SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Vizualizacija algoritma A*

Teon Banek

Voditelj: *Dr. sc. Marko Čupić*

Zagreb, travanj, 2012.

Sadržaj

1. Uvod

Traženje optimalnog puta je problem s kojim se svakodnevno susrećemo. Analiziranjem voznog reda javnog prijevoza pokušavamo procijeniti naš polazak i dolazak na željenu destinaciju. Vožnja gradom i biranje ulica na temelju gustoće prometa ili igranje neke igre na ploči također je susret s tim problemom. Naša svakodnevna rješenja su na intuitivnoj i iskustvenoj bazi te njihova pretvorba u matematički jezik, jezik koji možemo prenijeti računalima, postaje novi problem.

Jedno od prvih matematički definiranih rješenja traženja optimalnog puta u težinskom grafu je Dijkstrin algoritam. Ime je dobio po svom tvorcu, nizozemskom računarskom znanstveniku Edsger W. Dijkstri.

Njegov algoritam je postao osnova raznim modifikacijama. Današnja upotreba u računalnim programima pokriva širok spektar zadataka, poput:

- pretrage puta kroz ulice stvarnog grada,
- snalaženje u prostoru kao dio algoritama umjetne inteligencije te
- pomaganje u komuniciranju među usmjernicima (engl. *router*).

Najpoznatija i najraširenija modifikacija je algoritam A* (''A zvjezdica''). Popularnost je stekao svojom preciznošću i brzinom. Ono što taj algoritam razlikuje od Dijkstrinog jest korištenje heuristike.

Ovaj seminarski rad objašnjava što je to heuristika, kako ona utječe na rad algoritma A* te kako uopće A* funkcionira. Osim teorije opisan je i postupak implementacije A* algoritma u programskom jeziku Java. Kako bi se lakše uvidio način rada algoritma opisana je i izvedba grafičkog prikaza koraka pretrage.

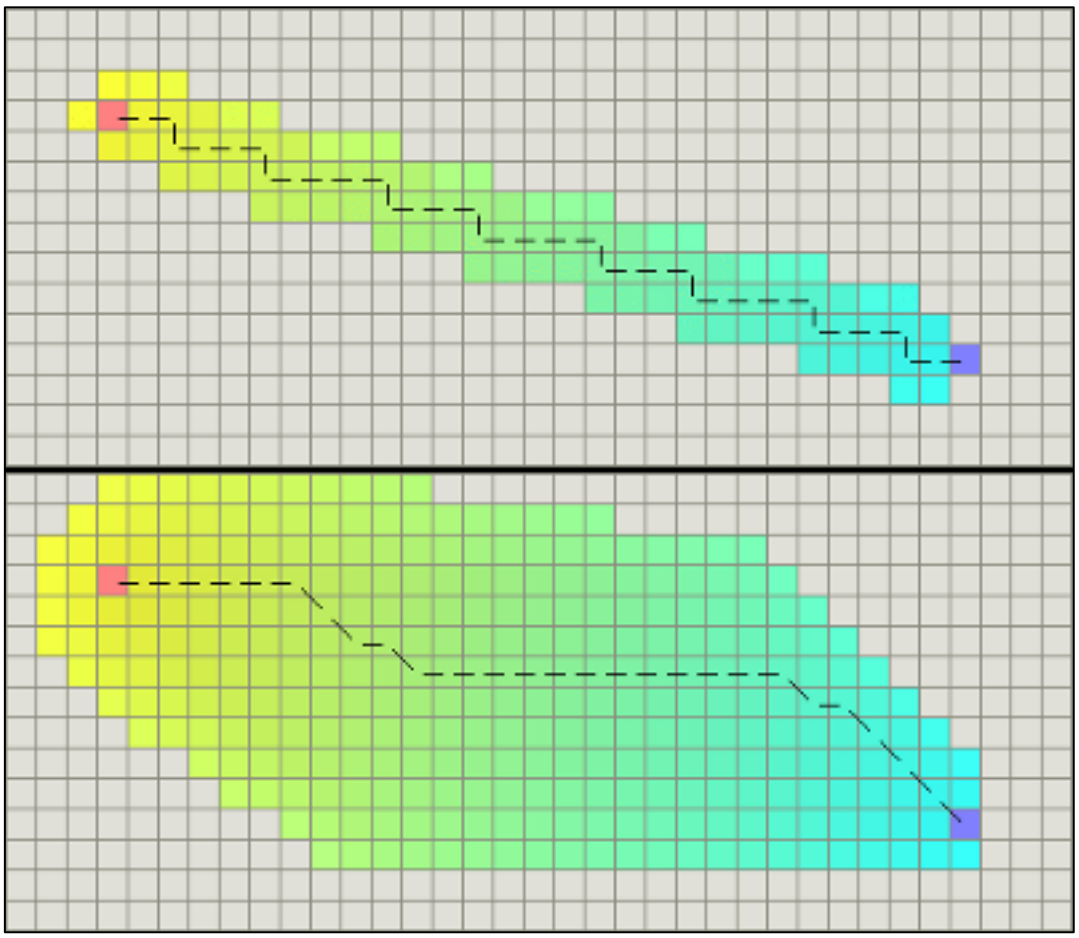
2. Heuristika

Termin heuristika dolazi iz Grčke riječi: "Εὑρίσκω" a u prijevodu bi značila: "pronaći" ili "otkriti". Današnje značenje riječi je drugačije. Heuristika je naziv tehnike rješavanja problema koristeći dostupne informacije i znanja [1].

Osnovni primjer heuristike je pravilo pokušaja i promašaja. U potrazi za rješenjem krećemo od potencijalnih rješenja i sužujemo ih sve do onog konačnog i točnog. Drugi primjer je rješavanje složenijeg problema razlaganjem na poznate jednostavnije oblike. Takav je način učenja u školi, naročito u rješavanju matematičkih zadataka. Nekim ljudima, tzv. 'vizualnim tipovima' je lakše problem riješiti vizualizacijom: bilo fizičkim crtanjem ili misaonim. Sve su to primjeri heuristike iz kojih se vidi njen velik značaj.

