PRETRAGA:

Lojdova slagalica:

- sastoji se od 4x4 polja koja su popunjena brojevima od [1, 15] a poslednje polje je prazno. To se moze predstaviti listom [1, 2, ... , 15, *] ako su svi brojevi na mestu. Cilj igre je dovesti sve brojeve u tu konfiguraciju. Brute force pristup bio bi da za svako stanje razmatramo sva moguca sledeca, pa za to sledece sva moguca sledeca i tako do kraja gde je garantovano pronalazenje resenja. Pokazalo se da je za neke pocetne konfiguracije slagalicu moguce sloziti u najvise 80 koraka pa bi to bilo neko gornje ogranicenje. Odatle sledi da je potrebno ispitati 2^80 (ako posmatramo prazno polje, njega u svakom stanju mozemo da pomerimo u tacno dva smera) mogucnosti sto je prakticno neizvodivo. Jedno od resenje jeste da se nekako _navodi* pretraga. Moguce navodjenje bilo bi da se igraju samo potezi koji vode do stanja koje je blize resenju. Da je sledece stanje koje razmatramo blize od trenutnog mozemo da izracunamo kao **sumu udaljenosti svih polja od mesta na kom treba da budu** (to je heuristika). Pakazace se da ovo ne vodi do resenja jer neka stanja nikada nemaju sledece stanje koje je blize resenju.

Problemi pretrage najpogodnije se predstavljaju **grafovima** gde su **cvorovi stanja** a **grane akcije**. Mogu biti **usmereni** (sah) ili **neusmereni** (Lojdova slagalica, putanje izmedju gradova...). Graf je neusmeren ako se akcijom koja ima istu cenu moze iz cvora A doci u B i iz cvora B doci u A. Obilaskom grafa formira se **stablo pretrage**. S obzirom da se neki cvor u grafu moze vise puta obici, u stablu se isti cvor moze vise puta javiti. Zbog toga mogu postojati **beskonacna stabla pretrage za konacne grafove**.

Algoritmi pretrage:

Resenje problema pretrage je niz koraka (akcija) koje treba izvesti da bi se iz pocetnog doslo u zavrsno stanje. U zavisnosti od samog tipa problema, koriste se razliciti algoritmi pretrage ali svi treba da zadovoljavaju sledece:

1. Potpunost:
    - garantuje da ce algoritam **naci resenje ako resenje postoji**. Ova osobina je **pozeljna** ali nekad **nije neophodna**. Naime, neki veoma teski problemi se mogu resiti koriscenjem heuristika koje nisu potpune ali ce u prosecnom slucaju mnogo pre dati resenje od algoritama koji su potpuni.
2. Optimalnost:
    - nadjeno **resenje uvek ima najmanju cenu**. Cena se racuna kao zbir cena akcija koje se preduzimaju. Nekad cena moze da se posmatra kao vremenska i prostorna slozenost algoritma pri cemu optimalan ima najmanju cenu. Optimalnost, kao i potpunost, **nekad nije neophodna** jer odredjeni neoptimalni algoritmi u prosecnom slucaju daju resenja koja su bliska optimalnim ali ih nalaze mnogo pre.
3. Vremenska i prostorna slozenost:
    - razmatraju se kao i kod svih drugih algoritama i vazne su i najgora i prosecna slozenost.

Algoritmi pretrage mogu biti **informisani** ili **neinformisani**. Ukoliko postoje informacije o kvalitetu pojedinacnih stanja onda se te informacije mogu koristiti za navodjenje pretrage i tada se radi o informisanim algoritmima. Jedan od tih algoritama

je i **A***. Ukoliko te informacije nisu poznate onda se pretraga ne moze navoditi vec je potrebno smisliti neko sistematicno (putpuno) pretrazivanje (u sirinu/dubinu npr.). Lojdova slagalica mora se resiti neinformisanim algoritmom jer pored cene svake akcije i neposredno dostupnih stanja, ni jedna druga informacije nije dostupna. Sa druge strane, rastojanje izmedju gradova moze se naci informisanim algoritmom koje bi koristilo vazdusno rastojanje koje lako moze da se izracuna kao euklidsko rastojanje izmedju dva cvora u grafu.

NEINFORMISANI ALGORITMI PRETRAGE:

Neinformisani algoritmi pretrage **nemaju dodatne informacije koje bi navodile pretragu** i tako je znacajno skratile. Umesto toga, koristi se sistematicno ispitivanje svih mogucih stanja. Neki od tih algoritama su DFS i BFS (ili bektreking kao modifikacija DFS-a). Lavirint je klasican problem koji se resava ovim algoritmima. **Depth first search** algoritam ispituje sto dalje cvorove od polaznog. Nerekurzivna implementacija koristi stek, odnosto LIFO strukturu, i kada dodje do cvora odakle ne moze dalje, onda skida poslednji element sa steka i od njega nastavlja pretragu ali pre toga svako poseceno polje oznaci tako da ne bismo upali u beskonacnu petlju. Kada dodjemo do ciljnog polja onda samo procitamo u obrnutom redu stek i dobili smo putanju. Algoritam staje kada je stek prazan. DFS prestavlja *pracenje desne strane* za pronalazak izlaza iz lavirinta.

Problem n dama: - kako na sahovskoj tabli dimenzije n * n postaviti n dama tako da se nikoje dve dame ne napadaju. Ovo je tipican primer neinformisane pretrage pomocu modifikovanog DFS-a koji koristi bektreking.

Bektreking se sastoji od nadogradjivanja parcijalnog resenja. U problemu n dama, pocetno parcijalno resenje je prazna tabla. Ovo parcijalno resenje se u svakom koraku nadogradjuje dodavanjem dame na neko od slobodnih polja. Posto dve dame ne mogu da se nalaze u istoj vrsti, razmatramo kolonu po kolonu. Glavna osobina bektrekinga je da ce prekinuti pretragu stabla cim se dogodi da parcijalno resenje ne moze da bude validno. U ovom slucaju to ce biti cim se neke dve dame napadaju. Tada se algoritam vraca na prvo stanje u kojem se ove dve dame nisu napadale (kada je parcijalno resenje moglo da vodi do resenja) i pritom odseca deo stabla u kom je prethodno bio (onaj gde je parcijalno resenje postalo nevalidno).

Breadth-first search je neinformisani algoritam pretrage koji uvek daje ciljni cvor koji je na najmanjem rastojanju od polaznog. BFS u red, FIFO strukturu, dodaje sve cvorove susedne trenutnom, zatim uzima prvi iz reda i dodaje na kraj sve njemu susedne i sve tako dok se ne dodje do ciljnog cvora ili praznog reda. DFS algoritam pogodniji je kada je usmeravanje pretrage moguce.

Vremenska slozenost i DFS-a i BFS-a je O(|V| + |E|). Prostorna O(|E|)

Dajkstrin algoritam koristi se za pronalazak najkracih puteva u grafu **koji nema negativne cene grana**. Po zavrsetku algoritma dobija se najkrace rastojanje do ciljnog cvora ali se algoritam lako moze modifikovati da se dobije najkrace rastojanje do svih cvorova:

1. Na pocetku, svi cvorovi grafa se ubace u skup Q i pritom se cena puta od pocetnog cvora do svakog stavi na +INF (sem za pocetni cvor gde se postavi na 0). Roditelj svakog cvora na pocetku je takodje nepoznat.
2. Zatim se iz skupa Q uzima cvor n koji ima najmanje rastojanje od pocetnog (u prvom koraku to je pocetni cvor). Ako je taj cvor ciljni onda se prekida algoritam i rekonstruise putanja preko roditelja. Ako to nije ciljni cvor, onda

se ispituje svaki svaki njegov sused m. Ako je prethodno rastojanje od pocetnog cvora do m vece od rastojanja od pocetnog cvora preko n pa do m onda se postavlja novo rastojanje od pocetnog cvora do m i ono je jednako rastojanju od pocetnog preko n do m.

Invarijanta petlje je da se za sve cvorove koji nisu u Q zna najkrace rastojanje od pocetnog. Slozenost Dajkstrinog algoritma, gde je skup Q implementiran preko povezane liste je O(|V|^2). Ako graf ima mnogo manje grana od |V|^2 onda se skup moze implementirati preko Min-Heap-a i tada je slozenost O((|V| + |E|) * log(|V|)).

INFORMISANI ALGORITMI PRETRAGE:

Informisana pretraga naziva se jos i **heuristicka pretraga** i ona u svakom stanju, pored informacija o sledecim mogucim akcijama, ima informacije i o tome koja akcija moze da dovede do stanja koja vise obecavaju. Ta informacija moze da bude neka mera *kvaliteta* stanja, koja ne mora da bude konstantna (moze da bude drugacija za isto stanje u zavisnosti o toga u kom smo stanju trenutno ili u kom je stanju celokupna pretraga). Heuristika "navodi" nas algoritam ka brzem pronalasku resenja. Taj *kvalitet* stanja cesto nije egzaktan vec prestavlja neku procenu. **Funkcija evaluacije** ocenjuje kvalitet stanja. Kazemo da je f(n) - funkcija evaluacije za stanje n.

Pohlepni algoritmi u svakom koraku uzimaju najbolje lokalno resenje u nadi da ce tako doci do najboljeg globalnog resenja. Pokazace se da vrlo cesto ovo nije slucaj i da se cak u nekim sutuacijama nece ni doci do resenje iako ono postoji. U primeru sa pronalaskom najkraceg puta izmedju gradova, dodatna informacija koju bi pohlepni algoritam koristio bila bi vazdusno rastojanje izmedju gradova. Pretpostavljamo da znamo vazudusna rastojanja iz svakog do svakog grada, tada bi pohlepni algoritam iz grada n za sledeci grad birao onaj sa najmanjim vazdusnim rastojanjem a cena bi bila jednaka pravoj kopnenoj ceni od pocetnog grada do grada n i najmanjoj vazusnoj ceni od grada n do sledeceg grada. Ni u ovom primeru iz knjige ne dobija se optimalno resenje. **Pohlepni algoritmi** cesto su veoma efikasni i laki za implementaciju ali **ne garantuju optimalnost pa ni potpunost**. Lojdovu slagalicu nije moguce resiti pohlepnim algoritmom jer oni kao kvalitet stanja najcesce uzimaju Menhetn rastojanje svakog polja od ciljnog, a cesto se desava da slagalica ima takvu konfiguraciju da pohlepni algoritam ne moze da pomeri ni jedno polje jer se kvalitet tog stanja smanjuje tj. on se odaljava od ciljnog (odnosno stiglo se do lokalnog maksimuma). Tada se pohlepni algoritam glavi jer on nikad ne sme da uzima lokalni minimum koji pogorsava globalni minimum do kog se doslo.

Pohlepni algoritmi se mogu koristiti u matematickoj optimizaciji i tada se nazivaju **algoritmi penjanja uzbrdo**. Oni u svakom koraku biraju susedna dopustiva resenje koja imaju najvise vrednosti funkcije cilja. Oni imaju neke slabosti:

1. Opasnost od lokalnih maksimuma:
    - postoji kada svi dopustivi susedi imaju manju vrednost funkcije cilja od trenutne, tj. stiglo se u tacku odakle se ne moze dalje ali vrednost funkcije cilja u toj tacki nije max globalna vec je samo max lokalna. Odavde pohlepni algoritam nema kuda pa staje i vraca pogresno resenje.
2. Neefikasnot u slucaju grebena:
    - greben prestavlja usku stazu koja opada ili raste duz nekog pravca. Tada ce algoritam penjanja uzbrdo biti neefikasan jer umesto da se pravolinijske penje on ce ici u cik-cak i tako biti mnogo sporiji.
3. Opasnost od platoa:

    - nastaje kada oblast prostora pretrage ima konstantu vrednost funkcije cilja. Tada pohlepni algoritam ne zna sta dalje da radi.

Ovi problemi mogu se resiti nekim varijacijama penjanja. Tako **stohasticko penjanje** ne bira uvek suseda sa najboljom vrednoscu funkcije cilja vec ce verovatnoca da on bude izabran biti proporcionalna toj vrednosti. Druga varijacija je **penjanje uzbrdo sa slucajnim resetovanjem** kod koga se nakon pronalaska nekog maksimuma algoritam ponovo pokrece iz slucajno izabrane tacke. Verovatnoca da ce se tako doci do globalnog maksimuma se priblizava 1 sto vise puta pokrenemo penjanje.

Najstrmiju spust (**gradijentni spust**) je samo -funkcija cilja.

Minimalna/maksimalna tacka ne mora da bude bas egzaktna vec je to ona iz koje bi pomeraj bio manji od nekog definisanog *epsilon*. Kada se spustamo moramo da pazimo da duzina koraka za koji se spustamo bude optimalna, tj. moramo da pazimo da korak **nije predugacak** jer ce doci do divergencije (proci cemo trazenu vrednost) i moramo da pazimo da korak **nije prekratak** jer cemo se mnogo sporo spustati. To mozemo da obezbedimo tako sto uzmemo da je taj korak

```
Lambda_n = 1 / (n + 1)
```

Sledeci korak:

```
a_(n+1) = a_n - Lambda_n * Delta * f(a_n)
```

Ni gradijentni spust ne daje garanciju da ce doci do globalnog minimuma vec to zavisi od izbora tacke odakle se krece. Losa strana ovog pristupa je sto sporo konvergira. --> VIDETI PRIMER 3.3

Pretraga Prvo najbolji:

- je osnova za razlicite algoritme pretrage grafa gde je svaki cvor stanje a svaka grana akcija. Kod ove pretrage za svaki cvor pamti se njegov roditelj (kao kod Dajkstre). Kako bi se izbegle beskonace petlje (da se nizu ista stanja) pamte se **dve liste stanja/cvorova:**
  1. lista zatvorenih stanja:
     - sva stanja za koja su vec ispitani svi susedi
  2. lista otvorenih stanja:
     - sva stanja koja su posecena ali jos nisu ispitani svi susedi tih stanja na listi

Na pocetku lista zatvorenih stanja je prazna a u listi otvorenih stanja nalazi se pocetni cvor. U svakom koraku iz liste otvorenih stanja uzima se cvor za **najboljom** f-jom cilja (*f(n)*) (pa zbog toga treba paziti preko koje strukture podataka se ovakva lista implementira) i ispituju se svi njegovi susedi. Ako se sused ne nalazi ni na jednoj listi, dodaj ga u otvorenu listu. Kad prodjes sve susede izbaci trenutni cvor iz otvorene u zatvorenu listu. Ako se naidje na ciljni cvor, algoritam se zaustavlja. Ako smo na primer iz liste otvorenih stanja uzeli cvor n i naidjemo na sused m koji se vec nalazi na bilo kojoj od dve liste, treba da ispitamo da li je postojeci put od pocetka do tog cvora m manji kada bismo isli od pocetnog cvora preko n do m, ako jeste onda kazemo da je novi roditelj cvora m cvor n. Pretraga Prvo najbolji **ne garantuje optimalno resenje** ali prethodna provera povecava tu sansu. Ako je **broj stanja i akcija konacan algoritam ce se zaustaviti**.

Ako za f(n) uzmemo da vraca dubinu cvora n onda se Prvo najbolji ponasa kao BFS, ako f(n) vraca zbir cena od polaznog cvora do njega, onda se ponasa kao Dajkstrin algoritam. Ono sto **razlikuje Prvo najbolji od pohlepnog algoritma** je sto je zahvaljujuci otvorenoj listi Prvo najbolji razmatra i alternative koje kod pohlepnog algoritma ne bi bile ispitane. Tamo gde bi pohlepni naisao na lokalni optimum ili plato, Prvo najbolji nece.

Algoritam A*: - predstavlja specijalan slucaj algoritma Prvo najbolji i uopstenje Dajkstrinog algoritma. I **potpun je i optimalan** je. Funkcija evaluacije algoritma A* ima sledeci oblik:

```
f(n) = g(n) + h(n)
f(n) - funkcija evaluacije za cvor n
g(n) - cena puta od polaznog cvora do cvora n
h(n) - procenjena cena najkraceg puta od cvora n do ciljnog cvora
```

Ponasanje i efikasnost algoritma A* zavisi upravo od kvaliteta heuristike odnosno procene. Tu procenu je cesto vrlo tesko naci, zna se uvek samo da je procena za ciljni cvor jednaka 0 a **nema opsteg postupka kojim bi se pronasla heuristika**. Posto je algoritam optimalan, kad god naidjemo na cvor cilja a cena puta preko nekog drugog cvora je veca od cene puta od trenutnog cvora, tu cenu puta azuriramo tako da je jednaka ovoj preko trenutnog cvora. **Razlika izmedju A* i Dajkstrinog algoritma je sto Dajkstrin algoritam uzima u obzir samo g(n)**, tj. cenu od pocetnog do n-tog cvora a nema procena tj. h(n).

Postupak A* je slican kao Dajkstrin. Ako naidjemo na cvor m koji je vec u otvorenoj ili zatvorenoj listi a nalazimo se u cvoru n. Onda pitamo da je put preko n do m kraci od vec postojeceg puta do m. Ako jeste, onda kazemo da je novi roditelj cvora m cvor n i azuriramo funkciju g(m) i prebacimo cvor m u otvorenu listu kako bismo mogli da azuriramo i njegove susede sa novom g(m) vrednoscu.

Svojstva algoritma A*:

1. Potpunost:
    - ako je skup cvorova i akcija konacan, i put izmedju dva cvora postoji, algoritam A* ce ga pronaci u konacnom broju koraka ma koliko losa heuristika bila.
2. Optimalnost:
    - algoritam A* je vratiti najkraci put od pocetnog do ciljnog cvora ukoliko je funkcija heuristike h(n) dopustiva.
    - **h(n) je *dopustiva* funkcija heuristike ako nikada ne precenjuje stvarno rastojanje od trenutnog do ciljnog cvora, tj. ako za svaki cvor vazi: h(n) <= h*(n)**
    - h*(n) - idealna heuristika (cena najkraceg puta od n do ciljnog cvora)
    - Funkcija h(n) je ***konzistentna*** ako za ciljni cvor n vazi h(n) = 0 i za svaka dva susedna cvora i, j vazi: c(i, j) + h(j) >= h(i)
    - c(i, j) - cena grane (moguce usmerene) izmedju i i j
    - **Ako je h(n) konzistentna onda je i dopustiva ali obrnutno ne mora da vazi!** VIDETI DOKAZ STRANA 33.

Lema:

- ako je h(n) konzistentna onda duz svakog puta kroz stablo pretrage f(n) nije opadajuce (monotona f-ja). *Dokaz:*

```
m - tekuci cvor, n - roditelj m
f(m) = g(m) + h(m) = g(n) + c(n, m) + h(m) >= (zbog konz.) g(n) + h(n) = f(n)
```

Lema:

- ako je h(n) konzistentna heuristika za niz cvorova redom proglasenim tekucim, f(n) daje neopadajuci niz. *Dokaz:*
  - u svakoj iteraciji biramo cvor iz liste otvorenih cvorova tako da f(n) ima najmanju vrednost, to znaci da cemo svaki sledeci put izabrati cvor cija je f(n) veca ili jednaka f(n) proslom cvoru.

Lema:

- do svakog cvora koji postane tekuci, nadjen je optimalan put

Slozenost algoritma A*: - slozenost je direktno zavisna od dobre procene funkcije heuristike. U najgorem slucaju slozenost A* eksponencijalna je broju cvorova na najkracem putu, tj. ista je kao BFS. Zato je vazno da f-ja heuristike bude blizu idealnoj ali da je nikada ne premasuje, tj. da za svaki cvor n vazi: `h(n) <= h*(n)`

Kada funkcija h(n) nije konzistentna, potrebno je proveravati i zatvorene cvorove.

Kada se radi o velikim mrezama, kao funkcija heuristike moze da se koristi **euklidsko rastojanje** ali da ne bismo racunali koren, mozemo da koristimo **Menhetn rastojanje** *(|x1 - x2| + |y1 - y2|),* ova heuristika je dopustiva jer nikada ne precenjuje rastojanje i garantuje optimalnost. S druge strane, ako su dozvoljeni i **dijagonalni potezi**, **Menhetn rastojanje nije dopustiva heuristika** jer moze da preceni rastojanje. Za dijagonalne poteze, mozemo da koristimo **Cebisevljevo rastojanje** *max(|x2 - x1|, |y2 - y1|)*

Algoritam A* najcesce je implementiran tako da je **otvorena lista binarni min-heap** kako bi u svakom koraju lako pronalazili onaj sa najmanjom vredoscu f(n), tada je slozenost **O(log|V|)**. **Zatvorena lista** implementirana je kao **hes tabela** pa je provera da li je cvor tu i dodavanje O(1). Treba obezbediti da sva aritmetika bude celobrojna. **Najgori slucaj za A*** je kada **nema puta izmedju pocetnog i ciljnog cvora**, pa pre pokretanja algoritma **treba proveriti da li pripadaju istoj komponenti povezanosti**.

GENETSKI ALGORITMI:

Genetski algoritmi koriste metaheuristike za resenje problema. ***Metaheuristke*** predstavljaju heuristike koje nisu specijalizovane za resavanje konkretnog problema vec se njima moze resavati skup problema koji na prvi pogled ne moraju da budu slicni. To se postize prilagodjavanjem odredjenih parametara. Losa strana metaheuristika je sto **ne garantuju pronalazak svih resenja** a odatle sledi da pronadjeno **resenje ne mora da bude optimalno** (dobijeno resenje moze biti lokalni maksimum a ne globalni). Oni mogu dati skup resenja sto je cesto dovoljno dobro. Genetski algoritmi koriste se za resavanje problema optmizacije ili pretrage (za NP-teske probleme...). Vrlo su vremenski i memorijski zahtevni ali su pogodni za paralelizaciju.

Hromozom/genotip -> reprezentacija jedinke

Cilj genetskog algoritma je naci vrednost za koju funkcija cilja dostize maksimum. Funkcija cilja ne mora da bude zadata eksplicitno vec implicitno kroz veci broj kriterijuma; ne mora da bude diferencijabilna ni neprekidna. Pored funkcije cilja postoji i **funkcija prilagodjenosti** i ona **oznacava kvalitet jedinke**. Funkcija cilja i

funkcija prilagodjenosti cesto se poklapaju ali i ne moraju. Na primer, funkcija prilagodjenosti moze jedinkama koje prelaze odredjenu vrednost funkcije cilja dodetljivati vrednost 1 a onima koje ne 0. Treba biti obazriv jer tako moze doci do prebrze konvergencije.

Opsti postupak genetskog algoritama:

- ulaz: podaci koji odredjuju funkciju cilja i parametre algoritma
- izlaz: najkvalitetnija jedinka

1. kreiraj pocetnu generaciju
2. izracunaj funkciju prilagodjenosti svake jedinke
3. petlja:
    - izaberi iz generacije skup jedinki za reprodukciju
    - ukrstanjem i mutacijom napravi nove jedinke
    - od novih i starih jedinki kreiraj novu generaciju
    - kraj petlje kada je neki od uslova ispunjen (funkcija cilja dostigla maksimum - pronadjeno resenje, doslo se do odredjenog broja generacija, funkcija priladjenosti pozvana odredjen broj puta, vrednost prilagodjenosti najbolje jedinke nije se popravila kroz nekoliko generacija, kombinacija nekih od ovih uslova)
4. izaberi najkvalitetniju jedinku u najnovijoj generaciji

Jedinke se mogu predstaviti na razlicite nacine: nizom bitova, matricama, stringovima... Bitno je da nad tom reprezentacijom mozemo da upotrebimo operatore ukrstanja i mutacije. Nasi operatori treba da zadovoljavaju uslov da **nove jedinke** koje stvaraju budu **samo one koje mogu da budu nase resenje**. Ili, ako to ne ispunjavaju onda treba da postoje mehanizmi kojima se takve jedinke ispravljaju.

Ako je jedinka predstavljena nizom n bitova onda je potrebno uspostaviti vezu preslikavanja (koja nije bijektivna) izmedju reprezentacije i realnog intervala [a, b] gde se resenje moze naci:

Broju X sa binarnom reprezentacijom a = 00...0, b = 11...1 odgovara realna reprezentacija:

```
X = a + (x * (b - a)) / 2^n - 1
```

Realnom broju X odgovara binarna reprezentacija:

```
X = floor((X - a) * (2^n - 1) / b - a)
```

Funkcija prilagodjenosti daje ocenu kvaliteta jedinke. Treba da bude definisana za sve jedinke, da bude brza za izracunavanje i da daje dobru sliku kvaliteta jedinke. Sto je funkcija prilagodjenosti veca to je veca sansa da ce ta jedinka biti izabrana za reprodukciju. Vrednost funkcije raste sa porastom broja generacija.

Detaljan opis postupka izvrsavanja genetskog algoritma:

1. Inicijalizacija:

- **broj jedinku** u generaciji je **najcesce fiksan** i definise sa kao parametar algoritma. Pocetna generacija dobija se najcesce slucajnim izborom a ponekad se eksplicitno u nju ubacuju jedinke za koje se veruje da ce biti blizu u prostoru pretrage gde se resenje nalazi (one navode nove generacije).

2. Selekcija:

- predstavlja **izbor jedinki iz tekuce generacije koje ce ucestovati u kreiranju nove generacije**. Najcesce se oslanja na funkciju prilagodjenosti gde se mogu birati samo najbolje jedinke ili gde se vrednost funkcije prilagodjenosti oslikava u verovatnocu da jedinka bude izabrana. Na taj nacin losije jedinke imaju sansu da ucestvuju u pravljenju nove generacije i tako se cuva **genetska raznolikost** i sprecava prerana konvergencija ka, najcesce, lokalnom ekstremumu. Postoje **dva tipa selekcije**:

    1. **ruletska**:

        - vrednosti funkcije prilagodjenosti odredjuje sansu da jedinka bude izabrana. Ista jedinka moze vise puta biti izabrana cime se znacajno smanjuje efikasnost algoritma.

    2. **turnirska**:

        - od skupa svih jedinki bira se **skup k jedinki** (k < |populacija|). Iz novog skupa **moze se birati jedinka sa najvecom prilagodjenosti ili se skup moze sortirati i i-ta jedinka u sortiranom skupu bira se sa verovatnocom:**

            ```
            p(1 - p)^(i - 1)
            p -> verovatnoca (drugi parametar turnirske selekcije)
            ```

        - Prethodno izabranim jedinkama moze biti **zabranjeno da opet budu izabrane**. Ako je velicina novog skupa **k = 1** onda se turnirska selekcija zasniva na **random izboru jedinki iz cele generacije**; ako je **p = 1** onda se **u novom skupu bira najprilagodjenija jedinka**.

3. Reprodukcija:

- **od selektovanih jedinki biraju se dve koje ce se sa nekom sansom (najcesce od 0.6 do 0.9) ukrstiti i dati jedno ili dva nova deteta**. Ukrstanje se moze realizovati na razlicite nacine. Ako jedinke reprezentujemo nizom bitova onda mozemo koristiti **visepoziciono ukrstanje** kod kojeg moze nastati **jedno ili dva deteta** i kod kojeg se oba roditelja na istom mestu/mestima dele i onda unakrsnim kombinovanjem nastaje dete. Pored visepozicionog ukrstanja postoji i **uniformno ukrstanje** kod kojeg **uvek nastaju dva deteta**. Dete nastaje tako sto se ide bit po bit kroz roditelje i za svaku poziciju sa odredjenom sansom (najcesce pola-pola) se iz jednog roditelja bit daje detetu a drugom detetu ide bit od drugog roditelja.

4. Mutacija:

- nakon ukrstanja sledi mutacija kod koje se definise sansa (najcesce **ispod 1% da se odredjeni bit** (ako je takva reprezentacija jedinke) **promeni**). **Mutacija omogucava da se pobegne od lokalnih ekstremuma**. Ako je, na primer, u jednoj generaciji odredjeni gen svih jedinki isti onda ne postoji sansa da se istrazi prostor pretrage gde taj gen nije takav. Time potencijalni globalni maksimum nikada nece biti ispitan. Koriscenjem mutacije ova pojava moze da se, potencijalno, izbegne. Ako je **sansa mutacije prevelika** onda je usmeravanje pretrage preslabo pa ona **lici na slucajnu pretragu**. Ako ne postoji onda se lako upada u lokalni ekstremum.

5. Politika zamene generacije:

- predstavlja nacin formiranja nove generacije jedinki. Postoje **dva osnovna pristupa**:
  1. **generacijski genetski algoritmi**:
     - biraju dovoljno predstavnika iz tekuce generacije da se njihovim ukrstanjem i mutiranjem kreira cela nova generacija koja ce zameniti staru.
  2. **algoritmi stabilnog stanja**:
     - cim se izaberi par roditelja vrsi se ukrstanje i mutacija i nova/e jedinke se ubacuju u zavisnosti od **politike zamene** koja moze da bude:
       - **zamena najgorih**:
         - nove jedinke zamenjuju najmanje prilagodjene u generaciji
       - **nasumicna zamena**:
         - random se biraju predstavnici iz stare generacije koji ce biti zamenjeni
       - **inverzna turnirska selekcija**:
         - umesto najprilagodjenijih biraju se najneprilagodjeniji

Elitizam se moze koristiti u obe ove politike i oznacava **pojavu kada se odredjene jedinke** (ili jedna) koje su po necemu najbolje **uvek stite od nestajanja**. Time se obezbedjuje da se neke kvalitetne osobine ne izgube u toku evolucije

Ponekad se genetskim algoritmom biraju paremetri drugih genetskih algoritama. **Od parametara u velikoj meri zavisi koliko ce sam algoritam biti dobar**. Oni ne moraju da budu staticki odredjeni vec se mogu menjati za vreme izvrsavanja algoritma.

TODO: Zavrsiti probleme od Skakaca do kraja

Resenje problema # PRETRAGA:

Lojdova slagalica:

- sastoji se od 4x4 polja koja su popunjena brojevima od [1, 15] a poslednje polje je prazno. To se moze predstaviti listom [1, 2, ... , 15, *] ako su svi brojevi na mestu. Cilj igre je dovesti sve brojeve u tu konfiguraciju. Brute force pristup bio bi da za svako stanje razmatramo sva moguca sledeca, pa za to sledece sva moguca sledeca i tako do kraja gde je garantovano pronalazenje resenja. Pokazalo se da je za neke pocetne konfiguracije slagalicu moguce sloziti u najvise 80 koraka pa bi to bilo neko gornje ogranicenje. Odatle sledi da je potrebno ispitati 2^80 (ako posmatramo prazno polje, njega u svakom stanju mozemo da pomerimo u tacno dva smera) mogucnosti sto je prakticno neizvodivo. Jedno od resenje jeste da se nekako _navodi* pretraga. Moguce navodjenje bilo bi da se igraju samo potezi koji vode do stanja koje je blize resenju. Da je sledece stanje koje razmatramo blize od trenutnog mozemo da izracunamo kao **sumu udaljenosti svih polja od mesta na kom treba da budu** (to je heuristika). Pakazace se da ovo ne vodi do resenja jer neka stanja nikada nemaju sledece stanje koje je blize resenju.

Problemi pretrage najpogodnije se predstavljaju **grafovima** gde su **cvorovi stanja** a **grane akcije**. Mogu biti **usmereni** (sah) ili **neusmereni** (Lojdova slagalica, putanje izmedju gradova...). Graf je neusmeren ako se akcijom koja ima istu cenu moze iz cvora

A doci u B i iz cvora B doci u A. Obilaskom grafa formira se **stablo pretrage**. S obzirom da se neki cvor u grafu moze vise puta obici, u stablu se isti cvor moze vise puta javiti. Zbog toga mogu postojati **beskonacna stabla pretrage za konacne grafove**.

Algoritmi pretrage:

Resenje problema pretrage je niz koraka (akcija) koje treba izvesti da bi se iz pocetnog doslo u zavrsno stanje. U zavisnosti od samog tipa problema, koriste se razliciti algoritmi pretrage ali svi treba da zadovoljavaju sledece:

1. Potpunost:
    - garantuje da ce algoritam **naci resenje ako resenje postoji**. Ova osobina je **pozeljna** ali nekad **nije neophodna**. Naime, neki veoma teski problemi se mogu resiti koriscenjem heuristika koje nisu potpune ali ce u prosecnom slucaju mnogo pre dati resenje od algoritama koji su potpuni.
2. Optimalnost:
    - nadjeno **resenje uvek ima najmanju cenu**. Cena se racuna kao zbir cena akcija koje se preduzimaju. Nekad cena moze da se posmatra kao vremenska i prostorna slozenost algoritma pri cemu optimalan ima najmanju cenu. Optimalnost, kao i potpunost, **nekad nije neophodna** jer odredjeni neoptimalni algoritmi u prosecnom slucaju daju resenja koja su bliska optimalnim ali ih nalaze mnogo pre.
3. Vremenska i prostorna slozenost:
    - razmatraju se kao i kod svih drugih algoritama i vazne su i najgora i prosecna slozenost.

Algoritmi pretrage mogu biti **informisani** ili **neinformisani**. Ukoliko postoje informacije o kvalitetu pojedinacnih stanja onda se te informacije mogu koristiti za navodjenje pretrage i tada se radi o informisanim algoritmima. Jedan od tih algoritama je i **A***. Ukoliko te informacije nisu poznate onda se pretraga ne moze navoditi vec je potrebno smisliti neko sistematicno (putpuno) pretrazivanje (u sirinu/dubinu npr.). Lojdova slagalica mora se resiti neinformisanim algoritmom jer pored cene svake akcije i neposredno dostupnih stanja, ni jedna druga informacije nije dostupna. Sa druge strane, rastojanje izmedju gradova moze se naci informisanim algoritmom koje bi koristilo vazdusno rastojanje koje lako moze da se izracuna kao euklidsko rastojanje izmedju dva cvora u grafu.

NEINFORMISANI ALGORITMI PRETRAGE:

Neinformisani algoritmi pretrage **nemaju dodatne informacije koje bi navodile pretragu** i tako je znacajno skratile. Umesto toga, koristi se sistematicno ispitivanje svih mogucih stanja. Neki od tih algoritama su DFS i BFS (ili bektreking kao modifikacija DFS-a). Lavirint je klasican problem koji se resava ovim algoritmima. **Depth first search** algoritam ispituje sto dalje cvorove od polaznog. Nerekurzivna implementacija koristi stek, odnosto LIFO strukturu, i kada dodje do cvora odakle ne moze dalje, onda skida poslednji element sa steka i od njega nastavlja pretragu ali pre toga svako poseceno polje oznaci tako da ne bismo upali u beskonacnu petlju. Kada dodjemo do ciljnog polja onda samo procitamo u obrnutom redu stek i dobili smo putanju. Algoritam staje kada je stek prazan. DFS prestavlja *pracenje desne strane* za pronalazak izlaza iz lavirinta.

Problem n dama: - kako na sahovskoj tabli dimenzije n * n postaviti n dama tako da se nikoje dve dame ne napadaju. Ovo je tipican primer neinformisane pretrage pomocu

modifikovanog DFS-a koji koristi bektreking.

Bektreking se sastoji od nadogradjivanja parcijalnog resenja. U problemu n dama, pocetno parcijalno resenje je prazna tabla. Ovo parcijalno resenje se u svakom koraku nadogradjuje dodavanjem dame na neko od slobodnih polja. Posto dve dame ne mogu da se nalaze u istoj vrsti, razmatramo kolonu po kolonu. Glavna osobina bektrekinga je da ce prekinuti pretragu stabla cim se dogodi da parcijalno resenje ne moze da bude validno. U ovom slucaju to ce biti cim se neke dve dame napadaju. Tada se algoritam vraca na prvo stanje u kojem se ove dve dame nisu napadale (kada je parcijalno resenje moglo da vodi do resenja) i pritom odseca deo stabla u kom je prethodno bio (onaj gde je parcijalno resenje postalo nevalidno).

Breadth-first search je neinformisani algoritam pretrage koji uvek daje ciljni cvor koji je na najmanjem rastojanju od polaznog. BFS u red, FIFO strukturu, dodaje sve cvorove susedne trenutnom, zatim uzima prvi iz reda i dodaje na kraj sve njemu susedne i sve tako dok se ne dodje do ciljnog cvora ili praznog reda. DFS algoritam pogodniji je kada je usmeravanje pretrage moguce.

Vremenska slozenost i DFS-a i BFS-a je O(|V| + |E|). Prostorna O(|E|)

Dajkstrin algoritam koristi se za pronalazak najkracih puteva u grafu **koji nema negativne cene grana**. Po zavrsetku algoritma dobija se najkrace rastojanje do ciljnog cvora ali se algoritam lako moze modifikovati da se dobije najkrace rastojanje do svih cvorova:

1. Na pocetku, svi cvorovi grafa se ubace u skup Q i pritom se cena puta od pocetnog cvora do svakog stavi na +INF (sem za pocetni cvor gde se postavi na 0). Roditelj svakog cvora na pocetku je takodje nepoznat.
2. Zatim se iz skupa Q uzima cvor n koji ima najmanje rastojanje od pocetnog (u prvom koraku to je pocetni cvor). Ako je taj cvor ciljni onda se prekida algoritam i rekonstruise putanja preko roditelja. Ako to nije ciljni cvor, onda se ispituje svaki svaki njegov sused m. Ako je prethodno rastojanje od pocetnog cvora do m vece od rastojanja od pocetnog cvora preko n pa do m onda se postavlja novo rastojanje od pocetnog cvora do m i ono je jednako rastojanju od pocetnog preko n do m.

Invarijanta petlje je da se za sve cvorove koji nisu u Q zna najkrace rastojanje od pocetnog. Slozenost Dajkstrinog algoritma, gde je skup Q implementiran preko povezane liste je O(|V|^2). Ako graf ima mnogo manje grana od |V|^2 onda se skup moze implementirati preko Min-Heap-a i tada je slozenost O((|V| + |E|) * log(|V|)).

INFORMISANI ALGORITMI PRETRAGE:

Informisana pretraga naziva se jos i **heuristicka pretraga** i ona u svakom stanju, pored informacija o sledecim mogucim akcijama, ima informacije i o tome koja akcija moze da dovede do stanja koja vise obecavaju. Ta informacija moze da bude neka mera *kvaliteta* stanja, koja ne mora da bude konstantna (moze da bude drugacija za isto stanje u zavisnosti o toga u kom smo stanju trenutno ili u kom je stanju celokupna pretraga). Heuristika "navodi" nas algoritam ka brzem pronalasku resenja. Taj *kvalitet* stanja cesto nije egzaktan vec prestavlja neku procenu. **Funkcija evaluacije** ocenjuje kvalitet stanja. Kazemo da je f(n) - funkcija evaluacije za stanje n.

Pohlepni algoritmi u svakom koraku uzimaju najbolje lokalno resenje u nadi da ce tako doci do najboljeg globalnog resenja. Pokazace se da vrlo cesto ovo nije slucaj i da se cak u nekim sutuacijama nece ni doci do resenje iako ono postoji. U primeru sa

pronalaskom najkraceg puta izmedju gradova, dodatna informacija koju bi pohlepni algoritam koristio bila bi vazdusno rastojanje izmedju gradova. Pretpostavljamo da znamo vazudusna rastojanja iz svakog do svakog grada, tada bi pohlepni algoritam iz grada n za sledeci grad birao onaj sa najmanjim vazdusnim rastojanjem a cena bi bila jednaka pravoj kopnenoj ceni od pocetnog grada do grada n i najmanjoj vazusnoj ceni od grada n do sledeceg grada. Ni u ovom primeru iz knjige ne dobija se optimalno resenje. **Pohlepni algoritmi** cesto su veoma efikasni i laki za implementaciju ali **ne garantuju optimalnost pa ni potpunost**. Lojdovu slagalicu nije moguce resiti pohlepnim algoritmom jer oni kao kvalitet stanja najcesce uzimaju Menhetn rastojanje svakog polja od ciljnog, a cesto se desava da slagalica ima takvu konfiguraciju da pohlepni algoritam ne moze da pomeri ni jedno polje jer se kvalitet tog stanja smanjuje tj. on se odaljava od ciljnog (odnosno stiglo se do lokalnog maksimuma). Tada se pohlepni algoritam glavi jer on nikad ne sme da uzima lokalni minimum koji pogorsava globalni minimum do kog se doslo.

Pohlepni algoritmi se mogu koristiti u matematickoj optimizaciji i tada se nazivaju **algoritmi penjanja uzbrdo**. Oni u svakom koraku biraju susedna dopustiva resenje koja imaju najvise vrednosti funkcije cilja. Oni imaju neke slabosti:

1. Opasnost od lokalnih maksimuma:
    - postoji kada svi dopustivi susedi imaju manju vrednost funkcije cilja od trenutne, tj. stiglo se u tacku odakle se ne moze dalje ali vrednost funkcije cilja u toj tacki nije max globalna vec je samo max lokalna. Odavde pohlepni algoritam nema kuda pa staje i vraca pogresno resenje.
2. Neefikasnot u slucaju grebena:
    - greben prestavlja usku stazu koja opada ili raste duz nekog pravca. Tada ce algoritam penjanja uzbrdo biti neefikasan jer umesto da se pravolinijske penje on ce ici u cik-cak i tako biti mnogo sporiji.
3. Opasnost od platoa:
    - nastaje kada oblast prostora pretrage ima konstantu vrednost funkcije cilja. Tada pohlepni algoritam ne zna sta dalje da radi.

Ovi problemi mogu se resiti nekim varijacijama penjanja. Tako **stohasticko penjanje** ne bira uvek suseda sa najboljom vrednoscu funkcije cilja vec ce verovatnoca da on bude izabran biti proporcionalna toj vrednosti. Druga varijacija je **penjanje uzbrdo sa slucajnim resetovanjem** kod koga se nakon pronalaska nekog maksimuma algoritam ponovo pokrece iz slucajno izabrane tacke. Verovatnoca da ce se tako doci do globalnog maksimuma se priblizava 1 sto vise puta pokrenemo penjanje.

Najstrmiju spust (**gradijentni spust**) je samo -funkcija cilja.

Minimalna/maksimalna tacka ne mora da bude bas egzaktna vec je to ona iz koje bi pomeraj bio manji od nekog definisanog *epsilon*. Kada se spustamo moramo da pazimo da duzina koraka za koji se spustamo bude optimalna, tj. moramo da pazimo da korak **nije predugacak** jer ce doci do divergencije (proci cemo trazenu vrednost) i moramo da pazimo da korak **nije prekratak** jer cemo se mnogo sporo spustati. To mozemo da obezbedimo tako sto uzmemo da je taj korak

```
Lambda_n = 1 / (n + 1)
```

Sledeci korak:

```
a_(n+1) = a_n - Lambda_n * Delta * f(a_n)
```

Ni gradijentni spust ne daje garanciju da ce doci do globalnog minimuma vec to zavisi od izbora tacke odakle se krece. Losa strana ovog pristupa je sto sporo konvergira. --> VIDETI PRIMER 3.3

Pretraga Prvo najbolji:

- je osnova za razlicite algoritme pretrage grafa gde je svaki cvor stanje a svaka grana akcija. Kod ove pretrage za svaki cvor pamti se njegov roditelj (kao kod Dajkstre). Kako bi se izbegle beskonace petlje (da se nizu ista stanja) pamte se **dve liste stanja/cvorova:**
  1. lista zatvorenih stanja:
     - sva stanja za koja su vec ispitani svi susedi
  2. lista otvorenih stanja:
     - sva stanja koja su posecena ali jos nisu ispitani svi susedi tih stanja na listi

Na pocetku lista zatvorenih stanja je prazna a u listi otvorenih stanja nalazi se pocetni cvor. U svakom koraku iz liste otvorenih stanja uzima se cvor za **najboljom** f-jom cilja (*f(n)*) (pa zbog toga treba paziti preko koje strukture podataka se ovakva lista implementira) i ispituju se svi njegovi susedi. Ako se sused ne nalazi ni na jednoj listi, dodaj ga u otvorenu listu. Kad prodjes sve susede izbaci trenutni cvor iz otvorene u zatvorenu listu. Ako se naidje na ciljni cvor, algoritam se zaustavlja. Ako smo na primer iz liste otvorenih stanja uzeli cvor n i naidjemo na sused m koji se vec nalazi na bilo kojoj od dve liste, treba da ispitamo da li je postojeci put od pocetka do tog cvora m manji kada bismo isli od pocetnog cvora preko n do m, ako jeste onda kazemo da je novi roditelj cvora m cvor n. Pretraga Prvo najbolji **ne garantuje optimalno resenje** ali prethodna provera povecava tu sansu. Ako je **broj stanja i akcija konacan algoritam ce se zaustaviti**.

Ako za f(n) uzmemo da vraca dubinu cvora n onda se Prvo najbolji ponasa kao BFS, ako f(n) vraca zbir cena od polaznog cvora do njega, onda se ponasa kao Dajkstrin algoritam. Ono sto **razlikuje Prvo najbolji od pohlepnog algoritma** je sto je zahvaljujuci otvorenoj listi Prvo najbolji razmatra i alternative koje kod pohlepnog algoritma ne bi bile ispitane. Tamo gde bi pohlepni naisao na lokalni optimum ili plato, Prvo najbolji nece.

Algoritam A*: - predstavlja specijalan slucaj algoritma Prvo najbolji i uopstenje Dajkstrinog algoritma. I **potpun je i optimalan** je. Funkcija evaluacije algoritma A* ima sledeci oblik:

```
f(n) = g(n) + h(n)
f(n) - funkcija evaluacije za cvor n
g(n) - cena puta od polaznog cvora do cvora n
h(n) - procenjena cena najkraceg puta od cvora n do ciljnog cvora
```

Ponasanje i efikasnost algoritma A* zavisi upravo od kvaliteta heuristike odnosno procene. Tu procenu je cesto vrlo tesko naci, zna se uvek samo da je procena za ciljni cvor jednaka 0 a **nema opsteg postupka kojim bi se pronasla heuristika**. Posto je algoritam optimalan, kad god naidjemo na cvor cilja a cena puta preko nekog drugog cvora je veca od cene puta od trenutnog cvora, tu cenu puta azuriramo tako da je jednaka ovoj preko trenutnog cvora. **Razlika izmedju A* i Dajkstrinog algoritma je sto Dajkstrin algoritam uzima u obzir samo g(n)**, tj. cenu od pocetnog do n-tog cvora a nema procena tj. h(n).

Postupak A* je slican kao Dajkstrin. Ako naidjemo na cvor m koji je vec u otvorenoj ili zatvorenoj listi a nalazimo se u cvoru n. Onda pitamo da je put preko n do m kraci od vec postojeceg puta do m. Ako jeste, onda kazemo da je novi roditelj cvora m cvor n i azuriramo funkciju g(m) i prebacimo cvor m u otvorenu listu kako bismo mogli da azuriramo i njegove susede sa novom g(m) vrednoscu.

Svojstva algoritma A*:

1. Potpunost:
    - ako je skup cvorova i akcija konacan, i put izmedju dva cvora postoji, algoritam A* ce ga pronaci u konacnom broju koraka ma koliko losa heuristika bila.
2. Optimalnost:
    - algoritam A* je vratiti najkraci put od pocetnog do ciljnog cvora ukoliko je funkcija heuristike h(n) dopustiva.
    - **h(n) je *dopustiva* funkcija heuristike ako nikada ne precenjuje stvarno rastojanje od trenutnog do ciljnog cvora, tj. ako za svaki cvor vazi: h(n) <= h*(n)**
    - h*(n) - idealna heuristika (cena najkraceg puta od n do ciljnog cvora)
    - Funkcija h(n) je ***konzistentna*** ako za ciljni cvor n vazi h(n) = 0 i za svaka dva susedna cvora i, j vazi: c(i, j) + h(j) >= h(i)
    - c(i, j) - cena grane (moguce usmerene) izmedju i i j
    - **Ako je h(n) konzistentna onda je i dopustiva ali obrnutno ne mora da vazi!** VIDETI DOKAZ STRANA 33.

Lema:

- ako je h(n) konzistentna onda duz svakog puta kroz stablo pretrage f(n) nije opadajuce (monotona f-ja). *Dokaz:*

```
m - tekuci cvor, n - roditelj m
f(m) = g(m) + h(m) = g(n) + c(n, m) + h(m) >= (zbog konz.) g(n) + h(n) = f(n)
```

Lema:

- ako je h(n) konzistentna heuristika za niz cvorova redom proglasenim tekucim, f(n) daje neopadajuci niz. *Dokaz:*
    - u svakoj iteraciji biramo cvor iz liste otvorenih cvorova tako da f(n) ima najmanju vrednost, to znaci da cemo svaki sledeci put izabrati cvor cija je f(n) veca ili jednaka f(n) proslom cvoru.

Lema:

- do svakog cvora koji postane tekuci, nadjen je optimalan put

Slozenost algoritma A*: - slozenost je direktno zavisna od dobre procene funkcije heuristike. U najgorem slucaju slozenost A* eksponencijalna je broju cvorova na najkracem putu, tj. ista je kao BFS. Zato je vazno da f-ja heuristike bude blizu idealnoj ali da je nikada ne premasuje, tj. da za svaki cvor n vazi: `h(n) <= h*(n)`

Kada funkcija h(n) nije konzistentna, potrebno je proveravati i zatvorene cvorove.

Kada se radi o velikim mrezama, kao funkcija heuristike moze da se koristi **euklidsko rastojanje** ali da ne bismo racunali koren, mozemo da koristimo **Menhetn rastojanje** *(|x1 - x2| + |y1 - y2|)*, ova heuristika je dopustiva jer nikada ne precenjuje rastojanje i

garantuje optimalnost. S druge strane, ako su dozvoljeni i **dijagonalni potezi**, **Menhetn rastojanje nije dopustiva heuristika** jer moze da preceni rastojanje. Za dijagonalne poteze, mozemo da koristimo **Cebisevljevo rastojanje** *max(|x2 - x1|, |y2 - y1|)*

Algoritam A* najcesce je implementiran tako da je **otvorena lista binarni min-heap** kako bi u svakom koraju lako pronalazili onaj sa najmanjom vredoscu f(n), tada je slozenost **O(log|V|)**. **Zatvorena lista** implementirana je kao **hes tabela** pa je provera da li je cvor tu i dodavanje O(1). Treba obezbediti da sva aritmetika bude celobrojna. **Najgori slucaj za A*** je kada **nema puta izmedju pocetnog i ciljnog cvora**, pa pre pokretanja algoritma **treba proveriti da li pripadaju istoj komponenti povezanosti**.

IGRANJE STRATESKIH IGARA:

Programi za igranje strateskih igara zasnovani su na algoritmima pretrage a u skorije vreme i na masinskom ucenju.

Postoje dve strategije izbora poteza:

1. ***Minimaks***:
    - vrsi se **pretrazivanje** stabla igre sa odredjenom f-jom evaluacije i eliminacijom nelegalnih poteza dolazeci do najboljeg poteza. U savrsenom svetu, kreiranjem celog stabla igre, do zavrsnih stanja, zaista bismo u svakom koraku mogli da biramo najbolji potez. Medjutim, u praksi to nije moguce zbog samog broja svih mogucih stanja (kako se krecemo kroz dubinu stabla, vrlo brzi dolazi do kombinatorne ekspolozije). Zbog toga, najcesce se ova tehnika implementira tako da pretrazuje stablo samo nekoliko poteza unapred. To se implementira raznim pametnim algoritmima pretrage koje navodi heuristika i dobrom ali jednostavnom f-jom evaluacije.
2. ***Koristeci tabelu***:
    - potez se bira konsultujuci unapred pripremljenu tabelu poteza. Ova tabela sastoji se iz dve kolone: u jednoj koloni su sva stanja u kojima se igrac moze nalaziti a u drugoj akcije koje igrac moze da preduzme u tim stanjima sa ocenom kvaliteta akcija/stanje. Ukoliko postoji, ova tabela daje siguran put do pobede. Problem nastaje zbog velikog broja stanja i vrlo brzi ova tehnika nije upotrebiva. Prva ovakva tabela kreirana je koriscenjem ***retrogradne analize***. Nakon toga, 2012. kreirana je i ***Lomonosov*** tabela.

Druga tehnika zahteva mnogo memorije a relativno malo racunske moci, dok prva tehnika zahteva ogromnu racunsku moc ali ne i mnogo memorije.

Legalni potezi i stablo igre:

Pravila konkretne igre odredjuju ***legalne pozicije*** a svaki dozvoljen potez iz legalne pozicije zove se ***legalna akcija***. Prostor stanja igre moze se predstaviti grafom ciji cvorovi su legalne pozicije a grane legalne akcije iz tig pozicija. Taj graf je najcesce usmeren (pesak u sahu moze da ide napred ali ne i nazad). **Stablo igre** je stablo u cijim cvorovima su legalne poziciji i sva deca cvora su stanja u koja se legalnim akcijama iz tog cvora moze doci do te dece. **Potpuno stablo igre** je stablo u cijem je korenu pocetna pozicija igre a listovi su zavrsne pozicije igre i svakom listu pridruzeno je znacenje - izubio, pobedio, nereseno.

Otvaranje:

Umesto da na samom pocetku partije program kreira stablo pretrage, rana faza partije zasniva se na koriscenju ***knjiga otvaranja***. Ova faza partije **traje 10ak poteza** ali se po potrebi moze pre prekinuti. Tokom otvaranja, program konsultuje bazu znanja (knjigu otvaranja) i bez mnogo racunanja odigrava poteze. Ove knjige su dobre jer je pocetak partije dovoljno podlozan dubljoj analizi pa je moguce stvoriti odredjene sablone poteza koji ce program dovesti u sto povoljniji polozan u sredisnjici. Ako protivnik odigra neki neuobicajen potez, program ce najverovatnije morati da napusti ovaj rezim rada i krene sa kreiranjem stabla pretrage. Obicno neki neuobicajeni potezi vise smetaju onome ko ih odigra nego sto programu smeta promena nacina biranja poteza. Knjiga otvaranja moze biti ***staticka*** (sadrzi konacan broj varijanti legalnih pozicija i legalnih akcija iz njih) ili se moze ***prosirivati*** tokom rada programa.

Sredisnjica:

Ovaj deo partije zasniva se na prvoj strategiji biranja poteza - Minimaks algoritmu. Tada dolazi do formiranja stabla igre koji predstavlja samo mali delic kompletnog stabla igra. Pretragom ovog stabla nastaje stablo pretrage. Bitno je da f-ja evaluacije bude sto bolja i brza. Ova f-ja primenjuje se samo na oredjene cvorove umesto na celo stablo. Kvalitet pretka nekog cvora racuna se na osnovu f-je evaluacije potomka. Pored f-je evaluacije bitno je navodjenje same pretrage koje se vrsi heuristikama.

