehledu jsem si nechal tři aplikace nabízející vizualizace algoritmů více typů. Nejprve se podívejme na tu asi nejméně zdařilou. Jedná se o projekt pocházející z univerzity v německém Marburgu s názvem DataStructureVisualization. Šlo o velmi nadějné dílo, jehož vývoj však byl zastaven v roce 2000 ve fázi beta verze, a to je bohužel citelně znát. Aplikace obsahuje velké množství algoritmů: třídění polí, práce s mnoha druhy spojových seznamů, s AVL, červeno-černými, splay a a-b stromy. Uživatelské prostředí je vcelku příjemné a počet funkcí je také ucházející. Problémem je ale samotná funkčnost algoritmů. Hned při mém prvním zkoušení tohoto programu se po vložení třetího prvku do AVL stromu porušila základní vlastnost BVS – levý syn byl větší než jeho rodič.

Další zajímavou aplikací je program Swan, který je vlastně souhrnem několika samostatných programů napsaných v c++ (zdrojové kódy jsou součástí). Tento nástroj byl na svou dobu jistě velmi pokrokový, ale jeho vývoj se zastavil již na konci devadesátých let, takže zejména co se týče grafické stránky a uživatelské přívětivosti, za dnešními aplikacemi pokulhává. Množství zvládaných algoritmů je vcelku velké – umí například binární vyhledávací stromy, toky v sítích, Minimax, červeno-černé stromy, Heapsort a mnoho

dalších. Jak již bylo řečeno, hlavní nevýhodou je nekvalitní grafika bez jakéhokoliv ztvárnění postupu algoritmu, ať již pomocí animací, šipek, či popisků.

A na jako úplně poslední se podíváme na jednoznačně nejlepší program, který nese jméno j-Algo. Tato původem německá aplikace se jako jediná přibližuje našemu programu AlgoShow v tom, že je možno do ní doplňovat vlastní algoritmy. Narozdíl od AlgoShow ale nenabízí vlastní editor algoritmů, nýbrž používá moduly napsané v Javě. Kvalita jednotlivých vizualizací tak záleží zejména na autorovi příslušného modulu. Součástí programu jsou celkem tři ukázkové algoritmy (AVL, Dijkstra, Knuth Morris Pratt) a navíc jeden speciální modul pro tvorbu vývojových diagramů. Trošku detailněji se podívejme na první dva jmenované algoritmy, které jsou i součástí AlgoShow. Práce s AVL stromy je v obou programech obdobná – a to jak z pohledu grafiky, tak komentování průběhu. V aplikaci j-Algo však chybí nahlédnutí do zdrojového kódu algoritmu. U Dijkstrova algoritmu je problémem německého programu zejména těžkopádné zadávání podoby grafu, které je v aplikaci AlgoShow uděláno vcelku intuitivně. Samotný průběh algoritmu je podobný, jen v případě j-Algo opět chybí možnost sledování zdrojového kódu. Mírnou výhodou javovského modulu je možnost zobrazení matice sousednosti při editování grafu.

Závěrem malé exkurze ke konkurenčním programům přidávám tabulku srovnávající některé z produktů, které mne zaujaly, s naším programem AlgoShow.

	xSortLab	DataStructure Visualization	Swan	j-Algo	AlgoShow
Druh aplikace	Java aplikace	Java aplikace	C++ - soubor programů	Java aplikace	C++ (MFC)
Vestavěné algoritmy	třídění v poli	třídění v poli, spojové sez., BVS, AVL, RBT, Splay, a-b stromy	BVS, RBT, Heap sort, toky v síti, KMP, Minimax	AVL, KMP, Dijkstra	třídění v poli, AVL, RBT, min. kostra, Dijkstra
Možnost rozšíření	Ne	Ne	Ne	Ano	Ano
Editor algoritmů	Ne	Ne	Ne	Ne	Ano
Komentář průběhu	Ano	Ne	Ne	Ano	Ano
Animace	Ano	Ano	Ne	Různé	Ne
Propojení se zdroj. kódem algoritmu	Ne	Ne	Ne	Ne	Ano
Poznámka	Porovnání algoritmů	Chyby ve funkčnosti			
Cena	0	0	0	0	0

Tabulka 2.2.1 – srovnání programů na vizualizaci algoritmů

3. Obecné řešení problému

3.1. Skriptovací jazyk a jeho interpretace

Vůbec prvním úkolem při návrhu programu bylo nalezení způsobu, jakým budou algoritmy zapisovány. Základním požadavkem bylo, aby tento způsob byl pro uživatele relativně jednoduchý na pochopení, ale zároveň aby nabízel dostatečné prostředky pro názorné předvedení běhu algoritmu. Jako jediná možnost splňující obě podmínky se ukázalo navržení vlastního skriptovacího jazyka. V následujících odstavcích se budu věnovat především principům jeho interpretace, detailní popis jazyka najdete v kapitole věnované programu AlgoShow z pohledu uživatele.

Návrh jazyka

Ve chvíli, kdy už bylo jasné, že je nutno navrhnout vlastní skriptovací jazyk, přišla na řadu otázka, jak má vypadat syntaxe a jak rozsáhlý by měl být. Co se týče syntaxe, bylo jasné, že není dobrý nápad vymýšlet vlastní, ale že by měla vycházet z nějakého existujícího a hojně používaného jazyka. V úvahu připadaly dva – ve výuce algoritmů asi nejpoužívanější – Pascal a C. Vzhledem k tomu, že samotný program AlgoShow je psán v C++, zvolil jsem druhou variantu a syntaxe tedy vychází z jazyka C.

Druhou neznámou bylo to, jak složitý by jazyk měl být. Nejprve se zabývejme klasickou procedurální stránkou jazyka, otázku speciálních funkcí týkajících se algoritmů necháme na později. Samozřejmostí bylo, že jazyk bude muset zvládat podmíněné příkazy, cykly, přiřazování do proměnných a vyhodnocování výrazů. Menší pochybnosti už nastaly u rozhodování, zda se bude skript zapisovat jako jeden dlouhý kus kódu, anebo ho bude možné rozdělit do funkcí a ty pak mezi sebou vzájemně volat. Jelikož by ale program měl zvládat i rekurzivní algoritmy, jako je například Quicksort, bylo nutné implementovat i volání funkcí. Další otázkou byl počet vestavěných typů. Ten se nakonec ustálil na třech. Prvním je „int“, tj. celé číslo, které zastupuje i neexistující typ bool, druhým „double“ nabízející i reprezentaci desetinných čísel a posledním „string“, jenž je obdobou typu string ze stejnojmenné knihovny jazyka C++. Je možné také deklarovat pole kteréhokoliv ze tří předcházejících typů, a to stejným způsobem jako v C. Navíc je zde ale možno pomocí tečkové notace přistupovat ke grafickým vlastnostem jak celého pole, tak jednotlivých buněk.

Speciální kapitolou jsou pak typy určené přímo pro jejich aplikaci v algoritmech, které disponují i mnoha možnostmi zobrazení. Prvním z nich je „Cell*“, který je ukazatelem na jakousi buňku, která mimo své hodnoty (typu int nebo double) obsahuje i tři ukazatele, aby byla snadno použitelná v algoritmech pracujících s binárními stromy nebo spojovými seznamy. Celkem pět datových typů je pak určeno specielně pro práci s grafy, jsou to „Graph“, „Vertex“ a „Edge“ reprezentující celý graf, resp. jeho vrchol či hranu, a typy „EdgeIterator“ a „VertexIterator“ sloužící ke snadnému procházení hran a vrcholů grafu (více o jejich použití v uživatelské části).

Kontrola syntaxe a „překlad“ kódu

I když rychlost není při vizualizaci algoritmů rozhodujícím kritériem, rozhodl jsem se, že skriptovací jazyk nebude interpretován přímo z uživatelem napsaného kódu, ale ten bude během spouštění algoritmu mírně upraven tak, aby se již interpret mohl spolehnout na to, že kód dodržuje určitý řád (například žádné mezery ve výrazech apod.). Tento překlad jsem spojil s kontrolou syntaxe.

Následovalo dilema, jakým způsobem kontrolovat syntaxi. Existovaly celkem dvě varianty – použít nějaký nástroj (nabízel se zejména Bison) nebo vše napsat vlastnoručně. S ohledem k rozsáhlosti kódu generovaného Bisonem a k relativní jednoduchosti našeho jazyka jsem se rozhodl pro vlastní ošetření všech možných situací. Této volby jsem sice v průběhu práce často litoval, ale domnívám se, že ve výsledku to přispělo k lepší čitelnosti a snadnějšímu pochopení zdrojového kódu celého programu.

Nyní se v krátkosti podívejme na princip fungování kontroly syntaxe a současného překladu kódu. Na úplném začátku dojde k odstranění komentářů a k načtení seznamu vestavěných funkcí (z důvodu kontroly správnosti jejich volání a případné duplicitě identifikátorů). Následuje zavolání kontroly deklarací. Pokud je deklarována proměnná, zkontroluje se jedinečnost jejího názvu (nesmí se shodovat ani s názvy klíčových slov) a uloží se spolu s jejím typem do tabulky pro pozdější kontrolu jejího použití. Jestliže je deklarována funkce, mohou nastat dvě varianty – končí středníkem, tj. jedná se pouze o definici hlavičky, a nebo za ní následuje tělo. V prvním případě se po kontrole údaje pouze uloží do tabulky a

řádek se smaže (pro interpret není definice hlavičky potřeba). Následuje-li tělo funkce, zavolá se funkce kontrolující blok příkazů.

Uvnitř funkce pak můžeme narazit celkem na pět druhů příkazů. Prvním je příkaz přiřazení. Zde dochází ke kontrole, zda přiřazovaný výraz je stejného typu jako proměnná, do které se má dosadit, případně je-li možná konverze (jsou povoleny konverze *int→double*, *double→int*, *int→string* a *double→string*). Dále se odstraní všechny přebytečné mezery a stejně jako u jiných příkazů se smaže koncový středník, který interpret nepotřebuje (po přeložení je na každém řádku přesně jeden příkaz).

Druhou variantou příkazu je cyklus resp. podmínka, jejichž vyhodnocování je obdobné. Existují celkem dvě varianty podmíněného příkazu (s *else* nebo bez *else* větve) a tři druhy cyklů (*while*, *do ... while*, *for*). Hlavním úkolem kontroly syntaxe je ověření korektnosti pravdivostního výrazu v podmínce cyklu. Jedná se zejména o odchycení případů, kdy se porovnávají neporovnatelné typy, resp. dochází k nepovoleným porovnáním (např. operátor „<“ u ukazatelů). Další komplikací je, že tělo cyklu (podmínky) může obsahovat pouze jeden příkaz, který nemusí být ohraničen složenými závorkami. V požadavcích interpreta však je, aby i jednořádkové příkazy byly ohraničeny těmito závorkami. Proto musí kontrola syntaxe tuto variantu podchytit a závorky v případě potřeby přidat. Speciální kapitolou je pak *for* cyklus, který interpret vůbec nezná a musí se tedy převést na funkčně ekvivalentní *while* cyklus, a to následujícím způsobem:

```
for (<inicializace>; <podmínka>; <změna>)   →   <inicializace>
{                                               while (<podmínka>)
        .....                                   {
}                                                       .....
                                                        <změna>
                                                }
```

Třetím a vcelku jednoduchým druhem příkazů jsou speciální vestavěné funkce programu, které nelze ošetřit jako volání funkcí klasických. Jedná se například o metody na grafech či grafových iterátorech, ke kterým se přistupuje pomocí tečkové notace. Jiným typem příkazu je potom *return*, který ukončuje danou funkci a určuje její návratovou hodnotu. Kontrola syntaxe spočívá jen v ověření, že výraz odpovídá návratovému typu.

V této sekci také dochází ke zkontrolování zápisu vstupní funkce *read*, která může od uživatele vyžádat zadání typů *int*, *double*, *string* či *Graph*. Posledním ošetřovaným příkazem v této části je „BREAK“ sloužící k přerušení skriptu – po stránce syntaxe dochází jen k ověření, že je přiřazena nějaká zarážka.

Další varianta příkazu je zavolání funkce. Z pohledu kontroly zápisu se jedná o ověření typově správných parametrů a samozřejmě existence volané funkce. Poslední možností, jak může vypadat příkaz, je opět nějaká deklarace. Ta je obdobou globální deklarace, jediným rozdílem je zákaz definic funkcí a to, že názvy a typy proměnných se ukládají do lokálních tabulek funkce. To proto, aby ve více funkcích mohly existovat proměnné stejného jména.

Tím je dokončena kontrola samotného kódu, poslední věcí, která se musí ověřit, je existence funkce *main*, která nesmí mít žádné parametry. V případě zjištění jakékoliv syntaktické chyby dojde k vyhození výjimky a následnému oznámení chyby uživateli.

Interpretace jazyka

Po úspěšné kontrole syntaxe se nám do paměti načte kód, který má určitá pevná pravidla: přesně jeden příkaz na řádek, žádné mezery ve výrazech, oddělení názvu typu a proměnné přesně jednou mezerou, i jednořádkový blok kódu musí být oddělen složenými závorkami. Díky tomu jsme schopni dosáhnout rychlejší interpretace. Jsme ale odkázáni na dokonalou kontrolu syntaxe, sebemenší chyba v zápisu by totiž mohla znemožnit běh skriptu.

Před samotným spuštěním běhu kódu je nejprve zavolána funkce, která projde všechny globální deklarace a uloží si údaje o jednotlivých funkcích – typy jejich parametrů, návratový typ a zejména řádky, v jejichž rozsahu je tělo funkce. Tyto údaje se ukládají do třídy, jejíž součástí je též seznam všech globálních proměnných a zásobník volání funkcí, kde ke každé funkci náleží také seznam jejich lokálních proměnných.

Pak již stačí nastavit ukazatel do kódu na první řádek funkce main a začít interpretovat. I zde máme několik základních typů příkazů. Vůbec první, co ověříme, bude, zda se nejedná o příkaz k přerušení běhu skriptu. V tom případě dojde k vyhození výjimky, která zajistí dočasné ukončení běhu algoritmu, ale jeho data ponechá v paměti. Pokud pak

dojde k opětovnému spuštění skriptu, pokračuje se v místě, kde se skončilo, tzn. na aktuálně zpracovávaném řádku funkce, která je na vrcholu zásobníku.

Jestliže se nejedná o přerušení celého skriptu, můžeme pokračovat dál. Další na řadě je možnost příkazu *return* ukončujícího aktuálně běžící funkci. V tom případě dojde k uložení návratové hodnoty a její poslání funkci, která se ocitne na vrcholu zásobníku. Číslo aktuálně zpracovávaného řádku se pak nastaví na místo, odkud byla ukončovaná funkce zavolána. Pokud je zásobník volání prázdný (tzn. byla ukončena funkce *main*), dojde k ukončení běhu skriptu.

Následuje případné ošetření příkazů pro práci s grafickými prvky (vykreslení textů, šipek, nastavení vlastností pole…) a s grafy (nastavení grafických vlastností, přístup k vrcholům a vestavěným funkcím, iterátory...). Zpracováním těchto příkazů se zde nebudu podrobně věnovat, jelikož se jedná o poměrně rozsáhlou ale ne příliš myšlenkově náročnou část.

Přiřazovací příkaz sám o sobě je vcelku jednoduchý, složitější je ale správné vyhodnocení výrazu, který se přiřazuje. U typu *string* se jedná pouze o sřetězování pomocí znaménka ‘+‘. Náročnější je vyhodnocování výrazů číselných, ve kterých se mohou vyskytovat operace sčítání, odčítání (a unární mínus), násobení, dělení a modulo. Samozřejmě je možné i uzávorkování výrazů. Navíc je nutno zachovat priority operací. Proto jsem přistoupil k řešení pomocí binárního stromu, který se postaví nad každým výrazem na základě priorit operací a závorek. Pak dochází k infixovému vyhodnocení výrazu.

Nedílnou součástí skriptů jsou i příkazy pro vstup a výstup (*read* a *write*). Jedná se vlastně o volání funkcí, ale s tím rozdílem, že je nutno zobrazit příslušný dialog a přerušit běh skriptu. V případě operace *write* dojde k přerušení automaticky při zobrazení dialogu *MessageBox*, varianta *read* je ale ošetřena vlastním dialogem, a tak musí dojít k vyhození výjimky, která zajistí přerušení skriptu. Po zavření dialogu se algoritmus opět rozběhne na místě, kde skončil.

Nyní přistoupíme k ošetření případného zavolání funkce. To je relativně jednoduché – stačí přidat na zásobník volanou funkci a nastavit číslo zpracovávaného řádku na první řádek volané funkce. Současně dojde k automatickému nadeklarování lokálních proměnných, které reprezentují parametry funkce předávané hodnotou (předávání odkazem náš jazyk neumí), a dosadí se do nich hodnoty parametrů, se kterými je funkce volána.

Posledním možným typem příkazu je uvození podmínky či cyklu. Oba případy jsou implementovány stejně (podmínka je vlastně cyklus, který proběhne buď jednou, nebo vůbec). Z důvodu možnosti vnořování cyklů obsahuje každý exemplář funkce na zásobníku vlastní zásobník cyklů. U každého cyklu je uložena řídící podmínka a rozsah řádků, ve kterých se cykly pohybují. Pak už stačí jen po každém doběhnutí cyklu vyhodnotit podmínku. Pravdivostní výrazy se stejně jako číselné vyhodnocují pomocí vystavění binárního stromu nad výrazem.

3.2. Stručné nahlédnutí na MFC

Aplikace AlgoShow je napsána v jazyce C++ s použitím knihovny MFC (Microsoft Foundation Class Library – více viz [1], [2]). Následujících několik řádků věnuji základním pojmům a principům fungování této knihovny.

