8. domaća zadaća

Zadaća se sastoji od dva zadatka.

Zadatak 1.

U ovom zadatku dorađujemo ljusku koju ste pripremili u šestoj domaćoj zadaći. Napravite novi Maven projekt te u njega prekopirajte implementaciju ljuske s naredbama koje ste razvili u okviru šeste domaće zadaće (ali samo to; nemojte kopirati onaj dio vezan uz kriptografiju). U tim paketima ćete imati "hw06" - neka tako ostane. Samo sve skupa prekopirajte.

Zagrijavanje. Proširite sada sučelje `Environment` sljedećim metodama:

```
Path getCurrentDirectory();
void setCurrentDirectory(Path path);
Object getSharedData(String key);
void setSharedData(String key, Object value);
```

Po pokretanju programa, poziv `getCurrentDirectory()` treba vraćati apsolutnu normaliziranu stazu koja odgovara trenutnom direktoriju pokrenutog java procesa (tj. tražite da se `"."` prebaci u apsolutnu stazu pa normalizira). `setCurrentDirectory(...)` omogućava da se kao trenutni direktorij koji će koristiti Vaša ljuska koristi zadani direktorij, ako isti postoji; u suprotnom pokušaj postavljanja takvog direktorija treba baciti iznimku. Implementacija sučelja `Environment` trenutni direktorij pamti kao jednu člansku varijablu (iz Jave ne možete doista mijenjati trenutni direktorij procesa).

Metode `getSharedData(...)` i `setSharedData(...)` omogućavaju da naredbe dijele/pamte određene podatke. Implementacija sučelja `Environment` za ovo koristi mapu. `getSharedData(...)`, ako se traži nepostojeći ključ, vraća `null`. Više naredbi koje dijele podatke ne smiju ih dijeliti kroz statičke varijable, već isključivo pohranom u ovu mapu.

Dodajte naredbe:

`pwd` - bez argumenata, printa u terminal trenutni direktorij kako je zapisan u `Environment` (ispis mora biti apsolutna staza jer se takva i čuva u `Environment`u)

`cd STAZA` - prima jedan argument (`STAZA`): novi trenutni direktorij i njega zapisuje u `Environment`.

Argument naredbe `cd` može biti relativna staza koja se razriješava s obzirom na aktualni trenutni direktorij kakav je zapisan u `Environment`u. Za ovo koristite metodu `resolve` koju nudi razred `Path`. Nemojte sami pisati metode za razrješavanje staza!

Prođite sada kroz sve vaše naredbe koje ste implementirali u prethodnoj zadaći, i doradite ih tako da svaki argument koji je bio staza sada bude razriješen s obzirom na trenutni direktorij zapisan u `Environment`u (čime će apsolutne staze ostati apsolutne, a relativne će se korektno razriješiti).

Dodajte naredbe:

`pushd STAZA` - naredba trenutni direktorij gura na stog i potom kao trenutni direktorij postavlja onaj zadan jedinim argumentom (`STAZA`). Naredba u *dijeljenim podatcima* pod ključem "`cdstack`" stvara stog (ako isti već ne postoji) i tamo zapisuje trenutni direktorij prije no što ga promijeni. Ako `STAZA` ne predstavlja postojeći direktorij, naredba ispisuje pogrešku i ne modificira stog niti trenutni direktorij).

`popd` - naredba je bez argumenata, skida sa stoga vršnu stazu i nju postavlja kao trenutni direktorij (ako takav postoji - primjerice, moguće da je u međuvremenu obrisan; u tom slučaju staza se ipak miče sa stoga ali se trenutni direktorij ne mijenja). Ako je stog prazan, naredba javlja pogrešku.

`listd` - naredba ispisuje u terminal sve staze koje su na stogu počev od one koja je posljednje dodana; pod "ispisuje" podrazumijeva se ispis same staze, ne sadržaj direktorija ili slično. Ako su na stogu tri staze, ispis će imati tri retka. Ako je stog prazan, ispisuje se "`Nema pohranjenih direktorija.`".

`dropd` - naredba sa stoga skida vršni direktorij (i odbacuje ga); trenutni direktorij se ne mijenja. Ako je stog prazan, naredba javlja pogrešku.

A sad zahtjevniji dio. Dodajte naredbu:

```
massrename DIR1 DIR2 CMD MASKA ostalo
```

Naredba služi masovnom preimenovanju/premještanju datoteka (ne direktorija!) koji su izravno u direktoriju `DIR1`. Datoteke će biti premještene u `DIR2` (koji može biti isti kao i `DIR1`). `MASKA` je regularni izraz napisan u skladu sa sintaksom podržanom razredom `Pattern` (https://docs.oracle.com/javase/9/docs/api/java/util/regex/Pattern.html) a koji selektira datoteke iz `DIR1` na koje će se uopće primijeniti postupak preimenovanja/premještanja. Prilikom uporabe regularnih izraza uvijek treba raditi uz postavljene zastavice `UNICODE_CASE` i `CASE_INSENSITIVE`.

Kako je ovaj zadatak dosta "opasan", naredba podržava nekoliko podnaredbi određenih s CMD. Za potrebe ilustracije, neka je u direktoriju `DIR1` smješteno sljedeće:

```
slika1-zagreb.jpg
slika2-zagreb.jpg
slika3-zagreb.jpg
slika4-zagreb.jpg
slika1-zadar.jpg
slika2-zadar.jpg
slika3-zadar.jpg
slika4-zadar.jpg
ljeto-2018-slika1.jpg
ljeto-2018-slika2.jpg
ljeto-2018-slika3.jpg
ljeto-2018-slika4.jpg
```

Ako je `CMD` jednak `filter`, naredba treba ispisati imena datoteka koje su selektirane maskom.

Primjerice:

```
massrename DIR1 DIR2 filter slika\d+-[^.]+\.jpg
slika1-zagreb.jpg
slika2-zagreb.jpg
slika3-zagreb.jpg
slika4-zagreb.jpg
slika1-zadar.jpg
slika2-zadar.jpg
slika3-zadar.jpg
slika4-zadar.jpg
```

(pri čemu redoslijed u ispisu nije bitan i može ovisiti o načinu na koji dohvaćate datoteke direktorija `DIR1`). Primijetite, `MASKU` ili pišemo pod navodnicima (pa joj pripada sve do sljedećih navodnika), ili je pišemo bez navodnika pa joj pripada sve do prvog razmaka/taba/... (ako je pod navodnicima, moramo paziti na dogovorene *escapeove*; izvan navodnika nema *escapeova*). S obzirom na dani primjer, potpuno jednako ponašanje ćemo dobiti i s:

```
massrename DIR1 DIR2 filter "slika\d+-[^.]+\.jpg"
slika1-zagreb.jpg
slika2-zagreb.jpg
slika3-zagreb.jpg
slika4-zagreb.jpg
slika1-zadar.jpg
slika2-zadar.jpg
slika3-zadar.jpg
slika4-zadar.jpg
```

kao i uz četvrti argument oblika `"slika\\d+-[^.]+\\.jpg"` (s obzirom na dogovorene *escapeove* koje koristimo pri parsiranju u ljusci). U sva tri slučaja, ako je:

```
String s = četvrtiArgumentIzGornjegPrimjera();
```

tada je:

```
s.charAt(0) = (char)115;  // slovo s
s.charAt(1) = (char)108;  // slovo l
s.charAt(2) = (char)105;  // slovo i
s.charAt(3) = (char)107;  // slovo k
s.charAt(4) = (char)97;  // slovo a
s.charAt(5) = (char)92;  // znak \
s.charAt(6) = (char)100;  // slovo d
...
```

Za filtriranje ćemo koristiti javine standardne razrede `Pattern` i `Matcher`. Prije no što krenete ovo implementirati, pogledajte još barem sljedeću podnaredbu i opis implementacije.

