Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Žiga Ham

Ogrodje za vizualizacijo algoritmov

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Marko Robnik-Šikonja

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00118/2013
Datum: 12.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ŽIGA HAM**

Naslov: **OGRODJE ZA VIZUALIZACIJO ALGORITMOV**
ALGORITHM VISUALIZATION FRAMEWORK

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Razumevanje in poučevanje algoritmov nam olajšajo različne vizualizacijo njihovega delovanja. Delovanje mnogih pomembnih algoritmov lahko predstavimo v obliki grafov, zato je smiselno, da k vizualizaciji pristopimo sistemsko, z izdelavo dovolj splošnega programskega orodja, ki bo omogočilo, da se delovanje algoritma grafično predstavi znotraj same kode algoritma.
Zasnujte splošno ogrodje za vizualizacijo algoritmov v obliki spletnega programskega orodja, ki uporabniku omogoča, da definira pomembne dogodke med izvajanjem algoritma. Ti dogodki naj se med izvajanjem algoritma prikažejo na zaslonu in tako pomagajo pri razumevanju dogajanja. Uporabnost pristopa ilustrirajte z nekaj primeri vizualizacij algoritmov.

Mentor: Dekan:

izr. prof. dr. Marko Robnik Šikonja prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Žiga Ham, z vpisno številko **63100252**, sem avtor diplomskega dela z naslovom:

Ogrodje za vizualizacijo algoritmov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Marka Robnika Šikonje,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 15. septembra 2013 Podpis avtorja:

Kazalo

Seznam uporabljenih kratic

- AJAX — (ang.) Asynchronous JavaScript and XML; asinhroni JavaScript in XML (skupina tehnik spletnega programiranja za ustvarjanje asinhronih spletnih aplikacij, konkretneje asinhrono prenašanje podatkov z in na spletni strežnik).
- HTML — (ang.) HyperText Markup Language; jezik za označevanje hiperteksta (glavni označevalni jezik za ustvarjanje spletnih aplikacij s sintakso, podobno XML).
- HTTP — (ang.) Hypertext Transfer Protocol; protokol za prenos hiperteksta (osnovni protokol za svetovni splet).
- MVC — (ang.) Model-view-controller; model-pogled-krmilnik (programska arhitektura, ki jo pogosto srečamo pri spletnih aplikacijah).
- JSON — (ang.) JavaScript Object Notation; JavaScript notacija za objekte (standardni tekstovni format, ki ga ponavadi uporabljamo za prenos podatkov med strežnikom in spletno aplikacijo).
- SVG — (ang.) Scalable Vector Graphics; skalabilna vektorska grafika (format za vektorske slike, zasnovan na XML).

Povzetek

Diplomska naloga opisuje razvoj in delovanje ogrodja za vizualizacijo algoritmov. Ogrodje je sestavljeno iz spletne strani, orodja za zajem dogodkov v algoritmih ter Javascript ogrodja za izdelavo vizualizacije. Spletna stran deluje kot zbirka vizualizacij in modulov, ki so sestavni deli vizualizacij. Orodje za zajem dogodkov deluje tako, da v programsko kodo dodamo posebne ukaze, ki zapisujejo dogajanje med izvajanjem algoritma. Ogrodje za izdelavo vizualizacij je narejeno z uporabo sodobnih spletnih tehnologij in namenjeno izvajanju v spletnem brskalniku. Pomaga nam pri sestavi vizualizacije in tolmači dogodke, ki so zapisani v dnevniku, narejenem z orodjem za zajem dogodkov. Diplomska naloga opisuje tudi izdelavo primerov vizualizacij, ki vključujejo iskanje najkrajše poti in topološko urejanje.

Ključne besede: algoritmi, vizualizacija, spletna aplikacija, ogrodje, ogrodje za vizualizacijo algoritmov, vizualizacija najkrajše poti, vizualizacija topološkega urejanja

Abstract

We present a design and implementation of an algorithm visualization framework. The framework consists of a website, a tool for capturing events in algorithms and a JavaScript framework for building visualizations. The website serves as a collection of visualizations and modules that are integral parts of visualizations. The tool for capturing events adds special commands to the source code, which record events during the execution of the algorithm. The JavaScript framework for building visualizations uses modern web technologies and runs in web browsers. It helps build visualizations and interprets the captured events. We designed and implemented visualization examples including the shortest path algorithm and topological sort.

Keywords: algorithms, visualization, web application, framework, algorithm visualization framework, shortest path visualization, topological sort visualization

Poglavje 1

Uvod

Vizualizacije algoritmov nam lahko pomagajo pri razumevanju in so zato uporabne pri učenju. Poleg tega jih lahko uporabljamo tudi za preučevanje in nadgrajevanje algoritmov.

Obstaja mnogo različnih vizualizacij algoritmov, kljub temu pa je ustvarjanje novih še vedno zamudno delo. Še posebej pri algoritmih z grafi se velik del truda ponavlja pri vsaki vizualizaciji. Z dobro načrtovanim orodjem bi lahko dosegli, da bi ustvarjanje vizualizacij postalo preprosto in hitro. To bi pomenilo, da bi vizualizacije postale uporabne tudi za prej nepraktične primere, kot je na primer razhroščevanje programov.

V drugem poglavju predstavimo zasnovo spletne aplikacije. V tretjem poglavju se posvetimo varnosti pri realizaciji spletnih aplikacij, ki gostujejo tujo izvorno kodo. V četrtem poglavju začnemo s prvim korakom pri ustvarjanju vizualizacije, ki je implementacija algoritma in zajem dogodkov. Če vas zanimajo le vizualizacije, začnite s tem poglavjem. V petem poglavju nadaljujemo z ustvarjanjem vizualizacije, na vrsti je ogrodje za predvajanje dogodkov in izrisovanje vizualizacije. V šestem poglavju predstavimo osnovne module, ki jih uporabljamo za izdelavo vizualizacije. V sedmem poglavju predstavimo nekaj primerov vizualizacij in v osmem poglavju predstavimo celoto narejenega ter podamo nekaj idej za nadaljnje delo.

Poglavje 2

Tehnologije spletne aplikacije

Spletna stran deluje kot zbirka vizualizacij in modulov za razvoj vizualizacij algoritmov, in seveda tudi kot predstavitev projekta. V tem poglavju na kratko predstavimo nekaj glavnih tehnologij, uporabljenih za izgradnjo spletne aplikacije.