MFC poskytuje jakousi objektovou nadstavbu nad souborem základních funkcí API sloužících k programování okenních aplikací pro operační systém Windows. Knihovna obsahuje hierarchii více jak 200 tříd pro základní objekty používané v okenních aplikacích. Při volání metod těchto tříd dochází k zavolání (typicky stejnojmenné) funkce API na objekt, který je třídou reprezentován.

Nyní si stručně popišme hierarchii tříd MFC. Společným předkem většiny objektů je třída *CWnd*, od ní se odvozují třídy pro práci s výjimkami, grafikou, soubory, pokročilé šablony (obdoba STL) a třídy zajišťující samotnou aplikační architekturu – mezi ně patří i *CFrameWnd* a *CDialog*, které reprezentují okno vystavěné na architektuře pohled/dokument resp. klasické dialogové okno (tímto způsobem je napsán i program AlgoShow). Dalšími potomky *CWnd* jsou třídy reprezentující různé ovládací prvky (tlačítka, seznamy apod.). Podrobný popis některých těchto objektů si ukážeme později při jejich konkrétním použití v programu. Speciální kapitolou jsou třídy, které nejsou odvozeny od žádného předka. Mezi ně patří například *CString* (slouží k práci s řetězci), *CTime* (operace s časem), *CRect* (souřadnice obdélníku) nebo *CPoint* (poloha bodu).

3.3. Vykreslování průběhu algoritmu

Aby program AlgoShow plnil svou základní funkci, bylo zapotřebí zajistit vykreslování grafických prvků zajišťujících samotnou vizualizaci algoritmu. Vybíral jsem ze dvou možných způsobů vykreslování – Windows GDI (*Graphics Device Interface*) a knihovnou OpenGL (*Open Graphics Library*). I když varianta OpenGL nabízí širší škálu funkcí, tak v našem případě by se jednalo o použití „kanónu na vrabce“. K vykreslování průběhu algoritmu jsou totiž dostačující služby poskytnuté v rozhraní GDI. Navíc podpora kreslení za pomoci GDI je zabudována přímo v MFC.

Samotné kreslení probíhá za použití třídy CDC z knihovny MFC, která ukrývá kontext zařízení systému Windows (kam my chceme zapisovat) spolu s příslušnými funkcemi rozhraní GDI do jednoho balíčku. Vykreslování v našem případě neprobíhá přímo do kontextu zařízení (DC) hlavního okna, ale do DC příslušejícímu třídě *AlgPictureCtrl*, která je odvozena ze třídy *CStatic* reprezentující statický rám ohraničující oblast, kde se průběh algoritmu zobrazuje.

Při příchodu žádosti o aktualizaci zobrazeného stavu algoritmu je zavolána funkce *Paint (), která* postupně zajistí vykreslení všech možných podporovaných grafických objektů. Musíme rozdělit objekty deklarované jako klasické proměnné, jež mají povolenou vlastnost vykreslování, a globální grafické objekty (tj. šipky a texty, které ve skriptu nemají vlastnost klasických proměnných). V prvním případě se konkrétně jedná o proměnné typu pole (jakéhokoliv povoleného typu), *Cell** a *Graph* – k jejich vykreslení dojde, ať už jsou deklarovány jako proměnné globální a nebo lokální (v kterékoliv funkci, která figuruje na zásobníku volání). Konkrétní informace o způsobu, jakým se má proměnná vykreslit, se funkce dozví přímo z vlastností proměnné, které se nastavují ve skriptu (včetně příznaku *visible* určujícího, zda má být proměnná viditelná). Co se týče vykreslení šipek a textů, je situace o něco jednodušší, protože existuje jen jeden seznam všech šipek a všech textů včetně jejich polohy a grafických vlastností.

Další otázkou bylo, v jakém okamžiku má k vykreslení stavu algoritmu dojít. Samozřejmostí bylo překreslení v případě reakce na zprávu WM_PAINT, kterou dostane program od systému v okamžiku, kdy je potřeba okno aktualizovat. Překreslování je ale potřebné i za běhu skriptu, aby měl uživatel možnost proces vizualizace sledovat. Je ovšem

nemožné volat vykreslování po každé instrukci skriptu, protože by jednak došlo k výraznému zpomalení běhu, a jednak by díky neustálému blikání bylo sledování algoritmu pro uživatele velice nepohodlné. Ideální variantou tedy je, aby k překreslení došlo jen v okamžicích, kdy je běh skriptu přerušen pomocí zarážky. Díky tomu je efekt neustálého blikání eliminován.

3.4. Ošetření editování skriptu

K vytváření skriptu by uživatelům stačil i obyčejný objekt *CEdit*, který umožňuje editaci textu. Jeho použití je však velmi nepohodlné – celý skript by musel být napsán jedním typem písma a jednou barvou a byla by tak značně omezena přehlednost rozsáhlejších kódů. Bylo proto potřeba použít ovládací prvek *RichEdit*, který nabízí daleko širší paletu funkcí. Pro naši potřebu však bylo třeba ještě několik funkcí přidat. Konkrétně se jedná o barvení syntaxe, vlastní *Undo* historii a vykreslování pohyblivé zarážky.

Nejvýraznějším zpříjemněním editace kódu je zřejmě barvení syntaxe. Pro tuto potřebu existuje mnoho volně šiřitelných knihoven. Ty ale většinou nabízejí barvení zavedených jazyků, nikoliv však našeho speciálního. I kdyby se podařilo sehnat knihovnu umožňující nastavení klíčových slov a způsobu zavedení komentářů, byla by neřešitelná specialita našeho skriptovacího jazyka. Tou je možnost určení řádků kódu, které uvidí uživatel při prohlížení algoritmu. Ty se označí přidáním hvězdičky na jejich začátek. Tyto řádky se pak pro přehlednost zobrazují tučným písmem, což žádná mnou nalezená existující knihovna nezvládá. Z těchto důvodů jsem byl donucen napsat vlastní metodu barvící syntaxi. Barvení má celkem tři fáze. V první z nich se nabarví (výchozí barvou je modrá, ale v nastavení ji lze změnit) všechna klíčová slova vyskytující se v textu, a to včetně těch, která se vyskytují v komentáři či uvnitř řetězce. Následuje společné barvení řetězců a komentářů. To probíhá současně z důvodu případného výskytu komentáře v řetězci nebo řetězce v komentáři. Poslední fází je nastavení tučného písma na řádcích označených hvězdičkou. K samotnému přebarvení syntaxe musí dojít při jakékoliv změně v kódu. V případě klasického psaní dochází vždy jen k přebarvení konkrétního řádku, na kterém editace probíhá. Pokud ale dojde k otevření nového souboru, vložení textu ze schránky, případně k jiné změně obsahu (příkaz zpět, vyjmutí výběru,…), je zapotřebí zavolat funkci barvení na celou změněnou oblast.

S napsáním vlastní funkce na barvení syntaxe ale vyvstal problém s nefunkčností zabudované metody *Undo* třídy *CRichEditCtrl*. Ta nabízí možnost vrácení poslední změny. Bohužel ale za změnu považuje i procesy (změny výběru) prováděné při barvení syntaxe. V praxi to tedy znamená nemožnost použití této metody pro naše potřeby. Při psaní skriptů je však funkce „Zpět“ velice užitečná, a tak bylo nutné navrhnout funkci vlastní. Ta kromě ignorování změn při barvení syntaxe nabízí také možnost nastavení počtu kroků, o které se lze vrátit. Realizace spočívá ve vytvoření bufferu, do kterého se ukládají veškeré změny kódu. Ty mohou nastat mnoha různými způsoby: klasickým psaním a mazáním, vložením textu, vyjmutím či smazáním výběru, nahrazením textu. Jedna položka bufferu (tj. jedna změna) obsahuje celkem čtyři informace: text před změnou a jeho poloha a text po změně a rozsah výběru po změně v rámci *RichEditu*. Na základě těchto údajů je již snadné vrátit libovolnou změnu. V případě zavolání naší metody *Undo* se po vrácení poslední změny přesunou údaje o této operaci do druhého (*Redo*) bufferu, který umožní operaci znovu provést.

Posledním rozšířením, které bylo nutné dodělat ke klasickému prvku *RichEdit*, je zobrazování zarážky při prohlížení běhu algoritmu. Zarážka se vždy objeví na pozici řádku, kde byl skript přerušen. Vykreslení zarážky probíhá při odchycení zprávy WM_PAINT nebo při rolování obsahu *RichEditu*. Vzhledem k tomu, že zarážka se kreslí mimo oblast samotného objektu *RichEdit*, nedochází při překreslení ke smazání staré zarážky. Tu je tedy nutno odstranit ručně – je nutno zapamatovat si její starou polohu a následně ji smazat.

3.5. Ukládání a otevírání skriptů

Pro praktickou použitelnost programu je samozřejmě nutností ukládání a následné otevírání skriptů. Informace se ukládají do klasického textového souboru, který má ale pevně daný formát. Na začátku jsou pomocí specielních značek postupně uloženy údaje o všech zarážkách – tzn. komentář, který k nim přísluší a jména všech sledovaných proměnných. Za údaji o zarážkách následuje další oddělující symbol a samotný zdrojový kód skriptu. Ten se ukládá tak, jak je napsán (tzn. před překladem).

4. Implementace

4.1. Základní třídy projektu a jejich spolupráce

Program AlgoShow je napsán v jazyce C++ s použitím knihovny MFC. V následující kapitole najdete seznam nejdůležitějších tříd projektu, popis jejich účelu a vzájemné spolupráce. Fungování tříd a jejich metod není probíráno do detailu – podrobnější informace lze nalézt přímo ve zdrojovém kódu, kde každou funkci doprovází komentář objasňující její účel. U složitějších funkcí je okomentován i princip jejich fungování.

- **třída aplikace – CAlgoShowApp**

  je potomkem třídy *CWinApp* knihovny MFC a reprezentuje samotnou aplikaci. Byla vygenerována programem MS Visual Studio a do jejího kódu nebylo zasahováno.

- **třída hlavního dialogového okna – CAlgoShowDlg**

  je třídou hlavního okna aplikace AlgoShow, jedná se o jednu z nejrozsáhlejších částí programu. Obsahuje ovládací prvky jak pro editování, tak prohlížení skriptů. Přepínání mezi těmito dvěma módy dochází pomocí metod *SetShowMode()* a *SetEditMode()* – v nich dochází pouze k nastavování viditelnosti a popisků jednotlivých ovládacích prvků tak, jak je požadováno od daného módu (včetně načtení příslušného menu).

  Součástí této třídy jsou také metody sloužící k ovládání běhu skriptu, které jsou volány jako reakce na příslušné události (stisk tlačítka, výběr položky v menu či klávesová zkratka). První z těchto metod je *CheckScript()*, která uloží text z *RichEditu* do paměti a zavolá příslušnou funkci na kontrolu syntaxe. V případě, že byla nalezena chyba (ve volané funkci byla vyhozena výjimka), zobrazí upozornění a ukáže uživateli, kde k chybě došlo (přejde na řádek, který ji obsahuje). Další metodou je *RunScript()* – ta spouští běh skriptu. Nejprve zavolá předchozí metodu ke kontrole syntaxe, a pokud je vše v pořádku, přepne do prohlížecího módu a zavolá funkci spouštějící skript. Potom čeká buď na úspěšné dokončení skriptu či jeho přerušení (naražení na zarážku nebo běhová chyba). Obdobnou metodou je *ContinueScript()* zajišťující pokračování přerušeného běhu a čekání na případné další přerušení. Poslední metodou tohoto typu je *CancelScript()*. Ta ukončí prohlížení skriptu a vrátí se do režimu editace.

Další důležitou metodou této třídy je *LoadBreak()* obsluhující zobrazení informací během přerušení algoritmu. V případě, že k přerušení dojde, zobrazí komentář příslušející dané záložce a hodnoty proměnných, které se mají sledovat. Nastaví také polohu grafické zarážky v závislosti na čísle řádku, kde byl skript přerušen (není-li řádek ve viditelné části, pak odroluje obsah tak, aby vidět byl). Součástí metody je také zjištění hodnot sledovaných proměnných, které probíhá pomocí funkcí používaných při interpretaci – díky tomu je umožněno i sledování hodnot složitějších konstrukcí (vlastnosti grafů či ukazatelů). Obdobně lze vkládat hodnoty proměnných do komentáře – dochází k nahrazení konstrukce *$<proměnná>$* za hodnotu obsaženou v požadované proměnné – pokud ta neexistuje či její typ není podporován, dojde k nahrazení prázdným řetězcem. Editace záložek (tj. přidávání komentářů, sledovaných proměnných a pojmenování záložek) je také ošetřena pomocí členských metod této třídy.

Otevírání a ukládání skriptů je řešeno pomocí metod *OpenScriptToEdit(), SaveToFile(), SaveAs()* – ty pracují se soubory speciálního formátu popsaného v podkapitole 3. 5. Pomocí čtyř různých metod je pak ošetřeno otevření některého z nástrojů usnadňujících práci (viz třídy *CGrafDialog* a *CToolsDialog*), obdobou je pak otevření dialogu pro nastavování vlastností programu. Poslední podstatnou metodou třídy hlavního dialogu je *OnZviditelnit()*, která slouží k přidání či odebrání hvězdičky ze začátku řádku skriptu. Tato metoda může být volána po použití příslušné volby v kontextovém menu, případně po stisknutí příslušné klávesové zkratky (Ctrl+B).

Z reakcí na události hlavního dialogu stojí za zmínku ošetření *WM_ON_SIZE*, která nastává při změně velikosti okna a při které dochází k přizpůsobení velikosti a pozice jednotlivých ovládacích prvků. To je řešeno za použití knihovny *Resizer.h* (viz. [4]). Ta ovšem z mně neznámého důvodu nezvládá korektní přesouvání komponent *GroupBox*, které se tak musí řešit zvlášť. Reaguje se také na otevření hlavního menu, a to kontrolou dostupných funkcí a nastavení možnosti jejich spuštění z menu. Pokud dojde ke změně textu v prvku *RichEdit* sloužícímu k editaci kódu, zavolá se příslušná metoda barvící syntax, při změně výběru zarážky pak dojde k načtení aktuálních údajů. Velkou skupinou jsou pak metody na ošetření příkazů v menu – ty jsou ale většinou krátké a jednoduché na pochopení, proto se jim zde nebudu podrobněji věnovat.

- **třída statického obrázku – AlgPictureCtrl**

  je oddděná od třídy *CStatic* a slouží k zobrazování průběhu algoritmu. Její jedinou podstatnou metodou je *OnPaint()* reagující na událost WM_ON_PAINT, která nastane při nutnosti překreslení ovládacího prvku. Z této metody je zavolána speciální funkce na vykreslení všech grafických komponent.

- **třída dialogu pro hledání – CFindDialog**

  je odvozena od třídy *CDialog* a slouží uživateli pro zadávání dat při hledání či nahrazování textu v editovaném kódu. Dialog má dva základní režimy fungování – pro vyhledávání a pro nahrazování – mezi nimi se přepíná pomocí metod *SetFindMode()* a *SetReplaceMode()*. Součástí jsou tři metody pro ošetření stisku tlačítek *Najít*, *Nahradit* a *Nahradit vše*. Ty pak zavolají příslušné ošetření za pomoci členských funkcí příslušného ovládacího prvku, ve kterém se hledá. Poslední reakcí na události je změna textu pro vyhledávání – pokud je políčko prázdné, je znemožněno spustit hledání.

- **třída dialogu pro editaci grafu – CGrafDialog**

  je relativně rozsáhlým potomkem *CDialog*. Slouží k editaci grafů – ať již pro účel generování kódu a nebo pro zadání grafu za běhu skriptu. Většinu plochy dialogu zabírá komponenta, do které se vykresluje editovaný graf. Její velikost se automaticky mění s velikostí okna (ošetření v reakci na událost ON_WM_SIZE). Uživatel může přepínat mezi čtyřmi základními režimy – přidáváním a editací vrcholů, resp. přidáváním a editací hran. Uživatel graf interaktivně „naklikává“ – takže je nutná metoda reagující na stisk levého tlačítka myši uvnitř grafu. V případě módu přidávání vrcholů se nemusí nic řešit a vrchol se prostě vsadí na místo kliknutí. Při editaci vrcholů je ale nutno z údajů o grafu a pozice kurzoru vyčíst, zda byl kliknutím zasažen nějaký vrchol a ten označit jako vybraný. Obdobné je to i při přidávání hran. Při jakékoliv změně pak dojde k překreslení grafu – je využitá stejná funkce jako při kreslení grafu při prohlížení algoritmu, jen je zde navíc možnost zvýraznit vybraný vrchol. Součástí dialogu je celá řada tlačítek a jiných ovládacích prvků pro nastavení vlastností hran a vrcholů – o těch se ale nebudu podrobněji rozepisovat., protože jejich ošetření je většinou triviální. Jediným trochu zajímavým případem je změna názvu vrcholu.

Vzhledem k tomu, že graf je indexován názvy vrcholů, tak není možné jméno změnit přímo, ale je zapotřebí vrchol se starým názvem z grafu vyjmout a opět ho přidat jako nový vrchol s pozměněným jménem. Poslední zajímavou funkcí je generování kódu při kladném ukončení dialogu ve chvíli, kdy slouží jako nástroj pro vložení grafu do skriptu. Pro tento účel je třídě dialogu předán ukazatel na prvek *RichEdit*, kam se má vytvořený kód vložit. Generují se celkem tři části – na začátek se vloží deklarace proměnné typu *Graph* a definice hlavičky inicializační funkce (ve tvaru *Init_Graph_<název grafu>()*). Na konec skriptu se pak přidá samotné tělo inicializační funkce, kde probíhá přidávání vrcholů a hran včetně nastavení jejich vlastností. Poslední vloženou částí je zavolání inicializační funkce – tento řádek se vloží na místo, kde se nacházel kurzor před spuštěním nástroje na vložení grafu.

