Algoritam smanjivanja prostora stanja pretrage

Sokol, Domina

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

ZAVRŠNI RAD

Algoritam smanjivanja prostora stanja pretrage

Domina Sokol

Split, rujan 2020.

Temeljna dokumentacijska kartica

Završni rad

Sveučilište u Splitu

Prirodoslovno-matematički fakultet

Odjel za informatiku

Ruđera Boškovića 33, 21000 Split, Hrvatska

Algoritam smanjivanja prostora stanja pretrage

Domina Sokol

SAŽETAK

Redukcija prostora stanja je tehnika optimizacije izvršenja heurističkih algoritama. Dijeli se na dva dijela prema kriteriju optimalnosti pronađenog rješenja: dopustiva redukcija prostora stanja i redukcija prostora stanja uz očuvanje rješenja. Prva skupina metoda odnosi se na one koje čuvaju optimalno rješenje – redukcija podstringova, detekcija slijepih ulica i redukcija pomoću simetrije. Druga skupina odnosi se na one metode koje nužno ne čuvaju optimalno rješenje, ali i dalje osiguravaju pronalazak nekog rješenja (redukcija pomoću makro akcija, odbacivanje nebitnog i redukcija parcijalnog uređaja).

U ovom završnom radu je korišten primjer pretraživanja labirinta. Prikazana su dva algoritma koja koriste ideje redukcije prostora stanja – Aho-Corasick algoritam i IDLA*.

Ključne riječi: prostor stanja, algoritam smanjivanja prostora stanja, redukcija prostora stanja

Rad je pohranjen u knjižnici Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu

Rad sadrži: 47 stranica, 31 grafička prikaza, 2 tablice i 10 literaturnih navoda. Izvornik je na hrvatskom jeziku.

Mentor: **Dr. sc. Goran Zaharija,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Ocjenjivači: **Dr. sc. Goran Zaharija,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Dr. sc. Saša Mladenović, *izvanredni profesor Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Dr. sc. Divna Krpan, *viši predavač Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen: rujan 2020.

Basic documentation card

Thesis

University of Split

Faculty of Science

Department of informatics

Ruđera Boškovića 33, 21000 Split, Croatia

State Space Pruning

Domina Sokol

ABSTRACT

State space pruning is an optimization technique applied to heuristic algorithms. It is divided into two parts according to the optimality of a found solution: admissible state space pruning and non-admissible state space pruning. The first group of methods is regarding to those which preserve the optimal solution to the given problem – substring pruning, dead-end pruning and symmetry reduction. The other group of methods is regarding to those which do not necessarily preserve the optimal solution, but do ensure some solution (macro problem solving, relevance cuts, partial order reduction).

In this bachelor thesis, the example of a labyrinth is frequently used. Two algorthims are shown which use the basic ideas of state space pruning – Aho-Corasick and IDLA*.

Key words: state space, state space pruning

Thesis deposited in library of Faculty of science, University of Split

Thesis consists of: 47 pages, 31 figures, 2 tables and 10 references

Original language: Croatian

Mentor: **Goran Zaharija, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*

Reviewers: **Goran Zaharija, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*

Saša Mladenović, Ph.D. *Associate Professor of Faculty of Science, University of Split*

Divna Krpan, Ph.D. Senior Lecturer *of Faculty of Science, University of Split*

Thesis accepted: September 2020.

Sadržaj

Uvod

Algoritmi reduciranja prostora stanja jedna su od tehnika optimizacije heurističkih algoritama pretraživanja u umjetnoj inteligenciji. Pretraživanje se vrši nad tzv. prostorom stanja (state space). Stanja su predstavnici rezultata određenih operacija algoritma primijenjenih na varijable. Prostor stanja je skupina svih mogućih stanja koja se pretražuju algoritmom. Ovisno o broju varijabli i veličini te složenosti algoritma, ovi prostori mogu vrlo brzo narasti.

Jedna od najučinkovitijih tehnika smanjenja velikih prostora stanja, tj. obrezivanja stabla pretrage su upravo algoritmi reduciranja prostora stanja. Njihovi ciljevi su smanjenje zauzete memorije pri izvršenju algoritma pretrage i skraćivanje vremena izvršavanja algoritma pretrage eliminacijom suvišnih stanja iz stabla pretrage čime je algoritam optimiziran, a prostor stanja znatno smanjen.

Moguće je i „previše“ smanjiti prostor stanja, odnosno izgubiti put koji bi vodio prema rješenju problema. Isto tako je moguće izgubiti optimalan put prema rješenju. Zbog toga se algoritmi reduciranja stanja dijele u dvije velike skupine: oni koje čuvaju optimalno rješenje i oni koje to ne čine.

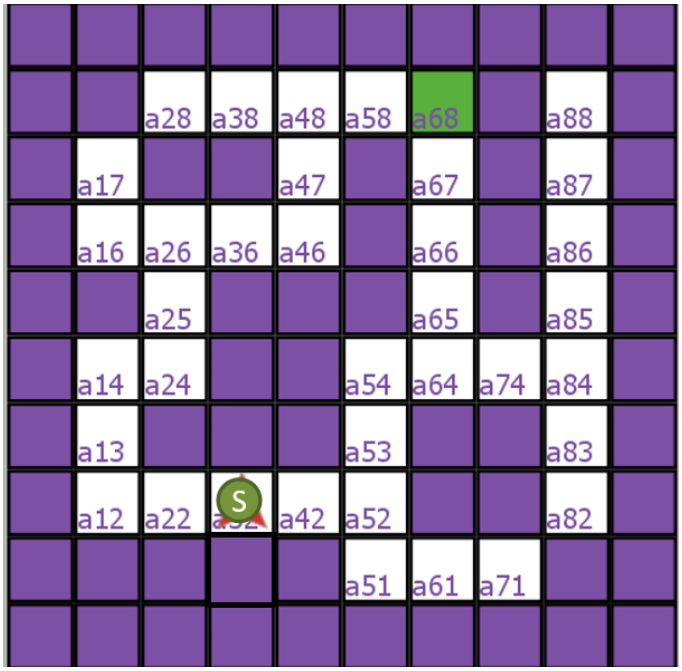
U ovom radu će se promotriti osnove teorije algoritama reduciranja prostora stanja, uključujući gore navedenu podjelu i dva poznata algoritma koja koriste ideje te teorije, Aho-Corasick algoritam i Incremental duplicate learning A*.

1. Razlozi za redukciju prostora stanja (motivacija)

1.1. Prostor stanja

Prostor stanja osnovan je koncept umjetne inteligencije. [1] Stanje označava trenutan status u izvršenju algoritma, odnosno fazu u kojoj se nalazi problem koji algoritam rješava. Smisao algoritma pretrage je doći od početnog stanja do ciljnog stanja, gdje su to zapravo apstrakcije nekih konkretnih događaja u stvarnosti. Početno stanje označava skup polaznih parametara, a ciljno stanje one parametre koje treba dobiti, tj. one kriterije koje je potrebno zadovoljiti. Između početnog i ciljnog stanja postoje i ostala, prijelazna. Iz jednog u drugo stanje dolazi se vršenjem operacija koje su prethodno zadane kao dozvoljene.

Sam prostor stanja je skup svih mogućih stanja u kojima se problem može naći. On sadrži sva početna, ciljna i prijelazna stanja na putu od jednog do drugog te akcije koje vode od jednog do drugog stanja. Najčešći i najzorniji primjer je labirint (Slika 1), odnosno problem pronalaska puta od početne pozicije do krajnje.



Slika 1. Općeniti prikaz labirinta, S predstavlja početno polje (eng. start), zeleno polje predstavlja cilj

Algoritmi prolaze prostorom stanja idući od jednog do drugog pomoću dozvoljenih operacija. Npr. u labirintu postoji slobodan prostor kojim se moguće kretati i stoga na svakoj koordinati postoje dozvoljene operacije: ići lijevo, desno, naprijed i natrag, ovisno o preprekama.

Problem nastaje kad prostor stanja postane prevelik, odnosno kad postoji veliki broj stanja kroz koja je moguće proći. To znači da vremenska složenost algoritama može narasti eksponencijalno brzo i događa se tzv. „kombinatorna eksplozija“ opcija. [1] Budući da je za izvršavanje algoritma na raspolaganju konačan i ograničen vremenski interval, potrebno je na neki način optimizirati algoritam tako da mu vrijeme izvršavanja upadne u zadani prihvatljivi okvir.

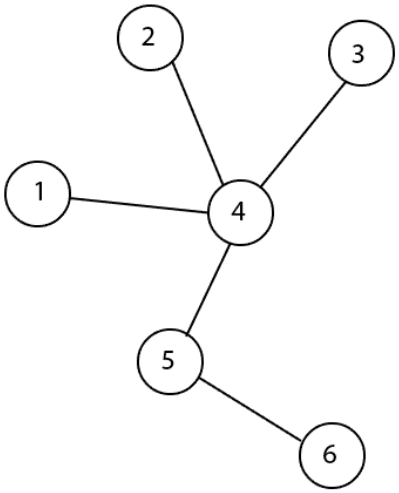
Jedan pristup tome je smanjenje prostora stanja gdje se primjenjuje algoritam redukcije prostora stanja uz odabrani algoritam pretrage.

2. Redukcija prostora stanja

Prostori stanja mogu se prikazati u ekvivalentnom obliku – stablu odluka. Stablo odluka je grafički i konceptualni prikaz prostora stanja u kojem je svako stanje, u kojem je moguće donijeti odluku o idućoj operaciji (gdje postoji izbor od dvije ili više operacija), predstavljeno čvorom. Čvorovi su povezani bridovima, kojima mogu biti dodijeljene težine.

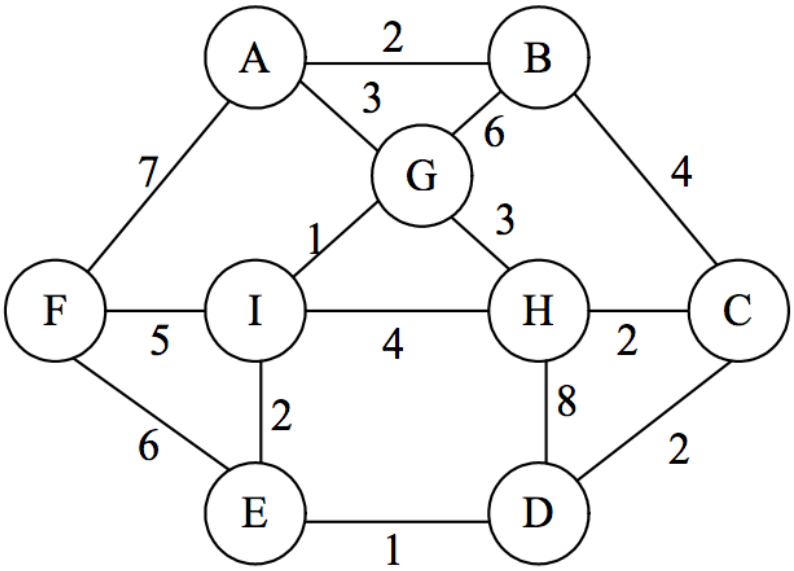
Definicija: Graf je uređena trojka G = (V,E,φ), gdje je V neprazni skup čiji se elementi nazivaju vrhovima ili čvorovima, E skup čiji se elementi nazivaju bridovima, a φ preslikavanje koje svakom bridu pridružuje neuređeni par vrhova. [2]

Definicija: Stablo je povezan graf bez ciklusa. [2]



Slika 2. Primjer stabla s numeriranim čvorovima.

Definicija: Težinski graf je uređeni par (G,ω) grafa G i funkcije ω : E → $\mathbb{R}^+$ koja svakom bridu pridružuje nenegativan realan broj koji se naziva težinom brida.



Slika 3. Primjer težinskog grafa. [1]

Algoritmi počinju svoje izvršavanje na korijenu stabla i poanta je doći do ciljnog čvora, po mogućnosti s „najmanjim troškom“ – u vidu vremena, težina na grafu ili nekim drugim parametrima. Poželjno je naći optimalan put, ako je to moguće. Postoje i varijante algoritama kojima nije moguće uopće naći cilj, ili je moguće naći cilj, ali ne i optimalan put.

U skladu s tim, algoritmi redukcije prostora stanja dijele se u dvije glavne skupine: u one koje pri smanjenju prostora stanja čuvaju optimalno rješenje i one koje to ne čine, već čuvaju samo svojstvo algoritma pretrage da nađe ikakvo rješenje. Prva grupa algoritama se naziva dopustivo reduciranje prostora stanja (eng. admissible), a potonja ne-dopustivo reduciranje prostora stanja (eng. non-admissible), odnosno reduciranje prostora stanja uz očuvanje rješenja (eng. solution preserving).

Većina pristupa smanjenju prostora stanja se oslanja na pronalaženje pravilnosti u prostoru stanja. Poželjno je iste iskoristiti u svrhu lakšeg pretraživanja prostora stanja. Pravilnosti se često nalaze promatranjem dekomponiranog i pojednostavljenog prostora stanja.