Funkcija evaluacije ne koristi prethodna kvalitet pozicija iz kojih se doslo u trenutnu nije gleda pozicije u koje se moze stici iz trenutne. Ona predstavlja ocenu kvaliteta stanja i na osnovu nje program donosi sve odluke. Moze se reci da je ona *mozak* programa i vrlo je vazno da se lako i brzo racuna s obzirom da ce se cesto racunati. Funkcija evaluacije preslikava se u segment [-M, M] gde se -M dodeljuje zavrsnim cvorovima u kojima je gubitnik program a M se dodeljuje zavrsnim cvorovima u kojima je pobednik program. U igrama nulte sume smisao f-je evaluacije u igri za dva igraca je suprotan - ono sto je najbolja pozicija za jednog najgora je za drugog igraca. ***Trovrednosna*** f-ja evaluacije racuna se samo za zavrsna stanja i moze oznacavati samo tri stvari: pobedu prvog igraca, pobedu drugog igraca ili nereseno. Da bi se ova f-ja koristila neophodno je napraviti kompletno stablo pretrage pa je ova f-ja gotovo neupotrebiva.

Algoritam Minimaks:

Algoritam Minimaks glavni je deo Senonove A seme za izbor poteza. Glavni parametri ovog algoritma su **dubina pretrazivanja** (skoro sigurno nije moguce formirati kompletno stablo pretrage odjednom pa ovaj parametar oznacava do koje *dubine* ce ono biti razmatrano) i **funkcija evaluacije**. Algoritam ce u svakom cvoru doneti odluku samo na osnovu ova dva parametra ne razmatrajuci prethodno stanje iz kojeg je dosao u trenutno (to znaci da ako je u prethodnom potezu program izabrao potez koji ce dovesti do saha u 3 poteza, u sledecem potezu nece uopste gledati da juri taj sah koji bi sada bio u dva poteza vec ce doneti odluku samo na osnovu trenutnog stanja i funkcije evaluacije) - ovo se naziva ***efekat horizonta***. Ovaj algoritam *bira* potez na osnovu funkcije evaluacije koja se racuna samo za onu decu trenutnog cvora koja su na datoj dubine. Zatim se vrednosti dece propagiraju na gore na sledeci nacin: ako je program na potezu onda se u trenutno stanje propagira **max** vrednost f-je evaluacije iz skupa dece; ako je protivnik na potezu onda se propagira **min** vrednost. Ovo se naziva ***minimaksizacija***

(postupak naizmenicnog minimizovanja i maksimizovanja).
https://en.wikipedia.org/wiki/Minimax#/media/File:Minimax.svg

Algoritam Alfa-beta:

Alfa-beta algoritam predstavlja modifikaciju Minimaks algoritma koji koristi **alfa i beta odsecanja** kao heuristiku kako bi ubrzala Minimaks algoritam. Ova odsecanja oznacavaju da odredjeni delovi stabla pretrage uopste nece morati da budu istrazeni i evaluirani. Alfa i beta odsecanja su analogna a razlika je samo u tome da li je program ili protivnik na potezu.

- recimo da je program na potezu (alfa odsecanja), to znaci da on zeli da za trenutno stanje i dubinu nadje MAX vrednost dece. Oznacimo sa a1, a2, ... , an sve vrednosti f-je evaluacije potomaka odmah ispod stanja u kome se program nalazi (stanja za koje trazimo MAX od ove dece). Recimo da smo izracunali n - 1 f-ju evaluacije i sada je na redu an. U ovom sloju (sloju gde su deca, tik ispod stanja programa) stanja dobijaju vrednosti f-je evaluacije tako sto za svoju decu traze MIN f-je evaluacije (ima smisla jer je posle programa protivnik na potezu pa nas zanima njegov najbolji tj. nas MIN dobar potez). Oznacimo sa b1, ... , bn decu stanja cija je f-ja evaluacije an. Nas zanima MIN vrednost iz ovog skupa b. Za an vazi da je an <= bi (bice manje od svih sem od jednog od kojeg ce bas uzeti vrednost). Ako nadjemo bilo koje bj (an <= bj) takvo da je bj <= ak onda mozemo da prekinemo dalje evaluiranje za cvor an. Drugim recima, ako je bilo koje dete naseg cvora an manje ili jednako od bilo kog cvora a1, ... , a_(n-1), ne zanima nas vrednost an jer svakako nece biti razmatrana s obzirom da nas interesuje MAX iz skupa a1, ... , an. Alfa-beta heuristika daje najbolje rezultate kada se prvo nailazi na najbolje poteze (ako idemo od najlosijih ka najboljim potezima, nece biti ni jednog odsecanja). Imajuci ovo u vidu, vazno je da prvo ispitujemo one legalne akcije koje se cine boljim i bas na tome se zasnivaju neke implementacije alfa-beta heuristike.

Kiler algoritam:

Kiler algoritam predstavlja nadogradnju Minimaks algoritma zasnovanog na alfa-beta heuristici i sluzi za povecanja sanse za alfa i beta odsecanja. Kiler algoritam sluzi za pronalazenje takozvanih ***kiler*** poteza (najboljih poteza). Ako posmatramo stablo pretrage i sve njegove nivoe, za svaki nivo dubine d formira se kratka lista kiler poteza. Kad god dodjemo u neko stanje i posmtramo dubinu d, prve legalne akcije koje treba da razmatramo su kiler potezi. Ako kiler potez ne postoji, mora se rucnu vrsiti pretraga (uz alfa i beta odsecanja) i kada nadjemo najbolji potez, njega oznacavamo kao kiler potez za tu dubinu d. Intuitivno, ako je neki potez kiler za dubinu d, velika je sansa da ce biti kiler potez i u drugim stanjima na istoj toj dubini d. Ako kiler potez za dubinu d postoji ali nije legalni potez u tom stanju, onda opet rucno moramo traziti najbolji potez iz tog stanja. Ako nadjemo neki da je bolji od kiler poteza koji smo prethodno probali, onda taj novi (bolji) postaje glavni kiler potez a stari (losije) se pamti i koristi kada najbolji kiler potez nije legalna akcija a sledeci najbolji jeste legalna akcija.

Iterativni Kiler algoritam omogucava postojanje kiler poteza i za pocetni nivo stanja igre. Time se znatno smanjuje broj grana za ispitivanje i ubrzava Alfa-beta/kiler algoritam. Zasniva se na iterativnom pronalazenju kiler poteza. Krece se od dubine 1 i ide sve do dubine d_max. Na taj nacin, u svakoj iteraciji, azurira se trenutni kiler potez i on moze da se upotrebi na pocetni cvor i u sledecoj iteraciji ispitivanje svih

legalnih poteza krece od njega. Za najbolji potez iz pocetnog cvora bira se onaj kiler potez dobijen u nakon d_max iteracija.

Stabilno pretrazivanje:

Glavna mana Minimaks algoritma je ogranicenost dubine pretrage i donosenja odluke bez razmatranja o tome kakva stanja slede nakon te dubine. Na primer, u sahu za neko stanje najbolji potez je takav da vodi do uzimanja protivnikovog topa, medjutim vec sledece stanje (ono koje je ispod nase prve dubine pretrage koja nas je navela da *jurimo* prtivnikovog topa) je takvo da mi sta god uradili gubimo damu (zrtvovali smo damu za topa). Zbog ovog problema nastalo je ***stabilno pretrazivanje*** koje za takozvana ***nestabilna stanja*** produzuje dubinu pretrage stabla. Stanje je nestabilno ako zadovoljava odredjene kriteriju koji zavise od same igre koja se igra (primer odozgo bio bi nestabilno stanje). Tako u modernim programima koji koriste Minimaks za igranje saha, dubina pretrage je oko 10 nivoa a nakon toga ide, za nestabilna stanja, ide jos oko 10 nivoa pretrage kao posledica stabilnog pretrazivanja.

Prekidi i vremenska ogranicenja:

Kada bi nas program imao beskonacno vremena po partiji, mogao bi da istrazi kompletno stablo pretrage i uvek izadje kao pobednik. To, naravno, nije slucaj i vreme je gotovo uvek ograniceno. Ako posmatramo sah, program treba da predvidi broj poteza u partiji i podeli vreme trajanja partije sa potezima kako bi za svaki potez imao priblizno isto vremena da pretrazuje. Ako vreme istekne (ili je prekid od strane korisnika) pre pretrage celog stabla do neke odredjene dubine d, potrebno je odigrati najbolji do tada pronadjeni potez. Svi prethodni algoritmi, izuzev iterativnog kiler algoritma **nemaju** ovu osobinu.

Faktor grananja govori o ocekivanom broju zavrsnih cvorova koje algoritam pretrage stabla treba da ispita. Ako je R faktor grananja i dubina d, onda je broj cvorova koje treba ispitati R^d. Za sredisnjicu saha je to [35, 38] a za igru Go oko 250.

Uniformno stablo je stablo pretrage ciji svaki cvor sem listova, ima tacno b dece (stepen).

Teorema:

- ako je stablo pretrage uniformno stepena b i premenjuje se Minimaks sa alfa-beta odsecanjima tako da se u svakom cvoru prvo ispituje najbolje potez, onda se ispituje O(b^(d/2)) listova gde je d je dubina. Kada koristimo kiler heuristiku ovo i nije nerealno ocekivati. To znaci da uz alfa-beta/kiler pristup mozemo da ispitamo dva puta vise nivoa nego koristeci obican Minimaks (dva puta vise jer je u slozenost d podeljeno sa 2). Vazno je da stablo bude uniformno ali to mnoge igre ne podrzavaju.

Teorema

- alfa-beta algoritam je asimptotski optimalan algoritam za pretrazivanje stabla igre. Drugim recima, ne postoji tehnika kojom se moze ispitati manje zavrsnih cvorova nego sto bi se to uradilo primenom alfa-beta odsecanja.

Zavrsnica:

Zavrsnice predstavljaju krajeve partija i mogu da budu izuzetno *kombinatorno eksplozivne* jer dobra partija dovodi do izuzetno komplikovanih stanja gde jedan igrac ima zagarantovan mat (kada pricamo o sahu) ali mu je potrebno 40 poteza i taj mat se

nece uociti koristeci tehnike iz sredisnjice gde pretraga ide do 10 (ili 20) poteza u dubinu. Bitno je da **taktika u zavrsnici bude *korektna*** (vodi do pobede ili remija). Taktika je ***optimalna*** ako vodi do pobede u najmanjem broju koraka (odnosno do gubitka u najvecem broju koraka ako je gubitaj vec neminovan). Ako je optimalna onda je i korektna ali suprotno ne vazi.

Bramerov opsti algoritam:

Bramerov opsti algoritam grupise sve legalne pozicije u skup Q*. Svaki element tog skupa moguce clan je tacno jednog poskup skupa Q* takav da se u njemu nalaze sustinski skoro iste pozicije (kralj i pesak - nije vazno gde se nalaze). Svakom od tih podskupova pridruzena je jedinstvena ocena i f-ja evaluacije a onda se generise skup svih legalnih poteza Q i u odnosu na njega i na ocene dodeljene podskupovima Q* bira se sledeci potez.

Monte Karlo pretraga stabla igre:

Monte Karlo algoritam koristi se kada nije moguce istraziti celo stablo igre pa se umesto toga koristi **slucajno uzorkovanje** (*semplovanje*). Ovim nacinom pretrage aproksimira se izgled stabla na osnovu nepotpunih informacija. Monte Karlo algoritam kicma je najboljih programa za igranje strateskih igara danasnjice (AlphaZero, AlphaGo) uz dodatak masinskog ucenja. Monte Karlo algoritam moze da se koristi bez ikakvog iskustva o igri ljudi koji implementiraju algoritam. Dovoljno je znati pravila. Postepeno se, u iteracijama, dopunjava stablo igre tako sto se biraju cvorovi i u njima se odigravaju partije simulacijama: do kraja partije, slucajno izabranim potezima, trivijalnim rasudjivanjem. Ovakvim simulacijama profinjuje se stvarna slika o vrednosti cvora. Iteracija se odvija na sledeci nacin:

- bira se cvor M (cvor za razgranavanje), to se naziva ***politika selekcije***, takav da nije zavrsni i da iz njega sledi stanje m koje nije pre vidjeno a zatim se tom stanju daje ocena 0/0 i on se pridruzuje stablu a zatim se iz njega simulira partije na nacin koji je prethodno definisan. Nakon toga novom cvoru m treba dati ocenu a to se moze uraditi, na primer, ako je igrac koji je bio na potezu u cvoru m pobedio dobija ocenu 1 i ta vrednost sluzi da se na neki nacin azuraju ocene cvorova od m pa do korena.

Politika selekcije:

- najcesce se cvor bira nasumicno. Definise se posebno za jedan nivo stabla i rekurzivno primenjuje. Jedna dobra politika selekcije prati **formulu** `w/n` gde je n broj simulacija iz tog cvora a w suma rezultata tih partija gde je pobeda, na primer, 1; remi 0.5; poraz 0. Za razliku od Minimax-a, ovde sto veci brojevi uvek govore pozitivne stvari (nema smene min/max vrednosti i tih tumacenja). Ova formula oznacava **verovatnocu pobede** igraca koji u toj poziciji na potezu. Selekcija se sprovodi od korena ka listovima u uvek se bira koren koji ima vecu verovatnocu pobede cime se obezbedjuje da se bolji potezi detaljnije analiziraju. Vazno je simulirati i iz onih cvorova gde je broj simulacija vrlo mali kako ne bi doslo do previda nekih potencijalno dobrih poteza. Stablo koje kreira ovaj algoritam nije nuzno simetricno, sto je dobro u svakodnevnoj realnoj primeni ali moze imati svoju cenu u vidu toga da se program previse fokusira na jednu granu i ne vidi neke znatno bolje poteze.
- *VIDETI ILUSTRACIJU IZVRSAVANJA NA STRANI 62 I PROCITATI OBJASNJENJE IZ PRVOG PASUSA STRANE 63!**

GENETSKI ALGORITMI:

Genetski algoritmi koriste metaheuristike za resenje problema. ***Metaheuristke*** predstavljaju heuristike koje nisu specijalizovane za resavanje konkretnog problema vec se njima moze resavati skup problema koji na prvi pogled ne moraju da budu slicni. To se postize prilagodjavanjem odredjenih parametara. Losa strana metaheuristika je sto **ne garantuju pronalazak svih resenja** a odatle sledi da pronadjeno **resenje ne mora da bude optimalno** (dobijeno resenje moze biti lokalni maksimum a ne globalni). Oni mogu dati skup resenja sto je cesto dovoljno dobro. Genetski algoritmi koriste se za resavanje problema optmizacije ili pretrage (za NP-teske probleme...). Vrlo su vremenski i memorijski zahtevni ali su pogodni za paralelizaciju.

Hromozom/genotip -> reprezentacija jedinke

Cilj genetskog algoritma je naci vrednost za koju funkcija cilja dostize maksimum. Funkcija cilja ne mora da bude zadata eksplicitno vec implicitno kroz veci broj kriterijuma; ne mora da bude diferencijabilna ni neprekidna. Pored funkcije cilja postoji i **funkcija prilagodjenosti** i ona **oznacava kvalitet jedinke**. Funkcija cilja i funkcija prilagodjenosti cesto se poklapaju ali i ne moraju. Na primer, funkcija prilagodjenosti moze jedinkama koje prelaze odredjenu vrednost funkcije cilja dodetljivati vrednost 1 a onima koje ne 0. Treba biti obazriv jer tako moze doci do prebrze konvergencije.

Opsti postupak genetskog algoritama:

- ulaz: podaci koji odredjuju funkciju cilja i parametre algoritma
- izlaz: najkvalitetnija jedinka

1. kreiraj pocetnu generaciju
2. izracunaj funkciju prilagodjenosti svake jedinke
3. petlja:
    - izaberi iz generacije skup jedinki za reprodukciju
    - ukrstanjem i mutacijom napravi nove jedinke
    - od novih i starih jedinki kreiraj novu generaciju
    - kraj petlje kada je neki od uslova ispunjen (funkcija cilja dostigla maksimum - pronadjeno resenje, doslo se do odredjenog broja generacija, funkcija priladjenosti pozvana odredjen broj puta, vrednost prilagodjenosti najbolje jedinke nije se popravila kroz nekoliko generacija, kombinacija nekih od ovih uslova)
4. izaberi najkvalitetniju jedinku u najnovijoj generaciji

Jedinke se mogu predstaviti na razlicite nacine: nizom bitova, matricama, stringovima... Bitno je da nad tom reprezentacijom mozemo da upotrebimo operatore ukrstanja i mutacije. Nasi operatori treba da zadovoljavaju uslov da **nove jedinke** koje stvaraju budu **samo one koje mogu da budu nase resenje**. Ili, ako to ne ispunjavaju onda treba da postoje mehanizmi kojima se takve jedinke ispravljaju.

Ako je jedinka predstavljena nizom n bitova onda je potrebno uspostaviti vezu preslikavanja (koja nije bijektivna) izmedju reprezentacije i realnog intervala [a, b] gde se resenje moze naci:

Broju X sa binarnom reprezentacijom a = 00...0, b = 11...1 odgovara realna reprezentacija:

```
X = a + (x * (b - a)) / 2^n - 1
```

Realnom broju X odgovara binarna reprezentacija:

```
X = floor((X - a) * (2^n - 1) / b - a)
```

Funkcija prilagodjenosti daje ocenu kvaliteta jedinke. Treba da bude definisana za sve jedinke, da bude brza za izracunavanje i da daje dobru sliku kvaliteta jedinke. Sto je funkcija prilagodjenosti veca to je veca sansa da ce ta jedinka biti izabrana za reprodukciju. Vrednost funkcije raste sa porastom broja generacija.

Detaljan opis postupka izvrsavanja genetskog algoritma:

1. Inicijalizacija:

- **broj jedinku** u generaciji je **najcesce fiksan** i definise sa kao parametar algoritma. Pocetna generacija dobija se najcesce slucajnim izborom a ponekad se eksplicitno u nju ubacuju jedinke za koje se veruje da ce biti blizu u prostoru pretrage gde se resenje nalazi (one navode nove generacije).

2. Selekcija:

- predstavlja **izbor jedinki iz tekuce generacije koje ce ucestovati u kreiranju nove generacije**. Najcesce se oslanja na funkciju prilagodjenosti gde se mogu birati samo najbolje jedinke ili gde se vrednost funkcije prilagodjenosti oslikava u verovatnocu da jedinka bude izabrana. Na taj nacin losije jedinke imaju sansu da ucestvuju u pravljenju nove generacije i tako se cuva **genetska raznolikost** i sprecava prerana konvergencija ka, najcesce, lokalnom ekstremumu. Postoje **dva tipa selekcije**:

    1. **ruletska**:

        - vrednosti funkcije prilagodjenosti odredjuje sansu da jedinka bude izabrana. Ista jedinka moze vise puta biti izabrana cime se znacajno smanjuje efikasnost algoritma.

    2. **turnirska**:

        - od skupa svih jedinki bira se **skup k jedinki** (k < |populacija|). Iz novog skupa **moze se birati jedinka sa najvecom prilagodjenosti ili se skup moze sortirati i i-ta jedinka u sortiranom skupu bira se sa verovatnocom:**

            ```
            p(1 - p)^(i - 1)
            p -> verovatnoca (drugi parametar turnirske selekcije)
            ```

        - Prethodno izabranim jedinkama moze biti **zabranjeno da opet budu izabrane**. Ako je velicina novog skupa **k = 1** onda se turnirska selekcija zasniva na **random izboru jedinki iz cele generacije**; ako je **p = 1** onda se **u novom skupu bira najprilagodjenija jedinka**.

3. Reprodukcija:

- **od selektovanih jedinki biraju se dve koje ce se sa nekom sansom (najcesce od 0.6 do 0.9) ukrstiti i dati jedno ili dva nova deteta**. Ukrstanje se moze realizovati na razlicite nacine. Ako jedinke reprezentujemo nizom bitova onda mozemo koristiti **visepoziciono ukrstanje** kod kojeg moze nastati **jedno ili dva**

deteta i kod kojeg se oba roditelja na istom mestu/mestima dele i onda unakrsnim kombinovanjem nastaje dete. Pored visepozicionog ukrstanja postoji i **uniformno ukrstanje** kod kojeg **uvek nastaju dva deteta**. Dete nastaje tako sto se ide bit po bit kroz roditelje i za svaku poziciju sa odredjenom sansom (najcesce pola-pola) se iz jednog roditelja bit daje detetu a drugom detetu ide bit od drugog roditelja.

4. Mutacija:

- nakon ukrstanja sledi mutacija kod koje se definise sansa (najcesce **ispod 1% da se odredjeni bit** (ako je takva reprezentacija jedinke) **promeni**). **Mutacija omogucava da se pobegne od lokalnih ekstremuma**. Ako je, na primer, u jednoj generaciji odredjeni gen svih jedinki isti onda ne postoji sansa da se istrazi prostor pretrage gde taj gen nije takav. Time potencijalni globalni maksimum nikada nece biti ispitan. Koriscenjem mutacije ova pojava moze da se, potencijalno, izbegne. Ako je **sansa mutacije prevelika** onda je usmeravanje pretrage preslabo pa ona **lici na slucajnu pretragu**. Ako ne postoji onda se lako upada u lokalni ekstremum.

5. Politika zamene generacije:

- predstavlja nacin formiranja nove generacije jedinki. Postoje **dva osnovna pristupa**:
  1. **generacijski genetski algoritmi**:
     - biraju dovoljno predstavnika iz tekuce generacije da se njihovim ukrstanjem i mutiranjem kreira cela nova generacija koja ce zameniti staru.
  2. **algoritmi stabilnog stanja**:
     - cim se izaberi par roditelja vrsi se ukrstanje i mutacija i nova/e jedinke se ubacuju u zavisnosti od **politike zamene** koja moze da bude:
       - **zamena najgorih**:
         - nove jedinke zamenjuju najmanje prilagodjene u generaciji
       - **nasumicna zamena**:
         - random se biraju predstavnici iz stare generacije koji ce biti zamenjeni
       - **inverzna turnirska selekcija**:
         - umesto najprilagodjenijih biraju se najneprilagodjeniji

Elitizam se moze koristiti u obe ove politike i oznacava **pojavu kada se odredjene jedinke** (ili jedna) koje su po necemu najbolje **uvek stite od nestajanja**. Time se obezbedjuje da se neke kvalitetne osobine ne izgube u toku evolucije

Ponekad se genetskim algoritmom biraju paremetri drugih genetskih algoritama. **Od parametara u velikoj meri zavisi koliko ce sam algoritam biti dobar**. Oni ne moraju da budu staticki odredjeni vec se mogu menjati za vreme izvrsavanja algoritma.

Resenje problema *Obilaska table skakacem* genetskim algoritmom:

- problem pronalaska putanje skakaca na sahovskoj tabli dimenzije n x n tako da skakac poseti sto vise polja ali ni jedno polje dva ili vise puta

> - *Kao primer za resenje naseg problema uzecemo tablu 5x5*

> - *Skakac krece iz donjeg levog ugla*
> - *Na svakom polju, skakacu su na raspolaganju za skok najmanje dva a najvise 8 polja. Zbog toga cemo za reprezentaciju polja koristiti 3 bita (oznacava jedan od osam mogucih poteza sa tog polja)*
> - *Skakac moze da napravi najvise 24 skoka (tabla je velicine 25 a ni na jedno polje se ne sme vratiti, ukljucujuci i pocetno)*
> - *Iz prethodna dva sledi da ce jedna nasa jedinka (nase resenje) biti predstavljeno kao hromozom koji se sastoji iz 24 x 3 bita*
> - *Ovim pristupom necemo doci do resenja kojim se obilazi cela tabla jer se ono zasniva na naprednijim tehnikama i genetski algoritam sam od sebe to ne moze da resi*

Automatsko rasudjivanje

Resavanje problema koriscenjem automatskog rasudjivanja

Automatsko rasudjivanje (automatsko rezonovanje) bavi se razvojem algoritama i programa koji mogu da rasudjuju automatski, koristeci neke **matematicko-dedukticke metode** u okviru pogodnih logickih okvira. Neki od logicnih okvira: iskazna logika, logika prvog reda, logika viseg reda, fazi logika.. Automatko rezonovanje koristi se kod matematickog dokazivanja, verifikacije softvera i hardvera (***Formalne metode bave se specifikovanjem, razvojem i verifikacijom soft. i hard. sistema***), pravljanja rasporeda.. Postoje 3 faze u resavanju problema koriscenjem automatkog rasudjivanja:

1. modelovanje problema:
    - predstavlja izrazavanje naseg problema na matematicki formalan nacin koji odgovara logickom okviru koji je najpogodniji u odnosu na prirodu naseg problema. Potrebno je utvrditi vrstu osnovnih objekata naseg problema kao i domen nepoznatih velicina u njemu (to mogu biti jednostavni iskazi, poznati i nepoznati celi brojevi..). Nakon toga, potrebno je definisati uslove koje moraju da vaze za objekte opisane u problemu.
2. resavanje problema opisanog u matematickim terminima:
    - jednom opisan problem potrebno je resiti i to se postize na razlicite nacine u zavisnosti od nacine opisa samog problema. Na primer, 9x9 Sudoku moze se resiti tako sto za svako od 81 polja i 9 brojeva ispitamo formulu zadovoljivosti gde je svako polje na True ili False pa odatle imamo 2^(81*9). Lako se vidi da ovo nije nesto sto je izvodivo na pa je umesto toga potrebno odabrati bolje nacine za resavanje. Jedan on njih je rezonovanje o iskazima na osnovu prethodnih evaluacija. Na primer, ako imamo iskaze p, q i r (zadatak kada na bal dolaze ambasador Perua, Kine, Ruande) i vazi da je p True i imamo izraz: (p ili q) ili (q i r), ceo deo nakon p uopste ne moramo da evaluiramo.
3. interpretiranje i analiziranje resenja:
    - interpretiranjem resenja prevodimo rezultat resavanja naseg problema nazad u domen postavke problema. Na primer, ukoliko je p tacno to onda znaci da na bal treba da dodje ambasador Perua. Dodatne analize o zakljuccima resenja nisu potrebne jer su one jednoznacne. Medjutim, cesto je potrebno analizirati resenje u odnosu na model zakljucivanja koji smo izabrali jer razliciti modeli i okiviri logike mogu da daju razlicite zakljucke i njih je potrebno dalje analizirati. Ako oba modela

valjano izvode resenje naseg problema, oblici u kojima dolazi taj zaljucak mogu a cesto i jesu drugaciji. Dalje, ako je nas sistem rezonovanja rekao da nesto nije zadovoljivo, to ne znaci da to stvarno nije zadovoljnivo vec da taj sistem nije znao da ga zadovolji a potrebno je dodatno ispitati da li je zadovoljivo ili stvarno nije. Mozemo da izaberemo i dobar sistem koji omogucava efikasno automatsko rezonovanje ali da on ne radi na svim instancama problema i tada treba utvrditi koji su to slucaji kada nas sistem nije valjan i videti da li postavka naseg problema potpada pod te granicne slucajeve. Algoritmi za rasudjivanje treba da imaju sledeca svojstva:

- **Potpunost**:
  - algoritam je u stanju da dokaze svako tvrdjenje iz svog domena koje je **tacno** (nekada je za to potrebno dugo vremena i to je u redu)
- **Saglasnost**:
  - ako algoritam tvrdi da je neko tvrdjenje tacno, ono zaista mora da bude tacno (potpunost je pozeljna a saglasnost potrebna osobina) Za **problem odlucivanja** (problem koji zahteva odgovora *da* ili *ne*) kazemo da je **odluciv** ako postoji algoritam koji moze da resi svaku njegovu instancu a algoritam sa takvim svojstvom zovemo **procedura odlucivanja** (problem SAT - problem ispitivanja iskazne zadovoljivost, jeste odluciv). **Poluodlucivi problemi** nisu odlucivi ali za njih postoje algoritmi (**procedure poluodlucivanja**) koji nalaze resenje za sve instance za koje je odgovor *da* (za instance gde je odgovor *ne* algoritam vraca *ne* ili se uopste ne zaustavlja sto predstavlja glavni problem poluodlucivih problema jer ne znamo da li ce ikada doci do resenja ili ne). *Halting* problem je poluodluciv.

Automatsko rasudjivanje u iskaznoj logici

U iskaznoj logici (*propositional logic*) razmatraju se iskazi (tvrdjenja) koji logickim veznicima mogu biti kombinovani u kompleksnije iskaze. Ona ima svoju **sintaksu** (opisuje jezik) i **semantiku** (definise istinitosnu vrednost iskaza). Glavni problemi iskazne logike su problemi da li je iskazna formula **valjana (tautologija)**. Iskazna formula je valjana akko se evaluira na tacno za sve moguce istinitosne vrednosti elemenata koji je cine ("Ana je zubar ili Ana nije zubar" je uvek tacno). Iskazna formula je **zadovoljiva** akko postoji bar jedna tacna evaluacije istinitosnih vrednosti elemenata koje cine iskaz ("Ana je zubar i Ana zivi u Beogradu" je tacno akko su tacna oba tvrdjenja; "Ana je zubar i Ana nije zubar" nikada nije tacno). **SAT** problem je problem ispitivanja zadovoljivnosti formule koja je u KNF obliku, ovaj problem je odluciv. Resavanje ovih problema moze da se izvede na mnogo nacini tj. nece svaki postupak imati iste korake pri dolasku do resenje i svaki od tih razlicitih koraka moze da bude tacno izveden ali glavna *caka* je usmeriti ovo izvodjenje tj. pretragu tako se do resenja najbrze dodje. Iskazna logika je izuzetno izrazajna i moze resavati najrazlicitije probleme, tj. probleme nad konacnim domenom: ako imamo 2^n stanja onda uz n promenljivih mozemo da modelujemo ovaj sistem. Zbog velike izrazajnosti smatra se "svajcarskim nozicem" modernog racunarstva.

Sintaksa i semantika iskazne logike

Iskazne formule se najcesce definisu za fiksiran, konacan skup iskaznih promenljivih (iskaznih slova), dve logicke konstante (T i ?) i skup osnovnih bulovskih operatora: negacije, konjukcija, disjunkcija, implikacija i ekvivalencija.

```
Iskazne promenljive i iskazne konst. su iskazne formule (atomicke iskazne formule;
literal je ili atomicka IF ili je negacija atomicke IF)).
Ako su A i B iskazne formule onda su i objekti dobijeni kombinovanjem ovih formula i
veznika takodjen iskazne formule.
Samo na ove nacine mogu biti dobijene iskazne formule.
```

Kako bi se izbeglo stavljanje viska zagrada, podrazumeva se sledeci prioritet veznika: ?, ^, v, =>, <=>.

Iskazne formule mozemo da zamisljamo u vidu stabla. Svako podstablo koje odgovara nekoj IF je takodje iskazna formula i naziva se **potformula**: svaka IF A je potformula samoj sebi; ako je A = ne B onda je svaka potformula formule B i potformula formule A. Ako je A = B i C, B ili C, B sledi C, B ekviv. C onda je svaka potformula B i C istovremeno i potformula A.

Supstitucija je zamena jedne iskazne formule drugom unutar neke iskazne formule. A[C->D] znaci da u IF A menjamo sva pojavljivanja IF C iskaznom formulom D i to na sledeci nacin:

- ako je A = C onda je A[C->D] = D
- ako je A /= C i A atomicka onda je A[C->D] = A
- ako je A /= C i A = ne B onda je A[C->D] = ne (B[C->D])
- ako je A /= C i A = (B1 ^ B2) ili (B1 ? B2) ili (B1 => B2) ili (B1 <=> B2) tada je A[C->D] = redom (B1[C->D] ^ B2[C->D]) ...

Semantika

Semantika oznacava pridruzivanje vrednosti iskaznoj formuli, odnosno njenu evaluaciju. Svaka iskazna formula moze da se evaluira na Tacno i Netacno odnosno 0 i 1. Istinitosna vrednost svake formule zavisi samo od istinitosne vrednosti njenih potformula odnosno istinitosna vrednost IF zavisi od istinitosnih vrednosti logickih promenljivih koje se u njoj javljaju. Funkciju koja pridruzuje istinitosnu vrednosti promenljvim zovemo **valuacija** a njeno prosirenje predstavlja **interpretacija**.

```
primer:
"Ana je zubar i Ana zivi u Beogradu"
p (iskazna promenljiva) - "Ana je zubar"
q - "Ana zivi u Beogradu"

Ako je istinitosna vrednost p = 0 tj. ako je valuacija v(p) = 0 Ana nije zubar a to se
interpretira kao I_v(p) = 0 i I_v(p ^ q) = 0 (ako je valuacija ta i ta onda je
interpretacija ta i ta)
```

```
? i ? - sintaksa
0 i 1 - semantika
```

Iskazna formula A je:

1. **zadovoljiva** ako postoji neka valuacije v takva da je v(A) = 1 (kaze se jos i da je to model za A)
2. **valjana** (tautologija) ako je za svaku valuaciju v v(A) = 1, tj. ako je svaka valuacija model za A
3. **nezadovoljiva** (poreciva, kontradiktorna) ako ne postoji valuacija v u kojoj je v(A) = 1
4. **poreciva** ako postoji valuacija v u kojoj je v(A) = 0

```
"Ana je zubar ili Ana nije zubar"
p - "Ana je zubar"
p v ne p:
    1. v(p) = 1
       I_v(p v ne p) = 1
    2. v(p) = 0
        I_v(p v ne p) = 1
Posto je IF u svakoj valuaciji p tacna => IF jeste tautologija
```

Tvrdjenje:

- ako su A i A => B tautologije onda je i B tautologija
- *Dokaz**
- pp. da A i A => B jesu tautologije a da B **nije** tautologija. To znaci da ce postojati interpretacija I takva da I(A => B) = I(1 => 0) = 0 (A se uvek evaluira na 0 posto je tautologija). To je kontradiktorno sa tim da je A => B tautologija jer se u prethodnom evaluirala u 0 pa odatle sledi da B mora uvek biti 1 tj. tautologija.

Skup iskaznih formula moze da bude:

1. zadovoljiv:
    - postoji valuacija v takva da je svaka iskazna formula iz skupa tacna
2. nezadovoljiv:
    - ne postoji valuacija v takva da je svaka IF iz skupa tacna

Teorema:

- valuacija v je model skupa formula {A1, A2, ... , An} akko je valuacija v tacna za A1^A2^...^An

Logicke posledice i logicki ekvivalentne formule:

"Boban zivi u Beogradu ili Boban zivi u Krusevcu", "Boban ne zivi u Beogradu" logicka posledica je da: "Boban zivi u Krusevcu"

Kase se da je A ekvivalentno sa B akko je svaki model iskazne formule A i model iskazne formule B u isto vreme i obratno. Pise se A ? B

B |= A i B ? A nisu formule iskazne logike (*objektne formule*) vec su formule koje govore o formulama (***metaformule***)

Klauza - disjunkcija vise literala

```
Vazi A |= B akko je A => B tautologija
Vazi A ? B akko je A <=> B tautologija
```

Teorema o zameni:

- ako je A ? B onda je i C[A->B] ? C

Ispitivanje zadovoljivosti i valjanosti u iskaznoj logici

Problem ispitivanja tautologicnosti jeste odluciv, a odlucivi su i problemi ispitivanja zadovoljivosti, nezadovoljivosti i porecivosti. Svi ovi problemi se lako mogu svesti jedan na drugi pa je dovoljno da imamo efikasno resenje samo za 1 od ova 4 i sve ostalo mozemo da izvedemo: A je tautologija akko ?A nije zadovoljiva; A je poreciva akko ?A zadovoljiva; A je nezadovoljiva akko A nije zadovoljiva.

Algoritam zasnovan na istinitosnim tablicama potpuno je neupotrebiv za realne probleme.

Normalne forme

- IF je KNF ako vazi A1^A2^...^An pri cemo je svaki Ai disjunkcija literala
- IF je DNF ako vazi A1vA2v...vAn pri cemo je svaki Ai konjukcija literala

IF moze da ima vise razlicitih KNF/DNF

KNF algoritam:

1. (dok god je moguce) eliminisi veznik <=>: A <=> B ? (A => B) ^ (B => A)
2. (dok god je moguce) eliminisi veznik =>: A => B ? ne A v B
3. (dok god je moguce) primeni: ne(A v B) ? ne A ^ ne B

   ```
   ne(A ^ B) ? ne A v ne B
   ```

4. (dok god je moguce) eliminisi duple negacije: ne ne A ? A
5. (dok god je moguce) primeni neku od logickih ekvivalencija: A v (B ^ C) ? (A v B) ^ (A v C)

   ```
   (B ^ C) v A ? (B v A) ^ (C v A)
   ```

Teorema o korektnosti algoritma KNF:

- algoritam KNF se zaustavlja i ako je ulazna IF F tada je na izlazu F' koja je u KNF-u i logicki ekvivalentna F.

Potpun skup veznika: {ne, ^, v} ali zahvaljujuci ekvivalenciji A v B ? ne (ne A ^ ne B) i skup {ne v} kao i skup {ne, ^} su potpuni. Postoje i jednocnalni potpuni skupovi veznika a to su Lukasijeviceva f-ja (nili) i Seferova f-ja (ni) a one su ekvivalentne: A nili B ? ne (A v B); A ni B ? ne (A ^ B).

Problem sa transoformisanjem IF u KNF oblik je sto ako imamo mnogo n disjunkata, izlazna (KNF) formula imace 2^n konjukata. Zbog toga se umesto KNF tranformacije koristi **Cejtinovo kodiranje** koje je linearno u smislu vremena i prostora (ako ulazna IF ima n logickih veznika, nakon ove transformacije postojace najvise 4*n klauza a svaka ce imati najvise 3 litarala) ali izlaz iz ovog kodiranja nije ekvivalentna IF vec ***slabo ekvivalentna***: pocetna formula je zadovoljiva akko je zadovoljiva rezultujuca formula (ovo je skoro uvek dovoljno dobro). Cejtinovo kodiranje za svaku IF A koja je podformula pocetne formule uvode novu iskaznu promenljvu koja se naziva ***definiciona promenljiva***. Ovo je najveci problem Cejtinove trans. (uvodi mnogo novih promenljivih).

Problem SAT i DPLL procedura

SAT (satisfiability) problem je problem pronalaska odgovora da li je data iskazna formula u svom KNF (najcesce) obliku zadovoljiva. Kako je SAT NP-kompletan problem (svi NP problemi mogu da se svedu na njega), efikasno resenje ovog problema je velika naucna zagonetka jer ako bi se nasao dobar (polinomijalan) algoritam koji resava SAT problem, to bi znacilo da je P = NP (i pronalazac bi dobio utesnih milion dolara). Jedna od procedura kojima se resava ovaj problem je DPLL algoritam. On kao ulazi uzima KNF oblik IF gde redosled klauza kao i literala unutar klauza nije vazan.

Teorema o korektnosti DPLL procedure:

- za svaku IF DPLL se zaustavlja i daje odgovor True akko je IF zadovoljiva.

Smatra se da je prazan skup klauza zadovoljiv; prazna klauza nezadovoljiva, kao i IF koja sadrzi praznu klauzu. Algoritam DPLL: Ulaz: multiskup (skup gde je dozvoljeno ponavljanje) klauza D = {C1, C2, ... , Cn} Izlaz: "DA" ako je IF zadovoljiva, "NE" inace

1. ako je D prazan vrati "DA"
2. zameni sve ?? sa T i sve ?T sa ?
3. obrisi sve literale ?
4. ako D sadrzi praznu klauzu onda vrati "NE"
5. {Korak tautology}: ako neka klauza Ci sadrzi T ili negaciju nekog literala onda vrati vrednost koju vraca DPLL(D\Ci)
6. {Koran unit propagation}: ako je neka klauza jedinicna i jednaka iskaznom slovu p onda vrati DPLL(D[p->T]) (jednaka ?p onda DPLL(D[p->?]))
7. {Korak pure literal}: ako D sadrzi literal p ali ne i ?p onda vrati DPLL(D[p->T]) (sadrzi ?p ali ne i p onda DPLL(D[p->?]))
8. {Korak split}: ako DPLL(D[p->T]) vrada "DA", vrati "DA"; inace vrati DPLL(D[p->?])

U najgorem slucaju DPLL algoritam je eksponencijalne vremenske slozNosti po broju iskaznih promenljivih zbog rekurzivne primene SPLIT pravila (i svi drugi najbolji SAT resavaci su iste slozenosti). DPLL moze da se posmatra kao pretrage stabla gde je heuristika koja navodi pravilo SPLIT i politika da se ono promenjuje na - iskaznu promenljivu sa najvise pojavljivanja u tekucoj IF, neko od iskaznih slova iz najkrace klauze... DPLL se koristi za ispitivanje zadovoljivosti IF ali kao sto je vec receno, to se lako moze primeniti i za druga ispitivanja (valjanosti, nezadovoljivosti, porecivosti)

Svodjenje problema na SAT i enkodiranje

OVAJ DEO POGLEDATI U KNJIZI, JA GA NISAM SPREMAO

Automatsko rasudjivanje u logici prvog reda

Logika prvog reda (predikatska logika) daleko je izrazajnija od iskazne logike a glavna prednost je sto pruza kvantifikovanje promenljivih (samo promenljivih) i postoje egzistencijalni i univerzalni kvantifikator. I u predikatskoj logici glavni problem je ispitivajne da li je neka formula valjana (zadovoljiva) ali tu dolazi do malog problema - u predikatska logici ovaj problem nije odluciv vec poluodluciv (ako je formula valjana to ce biti dokazano ali ako nije onda nece moci da utvrdi da nije valjana).

Sintaksa logike prvog reda

Sintaksa logike prvog reda prosirena je **kvantifikatorima egzistencije i univerzalnosti** (imaju najvisi prioritet) u odnosu na iskaznu logiku. Oni cini **opsti deo jezika logike prvog reda**. Pored ovoga, u izgradnju formula ulaze i elementi **signature** (L):

- najvise prebrojiv skup funkcijskih simbola (oni koji su arnosti 0 se zovu **simboli konstanti**)
- najvise prebojiv skup predikatskih simbola

```
f_/2 - funkcijski simbol f arnosti 2
term - element nekog domena (broj kod sabiranja na primer)
formule - tvrdjenja koja mogu biti tacna ili netacna
f(t1, t2, ... , tn) - primenjujemo f-ski simbol f na termove t1, t2, ... , tn
```

Signatura za problem sa 3 kutije koje su slozene jedna na drugu: L = ({}, {above_/2, below_/2}) V = {x, y, z} - skup promenljivih above(x, y), below (y, x), ?x ?above(x, x) - neke od formula

Promenljiva je **vezana (slobodna)** u formuli akko ima vezano (slobodno) pojavljivanje u toj formuli (?x, ?x - vezane za simbole ispred x; p(x, y) => (?x)q(x) - x u prvom delu slobodna u drugom vezana a y uvek slobodna). Sva pojavljivanja vezanih promenljivih mogu se preimenovati bez uticaja na znacenje formule (sin(x) ne zavisi od promenljve x). Znacenje formule zavisi od slobodnih promenljvih u njima. Ako formula nema slobodne promenljive zove se **zatvorena formula** ili **recenica**. Ako ima samo vezane promenljive koje su vezane samo "za svako" kvantifikatorom onda se f-la naziva ***univerzalnim zatvorenjem*** ((?x)(?y)...A) formule a ako ima samo "postoji" onda se naziva ***egzistencijalnim zatvorenjem*** formule.

Zamena

U logici prvog reda postoje dva vida zamene:

1. zamena promenljive termom (isto kao i jedina zamena u iskaznoj logici)
    - pri ovoj zameni moramo da pazimo na sledece: recimo da imamo ((?y)otac(x, y))[x->y] = (?z)otac(y, z) a ne (?y)otac(y, y) jer znacenje nije isto (term ne sme da sadrzi promenljive)

Supstitucija ? je niz uzastopnih zamena [x1->t1, x2->t2, ... , xn->tn] gde je xi promenljiva a ti term takav da ne sadrzi nijednu od promenljivih xi (na primer ? = [x->sokrat, y->platon] i s = ucitelj(x, y) vazi s? = ucitelj(sokrat, platon)) 2. zamena formule formulom (kao kod iskazne logike - zamenom podstabla formule zeljenom formulom): - A[B1->B2] - formula dobijena zamenom formule B1 formulom B2: * ako je A = B1 onda A[B1->B2] = B2 * ako je A atomicka i nije jednaka B1 onda je A[B1->B2] = A **IMA JOS FORMULA ALI SU PRILICNO INTUITIVNE (VIDETI STRANU 115, DEFINICIJA 9.5)**

Semantika

Za razliku od iskazne logike gde se promenljive preslikavaju u skup {0, 1}, kod predikatske logike to je nesto komplikovanije. Naime, tu se promenljive preslikavaju u ***univerzum (domen, nosac)*** (skup celih brojeva, skup stanovnika drzave, polozaj u prostoru..). Formula zavisi i od jos nekih dodatnih izbora i za formulu kazemo da je **valjana** akko je tacna za svaki od tih izvora. Na primer, kada gledamo nas slucaj sa 3 kutije koje su jedna na drugoj, u jednom univerzumu nas zanima poredak i one dobijaju vrednost u odnosu na f-ske simbole above_/2 i below_/2. U drugom univerzumu kutije se preslikavaju u cele brojeve i onda nas zanima relacija > i < izmedju njih. **Interpretacija** je odredjena L-strukturom (univerzumom i izabranim znacenjem f-skih i predikatskih simbola) i valuacijom (vrednoscu promenljivih). **Valuacija** v je preslikavanje koje elementu iz skupa promenljivih P dodeljuje vrednost d iz univerzuma U (v(x) = kutija B, v(y) = kutija A pa vazi: I_v(above(x, y) = "jeste iznad"(I_v(x), I_v(y)) = "jeste iznad"(B, A) = 1 (VIDETI SLIKU))). Interpretacije su iste kao kod iskazne logike uz dodatak interpretacija za kvantifikatore:

- I_v((?x)A) = { 1, ako za svaku valuaciju w ~_x v vazi I_w(A) = 1

  ```
          { 0 inace
  ```

- I_v((?x)A) = { 1, ako postoji valuacija w ~_x v takva da vazi I_w(A) = 1

  ```
          { 0 inace
  ```

Skup formula je **zadovoljiv (konzistentan)** ako ima barem jedan model.

Primer: postoji drzava koja se granici sa Italijom i Austrijom ?x (drzava(x) ^ graniciSe(Italija, x) ^ graniciSe(Austrija, x)) Italija, Austrija - simboli konstanti drzava_/1, graniciSe_/2 - predikatski simboli

Logicke posledice i ekvivalentne formule

Kazemo da je formula A **logicka posledica** skupa formula G na istom signaturom L i pisemo G |= A ako je svaki model za G model za A. Kazemo da su formule A i B **logicki ekvivalentne** ako vazi {A} |= B i (akko je A => B valjana) {B} |= A (akko je B => A valjana) i pisemo A ? B (akko je A <=> B valjana). Formula A je logicka posledica praznog skupa formula akko je A valjana i to se pise {} |= A ili samo |= A. Ako je skup f-la kontradiktoran onda je svaka formula njegova posledica (svaka formula je logicka posledica {?}).

Neke od logickih ekvivalencija (OSTALE POGLEDATI NA STRANI 120, PRIMER 9.20):

```
?(?x)A ? (?x)?A - De Morganov zakon
?(?x)A ? (?x)?A - De Morganov zakon
```

Pored ovoga vazno je znati da istorodni kvantifikatori mogu da zamene mesta (dva egzistencijalna kvantifikatora koja su jedan do drugog mogu da menjaju poredak (isto vazi i za univerzalni kvantifikator) ali dva razlicita (univerzalni i egzistencijalni) **ne smeju**).

Teorema o zameni:

- ako je B1 ? B2 onda vazi A[B1->B2] ? A

Mali primer izvodjenja: ?(?x)(A ? ?B) ? (?x)?(A ? ?B) ? (?x)(?A ? ??B) ? (?x)(?A ? B) ? (?x)(A => B)

Odlucivost i zadovoljivost

Kao sto je prethodno receno, ispitivanje zadovoljivosti u logici prvog reda nije odlucivo (poluodlucivo je). To znaci da ce za proizvoljnu zadovoljivu formulu algoritam vratiti da je zadovoljiva ali za proizvoljnu formulu koja nije valjana algoritam se najcesce nece zaustaviti. Zbog ovoga je i problem ispitivanja nezadovoljivosti poluodluciv. Zbog ovoga je i problem G |= A poluodluciv. Ukoliko mi ogranicimo nas problem tako da razmatra samo jedan model (da nas problem sa kutijama bude postaljen preko signature L = {{}, {above_/2, below_/2}}) bez nekog modela koji kutije preslikava u cele brojeve ili nesto drugo, onda **je** nas problem **odluciv**. Za ispitivanje valjanosti u ovim situacijama koriste se **SMT resavaci**. Kako problemi iskazne logike mogu da budu izuzetno slozeni i raznoliki, valjanost formule A se svodi na ispitivanje valjanosti G |= A i te procedure se zovu ***dokazivaci za logiku prvog reda***.

Prvi korak u ispitivanju da li je formula A zadovoljiva jeste prestavljanje te formule u nekoj formi pogodnoj za resavac. To je najcesce normalna forma. Jedna formula A moze da ima vise normalnih formi a i jednoj normalnoj forma moze da odgovara vise formula. Od A do normalne forme dolazi se transformacija koje koriste logicke ekvivalencije. **Definicija**:

- za formula A kazemo da je u ***Preneks normalnoj formi*** ako je oblika Q1x1Q2x2...QnXnA gde su Qi kvantifikatori a xi promenljive vezane za njih. U A delu ne sme da bude kvantifikatora kao ni slobodnih promenljivih koje nisu xi. U okviru ovoj algoritma najpre se resavamo <=> i => a zatim gledamo da li ima slobodnih promenljivih koje mogu da naprave problem (NA STRANI 122, VIDETI PRIMER 9.22).
- *Teorema o korektnosti PRENEX algoritma**:
- algoritam Prenex se zaustavlja i zadovoljava sledece svojstvo: ako je F na ulazu, na izlazu je F' u preneks normalnoj formi i vazi F ? F'.

Preneks normalna forma algoritam:

1. A <=> B ? (A => B) ^ (B => A) A => B ? ?A v B
2. ?(A ^ B) ? ?A v ?B ?(A v B) ? ?A ^ ?B ?(?x)A ? (?x)?A ?(?x)A ? (?x)?A
3. ??A ? A
4. (?xA) ? B ? (?x)(A ? B) (?xA) ? B ? (?x)(A ? B) B ? (?x)A ? (?x)(B ? A) B ? (?x)A ? (?x)(B ? A) (?xA) ? B ? (?x)(A ? B) (?xA) ? B ? (?x)(A ? B) B ? (?x)A ? (?x)(B ? A) B ? (?x)A ? (?x)(B ? A) paziti da u B nema slobodnih pojavljivanja x. Ako ima, preimenovati x koje je vezano za A.

Zarad lakse skolemizacije uvek biramo da je egzistencijalni kvantifikator pre univerzalnog.

Za A kazemo da je KNF-u (bez kvantifikatora) ako je u obliku je konjukcije klauza. Nacin dobijanja KNF-a je isti kao kod iskazne logike.

Nakon PRENEX-a i KNF-a formula je u obliku Q1x1...QnxnA gde je A u KNF-u. Dalje, najpogodnije bi nam bilo kada bismo sve ? kvantifikatore zamenili ?. To je moguce akko je polazna formula zadovoljiva, inace bismo dobili formule koje nisu zadovoljive. Proces zamene egzistencijalnog univerzalnim kvantifikatorom (proces eliminacije ?) se

naziva ***Skolemizacija***. Ona se zasniva prosirivanjem polazne signature **Skolemovim konstantama** (f-ski simboli arnosti 0) **Skolemovim funcijama**.

Teorema o skolemizaciji: ako je formula A signature L zadovoljiva i u preneks normalnoj formi, onda ce nakon skolemizacije biti dobijena formula B signature L' koja je takodje zadovoljiva. Postoje dva oblika skolemizacije:

1. (?x)p(x) --> p(const)
2. (?x)(?y)(?z)p(x, y, z) --> z = f(x, y) (p ide u funkciju gde su argumenti one promenljive vezane za sve univerzalne kvantifikatore ispred njega) ((?x)(?y)p(x, y, f(x, y))). Ovde f nije predikatak nego funkcijski simbol kojim prosirujemo signaturu.

VIDETI PRIMER 9.24 NA STRANI 124

Klauzalna forma je forma koja nastaje kada predikatsku formulu prvo transformisemo u preneks normalnu formu, pa deo bez kvantifikatora transformisemo u KNF pa na kraju skolemizacijom eliminisemo egzistencijalne kvantifikatore. Oblika je ?x1?x2 ... ?xnA gde je A formula bez kvantifikatora u KNF-u i bez slobodnih promenljivih pored, eventualno, x1, x2, ... , xn. Ako na formulu A izvrsimo proces pretvaranja u klauzalnu formu i dobijemo formulu B, ta formula B nije logicki ekvivalentna formuli A vec samo **slabo ekvivalentna** sto znaci da ako je A zadovoljiva i B ce biti i za nase ispitivanje to je sasvim taman. Ne postoji za svaku recenicu klauzalna forma ((?x)p(x)).

Unifikacija

- problem ispitivanja da li postoji zamena koja dva izraza cini jednakim. Ako za dva izraza e1 i e2 postoji zamena gama takva da e1*gama* = *e2*gama, onda kazemo da su izrazi e1, e2 unifikabilni i da je supstitucija gama unifikator. Dva izraza mogu da imaju vise unifikatora ali i ***najopstiji unifikator***. Za unifikator kazemo da je najopsitiji ako svaki unifikator izraza e1, e2 moze biti predstavljen kao kompozicija neke dve supstitucije.

Promenljiva moze da se unifikuje sa konstantom ali konstanta ne moze sa promenljivom!

VIDETI STRANU 127 ZA JOS O UNIFIKACIJI

Metod rezolucije

- je metod ispitivanja (ne)zadovljivosti formula predikatske logike u klauzalnoj formi. Nesto jednostavnije, metodom rezolucije mozemo da proveravamo da li je skup klauza koje cine formulu zadovoljiv. Formula moze da ima konjukte koji se ponavljaju ali se na osnovu logickih ekvivalencija (A ^ A ? A i A v A ? A) ponavljanja literala mogu eliminisati i formula se predstaviti kao skup klauza bez literala koji se ponavljaju. Formula je onda zadovoljiva akko postoji interpretacija u kojoj su sve njene klauze tacne a klauza je tacna akko je bar jedan njen literal tacan. Prazna klauza znaci da nije zadovoljiva tj. da cela formula nije zadovoljiva. Literali l i l^nadvuceno su ***medjusobno komplementni***. ***Dokazivanje pobijanjem*** slucaj da se nove klauze izvode beskonacno bez da se izvede prazna klauza. Ako u nekom koraku ne moze da se izvede ni jedna nova klauza onda formula nije zadovoljiva.