- **třída dialogu pro zadání hodnoty – CReadDialog**

  je potomkem *CDialog* a je relativně jednoduchá – obsahuje jen jednu podstatnou metodu *OnBnClickedOk()* ošetřující korektní ukončení dialogu a ukládající zadanou hodnotu proměnné. Na závěr ještě znovu spustí skript a čeká na případné další přerušení.

- **třída dialogu nastavení – CSettingsDialog**

  je opět odvozena od třídy *CDialog* a reprezentuje rozhraní sloužící k nastavení některých vlastností aplikace. Konkrétně se jedná o barvy, jakými se vybarvuje syntaxe (celkem 3 barvy: klíčových slov, komentářů a řetězců), velikost textu při editaci a prohlížení kódu a velikost bufferu *Undo* historie (může být v rozmezí 1 až 100). Z metod stojí za zmínku snad jen *OnPaint()*, která zajišťuje vykreslení aktuálně vybrané barvy vedle tlačítka pro její výběr, a *SaveSettings()* ukládající nastavení programu do *ini* souboru.

- **třída dialogu nástrojů – CToolsDialog**

  je poslední dialogovou třídou (potomek *CDialog*) zajišťující funkci nástrojů na vkládání legend, šipek a textů. Těmto třem variantám odpovídají i tři módy a metody, které tyto režimy nastavují. Hlavními výsadami této třídy jsou ale metody na vygenerování kódu a jeho vložení do editovaného skriptu. V případě šipek a textů se jedná pouze o jednořádkový kód – zavolání příslušné funkce s parametry získanými z dialogu. Při vkládání legendy, která je implementována jako jednoprvkové pole stringů, je ale zapotřebí postup obdobný

generování grafu, tj. přidání deklarace pole a hlavičky inicializační funkce na začátek, těla inicializační funkce na konec a samotné její zavolání na místo, kde se nachází kurzor.

- **třída pro editaci a prohlížení kódu – SyntaxRichEditCtrl**

  je potomkem třídy *CRichEditCtrl* sloužící pro pohodlnější psaní a prohlížení zdrojových kódů. Obecný popis způsobu rozšíření funkčnosti najdete v podkapitole 3.4, nyní se podívejme na konkrétní implementaci Asi nejdůležitější přidanou metodou je *VybarviSyntax()* zajišťující, jak již název napovídá, vybarvení kódu na základě syntaxe skriptovacího jazyka. V jejím průběhu jsou postupně nalezena a vybarvována klíčová slova, komentáře a řetězce. Navíc dochází k vysázení řádků začínajících hvězdičkou tučným písmem. Aby bylo zabráněno nepříjemnému blikání ovládacího prvku, je na začátku metody zakázáno jeho překreslování. To je pak na jejím konci opět povoleno. Kvůli funkci barvení syntaxe ale nelze použít pro návrat v historii editace vestavěnou funkci třídy *CRichEditCtrl* a bylo zapotřebí napsat vlastní metody *Undo()* a *Redo (). Ty* využívají funkcí podtřídy *CMyUndoBuffer*, o které bude řeč za chvíli. Kvůli změně implementace ukládání historie bylo třeba napsat i vlastní funkce pro vkládání, vyjímání a mazání textu – fungují stejně jako metody třídy *CRichEditCtrl*, jen s tím rozdílem, že ukládají své změny do bufferu historie. Ze stejného důvodu bylo nutno přepsat i metody zajišťující hledání a nahrazování textu.

  - **podtřída CMyUndoBuffer**

    je navržena pro účel ukládání změn. Obsahuje dva seznamy – pro operace *Undo* a *Redo*. Každá operace se ukládá do čtyř-prvkové struktury čítající text před změnou a text po změně a taktéž rozsah výběru před a po editaci. Na základě těchto údajů je snadné jakoukoliv změnu vrátit či opakovat.

- **třídy ukládající data o běžícím skriptu**
  - **Prom**

    je hlavní třída uchovávající v sobě veškerá data o průběhu algoritmu. Její součástí jsou seznamy všech globálních proměnných, údaje všech funkcích (včetně těch vestavěných) a zásobník volání funkcí. Kromě toho je její součástí i seznam zarážek obsahující údaje, které se mají při přerušení na daném místě zobrazit. Pro uživatele neviditelnou, ale podstatnou součástí je i seznam všech naalokovaných ukazatelů (do seznamu se přidá při každém volání

funkce *new*). Tento seznam slouží pro dealokaci paměti po ukončení skriptu (to je implementováno v destruktoru třídy). Součástí je i několik metod pro zjišťování informací o proměnných. První z nich je *GetType(string <název_proměnné>)* zjišťující typ proměnné (existuje celkem 11 možností), v případě její neexistence vrací –1. Pro práci s grafy je k dispozici celkem pět metod (pro každý grafový typ jedna), které na základě názvu proměnné vracejí ukazatel na požadovaný prvek (je-li hledání neúspěšné, vrací nulový ukazatel). Pomocí metody *clear()* lze pak vymazat veškerá data o běhu algoritmu a skript tak ukončit (dochází i k dealokaci ukazatelů). Tato třída obsahuje také jednu metodu používanou při interpretaci kódu. Je jí *CallFunction(string hlavicka)* zajišťující zavolání funkce s danou hlavičkou volání. V jejím průběhu dojde ke zkopírování parametrů do požadované funkce a k jejímu následnému zavolání. Funkce vrací instanci třídy **TNavrat**, jejímž obsahem může být hodnota všech čtyř typů, které je možno z funkce vrátit (*int*, *double*, *string*, *Cell**).

- **TFunkce**

  je třídou uchovávající informace o jedné instanci funkce. Kromě seznamu všech lokálních proměnných uchovává informaci o jméně funkce, způsobu, jakým byla zavolána (hlavička volání), návratovém typu, počtu, typu a názvů parametrů. Pro běh skriptu je nezbytný i údaj o rozsahu řádků, na kterých se funkce nachází a čísle řádku, který se aktuálně zpracovává. Samotná třída nenabízí žádné speciální metody, pouze dva typy konstruktor – bezparametrický a konstruktor přijímající název a seznam parametrů funkce.

- **TSipka**

  nese informaci o grafickém objektu šipka. Kromě údajů o jejím vzhledu obsahuje informace o objektech, které spojuje (může se jednat buď o buňku pole, nebo o ukazatel), a o směru, ve kterém je šipka k těmto objektům připojena.

- **TGraph**

  je třídou reprezentující graf. Její součástí je seznam všech vrcholů indexovaný jejich jmény a seznam všech hran (indexovaný dvojicí vrcholů, které spojuje). Dále jsou její součástí údaje o některých grafických vlastnostech a o tom, zda je graf orientovaný. Kromě konstruktoru obsahuje třída další čtyři metody pro práci s grafem. Implementačně jednoduchá

je metoda *AddVertex()* přidávající vrchol do grafu – stačí pouze přidat vrchol do seznamu. O něco náročnější je *AddEdge()*, která přidává hranu. Zde je nutno nejen přidat hranu, ale v případě, že graf není orientovaný, je zapotřebí přidat i hranu opačnou. Funkce pro práci s grafy totiž chápou všechny grafy jako orientované a proto je nutné u grafů neorientovaných hrany zdvojovat. Navíc se ještě v těle této metody musí do výchozích vrcholů hrany (u neorientovaných grafů do obou vrcholů) uložit údaj o tom, že z něho existuje hrana do jiného vrcholu. Reverzní metodou je *DeleteEdge()* odstraňující hranu z grafu. Poslední členskou funkcí třídy je *DeleteVertex()*, která nejprve musí odstranit všechny hrany vedoucí z/do vrcholu, který se má smazat. Pak již jen odstraní vrchol ze seznamu.

- **TBunka**

  reprezentuje jednu buňku pole. Obsahuje údaj o typu, který je v ní uložen (může být *int*, *double* nebo *string*), a také samotnou hodnotu jejího obsahu. Kromě toho je součástí celá řada grafických vlastností, na základě kterých se buňka poté vykresluje. Třída neobsahuje žádnou složitější metodu – pouze jednu pro nastavení výchozích grafických vlastností a čtyři druhy konstruktorů: bezparametrický a tři konstruktory pro všechny podporované typy, které zkopírují hodnotu svého parametru do hodnoty buňky a její grafické vlastnosti nastaví na výchozí.

- **Pole**

  je třída reprezentující libovolné pole (tzn. všech tří povolených typů). Mimo údaje o druhu pole a o jeho grafických vlastnostech obsahuje tři seznamy, které skrývají informace o jednotlivých buňkách pole. Dva z těchto seznamů budou vždy prázdné (použit bude jen seznam typu odpovídajícího typu pole). Třída obsahuje celkem dva konstruktory – bezparametrický a konstruktor, který vytvoří prázdné pole o požadované velikosti a vzhledu. Metoda *Size()* slouží k zjištění velikosti pole. Dále jsou součástí dvě členské funkce pro zjišťování resp. nastavování vzhledu některé z buněk pole. Velice důležité pak jsou dvě přetížené metody *Get(...) a Set(...)*, které nastaví resp. získají hodnotu obsaženou v poli pod daným indexem (pole je indexováno od nuly). V případě přístupu na neplatný index pak dojde k vyhození výjimky reprezentující běhovou chybu.

- **některé menší třídy a struktury**
  - **TCykly –** informace o rozsahu cyklu a podmínce jeho pokračování
  - **TEdgeIterator –** data o iterátoru na hranách grafu
  - **TVertexIterator –** obdoba pro vrcholy grafu
  - **HranaSVahou –** obsahuje operátor „<" pro porovnání hran na základě jejich váhy
  - **THrana –** údaje o hraně grafu
  - **TVrchol –** údaje o vlastnostech vrcholu
  - **GrafovyVrchol –** struktura obsahující data o vrcholu a hran z něho vedoucích
  - **TTexty –** uchovává vlastnosti grafického nápisu
  - **TBunkaPole –** data o vzhledu buňky pole
  - **Breaks –** údaje o zarážce (její komentář a seznam sledovaných proměnných)

- **cizí třídy použité v programu (viz. [4])**
  - **CResizer**

    je jedinou mnou nevytvořenou třídou v programu a zajišťuje automatickou změnu velikosti a pozice ovládacích prvků při změně velikosti okna.

4.2. Implementace kontroly syntaxe a překladu

Kontrola syntaxe zapsaného kódu a jeho upravení pro následnou interpretaci není implementována uvnitř jedné třídy, ale je tvořena řadou samostatných funkcí. V následujících odstavcích si popíšeme jen ty nejdůležitější z nich. Hlavní funkcí je *CheckCode(...)*, která zajišťuje zkontrolování kódu skriptu. Jako parametr dostane uživatelem zapsaný kód a vrací kód upravený pro interpretaci a také tabulku pro převod čísel řádků, na kterých se vyskytují zarážky (pro možnost zobrazení aktuální polohy ve skriptu). Po odstranění komentářů a načtení seznamu vestavěných funkcí následuje samotná kontrola kódu – ta spočívá v opakovaném volání funkce *CheckDeklarace(...)* do doby, než se dojde na konec skriptu.

Funkce *CheckDeklarace(...)* může pracovat ve dvou režimech – buď kontroluje deklarace v těle funkce nebo deklarace globální. V obou případech mohou být deklarovány proměnné všech podporovaných typů a jsou uloženy pomocí funkce *AddVar(...)* do tabulky proměnných (před jejich uložením je zkontrolována jedinečnost a správnost jejich názvu).

Pokud za názvem následuje znak „=“, dojde také ke kontrole, zda je možné přiřadit do proměnné danou hodnotu. Nacházíme-li se v části globálních deklarací, je také možno definovat hlavičku funkce či její tělo. Pokud se jedná o definici těla funkce, zkontroluje se nejprve, zda odpovídá předem deklarované hlavičce a pokud ano, zavolá se *CheckCodeBlock(...)*, která kontroluje jednotlivé příkazy v těle funkce.

Pro kontrolu správnosti zápisu jednotlivých příkazů slouží *CheckCommand(...)*. Ta zkouší postupně volat pět funkcí (každá pro jiný typ příkazu – přiřazení, deklarace, cyklus nebo podmínka, volání funkce, speciální vestavěné funkce), které na základě podoby příkazu usoudí, zda je v jejich kompetenci ho zkontrolovat a případně tak učiní (více o principu jejich fungování najdete v podkapitole 3.1). Pro jednotlivé varianty příkazů pak existuje mnoho pomocných funkcí, jejichž pochopení je ale snadné ze zdrojového kódu a jeho komentáře, a proto je zde nebudu popisovat.

O trochu zajímavějším problémem byla kontrola pravdivostních a jiných výrazů. Pokud se narazí ve skriptu na místo, kde by měl následovat výraz nějakého typu, následuje jeho kontrola. Jednodušší variantou je výraz, který má být vyhodnocen jako řetězec. V tomto případě se může jednat pouze o konstantní řetězce a proměnné typu *string* pospojované pomocí znaménka „+“. Tato varianta je ošetřena funkcí *Get_String(...)*. Zajímavější možností jsou číselné výrazy řešené funkcí *GetCislo(...)*. U nich se mohou kromě konstantních čísel objevovat proměnné typu *int* a *double*. Mezi nimi se pak může vyskytovat vždy právě jedno znaménko (s výjimkou unárního minus). Poslední věcí, kterou je nutno zkontrolovat, je korektní uzávorkování. Nejsložitější variantou jsou ale výrazy pravdivostní, jejichž správný zápis ověřuje funkce *Get_Bool(...)*. Celý pravdivostní výraz je složen z jednoduchých pravdivostních výrazů (kontrolovaných funkcí *GetSimpleBool(...)*) pospojovaných pomocí operátorů „&&“ a „||“. Jednoduchými výrazy rozumíme buď porovnání dvou výrazů nebo proměnnou číselného či ukazatelového typu, která se testuje na (ne)rovnost nule. U porovnání je složité zejména zajištění, aby na obou stranách byly výrazy stejného typu. Typ výrazu se zjišťuje pomocí jeho prvního členu – může se jednat o název proměnné (pak typ výrazu odpovídá typu proměnné), zápis čísla (pak je to výraz číselný) či konstantního řetězce. Následně stačí již jen ověřit, zda na daném typu je dané porovnání povoleno (například u ukazatelů je povolen pouze test na (ne)rovnost, ale není možno použít operátory „<“ či „>“). I

zde je nakonec zapotřebí zkontrolovat korektní uzávorkování. Navíc je nutno vyřešit oddělení závorek patřících k číselnému výrazu a závorek použitých v rámci výrazu pravdivostního.

4.3 Implementace interpretu jazyka

Stejně jako kontrola syntaxe a překlad, je i interpretace skriptu tvořena celou řadou samostatných funkcí. Za běhu algoritmu jsou veškerá data o něm uložena v třídách, které byly popsány výše (pokud je v následujících odstavcích zmíněno, že data uložíme, pak je tím myšleno právě do této struktury).

Vůbec první funkcí, která se volá ještě před začátkem samotné interpretace, je *PreProcess()*. Jedná se o jakousi inicializaci běhu skriptu. Během ní dojde ke zpracování deklarací globálních proměnných a uložení údaj o všech funkcích (jméno, návratový typ, parametry a rozsah řádků).

Stěžejní funkcí je ale ta s názvem *Go(...)*, která dostane jako parametr instanci třídy *TFunkce* obsahující veškeré údaje o funkci, která je interpretována. Jejím úkolem je interpretovat příkazy všeho druhu. Na většinu typů si volá funkce pomocné, pouze příkazy *BREAK* a *return* ošetřuje sama. V případě přerušení skriptu (pomocí *BREAK*) stačí vyhodit výjimku, která samotné přerušení zajistí. Varianta *return* pak zajišťuje ukončení funkce – stačí tedy ukončit funkci *Go(...)* a odebrat vrchol zásobníku volání funkcí.

Ve stručnosti se podívejme na jednotlivé funkce zajišťující zpracování konkrétních typů příkazů (více informací o principu fungování lze získat z komentáře zdrojového kódu). *Grafy(...)* interpretuje příkazy sloužící pro práci s grafy a jejich iterátory, k nimž je přistupováno pomocí tečkové notace. Obdobou je funkce *Grafika(...)*, která obsluhuje přístup a nastavení grafických vlastností pole, položek pole a ukazatelů. Pouze dva příkazy dokáže zpracovat funkce *InOut(...)*. Jedná se ale poměrně implementačně náročné ošetření uživatelského vstupu a výstupu. Výpis na obrazovku zajišťuje standardní metoda *MessageBox(...)* z knihovny MFC. Vstupy probíhají pomocí dvou dialogů (ty jsou předány jako parametry). Ve variantě *read* s parametrem typu graf se zobrazí dialog pro editaci grafu, v ostatních případech pak jednoduchý dialog pro zadání hodnoty jedné proměnné. Název funkce *Podminky(...)* je mírně matoucí, protože zpracování podmínek, které zde původně bylo, jsem nakonec přesunul jinam, a tak jediným jejím úkolem je provedení deklarace

lokálních proměnných a příkazu přiřazení (na tom je nejsložitější vyhodnocování výrazů popsané v následujícím odstavci). Samotná interpretace cyklů a podmínek je ošetřena ve funkci *Simple(...)*. Pokud je zjištěno, že se jedná o cyklus, je přidán na vrchol zásobníku cyklů spolu s řídící podmínkou. Jedná-li se o podmínku, je její tělo také přidáno na zásobník cyklů, ale s podmínkou, která dovolí jen jedno provedení těla. Poté následuje samotné cyklení spočívající v opakovaném zpracovávání těla cyklu, dokud platí řídící podmínka. Poslední funkcí ošetřující jednotlivé příkazy je *Funkce(...)*, která zajišťuje zavolání funkce ve skriptu. To je vcelku jednoduché – stačí vyhodnotit parametry, přidat volanou funkci na zásobník volání a rekurzivně spustit *Go(...)* s parametry odpovídajícími funkci, kterou spouštíme.