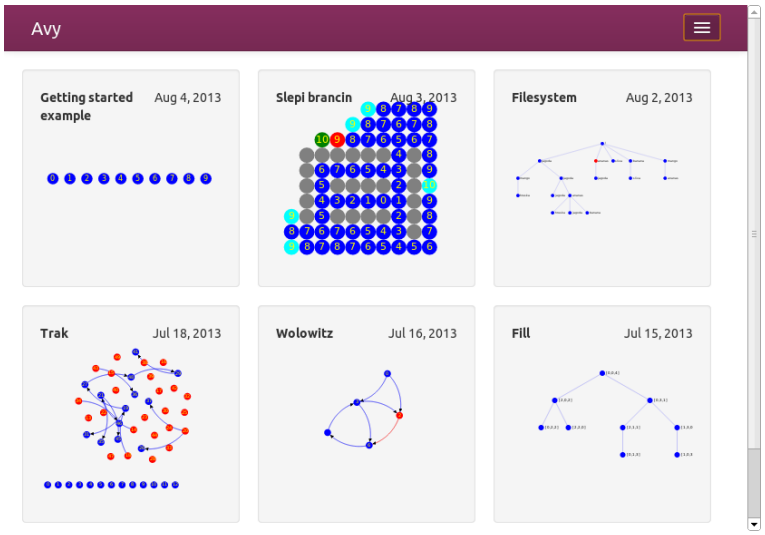
2.1 AppEngine

Google App Engine[7] je storitev v oblaku za gostovanje spletnih aplikacij. Ponuja nam gostovanje aplikacij v jezikih Java/Python/Go. Poleg gostovanja naše aplikacije nam ponuja tudi dodatne storitve, kot je podatkovna baza.

Razlog, da smo se odločili za to platformo, je predvsem v tem, da nam brez dodatnega dela omogoča skaliranje naše aplikacije. Poleg tega pa storitev postane plačljiva šele, ko ima naša aplikacija več prometa in nikoli ne plačamo več kot le za naš promet.

2.2 AngularJS

AngularJS[4] je odprtokodno ogrodje za Javascript spletne aplikacije s poudarkom na MVC. MVC (model-pogled-krmilnik, ang. *model-view-controller*) je programska arhitektura, ki jo pogosto srečamo pri spletnih aplikacijah.



Slika 2.1: Zaslonski posnetek spletne strani.

AngularJS spletne strani delujejo tako, da se aplikacija naloži, ko obiščemo prvo stran, ostale vsebine pa se naložijo preko asinhronih zahtev (AJAX).

Tak tip spletne strani poenostavi delo spletnega strežnika, saj potrebuje le vmesnik za pridobivanje vsebine, HTML pa se generira lokalno pri uporabniku. Slabost je stroga zahteva po Javascriptu pri uporabniku, kar pomeni, da bomo imeli težave s spletnimi iskalniki.

2.3 Go

Go[11] je programski jezik, namenjen predvsem spletnim storitvam. Sintaksa je nekoliko podobna jeziku C, vendar se jezika močno razlikujeta. Zasnovan je bil kot način za lažje grajenje spletnih storitev z enako učinkovitostjo, kot tiste, narejene v C++. Zaradi tega ima že brez dodatnih knjižnic podporo sočasnosti (gorutine), HTTP strežnik, JSON (Javascript notacija za objekte, ang. *JavaScript Object Notation*) razčlenjevalnik in še veliko drugih stvari. Od jezikov, kot sta C++ in Java, se razlikuje tudi po nekoliko drugačni implementaciji objektno orientiranega programiranja.

Poglavje 3

Varnost

Ker spletna aplikacija posreduje izvorno kodo oziroma kar poljubne dokumente, ki jih naložijo uporabniki, je zelo pomembna zaščita pred napadi. Poleg vseh običajnih načinov napada na spletne strani se v tem primeru pojavijo še številni drugi. Gostujoča izvorna koda se na spletni strani tudi izvaja, kar predstavlja še dodatno ranljivost.

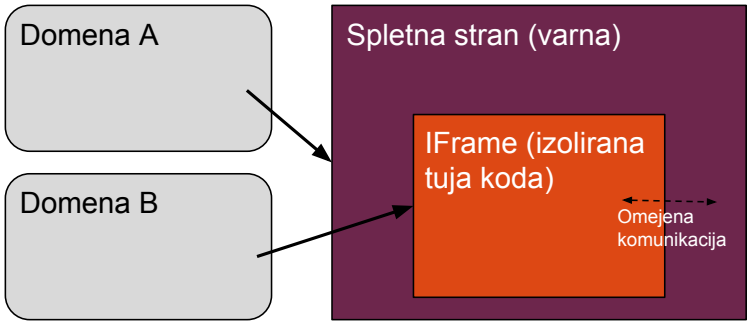
V tem poglavju predstavimo nevarnosti in možne zaščite pred temi nevarnostmi.

3.1 Nevarnosti

Izvajanje tuje kode na spletni strani nam predstavlja dve osnovni nevarnosti:

- Kraja seje — če bi tuja izvorna koda lahko brala piškotke ali dele spletne strani, bi napadalec lahko prišel do seje in posledično do celotnega uporabniškega računa.

- Spreminjanje uporabniške izkušnje — napadalec lahko v nekaterih primerih spremeni vsebino spletne strani ali pa preusmeri uporabnika na drugo spletno stran. Zelo pomembno je, da tega ne dopuščamo, saj lahko vodi do kraje osebnih podatkov.



Slika 3.1: Posredovanje tuje kode z druge domene v IFrame.

3.2 Možne rešitve

3.2.1 Tuje kode ne posredujemo in izvajamo v prvotni obliki

Ena izmed možnih rešitev je, da tuje izvorne kode ne posredujemo in izvajamo neposredno. Pred izvajanjem lahko izvorno kodo pregledamo za morebitne problematične ukaze. Pri tem pristopu moramo biti temeljiti. Če pozabimo na kakšno, mogoče nekoliko nenavadno, možnost izvajanja operacije, lahko to postane ranljivost. Zaradi tega je dobro uporabljati že preizkušene rešitve, kot je Google Caja[8]. Caja je orodje za varno izvajanje tuje spletne aplikacije znotraj naše spletne strani (ang. *embed*). Ne smemo pozabiti, da izvorne kode ne smemo posredovati v prvotni obliki, tudi če je ne izvajamo. Napadalec lahko kodo, ki jo posreduje spletna stran, pokliče iz svoje spletne strani (IFrame, preusmeritev), in ker je bila posredovana z našega strežnika, bo imela dostop do naših piškotkov in drugih storitev. Dovolj je, če tujo kodo spremenimo tako, da ni neposredno izvedljiva (npr. dodamo glavo).