2.1. Dopustivo reduciranje prostora stanja

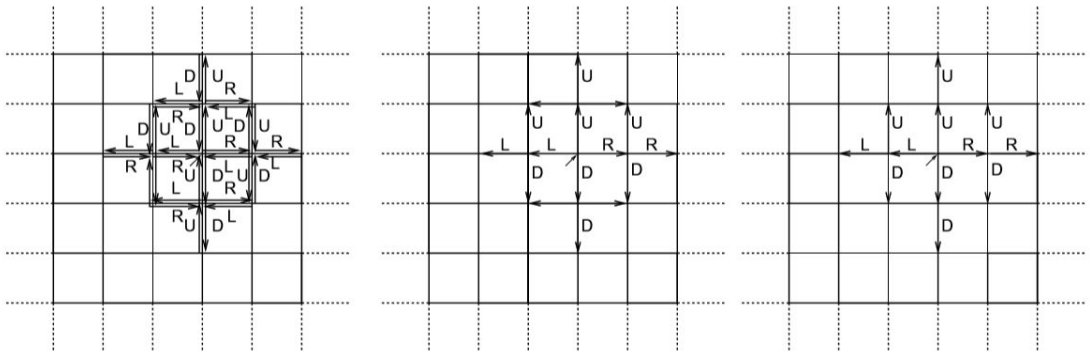
Dopustivo reduciranje prostora stanja čuva optimalno rješenje na reduciranom prostoru stanja, tj. čak i nakon znatnog smanjenja broja stanja, odabrani algoritam pretrage može pronaći optimalan put od početka do cilja [1].

Tri su faze u procesu dopustive redukcije prostora stanja [1]:

- Reduciranje podstringova – uklanjanje skupa zabranjenih nizova operacija,
- Detekcija slijepih ulica,
- Redukcija na temelju simetrije – smanjivanje prostora stanja fokusirajući se na predstavnike klasa neke relacije.

2.1.1. Reduciranje podstringova

Jedna od osnovnih radnji koju većina algoritama vrši čak i bez redukcije prostora stanja je zabrana direktnog povratka na neposredno prethodno stanje. Npr., u pretrazi labirinta dozvoljeno je ići „naprijed“, ali nije dozvoljeno (a ni logično) odmah iza toga vratiti se na isto mjesto inverznom operacijom „natrag“. Poželjno je ukloniti, tj. zabraniti operacije koje uvijek daju isti rezultat kao neka druga operacija, tzv. duplikate (eng. duplicate) i direktne inverze operacija opisane u prethodnoj rečenici.



Slika 4. Slijeva nadesno prikazani su potpun prostor stanja za kretanje u 2D labirintu, njegov reducirani prostor bez duplikata i reducirani prostor koji sadrži samo prečace. [1]

Cilj redukcije podstringova je izbaciti staze iz stabla pretrage koje sadrže duplikate. Duplikati su zabranjeni jer je unaprijed poznato da se istom stanju može pristupiti drugim putem koji je možda kraći. Spomenuti kraći putevi nazivaju se prečaci (eng. shortcut). Postavlja se pitanje kako pronaći parove duplikata i prečaca?

Jedno od rješenja je provesti početnu pretragu do određene dubine stabla pretrage, zapisujući sva prijeđena stanja u hash tablicu. Kolizije unutar hash tablice ukazuju na duplikate. Kada se to dogodi, akcije koje su stvorile koliziju se uspoređuju koristeći leksikografski uređaj te se ona veća smatra duplikatom, a ona kraća prečacem.

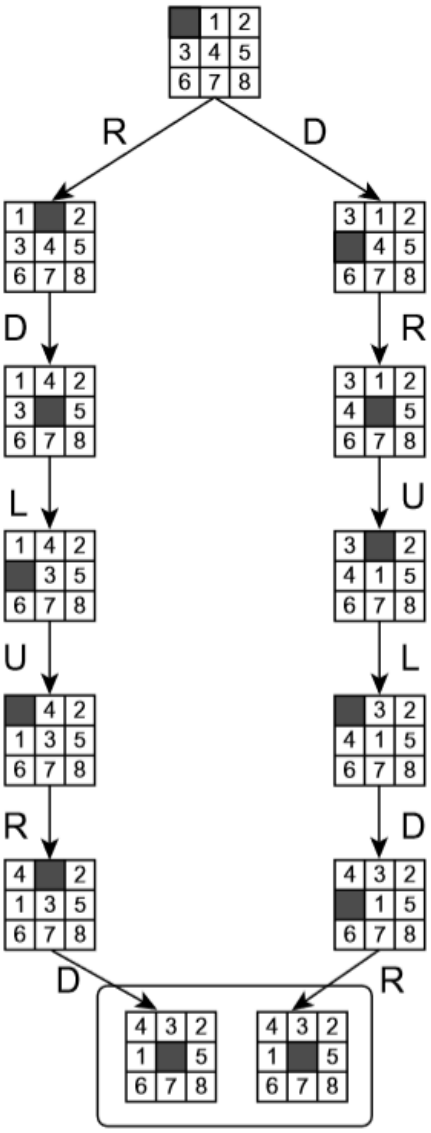
<u>Definicija:</u> *(leksikografski uređaj)* [3]

Neka su $(X, \leq_X)$, $(Y, \leq_Y)$ uređeni skupovi. Relacija $\leq_l$ definirana na Kartezijevom produktu $X \times Y$ na način:

$$(x, y) \leq_l (x', y') \leftrightarrow (x <_X x') \vee (x = x' \wedge y \leq_Y y')$$

je uređaj i naziva se leksikografski uređaj.

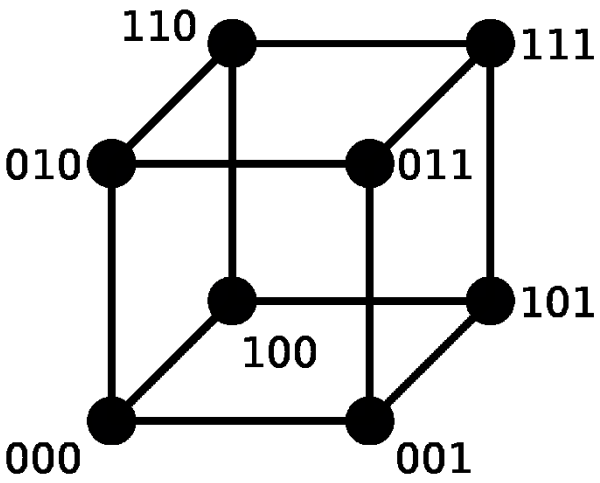
Još jedna opcija rješenja na općenitom neusmjerenom grafu (za razliku od stabla) je traženje ciklusa uspoređujući novonastali čvor stabla s prethodnima koji vode do njega. Rezultat je ciklus koji se potom dijeli u parove duplikata i prečaca. Npr., ako je početni čvor od interesa u labirintu na koordinatama (x, y), moguće je kreirati više različitih ciklusa koji se vraćaju u njega. Nije nužno da ciklusi prolaze različitim koordinatama, dovoljno je promijeniti operacije prvog ciklusa u inverzne da se dobije drugi ciklus koji obuhvaća ista polja.



Slika 5. Primjer ciklusa s duplikatom. [1]

Inverzne operacije (npr. lijevo je inverz od desno) se smatraju mogućima zbog toga što je graf neusmjeren. Moguće je konstruirati heuristiku koja minimizira duljinu ciklusa.

Za takve pretrage koje se fokusiraju na cikluse, prikladne su ocjene udaljenosti od stanja do stanja kao što je npr. Hammingova udaljenost [4]. Ona se definira za stringove jednake duljine i označava broj pozicija na kojima se oni razlikuju. Drugim riječima, Hammingova udaljenost mjeri najmanji broj zamjena znakova koje su potrebne kako bi se jedan string pretvorio u drugi.



Slika 6. U kubnom grafu susjedni čvorovi udaljeni su za 1 prema Hammingovoj udaljenosti.

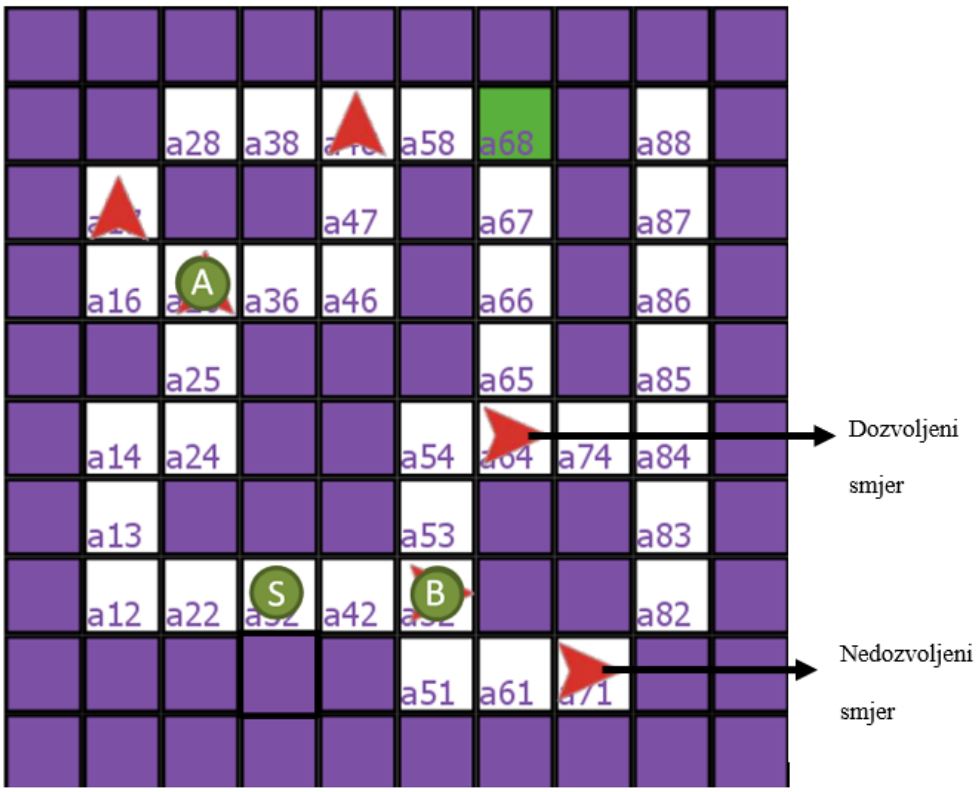
Grafovi prostora stanja koji imaju cijenu (u vidu nekog konkretnog parametra u stvarnosti) postaju težinski grafovi jer je potrebno cijenu uključiti u rješavanje problema na grafu. Težinska funkcija bit će zadana na skupu stringova akcije. Zato je prirodno uvesti sljedeći važan pojam u teoriji algoritama reduciranja prostora stanja:

Definicija: (redukcijski par) [2]

Neka je G težinski graf sa skupom akcija E i težinskom funkcijom $\omega : E \rightarrow \mathbb{R}^+$. *Uređeni par* $(c,d) \in E \times E$ *je redukcijski par ako zadovoljava uvjete:*

1. $\omega(d) > \omega(c)$ *ili* $\omega(d) = \omega(c) \rightarrow c \leq_{leksikografski} d$
2. *za svaki* $u \in S$, *d je primjenjiv na u akko je c primjenjiv na u*
3. *za svaki* $u \in S$, $d(u) = c(u)$ *ako su oba primjenjiva na u (postižu isti rezultat).*

Primjenjivost akcije ovisi o samom problemu. Npr., u primjeru labirinta primjenjiva akcija je kretanje u slobodnom smjeru, a neprimjenjiva kretanje direktno prema neposrednoj prepreci (Slika 7).



Slika 7. Primjer dozvoljene i zabranjene akcije.

Ako su zadovoljena sva tri uvjeta iz definicije redukcijskog para, nije više potrebno tražiti prečace već se dovoljno osloniti na pronađene duplikate koje se uklanja iz grafa. Očito je da ovaj način smanjenja prostora stanja čuva optimalna rješenja.

2.1.2. Redukcijski automati

Problem s prethodnim pristupom duplikatima je to što broj duplikata može biti iznimno velik pa gore opisana pretraga može potrajati. Primjer problema gdje se to može dogoditi je traženje uzorka u tekstu. Poznat je pod nazivom bibliografijski problem pretraživanja i može se riješiti konstrukcijom automata s konačnim brojem stanja, „primatelja podstringova“ (eng. substring acceptor) [5]. Njemu se proslijedi dotični tekst i on radi sinkronizirano s originalnom pretragom. Stablo se obrezuje kada automat prijeđe u stanje primatelja. Reduciranje podstringova je korisno kod algoritama kao što je IDLA* (incremental duplicate learning A*) jer radi ukorak s originalnom pretragom i dinamički uklanja suvišne grane stabla što znatno ubrzava algoritam.

Kako redukcija stabla uklanja njegove grane, ono zapravo smanjuje njegov faktor grananja. Automati se mogu primijeniti i u svrhu predviđanja buduće veličine stabla. Jedan od važnijih algoritama koji koristi deterministčke automate s konačnim brojem stanja je Aho-Corasick algoritam.

Aho-Corasick algoritam

Neka je zadan niz stringova ukupne duljine m (suma duljina svih stringova). Aho-Corasick algoritam konstruira strukturu podataka sličnu stablu (stablo uz nekoliko dodatnih bridova) pa zatim automat [6].

Izrada stabla:

Počevši od korijena, svaki čvor koji se dodaje u stablo označava se slovom i to na način da put od korijena do svakog lista odgovara jednom od zadanih stringova. U skladu s tim, svakom listu se dodjeljuje vrijednost koja će biti odgovarajući string. Postiže se jedan-na-jedan korespondencija između stingova i listova stabla [6].