Jedinou složitější věcí při interpretaci, kterou jsme zatím neprobrali, je vyhodnocování výrazů. Prvním krokem je nahrazení názvů proměnných za jejich hodnoty. Pomineme relativně snadné vyhodnocení sřetězování *stringů* a podíváme se na zpracování výrazů číselných. K tomuto účelu byla vytvořena třída *Strom* reprezentující uzel binárního stromu obsahující jeden element (číslo nebo znaménko). Konstruktor třídy dostane jako parametr číselný výraz. Ten rozebere podle priorit operací a do kořene dosadí znaménko s prioritou nejnižší (je-li jich na tom více stejně, vybere se to první). Výraz se tak rozpadne na dvě části, které se pošlou jako parametry pro konstrukci dvou synů. Takto se pokračuje, dokud nejsou ve všech listech konstanty (tj. čísla). Následně se zavolá metoda *Vypocti()*, která zajistí výpočet stromu odspodu. Obdobný postup je uplatněn i u třídy *BoolStrom* pro vyhodnocování pravdivostních výrazů. Jediným rozdílem je jiná podoba znamének a s nimi souvisejících operací.

5. Program AlgoShow z pohledu uživatele

5.1. Obecně

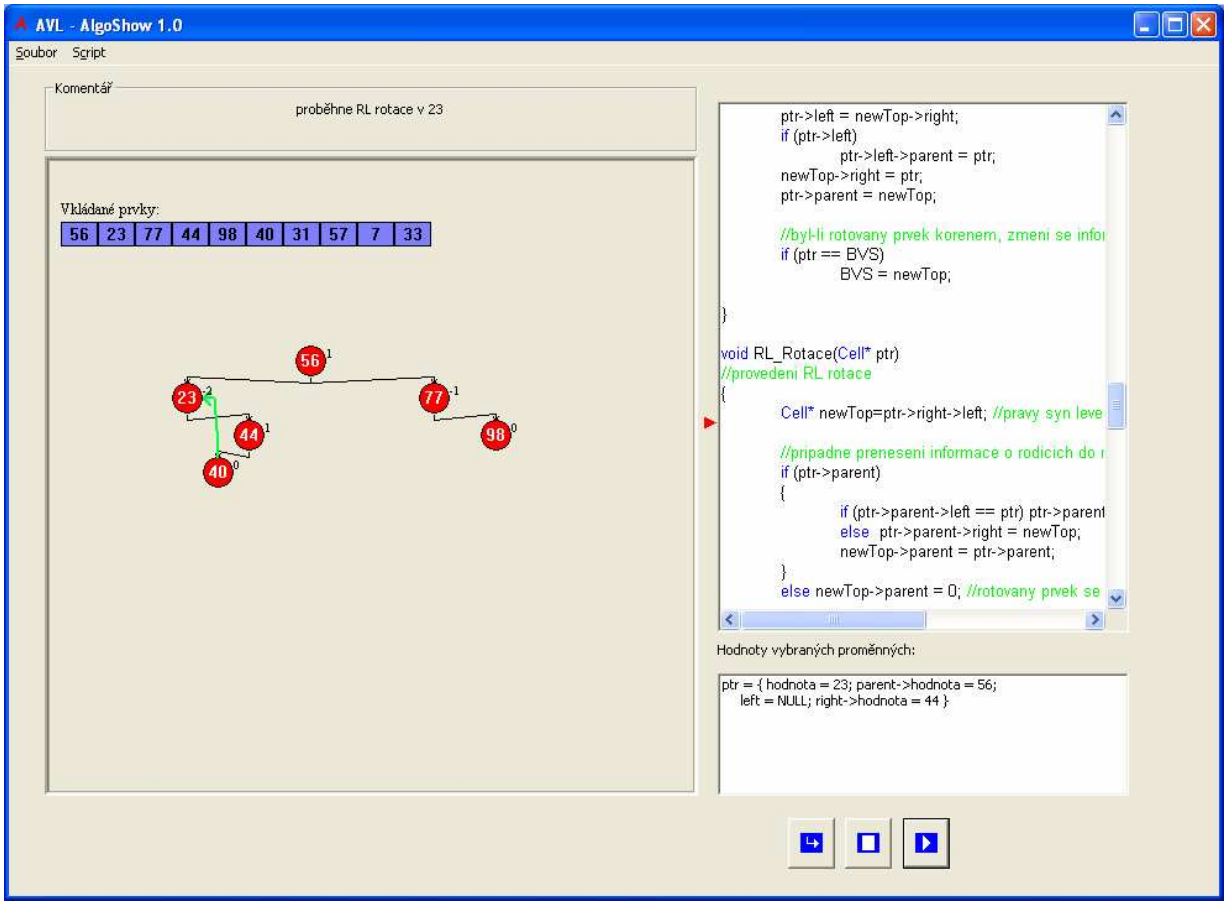
Aplikace AlgoShow je nástroj sloužící ke snadnějšímu pochopení algoritmů pomocí vizualizace jejich běhu. Algoritmy nejsou v programu zabudovány napevno, ale uživatelé si mohou vytvářet své vlastní pomocí skriptů.

Program je určen pro dvě skupiny lidí – pro ty, kteří chtějí algoritmy jen prohlížet a pro ty, kteří je chtějí i vytvářet. Prohlížení vizualizací je snadné a zvládne ho téměř každý, u psaní skriptů je ale potřeba nastudovat alespoň základy skriptovacího jazyka (vychází z jazyka C). Není však nutné ovládat všechny grafické funkce, protože součástí aplikace je několik nástrojů, které psaní skriptu usnadňují.

5.2. Prohlížení algoritmů

Prohlížení již existujících vizualizací je velice snadné. Po startu aplikace se dostaneme do části sloužící k editaci skriptu. Zde je jedinou nutností otevřít soubor se zvoleným algoritmem a spustit ho. Otevření můžeme provést volbou z menu *Soubor – Otevřít* nebo pomocí klávesové zkratky Ctrl+O. Poté vybereme skript pro otevření (soubory s příponou *.alg*). Poslední věcí, kterou musíme udělat, je samotné spuštění vizualizace. To můžeme provést třemi způsoby: volbou z menu *Script – Spustit*, klávesou F5 nebo stiskem tlačítka v pravém dolním rohu okna. Pokud dojde místo spuštění k ohlášení syntaktické chyby, je skript špatně napsaný a nemůže být spuštěn. V tom případě je vhodné kontaktovat autora skriptu, případně se skript pokusit opravit svépomocí.

V tuto chvíli dochází k přepnutí programu do režimu prohlížení vizualizací. Pohyb v této části je již intuitivní. Rozložení ovládacích prvků si popíšeme na základě následujícího obrázku.

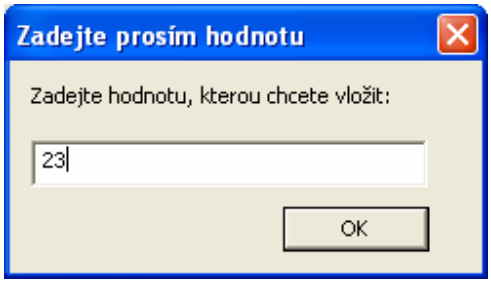


Obr. 5.2.1 – režim prohlížení vizualizací

Nejpodstatnější částí obrazovky je plocha, na které se zobrazují grafické prvky algoritmu (tj. pole, nápisy, šipky, stromy apod.). Ta se nachází v levé části. Přímo nad ní se pak zobrazuje komentář k aktuální zarážce. Na pravé straně lze pak najít zdrojový kód algoritmu (pouze pro čtení, nelze ho měnit). Aktuální pozice v něm je pak označena červenou zarážkou ve tvaru trojúhelníku. Pod kódem pak najdeme seznam proměnných, které určil autor skriptu jako podstatné, a jejich aktuální hodnoty. Důležitá jsou tři tlačítka umístěná v pravé dolní části. To nejvíce vpravo, jak již jeho ikona napovídá, slouží k pokračování v přerušeném skriptu. Stejného efektu lze dosáhnout i pomocí menu a volby *Script – Pokračovat* nebo klávesou F5. Prostřední tlačítko ukončí běh algoritmu a vrátí se do editační části programu. K tomuto účelu lze použít i volbu *Script – Ukončit script* či kombinaci kláves Shift+F5. Poslední možností je restart skriptu, čehož lze dosáhnout kliknutím na nejlevější tlačítko, volbou *Script – Restart* či stiskem kláves Ctrl+Shift+F5. V menu lze ještě nalézt dvě

další volby, a to ukončení programu (Alt+F4) a otevření jiného skriptu (Ctrl+O). Pokud se rozhodnete otevřít skript, vrátíte se do režimu editace a musíte algoritmus spustit, ať již z menu, tlačítkem či stiskem F5.

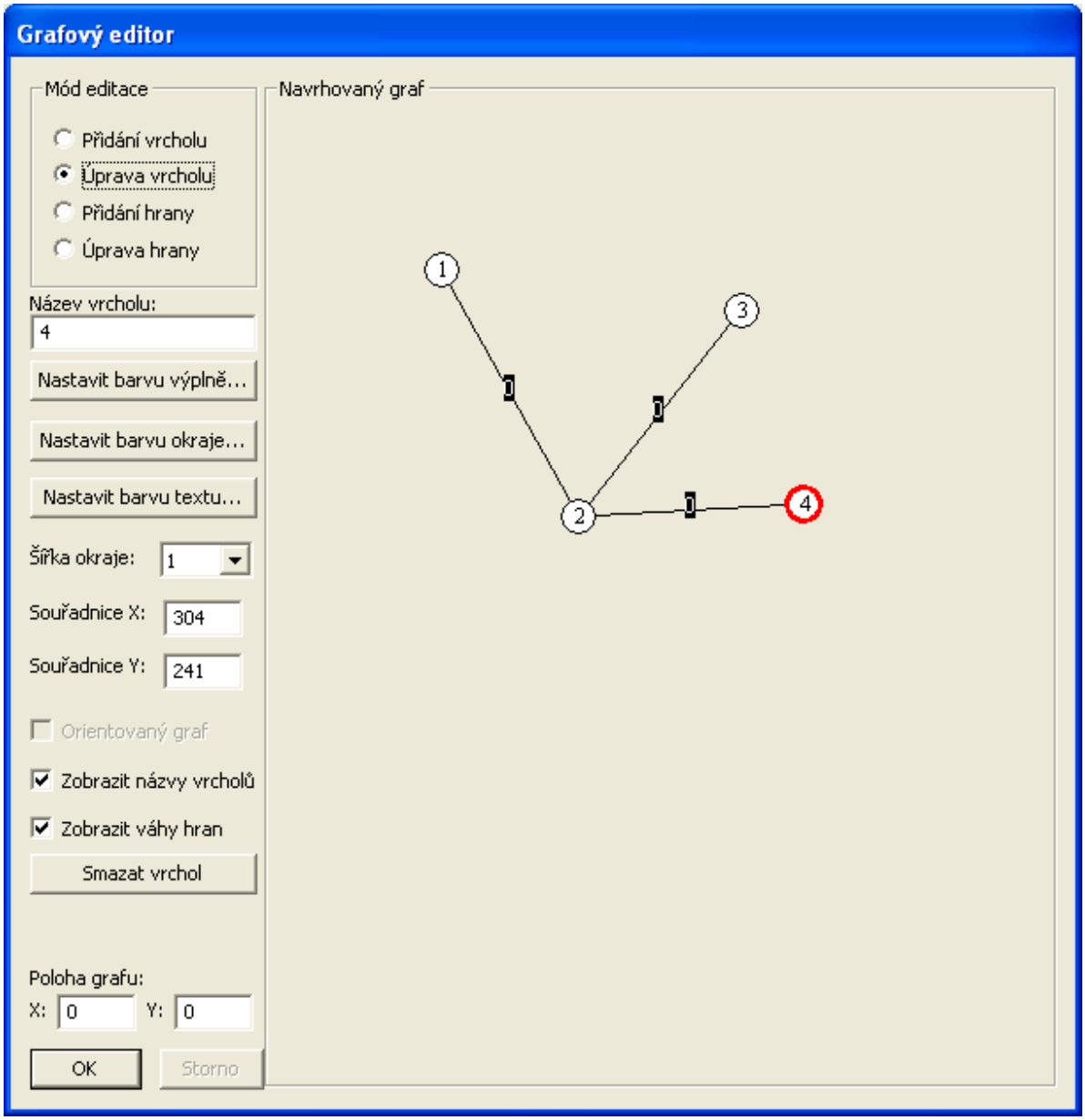
Během prohlížení vizualizace po vás může být požadováno zadání hodnot některých proměnných (např. při vkládání prvku do stromu). Jedná-li se o obyčejnou proměnnou obsahující číslo či řetězec, objeví se pouze jednoduchý dialog (Obr. *5.2.2*), kde stačí zadat hodnotu a potvrdit tlačítkem *OK*.



Obr 5.2.2 – dialog pro zadání hodnoty proměnné

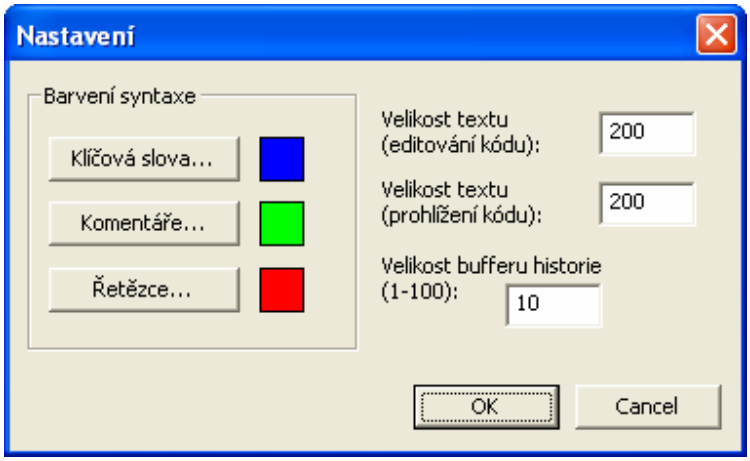
Další možností je zadání grafu – v tom případě se zobrazí o trochu složitější dialog (*viz. Obr. 5.2.3*). Úkolem uživatele je „naklikat“ graf. Vpravo nahoře je možno přepínat mezi čtyřmi režimy: přidáváním vrcholu, úpravou vrcholu, přidáváním hrany a úpravou existující hrany. Chceme-li přidat vrchol, nastavíme nejprve jeho požadované vlastnosti (barvy textu, výplně a okraje, šířku okraje). Samotné přidání proběhne kliknutím levého tlačítka myši do oblasti grafu (vrchol se přidá na místo, kde bylo kliknuto). Chceme-li editovat existující vrchol, přepneme se do příslušného módu a levým tlačítkem vybereme vrchol, který chceme změnit. Kromě barev a šířky okraje můžeme upravit i jeho název či souřadnice. Označený vrchol můžeme také smazat (dojde i ke smazání všech hran, které do/z něho vedou). Pro přidání hrany do grafu se přepneme do odpovídajícího režimu a nastavíme vlastnosti, které má vkládaná hrana mít (barva, šířka, váha). Vložení hrany pak proběhne snadno pomocí kliknutí na vrcholy, které má hrana spojovat. U orientovaného grafu směřuje hrana k vrcholu, na který bylo kliknuto jako na druhý. Pokud chceme editovat existující hranu, přepneme se do stejnojmenného módu. Ze seznamu vybereme nejprve výchozí vrchol hrany a následně vrchol cílový. Poté již můžeme nastavit váhu, barvu a šířku hrany, případně hranu smazat. Nastavovat můžeme též vlastnosti celého grafu. Můžeme si určit, zda se mají zobrazovat

názvy vrcholů a váhy hran. Dále je možno nastavit, zda se jedná o orientovaný graf (lze nastavovat jen, pokud nemá zatím žádné hrany). Poslední možnou volbou je určení souřadnic grafu. Ty udávají, od jakého bodu se počítají relativní souřadnice vrcholů. Změna se neprojeví při náhledu na editovaný graf, ale až při jeho zobrazení za běhu algoritmu. Pokud již je nastaveno vše potřebné, můžeme naši volbu potvrdit tlačítkem *OK* a pokračovat v prohlížení vizualizace.



Obr. 5.2.3 – editování grafu za běhu algoritmu

Poslední věcí, kterou by mohl uživatel prohlížející algoritmy potřebovat, je nastavení některých vlastností programu. K těm se dostaneme pomocí menu (*Nastavení – Možnosti*). Zobrazený dialog (*viz. Obr. 5.2.4*) nám nabízí možnost změn barev, kterými se barví syntaxe zdrojového kódu (jak v režimu prohlížení, tak v módu editace). Dále je možno nastavit velikost textu (pro každý režim zvlášť). Poslední volbou je počet kroků, o kolik je možno se vrátit v historii editace kódu. To však využijí jen uživatelé vytvářející vlastní algoritmy.