3.2.2 Tujo kodo posredujemo in izvajamo na drugi domeni

Kot smo že omenili, je že samo posredovanje tuje kode lahko nevarno. Tem nevarnostim se izognemo tako, da tuje datoteke posredujemo na drugi do-

meni. Če se napadalec prebije do piškotkov, bodo le ti prazni, saj so piškotki omejeni na domeno. Prav tako nas bodo ščitile zaščite, ki prepovedujejo večdomensko izvajanje kode (ang. *cross-site scripting*).

Izvajanje lahko zagotovimo tako, da tujo kodo, ali pa kar del spletne strani, izvajamo v elementu IFrame. Če je spletna stran, ki se izvaja v elementu IFrame, na drugi domeni, bo dostop do glavne strani omejen na sporočila[6]. Ko implementiramo to rešitev, je najboljše, da si zamislimo, da ima napadalec popoln nadzor nad vsebino v elementu IFrame (oranžen pravokotnik na sliki 3.1), in mu zato ne podajamo osebnih podatkov. IFrame (in ločena domena) tako služi kot pregrada med nevarnim (anonimnim) svetom in našo spletno stranjo.

Poddomene in piškotki

Piškotki (ang. *cookies*) so omejeni na določeno domeno. Če je naslov naše spletne strani *example.com*, tuja koda na *example.org* nima dostopa do naših piškotkov. Za poddomene veljajo posebna pravila[1].

Poddomena lahko bere piškotke glavne domene. Torej če domena *example.com* nastavi piškotek, ga lahko bere tudi *x.example.com*. Ker želimo to preprečiti, moramo ločiti poddomeni (npr. *x.example.com* in *y.example.com*).

Branje smo rešili, ostaja nam še pisanje. Pisanja žal ne moremo popolnoma preprečiti. Vsaka poddomena lahko nastavi piškotek na glavno domeno (npr. *x.example.com* lahko nastavi za *example.com*) in posledično ga lahko prebere druga poddomena (npr. *y.example.com*). Temu pravimo vsiljevanje piškotkov (ang. *cookie forcing*). Na srečo to ni velik varnostni problem. Medtem ko lahko z branjem piškotka nekomu ukrademo sejo, s pisanjem lahko le otežimo uporabo storitve (npr. uporabniku izbrišemo sejni piškotek in ga tako odjavimo iz spletne strani). Za popolno varnost je najbolje uporabiti popolnoma ločeno domeno.

Ostale nevarnosti

Piškotki so del osnovne funkcionalnosti brskalnikov in zato za njih veljajo stroga pravila, ki so določena s standardom[1] (vseeno se nekateri brskalniki ne držijo pravil, ali pa uporabljajo zastarelo različico standarda). Na drugi strani imamo nekoliko bolj kaotično situacijo. Vtičniki, kot sta Flash ali Java, imajo svoja pravila za dostop med spletnimi stranmi. Če si želimo zagotoviti varnost, moramo uporabiti tudi ločen IP naslov (poleg druge domene) za posredovanje tuje kode.

Eden izmed problemov je varnostna luknja v starih verzijah Java vtičnika, ki omogoča krajo piškotka z druge domene na istem IP naslovu[5].

Poglavje 4

Implementacija algoritmov in zajem dogodkov

Prvi korak pri vizualizaciji algoritma je implementacija tega algoritma. Če želimo prikazati potek algoritma, moramo nato zajeti posamezne korake. To lahko storimo na različne načine, katerega bomo izbrali, pa je odvisno od naših ciljev.

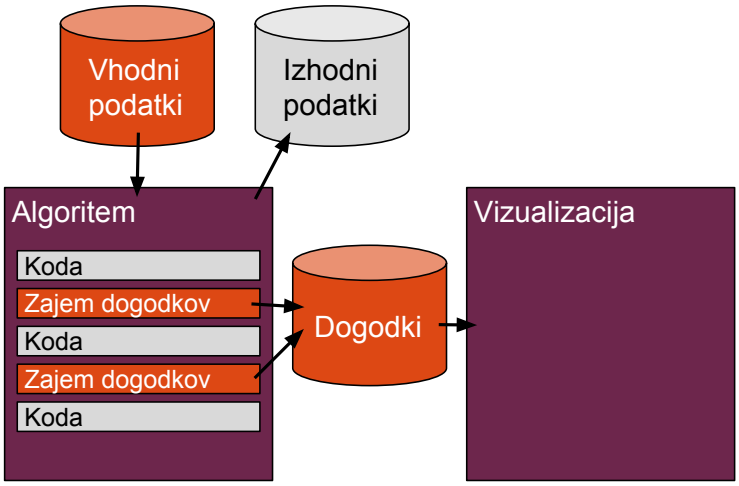
V tem poglavju predstavimo orodje za zajem dogodkov. Začnemo s cilji, nato nadaljujemo z implementacijo v različnih jezikih.

4.1 Cilji ogrodja

- Vsi programski jeziki — rešitev naj ne bo omejena na en programski jezik. Uporabniki imajo različne želje. Nekateri programski jeziki so bolj primerni za določene probleme.

- Enostavna vgradnja v obstoječe programe — dodajanje vizualizacije obstoječi implementaciji mora biti čim bolj enostavno. To pomeni, da implementacija algoritma ostane taka, kot smo je navajeni.

- Ne spremeni toka programa — vizualizacija naj ne spremeni delovanja programa, torej lahko samo dodaja funkcionalnosti.



Slika 4.1: Tok podatkov pri zajemanju dogodkov.

- Možen izklop — zajemanje dogodkov naj bo možno izklopiti. Potratno bi bilo zajemati dogodke vedno, kadar uporabljamo program, vodenje dveh ločenih verzij (z zajemanjem in brez) pa ni praktično.

  Če je možno, naj program deluje tudi, če knjižnice za zajemanje ni na voljo. Tako imamo lahko delujoč program brez zunanjih odvisnosti, kar je primerno tudi za spletne sodnike (avtomatizirani sistemi, ki preverijo pravilnost algoritma za reševanje določene naloge).

4.2 Format zapisa dogodkov

Format zapisa dogodkov je skupen vizualizacijski jezik za vse programske jezike, ki jih uporabljamo. Ne glede na to, kateri programski jezik smo uporabili, mora biti opis vizualizacije enak in ta končni rezultat mora biti razumljiv za Javascript implementacijo vizualizacije.

Za vsak dogodek zapišemo ukaz in njegove parametre. Ukaz je predstavljen kot niz znakov. Parametri so lahko različnih tipov. Podpirati želimo vsaj numerične in tekstovne parametre, uporabna pa bi bila tudi podpora kompleksnih tipov. Glede na to, da se bodo parametri naprej obdelovali v

Javascriptu in je shema prosta (vsebina je odvisna od posameznega modula), je najbolj primerna oblika za serializacijo parametrov JSON. Dobra izbira je tudi zato, ker obstaja že veliko knjižnic za serializacijo, v nekaterih jezikih pa je serializacija celo del standardne knjižnice.

JSON je, poenostavljeno povedano, sintaksa za števila, nize, sezname in objekte v programskem jeziku Javascript. Poenostavljeno pravimo zato, ker so v resnici stvari bolj kompleksne, je pa vseeno ena bolj preprostih oblik serializacije. Razlike so večinoma v tem, da Javascript omogoča več enakovrednih načinov zapisa, JSON pa predstavlja eno od teh možnosti. Torej je JSON z majhno izjemo[3] podmnožica Javascripta.

Ko imamo določen način serializacije posameznih delov, lahko določimo še celotno obliko. Ukazi so ločeni z znakom za novo vrstico (torej je vsak ukaz s parametri v svoji vrstici), posamezna vrstica pa izgleda takole:

```
ukaz(paramter1, parameter2, parameter3, ...)
```

4.3 Python

Začnimo s Pythonom, ker je skriptni jezik z vgrajeno JSON podporo in je zato v njem implementacija tega orodja najenostavnejša.

Najtežje dosegljiv cilj je možnost izklopa zajemanja dogodkov, ko knjižnice ni na voljo. Del kode, ki preveri, če želimo vklopiti knjižnico, in nadomestek za knjižnico, morata biti vedno v vsaki datoteki, ki izkorišča to možnost. Zaradi tega želimo ustvariti čim manjši nadomestek za knjižnico. To lahko naredimo tako, da vso funkcionalnost zapakiramo v eno funkcijo s spremenljivo dolžino seznama argumentov. V tem primeru bo nadomestek vseboval samo eno prazno funkcijo.

```
def avy(*argv):
    pass
```

Ker Python omogoča preverjanje tipov parametrov med izvajanjem, lahko prava funkcija (dejanska implementacija funkcionalnosti) enostavno izvede

pravilno akcijo. Za doseg našega cilja nam manjka še nalaganje pravilnega dela kode. To dosežemo, ali z enostavnim if stavkom, ki preveri za parameter v ukazni vrstici, ali s try except blokom, ki ujame napako pri nalaganja knjižnice.

```
try:
    from avy import avy
except:
    nadomestek
```

Naslednji korak je dejanska implementacija *avy* funkcije, kar je v Pythonu zelo enostavno. Prvi parameter neposredno zapišemo v datoteko (gre za ukaz), ostale (gre za parametre ukaza) pa serializiramo z modulom JSON, ki je del standardne knjižnice.

```
json.dumps(arg)
```

Enostaven primer zajema dogodkov v jeziku Python izgleda tako:

```
avy('start') # Začne zajem dogodkov.
list = []
for i in range(10):
  avy('step') # Korak v animaciji vizualizacije.
  list.append(i)
  # Zajame dogodek dodajanja v sezenam.
  # list.add je ciljna metoda, i je parameter
  avy('list.add', i)
```

4.4 C++

V programskem jeziku C++ je implementacija nekoliko bolj zahtevna. Razloga za to sta slabša podpora spremenljivi dolžini seznama argumentov in nezmožnost preverjanja tipov argumentov med izvajanjem (statično tipiziran jezik).

Kot v Pythonu, je tudi tu najbolje zapakirati vso funkcionalnost v eno funkcijo, zato da je nadomestna implementacija čim krajša. Funkcijo bomo v tem primeru skrili za makro, saj to prinese nekaj prednosti:

1. Prazna implementacija makroja je krajša in preprostejša.

```
#define AVY(...)
```

2. Znotraj klica praznega makroja lahko kličemo tudi nedefinirane funkcije/makroje, saj se bo celoten klic enostavno izbrisal pred prevajanjem. To pomeni, da imamo lahko dodatne pomožne funkcije/makroje, ki jih ne bo potrebno implementirati v nadomestni implementaciji.

Ko imamo nadomestno implementacijo, nam manjka le še pogoj za izklop. Pogoj za vključitev knjižnice mora biti seveda predprocesorski, lahko pa imamo še dodaten pogoj za izklop ob izvajanju programa (če je bila knjižnica naložena ob prevajanju). To naredimo z *ifdef*.

```
#ifdef EAVY
#include <avy.h>
#else
#define AVY(...)
#endif
```

Pri implementaciji knjižnice v jeziku C++ naletimo na dve težavi. Prva težava je dolžina seznama parametrov. Kljub temu, da C++ omogoča spremenljivo dolžino seznama parametrov, ob tem ne poda dolžine tega seznama. Ta problem rešimo s spretno uporabo makrojev. Rešitev ni najlepša, vendar enakovredne funkcionalnosti ne moremo doseči drugače.

```
#define AVY_N_ARGS(args...) AVY_N_ARGS_HELPER(dummy, \
  ##args, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
#define AVY_N_ARGS_HELPER(dummy, x1, x2, x3, x4, x5, \
  x6, x7, x8, x9, n, x...) n
```

Argumenti, ki se vrinejo v klic drugega makroja pri *##args*, zamaknejo številke 9, 8, 7... za toliko mest, kot je argumentov. Tako bo drugi makro kot argument n dobil število argumentov, vrinjenih preko *##args*. Tako dobimo število argumentov kot izhod makroja. Slabost te implementacije je, da deluje do maksimalno 9 argumentov oz. do toliko, kot naštejemo števil pri implementaciji.

Druga težava je določanje tipa parametrov. Med prevajanjem v jeziku C++ poznamo tip spremenljivk, med izvajanjem (ko uporabljamo seznam argumentov spremenljive dolžine) pa ne. Ker ne poznamo tipa spremenljivke, ne vemo, kako bi jo serializirali. Ena od rešitev je, da uporabimo prekrivanje funkcij z različnimi tipi argumentov.

```
zapakiraj(int a);
zapakiraj(const char *a);
```

Te funkcije zapakirajo argumente (različnih tipov) v neko skupno obliko (npr. niz znakov). V funkciji s seznamom argumentov spremenljive dolžine potem vemo, da so vsi argumenti v tej skupni obliki. Slabost implementacija je to, da moramo vse argumente v našo funkcijo poslati skozi funkcijo za "pakiranje":

```
glavna_funkcija(zapakiraj(3), zapakiraj("niz"));
```

Če vse to združimo, dobimo končno obliko našega vmesnika za C++:

```
AVY(ime, P(parameter1), P(parameter2), ...)
```

Enostaven primer zajema dogodkov v jeziku C++ izgleda tako:

```
int main() {
  AVY(start); // Začne zajem dogodkov.
  vector<int> list;
  for (int i = 0; i < 10; i++) {
    AVY(step); // Korak v animaciji vizualizacije.
    list.push_back(i);
    // Zajame dogodek dodajanja v sezenam.
```

```
    // list.add je ciljna metoda, i je parameter
    AVY(list.add, P(i));
  }
}
```

4.5 Java

V programskem jeziku Java nam zaradi strogosti prevajalnika ni uspelo omogočiti možnosti delovanja, ko knjižnica ni prisotna. Orodje je implementirano kot običajna knjižnica z razredom za zajem dogodkov. V program jo vključimo s klicem *import*:

```
import net.algoviz.Avy;
```

Pri prevajanju pa moramo vključiti datoteko *avy.jar*, ki vsebuje tudi knjižnico Gson[10] za JSON serializacijo argumentov.