Potrebno je ustanoviti strukturu podataka koja će sadržavati sve ove informacije o listovima. Ovdje je to struktura „struct“ koja sadrži zastavicu, odnosno varijablu tipa bool koja označava je li trenutni čvor list ili nije, i listu čvorova djece trenutnog čvora. Indeks *i* označava poziciju čvora u listi i poprima vrijednost -1 ako idućeg čvora nema (trenutni čvor je list) [6].

```
const int K = 26;

struct Vertex {
    int next[K];
    bool leaf = false;

    Vertex() {
        fill(begin(next), end(next), -1);
    }
};

vector<Vertex> trie;
```

Kod 1. Deklaracija stabla [6]

Idući korak je implementacija funkcije koja dodaje stringove u stablo. Počevši od korijena, prate se bridovi (dok god postoje) koji odgovaraju znakovima sadržanim u stringu. Ako nema brida za neki znak, stvori se novi brid. Proces staje kad se dosegne kraj stringa i zadnji čvor (koji referira na zadnje slovo) postaje list.

```
void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (trie[v].next[c] == -1) {
            trie[v].next[c] = trie.size();
            trie.emplace_back();
        }
        v = trie[v].next[c];
    }
    trie[v].leaf = true;
}
```

Kod 2. Dodavanje stringa [6]

Izrada automata:

Gledajući svaki vrh, njegov odgovarajući string je prefiks jednog ili više zadanih stringova. Prema tome, svaki vrh može se shvatiti kao pozicija u jednom ili više stringova. Štoviše, oni se mogu shvatiti kao stanja determinističkom automatu s konačnim brojem stanja. Iz svakog stanja može se prijeći u drugo koristeći neko novo slovo. Npr., ako je trenutni string „ab“ kojem odgovara stanje broj 2, do stringa „abc“ kojem odgovara stanje 3 može se doći dodavanjem znaka „c“ [6].

Stoga, bridove stabla moguće je shvatiti kao prijelaze u automatu s obzirom na odgovarajuće slovo. Međutim, u automatu ne može postojati restrikcija prijelaza iz jednog stanja u drugo. To je problem jer u slučaju pokušaja prijelaska iz stanja u stanje koristeći neko slovo, ne smije se dogoditi da odgovarajućeg brida nema. Točnije rečeno, prijelaz se svejedno mora obaviti u *neki* čvor. Tada je potrebno pronaći stanje koje odgovara najdužem sufiksu trenutnog stringa i pokušati izvršiti prijelaz na tom čvoru. Npr., neka se u stablu nalaze stringovi „ab“ i „bc“. Ukoliko je trenutna pozicija čvor „ab“ i željeni prijelaz je koristeći znak „c“, algoritam je prisiljen pronaći čvor „b“ i tamo izvršiti prijelaz (jer je tu moguć) [6].

Sufiks veza (eng. suffix link) za čvor *p* je brid koji je usmjeren prema najdužem sufiksu stringa koji odgovara čvoru *p*. Poseban slučaj je korijen stabla gdje će sufiks veza biti petlja (pokazivat će na isti čvor odakle je počela). Sada je moguće preformulirati uvjete za prijelaze na sljedeći način: dok god iz trenutnog čvora nema prijelaza koristeći trenutno slovo ili dok se ne dosegne korijen, slijedi se sufiks veza [1].

Problem konstrukcije automata sveden je na pronalazak sufiks veza za sve čvorove stabla. One će se pak izgraditi koristeći prijelaze. Korisno je primijetiti da se za pronalazak sufiks veze nekog čvora *v* može ići do njegovog pretka *p* i onda slijediti *njegovu* sufiks vezu te tada odraditi prijelaz traženim znakom. Sada je i problem traženja sufiks veza pojednostavljen i sveden na traženje jedne sufiks veze i prijelaza i to za čvorove bliže korijenu. Zato postoji rekurzivna ovisnost koja se može riješiti u linearnom vremenu [1].

Sljedeći kod odgovara gore opisanom postupku. Slovom *p* označava se čvor predak, a rječju *pch* znak koji odgovara prijelazu iz *p* u *v*, za svaki čvor *v*. Također, za svaki čvor će se pohraniti sufiks veza u varijabli *link* koja će imati vrijednost -1 ako veza još nije izračunata. U listi *go[k]* bit će sadržani prijelazi automata za svaki simbol gdje će opet vrijednost -1 biti korištena ako prijelaz još nije izračunat [6].

```
const int K = 26;

struct Vertex {
    int next[K];
    bool leaf = false;
    int p = -1;
    char pch;
    int link = -1;
    int go[K];

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);
```

```
void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].leaf = true;
}

int go(int v, char ch);

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
}
```

Kod 3. Implementacija Aho-Corasick algoritma [6]

Traženje podstringova svodi se na postupak linearne vremenske složenosti koristeći pamćenje sufiks veza i prijelaza. Zbog toga, vrijedi sljedeći rezultat:

Teorem: najraniji pronalazak k stringova ukupne duljine d u tekstu duljine n moguće je ostvariti u vremenu O(n+d) [1].

Aho-Corasick algoritam ima mnoge primjene, a neke od najčešćih su [1]:

- pronalazak zadanih riječi u tekstu,
- pronalazak leksikografski najmanjeg stringa zadane duljine koji se razlikuje od svih stringova u nekom zadanom skupu,
- pronalazak najkraćeg stringa koji sadrži sve zadane podstringove,
- pronalazak leksikografski najmanjeg stringa duljine l koji sadrži k stringova.

Incremental duplicate learning A*

Jedan od izazova je dinamičko primjenjivanje redukcije podstringova, tj. tijekom same izvedbe algoritma pretrage. Generalizirana sufiks-stabla omogućavaju brisanje i dodavanje čvorova uz to čuvajući brzinu traženja substringova. Budući da Aho-Corasick algoritam nije dorastao tom zadatku, javila se potreba za nekim drugim algoritmom [1].

Za razliku od strukture rječnika, generalizirana sufiks stabla omogućavaju umetanje i brisanje čvorova tijekom pretrage uz to održavajući brzinu traženja podstringova. Za dinamički algoritam koji uči, spaja se detekcija duplikata s algoritmom A*. Rezultirajući algoritam naziva se Incremental duplicate learning A* (IDLA*) sa sljedećim pseudokodom:

Algorithm 1 IDLA*

Ulaz: ulaz
Izlaz: izlaz

```
1: Otvoren ← {s}; Zatvoren ← ∅; q(s)← q0
2: while Otvoren ≠ ∅ do
3:     u ← argmin_f Otvoren
4:     Otvoren ← Otvoren\{u}; Zatvoren ← Zatvoren ∪{u}
5:     if Cilj(u) then return Staza(u)
6:     end if
7:     Sljedbenik(v) ← Proširi(u)
8:     for all v ∈ Sljedbenik(u), u -a-> v, a ∈ Σ do
9:         q(v) ← Δ(q(u),a)
10:        if q(v)∈F then
11:            nastavi
12:        end if
13:        if v ∈ Zatvoren then
14:            v' ← Pronađi(Zatvoren, v)
15:            if w(Staza(v)) < w(Staza(v')) then
16:                m ← Staza(v')
17:            else
18:                m ← Staza(v)
19:            end if
20:            m ← m [Najduži_zajednički_prefiks(Staza(v), Staza(v'))..|m|]
21:            D ← D∪{m}
22:        end if
23:        Poboljšaj(u,v)
24:    end for
25: end while
```

Slika 8. Pseudokod algoritma IDLA* [1].

Algoritam uzima problem prostora stanja i rječnički automat D kao ulazne parametre. Ako su pronađeni uređeni parovi (d, c) pruning parovi (eng. *pruning pair*), tada se pohranjuje samo string d. Inače, dodatno se pohranjuje i prečac c te se vrši provjera primjenjivosti. U svakom slučaju, zahvaljujući optimalnosti A* algoritma, IDLA* također daje optimalno rješenje [1].

2.1.3. Detekcija slijepih ulica

Postoje domene problema koje sadrže operacije koje nemaju inverz, tj. jednom napravljena akcija se ne može povući. To vodi pojavama slijepih ulica, što je problem. Naime, ako se algoritam nađe u jednoj, to znači da ne može dalje napredovati prema rješenju. Zapravo rješenje ostaje van dohvata algoritma.

Cilj pronalaska slijepih ulica je naći i prepoznati ovakve pojave i to što je prije moguće. To se radi prepoznavanjem grana u stablu odluke koje vode prema ovakvim slijepim ulicama. Postoje algoritmi koji specifično ciljaju na rješavanje ovog problema [1].

Vrijedi [1]:

- slijepa ulica može se definirati rekurzivno: kao trenutna pozicija (koja ne vodi nigdje dalje) ili kao pozicija koja nije ciljna ali svaki mogući potez s nje vodi u slijepu ulicu.
- Moguće je definirati relaciju < na skupu problema na sljedeći način: $v < u$ ako je v pod problem od u. Sada se zna da u nužno nije rješiv ako v nije rješiv.

Pohranjivanje rješenja pod problema naziva se pamćenjem. Cilj je naučiti i generalizirati slijepe ulice kada se na njih naiđe u pretrazi kako bi se saznanja o njima mogla iskoristiti kao redukcijski alat u daljnjoj pretrazi, čime se prodire dublje u stablo pretrage režući nepotrebne grane [1].

Dekompozicija stabla može se pobuditi na svakom koraku, svakih nekoliko koraka ili samo u posebnim situacijama. Potrebno je osigurati da detekcija slijepih ulica bude brza i da pretraga ne ide preduboko u stablo. Potproblemi bi trebali biti jednostavniji za riješiti od nadproblema, inače dekompozicija nema smisla [1].

2.1.4. Redukcija prostora stanja na temelju simetrije

Potencijalne simetrije u prostoru stanja bitan su faktor pri njegovom smanjivanju. Ako postoji više akcija koje se mogu po nekom kriteriju poistovjetiti, dovoljno je uzeti samo jednog predstavnika te skupine. Očito je potrebno uvesti pojam relacije ekvivalencije [1].

Definicija: Relacija ~ je relacija ekvivalencije ako je [7]*:*

- *refleksivna ($u \sim u$),*
- *simetrična ($u \sim v \rightarrow v \sim u$),*
- *tranzitivna ($u \sim v \wedge v \sim w \rightarrow u \sim w$).*

Svaka relacija ekvivalencije inducira klase ekvivalencije.

Definicija [7]*: Neka je ~ relacija ekvivalencije na skupu A. Za svaki $a \in A$, skup*

$$[a] = \{ x \in A : x \sim a \} \subseteq \mathrm{A}$$

svih elemenata iz A koji su u relaciji ~ s elementom a, tj. koji su s njim ekvivalentni naziva se klasa ekvivalencije. Element a naziva se reprezentantom te klase.

Dakle, za predstavljanje cijele jedne klase ekvivalencije dovoljno je uzeti jednog njenog predstavnika. Simetrije će se tretirati upravo na ovaj način.

Definicija [1]*: Bijekcija $\psi : S \rightarrow S$ je simetrija ako:*

- *$\psi(s) = s$,*
- *$\psi(t) \in T$, za svaki $t \in T$,*
- *za svakog nasljednika v od u postoji akcija koja vodi od $\psi(u)$ do $\psi(v)$.*

Svaki skup simetrija Y generira podgrupu g(Y) koja se naziva grupom simetrije. Podgrupa g(Y) inducira relaciju ekvivalencije $\sim_Y$ na stanjima definiranu na sljedeći način:

$$u \sim_Y v \text{ ako i samo ako } \psi(u) = v \text{ i } \psi \in g(Y).$$

Takva relacija ekvivalencije naziva se relacijom simetrije na P induciranom s Y.

Dakle, bijekcija koja preslikava početno stanje u početno stanje, ostala stanja permutira uz očuvanje njihove međusobne povezanosti (u odnosu na zadanu akciju) će biti simetrija. Skup simetrija ima svojstva grupe.

Definicija [8]: *(grupa)*

Neprazan skup X naziva se grupa ako je na njemu definirana binarna operacija × sa svojstvima:

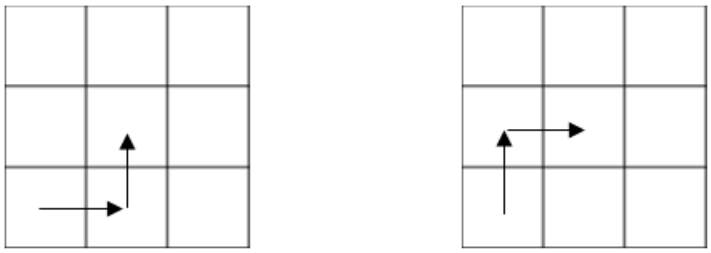
$$(\text{za svaki } x,y,z \in X)\ x \times (y \times z) = (x \times y) \times z \text{ (asocijativnost)}$$

$$(\text{postoji } e \in X \text{ tako da za svaki } x \in X)\ x \times e = e \times x = x \text{ (postojanje neutralnog elementa)}$$

$$(\text{za svaki } x \in X \text{ postoji } x' \in X)\ x \times x' = x' \times x = e \text{ (invertibilnost elemenata).}$$

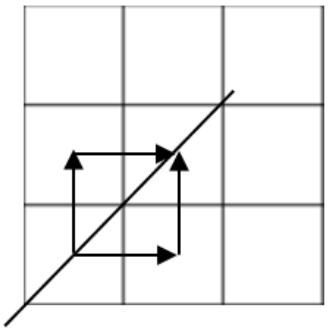
Ta grupa inducira relaciju ekvivalencije na skupu stanja. Dva stanja su ekvivalentna ako i samo ako postoji neka simetrija iz grupe simetrija koja preslikava jedno stanje u drugo. Drugim riječima, dva stanja su ekvivalentna (a time jednake važnosti) ako su simetrična s obzirom na neku zadanu funkciju simetrije (npr. osnu simetriju u ravnini).

Primjer na labirintu:



Slika 9. Ekvivalentni pokreti

Ako u labirintu postoji slobodan prostor, moguće je da se dogodi situacija na gornjoj slici. Moguće je od jednog polja do drugog doći na dva različita načina, i to putevima koji su jednake duljine. Ako je jedini kriterij optimalnosti puta njegova duljina, onda će ova dva puta biti jednake važnosti, tj. bit će ekvivalentni. U ovom slučaju, funkcija simetrije je upravo osna simetrija s obzirom na pravac provučen kroz središta početnog i krajnjeg polja poteza.



Slika 10. Osno simetrične akcije.

2.2. Redukcija prostora stanja uz očuvanje rješenja

Tehnike reduciranja prostora stanja uz očuvanje rješenja ne čuvaju (nužno) optimalno rješenje problema. One dozvoljavaju znatno smanjenje prostora stanja po cijenu optimalnosti, čime zapravo povećavaju učinkovitost algoritma pretrage u pronalasku rješenja problema. Tri su glavne faze u provođenju reduciranja prostora stanja uz očuvanje rješenja [1]:

- dodavanje makro akcija – promatranje prostora stanja nastalih od polaznog dodavanjem tzv. makro akcija,
- odbacivanje nebitnog – sprječavanje algoritma da isproba sve moguće operacije u svakom stanju,
- redukcija pomoću parcijalnog uređaja – iskorištavanje komutativnosti operacija i smanjenje prostora stanja s obzirom na parcijalno zadan cilj.

2.2.1. Dodavanje makro akcija

Ponekad je moguće grupirati akcije i proglasiti ih novom cjelinom. Na ovaj način je moguće izvršiti više manjih operacija odjednom. Novonastali složeniji operatori zamjenjuju prethodne i time se smanjuje skup operatora prostora stanja. Učinak se postiže u vidu smanjenja količine odluka koje je moguće donijeti pri prelasku iz stanja u stanje. Dakle, potrebno je provesti mnogo manje provjera kako bi algoritam odlučio svoj idući korak. Naravno, ne smije se pretjerati s grupiranjem operacija jer tada nastaje opasnost da cilj ostane van dohvata [1].

Ovdje će se promotriti strategija koja ne čuva optimalna rješenja, ali osigurava postojanje barem jednog rješenja. Makro operator je uređena n-torka n operacija koje se izvršavaju zajedno. Formalnije, makro operator odgovara dodatnom bridu između 2 čvora grafa koji skraćuje put od jednog do drugog čvora [1].

Makro operatori pretvaraju ne težinski graf u težinski. Težina brida koji odgovara makro operatoru bit će zbroj težina operacija koje on sadrži. Kako dodavanje bridova u graf neće nikad smanjiti stupnjeve čvorova, to dodavanje makro operatora neće narušiti rješivost problema. Najočitiji postupak koji se nameće je povezivanje svaka dva čvora grafa, odnosno nadopuna grafa do potpuno povezanog. Nažalost, takva strategija je „preskupa“ [1].

Budući da glavni cilj ipak nije sačuvati optimalna rješenja nego naći jedno od rješenja, dozvoljeno je koristiti tzv. ne-dopustive makro operatore – one koji ne uzimaju nužno najkraći

put od čvora do čvora. Druga opcija je brisati dodane bridove nakon što se iskoriste. Važnost makro operatora je u tome što se mogu dodati prije izvršenja algoritma pretrage na čitavom grafu. Ovaj proces se naziva makro učenje [1].

Jedan način na koji se umeću ne-dopustivi makro operatori je taj da se u graf doda odgovarajući brid s težinom manjom od one optimalnog brida kako bi se novom bridu povećao prioritet. Zapravo se algoritam pretrage prisili proći željenom stazom. Drugi način je ograničiti pretragu samo na makro operatore, zanemarujući originalne bridove. Naravno, ovo je moguće samo ako ciljno stanje ostaje dostupno [1].

Problem se dekomponira u uređenu listu podciljeva, a za svaki od njih se definira skup makro operatora koji transformiraju stanje u idući podcilj. Za svako stanje se odredi podcilj kojem se tada pristupa primjenom makro operatora [1].

Konstruira se makro tablica, koristeći BFS ili DFS unatrag, odnosno s početkom u svakom cilju i to upotrebom inverznih operatora [1].

2.2.2. Odbacivanje nebitnog

Akcije mogu biti bitne ili nebitne, s obzirom na neki zadani kriterij (npr. brzina pretrage). Važno je raspoznati koje su koje jer ukidanje nebitnih akcija bitno ubrzava pretragu. Npr., IDLA* algoritam istražuje svaku moguću opciju, ne uzimajući u obzir potencijalne podciljeve koji bi bitno ubrzali pronalazak glavnog cilja. Zapravo, algoritmi istražuju čak i one puteve koji ljudi nikad ne bi jer nemaju moć raspoznavanja bitnog od nebitnog. Zato se uvode rezovi nebitnog koji pokušavaju ograničiti algoritam kod izbora iduće akcije [1].

Važan pojam koji se koristi je pojam „utjecaja“ između akcija. Dvije akcije su „udaljene“ ako nemaju utjecaj jedna na drugu. Definicija udaljenosti ovisi o konkretnom problemu. Npr., međusobni utjecaj dva polja labirinta je obrnuto proporcionalan broju putova koji ih povezuju [1].

Postoje dva načina na koja se akcije mogu obrezivati. Prva opcija je vidjeti je li u zadnjih k pokreta zadan broj l udaljenih akcija napravljen. Ovaj način sprječava nasumičnu izmjenu pretraživanja odvojenih dijelova labirinta, tj. „skakanje“ s jednog mjesta na drugo [1].

Druga opcija je odbaciti akciju koja je udaljena od neposrednog prethodnika, ali nije od neke akcije u prošlih k. Ovo sprječava povratak na prethodno istražena područja koja su maloprije napuštena. Ako je parametar l jednak 1, prvi način podrazumijeva drugi [1].

Rezanje nebitnog ne garantira očuvanje optimalnih rješenja. Ipak, dodavajući element nasumičnosti rezovima, moguće je spriječiti obrezivanje optimalnih rješenja. To se postiže koristeći vjerojatnosni faktor koji određuje sigurnost reza [1].

2.2.3. Redukcija pomoću parcijalnog uređaja

Metode redukcije parcijalnog uređaja oslanjaju se na komutativnost operacija kako bi se postiglo smanjenje prostora stanja tako da novonastali prostor stanja bude ekvivalentan početnom. Algoritam koji reducira prostor stanja istražuje samo neku djecu svakog čvora, a ne svu. Skup dozvoljenih operacija za neki čvor naziva se dozvoljeni skup (eng. enabled set), a označava se s enabled(u). Algoritam izabire podskup ovog skupa koji se naziva dovoljni skup (eng. ample set), a označava sa ample(u). Stanje u naziva se potpuno proširenim ako vrijedi ample(u) = enabled(u), inače se naziva parcijalno proširenim [1].

Metoda redukcije parcijalnog uređaja temelji se na zapažanju da redoslijed izvršavanja pojedinih operacija nije bitan što vodi pojmu nezavisnosti operacija:

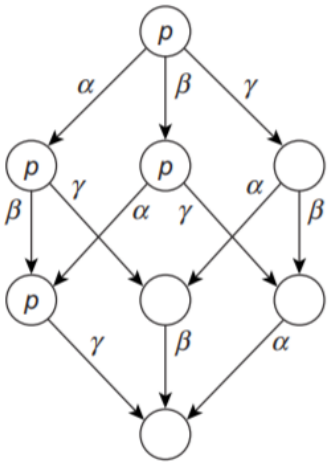
Definicija [1]*: Dvije akcije a_1 i a_2 su nezavisne ako vrijedi:*

- *a_1 i a_2 se međusobno ne isključuju,*
- *a_1 i a_2 su komutativne.*

Još jedan ključan pojam je nevidljivost akcija s obzirom na cilj.

Definicija [1]*: Akcija α je nevidljiva s obzirom na skup propozicija cilja T ako:*

$$\text{za svaki par stanja } u, v \text{ iz } v = \alpha(u) \text{ slijedi } u \cap T = v \cap T.$$



Slika 11.Akcije α, β i γ su u parovima nezavisne [1].

Glavna svrha konstrukcije dovoljnog skupa je izabrati takav podskup djece nekog stanja tako da reducirani prostor stanja bude ekvivalentan polaznom s obzirom na cilj. Konstrukcija bi trebala znatno smanjiti prostor stanja i pritom ne biti prevelikog troška. Također, dovoljno putova mora „preživjeti“ kako bi se moglo doći do točnog rezultata [1].

Sljedeća četiri uvjeta dovoljna su za osiguranje ispravne konstrukcije reduciranog prostora stanja za dani cilj [1]:

1. ample(u) je prazan točno onda kad je enabled(u) prazan
2. na svakoj stazi u polaznom prostoru stanja koja počinje na čvoru u, akcija koja ovisi o akciji iz ample(u) ne smije se dogoditi prije spomenute akcije iz ample(u)
3. ako stanje u nije potpuno prošireno, onda svaka akcija α iz ample(u) mora biti nevidljiva s obzirom na cilj
4. ako je za svako stanje u ciklusu reduciranog prostora stanja akcija α enabled, onda α mora biti u ample setu nekog nasljednika nekog čvora iz ciklusa.

Uvjeti 1,2,3 ili njihove aproksimacije mogu se implementirati odvojeno od algoritma pretrage. Provjeravanje uvjeta 1 i 3 relativno je lako i ne ovisi o izboru algoritma. Međutim, 2 se provjerava nešto teže. Štoviše, pokazano je da se 2 provjerava barem toliko teško koliko se nalazi cilj za cijeli prostor stanja. Zato se umjesto direktne provjere svojstva 2 provjerava neki jači uvjet koji se može provjeriti neovisno o algoritmu pretrage. Očito je da onda ispunjenje

jačeg uvjeta povlači ispunjenje slabijeg uvjeta, odnosno traženog uvjeta 2. Uvjet 4 ovisi o korištenom algoritmu pretrage [1].

Provjera uvjeta 4 može se svesti na traženje ciklusa tijekom pretrage. Oni se lako nalaze koristeći DFS: svaki ciklus sadrži „brid unatrag“ (eng. backward edge), odnosno brid koji povezuje zadnji čvor s nekim koji je već pohranjen na stogu pretrage (već istražen čvor). Stoga, izbjegavanje dovoljnih skupova koji sadrže bridove unatrag, osim kad je stanje potpuno prošireno, osigurava ispunjenje uvjeta 4 kad se koristi IDLA*, zato što IDLA* vrši DFS. Rezultirajući uvjet ciklusa temeljenog na stogu 4_{stog} može se izreći na sljedeći način [1]:

Ako stanje u nije potpuno prošireno, onda barem jedna akcija iz ample(u) ne vodi u stanje koje je sadržano u stogu pretrage.

Implementacija uvjeta 4_{stog} za strategije temeljene na DFS označi svako prošireno stanje na stogu tako da se zatvorenost stoga može provjeriti u konstantnom vremenu [1].

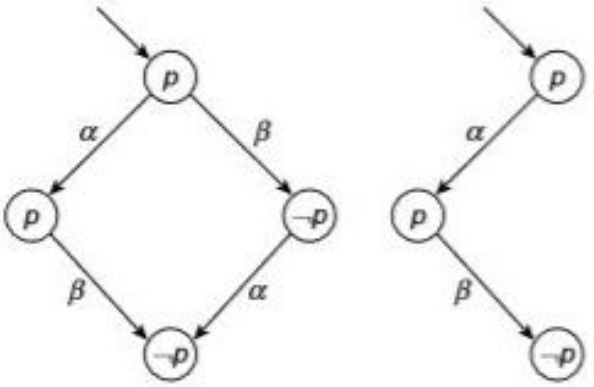
Detekcija ciklusa algoritmima koji ne koriste DFS je nešto kompliciranija. Da bi ciklus postojao, potrebno je vratiti se na već posjećeno stanje. Ako se tijekom pretrage naiđe na stanje koje je otprije posjećeno, da bi se provjerilo pripada li to stanje nekom ciklusu, potrebno je provjeriti je li moguće vratiti se na njega ako se krenulo od njega. Ovo povećava vremensku složenost algoritma na kvadratnu. Zbog toga se uzima da je stanje dio ciklusa čim je prethodno posjećeno. Ova ideja vodi slabijim redukcijama [1].

Provjera 4_{stog} ne može se koristiti bez stoga pretrage. Zato se uvodi alternativni uvjet koji garantira ispravnu redukciju [1]:

$4_{duplikat}$: *ako čvor u nije potpuno proširen, onda barem jedna akcija iz ample(u) ne vodi u prethodno istraženi čvor.*

Da bi se dokazalo da je korištenje uvjeta $4_{duplikat}$ točno za tehniku redukcije parcijalnog uređaja, koristi se indukcija na poretku istraživanja čvorova. Počinje se od potpuno istraženog stabla i vraća se unatrag po svim čvorovima u skladu sa provedenim algoritmom pretrage. Nakon kraja pretrage, za svaki čvor u sigurno je da se svaka dozvoljena akcija izvrši ili u skupu ample(u) ili na nekom čvoru koji se pojavljuje kasnije u pretrazi. Na taj način je osigurano da se nijedna akcija neće preskočiti. Činjenica da je ovaj postupak proveden nad svakim čvorom iz grafa garantira da je redukcija točna [1].

Redukcija parcijalnog uređaja čuva potpunost algoritma, tj. nijedno rješenje se ne gubi, ali ne čuva optimalnost. Štoviše, najkraća staza do rješenja u reduciranom prostoru stanja može biti duža nego najkraća staza do rješenja u polaznom prostoru. Razlog tome leži u činjenici da spomenuta ekvivalencija reduciranog i originalnog prostora stanja ne uzima u obzir dužinu blokova akcija.



Slika 12.Originalni i reducirani prostor. [1]

Ovaj problem može se umanjiti obradom rješenja. Ideja je ignorirati akcije koje su nezavisne s obzirom na one akcije koje vode do cilja jer one nisu bitne za pronalazak cilja. Pritom je potrebno osigurati da ukinute akcije ne mogu utjecati na dozvoljenost akcija koje vode prema cilju. Ovaj pristup može smanjiti duljinu puta prema cilju, ali istovremeno ne garantira da će konačno rješenje biti optimalno.

3. Primjena redukcije prostora stanja na labirint

Unutar ovog poglavlja pokazat će se implementacija redukcije prostora stanja pomoću programskog jezika Netlogo i analiza pomoću alata Behavior Space.

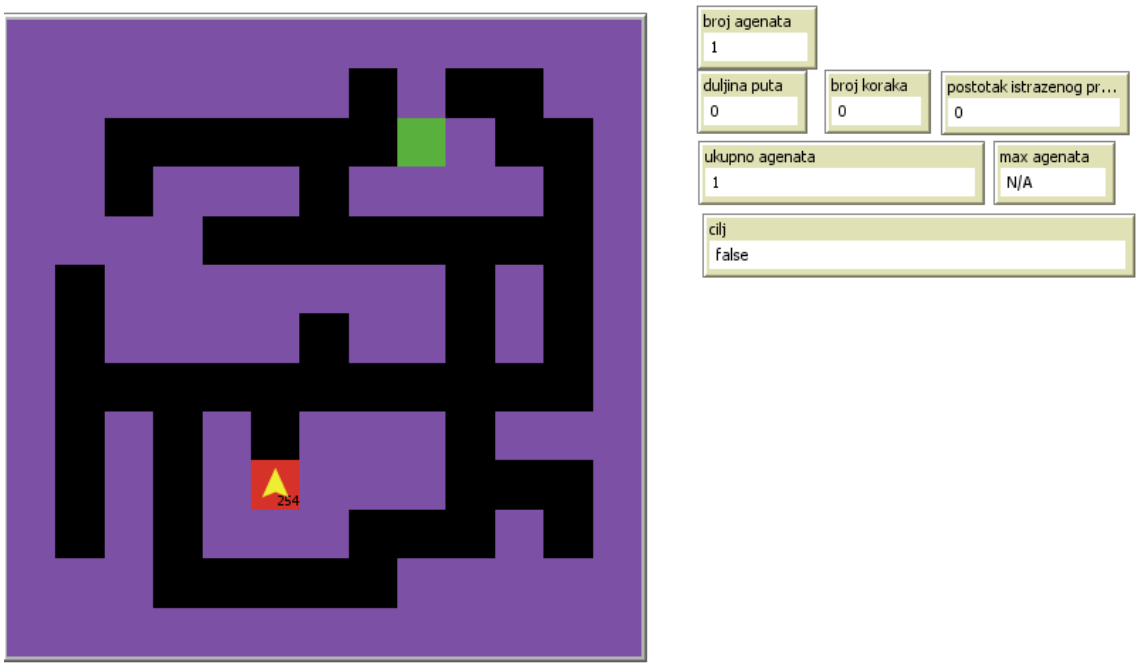
3.1. Analiza algoritma A* u kombinaciji s algoritmima redukcije

Kao demonstraciju neke od tehnika redukcije prostora stanja pokazat će se redukcija prostora stanja labirinta. Algoritam koristi procjenu udaljenosti do cilja kao kriterij procjene povoljnosti polja. Promatraju se tri različite metrike:

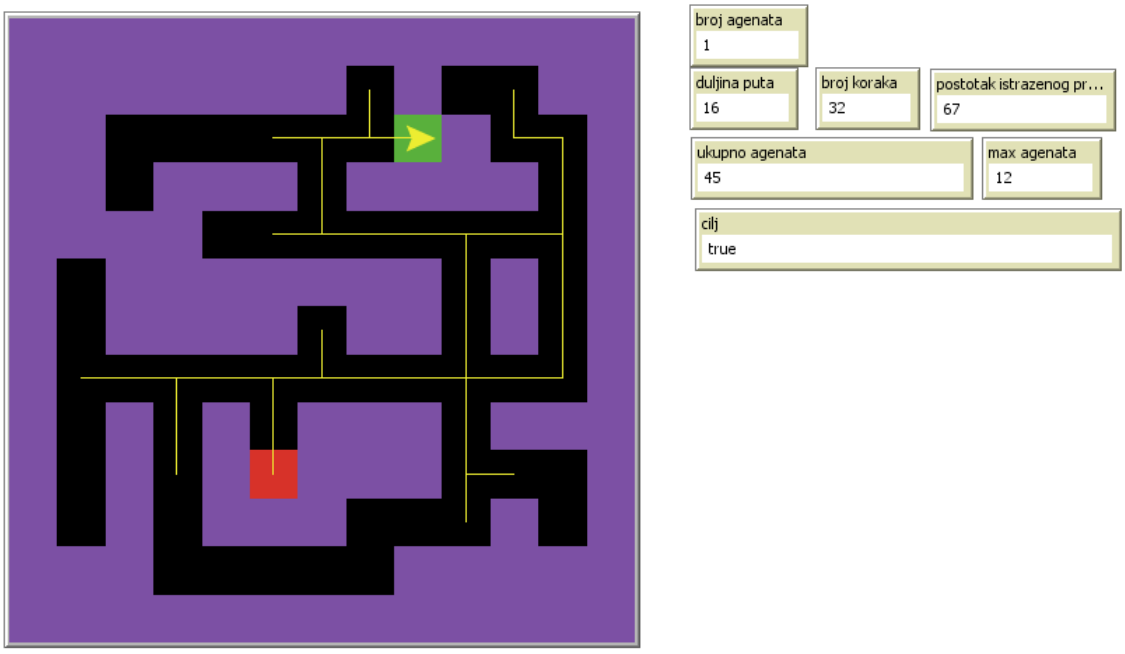
- $d_1((a_1, b_1), (a_2, b_2)) = |a_1 - a_2| + |b_1 - b_2|$ (Manhattan),
- $d_2((a_1, b_1), (a_2, b_2)) = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}$ (Euklidska),
- $d_\infty((a_1, b_1), (a_2, b_2)) = \max\{|a_1 - a_2| + |b_1 - b_2|\}$ (Čebiševljeva).

Za algoritam pretrage odabran je A* s heuristikom a = p + q, gdje p označava prijeđenu udaljenost, a q procjenu udaljenosti do cilja. Vrijednost q se mijenja ovisno o korištenoj metrici. A* je odabran i zbog toga što pronalazi optimalno rješenje pa se može vidjeti kako redukcija prostora stanja utječe na očuvanje optimalnosti algoritma pretrage. U ovom slučaju optimalno rješenje je najkraći put do cilja, zato što se prostor stanja (labirint) može poistovjetiti s ne težinskim grafom. Dakle, nema dodatnih faktora koje treba uračunati pri kretanju, odnosno svaki pomak u labirintu ima istu cijenu.

Zbog jednostavnosti pisanja State Space Pruning će se u daljnjem tekstu pisati skraćeno SSP.

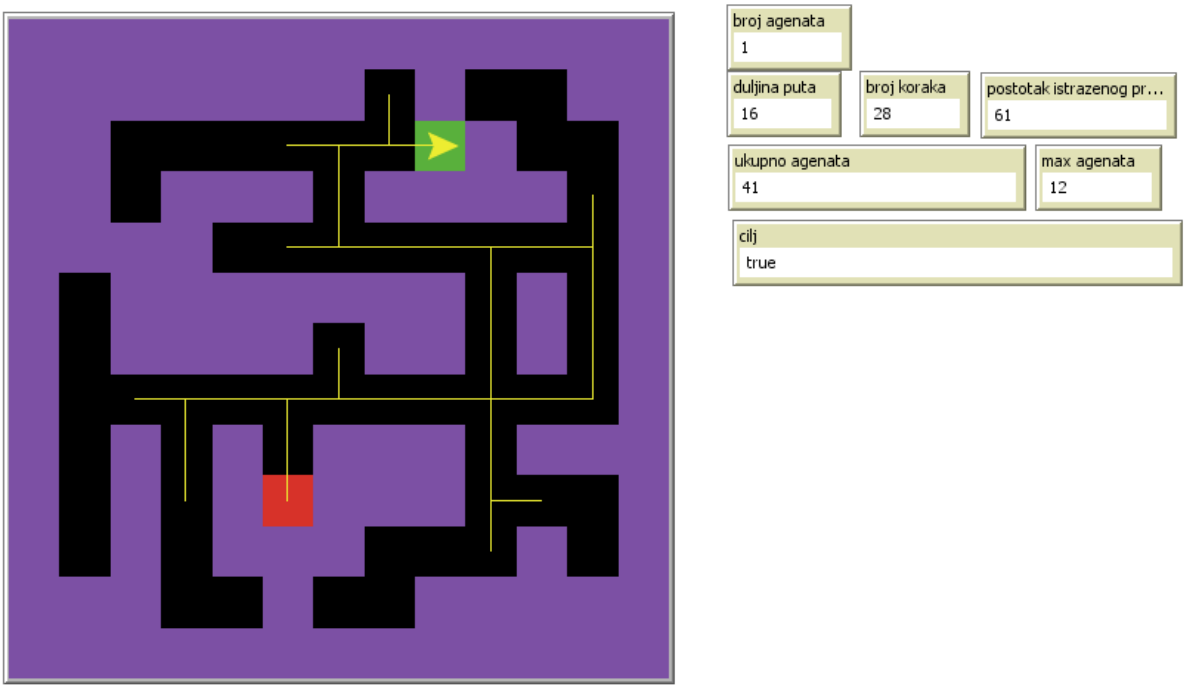


Slika 13. Početno stanje: (crveno polje je početak, zeleno cilj)



Slika 14. A* bez redukcije prostora stanja na labirintu s ciklusom

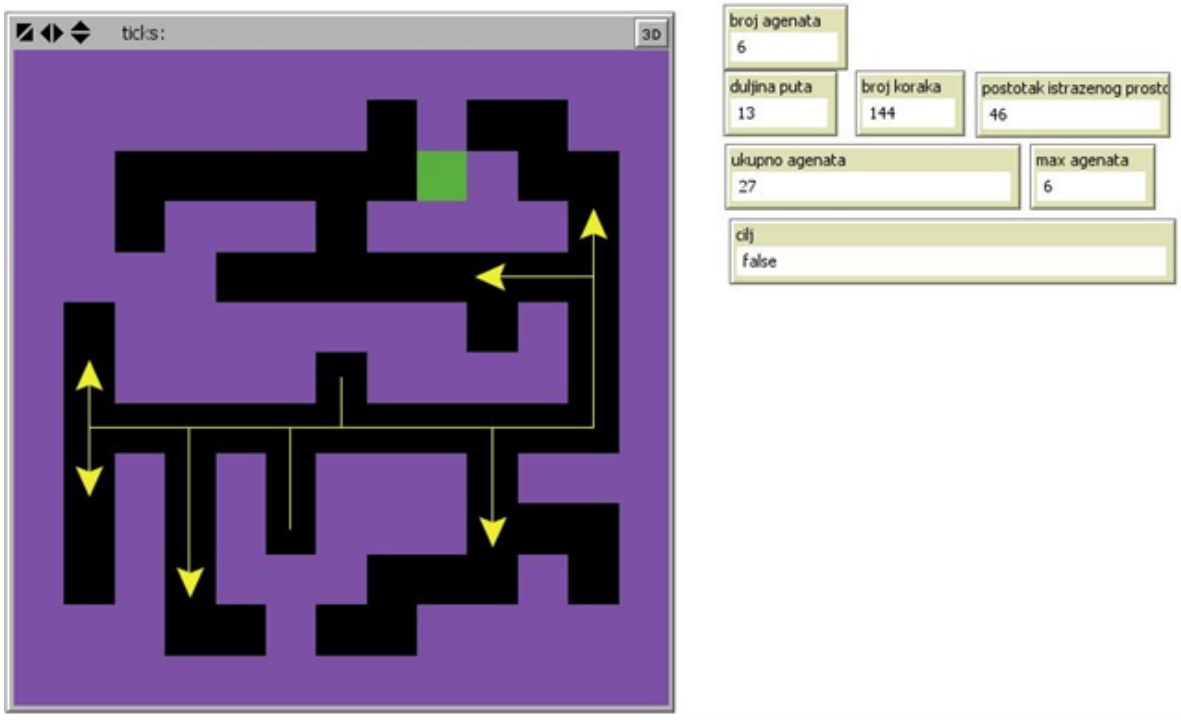
:



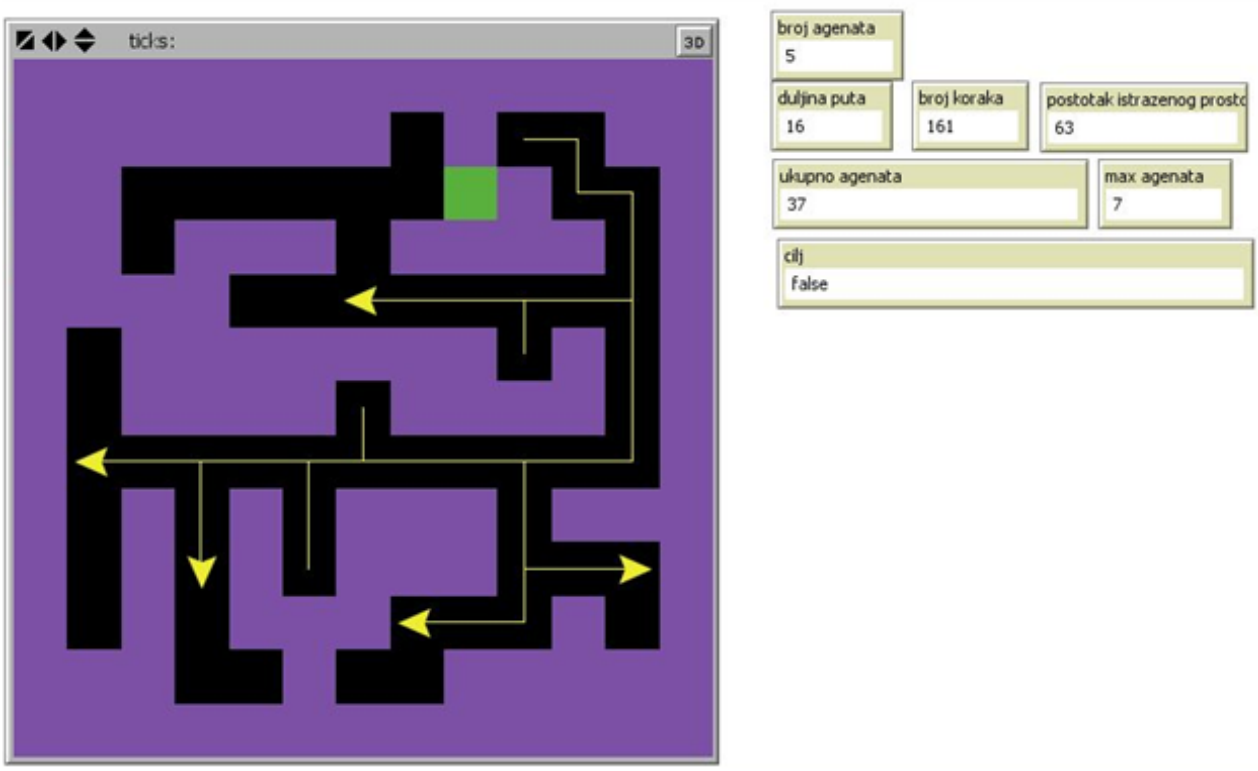
Slika 15. A* bez redukcije prostora stanja na labirintu bez ciklusa

Izneseni zaključci u odnosu na A* bez redukcije:

- Ciklusi na A* naizgled nemaju velikog utjecaja – tu pomaže činjenica da je labirint malen.
- Za A* algoritam korištena je samo Manhattan metrika (koja je standardni izbor metrike za labirinte gdje je kretanje agenata ograničeno na Von Neumannovo susjedstvo [9]).
- Algoritam je brži na labirintu s ciklusom – cilj je pronađen u 28 koraka, a na labirintu bez ciklusa u 43 koraka.
- Značajna promjena je i manji broj agenata na labirintu s ciklusom - to se dogodilo jer ima manje slijepih ulica pa se manje agenata odbacuje.
- Iz istog razloga je veći broj istovremeno aktivnih agenata – 10.

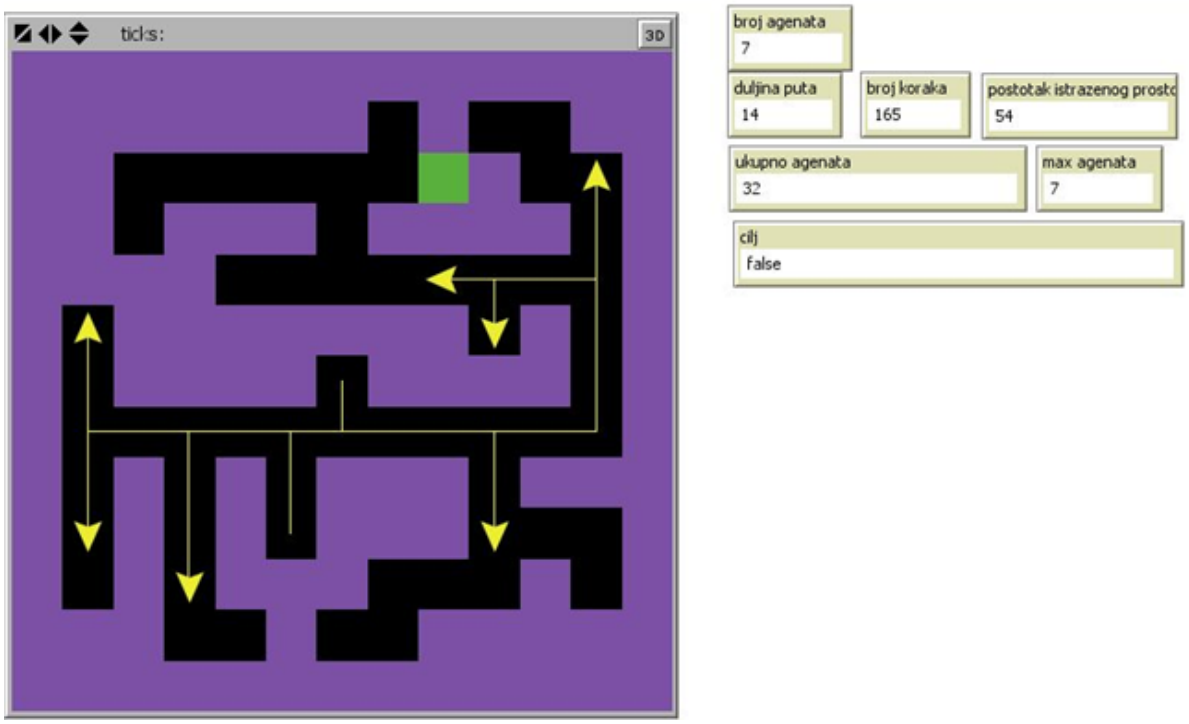


Slika 16. SSP na labirintu bez ciklusa, s euklidskom udaljenosti



Slika 17. SSP na labirintu bez ciklusa, sa Manhattan udaljenosti

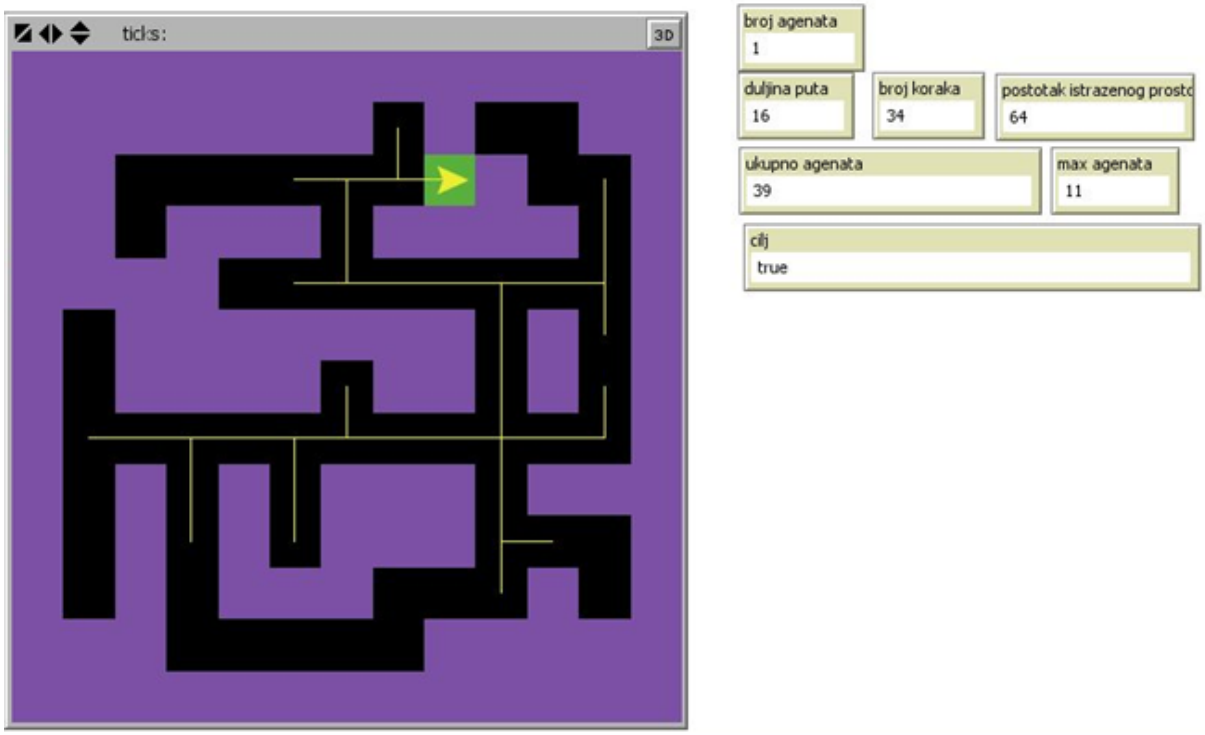
:



Slika 18. SSP na labirintu bez ciklusa, sa Čebiševljevom udaljenosti

Izneseni zaključci u odnosu na A* s redukcijom:

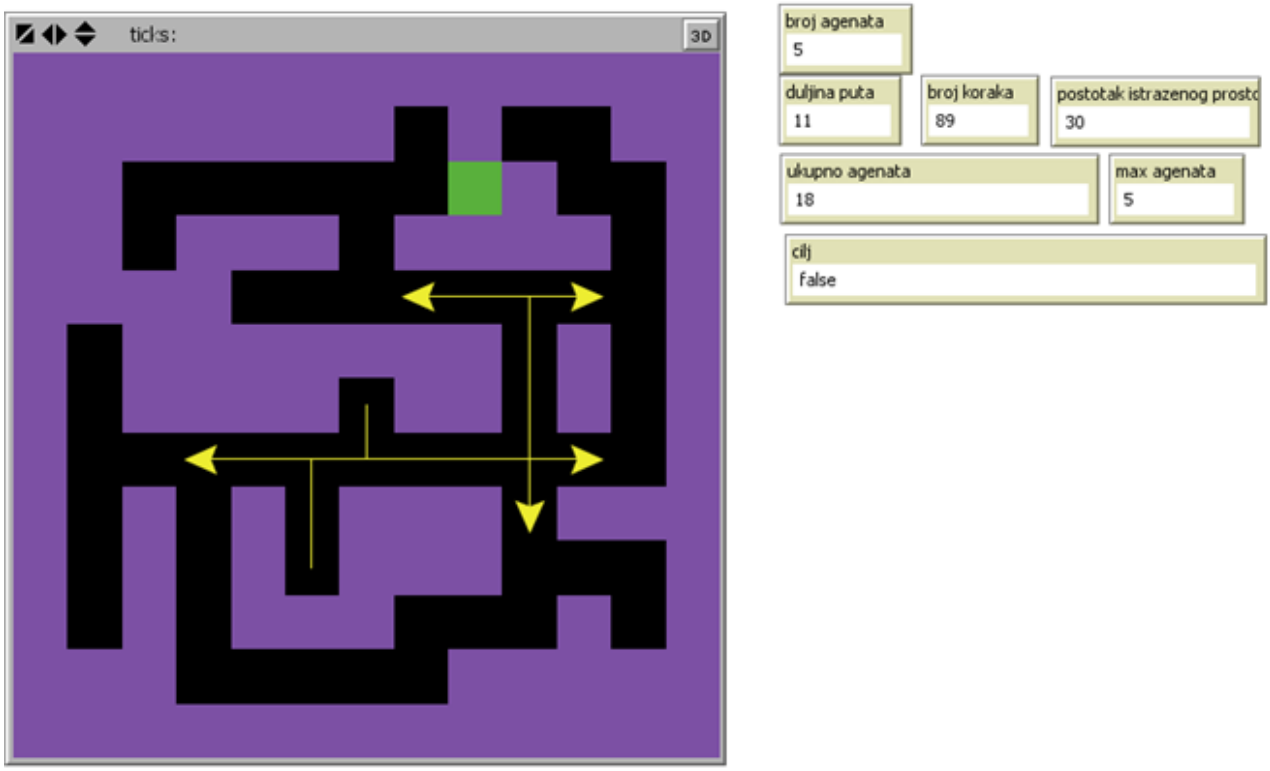
- Ovdje je očita činjenica da redukcija prostora stanja ne mora osiguravati pronalazak cilja – moguće je da se cilj ne pronađe.
- Ovaj rezultat proizlazi iz neadekvatno odabranog kriterija za redukciju labirinta, odnosno prevelik dio labirinta se zanemaruje pri pretrazi zbog toga što je kriterij nedovoljno precizan.
- Spomenuti kriterij je provjera je li procjena udaljenosti od cilja trenutnog agenta manja od aritmetičke sredine procjena udaljenosti od cilja svih agenata.



Slika 19. SSP na labirintu s ciklusom, s Manhattan udaljenosti

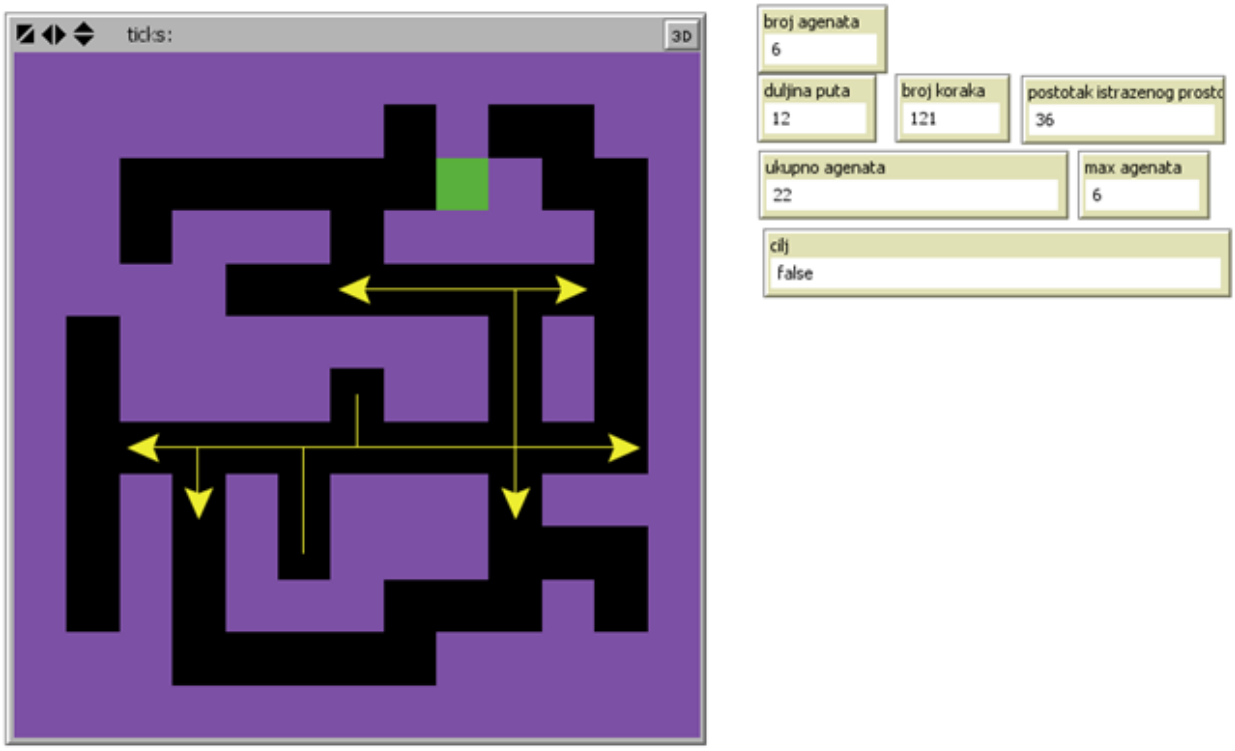
Izneseni zaključci u odnosu na A* s redukcijom s Manhattan metrikom :

- U ovoj iteraciji algoritma promijenjen je spomenuti kriterij: sada je umjesto aritmetičke sredine kao uvjet zadano malo više od polovice sume procjena udaljenosti od cilja svih agenata – 51%.
- Cilj se ne pronalazi ako se taj postotak smanji na manje od 50% globalne varijable pp (suma procjena udaljenosti do cilja svih trenutnih agenata).
- Uzrok tome leži u činjenici da se globalna varijabla pp dijeli s brojem trenutnih agenata koji može narasti i time pretjerano smanjiti vrijednost aritmetičke sredine.



Slika 20. SSP s ciklusom, s euklidskom udaljenosti

:



Slika 21. SSP s ciklusom, sa Čebiševljevom udaljenosti

Moguće je ne doći do traženog cilja s pretjeranom redukcijom stabla pretrage.

Zaključci:

- Vidi se da se cilj ne pronalazi ni za ostale dvije metrike ako se ovaj kriterij usporedbe ne promijeni.
- Dakle, zaključuje se da je moguće provesti adekvatnu tehniku redukcije prostora stanja tako da se zadrži mogućnost pronalaska cilja, ali je važno uzeti u obzir detalje računa, tj. matematičku pozadinu kriterija.
- Izbor metrike nije utjecao na pronalazak cilja, ali je utjecao na brzinu pronalaska cilja.

Promjena u kodu:

```
;set pp pp / (count(subjekti))
      set pp pp * 0.501
```

Kod 3. Promjena u kodu

Budući da je A* višeagentska pretraga, aritmetička sredina se nije pokazala prikladnom jer se globalna varijabla pp (suma procjena udaljenosti do cilja svih trenutnih agenata) dijeli s brojem trenutnih agenata koji može narasti i time pretjerano smanjiti vrijednost aritmetičke sredine.

Promjenom parametra s pp/broj agenata na pp*0.501 postiže se traženi pronalazak cilja unutar labirinta.

3.2. Analiza elementarnih algoritama i algoritama redukcije na većem broju labirinata

U sljedećoj tablici (Tablica 1) prikazani su rezultati Behavior Space analize algoritama redukcije ssp i ssp2 i usporedba s ostalim algoritmima.

Varijable:

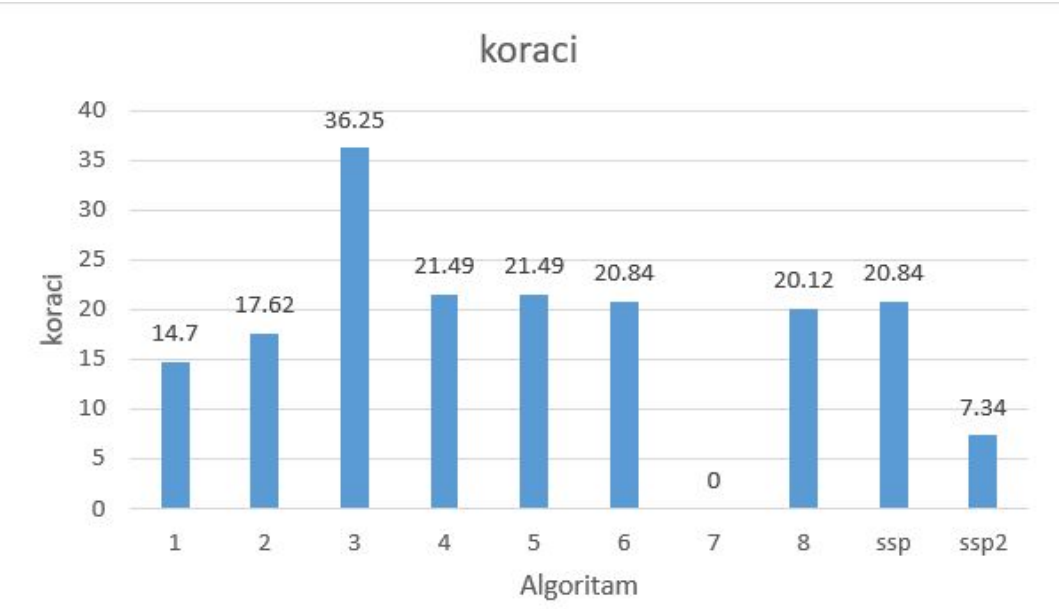
- koraci – broj koraka agenta od početka do cilja
- ticks – broj koraka algoritma
- prostor – količina istraženih polja
- b – ukupni broj agenata
- maxAg – najveći broj agenata koji su djelovali istovremeno
- % cilj – postotak uspješnog pronalska cilja

Algoritam	Koraci	ticks	prostor	b	maxAg	% cilj
Iterativna pretraga po dubini	14,7	873	55,25	1271,38	89,54	15
Dijkstra algoritam	17,62	696,68	74,47	1193,87	496,4	40
Pretraga po dubini	36,25	43,72	51,22	58,14	13,56	100
Valna fronta	21,49	20,49	21,10	1	1	100

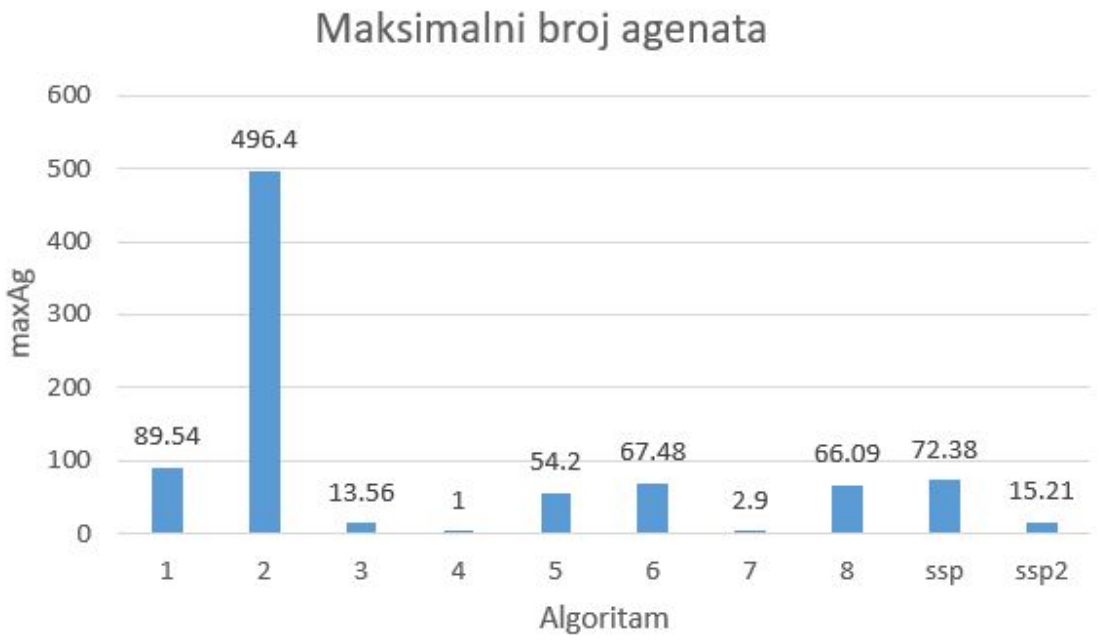
Pretraga po širini	21,49	20,49	85,27	359,15	54,2	100
A* algoritam	20,84	106,38	46,65	174,67	67,48	97
Ograničena pretraga po dubini	0	14,51	14,38	20,13	2,9	0
Pohlepna pretraga	20,12	102,28	32,34	169,31	66,09	92
State Space Pruning(SSP)	20,84	121,85	46,60	187,69	72,38	97
State Space Pruning 2(SSP2)	7,34	688,96	33,35	46,48	15,21	32

Tablica 1. Prosječne vrijednosti u odnosu na primijenjeni algoritam

Dijagrami rezultata:

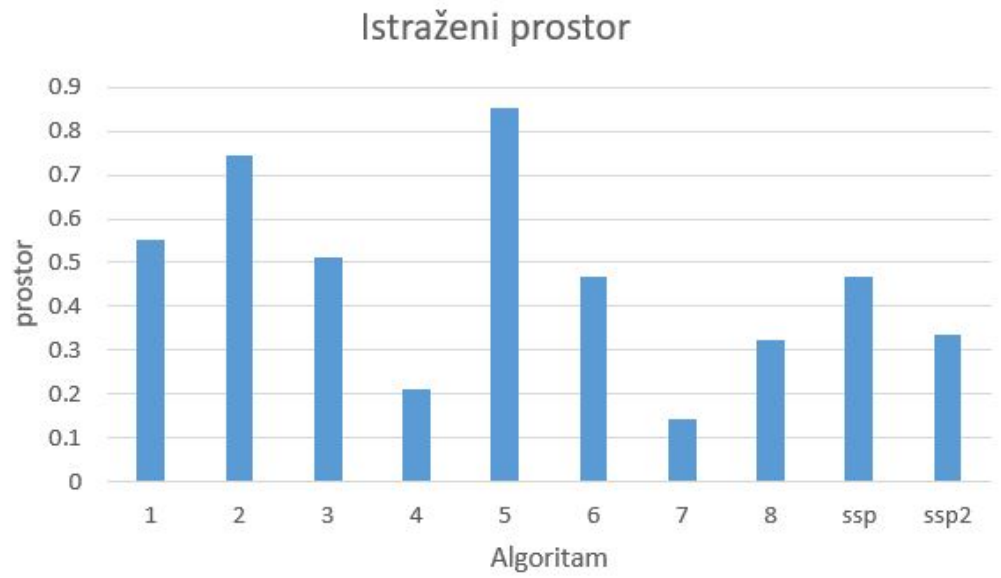


Slika 22. Korak u funkciji Algoritma

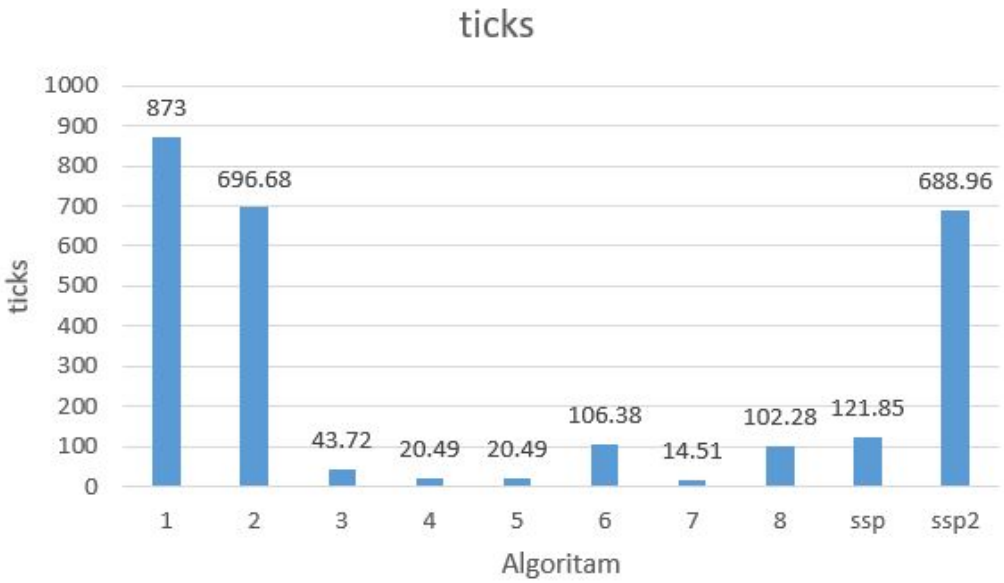


Slika 23. Maksimalni broj agenata u funkciji Algoritma

Slika 24. Postotak pronalaska cilja u funkciji Algoritma



Slika 25. Istraženi prostor u funkciji Algoritma



Slika 26. Broj iteracija algoritma u funkciji Algoritma

Algoritmi SSP i SSP2 temelje se na A* algoritmu. Mijenjaju njegovu heuristiku u ovisnosti o odabiru metrike za računanje udaljenosti od cilja. Uzimaju u obzir prosjek udaljenosti do cilja svih trenutnih agenata i na temelju te aritmetičke sredine biraju najpovoljnijeg agenta za idući korak pretrage. Na ovaj način se redukcija prostora stanja izvodi paralelno s odabranim algoritmom pretrage i zbog toga je labirint dovoljno istražiti jednom.

Postoje mnogi algoritmi redukcije koji se ne izvršavaju dinamički (paralelno s algoritmom pretrage). Oni se izvrše prije nego li se izvrši odabrani algoritam pretrage čime se dobiva svojevrsna slojevita pretraga. Očito se u ovom slučaju labirint istražuje više od jednog puta (minimalno dvaput, po jednom za svaki od algoritama). Jedan od takvih algoritama je Jump Point Search algoritam.

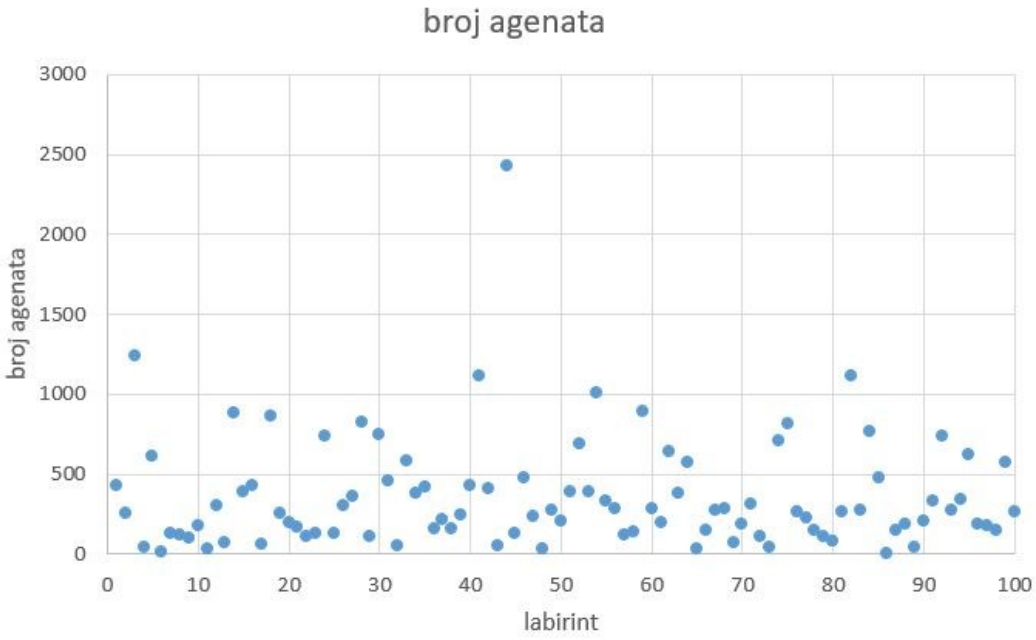
Jump Point Search algoritam

Jump Point Search algoritam (u daljnjem tekstu JPS) je algoritam redukcije prostora stanja koji definira tzv. odskočne točke (eng. jump point). One postaju jedina bitna polja labirinta u algoritmu pretrage koji se potom izvršava. Dakle, JPS bitno smanjuje broj polja labirinta kojima se agenti kreću. Samim time, smanjuje se količina prostora kojeg je potrebno istražiti, a tako i brzina algoritma pretrage [10].

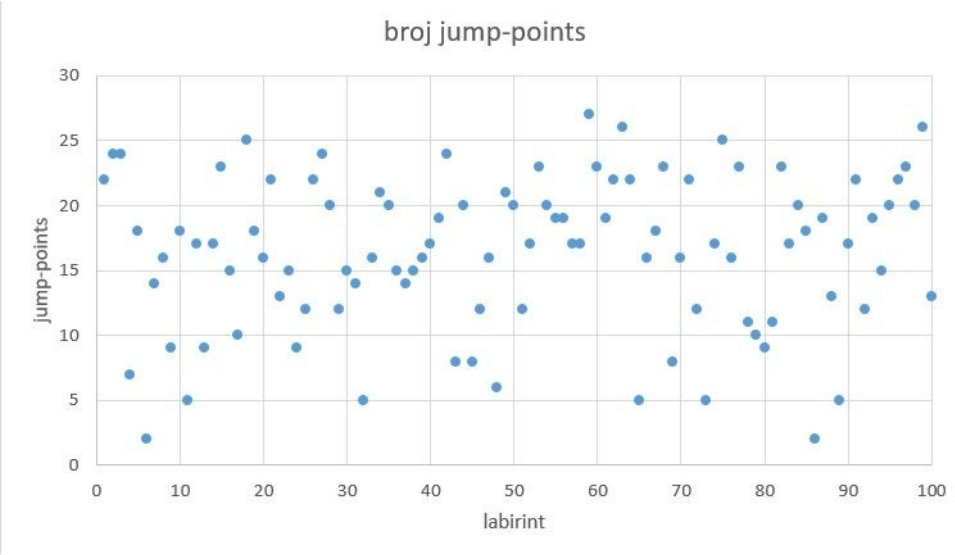
Ovdje se kao algoritam pretrage odabralo A*.

Koraci	Broj agenata	Jump Point	Slobodan proctor	Križanja
20,49	360,28	16,36	15,13	33,67

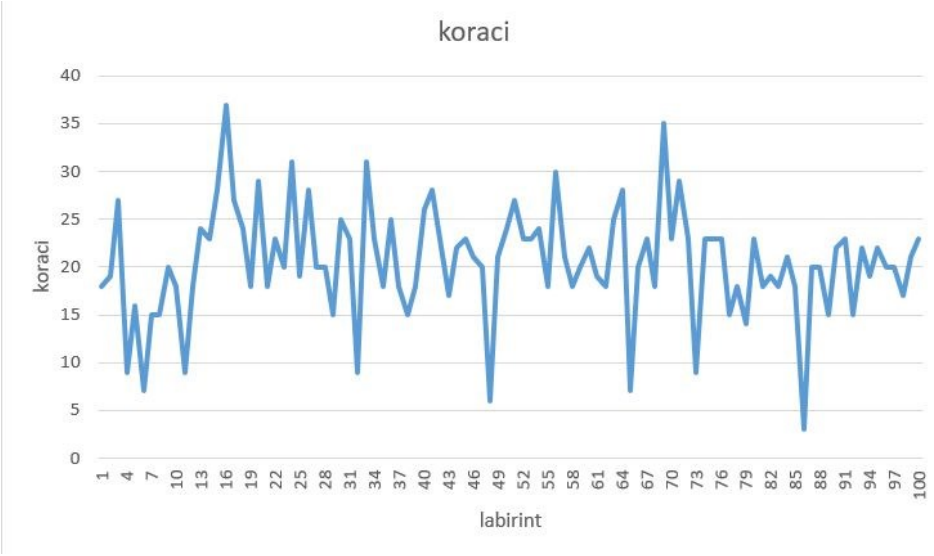
Tablica 2. Prosjeci varijabli JPS algoritma u kombinaciji s A* algoritmom



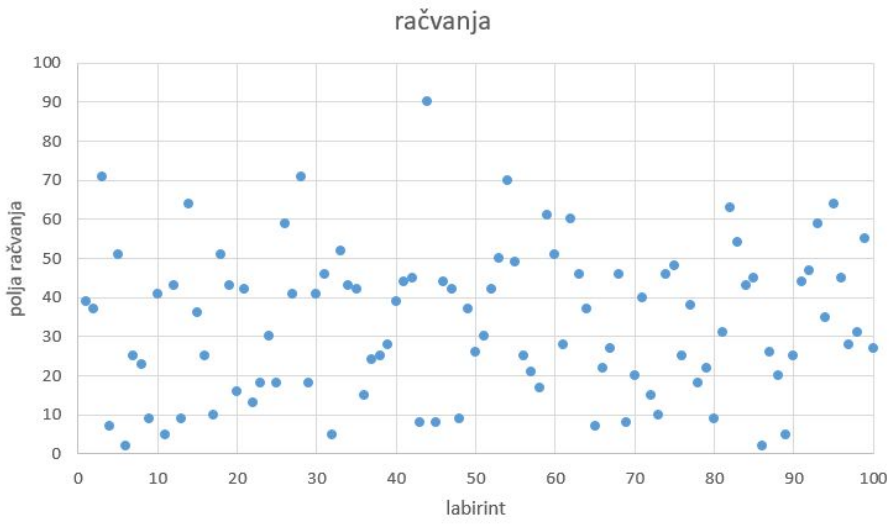
Slika 27. Broj agenata u funkciji Algoritma



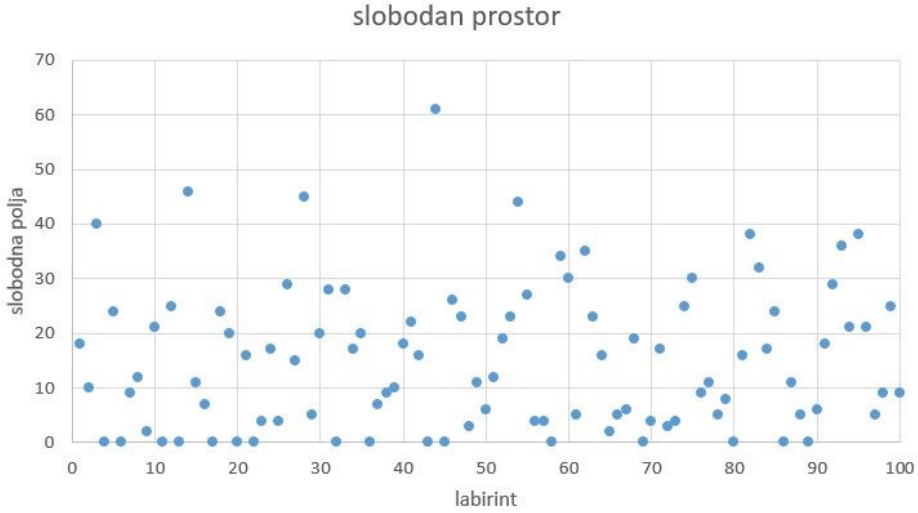
Slika 28. Broj odskočnih točki u funkciji Algoritma



Slika 29. Korak u funkciji Algoritma



Slika 30. Broj križanja u funkciji Algoritma



Slika 31. Slobodan prostor u funkciji Algoritma

Zaključak

Cilj ovog završnog rada je predstaviti kratak, ali potpun i cjelovit uvod u teoriju redukcije prostora stanja. Ova ideja optimizacije izvršavanja algoritama oslanja se na smanjivanje opcija koje je moguće istražiti, umjesto klasičnog pristupa promjene samog koda algoritma pretrage.

Dana je osnovna matematička podloga potrebna za smisleno razmatranje problema, kao i općeniti pregled raznih tehnika redukcije prostora stanja iz perspektive računalne znanosti. Posebnost redukcije prostora stanja je da se može primijeniti na proizvoljan heuristički algoritam, odnosno da je primjenjiv na brojne probleme koji se rješavaju spomenutim heurističkim algoritmima. Dakle, reduciranje prostora stanja je svestrani način pristupa optimizaciji algoritama i nije potpuno ovisan o konkretnom problemu ni algoritmu pretrage.

Ovdje su predstavljeni neki od poznatijih primjera algoritama koji sadrže ideje redukcije prostora stanja: Aho-Corasick algoritam i IDLA* (Incremental duplicate learning A*). Za prvi je prikazana C++ implementacija, a za potonji pseudokod. Također je dana kratka statistička obrada utjecaja redukcije prostora stanja primijenjene na labirint uz A* kao algoritam pretrage.

Literatura

[1] S. Edelkamp i S. Schroedl, Heuristic search Theory and Applicationn, Academic Press, 2012.

[2] A. Golemac, »Osnove teorije grafova,« Split, 2017.

[3] V. Matijević, Uvod u teoriju skupova, Split, 2010.

[4] B. Waggener, Pulse Code Modulation Tehniques, Springer P., 1995.

[5] Y. Y. Song, »An Introduction to Formal Languages and Machine Computation,« World Scientific Publishing Co. , Singapore, 1998.

[6] A.-C. algorithm, »CP-Algorithms,« 2014. [Mrežno]. Available: https://cp-algorithms.com/string/aho_corasick.html. [Pokušaj pristupa 13 September 2020].

[7] M. B. Klaričić i S. Braić, »Uvod u matematiku,« Split, 2017.

[8] S. Krešić Jurič, Algebarske strukture - Skripta, Split, 2013.

[9] T. Toffoli i N. Margolus, Cellular Automata Machines: A New Environment for Modeling, MIT Press, 1987.

[10] D. Harabor i A. Grastien, »Online Graph Pruning for Pathfinder on Grid Maps«.

Skraćenice

IDLA* = incremental duplicate learning A*

DFS = depth-first search

BFS = breadth-first search

JPS = jump-point search

SSP = state space pruning