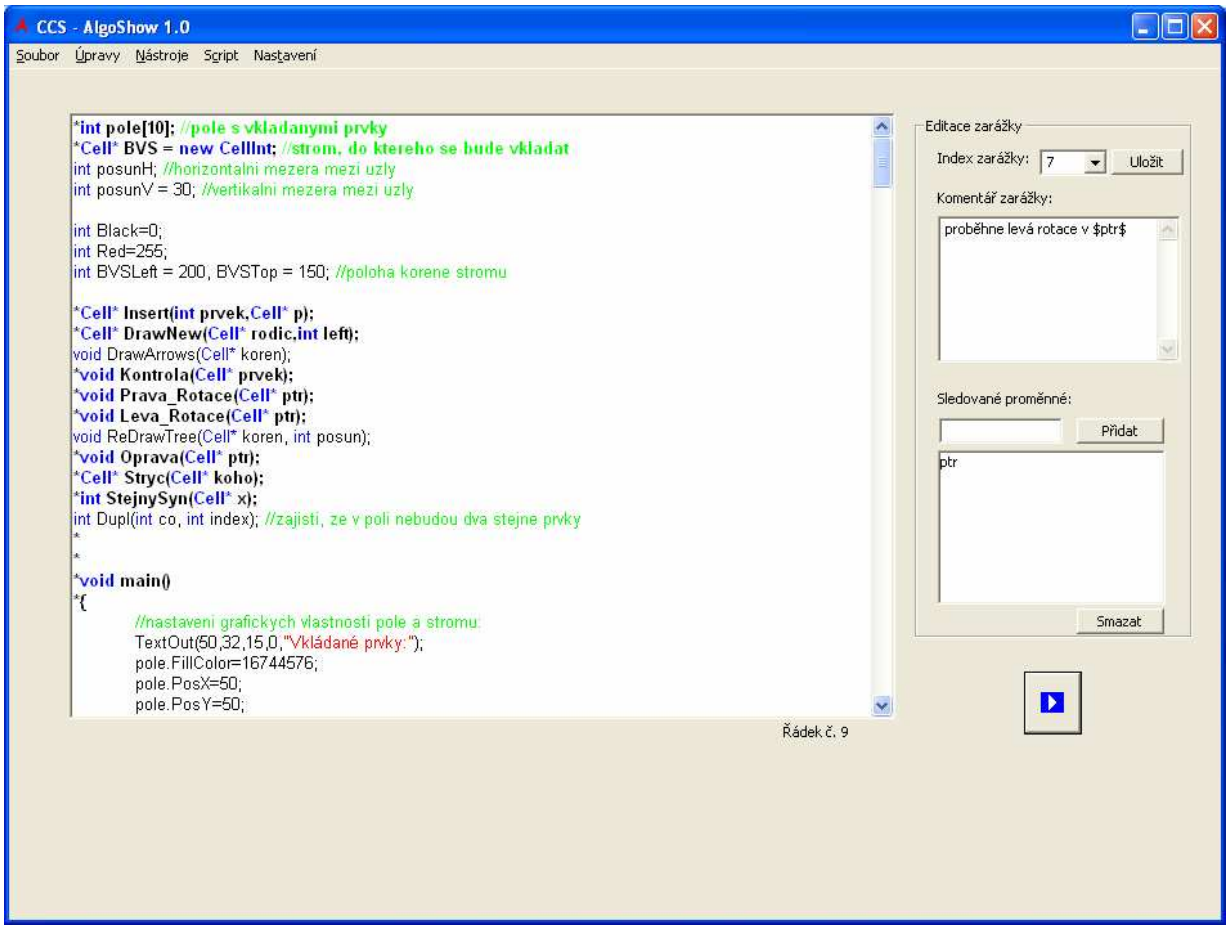


Obr. 5.2.4 – nastavení programu

5.3. Editování skriptů

Pokud chceme vizualizace nejenom prohlížet, ale i vytvářet, bude nutné si kromě skriptovacího jazyka (jeho popis se nachází v další podkapitole) osvojit i prostředí, v němž k editaci skriptů dochází. Do tohoto prostředí jsme přepnuti ihned po startu aplikace.

Můžeme se buď pustit do tvorby zcela nového skriptu (volba v menu *Soubor – Nový*, případně kombinace Ctrl+N) nebo upravovat nějaký již existující. Ten otevřeme opět buď pomocí hlavního menu (*Soubor – Otevřít*) nebo klávesami Ctrl+O. Nyní se už můžeme pustit do samotného návrhu algoritmu. Ten spočívá v napsání příslušného zdrojového kódu ve skriptovacím jazyce a nastavením údajů, které se mají zobrazit při jednotlivých přerušeních běhu algoritmu. K tomu nám slouží prostředí, které vidíme na následujícím obrázku.



Obr. 5.3.1 – prostředí pro editaci skriptů

Nejdůležitější součástí je prostor pro editaci zdrojového kódu nacházející se v levé části okna. Tam můžeme kromě vepisování kódu použít také kontextové menu, které se zobrazí po kliknutí pravým tlačítkem myši. To nabízí kromě klasických voleb pro návrat v historii editace či kopírování a vkládání textu také možnost *Z(ne)viditelnit řádek*. Ta slouží k přidání resp.odebrání hvězdičky ze začátku řádku. Řádky s hvězdičkou na začátku jsou viditelné pro uživatele při prohlížení skriptu, zatímco ostatní mu zůstávají skryty. Stejného efektu můžeme dosáhnout také klávesovou zkratkou Ctrl+B. Kontextová nabídka nám také umožňuje spustit nástroje pro usnadnění práce s grafickými prvky, o kterých se zmíníme později. Pod zdrojovým kódem můžeme ještě najít údaj o čísle řádku, na kterém se aktuálně nachází kurzor.

V pravé části okna se nacházejí prvky pro editaci údajů o jednotlivých zarážkách. Nejprve je nutné vybrat číslo zarážky, kterou chceme editovat – můžeme vytvořit novou nebo

si vybrat ze seznamu nějakou již existující. U zarážky je potřeba doplnit dva základní údaje. Prvním z nich je komentář, který se zobrazuje uživateli při prohlížení skriptu. Ten může kromě textu obsahovat také aktuální hodnotu proměnné. Ta se vloží do textu následujícím způsobem: *$<název_proměnné>$*, tzn. vložením názvu proměnné mezi dva dolarové znaky. Proměnná může být jakéhokoliv snadno zobrazitelného typu, tj. číslo nebo řetězec. Nemusí se jednat přímo o proměnnou typu *double*, *int* či *string*, ale může to být také například vlastnost grafu, ukazatele apod., která se jako číslo či řetězec vyhodnotí. Je také možno sledovat proměnnou typu ukazatel – v tom případě se zobrazí hodnota v něm uložená. Pokud se proměnnou nepodaří vyhodnotit (tzn. neexistuje nebo se nejedná o podporovanou konstrukci), je vyhodnocena jako prázdný řetězec a v komentáři se tedy její hodnota neobjeví. V dolní části oblasti pro editaci zarážek pak můžeme zadat proměnné, jejichž hodnoty chceme uživateli při přerušení skriptu ukázat. Platí zde stejná pravidla jako pro vkládání hodnot proměnných do komentářů. Jediným rozdílem je typ ukazatel. Zde se nezobrazuje jen hodnota přímo v něm, ale také hodnoty v jeho synech a v rodiči. Sledované proměnné můžeme také odebírat – stačí označit její název v seznamu a kliknout na tlačítko *Smazat*. Pokud jsme již doplnili všechny potřebné údaje o zarážce musíme ji ještě uložit stiskem stejnojmenného tlačítka.

Posledním ovládacím prvkem na ploše okna je tlačítko sloužící ke spuštění skriptu, které se nachází v pravé dolní části. Po jeho stisknutí (případně volbě v menu *Script – Spustit* nebo stisknutí F5) dojde ke kontrole syntaxe a případnému spuštění editovaného skriptu. Je-li objevena syntaktická chyba, je to oznámeno a uživatel je přesunut na řádek, kde se chyba nachází. Ke spuštění skriptu nemůže dojít do doby, než budou odstraněny všechny chyby. Pokud se chcete pouze ujistit o tom, že kód je korektní, ale nechcete spouštět skript, můžete použít příkaz v menu *Script – Zkontrolovat* (téhož docílíme stiskem F7). V tom případě vám pouze bude oznámeno, zda je skript v pořádku a kde je případně chyba.

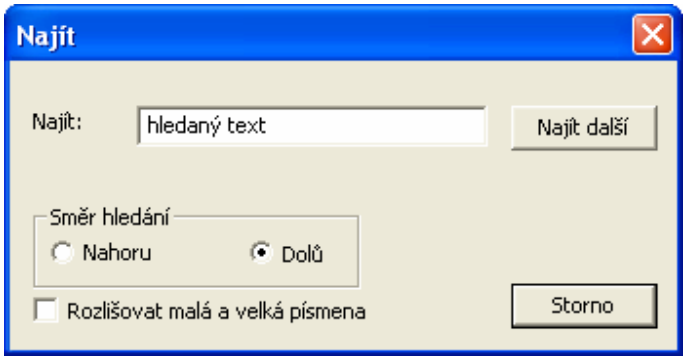
Nyní se podívejme na dostupné operace se souborem skriptu. Chceme-li založit nový soubor se zdrojovým kódem, učiníme tak volbou *Soubor – Nový* nebo klávesovou zkratkou Ctrl+N. Čistý soubor se automaticky otevře po spuštění programu. Pokud máme v úmyslu editovat nějaký již existující algoritmus, otevřeme ho výběrem *Soubor – Otevřít* případně stiskem Ctrl+O. Zobrazí se nám dialog pro nalezení souboru, který chceme otevřít. Soubory s vizualizacemi algoritmů programu AlgoShow mají příponu *.alg*. Po vybrání a potvrzení se

soubor otevře. Nezbytností pro editování skriptů je možnost jejich uložení. Pokud chceme skript uložit pod jménem, které aktuálně má, použijeme volbu *Soubor – Uložit* nebo klávesy Ctrl + S. Nebyl-li algoritmus ještě uložen pod žádným jménem (v hlavičce okna je jako název souboru uvedeno *Beze jména*), budete požádání o zadání názvu, pod kterým se má soubor uložit. Chceme-li skript uložit pod jiným jménem, vybereme z menu příkaz *Soubor – Uložit jako* a zadáme, kam se má algoritmus uložit.

Součástí aplikace jsou i funkce usnadňující editování zdrojového kódu. Jednou z nich je volba *Úpravy - Zpět* (nebo klávesová zkratka Ctrl+Z či příslušný příkaz v kontextovém menu). Ta umožňuje vrátit nechtěné úpravy ve zdrojovém kódu. Počet kroků, o které je možno se vrátit, lze nastavit v dialogu *Možnosti*. Může se pohybovat v rozmezí 1 až 100, výchozí nastavení je 10. Pokud jsme to s vracením přehnali, můžeme stornovanou operaci zopakovat pomocí příkazu *Úpravy – Znovu* (nebo stiskem Ctrl+Y či volbou v kontextové nabídce).

Samozřejmostí je podpora práce se schránkou při editaci kódu. Program nabízí všechy tři základní operace. Kopírovat, vkládat či vyjímat text lze za pomoci příslušné volby v kontextovém menu či hlavní nabídce (v podmenu *Úpravy*). Lze samozřejmě užít všeobecně známých klávesových zkratek, tj. Ctrl+V pro vkládání, Ctrl+X pro vyjímání a Ctrl+C pro kopírování textu. Také je možno text smazat klávesou *Delete* nebo příslušným příkazem v hlavním či kontextovém menu. Stiskem kombinace Ctrl+A (nebo výběrem *Úpravy – Vybrat vše*) pak můžeme označit celý zdrojový kód.

Program AlgoShow také umožňuje vyhledávání a nahrazování (i hromadné) textu. Chceme-li pouze vyhledávat, použijeme volbu *Úpravy – Najít* nebo stiskneme Ctrl+F.



Obr. 5.3.2 – dialog pro vyhledávání v kódu

Zobrazí se nám dialog, který vidíme na obrázku 5.3.2. Zvolíme si, jaký text chceme hledat a nastavíme podmínky vyhledávání. Těmi jsou rozlišování malých a velkých písmen a směr hledání. Směrem hledání se rozumí, na jakou stranu od aktuální polohy kurzoru se má vyhledávat. Po stisku tlačítka *Najít další* je buď oznámeno, že text nelze najít nebo je nalezený text označen. I po případném zavření dialogu můžeme hledat dál, a to pomocí příkazu *Úpravy – Najít další* či stiskem F3. V tom případě dojde k hledání textu, který byl vyhledáván jako poslední, a to i se stejnými vlastnostmi vyhledávání. Pokud je text nalezen, označí se, pokud ne, nestane se nic.

Pokud máme v úmyslu nejen vyhledávat, ale také nahrazovat, vybereme v menu *Úpravy – Nahradit* nebo stiskneme kombinaci kláves Ctrl+H. V tom případě se nám zobrazí následující dialog.

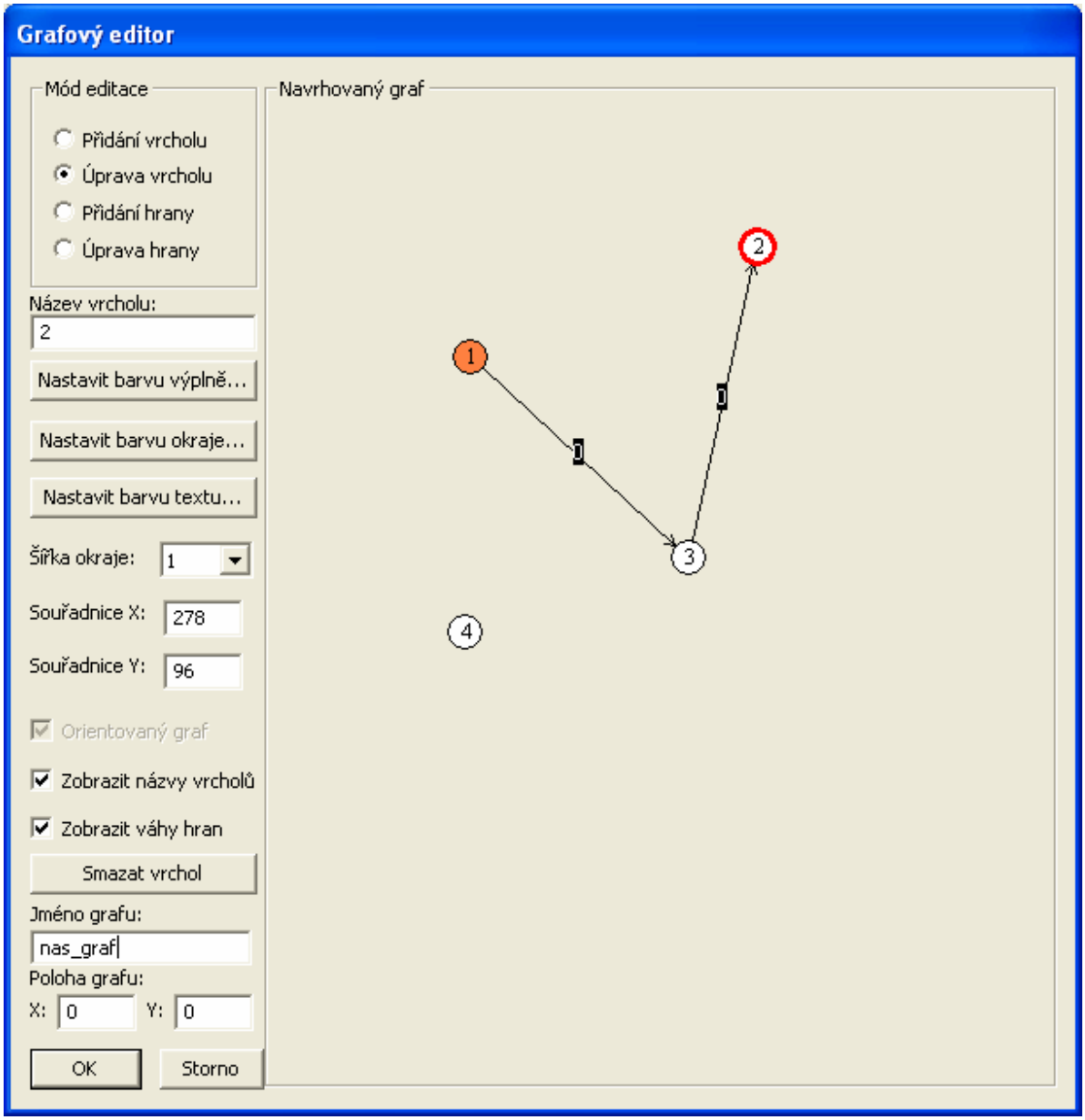
Obr. 5.3.3 – dialog pro nahrazování textu

Nejprve musíme doplnit co a čím chceme nahradit. Také musíme určit směr vyhledávání a zda záleží na velikosti písmen. Máme-li v úmyslu text nejprve jen najít, stiskneme tlačítko *Najít další*. V případě úspěšného nalezení se text označí a my ho můžeme nahradit kliknutím na tlačítko Nahradit. Pokud nechceme ručně potvrzovat nahrazení všech výskytů, můžeme použít volbu *Nahradit vše*.

Poslední skupinou podstatných funkcí programu je soubor nástrojů usnadňující psaní kódu. Těchto pomocníků je celkem pět a můžeme je spustit z hlavního menu (podnabídka *Nástroje*), kontextové nabídky či za pomoci příslušné klávesové zkratky.

Nejjednodušším nástrojem je vložení barvy (Ctrl+Alt+C). V dialogu stačí navolit barvu podle vašich představ a následně dojde ke vložení jejího číselného kódu na místo, kde se nacházel kurzor ve chvíli, kdy byl nástroj spuštěn.