Za izoliranje dijelova imena koristit ćemo mogućnost grupiranja koju nudi `Pattern`. Podnaredba `groups` treba ispisati sve grupe za sve selektirane datoteke:

```
massrename DIR1 DIR2 groups slika(\d+)-([^.]+)\.jpg
slika1-zagreb.jpg 0: slika1-zagreb.jpg 1: 1 2: zagreb
slika2-zagreb.jpg 0: slika2-zagreb.jpg 1: 2 2: zagreb
slika3-zagreb.jpg 0: slika3-zagreb.jpg 1: 3 2: zagreb
slika4-zagreb.jpg 0: slika4-zagreb.jpg 1: 4 2: zagreb
slika1-zadar.jpg 0: slika1-zadar.jpg 1: 1 2: zadar
slika2-zadar.jpg 0: slika2-zadar.jpg 1: 2 2: zadar
slika3-zadar.jpg 0: slika3-zadar.jpg 1: 3 2: zadar
slika4-zadar.jpg 0: slika4-zadar.jpg 1: 4 2: zadar
```

Kako u maski imamo dvije zagrade - definirane su dvije grupe (grupa 1 i grupa 2) te implicitna grupa 0; stoga iza imena svake datoteke imamo za grupe 0, 1 i 2 prikazano na što su se mapirale.

Kako biste podržali prethodne dvije podnaredbe, napišite razred `FilterResult` čiji primjerci predstavljaju po jednu odabranu datoteku zajedno s informacijama o broju pronađenih grupa te metodom za dohvat grupe. Razred treba ponuditi sljedeće javne metode:

```
String toString();         // vraća ime datoteke (bez staze)
int numberOfGroups();      // koliko je grupa pronađeno
String group(int index); // dohvat tražene grupe:
                           // vrijedi 0 <= index <= numberOfGroups()
```

Kroz konstruktor pošaljite sve relevantno da biste mogli dohvaćati tražene podatke.

Sada napišite privatnu statičku metodu:

```
List<FilterResult> filter(Path dir, String pattern) throws IOException;
```

koja trči po predanom direktoriju i vraća listu svih datoteka čije je ime zadovoljavalo predani uzorak.

Podnaredbe `filter` i `groups` sada možete riješiti oslanjajući se na napisanu metodu.

Ako je podnaredba `show`, tada naredba prima još jedan argument: `IZRAZ` koji definira kako se generira novo ime. Naredba ispisuje selektirana imena i nova imena. Primjer je dan u nastavku.

```
massrename DIR1 DIR2 show slika(\d+)-([^.]+)\.jpg gradovi-${2}-${1,03}.jpg
slika1-zagreb.jpg => gradovi-zagreb-001.jpg
slika2-zagreb.jpg => gradovi-zagreb-002.jpg
slika3-zagreb.jpg => gradovi-zagreb-003.jpg
slika4-zagreb.jpg => gradovi-zagreb-004.jpg
slika1-zadar.jpg => gradovi-zadar-001.jpg
slika2-zadar.jpg => gradovi-zadar-002.jpg
slika3-zadar.jpg => gradovi-zadar-003.jpg
slika4-zadar.jpg => gradovi-zadar-004.jpg
```

`IZRAZ` može biti ili nešto pod navodnicima (pa se mogu pojavljivati praznine) ili kompaktan niz znakova (do prvog razmaka/taba/...). Izraz može sadržavati supstitucijske naredbe koje su oblika `${brojGrupe}` ili `${brojGrupe,dodatnoPojašnjenje}`. Ako je supstitucijska naredba oblika `${brojGrupe}`, ona "sebe" zamijenjuje nizom koji je mapiran na zadanu grupu. Prilikom parsiranja izraza, obratite pažnju da ovo mora biti cijeli nenegativan broj (više od toga u trenutku

parsiranja nećemo znati) pa ako nešto ne štima, javite pogrešku. Ako je supstitucijska naredba oblika `${brojGrupe,dodatnoPojašnjenje}`, tada dodatno pojašnjenje mora biti broj ili nula broj (pri čemu broj može biti višeznamenkasti). Sam broj određuje koliko će minimalno znakova biti "emitirano" prilikom zapisivanja tražene grupe; npr. `"${1,3}"` bi značilo da se zapiše grupa 1, minimalno na tri znaka širine; ako je grupa 1 dulja od toga, zapisuje se čitava; ako je kraća, naprije se ispisuje potreban broj praznina (`SPACE`) a potom grupa, tako da je ukupan broj znakova tada jednak 3. `"${1,03}"` definira da se umjesto praznina nadopune rade znakom 0. Specifični primjeri prethodnih pravila: `"${1,0}"` bi značilo da je minimalna širina 0 (a nadopuna razmacima) dok bi `"${1,00}"` značilo također da je minimalna širina 0 i nadopuna nulama. Implementacijski, niti u jednom od ova dva slučaja se ne radi nadopuna, ali izrazi su sintaksno ispravni.

Konačno, podnaredba `execute` će napraviti zadano preimenovanje/premještanje. Koristite `Files#move` za provedbu.

```
massrename DIR1 DIR2 execute slika(\d+)-([^.]+)\.jpg gradovi-${2}-${1,03}.jpg
DIR1/slika1-zagreb.jpg => DIR2/gradovi-zagreb-001.jpg
DIR1/slika2-zagreb.jpg => DIR2/gradovi-zagreb-002.jpg
DIR1/slika3-zagreb.jpg => DIR2/gradovi-zagreb-003.jpg
DIR1/slika4-zagreb.jpg => DIR2/gradovi-zagreb-004.jpg
DIR1/slika1-zadar.jpg => DIR2/gradovi-zadar-001.jpg
DIR1/slika2-zadar.jpg => DIR2/gradovi-zadar-002.jpg
DIR1/slika3-zadar.jpg => DIR2/gradovi-zadar-003.jpg
DIR1/slika4-zadar.jpg => DIR2/gradovi-zadar-004.jpg
```

Naredba `massrename` svakim pokretanjem obavlja sve relevantne korake ispočetka te nigdje ništa ne pamti. Tako podnaredba `groups` najprije obavlja filtriranje, a potom za sve selektirane datoteke ispisuje mapirane grupe.

Implementacijski naputak. Za izvedbu generiranja imena definirajte sučelja:

```
NameBuilder
  void execute(FilterResult result, StringBuilder sb)
```

Objekti tipa `NameBuilder` generiraju dijelove imena zapisivanjem u `StringBuilder` koji dobiju preko argumenta u metodi `execute`. Napravite razred `NameBuilderParser` koji kroz konstruktor dobiva `IZRAZ`, parsira ga i vraća jedan `NameBuilder` objekt:

```
NameBuilderParser:
  public NameBuilderParser(String izraz);
  public NameBuilder getNameBuilder();
  private ... vaše ostale potrebne metode ...
```

Pogledajmo primjer:

```
NameBuilderParser parser =
           new NameBuilderParser("gradovi-${2}-${1,03}.jpg");
NameBuilder builder = parser.getNameBuilder();
```

Parser će na temelju predanog izraza napraviti:

- jedan objekt tipa `NameBuilder` koji će u metodi `execute` u predani `StringBuilder` zapisati `"gradovi-"`
- jedan objekt tipa `NameBuilder` koji će u metodi `execute` u predani `StringBuilder` zapisati na što god je postavljena grupa 2

- jedan objekt tipa `NameBuilder` koji će u metodi `execute` u predani `StringBuilder` zapisati `"-"`
- jedan objekt tipa `NameBuilder` koji će u metodi `execute` u predani `StringBuilder` zapisati na što god je postavljena grupa 1, na minimalno tri znaka širine uz dopunu nulama
- jedan objekt tipa `NameBuilder` koji će u metodi `execute` u predani `StringBuilder` zapisati `".jpg"`
- jedan objekt tipa `NameBuilder` koji će imati reference na ove prethodno stvorene `NameBuilder`e i koji će u metodi `execute` redom nad svima njima pozvati `execute` (općeniti Kompozit); alternativno, kao kod komparatora možete u sučelje dodati metodu `"then"` koja će omogućiti izgradnju binarnih kompozita.