U današnje doba provode se iscrpna istraživanja na području robotike i umjetne inteligencije. Jedan od bitnih problema u tim znanstvenim granama jest modeliranje heuristike. Mi ćemo se zadržati na površini jednostavnijih problema koji se mogu bez problema riješiti algoritamskom heuristikom.

Oblikovanje algoritma heuristike kao pomoć u traženju najkraćeg puta kroz graf svodi se na računanje udaljenosti od trenutnog čvora do završnog. Bitno je da izračun udaljenosti nije veći od postojećeg najkraćeg puta, tj. heuristika ne smije precijeniti najkraći put. Takvu heuristiku zovemo optimističnom heuristikom (engl. *admissible/optimistic heuristic*) jer optimistično zamišljamo da postoji najkraći mogući put. Ukoliko heuristika nije optimistična ona može dovesti do krive odluke jer algoritam neće vidjeti postojeći kraći put. U najboljem slučaju takva heuristika može dati optimalno rješenje ali češće daje suboptimalno rješenje. S druge strane, u najgorem slučaju naš algoritam pretrage može zapeti u beskonačnoj petlji između analize dva čvora. Zanimljivost modeliranja heuristike jest u tome što ona utječe na brzinu izvođenja algoritma i izgled puta. Korištenje heuristike koja nije optimistična može povećati brzinu izvođenja algoritma. Vrijedi obrat, pa zato nije dobro uzeti preoptimističnu heuristiku jer može doći do nepotrebnog usporavanja. Osim utjecaja na brzinu, različite formule izračuna utječu na brzinu zaobilaženja prepreka i izgled puta. Rezultat korištenja dviju različitih formula heuristike se može vidjeti na slici 1 [2].



Slika 1. Usporedba heuristika: Manhattan (gornja) i euklidska udaljenost (donja)

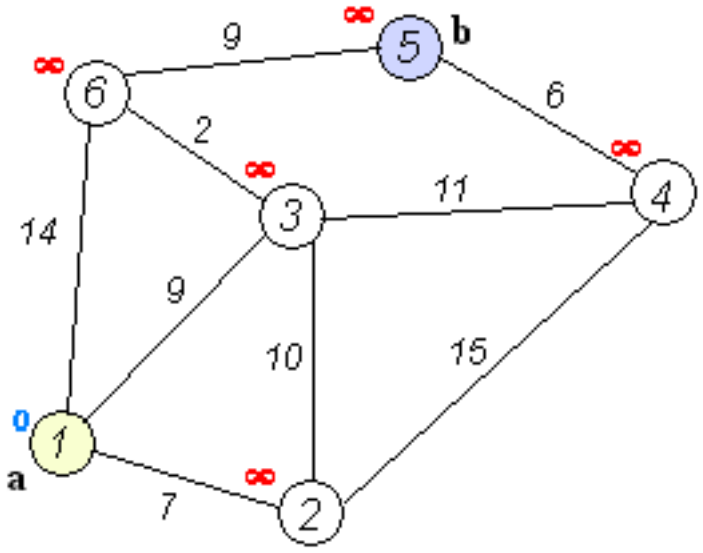
Ponekad nam je u interesu uzeti brži način rješavanja, kada ne možemo ili ne želimo koristiti puno računalnih resursa. Modeliranje kretanja u prirodi postaje bolje ako izmijenimo heuristiku tako da daje glatke putove. Značaj i modeliranje heuristike kako bismo dobili željen rezultat je bitan te u njoj leži fleksibilnost algoritma A*.

3. Implementacija algoritma A* u programskom jeziku Java

Započet ćemo opisom rada algoritma A* i njegovom usporedbom s Dijkstrinim algoritmom. Zatim prelazimo na bitne dijelove programskog ostvarenja algoritma u programskom jeziku Java. U završetku poglavlja proći ćemo kroz realizaciju grafičkog korisničkog sučelja koje će omogućiti korisniku pregled i manipulaciju rada algoritma.

3.1 Usporedba A* i Dijkstrinog algoritma

Oba algoritma započinju svoju pretragu iz danog početnog čvora na grafu (primjer na slici 2). Dijkstrin algoritam formira skupove posjećenih i neposjećenih čvorova [3]. Inicijalno u skupu posjećenih čvorova se nalazi početni čvor a u skupu neposjećenih preostali čvorovi grafa. Također svaki čvor dobiva osnovnu udaljenost beskonačno veliku a udaljenost početnog čvora iznosi 0.



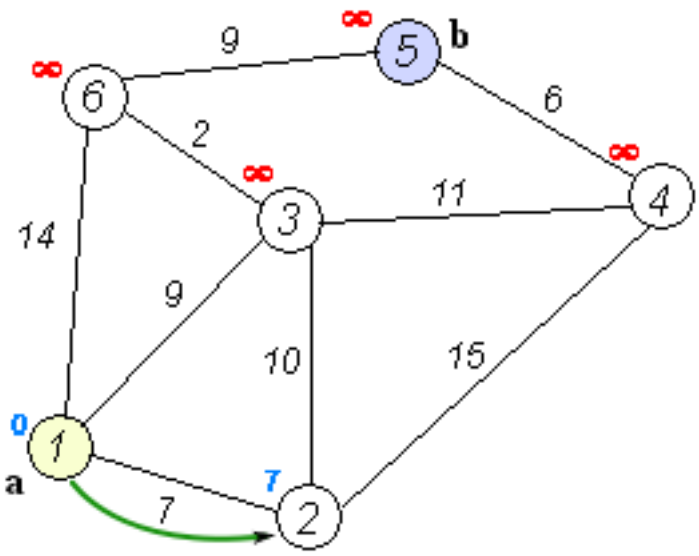
Slika 2. Graf u početnom stanju

Dijkstrin algoritam analizira sve neposjećene čvorove koji su susjedni trenutnom čvoru na kojem se nalazi. Za svaki takav čvor računa se udaljenost zbrajanjem udaljenosti trenutnog čvora i težine brida koji ih spaja (izraz 1).

$$d(x, y) = g(x) + e_{xy} \tag{1}$$

Ukoliko je izračunata udaljenost za čvor manja no što je prethodno zapisano tada se prepisuje udaljenost toga čvora manjom vrijednošću. U početnom stanju svaki susjedni čvor početnom ima beskonačnu udaljenost pa će se ona zamijeniti vrijednošću brida koji ih spaja. Korak nakon početnog stanja se može vidjeti na slici 3. U kasnijem prolasku algoritma postojat će već zapisane konačne vrijednosti te će

prethodno spomenuti princip eliminirati duže putove do neposjećenih čvorova. Kada se izračuna udaljenost svim neposjećenim čvorovima oko trenutnog, trenutni postaje posjećeni čvor te se njegova udaljenost smatra potpuno izračunatom i minimalnom. Sljedeći trenutni čvor postaje susjedni neposjećeni čvor s najmanjom udaljenošću te se algoritam ponavlja. Prestanak rada algoritma predstavlja posjećivanje ciljnog čvora i time je nađen najkraći put. Drugi način prekida algoritma jest nemogućnost biranja neposjećenog čvora s konačnom udaljenosti, tj. najmanja moguća udaljenost je beskonačna. U tom slučaju je Dijkstrin algoritam obišao cijeli graf i nema puta do ostalih čvorova.



Slika 3. Drugi korak

Algoritam A* provodi analizu čvorova na sličan način kao i Dijkstrin. A* formira prazne skupove otvorenih i zatvorenih čvorova [4]. U skup otvorenih čvorova dodaje se početni čvor. Trenutni čvor algoritma postaje čvor s najmanjom cijenom iz skupa otvorenih. Sljedeći korak algoritma računa cijene putova do svih susjeda trenutnog čvora. Zatim se susjedni čvorovi dodaju u listu otvorenih a trenutni čvor prebacuje u listu zatvorenih. Za čvorove u zatvorenoj listi se više ne računa cijena puta. Time je ostvaren prvi prolaz kroz algoritam te odabiranjem novog trenutnog čvora započinje drugi. Algoritam prestaje s radom ukoliko dođe do cilja ili pretraži sve čvorove u grafu i ne nađe put. Glavna razlika A* u odnosu na Dijkstrin algoritam je u izračunu udaljenosti susjednih čvorova. Osim zbrajanja udaljenosti trenutnog čvora i težine brida koji spaja trenutni sa susjednim dodaje se i vrijednost funkcije heuristike (izraz 2).

$$d(x,y) = g(x) + h(x) + e_{xy} \tag{2}$$

Funkcija heuristike je procjena udaljenosti do cilja te ona u izračunu ukupne udaljenosti susjednog čvora daje težinu približavanja ili udaljavanja od ciljnog čvora. Takav princip računanja utječe na biranje trenutnog čvora iz skupa otvorenih jer daje prednost čvorovima koji vode bliže cilju a ne nužno onima koji su najkraće udaljeni od trenutnog čvora. U većini slučajeva algoritam A* pretražuje graf u smjeru od početka prema cilju te to smanjuje ukupnu analizu čvorova.

3.2 Programsko ostvarenje

Kod[1] programa je napisan koristeći razvojno okruženje Eclipse. Drugo popularno okruženje je NetBeans u kojem je moguće grafički realizirati grafičko korisničko sučelje programa. Za razvoj korisničkog sučelja korištena je tehnologija Java Swing [5].

Prvi problem o kojem treba razmisliti jest reprezentacija grafa. Implementacija klasičnog grafa zahtjeva memoriranje veza između vrhova. Prikaz područjima poput zemalja na geografskoj karti ljepše izgleda krajnjem korisniku ali cijene između područja nisu lako vidljive. Efikasno rješenje je dvodimenzionalno polje vrhova. U takvoj konstrukciji grafa nije potrebno memorirati povezanost jer je evidentno da svaki vrh ima 8 susjeda oko sebe. Poseban slučaj su rubni vrhovi no pregled njihovih susjeda je također vrlo jednostavan. Najveću prednost implementacije takvog grafa donosi izračun doprinosa heuristike. Naime, ako graf postavimo u Kartezijev koordinatni sustav lako možemo na razne načine računati udaljenost između dvije točke.

Naš vrh će predstavljati razred `Tile`, opisan sljedećim kodom.

```
public class Tile {

    public static final int WIDTH = 32;
    public static final int HEIGHT = 32;

    private BufferedImage img = null;
    private int x;
    private int y;

    private boolean blocking = false;
    private boolean start = false;
    private boolean goal = false;
    private boolean next = false;
    private boolean closed = false;
    private boolean path = false;

    public JLabel labelF = new JLabel();
```

U njemu je sadržana veličina u pikselima te slika koja ga predstavlja. Mjesto u grafu je zapisano koordinatama (`x`, `y`). `Tile` u sebi ima i neke zastavice vezane uz svoju reprezentaciju ovisno o radu algoritma. `blocking` označava da li se kroz ovaj vrh može proći. Korisniku je omogućeno kreiranje vlastitog grafa manipuliranjem te zastavice. `Start` i `goal` govore da li je ovaj vrh početak ili cilj pretrage. `next`

[1] Svi prikazani kodovi nisu kompletne kopije iz realiziranog programa. Također u ovim primjerima kodova su uklonjeni komentari kako ne bi zauzimali mjesto u dokumentu.

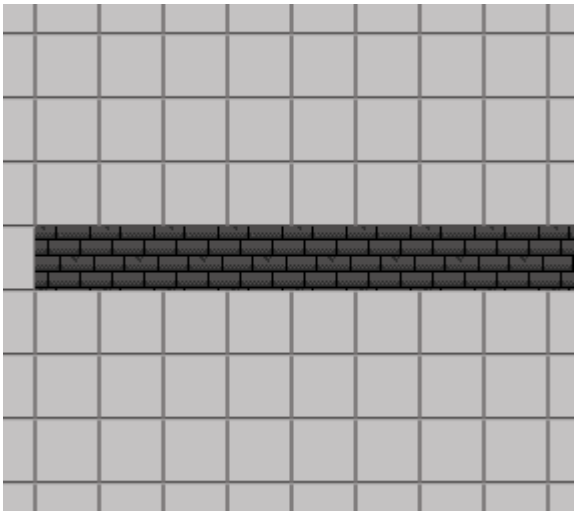
markira da je ovaj vrh sljedeći koji će biti dodan u listu zatvorenih vrhova za što služi zastavica `closed`. Posljednja zastavica `path` označava da ovaj vrh čini optimalan put.

Još jedna varijabla koja se nalazi ovdje je i `labelF` u kojoj je zapisana cijena ovog vrha

```
public Tile(BufferedImage img, int x, int y) {
    super();
    this.img = img;
    this.x = x;
    this.y = y;
}

public void paint(Graphics g) {
    int dx1 = x * WIDTH;
    int dx2 = dx1 + WIDTH;
    int dy1 = y * HEIGHT;
    int dy2 = dy1 + HEIGHT;
    int sx1 = 0;
    int sx2 = WIDTH;
    int sy1 = 0;

    if (start) {
        sy1 = HEIGHT * 2;
    } else if (goal) {
        sy1 = HEIGHT * 3;
    } else if (path) {
        sy1 = HEIGHT * 4;
    } else if (closed) {
        sy1 = HEIGHT * 5;
    } else if (next) {
        sy1 = HEIGHT * 6;
    } else if (blocking) {
        sy1 = HEIGHT;
    }
    int sy2 = sy1 + HEIGHT;

    g.drawImage(img, dx1, dy1, dx2, dy2, sx1, sy1, sx2, sy2, null);
}
```

Konstruktor ovog objekta uzima referencu na sliku koja ga reprezentira te poziciju u grafu.

Funkcijom `paint` se objekt iscrta na grafičko korisničko sučelje. Slika koja ga reprezentira u sebi sadrži više slika što je memorijski efikasnije no svaka slika zasebno. Zastavice služe za određivanje koji dio slike, tj. koja zasebna slika će se iscrtati kao reprezentacija ovog vrha. Ukoliko je postavljen `blocking` iscrtat će se slika zida, ako je vrh u `closed` listi iscrtat će se drugom bojom kako bi ga korisnik lakše uočio itd. Primjeri izgleda se mogu vidjeti na slici 4.



Slika 4. Izgled vrhova u korisničkom sučelju

Graf reprezentira `Tilemap`.

```
public class TileMap extends JPanel {

    private static final long serialVersionUID = 1L;

    private BufferedImage tileset = null;
    private Tile[] tiles = null;
    private int mapWidth;
    private int mapHeight;

    private boolean pickStart = false;
    private boolean pickGoal = false;

    private Tile startTile = null;
    private Tile goalTile = null;
```

Razred nasljeđuje `JPanel` i time postaje komponenta korisničkog sučelja. `TileMap` u sebi sadrži polje `Tile` objekata. Dimenzije ovog grafa se nalaze u varijablama `mapWidth` i `mapHeight`. One su izražene kao broj vrhova a ne broj piksela koje ovaj graf zauzima na ekranu. Dvije zastavice: `pickStart` i `pickGoal` pamte da li je korisnik odabrao početak i cilj na ovom grafu. Također `startTile` i `goalTile` su reference na odabrane početne i ciljne vrhove.

```
public TileMap(int width, int height) {
        setLayout(null);
        this.mapWidth = width;
        this.mapHeight = height;
        tiles = new Tile[width * height];
        setBorder(BorderFactory.createLineBorder(Color.black));

        MouseAdapter mouseAdapter = new TileMapMouseListener(this);
        addMouseListener(mouseAdapter);
        addMouseMotionListener(mouseAdapter);

        try {
            URL url = this.getClass().getResource("/gfx/tileset.png");
            tileset = ImageIO.read(url);
        } catch (IOException e) {
            System.err.println(
                  "Unable to read image resource: ./gfx/tileset.png");
            System.exit(-1);
        }
        createMap();
    }
```

Argument ovog konstruktora je dimenzija grafa. Prilikom stvaranja kreira se prostor za vrhove te se u memoriju učitava slika koju oni koriste. Nasljeđivanjem `JPanel` razreda omogućeno je dodavanje promatrača `MouseListener` za praćenje korisničkih akcija nad ovim grafom. Na kraju se stvaraju `Tile` objekti funkcijom `createMap`.

```
    private void createMap() {
        for (int y = 0; y < mapHeight; y++) {
            for (int x = 0; x < mapWidth; x++) {
                tiles[y * mapWidth + x] = new Tile(tileset, x, y);
                add(tiles[y * mapWidth + x].labelF);
                tiles[y * mapWidth + x].labelF.setBounds(x * Tile.WIDTH, y
                        * Tile.HEIGHT, Tile.WIDTH + 4, Tile.HEIGHT / 2);
            }
        }
    }
```

Polje `tiles` sprema objekte po koordinatama kako bi njihovo dohvaćanje bilo brzo.

Prethodno spomenuti `MouseListener` je razred `TileMapMouseListener`.

```
public class TileMapMouseListener extends MouseAdapter {

    private TileMap tileMap;

    public TileMapMouseListener(TileMap tileMap) {
        this.tileMap = tileMap;
    }

    @Override
    public void mousePressed(MouseEvent e) {
        int mapX = e.getX() / Tile.WIDTH;
        int mapY = e.getY() / Tile.HEIGHT;
        performAction(mapX, mapY, e.getModifiersEx());
    }

    @Override
    public void mouseDragged(MouseEvent e) {
        if (e.getX() >= tileMap.getMapWidth() * Tile.WIDTH || e.getX() < 0
                || e.getY() >= tileMap.getMapHeight() * Tile.HEIGHT
                || e.getY() < 0) {
            return;
        }
        int mapX = e.getX() / Tile.WIDTH;
        int mapY = e.getY() / Tile.HEIGHT;
        performAction(mapX, mapY, e.getModifiersEx());
    }
```

U sebi sadrži referencu na `TileMap` nad kojim očekuje signale korisničkog miša. Ta referenca se postavi u konstruktoru. Nova implementacija funkcija `mousePressed` i `mouseDragged` preračunava koordinate miša u koordinate vrhova nad kojima se odvila akcija. Dobivene koordinate prosljeđuje funkciji `performAction`.

```
    private void performAction(int mapX, int mapY, int button) {
        if (button == MouseEvent.BUTTON1_DOWN_MASK) {
            if (tileMap.isPickStart()) {
                tileMap.setStartTile(mapX, mapY);
            } else if (tileMap.isPickGoal()) {
                tileMap.setGoalTile(mapX, mapY);
            } else {
                blockTile(mapX, mapY, true);
            }
        } else if (button == MouseEvent.BUTTON3_DOWN_MASK) {
            blockTile(mapX, mapY, false);
        }
    }
```

Ova funkcija pogleda koja tipka je pritisnuta na mišu i ovisno o njoj izvede akciju nad grafom. Pritiskom lijeve tipke miša vrh nad kojim se nalazi miš postane početni vrh ili ciljni vrh ako je korisnik definirao da želi postaviti početak ili cilj. Ukoliko nije ništa od toga definirano lijeva tipka postavlja vrh u neprolazno stanje. Pritisak desne tipke miša vrača vrh u prolazno stanje postavljanjem `blocking` na vrijednost `false`.

Kada smo postavili graf i akcije nad njime vrijeme je za implementaciju samog algoritma. Prije same implementacije definirana je enumeracija statusa izvedbe algoritma.

```
public enum PathfinderStatus {

    CONTINUE,
    NO_PATH_FOUND,
    PATH_FOUND
}
```

Ova tri statusa respektivno znače:

- algoritam još nije završio,
- algoritam je završio ali nije pronađen put te
- algoritam je završio i pronađen je optimalan put.

Algoritam je implementiran razredom `Pathfinder` kao zasebna dretva. Time je rasterećena glavna dretva korisničkog sučelja (engl. *Event Dispatching Thread*), što je vrlo poželjno jer omogućuje korisniku gašenje aplikacije ako se algoritam negdje zamrzne. Druga prednost takve implementacije jest mogućnost pokretanja više pretraga odjednom što je značajnije u praktičnim situacijama pa je ovakav kod lako ponovo iskoristiti.

```
public class Pathfinder extends Thread {

    private Set<Tile> closedSet = new LinkedHashSet<>();
    private Set<Tile> openSet = new LinkedHashSet<>();
    private Map<Tile, Tile> cameFrom = new HashMap<Tile, Tile>();

    private Tile start;
    private Tile goal;
    private TileMap tileMap;

    private int[] gScore;
    private int[] hScore;
    private int[] fScore;

    private volatile int delay;

    private boolean moveDiagonally = false;
    private volatile boolean fastForwarding = false;
    private volatile boolean running = true;
    private volatile boolean nextStep = true;
```

Razred sadrži skupove zatvorenih i otvorenih vrhova u kolekcijama `closedSet` i `openSet`. Skup `cameFrom` služi za rekonstrukciju puta jer algoritam u njemu pamti kako je došao do cilja. U razredu se nalaze reference:

- `start` na početni vrh,
- `goal` na ciljni vrh i
- `tileMap` na graf nad kojim se vrši pretraga.

Također sadrži polja izračuna cijena kako neovisno o drugoj dretvi može obaviti računanje.

Varijabla `delay` pamti koliko korisnik želi usporiti izvođenje algoritma u milisekundama.

Definirane su i zastavice za restrikciju i manipulaciju algoritma:

- `moveDiagonally` memorira dopuštene smjerove kretanja,
- `nextStep` pokreće jedan korak algoritma,
- `fastForward` prolazi kroz sve korake algoritma te
- `running` služi za gašenje dretve.

```
public Pathfinder(Tile start, Tile goal, TileMap tileMap,
            boolean moveDiagonally, int delay) {
        super();
        this.start = start;
        this.goal = goal;
        this.tileMap = tileMap;
        this.moveDiagonally = moveDiagonally;
        this.delay = delay;

        gScore = new int[tileMap.getMapWidth() * tileMap.getMapHeight()];
        hScore = new int[tileMap.getMapWidth() * tileMap.getMapHeight()];
        fScore = new int[tileMap.getMapWidth() * tileMap.getMapHeight()];
        for (int i = 0; i < tileMap.getMapHeight() * tileMap.getMapWidth();
                i++) {
            gScore[i] = 0;
            hScore[i] = Integer.MAX_VALUE;
            fScore[i] = Integer.MAX_VALUE;
        }
    }
```

Konstruktor uzima početni i ciljni vrh te graf nad kojim se vrši pretraga. Preostala dva parametra služe za postavljanje kašnjenja i restrikcije kretanja u grafu. Također se postave vrijednosti cijena na maksimalni mogući broj.

```
    @Override
    public void run() {
        while (running) {
            if (fastForwarding || nextStep) {
                if (findPath() != PathfinderStatus.CONTINUE) {
                    running = false;
                }
                nextStep = false;
            }
            try {
                Thread.sleep(100);
            } catch (InterruptedException ignorable) {
            }
        }
    }
```

U metodi `run` se nalazi `while` petlja koja provjerava zastavicom `running` da li se ova dretva još uvijek izvršava. Ako korisnik želi sljedeći korak algoritma ili je ubrzao korake poziva se funkcija `findPath`. Ukoliko ona vrati neki od završnih statusa prekida se izvođenje dretve. Na kraju petlje ostavimo ovu dretvu na 100ms spavanja. Potencijalnu iznimku možemo ignorirati jer nam nije važno da spavanje traje točno 100ms a prekid koji je izazvao iznimku ionako ova dretva nema razloga obraditi.

Ulaskom u sljedeću funkciju započinje algoritam A*:

```
public PathfinderStatus findPath() {
    openSet.add(start);
    int startNode = start.getY()*tileMap.getMapWidth() + start.getX();
    gScore[startNode] = 0;
    hScore[startNode] = calculateHeuristic(start.getX(), start.getY(),
            goal.getX(), goal.getY());
    fScore[startNode] = gScore[startNode] + hScore[startNode];

    do {
        if (openSet.isEmpty()) {
            return PathfinderStatus.NO_PATH_FOUND;
        }
        if (step() == PathfinderStatus.PATH_FOUND) {
            return PathfinderStatus.PATH_FOUND;
        }
        Tile nextTile = findLowestF();
        int fVal = fScore[nextTile.getY() * tileMap.getMapWidth()
                + nextTile.getX()];
        SwingUtilities.invokeLater(new modifyTile(nextTile, tileMap,
                                            true, false, false, fVal));
        if (fastForwarding) {
            try {
                Thread.sleep(delay);
            } catch (InterruptedException ignorable) {
            }
        }
    } while (fastForwarding && running);

    return PathfinderStatus.CONTINUE;
}
```

Prvi korak je dodavanje početnog cilja u listu otvorenih vrhova. U polja `gScore`, `hScore` i `fScore` se respektivno upisuje cijena početnog vrha, vrijednost heuristike početnog vrha te konačna težina početnog vrha. Zatim imamo glavnu petlju algoritma, koja se izvršava dok je dretva živa i dok korisnik želi ubrzati korake. Ukoliko je skup otvorenih vrhova prazan, funkcija vraća da ne postoji put do cilja. Potom ulazimo u funkciju `step` koja prolazi kroz jedan korak algoritma. Ako ona vrati da je našla put izlazimo iz funkcije. Pripremamo se za sljedeći korak algoritma traženjem vrha s najmanjom totalnom cijenom u skupu otvorenih. Predajemo zadatak mijenjanja podataka u grafu glavnoj dretvi funkcijom `SwingUtilities.invokeLater`. Pred kraj petlje algoritam se pauzira na vrijeme koje je odredio korisnik.

```
private PathfinderStatus step() {
    Tile current = findLowestF();
    if (current == goal) {
        reconstructPath(current);
        return PathfinderStatus.PATH_FOUND;
    }

    openSet.remove(current);
    closedSet.add(current);

    SwingUtilities.invokeLater(new modifyTile(current, tileMap, false,
            true, false, fScore[current.getY() * tileMap.getMapWidth()
                    + current.getX()]));

    checkNeighbors(getNeighbors(current), current);

    return PathfinderStatus.CONTINUE;
}
```

Metoda `step` iz skupa otvorenih čvorova pronalazi onaj s najnižom cijenom puta te ga postavlja kao trenutni čvor. Ako taj čvor odgovara cilju, algoritam je pronašao put koji se rekonstruira metodom `reconstructPath`. Rekonstrukcijom puta završava rad algoritma. Ukoliko još nismo stigli do cilja, trenutni čvor se prebacuje u skup zatvorenih. Promjenu dojavljujemo dretvi koja mijenja grafičko korisničko sučelje kako bi vizualizirali rad algoritma. Zatim izračunamo cijene vrhova susjednih trenutnom metodom u nastavku.

```
private void checkNeighbors(Set<Tile> neighbors, Tile current) {
    for (Tile n : neighbors) {
        if (closedSet.contains(n) || n.equals(current)) {
            continue;
        }
        int nIndex = n.getY() * tileMap.getMapWidth() + n.getX();
        int curIndex = current.getY() * tileMap.getMapWidth()
                + current.getX();
        int tentativeG = gScore[curIndex] + costBetween(current, n);
        boolean tentativeBetter = false;
        if (openSet.contains(n) == false) {
            openSet.add(n);
            hScore[nIndex] = calculateHeuristic(n.getX(), n.getY(),
                    goal.getX(), goal.getY());
            tentativeBetter = true;
        } else if (tentativeG < gScore[nIndex]) {
            tentativeBetter = true;
        }
        if (tentativeBetter) {
            cameFrom.put(n, current);
            gScore[nIndex] = tentativeG;
            fScore[nIndex] = gScore[nIndex] + hScore[nIndex];
            SwingUtilities.invokeLater(new modifyTile(n, tileMap, false
                    false, false, fScore[nIndex]));
        }
    }
}
```

Za svakog susjeda provjerimo ako je u listi zatvorenih te ga preskačemo jer to znači da je za njega već izračunat minimalan put. Ako nije u listi zatvorenih, izračunamo cijenu puta do njega kroz trenutni čvor. Ukoliko je dobiveni izračun manji od prethodno izračunatog tada za taj čvor upisujemo novu minimalnu cijenu. Također

ako taj čvor nije bio ni u listi zatvorenih ni otvorenih, dodaje se u listu otvorenih čvorova.

Preostao je još jedan bitan dio izvedbe algoritma opisan sljedećim kodom.

```
    private int calculateHeuristic(int x1, int y1, int x2, int y2) {
        // Manhattan distance multiplied by lowest cost of moving to a tile
        int h = (Math.abs(x2 - x1) + Math.abs(y2 - y1)) * 10;
        return h;
    }
```

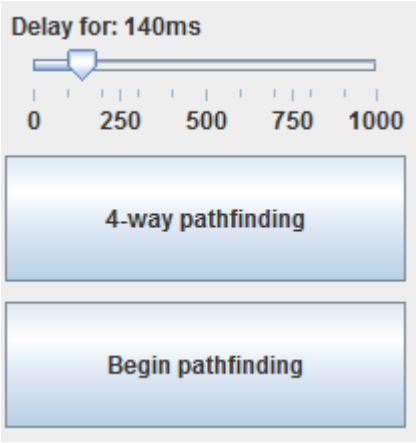
Izračun heuristike temelji se na Manhattan udaljenosti koju množimo najmanjom cijenom pomaka. Manhattan udaljenost prestaje biti optimistična heuristika tj. postaje pesimistična ako je dozvoljeno dijagonalno kretanje.

Postavili smo temelje algoritma i grafa nad kojim algoritam djeluje, te je došao red na kompletiranje programa.

Glavni program čini razred `App` koji nasljeđuje `JFrame`.

```
public class App extends JFrame {

    private static final long serialVersionUID = 1L;

    private JButton toggleDirectionsButton = new JButton("4-way
      pathfinding");
    private SliderLabel delayLabel = new SliderLabel();

    private TileMap tileMap = null;

    private Pathfinder aStar = null;

    public App() {

        setResizable(false);
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setTitle("A* pathfinding V1.0");

        initGui();
    }
```

Korisniku ćemo ponuditi gumb za izmjenu mogućih smjerova kretanja kao indirektan način pretvorbe optimistične heuristike u pesimističnu te obratno. Njegov naziv će se sukladno tome izmjenjivati pa je referenca dio razreda. `SliderLabel` će dinamički mijenjati svoj tekst ovisno o postavkama kašnjenja algoritma (slika 5).



Razred sadrži i reference na graf te dretvu koja primjenjuje algoritam A* na dan graf.

Konstruktor postavlja osnovne vrijednosti prozora aplikacije poput naslova.

Pozivom metode `initGUI` se pozicionira preostali dio grafičkog korisničkog sučelja.

Slika 5. Dio sučelja

```
private void initGui() {
    JButton resetButton = new JButton("Reset");
    JButton startButton = new JButton("Pick start");
    JButton goalButton = new JButton("Pick goal");
    JButton startAlgorithmButton = new JButton("Begin pathfinding");
    JButton nextStepButton = new JButton("Next step");
    JButton toggleSkipButton = new JButton("Skip steps");

    ActionListener bal = new ButtonActionListener();

    resetButton.setActionCommand("reset");
    resetButton.addActionListener(bal);
    ...
    toggleDirectionsButton.setActionCommand("directions");
    toggleDirectionsButton.addActionListener(bal);

    JSlider delaySlider = new JSlider(JSlider.HORIZONTAL, 0, 1000, 0);
    delaySlider.addChangeListener(delayLabel);

    JPanel panel = new JPanel();
    tileMap = new TileMap(20, 20);
    panel.add(tileMap);

    addWindowListener(new WindowListener() {

        @Override
        public void windowClosed(WindowEvent e) {
            if (aStar != null) {
                aStar.setRunning(false);
            }
        }
    });
}
```

Korisniku će gumbi ponuditi sljedeću manipulaciju grafom:

- resetiranje na početne postavke,
- postavljanje početnog vrha i
- postavljanje ciljnog vrha.

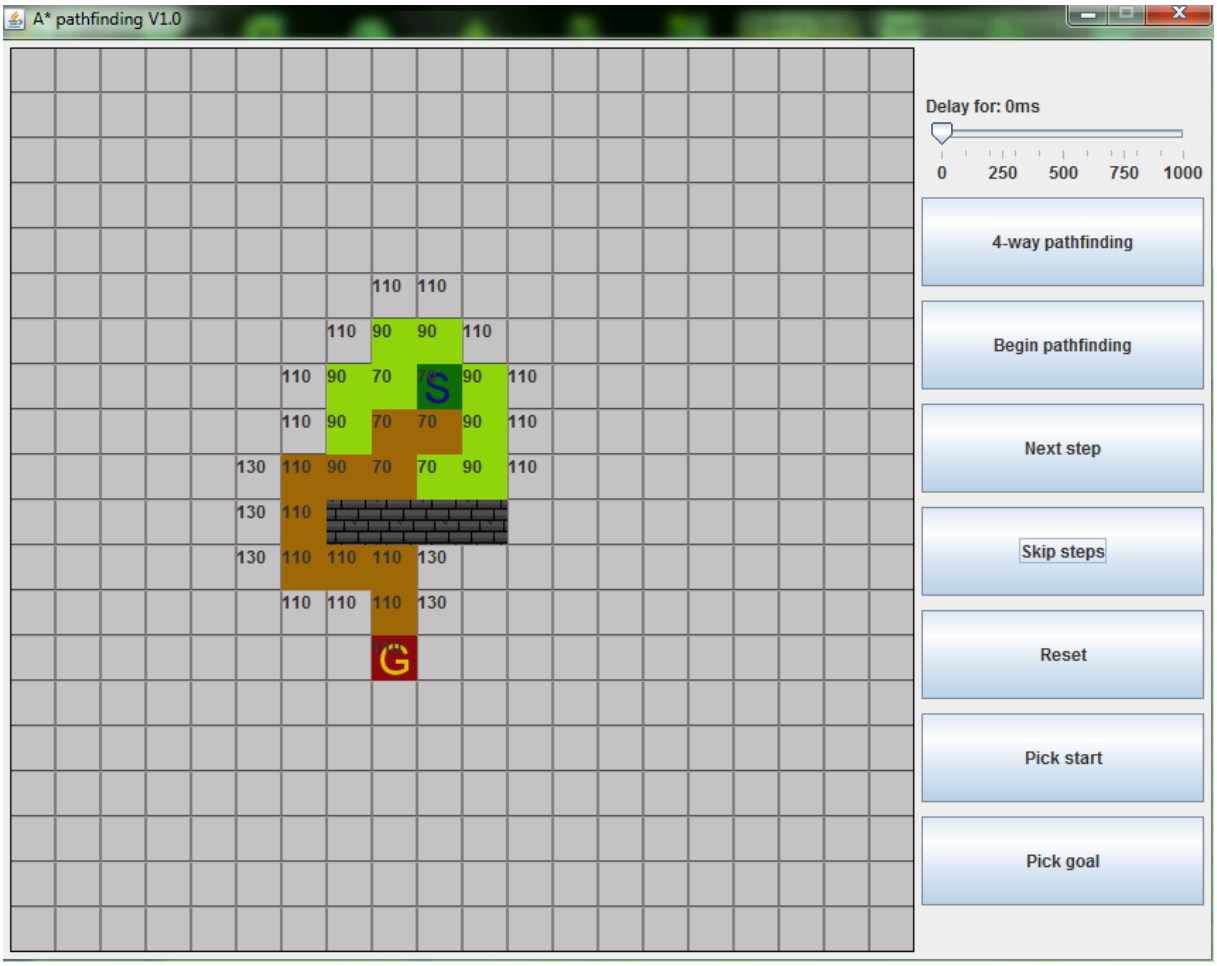
Manipulaciju algoritmom osim spomenute izmjene smjerova kretanja nude gumbi:

- započinjanje algoritma,
- pomak na sljedeći korak algoritma te
- prolaz kroz sve korake.

Na navedene gumbe prijavimo `ActionListener` koji će napraviti željene izmjene korisnika. Modifikaciju vremenskog kašnjenja algoritma nudi `JSlider` komponenta na koju se registrira `delayLabel` kako bi izmjenjivala svoj tekst ovisno o trenutno postavljenom kašnjenju. Sve te komponente i naš graf dodamo u panel kojeg zatim stavimo u sadržaj našeg prozora. Potrebno je namjestiti pozicije komponenti u grafičkom korisničkom sučelju, no to ovdje nije prikazano. Prije samog kraja metode definiramo novi `WindowListener` kako bi pravilno zatvorili dretve prije zatvaranja aplikacije.

```
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                App app = new App();
                app.pack();
                app.setVisible(true);
            }
        });
    }
```

Metoda `main` pokreće aplikaciju predavanjem zadatka kreacije prozora dretvi zaduženoj za grafičko korisničko sučelje. Konačan rezultat može se vidjeti na slici 6.



Slika 6. Realizirana aplikacija

4. Zaključak

Programski jezik Java uvelike olakšava implementaciju algoritma A* svojim gotovim programskim strukturama. Osim samih programskih struktura, pri implementaciji nije potrebno paziti na oslobađanje memorijskih resursa jer Java virtualni stroj (engl. *Java Virtual Machine*) ima mehanizam automatskog upravljanja memorijom (engl. *garbage collection*). Prednost koju Java postiže nad višim programskim jezicima poput jezika Python jest njena brzina. Konstantnim unaprjeđenjima programskog jezika Java i virtualnog stroja brzine izvođenja postaju bolje te su danas sumjerljive izvođenju programskog jezika C++.

Vizualizacija algoritma je izvedena korištenjem tehnologije Java Swing za izradu grafičkog korisničkog sučelja. Izgrađeno sučelje se dobro prilagođava različitim operacijskim sustavima i lijepo izgleda. Korištenje biblioteke Swing zahtjeva od programskog inženjera više upoznavanja no što je poželjno te ponekad nije lako dobiti željeni izgled.

Java virtualni stroj preuzima odgovornost prilagođavanja programa operacijskom sustavu na kojem se izvršava. Time je omogućena lakša prenosivost i izrada aplikacija.

5. Literatura

1. Heuristic, http://en.wikipedia.org/wiki/Heuristic,
   posjećeno 2. travnja 2012.
2. Amit's A* Pages, http://theory.stanford.edu/~amitp/GameProgramming/,
   posjećeno 6. svibnja 2012.
3. Dijkstra's algorithm, http://en.wikipedia.org/wiki/Dijkstra's_algorithm,
   posjećeno 2. travnja 2012.
4. A* search algorithm, http://en.wikipedia.org/wiki/A*_search_algorithm,
   posjećeno 2. travnja 2012.
5. Graphical User Interfaces, http://docs.oracle.com/javase/tutorial/ui/index.html,
   posjećeno 31. svibnja 2012.

6. Sažetak

Tema ovog rada je implementacija i vizualni prikaz rada algoritma A* u programskom jeziku Java.

Algoritam A* je modifikacija Dijkstrinog algoritma pretrage najkraćeg puta u težinskom grafu. Spomenuta modifikacija je korištenje heuristike. Osnova je princip rada Dijkstrinog algoritma ali primjenom heuristike algoritam A* uzima u obzir i udaljenost do cilja. Takav pristup je u većini slučajeva efikasniji te omogućuje pronalaženje puta u beskonačnim grafovima.

U radu je opisan način rada algoritma te utjecaj heuristike na sam algoritam. Korištenje određenih heuristika može dovesti do biranja neoptimalnog puta, no također ista ta heuristika u drugoj situaciji može dovesti do željenih rezultata. Uz teoriju algoritma rad pokriva isječke bitnog dijela implementacije u programskom jeziku Java.

Vizualizacija je ostvarena grafičkim prikazom koraka algoritma i usporedbom dvije različite heuristike. Osim objašnjenja algoritma A*, sagledavaju se prednosti i mane implementacije algoritma u programskom jeziku Java.