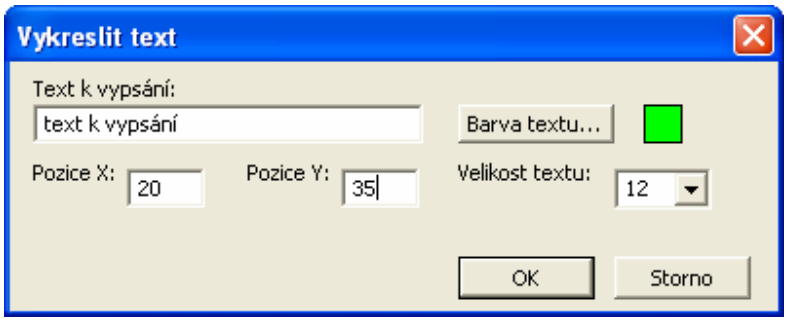
Asi nejsložitějším nástrojem je ten, který slouží ke vložení kódu inicializujícího graf. Kromě volby z menu můžeme tohoto pomocníka spustit také klávesovou zkratkou Ctrl+Alt+G. Ukáže se nám následující dialog.



Obr. 5.3.4 – dialog pro editaci grafu pro generování kódu

Jedná se o mírně pozměněný dialog, který se zobrazuje při načítání grafu během prohlížení algoritmu a je popsán v minulé podkapitole. I zde je tedy možno přidávat vrcholy a hrany a také je následně upravovat (postup je stejný a proto ho zde nebudu podrobně popisovat). Jedinou věcí, která je zde navíc, je nutnost zadání jména grafu. Jménem grafu se rozumí název proměnné, která bude graf reprezentovat. Tento název tedy musí splňovat stejná kritéria jako jméno proměnné (viz popis skriptovacího jazyka). Po odsouhlasení podoby grafu dojde k vygenerování tří částí kódu. Na úplný začátek se přidá deklarace proměnné, která reprezentuje graf a hlavičky inicializační funkce grafu (její název je ve tvaru *Init_Graph_<název_grafu>*). Na konec skriptu se pak vygeneruje samotné tělo inicializační funkce grafu, kde dochází k přidávání vrcholů a hran a nastavování jejich vlastností. Na závěr se ještě přidá zavolání inicializační funkce, a to na místo, odkud byl nástroj spuštěn (tj. tam, kde se nacházel kurzor).

Další usnadnění práce umožňuje nástroj pro navrhování nápisů, které se mají během vizualizace objevovat. Ke spuštění tohoto pomocníka můžete kromě hlavního a kontextového menu použít také klávesovou zkratku Ctrl+Alt+T. Poté se vám zobrazí tento dialog:



Obr. 5.3.5 – nástroj pro vkládání nápisů

Nejdůležitější věcí je doplnění textu, který se má uživateli objevit. Pomocí tohoto nástroje je možno vypisovat pouze konstantní řetězce. Pro možnost použití aktuálních hodnot proměnných si prostudujte popis funkce *TextOut(...)* v následující podkapitole a upravte kód ručně. U vykreslovaného nápisu můžeme také natavit pozici, ve které se má nápis objevit (počítá se od levého horního okraje), jeho velikost a barvu. Po odsouhlasení údajů dojde k vygenerování volání funkce, která vykreslí text. Pokud chceme nápis smazat, musíme

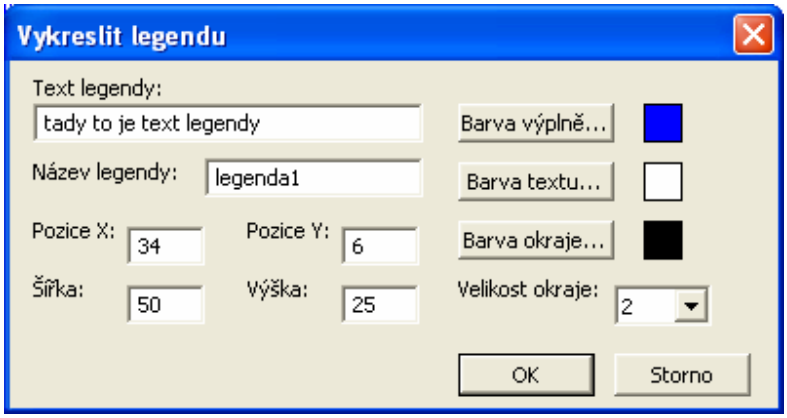
použít funkci *DeleteTexts(...)* – viz popis skriptovacího jazyka. Ukázku toho, jak vypadá výsledný text, můžete najít na obrázku 5.3.8.

Obdobným pomocníkem je nástroj pro vytváření šipek, které se zobrazují uživateli při prohlížení algoritmu. K jeho zobrazení je možno kromě menu použít kombinaci kláves Ctrl+Alt+S.

Obr. 5.3.6 – nástroj pro navrhování šipek

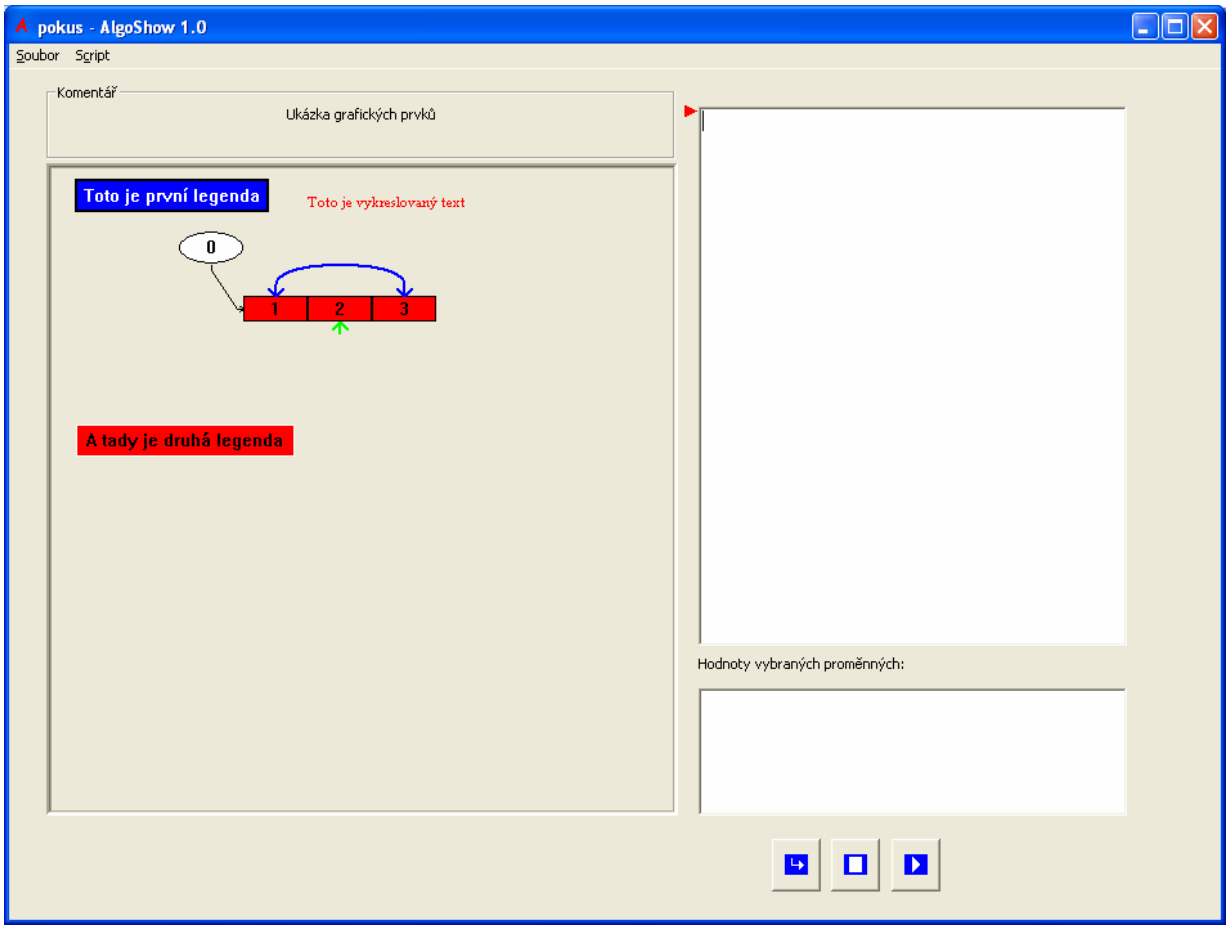
Nejpodstatnějším údajem, který musíme určit je, odkud a kam šipka povede. Jejím krajním bodem může být buňka pole nebo ukazatel. Samotný nástroj nekontroluje, zda vámi zadané hodnoty opravdu vyjadřují proměnné požadovaného typu, a na případnou chybu tak budete upozorněni až při kontrole syntaxe. Výchozí a koncový bod mohou být i shodné. U obou konců šipky lze nastavit, ze které strany budou k objektu, na který jsou navázány, připojeny (na výběr jsou možnosti shora, zdola, zleva, zprava). Je také nutno určit, zda se bude jednat o šipku obyčejnou (→) nebo oboustrannou (↔). Jak vypadá taková šipka ve výsledku si můžete prohlédnout na obrázku 5.3.8. Dále je možno určit, jakou barvu bude šipka mít a také její šířku (v rozmezí 1–4). Po kliknutí na tlačítko OK dojde k vygenerování funkce, která šipku vykresluje. K odstranění šipky je nutno použít funkci *DeleteArrows()*, jejíž podrobnější popis najdete v následující podkapitole.

Poslední prozatím nepopsaný nástroj slouží ke generování kódu, který zobrazuje legendu (ale může být použit například i pro vypsání jakéhokoliv textu). K tomuto pomocníkovi přistoupíme buď z hlavní či kontextové nabídky nebo pomocí klávesové zkratky Ctrl+Alt+L.



Obr. 5.3.7 – nástroj pro vytvoření legendy

Kromě zadání samotného textu legendy je nutné také doplnění jejího názvu (a tedy názvu proměnné, přes kterou se k legendě bude přistupovat). Legenda je implementována jako jednopoložkové pole typu *string*. Pokud tedy chceme text legendy za běhu programu měnit, přistupujeme k němu konstrukcí *<název legendy>[0]*. V dialogu můžeme také nastavit pozici, na které se legenda objeví (počítáno v pixelech od levého horního rohu), její šířku a výšku. Dále barvu textu, výplně, okraje a šířku okraje legendy (rozmezí 1–4, pokud okraj nechceme, dáme mu stejnou barvu, jako má výplň). Po nastavení všech potřebných údajů se vygeneruje kód, který má tři části. Na úplný začátek skriptu se přidá deklarace proměnné reprezentující legendu a hlavička inicializační funkce legendy (má název *Init_Legend_<název legendy>*). Samotné tělo této funkce se pak přidá na konec zdrojového kódu. Na místo, kde se nacházel kurzor před spuštěním nástroje, se vygeneruje příkaz volající tuto inicializační funkci. Jak může taková legenda vypadat v praxi, můžeme vidět na obrázku 5.3.8.



Obr. 5.3.8 – ukázka některých grafických prvků

5.4. Popis skriptovacího jazyka

Aby vůbec bylo možné navrhovat vlastní vizualizace algoritmů, je nejprve nutné naučit se alespoň základy jazyka, ve kterém se skripty zapisují. Tento jazyk vychází z principů jazyka C, ale přináší i některá rozšíření (zejména pro práci s grafikou). Na několika následujících stranách najdete kompletní popis, jak v tomto jazyce vytvářet skripty, a to včetně malých příkladů. V první části si přiblížíme základy jeho fungování včetně podporovaných datových typů, podmíněných příkazů, cyklů a podpory vstupu/výstupu. Tuto část mohou lidé ovládající programování v C/C++ projít jen v rychlosti. Druhá půlka ale patří popisu práce s grafikou, která je pro tento jazyk specifická a je nezbytné ji dobře pochopit.

Struktura skriptu

Skript je rozdělen do samostatných bloků, které nazýváme funkcemi. Tyto bloky kódu se mohou navzájem spouštět a předávat si informace. Jedinou částí skriptu, která se může nacházet mimo tyto bloky, je deklarace globálních proměnných, tedy proměnných, s jejichž daty mohou pracovat všechny funkce. Skript musí obsahovat funkci s názvem *main* (bez parametrů, návratový typ je *void*). Tato funkce je zavolána jako první po spuštění skriptu a její ukončení znamená i ukončení skriptu jako celku. Speciální kapitolou jsou komentáře, které se nám budou hodit u některých příkladů, a proto jejich zápis vysvětlím již teď. Všechny komentáře v našem jazyku jsou jednořádkové a začínají stejně jako v C dvojznakem ***„//“*** – například ***„tady není komentář // a tady už je komentář“***.

Základní datové typy

- ***int*** – celé číslo
- ***double*** – desetinné číslo
- ***string*** – řetězec
- ***Cell**** – ukazatel na buňku binárního stromu či spojového seznamu
- ***void –*** prázdný datový typ (používá se pouze jako návratový typ funkcí, které nic nevrací)
- ***pole*** – soubor několika položek stejného typu (mohou být *int*, *double* či *string*)

Deklarace proměnných

Deklarace proměnných se rozděluje na dva typy – globální a lokální. Globální proměnné se deklarují mimo funkce a k jejich datům je možno přistoupit ze všech funkcí. Lokální se deklarují přímo v těle funkce a jsou přístupné pouze uvnitř ní. Samotná deklarace proměnné má následující tvar: ***<typ proměnné> <název>([<velikost pole>])*** *(= <počáteční hodnota>, <další proměnné>)****;***. Typem proměnné rozumíme jakýkoliv datový typ s výjimkou *void*. Název proměnné se může skládat z písmen (bez diakritiky), číslic a podtržítka. Prvním znakem ale nesmí být číslice. Název proměnné se nesmí shodovat

s žádným z klíčových slov jazyka. Skriptovací jazyk je stejně jako C „case sensitive“, tzn. rozlišuje malá a velká písmena v názvu identifikátorů. Chceme-li deklarovat pole (pouze u typů *int, double, string*), přidáme za název proměnné jeho požadovanou velikost v hranatých závorkách. Při deklaraci můžeme do proměnné (nejedná-li se o pole) dosadit její hodnotu. Pokud chceme deklarovat více proměnných stejného typu, nemusíme mít každou na zvláštním řádku, ale můžeme je napsat v jednom řádku a oddělíme je od sebe čárkou. Na závěr řádku s deklarací musíme (stejně jako u jakéhokoliv jiného příkazu) přidat středník. Proměnná musí být deklarována před prvním použitím v programu. Pokud není inicializovaná, je jejím obsahem výchozí hodnota – pro číselné typy je to nula, pro *string* prázdný řetězec a nulový ukazatel pro typ *Cell**. Zde jsou některé příklady deklarací proměnných:

příklad 5.4.1 – deklarace proměnných:

```
int cislo; // deklarace proměnné celočíselného typu
int a = 1, b, c = 3 + a; // deklarace více proměnných v seznamu a přiřazení hodnoty některým z nich
double pi = 3.14, PI = 3.1416; // pi a PI jsou dvě odlišné proměnné
string str = "konstantní řetězec"; // deklarace proměnné a dosazení její hodnoty
string pole[4]; // deklarace čtyř-prvkového pole řetězců
Cell* ukazatel; // deklarace proměnné typu ukazatel
```

Přiřazovací příkaz, výrazy a možné konverze

Pokud chceme do nějaké proměnné dosadit konkrétní hodnotu, použijeme k tomu operátor rovná se („=“). Celý příkaz je tedy tvaru ***„promenna = hodnota;“***. Dosazovat lze jen hodnotu, jejíž typ je shodný s typem proměnné nebo je alespoň možná konverze.

V našem jazyce existují celkem čtyři druhy konverze. První dva jsou převody mezi číselnými typy. U převodu *double* → *int* dochází k odstranění desetinné části čísla (tzn. 4.1 → 4, 4.9 → 4) a tím se ztrácí přesnost. V opačném případě, tedy *int* → *double*, ke ztrátě přesnosti nedochází, pouze se za číslo přidá desetinná tečka (4 → 4.0). Tyto dvě předcházející konverze lze aplikovat i na celý výraz a nejen na jednotlivé proměnné (tzn. lze výraz vyhodnocený jako *double* dosadit do proměnné typu *int*). U dalších přípustných konverzí (*double* → *string a int* → *string*) již není možné konvertovat celé výrazy, ale pouze

samostatné proměnné či konstanty. Je tedy povolen zápis ***„string s = a;“***, ale není možno napsat ***„string s = a – b;“***, kde *a* a *b* jsou proměnné číselného typu.