Enostaven primer zajema dogodkov v jeziku Java izgleda tako:

```
public class Algorithm {
  public static void main(String[] argv) {
    Avy.cmd("start"); // Začne zajem dogodkov.
    ArrayList<Integer> list = new ArrayList<Integer>();
    for (int i = 0; i < 10; i++) {
      Avy.cmd("step"); // Korak v animaciji vizualizacije.
      list.add(i);
      // Zajame dogodek dodajanja v sezenam.
      // list.add je ciljna metoda, i je parameter
      Avy.cmd("list.add", i);
    }
    Avy.close(); // Zapre vse odprte datoteke.
  }
}
```

Poglavje 5

Ogrodje za vizualizacijo

Ko imamo algoritem in dogodke za vizualizacijo, je naslednji korak prikaz vizualizacije. Ker vsake vizualizacije ne želimo izdelovati od začetka, potrebujemo ogrodje, ki nudi osnovne gradnike in pomaga pri ponovni uporabi delov vizualizacije.

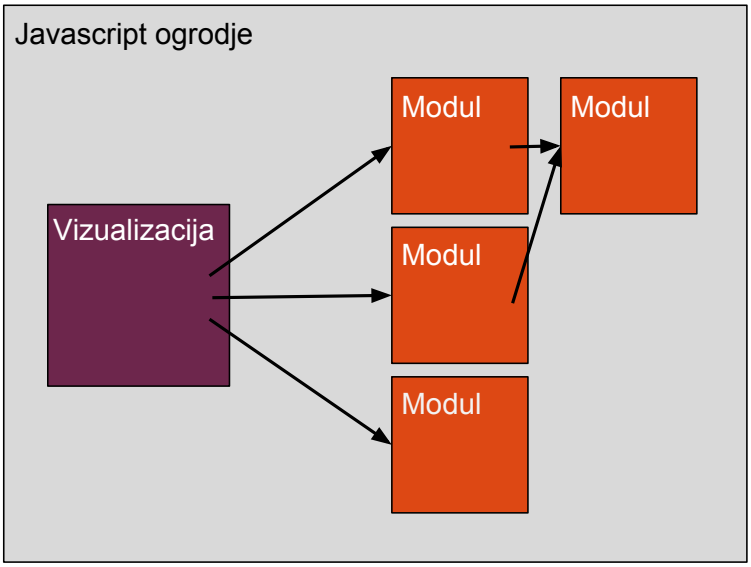
5.1 Modularna zasnova

Ogrodje je zasnovano modularno - vsak sklop funkcionalnosti je zapakiran v svoj modul. Vsak modul lahko tudi uporablja poljubno število drugih modulov. Tako lahko funkcionalnost gradimo v več nivojih.

Moduli lahko v splošnem implementirajo poljubne metode in imajo poljubne lastnosti, če pa želijo biti kompatibilni z določenimi moduli, morajo implementirati metode, ki jih ti moduli zahtevajo. Na primer, modul za prikaz vozlišča grafa mora implementirati metode *enter*, *exit*, *update* in *updatePosition*, da ga lahko uporabljamo z modulom *graph*.

5.1.1 RequireJS

Moduli lahko druge module zahtevajo kot argument, ali pa jih naložijo sami kot odvisnost s pomočjo RequireJS.



Slika 5.1: Modularna zasnova ogrodja.

RequireJS je nalagalnik datotek in modulov za Javascript, namenjen predvsem nalaganju Javascript modulov v spletnih brskalnikih. Omogoča nam, da vsak modul zahteva svoje odvisnosti in to, da ni treba vseh modulov vnaprej naložiti z običajnimi *script* HTML elementi.

5.2 Risalna površina

Risalna površina (*svg*) je eden izmed jedrnih modulov ogrodja za vizualizacijo. Modul je zadolžen za ustvarjanje SVG (skalabilna vektorska grafika, ang. *Scalable Vector Graphics*) elementa, delitev površine na več delov in nekatera uporabniška orodja.

Površina se deli vertikalno na več delov (lahko tudi rekurzivno, torej se vsak del lahko deli še naprej). Vsak del določi višino, ki jo potrebuje. Višina celotne risalne površine je seštevek zahtevanih višin posameznih delov.

Risalna površina ponuja tudi prenos datoteke SVG, ki vsebuje trenutno prikazano vizualizacijo.

5.2.1 D3

Za pomoč pri konstrukciji in manipulaciji SVG vsebine risalna površina in ostali moduli uporabljajo knjižnico D3[9]. Za osnovno uporabo ogrodja nam knjižnice D3 ni treba poznati, če pa želimo ustvarjati svoje module za izrisovanje, je potrebno osnovno poznavanje te knjižnice.

D3 (podatkovno vodeni dokumenti, ang. *Data-Driven Documents*) je Javascript knjižnica, ki z uporabo podatkov ustvari dinamično predstavitev. Lahko jo uporabljamo za vse predstavitve, najpogosteje pa se uporablja pri SVG grafiki. Osnovni princip knjižnice je ustvarjanje izbire (ang. *selection*) z uporabo CSS izbirnika (ang. *CSS selector*). Izbiri potem dodelimo podatke (metoda *data*, vsak element izbire se enači z vrstico podatkov), ter določimo, kaj naj naredi z vsako obstoječo vrstico, ki ima pripadajoč podatek, z vsakim novim podatkom (metoda *enter*, ponavadi ustvarimo novo vrstico), ter kaj naj naredi z vrsticami, ki nimajo pripadajočega podatka (metoda *exit*, ponavadi izbrišemo vrstico).

5.3 Predvajanje dogodkov

Drugi jedrni modul je predvajanje dogodkov (*playback*). Zadolžen je za branje dogodkov, ki smo jih zapisali pri zajemanju dogodkov (poglavje 4). Med predvajanjem za vsak dogodek pokliče pripadajočo funkcijo.

Modul omogoča avtomatsko predvajanje (funkcije izvaja z določenim zamikom) ter ročno premikanje naprej in nazaj. Za premikanje nazaj za vsak klic funkcije, ki ga želimo razveljaviti, pokliče funkcijo *reverse*. To funkcijo mora implementirati vsak modul, ki vsebuje funkcije, ki podpirajo ta način.

Poglavje 6

Moduli

V tem poglavju predstavimo uporabo in delovanje posameznih modulov za vizualizacije. Moduli so osrednjega pomena pri gradnji vizualizacije.

6.1 Vozlišče (*node*)

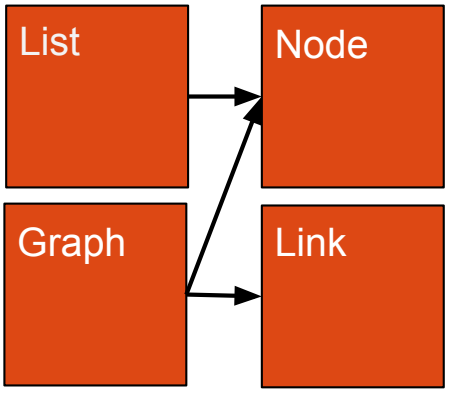
Vozlišče je osnovni modul za prikazovanje vozlišča pri grafih. Narejen je kot samostojen modul, zato da ga lahko uporabljajo vsi ostali moduli, ki potrebujejo to funkcionalnost.

Primer (instanco) tega modula ostali moduli vzamejo kot parameter, tako ga lahko konfiguriramo še preden ga podamo naprej. Opcije so parametri konstruktorja, in sicer:

- radius — polmer kroga za izris,
- text — vir niza znakov, ki ga nariše nad krog,
- textOptions — objekt z nastavitvami za niz znakov (barva, velikost in pozicija).

Če želimo, lahko ta modul nadomestimo s svojo implementacijo, zadostiti pa moramo vmesniku (implementirati moramo enake metode). Metode so:

- enter — nariše nova vozlišča,



Slika 6.1: Odvisnosti med moduli.

- exit — izbriše vozlišča, ki ne obstajajo več,
- update — posodobi vozlišča (barva, besedilo),
- updatePosition — posodobi pozicijo vozlišča.

6.2 Povezava (*link*)

Kot vozlišče je tudi povezava samostojen modul, ki se uporablja pri prikazovanju grafov. Predstavlja povezavo med dvema vozliščema.

Vsebuje dve izvedbi:

- simple — navadna črta,
- arrow — puščica namenjena usmerjenim grafom.

Kot pri vozlišču podamo primer tega modula drugim modulom, dodatnih opcij nima. Lahko naredimo svojo implementacijo z metodami:

- enter — nariše nove povezave,
- exit — izbriše povezave, ki ne obstajajo več,
- update — posodobi povezave (barva, debelina),
- updatePosition — posodobi pozicijo povezave.

6.3 Seznam (*list*)

Modul seznam uporabljamo za prikaz urejenega seznama elementov. V algoritmih so to lahko tabela, vrsta, vektor itd. Za prikaz posameznega elementa uporablja modul vozlišče oz. modul z ekvivalentnim vmesnikom, ki ga podamo kot parameter.

Vmesnik vsebuje metode:

- add — doda nov element na konec seznama. Zahteva identifikator (ID) in dodatne podatke.
- del — izbriše element z danim ID-jem.
- pop — izbriše element s konca seznama.
- update — posodobi element z danim ID-jem. Zahteva ID in podatke, element mora že obstajati, podatki pa so lahko podani samo deloma.

6.4 Graf (*graph*)

Modul graf je zbirka več modulov za prikazovanje grafov.

6.4.1 Graf na osnovi sil (*graph/force*)

Graf na osnovi sil lahko uporabimo za prikaz vsakega grafa. Je najbolj splošen prikaz in ne postavlja omejitev glede oblike grafa. Zaradi tega ne podaja dodatnih informacij s pozicijami vozlišč. Vozlišča so razporejena tako, da je graf čim bolj pregleden. Običajno uporabimo manj splošen prikaz.

Zaradi splošnosti prikaza je tako narejen tudi vmesnik. Podamo mu vozlišča in povezave, lahko pa tudi samo povezave (modul manjkajoča vozlišča naredi avtomatsko).

- addNode — doda vozlišče. Zahteva ID in dodatne podatke.
- delNode — izbriše vozlišče (in morebitne povezave s tem vozliščem). Zahteva ID vozlišča.

- updateNode — posodobi podatke o vozlišču. Zahteva ID in podatke.
- addLink — doda povezavo. Zahteva ID izvornega in ciljnega vozlišča ter dodatne podatke.
- delLink — izbriše povezavo. Zahteva ID izvornega in ciljnega vozlišča.
- updateLink — posodobi podatke o povezavi. Zahteva ID izvornega in ciljnega vozlišča ter podatke.

6.4.2 Drevo (*graph/tree*)

Drevo je neusmerjen graf, v katerem sta poljubni dve vozlišči povezani z natanko eno enostavno potjo. Gre za povezan graf brez ciklov. Ime drevo si je izmislil matematik Arthur Cayley[2].

Vmesnik zahteva, da vsakemu (razen prvega) vozlišču dodelimo starševsko vozlišče. To je edini način povezovanja vozlišč. Tako vedno dobimo drevo (in tudi vsa možna drevesa). Iz prikaza je razvidno tudi starševsko vozlišče.

- addNode — doda vozlišče. Zahteva ID, ID starševskega vozlišča (null za prvo vozlišče) in dodatne podatke.
- delNode — izbriše vozlišče. Zahteva ID vozlišča.
- updateNode — posodobi podatke o vozlišču. Zahteva ID in podatke.
- reparentNode — spremeni starševsko vozlišče. Zahteva ID vozlišča in ID novega starševskega vozlišča.

6.4.3 2d polje (*graph/grid*)

Modul 2d polje je namenjen prikazovanju grafov v obliki mreže. Za to obliko se ponavadi odločimo, kadar imajo vozlišča lokacije v 2d mreži. Primer takega problema je iskanje poti v prostoru/labirintu.

Vmesnik je preprost, saj ne potrebujemo eksplicitnih povezav.

- set — nastavi podatke za lokacijo x, y. Zahteva koordinati x in y ter dodatke podatke.
- del — izbriše podatke na lokaciji x, y. Zahteva koordinati x in y.