Poziv `parser.getNameBuilder()` će vratiti upravo referencu na ovaj posljednji objekt. Primijetite da sve u svemu imate tri različite vrste `NameBuilder` objekata (čitaj: tri konkretna razreda): jedan koji uvijek upisuje konstantan string koji mora primiti kroz konstruktor, jedan koji uvijek zapisuje zadanu grupu uz eventualno zadanu minimalnu širinu (podatke prima kroz konstruktor) te jedan koji kroz konstruktor prima reference na niz drugih i u svojoj `execute` poziva njihove `execute`. Umjesto da radite ove razrede kao imenovane, mogli biste pripremiti niz statičkih metoda koji ih grade kao lambde na temelju predanih argumenata; nešto poput:

```
static NameBuilder text(String t) { return (...) -> {...} }
static NameBuilder group(int index) { return (...) -> {...} }
static NameBuilder group(int index, char padding, int minWidth) { return
(...) -> {...} }
```

Uz ovakve metode dobivate vrlo praktičan API za izgradnju koji možete koristiti i u parseru, a primjer uporabe bi bio:

```
NameBuilder nb = text("gradovi-").group(2).text("-").group(1, '0', 3);
```

Vezano za zadavanje supstitucijskih naredbi u `IZRAZU` (npr. `${1,03}`), pravila su sljedeća. Naredba započinje s `${` (nema razmaka između; `$ {` ne započinje supstitucijsku naredbu). Jednom kad je naredba započela, pripada joj sve do `}`. Unutra mogu biti proizvoljni razmaci (ali ne između znamenaka broja); npr. `${1,03}`; `${ 1 , 03 }`; `${1 , 03}`; `${ 1 }`; ako unutar izraza nešto ne štima, parser treba baciti iznimku a naredba treba korisniku ispisati prikladnu poruku. Ne postoje nikakvi escapeovi. Primjerice, `${$1}` mora pri parsiranju baciti iznimku.

Jednom kad ste ovo složili na opisani način, pseudokod postupka preimenovanja ponovno možemo osloniti na metodu `filter` koju ste prethodno napisali:

```
List<FilterResult> files = filter(DIR1, MASKA);
NameBuilderParser parser = new NameBuilderParser(IZRAZ);
NameBuilder builder = parser.getNameBuilder();
for(FilterResult file : files) {
  StringBuilder sb = new StringBuilder();
  builder.execute(file, sb);
  String novoIme = sb.toString();
  preimenuj/premjesti file u DIR2/novoIme
}
```

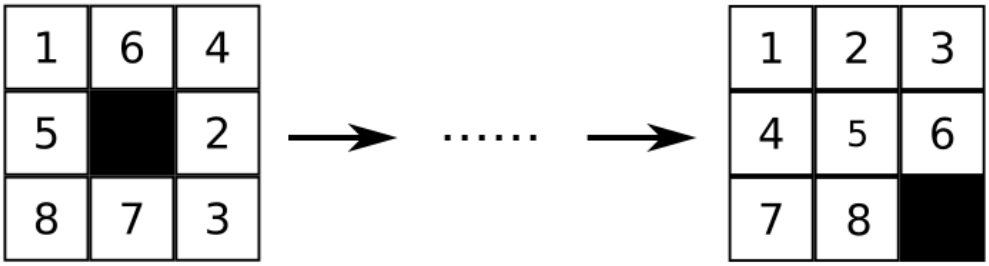
Primijetite: parser parsira `IZRAZ` samo jednom i praktički stvara "program" za izgradnju imena. `Pattern` se također stvara samo jednom. Za svaku datoteku koja je preživjela filtriranje stvara se novi `StringBuilder` i nad njim pokreće program za izgradnju imena.

Ako se prilikom izgradnje imena dogodi pogreška, ili prilikom preimenovanja, naredba se prekida i u ljusci se ispisuje prikladna poruka pogreške.

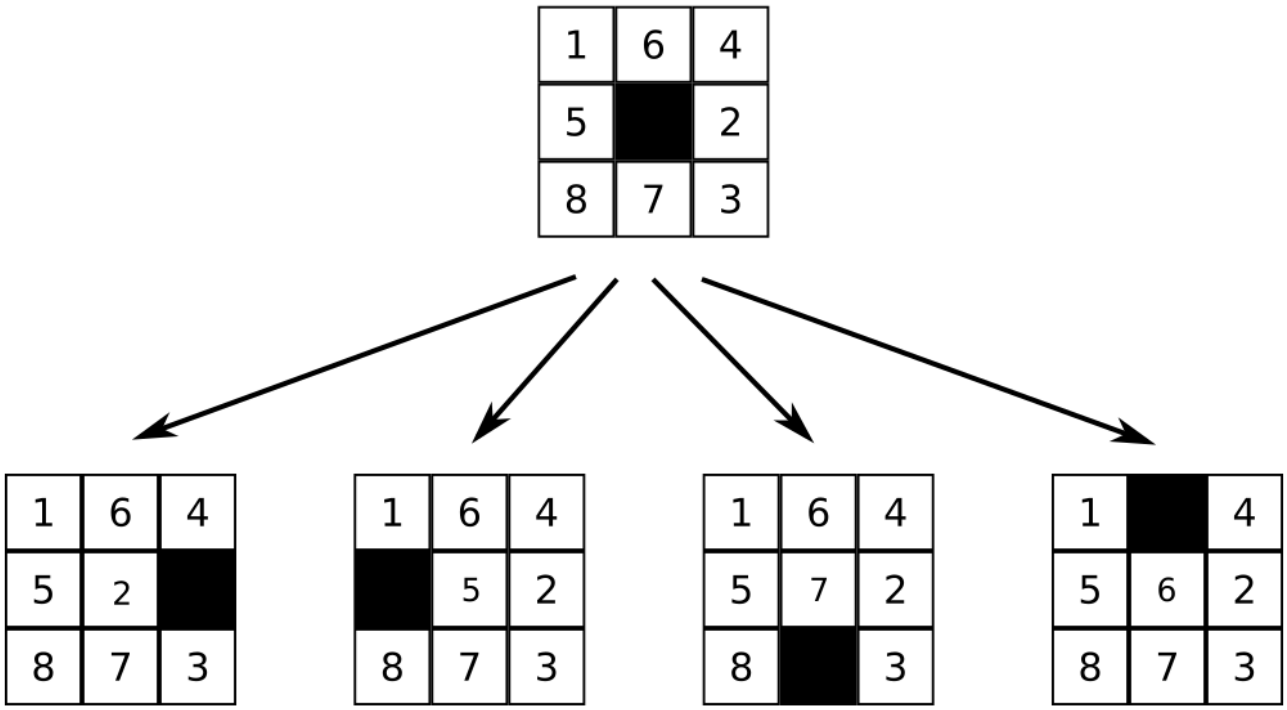
Zadatak 2.

U okviru ovog zadatka igrat ćemo se s pojednostavljenim inačicama osnovnih algoritama pretraživanja prostora stanja (engl. *state-space search algorithms*). Ovi algoritmi spadaju u osnovne algoritme umjetne inteligencije, a time i računarske znanosti.

Razmotrimo jedan konkretan problem: problem slagalice prikazan na sljedećoj slici, pa ćemo zatim poopćiti razmatranje.



Zanima nas slijed koraka koje je potrebno učiniti kako bismo od početno razdešene slagalice prikazane s lijeve strane došli do složene slagalice koja je prikazana na desnoj strani. Svaku konkretnu konfiguracije slagalice zovemo jednim ***stanjem*** razmatranog problema. Postupak rješavanja problema svodi se na provođenje niza koraka, pri čemu se u jednom koraku događa jedna promjena stanja. Primjerice, iz stanja prikazanog na prethodnoj slici (lijevo), jednim korakom (kažemo još "potezom") možemo prijeći u jedno o četiri stanja, kako to prikazuje slika u nastavku.



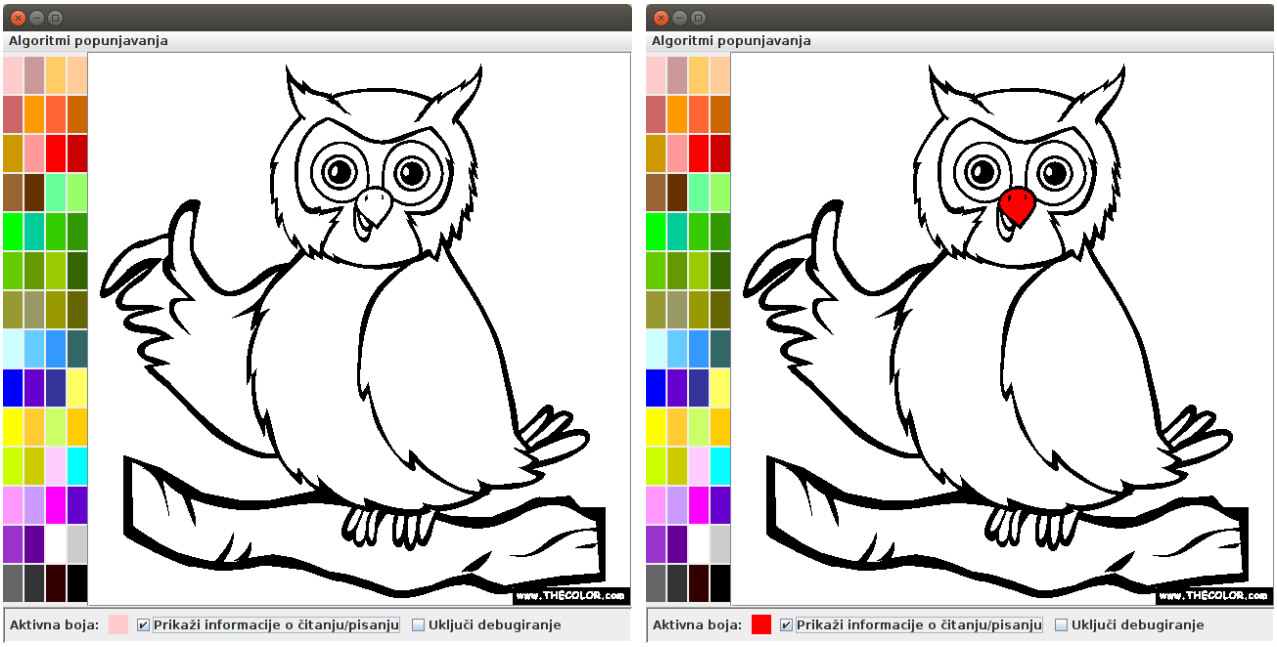
Da bismo mogli krenuti u izradu algoritma koji će rješavati problem slagalice, moramo definirati tri elementa.

1. *Početno stanje* – označit ćemo ga sa $\boldsymbol{s}_0$. Na primjeru slagalice, to je početna konfiguracija koja nam je zadana.
2. *Funkcija sljedbenika* – označit ćemo je sa `succ(`$\boldsymbol{s}$`)`, koja za predano stanje s vraća skup stanja u koja je moguće prijeći u jednom koraku. Na primjeru slagalice, prethodna slika (donji dio) prikazuje četiri stanja u koja je moguće prijeći iz zadanog stanja (gornji dio slike). Za slagalicu, ta će funkcija svakom stanju pridružiti skup od dva, tri ili četiri stanja.
3. *Ispitni predikat* – označit ćemo ga sa `goal(`$\boldsymbol{s}$`)`, koji za svako stanje s vraća true ili false, ovisno o tome je li s ciljno stanje ili nije. Na primjeru slagalice, `goal(`$\boldsymbol{s}$`)` će svakoj konfiguraciji pridružiti vrijednost `false`, osim konfiguraciji u kojoj su, počev od gornje lijeve pozicije, pločice složene redom od 1 do 9 i praznina je u donjem desnom kutu; to će stanje biti preslikano u vrijednost `true`.

Svaki ovakav problem bit će stoga definiran kao uređena trojka $\{\boldsymbol{s}_0, succ(\boldsymbol{s}), goal(\boldsymbol{s})\}$, a rješenje problema bit će niz stanja $\{\boldsymbol{s}_0, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_{n-1}, \boldsymbol{s}_n\}$ pri čemu će vrijediti $goal(\boldsymbol{s}_i)$=false za i od 0 do n-1, te $goal(\boldsymbol{s}_n)$=true. Prilikom izrade algoritma koji će rješavati ovakav problem možemo postaviti i dodatne zahtjeve. Primjerice, možemo tražiti da algoritam ima svojstvo *potpunosti*, što znači da će uvijek pronaći rješenje problema, ako ono postoji. Možemo tražiti da algoritam ima svojstvo *optimalnosti*, što znači da će uvijek pronaći rješenje s minimalnom cijenom (na primjeru slagalice, rješenje koje se sastoji od minimalnog broja koraka).

Kako smo općenit problem pretraživanja prostora stanja definirali kao uređenu trojku od kojih su dva elementa funkcije, implementacijski to znači da ćemo moći pisati općenite metode koje rješavanje problema rade koristeći različite načine pretraživanja prostora stanja, a za koje su spomenute funkcije obične *strategije*! To će nam pak omogućiti da pripremimo malu biblioteku algoritama za rješavanje problema pretraživanja prostora stanja, te da potom za svaki konkretan problem koji trebamo riješiti napišemo prikladne strategije.

No prije no što krenemo u rješavanje opisanog problema, razmotrimo najprije jedan jednostavniji problem koji je podskup opisanih problema: radi se o problemu popunjavanja zatvorenih područja slike: vidi sliku u nastavku (slike/predlošci za popunjavanje preuzete su s www.thecolor.com).



Na slici, lijevo, je prikazana skica sove. Želimo napraviti aplikaciju koja će korisniku omogućiti da odabere boju koju treba iskoristiti za popunjavanje, te da potom na klik miša tom bojom oboji područje u koje je korisnik kliknuo. Rezultat popunjavanja gdje je korisnik kliknuo unutar kljuna prikazan je na slici desno.

Za Vas sam već pripremio (i stavio na ferka) biblioteku `marcupic.opjj.statespace:coloring-framework:1.0` koja nudi implementaciju ovakve aplikacije. Skinite jar, importajte ga u Vaš lokalni maven repozitorij uz maven-koordinate koje sam naveo. Zatim dodajte u Vaš `pom.xml` ovisnost prema ovoj biblioteci. U paket `coloring.demo` dodajte glavni program `Bojanje1` koji sadrži sljedeću metodu `main`:

```
public static void main(String[] args) {
	FillApp.run(FillApp.OWL, null);  // ili FillApp.ROSE
}
```

te pokrenite program. Prvi argument metode `run` je staza do slike koju treba prikazati (možete slati staze i do Vaših slika ako ih preuzmete/nacrtate i stavite negdje na disk), a drugi lista Vaših algoritama popunjavanja (pa stoga `null` u ovom trenutku). Iz izbornika *Algoritmi popunjavanja* moći ćete odabirati različite postupke popunjavanja. Klikom na boje u paleti moći ćete odabrati boju kojom želite raditi popunjavanje. Na dnu su dva *checkbox*-a kojima možete kontrolirati želite li dobivati informacije o "učinkovitosti" algoritma, te želite li izvoditi algoritam malo po malo (*Uključi debugiranje*). Isprobajte aplikaciju da dobijete osjećaj što trebamo napraviti.

Svaka slika u aplikaciji je predstavljena objektom s kojim možemo razgovarati kroz sučelje `Picture`:

```
package marcupic.opjj.statespace.coloring;

public interface Picture {
	int getPixelColor(int x, int y);
	void setPixelColor(int x, int y, int rgb);
	int getWidth();
	int getHeight();
}
```

Na raspolaganju su metode za dohvat informacija o širini i visini slike (koordinate slikovnog elementa u gornjem lijevom uglu su `0,0`; *x*-os ide u desno do `getWidth()-1`; *y*-os ide prema dolje do `getHeight()-1`). Boja slikovnog elementa (*pixela*) na koordinatama x,y čuva se kao jedan `int` (četiri grupe od 8-bita; bitovi prve grupe su 0, bitovi druge grupe čuvaju intenzitet crvene boje, bitovi treće grupe čuvaju intenzitet zelene boje, a bitovi četvrte grupe čuvaju intenzitet plave boje; time je struktura oblika: `0rgb`). Funkcija `getPixelColor` dohvaća boju slikovnog elementa na traženoj poziciji, a funkcija `setPixelColor` postavlja boju na onu predanu kao treći argument metode.

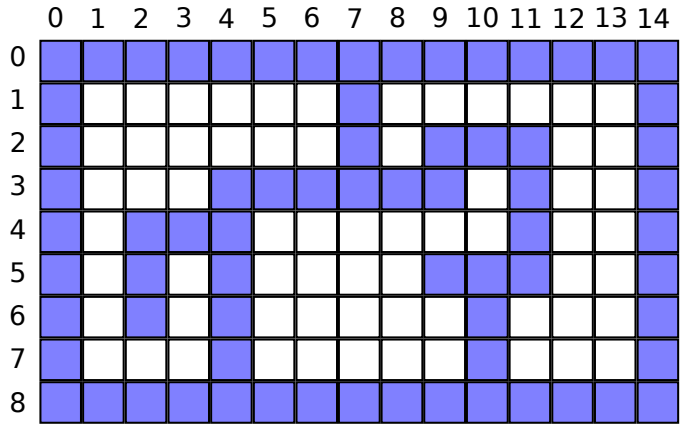
Algoritam popunjavanja modeliran je sučeljem `FillAlgorithm`:

```
package marcupic.opjj.statespace.coloring;

public interface FillAlgorithm {
	void fill(int x, int y, int color, Picture picture);
	String getAlgorithmTitle();
}
```

Metoda `fill` kao argumente dobiva redom *x,y* koordinate slikovnog elementa na koji je korisnik kliknuo mišem (nazovimo to *referentnom točkom*), zatim boju koja je odabrana u paleti te referencu na sliku. Zadaća ove metode je popuniti područje u koje je korisnik kliknuo predanom bojom. Metoda `getAlgorithmTitle` vraća naziv algoritma koji se koristi.

Kako riješiti popunjavanje? Ovo će nam biti radna ideja. Na poziciji na koju je korisnik kliknuo očitat ćemo boju sa slike. Zatim ćemo pretpostaviti da naše područje čine svi slikovni elementi u okolini referentne točke koji su iste boje, te svi slikovni elementi u njihovoj okolini koji su iste boje, te svi slikovni elementi koji su u njihovoj okolini iste boje, te svi … shvatili ste ideju. Ako je neki slikovni element drugačije boje od očitane s referentne točke, taj više nije dio područja koje bojamo te njegove susjede ne gledamo. Pogledajmo ovo na jednom "uvećanom" primjeru prikazanom u nastavku. Imamo sliku dimenzija 15x9. Ako korisnik klikne na lokaciju (1,1), očitali bismo da je boja tog slikovnog elementa bijela i to bismo zapamtili. Zatim bismo morali istražiti susjedstvo te lokacije: slikovne elemente na pozicijama (0,1) - lijevo, (1,0) - iznad, (2,1) - desno, (1,2) - ispod. Za prva dva bismo uočili im boja nije bijela pa njihovo susjedstvo dalje ne bismo gledali. Za druga dva bismo uočili da im je boja bijela, pa bismo krenuli istraživati njihovo susjedstvo, i tako dalje.



Kako opisani postupak u svakom trenutku treba znati koje još lokacije treba istražiti, nekako ćemo to morati pamtiti (u nekoj kolekciji). Stoga ćemo u paket `coloring.algorithms` dodati razred `Pixel` koji ima dvije javne članske varijable tipa `int`: `x` i `y`, napisane `hashCode` i `equals`, i `toString` koji vraća string oblika "(x,y)".

Pseudokod algoritma koji radi bojanje stoga bi bio sljedeći:

```
popuni(x,y,color,slika)
  refboja = slika.ocitaj(x,y)
  kolekcija zaIstraziti = {}
  zaIstraziti.dodaj(Pixel(x,y))
  dok zaIstraziti nije prazna
    Pixel trenutni = zaistraziti.skiniJedanElement()
    ako slika.ocitaj(trenutni.x, trenutni.y) nije refboja: continue
    slika.postavi(trenutni.x, trenutni.y, color)
    u zaIstraziti dodaj susjedne pixele od trenutnog
  kraj
kraj
```

Primijetite da ovdje imamo niz stupnjeva slobode. Za početak: kakva je točno kolekcija `zaIstraziti` – skup, lista, sortirana lista? Ako je obična lista, kamo se točno dodaju novi elementi: na početak ili na kraj? Od kuda točno `zaIspitati.skiniJedanElement` skida element: s početka ili s kraja?

Da bismo poopćili primjer, na tren ćemo se maknuti od konkretnog problema bojanja, pa ćemo primijetiti da rješavamo problem obilaska ograničenog podprostora u prostoru stanja, koji možemo definirati kao uređenu četvorku {s_0, `process(`s`)`, `succ(`s`)`, `acceptable(`s`)`} gdje je:

1. s_0 početno stanje,
2. `process(`s`)` je funkcija koja obavlja zadani posao nad stanjem s,
3. `succ(`s`)` funkcija sljedbenika koja za stanje s vraća skup njegovim izravnih susjeda te
4. `acceptable(`s`)` ispitni predikat koji za stanje s vraća treba li ga obraditi i zatim nastaviti pretragu na njegove susjede, ili to stanje više nije bio podprostora koji obilazimo.

Sada kompletan obilazak podprostora možemo modelirati na sljedeći način:

```
obiđi(s0,process(s),succ(s),acceptable(s))
  zaIstraziti = {s0}
  dok zaIstraziti nije prazna
    si = zaIstraziti.skini
    ako nije acceptable(s): continue
    process(si)
    u zaIstraziti dodaj succ(si)
  kraj
kraj
```

Što nam to znači za implementaciju? Funkcija `process` očito je stragija tipa `Consumer<S>`, funkcija `succ` bi mogla biti strategija tipa `Function<S,List<S>>` (konceptualno, bolje bi bilo `Function<S,Set<S>>`, no implementacije lista u Javi su memorijski kompaktnije pa ćemo njih koristiti), dok bi funkcija `acceptable` bila strategija tipa `Predicate<S>`. Pišemo li jedan razred koji će metodi obiđi moći ponuditi sve relevantne informacije, tada na neki način još trebamo omogućiti da klijent pita za početno stanje. To možemo napraviti tako da implementiramo još i funkcijsko sučelje `Supplier<S>`.

Napišite u paketu `coloring.algorithms` razred `Coloring` koji implementira sva četiri sučelja nad tipom `Pixel`. Razred treba još imati i četiri privatne članske varijable:

```
  Pixel reference;
  Picture picture;
  int fillColor;
  int refColor;
```

zatim konstruktor koji prima `reference`, `picture` i `fillColor`, a sam postavlja `refColor` na boju koju u slici ima slikovni element čija je lokacija predana kroz `reference`.

U isti paket sada dodajte razred `SubspaceExploreUtil`. U njega stavite javnu statičku metodu sljedeće signature:

```
public static <S> void bfs(
  Supplier<S> s0,
  Consumer<S> process,
  Function<S,List<S>> succ,
  Predicate<S> acceptable
);
```

i potom napišite tu metodu tako ponudi implementaciju prema pseudokodu `obiđi` pri čemu se kao kolekcija `zaIstražiti` koristi lista, skidanje se radi s početka liste, a dodavanje se radi na kraj liste. Stoga kao prikladnu implementaciju odaberite `LinkedList`.

Napravite sada kopiju razreda `Bojanje1` u `Bojanje2`, dodajte jedan konkretan algoritam bojanja koji se oslanja na napisanu metodu `SubspaceExploreUtil.bfs` i implementaciju konkretnog problema koju ste napravili kroz razred `Coloring`:

```
public static void main(String[] args) {
	FillApp.run(FillApp.OWL, Arrays.asList(bfs));
}

private static final FillAlgorithm bfs = new FillAlgorithm() {

	@Override
	public String getAlgorithmTitle() {
		return "Moj bfs!";
	}

	@Override
	public void fill(int x, int y, int color, Picture picture) {
		Coloring col = new Coloring(new Pixel(x,y), picture, color);
		SubspaceExploreUtil.bfs(col, col, col, col);
	}
};
```

Primijetite, sada u metodi `run` dajemo dopunsku listu koja sadrži referencu na naš prvi algoritam popunjavanja. Pokrenite program te iz izbornika *Algoritmi popunjavanja* odaberite ovaj novi algoritam. Odvucite prozor u gornji lijevi ugao ekrana, uključite "*debug*" na dnu prozora, odaberite primjerice crvenu za boju ispune, pa kliknite na središnji dio tijela sove. Pustite algoritam da boja sto po sto slikovnih elemenata. Možete li objasniti što se događa? Ključ je u opažanju da algoritam nova stanja koja treba istražiti dodaje na kraj liste, a elemente uzima s početka liste. Skicirajte si ovo na papiru pa bi trebalo postati jasno zašto se ovakav obilazak naziva bfs (engl. *Breadth-First Search*).

Dodajte sada u razred `SubspaceExploreUtil` javnu statičku metodu sljedeće signature:

```
public static <S> void dfs(
  Supplier<S> s0,
  Consumer<S> process,
  Function<S,List<S>> succ,
  Predicate<S> acceptable
);
```

i potom napišite tu metodu tako ponudi implementaciju prema pseudokodu `obiđi` pri čemu se kao kolekcija `zaIstražiti` koristi lista, skidanje se radi s početka liste, a dodavanje se također radi na početak liste (imate metodu `List#addAll(index,collection)` koju iskoristite za ovo tako da u jedmo pozivu na početak umetnete kompletnu listu susjeda). Kao prikladnu implementaciju odaberite `LinkedList`. Potom dodajte u razred `Bojanje2` statičku konstantu `dfs` koja je implementacija algoritma bojanja koji se oslanja na ovu metodu, pa dodajte i tu referencu u listu koju šaljete u metodi `run`. Pokrenite i pogledajte sada kroz debug kako se ponaša program, odnosno kojim redoslijedom istražuje prostor. Olovka – papir – možete li si objasniti zašto se ovakav način obilaska naziva dfs (engl. *Depth-First Search*)?

Razmotrimo sada učinkovitost ovih algoritama. Uz uključen prikaz informacija uočit ćete da popunjavanje centralnog dijela sove rezultira s 109 646 čitanja boje slikovnog elementa, a samo s 27 411 pisanja boje slikovnih elemenata. Drugim riječima: da bismo obojali površinu od cca 27 000 slikovnih elemenata, generirali smo preko 100 000 stanja koja su prošla kroz listu `zaIstražiti`. Otkuda takav višak stanja? Razlog je činjenica da postoji više stanja koja dijele istog susjeda. U ovom konkretnom primjeru, svako stanje (slikovni element) je zajednički susjed od četiri druga

stanja (slikovna elementa), pa ekspanzijom susjedstva svakog od njih nastaje po jedna kopija, čime dobijemo 4 puta veći broj stanja koja trebamo istražiti. I doista, 109 646 = (4 x 27 411) + 1. Učinkovitost ovih postupaka možemo popraviti tako pratimo koja smo sve stanja već istražili ili ih već imamo zakazane za istraživanje, pa njih izbrišemo iz vraćene liste susjeda prije no što napravimo umetanje u listu `zaIstražiti`. Kolekciju koja prati što smo već posjetili (a u ovom primjeru dopunjujemo s: ili zakazali za posjetu) uobičajeno nazivamo *listom posjećenih stanja*. Pri tome se pojam "liste" ovdje odnosi na koncept – ne implementaciju.

Dodajte sada u razred `SubspaceExploreUtil` javnu statičku metodu sljedeće signature:

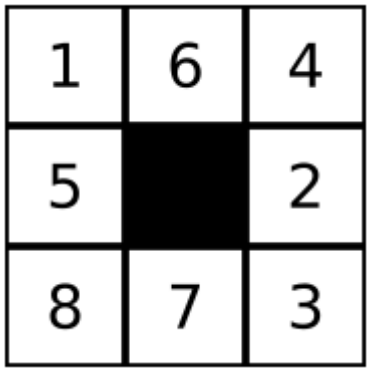
```
public static <S> void bfsv(
  Supplier<S> s0,
  Consumer<S> process,
  Function<S,List<S>> succ,
  Predicate<S> acceptable
);
```

i potom napišite tu metodu tako ponudi implementaciju prema pseudokodu `obiđi_uz_listu_posjećenih` (dan u nastavku) pri čemu se kao kolekcija `zaIstražiti` koristi lista, skidanje se radi s početka liste, a dodavanje na kraj liste. Kao prikladnu implementaciju odaberite `LinkedList`. Kolekcija `posjećeno` će biti `Set`, a kao implementaciju odaberite `HashSet`. Potom dodajte u razred `Bojanje2` statičku konstantu `bfsv` koja je implementacija algoritma bojanja koji se oslanja na ovu metodu, pa dodajte i tu referencu u listu koju šaljete u metodi `run`.

```
obiđi_uz_listu_posjećenih(s0,process(s),succ(s),acceptable(s))
  zaIstraziti = {s0}
  posjećeno = {s0}
  dok zaIstraziti nije prazna
    si = zaIstraziti.skini
    ako nije acceptable(s): continue
    process(si)
    djeca = succ(si)
    iz djeca ukloni sve koji su u posjećeno
    u zaIstraziti dodaj djeca
    u posjećeno dodaj djeca
  kraj
kraj
```

Sada bi postupak bojanja centralnog dijela trebao obaviti samo 28300 čitanja i 27411 pisanja.

Povratak na slagalicu. Sada kada ste ovo napravili i razumijete razliku između navedenih metoda obilazaka, spremni smo za rješavanje problema slagalice. Prisjetite se, problem pretraživanja prostora stanja definirali smo kao uređenu trojku {$\boldsymbol{s}_0$, succ($\boldsymbol{s}$), goal($\boldsymbol{s}$)}. U ovom problemu znamo koje je početno stanje, a zanima nas put od njega do ciljnog stanja. Razmotrimo stoga ponovno jedno konkretno stanje, na primjeru slagalice.



Mogli bismo primjerice pločicu 2 pomaknuti u centar i time dobiti sljedeće stanje. Iz tog novog stanja imamo tri moguća susjeda: možemo 4 spustiti dolje, možemo 3 dignuti gore ili možemo 2 pomaknuti desno. Uzmemo li ovu posljednju opciju, ponovno smo završili u početnom stanju. Primijetite, međutim, da za igrača igre ono prvo početno stanje i ovo u kojem smo završili nakon dva poteza zapravo nisu jednakovrijedna – u posljednjem slučaju "potrošili" smo dva poteza i to nas košta. Stoga ćemo pri modeliranju algoritma koji obavlja postupak pretraživanja prostora stanja morati uvesti još jedan pojam: čvor stabla pretraživanja. Čvor (koristit ćemo naziv engl. *Node*) je objekt koji čuva:

1. referencu na stanje u kojem se nalazimo,
2. referencu na čvor koji nam je roditelj (i koji čuva referencu na stanje iz kojeg je jednim potezom nastalo stanje koje mi čuvamo) te
3. cijenu koju smo do sada platili da bismo došli u stanje koje čuvamo (ovdje će cijena odgovarati broju poteza koje smo odigrali od početka igre).

Uz opisano, čvor ćemo modelirati parametriziranim razredom `Node<S>` koji ima konstruktor koji prima (redoslijedom koji je ovdje naveden) referencu na roditeljski čvor, stanje (tipa `S`) i cijenu (tipa `double`), te ima javne metode:

```
  S getState();
  double getCost();
  Node<S> getParent();
```

Napravite paket `searching.algorithms` te u njega dodajte ovaj razred uz cjelovitu implementaciju.

Sada možemo dati i pseudokod algoritma pretraživanja prostora stanja:

```
pretraži(s0,succ(s),goal(s))
  zaIstraziti = {Node(null,s0,0)}
  dok zaIstraziti nije prazna
    ni = zaIstraziti.skini
    ako goal(ni.getState()): return ni // preko ni.getParent() imamo stazu...
    u zaIstraziti dodaj {Node(ni,sj,ni.cost+cj) | (sj,cj) iz succ(ni.getState())}
  kraj
kraj
```

Primijetite koje su razlike u odnosu na pseudokod algoritma obiđi: naše kolekcije sada više ne čuvaju stanja već čuvaju čvorove. Također, funkcija `succ(`$\boldsymbol{s}_i$`)` više ne vraća skup susjednih stanja, već vraća skup uređenih parova ($\boldsymbol{s}_j$,$\boldsymbol{c}_j$) gdje je prvi element para susjedno stanje, a drugi element para cijena koju plaćamo kada iz stanja $\boldsymbol{s}_i$ pređemo u stanje $\boldsymbol{s}_j$. Na ovaj način imamo definiran poprilično općenit algoritam pretraživanja koji možemo iskoristiti i za zadatke pronalaska najkraćeg puta između dvije zadane točke (ali zamislite scenarij u kojem prolazimo kroz krajolik koji ima uzvisine i udoline, pa tražimo "najkomforniji" put: tada bi nas, primjerice, više koštao put kojim se penjemo no put kojim se spuštamo).

Dodajte u paket `searching.algorithms` razred `Transition<S>` koji predstavlja uređeni par ($\boldsymbol{s}_j$,$\boldsymbol{c}_j$); treba imati članske varijable `state` i `cost`, konstruktor koji ih prima te gettere.

Uz ovako definirane razrede, argumenti metode pretraži bili bi strategija tipa `Supplier<S>` za dohvat početnog stanja, strategija tipa `Function<S,List<Transition<S>>>` za funkciju sljedbenika te strategija tipa `Predicate<S>` za ispitni predikat.

U paketu `searching.slagalica` napravite razred `Slagalica` koji implementira sva tri sučelja pri čemu parametar `S` odgovara tipu `KonfiguracijaSlagalice`. Stanje slagalice predstavit ćemo razredom `KonfiguracijaSlagalice` koji interno konfiguraciju čuva u *privatnom* polju cijelih brojeva (`int[]`). Radi se o polju od devet elemenata: pobrojat ćemo polja slagalice s lijeva u desno, odozgo prema dolje, te u polje upisati koji broj piše na tim pozicijama; vrijednost `0` će označavati prazno polje. U ovom zapisu konfiguraciju slagalice prikazanu na prethodnoj stranici čuvali bismo u polju `int[] {1,6,4,5,0,2,8,7,3}`. Razred `KonfiguracijaSlagalice` kroz konstruktor prima referencu na polje opisanog formata koje pamti. Razred nudi i javni getter `int[] getPolje()` koji vraća *kopiju* internog polja, te javnu metodu `int indexOfSpace()` koja vraća indeks elementa 0 iz internog polja. U prethodno danom primjeru polja, ova bi metoda vratila 4.

Razred `Slagalica` treba imati konstruktor koji prima početnu konfiguraciju (referenca na `KonfiguracijaSlagalice`). Napišite sada metode potrebne zbog implementiranih sučelja. Funkcija `succ(`$\boldsymbol{s}_i$`)` vraća uređene parove gdje je cijena prijelaza uvijek 1 ("košta" nas jedan potez koji smo napravili).

U paket `searching.algorithms` dodajte razred `SearchUtil` koji ima javnu statičku metodu sljedeće signature:

```
public static <S> Node<S> bfs(
  Supplier<S> s0,
  Function<S, List<Transition<S>>> succ,
  Predicate<S> goal);
```

Napišite implementaciju te metode kao *bfs*, prema pseudokodu metode pretraži s prethodne stranice. Kolekcija `zaIstražiti` treba biti `List`, implementacije `LinkedList`. Skidanje radimo s početka, a dodavanje ide na kraj. Pazite: kolekcija sada čuva čvorove, ne izravno stanja. Prvi čvor nema roditelja, čuva početno stanje i cijena mu je 0. Primijetite da taj postupak ne garantira pronalazak rješenja s minimalnom cijenom jer ista nema utjecaja na redoslijed ispitivanja čvorova. Međutim, mi ćemo ovaj postupak primijeniti na problem slagalice kod kojega cijena uvijek monotono raste za 1, pa ćemo u tom slučaju doista i dobiti rješenje s minimalnim brojem poteza. Želimo li algoritam koji bi za opći slučaj garantirao pronalazak rješenja s minimalnom cijenom, trebali bismo iz liste uvijek skidati čvor koji od svih pohranjenih u listu ima minimalnu cijenu (pa bi prikladna struktura bio neki oblik prioritetnog reda – što ne radimo u ovoj zadaći). Takav algoritam ima I svoj naziv: *ucs* (od engl. *Uniform Cost Search*).

Sada u paket `searching.demo` dodajte razred `SlagalicaDemo` prikazan u nastavku.

```
package searching.demo;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import searching.algorithms.Node;
import searching.algorithms.SearchUtil;
import searching.slagalica.KonfiguracijaSlagalice;
import searching.slagalica.Slagalica;

public class SlagalicaDemo {

	public static void main(String[] args) {
		Slagalica slagalica = new Slagalica(
			new KonfiguracijaSlagalice(new int[] {2,3,0,1,4,6,7,5,8})
		);

		Node<KonfiguracijaSlagalice> rješenje =
			SearchUtil.bfs(slagalica, slagalica, slagalica);

		if(rješenje==null) {
			System.out.println("Nisam uspio pronaći rješenje.");
		} else {
			System.out.println(
				"Imam rješenje. Broj poteza je: " + rješenje.getCost()
			);
			List<KonfiguracijaSlagalice> lista = new ArrayList<>();
			Node<KonfiguracijaSlagalice> trenutni = rješenje;
			while(trenutni != null) {
				lista.add(trenutni.getState());
				trenutni = trenutni.getParent();
			}
			Collections.reverse(lista);
			lista.stream().forEach(k -> {
				System.out.println(k);
				System.out.println();
			});
		}
	}

}
```

Pokretanjem programa trebate dobiti sljedeći ispis:

```
Imam rješenje. Broj poteza je: 6.0
2 3 *
1 4 6
7 5 8

2 * 3
1 4 6
7 5 8

* 2 3
1 4 6
7 5 8
```

```
1 2 3
* 4 6
7 5 8

1 2 3
4 * 6
7 5 8

1 2 3
4 5 6
7 * 8

1 2 3
4 5 6
7 8 *
```

što ilustrira korak po korak kako od početne konfiguracije dolazimo do konačne konfiguracije. Da biste dobili ovakav ispis, još ste na jednom mjestu nešto trebali doraditi – napravite to.

Primijetite da u ovom primjeru nismo radili s konfiguracijom prikazanom na slikama u ovoj uputi. Taj je problem, nažalost, nerješiv, i naš program nikada neće završiti, jer neprestano istražuje do kuda može doći s još jednim potezom više. Zapravo, strogo gledajući, program će završiti: kako mu lista `zaIstražiti` neprestano raste, u jednom trenutku program će ostati bez memorije pa će se srušiti uz `OutOfMemoryError`.

Uz opažanje da u našem slučaju cijena raste monotono i za konstantan iznos, možemo napisati i bolju implementaciju (*bfsv*) koja bi dodatno koristila skup posjećenih stanja (doista stanja, ne čvorova), i koja bi u listu `zaIstražiti` dodavala samo čvorove čija stanja nisu u skupu posjećenih stanja (istovremeno dodajući onda i ta stanja u skup posjećenih stanja). Dodajte tu implementaciju u razred `SearchUtil` pa u metodi `main` razreda `SlagalicaDemo` sada pozovite tu metodu. Uz rješivu konfiguraciju `int[] {2,3,0,1,4,6,7,5,8}` program će ispisati rješenje duljine 6, za pokrenut nad nerješivom konfiguracijom `int[] {1,6,4,5,0,2,8,7,3}` program će ispisati da nema rješenja.

Puno više o ovakvim i sličnim algoritmima te njihovim unaprjeđenjima (primjerice, kako smanjiti broj čvorova koje algoritam pretraživanja generira i istražuje, oslanjajući se na heuristike), a koji imaju mnoge primjene, uključivo razvoj inteligentnih igrača računalnih igara, pomoć pri automatiziranom upravljanju i slično, ako će Vas zanimati, moći ćete čuti na odgovarajućim kolegijima računarske znanosti (poput *Umjetne inteligencije* na trećoj godini preddiplomskog studija).

Na ferka sam Vam stavio još jedan jar: slagalica-gui-1.0.jar. Importajte ga u Vaš lokalni maven repozitorij na koordinate `marcupic.opjj.statespace:slagalica-gui:1.0`. Pripazite: kod koji je u njemu očekuje da ste prethodne razrede imenovali točno kako je navedeno u ovoj uputi, te da ste ih smjestili u pakete točno kako je propisano. Dodajte sada u Vaš `pom.xml` ovisnost prema ovoj biblioteci. Biblioteka nudi razred `SlagalicaViewer` s jednom javnom statičkom metodom:

```
public static void display(Node<KonfiguracijaSlagalice> rješenje);
```

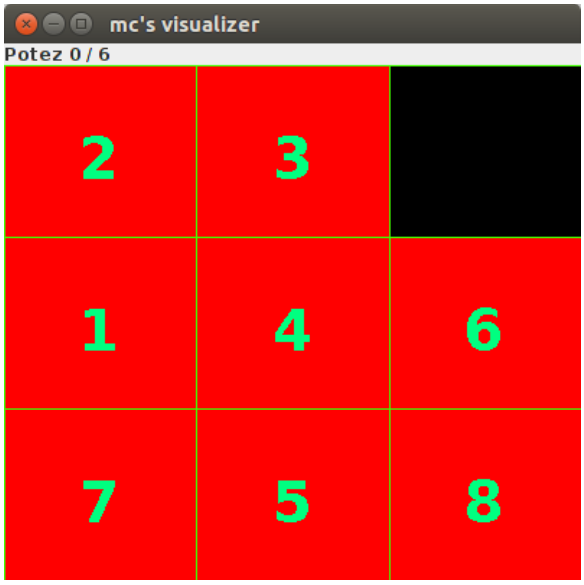
U paketu `searching.demo` dodajte razred `SlagalicaMain` kao kopiju razreda `SlagalicaDemo` koji smo prethodno prikazali. Potom preradite metodu `main` tako da korisnik konfiguraciju slagalice zadaje kao jedan argument u naredbenom retku:

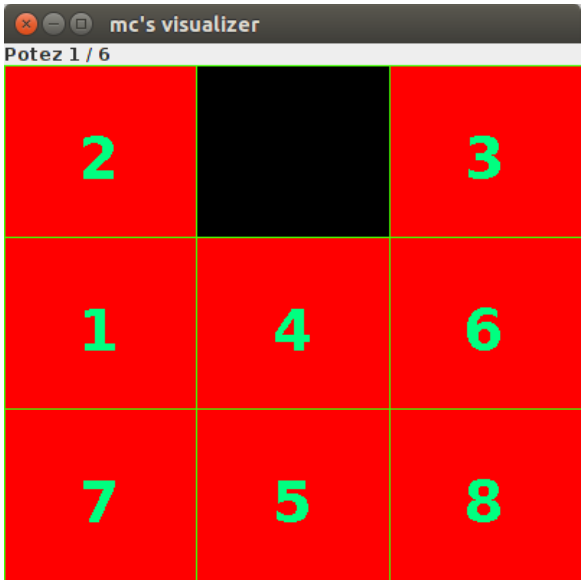
```
java -cp štoVećTreba searching.demo.SlagalicaMain 230146758
```

Vaš program treba provjeriti da je konfiguracija ispravna (duljina: 9, prisutne sve znamenke od 0 do 8). Ako to nije slučaj, prijavite pogrešku korisniku i prekinite program. U suprotnom, riješite slagalicu pozivom Vaše pripremljene metode `bfsv`, te modificirajte granu u koju ulazimo ako slagalica ima rješenje, tako da uz ispis na ekran na kraj još pozove:

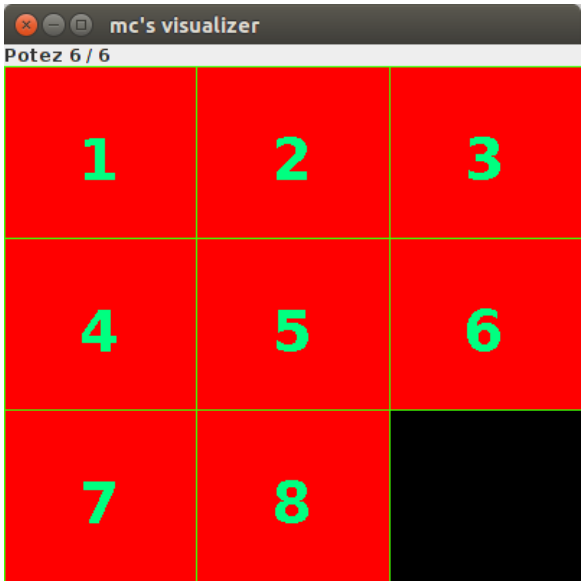
```
SlagalicaViewer.display(rješenje);
```

Dobit ćete grafički prikaz slagalice. Na lijevi klik miša odrađuje se sljedeći potez, na desni klik prethodni potez.





...



Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else). For this homework you can use Java Collection Framework classes. Document your code!

The consultations are at standard times. Feel free to drop by my office after email announcement.

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

Since midterm exams are in progress, you are not required to write any junit tests. But I suggest you do it anyway (especially with parser for `IZRAZ`).

When your homework is done, pack it in zip archive with name `hw08-0000000000.zip` (replace zeros with your JMBAG). Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is Friday, May 3rd 2019. at 07:00 AM.