Podívejme se nyní na přípustné operace s číselnými a řetězcovými výrazy. U číselných výrazů máme k dispozici následujících pět operátorů:

- vysoká priorita:
  - ***** - násobení dvou čísel
  - / - dělení dvou čísel
  - **%** - zbytek po celočíselném dělení (modulo)
- nízká priorita:
  - **+** - sečtení dvou čísel
  - **-** - odečtení dvou čísel

Pokud si chceme určit vlastní prioritu operací, použijeme k tomu kulaté závorky (viz příklad 5.4.2). U řetězců je dostupný pouze jeden operátor, a to „+“ sloužící k jejich sřetězování. Tímto způsobem lze pospojovat libovolný počet konstantních řetězců a proměnných typu *string*, *double* nebo *int*. V případě číselných typů musí jít ale pouze o proměnné, nikoliv o konstantní čísla.

příklad 5.4.2 – přiřazení, výrazy, konverze:

```
double deleni1 = 5 / 2; // výsledkem je 2.5
int deleni2 = 5 / 2; // výsledkem je 2 (byla odstraněna desetinná část)
int i = 5 % 2 + 2; // výsledek je 3 ( protože (5 mod 2 = 1) + 2 = 3 )
int j = 5 % (2 + 2); // výsledek je 1 ( 5 mod (2+2 = 4) = 1 )
string s = "výsledek je " + deleni1; // výsledek je: "výsledek je 2.5"
int a = 2, b = 3;
string s1 = a + b; // výsledkem je řetězec "23" ( nikoliv "5" !!! )
~~string s1 = 2 + 3;~~ // nepovolený zápis – nelze zřetězovat čísla
~~string error = j * i;~~ // nepovolená konstrukce - dosazení číselného výrazu do proměnné typu string
```

Zápis hlavičky a těla funkce a její volání

Hlavička funkce udává její jméno, návratový typ, typy a názvy parametrů. Zápis je následující: **<*návratový typ*> <*název funkce*> (**<*seznam parametrů*>**)**. Korektním zápisem hlavičky je například *„**double Vydel(int a, int b)“*** – tato funkce s názvem *Vydel* požaduje dva celočíselné parametry a vrací hodnotu typu *double*. Funkce také nemusí mít žádné parametry – tak tomu je například u funkce *main*. Její hlavička vypadá následovně: ***„void main()“***. Máme-li v úmyslu zavolat funkci ještě před definicí jejího těla, musíme předem alespoň nadefinovat tvar její hlavičky. V tom případě napíšeme hlavičku a za ní středník. Pokud chceme definovat zrovna i tělo funkce, následuje za hlavičkou blok příkazů ohraničený složenými závorkami.

V těle funkce se mohou nacházet příkazy jakéhokoliv typu (s výjimkou definice funkce). Ty si postupně popíšeme v následujících částech textu, nyní se však podívejme na jeden příkaz, který úzce souvisí s funkcemi. Je jím ***return***. Ten slouží pro ukončení běhu funkce a případný návrat nějaké hodnoty. Pokud má ukončovaná funkce návratový typ *void*, není nutné příkaz *return* použít. V případě jeho užití však za ním nesmí následovat žádný výraz, protože funkce s návratovým typem *void* nemůže nic vracet. U ostatních možných návratových typů (*int*, *double*, *string*, *Cell**) je pak nutno *return* použít vždy. Je nutno navíc zadat hodnotu, jakou má funkce vracet. Tato hodnota se napíše za klíčové slovo *return* – může se jednat o konstantu, proměnnou či celý výraz odpovídající návratovému typu. Ukázku správného zápisu najdete v příkladu 5.4.3.

Pokud chceme funkci zavolat, můžeme tak učinit pomocí samostatného příkazu. Jeho tvar vypadá následovně: ***„JmenoFunkce(<hodnota 1. parametru>,<hodnota 2. parametru>, ...);“***. Tento způsob volání lze použít pro funkce se všemi návratovými typy. Zajistíme tím pouze provedení těla funkce, ale nemáme možnost si uložit výsledek jejího volání. Pokud si chceme návratovou hodnotu pamatovat, musíme použít volání funkce uvnitř nějakého výrazu. Funkce se v tom případě chová stejně jako proměnná typu odpovídajícího návratového typu funkce. Tento způsob volání nelze použít u funkcí s návratovým typem *void* nebo tam, kde neexistuje přípustná konverze. Funkce se mohou volat i rekurzivně – to znamená, že funkce může zavolat sama sebe. Pro pochopení správného zápisu volání funkce snad pomůže následující příklad.

příklad 5.4.3 – práce s funkcemi:

```
double PI = 3.1416; // globální proměnná
double ObsahKruhu(int polomer); // definice hlavičky funkce

void main() //definice těla hlavní funkce programu
{
	double obsah = 0; // lokální proměnná, dosazení hodnoty 0 není potřebné(je výchozí)
	int polomer = 2; // lokální proměnná reprezentující poloměr kruhu
	obsah = ObsahKruhu(polomer); // dosazení výsledku volání funkce do proměnné
	return; // návrat z funkce - v tomto případě není příkaz nutný
}

double ObsahKruhu(int polomer) //definice těla funkce
{
	return PI * polomer * polomer; // návrat hodnoty
}
```

Vstup a výstup

Pro dosažení co největší interaktivity s uživatelem.je nutné s ním komunikovat. Kromě grafického výstupu informací (o tom bude řeč až za chvíli) existuje možnost vypsání informace za pomoci speciálního dialogu. Samotné zobrazení dialogu a přerušení běhu skriptu do doby jeho zavření zajišťuje interpret jazyka automaticky. Nám stačí pouze zadat text, jaký se má uživateli ukázat. Zobrazení dialogu je velice jednoduché – stačí zavolat vestavěnou funkci *write* s parametrem typu *string*, který udává, jaký text se má vypsat. Ukázku takového výstupu najdete v příkladu 5.4.4.

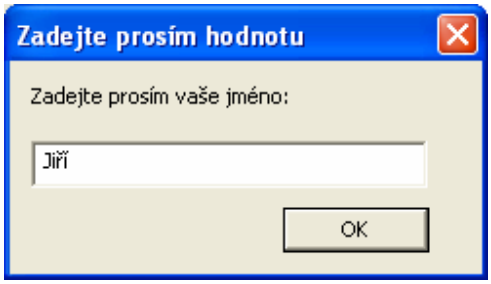
Obdobně jednoduché je i vyžádání nějakého údaje od uživatele. K tomu se využívá funkce *read*. Ta vyžaduje předání dvou parametrů. Prvním je proměnná, do které se má výsledek načíst a která může být typu *string*, *int* nebo *double*. Druhým parametrem je výraz typu *string*, který udává nápis, jenž se má zobrazit uživateli (typicky ve tvaru „Zadejte hodnotu...“). V okamžiku zavolání této funkce se otevře dialog, ve kterém je uživatel vyzván vámi zvoleným textem k zadání hodnoty proměnné. Pokud je požadována hodnota číselného typu a uživatel zadá neplatný výraz (například řetězec), je do proměnné dosazena nulová hodnota. Existuje také varianta funkce *read* pro zadání podoby grafu. Ta si žádá pouze jeden

parametr a její podrobný popis najdete v části věnované grafickým vlastnostem skriptovacího jazyka. Ukázku použití funkce *read* najdete v následujícím příkladu.

příklad 5.4.4

```
string jmeno; // proměnná pro uložení jména

void main() //definice těla hlavní funkce programu
{
        read (jmeno, "Zadejte prosím vaše jméno:"); // vyžádání hodnoty proměnné (Obr. 5.4.5)
        write ("Dobrý den, " + jmeno + " !!!"); // zobrazení textu v dialogu (Obr. 5.4.6)
}
```



Obr. 5.4.5 – zadání hodnoty proměnné uživatelem použitím funkce read

Obr. 5.4.6 – zobrazení textu uživateli pomocí funkce write

Použití ukazatelů

Skriptovací jazyk umožňuje také práci s ukazateli. Nenajdete zde však obdobu ukazatelů z jiných známých jazyků, ve kterých lze definovat ukazatel na libovolný typ, ale pouze ukazatel na specielně definovanou buňku. Ta je uzpůsobena k pohodlné práci se spojovými seznamy a binárními stromy. Před samotným použitím buňky musíme udělat dva základní kroky: nejprve deklarujeme proměnnou typu *Cell** a pak pro ni musíme alokovat

paměť. To uděláme pomocí klíčového slova *new*. Za ním musí následovat jeden ze dvou možných typů buněk – *CellInt* a *CellDouble*. Ty se, jak již název napovídá, liší v typu hodnoty, kterou buňka uchovává. Chceme-li tedy deklarovat a alokovat ukazatel na buňku obsahující celé číslo, použijeme příkaz ***„Cell* ptr = new CellInt;“***. Má-li obsahovat desetinné číslo, zvolíme variantu ***„Cell* ptr = new CellDouble;“***. Narozdíl od většiny běžných jazyků není potřeba ukazatele dealokovat – to zajistí na konci skriptu interpret automaticky. Dokud není ukazatel alokován nebo mu není přiřazena hodnota jiného ukazatele, je jeho hodnota nulová.

K hodnotám uloženým uvnitř buňky přistupujeme pomocí šipkové notace (tedy dvojznaku „->“). Pozor, v případě přístupu na nulový ukazatel bude ohlášena chyba a skript se ukončí. V tuto chvíli se zatím nebudeme zabývat nastavováním grafických vlastností, ale vysvětlíme si čtyři položky, které by se nám budou hodit, i když nebudeme pracovat s grafikou. Základem je hodnota uložená v buňce. K ní přistoupíme konstrukcí *ukazatel->hodnota*. Hodnota je typu *int* nebo *double* (záleží, jakým způsobem jsme ukazatel alokovali). Lze z ní samozřejmě jak číst, tak do ní i zapisovat. Dalšími třemi potřebnými vlastnostmi buňky jsou ukazatele na její tři sousedy – tedy rodiče, levého a pravého syna (*ukazatel->parent, ukazatel->left, ukazatel->right*). To je velmi užitečné pro práci s binárními stromy či spojovými seznamy (tam se využijí pouze jeden či dva ukazatele). Vynulovat ukazatel můžeme dosazením číselné hodnoty 0 nebo konstanty NULL.

příklad 5.4.7 – práce s ukazateli:

```
Cell* ptr1 = new CellInt; // deklarace ukazatele a alokování buňky s celým číslem
void main() // definice těla hlavní funkce programu
{
        Cell* ptr2, ptr3; // deklarace dalších ukazatelů
        ptr2 = new CellDouble; // alokace buňky obsahující desetinné číslo
        ptr1->left = ptr2; // levým synem ptr1 se stává ptr2
        ptr2->parent = ptr1; // rodičem ptr2 se stává ptr1
        ptr1->hodnota = 4; // hodnota v ptr1 je 4
        ptr2->hodnota = ptr1->hodnota + 1; // hodnota v ptr2 je 5
        ptr3 = NULL; // vynulování ukazatele (není potřeba, defaultně je nulový)
        ptr3->hodnota = 1; // nastane běhová chyba - přístup na nulový ukazatel
}
```

Podmíněný příkaz a zápis pravdivostních výrazů

Nepostradatelnou součástí skriptovacího jazyka je příkaz umožňující větvení programu na základě vyhodnocení platnosti nějaké podmínky. Zápis je stejný jako v programovacím jazyce C:

if (<podmínka>)
    příkaz1;
else
    příkaz2;

Příkazem rozumíme buď jeden příkaz (ukončený středníkem) nebo celý blok příkazů uzavřený ve složených závorkách. Větev *else* není povinná a vztahuje se vždy k nejbližšímu *if*. Pokud chceme, aby se vztahovala k jinému *if*, musíme to zajistit pomocí uzavření bloku do složených závorek. Větev *if* se provede, pokud je podmínka splněna. V opačném případě se interpretuje větev *else*.

Nyní se podívejme, jakým způsobem můžeme zapsat podmínku. Nejprve se zaměřme na jednoduché podmínky. Těmi rozumíme test na nulovou hodnotu nebo porovnání dvou hodnot. Pokud je podmínkou pouze proměnná typu *int*, *double* či *Cell**, je vyhodnocena jako pravdivá, je-li hodnota proměnné nenulová. V opačném případě je výraz vyhodnocen jako nepravda. Přidáním znaku „!“ před jednoduchou podmínku dosáhneme její negace. Teď se podívejme, jaké možnosti porovnání dvou proměnných se nám nabízí u jednotlivých typů:

int – int: „==”, “!=”, “<”, “>”, “<=”, “>=”
int – double: „==”, “!=”, “<”, “>”, “<=”, “>=”
double – double: „==”, “!=”, “<”, “>”, “<=”, “>=”
string – string: „==”, “!=”
Cell* – Cell*: „==”, “!=”

Jednotlivé jednoduché podmínky pak můžeme spojovat pomocí logických spojek „&&“ (AND) a „||“ (OR). V následujících dvou tabulkách můžete vidět, jakým způsobem tyto spojky vyhodnotí výraz.

&&	PRAVDA	NEPRAVDA
PRAVDA	PRAVDA	NEPRAVDA
NEPRAVDA	NEPRAVDA	NEPRAVDA

||	PRAVDA	NEPRAVDA
PRAVDA	PRAVDA	PRAVDA
NEPRAVDA	PRAVDA	NEPRAVDA

tabulka 5.4.8 – vyhodnocování spojek AND a OR

Prioritu vyhodnocování spojek && a || můžeme určit pomocí kulatých závorek. Následující ukázka kódu vysvětluje základy práce s podmíněnými výrazy a uvádí i některé příklady zápisu podmínek.

příklad 5.4.9 – podmínky a podmíněné výrazy:

```
Cell* nulovy_ukazatel = NULL;
int a = 2, b = a + 3;
string retezec = "ano";

void main()
{
	if (nulovy_ukazatel) // podmínka je nepravdivá - ukazatel je nulový
		return; // tento příkaz se neprovede
	else // tato větev se provede, protože podmínka byla nepravdivá
	{
		a = a+1;
		b = b+1;
	}

	if (retezec == "ano" || 6 < 5) // podmínka je pravdivá (řetězce jsou shodné)
		write ("pravda");

	if ((1 == 1 || 1 != 1) && a && !b)
	//nepravdivá podmínka (kdyby b nebylo negované, bylo by vše v pořádku)
		return; // toto se neprovede
}
```

Cykly

Cykly slouží k opakování bloku kódu po dobu platnosti určité podmínky. V našem skriptovacím jazyce najdeme celkem tři druhy cyklů. Prvním z nich je *while* cyklus s podmínkou na začátku, který provádí své tělo, dokud platí řídící podmínka. Tělem cyklu může být jeden příkaz ukončený středníkem nebo celý blok příkazů uzavřený ve složených závorkách.

while (<podmínka>)
    příkaz;

Obdobou je *do – while* cyklus. Ten také provádí své tělo, dokud platí podmínka. Ta se ale vyhodnocuje až po interpretaci těla, tudíž cyklus proběhne alespoň jednou, i když podmínka je nepravdivá.

do
    příkaz;
while (<podmínka>);

Posledním druhem je takzvaný *for-cyklus*. Ten se typicky používá, pokud chceme tělo provést v určitém počtu opakování. Ve své hlavičce požaduje celkem tři údaje. Prvním z nich je inicializace, typicky nastavení výchozí hodnoty řídící proměnné. Druhým údajem (odděleným středníkem) je podmínka, za jejíž platnosti se v cyklu pokračuje. Na konec je potřeba zadat inkrementační příkaz, tedy příkaz,který se provede vždy na konci těla cyklu. Typicky se jedná o zvýšení hodnoty řídící proměnné o jedna. Tělem cyklu opět může být jak samostatný příkaz, tak i blok příkazů.

for (inicializace; podmínka; inkrementace)
    příkaz;

Práce s grafikou ve skriptovacím jazyce

Pole

Jak již bylo řečeno v předcházejících odstavcích, je možno deklarovat pole typů *int*, *double* a *string*. Tato pole neuchovávají jen samotná data, ale také údaje o tom, zda a jakým způsobem se mají zobrazovat uživateli. Grafické vlastnosti můžeme nastavovat poli jako celku i jednotlivým buňkám. K vlastnostem se přistupuje pomocí tečkové notace. Podívejme se na přehled všech vlastností:

vlastnosti celého pole:

visible – udává, zda má být pole vykreslováno (0 = nevykreslovat, nenulová hodnota znamená vykreslovat) – výchozí hodnota je nula

BorderColor – uchovává kód barvy, kterou se barví okraj všech buněk (výchozí je černá)

FillColor – barva, kterou má výplň buněk (výchozí je bílá barva)

TextColor – barva textu (nebo čísel) v buňkách (výchozí je černá)

BorderWidth – šířka okraje buněk (v pixelech, výchozí je 1)

CellWidth – šířka jednotlivých buněk (v pixelech, výchozí je 50)

Height – výška pole (v pixelech, výchozí je 25)

PosX – horizontální poloha pole (počítáno od levého okraje)

PosY – vertikální poloha pole (počítáno od vrchu)

vlastnosti jednotlivých buněk:

BorderColor – uchovává kód barvy, kterou se barví okraj buňky (výchozí je černá)

FillColor – barva, kterou má výplň buňky (výchozí je bílá barva)

TextColor – barva textu (nebo čísel) v buňce (výchozí je černá)

BorderWidth – šířka okraje buňky (v pixelech, výchozí je 1)

Width – šířka buňky (v pixelech, výchozí je 50)

příklad 5.4.10 – nastavení vlastností polí:

```
int pole[3]; // deklarace troj-prvkového pole celých čísel
void main()
{
        pole.visible = 1; // zviditelnění pole
        pole.CellWidth = 70; // nastavení šířky všech buněk
        pole[1].Width = 40; // prostřední bude užší
        pole.FillColor = 255; // červená výplň
        pole[0].TextColor = 16777215; // bílá barva textu v 1. buňce
}
```

a toto je výsledek:

Ukazatele

Stejně jako u polí, je i u ukazatelů možno nastavit vlastnosti, kterými se řídí jejich vykreslování. Ke grafickým vlastnostem ukazatelů přistupujeme pomocí operátoru „->“. Nikdy nesmíme přistupovat na nulový ukazatel – v tom případě by nastala běhová chyba a skript by byl ukončen. Zde je přehled vlastností, které je možno nastavovat:

visible – udává, zda má být ukazatel vykreslován (0 = nevykreslovat, nenulová hodnota znamená vykreslovat) – výchozí hodnota je nenulová

BorderColor – uchovává kód barvy, kterou se barví okraj buňky (výchozí je černá)

FillColor – barva, kterou má výplň buňky (výchozí je bílá barva)

TextColor – barva textu (nebo čísel) v buňce (výchozí je černá)

Ellipse – udává, zda má být buňka kulatá (elipsová) (0 = hranatá, nenula = kulatá), výchozí je hranatý tvar