Poglavje 7

Primeri vizualizacij

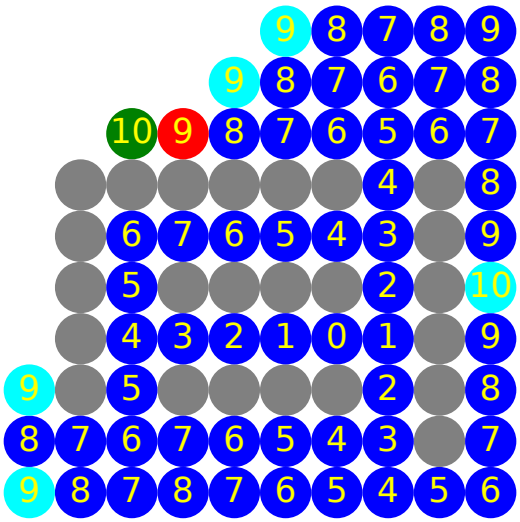
Oglejmo si nekaj celotnih primerov vizualizacij. Za primere smo vzeli naloge iz preteklih računalniških tekmovanj v Sloveniji. Vse primere najdemo na spletni strani[12] in v priloženi izvorni kodi.

7.1 Iskanje v globino in širino

Dober primer uporabe modula 2d polje (poglavje 6.4.3) je iskanje najkrajše poti v mreži, kot je zahtevano v nalogi *Slepi brancin* s tekmovanja *Playoff za IOI 2008*. Naloga zahteva, da v 2d mreži najdemo najkrajšo pot med dvema točkama in se izogibamo danim oviram. Z vsakega polja se lahko premaknemo le na sosednja štiri polja.

Mrežo, po kateri se premikamo, predstavimo z modulom 2d polje. Ovire (polja, po katerih se ne smemo premikati) smo pobarvali s sivo barvo, cilj pa z zeleno (glej sliko 7.1).

Začnimo z implementacijo vizualizacije. Poleg osnovnega ogrodja potrebujemo le instanco modula 2d polje (*grid*) in klic modula za predvajanje dogodkov (*playback*).



Slika 7.1: Vizualizacija algoritma za nalogo Slepi brancin (iskanje v širino).

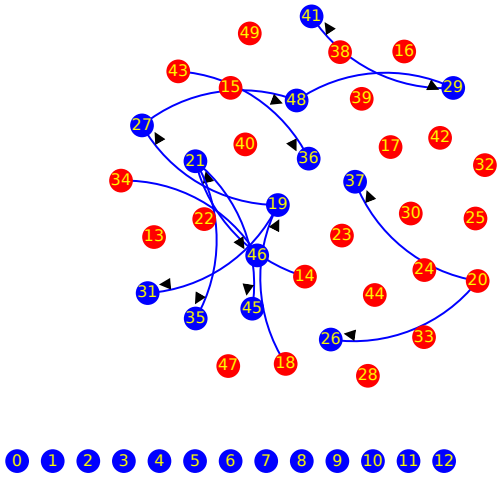
```
var v = {};
v.grid = new grid(svg, new nodeSimple(15, 'value'),
    new linkSimple(), [30, 30]);
playback.load(v);
```

Zajem dogodkov dodamo v naš algoritem (primer je v jeziku C++).

```
// Začne zajem dogodkov.
AVY(start);
// Nastavi x, y na sivo.
AVY(grid.set, P(x), P(y), M(color, P("gray")));
// Naredi premor pri vizualizaciji oz. korak vizualizacije.
AVY(step);
```

Grid se nanaša na lastnost grid našega objekta v v kodi za vizualizacijo, *set* pa je metoda tega razreda (glej poglavje 6.4.3).

Kot smo na začetku prepovedana polja nastavili na sivo, med iskanjem poti nastavljamo barve drugih polj in tako prikazujemo potek iskanja. Ostane



Slika 7.2: Vizualizacija algoritma za nalogo Tekoči trak (topološko urejanje).

nam še izpis številke nad krogom. Če pogledamo kodo vizualizacije zgoraj, smo pri konstruktorju *nodeSimple* kot drugi argument podali *value*. To pomeni, da bo niz znakov za izpis v podatku *value*. Nastavimo ga:

```
AVY(grid.set, P(x), P(y), M(value, P(c)));
```

Zdaj imam vse kar potrebujemo. Če program poženemo z nekimi vhodnimi podatki, bo knjižnica Avy zgenerirala datoteko *0.avy*, ki bo vsebovala vse dogodke. To datoteko skupaj z implementacijo vizualizacije (*avy.js*) naložimo na spletno stran in dobimo končno vizualizacijo.

7.2 Topološko urejanje

Za naslednji primer imamo nalogo Tekoči trak s priprav na olimpijado IOI 2010. Naloga zahteva, da razporedimo stopnje v proizvodnji. Nekatere stopnje zahtevajo, da se pred njimi že končajo druge stopnje. Če vsako stopnjo predstavimo z vozliščem, vsako zahtevo pa z usmerjeno povezavo, dobimo graf, ki ga moramo topološko urediti. Naloga nam zagotavlja, da je rešljiva, kar pomeni, da v grafu ni ciklov.

Ker graf nima ciklov, najprej pomislimo na drevo, a žal ne moremo uporabiti modula za drevo, saj graf ni povezan. Pri našem grafu bi lahko rekli, da gre za gozd (več nepovezanih dreves). Lahko bi naredili modul za gozdove, lahko pa se zadovoljimo z grafom na osnovi sil. Poleg grafa potrebujemo še prikaz rezultata (urejene stopnje proizvodnje). Uporabimo modul za seznam. Primer je ilustriran na sliki 7.2.

```
var v = {};
v.graph = new graph(svg.add(), nodeStyle, linkStyle);
v.list = new list(svg.add(), nodeStyle);
playback.load(v);
```

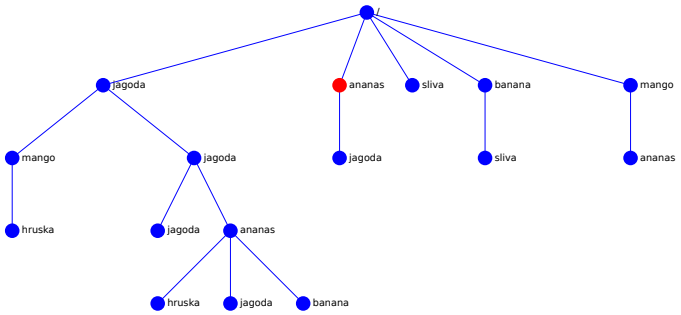
Koda je podobna, kot pri prvem primeru, le da imamo dva modula za prikaz. *Svg* modul poskrbi, da se prikaza ne prekrivata (za vsakega kličemo *svg.add*).

V algoritem moramo dodati zajemanje dogodkov (primer je v jeziku C++).

```
// Doda vozlišče z ID-jem i.
AVY(graph.addNode, P(i));
// Doda povezavo med vozliščema a in b.
AVY(graph.addLink, P(a), P(b));
// Nastavi vozlišče i kot izbrano, kar prikažemo z npr.
// spreminjanjem barve, ki je del modula za prikaz
   vozlišča.
AVY(graph.updateNode, P(i), M(selected, P(true)));
// Izbriše vozlišče t z grafa.
AVY(graph.delNode, P(t));
// Doda vozlišče t na seznam.
AVY(list.add, P(t));
```

7.3 Datotečni sistem (drevo)

Za zadnji primer imamo še nalogo Datotečni sistem z 2. kola tekmovanja UPM 2013. Naloga zahteva simuliranje premikanja po datotečnem sistemu,



Slika 7.3: Vizualizacija datotečnega sistema s pomočjo drevesa.

kot se to dogaja na primer v Linux lupini Bash.

Najprej si zamislimo izgled vizualizacije. Posamezne datoteke in mape si lahko predstavljamo kot vozlišča v drevesu. Datoteke so listi drevesa, mape pa lahko vsebujejo druge mape in datoteke. Mapo, v kateri se trenutno nahajamo, pobarvamo z drugačno barvo.

Vizualizacijo bi lahko implementirali tako, kot vse do sedaj, želimo pa pokazati, da lahko del logike prenesemo iz zajema dogodkov v Javascript implementacijo in tako pridemo do bolj pregledne kode. Namesto da bi pri zajemu dogodkov neposredno klicali modul za drevo, smo naredili poseben vmesnik z metodama *addNode* (doda vozlišče) in *select* (spremeni izbiro trenutnega vozlišča). Če v vizualizacijo dodajamo svoje metode, moramo biti pozorni na premikanje po vizualizaciji nazaj (metoda *reverse*). Pri metodi *select* spremenimo do sedaj izbrano vozlišče

```
v.graph.updateNode(selected, {selected: false});
v.graph.updateNode(id, {selected: true});
selected = id;
```

in vodimo zgodovino sprememb v seznamu *history*.

```
history.push(function(prevSelected) {
  v.graph.__reverse__();
  v.graph.__reverse__();
  selected = prevSelected;
```

```
}.bind(this, selected));
```

V ta seznam zapišemo, kako se dejanje razveljavi (konkretneje, v seznam dodamo funkcijo, ki dejanje razveljavi). V našem primeru moramo dvakrat poklicati *reverse* na modulu za graf (enkrat za vsak klic *updateNode*), ter popraviti spremenljivko, ki hrani trenutno izbrano vozlišče.

Ker smo del dela opravili že pri implementaciji vizualizacije, je zajem dogodkov zdaj zelo preprost. Ob vsaki ustvarjeni datoteki ali mapi kličemo (primer je v jeziku Java):

```
Avy.cmd("self.addNode", n.hashCode(), name, hashCode());
```

Ob spremembi trenutne mape pa:

```
Avy.cmd("self.select", cur.hashCode());
```

Razpršeno kodo (hash code, ki je unikatni identifikator objekta, v C++ bi namesto tega lahko uporabili naslov v pomnilniku) uporabljamo kot ID vozlišča. Namesto tega bi lahko uporabili tudi celotno pot ali kaj podobnega.

Poglavje 8

Sklep

Narejeno je bilo ogrodje za vizualizacijo, kar vključuje orodje za zajemanje dogodkov, Javascript modularno ogrodje, osnovne module za prikaz vizualizacij ter spletno stran za gostovanje vizualizacij in modulov. Narejeni so bili tudi primeri vizualizacij algoritmov z grafi (najkrajša pot, topološko urejanje itd.).

Izdelano ogrodje za vizualizacijo olajša izdelavo vizualizacij in bo, upamo, v prihodnosti povečalo število vizualizacij. Narejeni so bili osnovni moduli za delo z grafi, modularna zasnova pa je zastavljena za vse tipe vizualizacij. Vsaka naslednja vizualizacija olajša izdelavo prihodnjih vizualizacij, zaradi vedno večje zbirke modulov, ki se nabirajo na spletni strani.

Poleg dodatnih modulov obstaja še veliko drugih možnosti za nadgradnjo. Izvajanje programa in izrisovanje vizualizacije bi lahko potekalo vzporedno, namesto da najprej posnamemo dogodke in jih potem predvajamo. To bi lahko povečalo pomen vizualizacij pri razhroščevanju programov, saj bi lahko korak za korak sledili izvajanju. Risalni površini za vizualizacije bi lahko dodali boljše prilagajanje resoluciji zaslona ter možnost za povečavo. Spletni strani bi lahko dodali možnost izvajanja algoritmov, tako da uporabniku ne bi bilo treba izvajati programa na svojem računalniku.

Pri vnovični zasnovi ogrodja bi razmislil o spremembi nalagalnika modulov, saj je lahko ob velikem številu modulov nalaganje počasno. V trenutni

izvedbi za module ne vodimo verzij zato nalagalnik težko izkoristi predpomnjenje (ang. *caching*).

Slike

Literatura

[1] A. Barth (2011), HTTP State Management Mechanism, RFC 6265, http://tools.ietf.org/html/rfc6265 (zadnji dostop 2.12.2013)

[2] A. Cayley (1857), "On the theory of the analytical forms called trees", Philosophical Magazine, 4th series, 13 : 172-176

[3] Magnus Holm, "JSON: The JavaScript subset that isn't", http://timelessrepo.com/json-isnt-a-javascript-subset (zadnji dostop 2.12.2013)

[4] AngularJS, http://angularjs.org/ (zadnji dostop 2.12.2013)

[5] Cross-Site Scripting (XSS) Security Vulnerability in the Sun Java System Access Manager Cross-Domain Controller (CDC), http://download.oracle.com/sunalerts/1020343.1.html (zadnji dostop 2.12.2013)

[6] Dokumentacija metode *postMessage*, http://docs.webplatform.org/wiki/dom/window/postMessage_(window) (zadnji dostop 2.12.2013)

[7] Google AppEngine, https://cloud.google.com/products/app-engine/ (zadnji dostop 2.12.2013)

[8] Google Caja, https://developers.google.com/caja/ (zadnji dostop 2.12.2013)

[9] Knjižnica D3, http://d3js.org/ (zadnji dostop 2.12.2013)

[10] Knjižnica Gson, `https://code.google.com/p/google-gson/` (zadnji dostop 2.12.2013)

[11] Programski jezik Go, `http://golang.org/` (zadnji dostop 2.12.2013)

[12] Spletna stran za vizualizacije, `http://www.algoviz.net` (zadnji dostop 2.12.2013)