Width – šířka buňky v pixelech (výchozí hodnota je 50)

Height – výška buňky v pixelech (výchozí hodnota je 25)

Top – vertikální poloha buňky (počítáno od vrchu, výchozí je 0)

Left – horizontální poloha buňky (počítáno odleva, výchozí je 0)

příklad 5.4.11 – nastavení vlastností ukazatelů:

```
Cell* ptr = new CellInt; // alokování buňky
void main ()
{
        ptr->Height = 35; // nastavení šířky buňky
        ptr->FillColor = 16711680; // barva výplně
        ptr->BorderWidth = 3; // nastavení šířky okraje
        ptr->Ellipse = 1; // nastavení tvaru na elipsu
        ptr->hodnota = 99; // nastavení hodnoty uložené v buňce
}
```

a takhle vypadá výsledek:

Vykreslování textů

Na plochu, kde se zobrazuje vizualizace algoritmu, můžeme také vypisovat texty. K tomu nám poslouží zavolání funkce ***TextOut (int X, int Y, int velikost, int barva, string text)***, kde jednotlivé parametry mají následující význam:

X – odsazení textu od levého okraje (v pixelech)

Y – odsazení textu odshora (v pixelech)

velikost – velikost textu

barva – kód barvy, jakou se má text vykreslit

text – samotný text, který má být vypsán

Pokud chceme nápis smazat, zavoláme funkci ***DeleteTexts()***. Tou odstraníme veškeré nápisy. Není možné odstraňovat texty jednotlivě, protože nejsou nijak pojmenovávány

Kreslení šipek

Skriptovací jazyk nám umožňuje spojovat logicky související prvky za pomoci šipek. Můžeme jimi propojovat prvky dvou typů: buňky pole a buňky, na které míří nějaký ukazatel. Šipku nakreslíme zavoláním funkce ***Arrow(odkud, orientace_odkud, kam, orientace_kam, int sirka, int barva, int oblouk, int oboustranna)***. Význam jednotlivých parametrů je následující:

odkud – proměnná typu buňka pole nebo ukazatel, od které povede šipka

orientace_odkud – udává, z jaké strany bude šipka „přilepena“ k výchozímu prvku, může nabývat čtyř hodnot: L = zleva, R = zprava, U = shora, D = zdola

kam – proměnná typu buňka pole nebo ukazatel, do které povede šipka

orientace_kam – udává, z jaké strany bude šipka „přilepena“ k cílovému prvku, může nabývat čtyř hodnot: L = zleva, R = zprava, U = shora, D = zdola

sirka – udává tloušťku šipky v pixelech

barva – kód barvy, jakou má šipka mít

oblouk – umožňuje určit zaoblení šipky

oboustranna – udává, zda má být šipka oboustranná (↔; nenulová hodnota) nebo jednostranná (→; nulová hodnota)

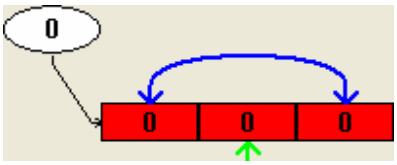
Pro pohodlnější vytváření šipek doporučuji používat nástroj popsaný v podkapitole 5.3. Chceme-li šipky smazat (opět lze odstranit pouze všechny najednou), zavoláme funkci ***DeleteArrows()***.

příklad 5.4.12 – kreslení šipek:

```
void main()
{
        int pole[3]; // deklarace pole o 3 prvcích
        Cell* ptr = new CellInt; // ukazatel na buňku
        // zde by následovalo nastavení grafických vlastností pole a ukazatele
        // šipka od ukazatele k 1. položce pole (jednostranná, černá, bez oblouku) :
        Arrow(ptr,D,pole[0],L,1,0,0,0);
        // šipka mající stejný výchozí a cílový bod (šířka 2, barva zelená) :
        Arrow(pole[1],D,pole[1],D,2,65280,0,0);
```

```
    // zaoblená šipka mezi dvěma buňkami pole, oboustranná, modrá barva :
    Arrow(pole[0],U,pole[2],U,2,16711680,-21,1);
}
```

A toto je výsledek:



Práce s grafy

Relativně rozsáhlou oblastí skriptovacího jazyka je práce s grafy. K tomuto účelu existuje celkem pět datových typů. Nejprve se podívejme na tři základní:

Graph – reprezentuje graf, tedy množinu vrcholů a hran

Vertex – uchovává grafické vlastnosti vrcholu grafu

Edge – uchovává grafické vlastnosti hrany a její váhu

Typy Vertex a Edge slouží pouze jako nástroj pro nastavení vlastností hran a vrcholů před jejich vložením do grafu. Nejsou tedy viditelné, ale i tak je potřeba nastavit jejich grafické vlastnosti, protože v okamžiku jejich vložení do grafu se viditelnými stávají. K vlastnostem se přistupuje pomocí tečkové notace. Zde je jejich přehled:

vlastnosti typu Vertex (vrchol):

BorderColor – uchovává kód barvy, kterou se barví okraj vrcholu (výchozí je černá)

FillColor – barva, kterou má výplň vrcholu (výchozí je bílá barva)

TextColor – barva textu, kterým se vypisuje název vrcholu (výchozí je černá)

BorderWidth – šířka okraje vrcholu (v pixelech, výchozí je 1)

Width– průměr vrcholu (v pixelech, výchozí je 20)

vzdalenost – pomocná proměnná pro uchování vzdálenosti při algoritmech na hledání nejkratší cesty

Left – horizontální poloha vrcholu (počítáno od polohy grafu)

Top – vertikální poloha vrcholu (počítáno od polohy grafu)

vlastnosti typu Edge (hrana):

color – barva hrany (výchozí je černá)

width – šířka hrany (v pixelech, výchozí je 1)

vaha – váha hrany (pro algoritmy, které ji využívají, výchozí je 0)

Nyní si ukážeme, jakým způsobem přidáme vrchol či hranu do grafu. K tomu nám slouží dvě funkce, ke kterým přistupujeme pomocí tečkové notace na proměnné typu *Graph*. K přidání vrcholu slouží ***AddVertex(Vertex vrchol, string nazev_vrchol,)***, kde druhým parametrem je řetězec udávající, pod jakým názvem se má vrchol přidat. Grafické vlastnosti vrcholu pak udává první parametr. Pokud přidáváme vrchol, pod jehož názvem již nějaký v grafu je, nebude vrchol přidán. Chceme-li přidávat hranu, využijeme funkci ***AddEdge(string odkud, string kam, Edge hrana)***. První dva parametry udávají názvy vrcholů, mezi kterými vede hrana. Pokud zadáme neexistující vrchol, dojde k běhové chybě. Třetím parametrem udáme vlastnosti, které má hrana mít a které jsou uloženy v proměnné typu *Edge*. Pro odstranění vrcholu použijeme funkci ***DeleteVertex(string nazev)***. Ta odstraní vrchol se shodným názvem, jako má parametr. Neexistuje-li, nastane běhová chyba. Obdobou je funkce ***DeleteEdge(string odkud, string kam)*** mazající hranu grafu.

Zde najdete seznam dalších nastavitelných vlastností typu *Graph*:

VertexFillColor – změnou hodnoty dojde k nastavení barev všech vrcholů grafu na požadovanou hodnotu

VertexBorderColor – nastavení barvy okraje všech vrcholů grafu

VertexWidth – nastavení šířky všech vrcholů grafu

VertexTextColor - nastavení barvy textu všech vrcholů grafu

VertexBorderWidth - nastavení šířky okraje všech vrcholů grafu

TextSize – velikost písma při vykreslování grafu

EdgeColor – nastavení barvy všech hran grafu

EdgeWidth – nastavení šířky všech hran grafu

Top – vertikální poloha grafu (od horního okraje)

Left – horizontální poloha grafu (od levého okraje)

Oriented – udává, zda je graf orientovaný (0 = neorientovaný = výchozí hodnota, nenula = orientovaný)

visible – udává, zda se má graf vykreslovat (0 = nevykreslovat, nenula = vykreslovat = výchozí hodnota)

ShowNames – udává, zda se mají vypisovat názvy vrcholů (0 = nevykreslovat, nenula = vykreslovat = výchozí hodnota)

ShowNumbers – udává, zda se mají vykreslovat váhy hran (0 = nevykreslovat, nenula = vykreslovat = výchozí hodnota)

Dále existují ještě tři funkce usnadňující nám práci s grafy (jsou to klasické funkce, ne členské funkce grafu):

ExistEdge(Graph graf, string odkud, string kam) – vrací nenulovou hodnotu, pokud v grafu (první parametr) existuje požadovaná hrana (2. a 3. parametr), jinak vrací nulu

ExistVertex(Graph graf, string vrchol) – vrací nenulovou hodnotu, pokud v grafu (první parametr) existuje požadovaný vrchol (2. parametr), jinak vrací nulu

ExistPath(Graph graf, string odkud, string kam) – vrací nenulovou hodnotu, pokud v grafu (první parametr) existuje cesta mezi dvěma vrcholy (2. a 3. parametr), jinak vrací nulu

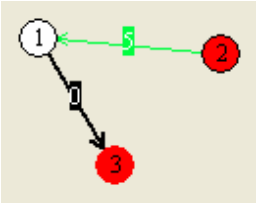
K vytváření grafu doporučuji použít nástroj popsaný v podkapitole 5.3. Následuje ukázka kódu pro vytvoření jednoduchého grafu:

příklad 5.4.13 – návrh grafu:

```
Graph graf; // proměnná reprezentující graf
void main()
{
        graf.Oriented = 1; // chceme orientovaný graf
        Vertex vrchol; // pomocná proměnná pro vkládání vrcholů
        vrchol.Left = 143; // X-ová souřadnice
        vrchol.Top = 162; // Y-ová souřadnice
        graf.AddVertex(vrchol, "1"); // přidání vrcholu do grafu
```

```
        vrchol.FillColor = 255; // barva výplně
        vrchol.Left = 236; // X-ová souřadnice
        vrchol.Top = 170; // Y-ová souřadnice
        graf.AddVertex(vrchol, "2"); // přidání vrcholu do grafu
        vrchol.BorderColor = 255; // barva okraje
        vrchol.Left = 182; // X-ová souřadnice
        vrchol.Top = 227; // Y-ová souřadnice
        graf.AddVertex(vrchol, "3"); // přidání vrcholu do grafu
        Edge hrana; // pomocná proměnná pro vkládání hran
        hrana.width = 2; // šířka hrany
        graf.AddEdge("1", "3", hrana); // přidání hrany do grafu
        hrana.color = 4259584; // barva hrany
        hrana.width = 1; // šířka hrany
        hrana.vaha = 5; // váha hrany
        graf.AddEdge("2", "1", hrana); // přidání hrany do grafu
}
```

a takto vypadá vytvořený graf:



Iterátory na grafech

Pro usnadnění práce s grafy jsou ve skriptovacím jazyce definovány datové typy ***EdgeIterator*** a ***VertexIterator*** sloužící k procházení seznamu všech hran resp. vrcholů grafu. Po deklaraci proměnných tohoto typu je nutno do nich dosadit iterátor odpovídající nějakému grafu. To se zajistí zavoláním funkce ***SetEdgeIterator(Graph graf)*** resp. ***SetVertexIterator(Graph graf)***. Máme-li iterátor spojen s konkrétním grafem, určíme jeho počáteční hodnotu, a to zavoláním jejich členských funkcí (přístup tečkovou notací) ***First()*** nebo ***Last()***. Jak již názvy napovídají, první jmenovaná nastaví iterátor na první hodnotu seznamu a druhá na prvek poslední. Chceme-li iterátor posunout o jednu pozici vpřed,

užijeme funkci ***Next()***. Posun vzad zajistíme zavoláním ***Prev()***. Seznam vrcholů je řazen abecedně podle jmen vrcholů, hrany jsou řazeny vzestupně podle jejich vah. Seznam hran je optimalizován pro orientované grafy. V případě grafu neorientovaného se tedy každá hrana projde dvakrát (v různých pořadích jejich vrcholů).

Pokud chceme z iterátoru číst, musíme si nejprve ověřit, že je platný (tzn. je spojen s nějakým grafem a ukazuje na platný prvek). Tuto kontrolu lze provést snadno – stačí napsat do podmíněného příkazu či cyklu jako podmínku název iterátoru. Ten je vyhodnocen jako pravdivý, pokud je platný. V opačném případě bude výraz vyhodnocen jako nepravda.

Samotné načtení dat z iterátoru zajišťuje funkce ***Load(...)***. V případě načítání vrcholu bere funkce jeden parametr – tím musí být proměnná typu *string*, do které se uloží název načteného vrcholu. Načítáme-li data z iterátoru hran, musíme předat funkci jako parametry dvě proměnné typu *string*, do kterých se uloží název výchozího a cílového vrcholu hrany.

příklad 5.4.14 – vypsání všech vrcholů a hran grafu (pomocí iterátorů):

```
Graph graf;
void main()
{
        VertexIterator vi; //iterátor na procházení hran
        vi = SetVertexIterator(graf); // inicializace iterátoru
        string v1, v2; // pomocné proměnné
        vi.First(); // vybrání prvního vrcholu
        while (vi) // dokud máme vrcholy
        {
                vi.Load(v1); //načtení názvu vrcholu do proměnné
                write("Vrchol: "+v1); // vypsání vrcholu
                vi.Next(); // vybrání dalšího vrcholu
        }
        EdgeIterator ei  = SetEdgeIterator(graf); // inicializace iterátoru hran
        ei.Last(); // vybrání první hrany (ta s největší vahou)
        while (ei) { // dokud máme hrany
                ei.Load(v1,v2); //načtení názvu vrcholů hrany do proměnných
                write("Hrana z "+v1+" do "+v2); // vypsání hrany
                ei.Prev(); // vybrání předchozí hrany
        }
}
```

Přerušení běhu skriptu

Aby bylo uživateli umožněno pozorování běhu algoritmu a nebyl mu ukázán pouze jeho výsledek, je nezbytná možnost přerušení skriptu a současného vykreslení aktuálního stavu algoritmu.

Skript se přerušuje příkazem ***BREAK(<číslo zarážky>)***. Parametrem příkazu je číslo (musí se jednat o číslici, nikoliv o číselný výraz či proměnnou) udávající, se kterou zarážkou je přerušení svázáno. U jednotlivých zarážek můžeme nastavit komentář a seznam sledovaných proměnných (viz podkapitola 5.3).

6. Závěr

6.1. Zhodnocení programu vzhledem k cílům práce

Domnívám se, že program AlgoShow splnil všechny cíle, které byly stanoveny. Aplikace je plně použitelná pro názornou výuku algoritmů a je možno relativně jednoduše navrhovat vlastní vizualizace algoritmů.

Základním požadavkem byla právě možnost rozšiřitelnosti programu o další vizualizace, a to pokud možno co nejjednodušším způsobem. To se dle mého názoru podařilo díky návrhu vlastního skriptovacího jazyka, který umožňuje snadné (a v některých případech i zcela automatické) zobrazování grafických prvků ulehčujících uživateli pochopení algoritmu.

Úkolem bylo, aby skriptovací jazyk podporoval zejména algoritmy na třídění v poli, práci s binárními stromy (resp. spojovými seznamy) a grafy. Pro tyto oblasti má jazyk zabudovány některé specifické funkce, ale neznamená to, že v jazyce nelze napsat algoritmus, který do nich nespadá.

V oblasti uživatelské přívětivosti bylo také dosaženo cíle, aby ukazování průběhu algoritmu bylo co nejpřehlednější a přitom nabídlo co nejvíce informací. Kromě samotné vizualizace a zdrojového kódu jsou uživateli zobrazovány také informace o hodnotách proměnných a komentář o právě prováděné akci.

Původní požadavek, aby navrhování vlastních vizualizací bylo co nejjednodušší, byl splněn díky řadě intuitivních nástrojů usnadňujících psaní zdrojového kódu.

6.2. Možná rozšíření do budoucna

I když je program AlgoShow již plně použitelný, jistě by se nabízela řada možností, jak jeho funkce ještě rozšířit.

Přidávání nových vizualizací je umožněno uživatelům, kteří je mohou navrhnout ve speciálním skriptovacím jazyce. Rozšířit by se ale dal samotný skriptovací jazyk tak, aby nabízel specifické funkce i pro další typy algoritmů a datových struktur. Nabízela by se například práce s automaty a gramatikami či se složitějšími dynamickými datovými strukturami (B-stromy apod.).

Samotná vizualizace by se pak dala rozšířit o možnost animací či použití 3D zobrazení. To by ale vyžadovalo použití jiného způsobu vykreslování, protože GDI je pro složitější grafické operace nepoužitelné.

I přesto, že rychlost není u vizualizací algoritmů rozhodující vlastností, bylo by možno úpravou způsobu interpretace skriptů dosáhnout zrychlení běhu algoritmů. V tom případě by bylo nutné použít překladu skriptu do mezikódu, který by bylo jednodušší interpretovat.

Jistě by se dalo upravovat k lepšímu i uživatelské rozhraní a našlo by se i mnoho dalších námětů na rozšíření, ale to je již spíše otázka dlouhodobého vývoje programu a reakcí na případné podněty uživatelů.

7. Seznam použité literatury

[1] MSDN Library: http://msdn.microsoft.com

[2] Jeff Prosise (2000): Programování ve Windows pomocí MFC, Computer Press

[3] Algoritmus – http://cs.wikipedia.org/wiki/Algoritmus

[4] CodeGuru – http://www.codeguru.com

[5] http://web-cat.cs.vt.edu/AlgovizWiki/Catalog

[6] http://www.cs.hope.edu/alganim/animator/Animator.html

[7] http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/AVLtree/avl.html

[8] http://www.geocities.com/siliconvalley/network/1854/Rbt.html

[9] http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm